Object Membership

The Core Structure of Object Technology

An abstract structure is described that is believed to be central to object-oriented programming and modelling. In its main form, the structure is built from three relations between objects: $\in$, $\leq$ and $.ec$. The most fundamental relation, $\in$, is called *object membership* and is a refinement of the *instance-of* relation. The $\leq$ relation is the *inheritance* between objects. Finally, $.ec$ is a partial map that forms a distinguished subrelation of $\in$.

The structure arises as a generalization of the innermost core of the object model of the Ruby programming language. In Ruby, the $.ec$ map is total – every object has an *eigenclass*. The following equalities hold:

$$(\geq) = (.ec) \odot (\ni) \quad \text{and} \quad (\in) = (.ec) \odot (\leq).$$

The second equality says that object membership is the composition of infinite regress of eigenclasses with inheritance.

As an essential feature, the structure supports circular objects, i.e. objects x such that $x \in x$. This is also the main indicator of applicability. It is shown how the structure applies to class-based programming languages (Ruby, Python, Java, Scala, Smalltalk, Objective-C, CLOS, Perl and Dylan), prototypal languages (JavaScript), and ontology languages (RDF Schema, OWL Full).

A general mathematical structure of object membership is described in an affiliate document[46]. A gradual set-theoretic representation is provided with the exact correspondence of $\in$, $\leq$, $.ec$ and derived constituents of the structure to fundamental notions of set theory.

As a result, a uniform and mathematically precise view of an essential part of object technology is provided.

Author

Ondřej Pavlata
Jablonec nad Nisou
Czech Republic
atalon (at) atalon (dot) cz

Document date

Initial release	August 24, 2012
Last major release	March 12, 2015
Last update	February 16, 2016

HTML version

An HTML version of this document is available at http://www.atalon.cz/om/object-membership/.

Warning

This document has been created without any prepublication review except those made by the author himself.

Table of contents

Introduction

This document focuses on the core structure of *object technology*. We use the this term in accordance to the Journal of Object Technology [60] for a generalization of *object-oriented programming* (OOP) so that also other "OO"-terms are included. Basically, we are interested in *object models* [6]. The word "model" indicates that something is abstractly represented in a structured way. The word "object" indicates uniformity – there is a uniform unit of representation called *object*.

The approach

We can further combine the terms from the previous paragraph and say that an object model is itself based on an abstract structure between objects. The term *structure between objects* can be further interpreted as *family of relations between objects*. The term *abstract* suggests *abstraction*. For every non-trivial object model *M* there is a hierarchy of models that are abstractions of *M*. Around the top of the hierarchy there are *core structures* – they are formed by the most fundamental relations between objects.

This document identifies the following three relations as the core relations of object technology:

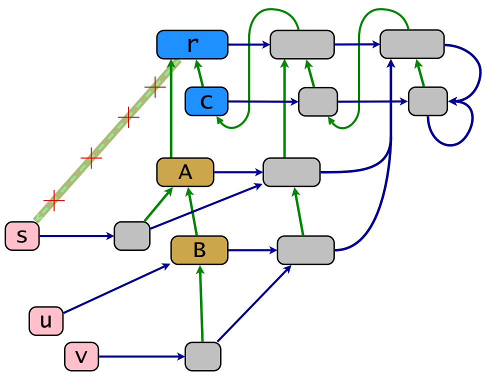
- ϵ, the *object membership* relation, is the most fundamental relation between objects. It is a generalization of the *instance-of* relation.
- $\leq$, the *inheritance* relation, is considered the second most important.
- *.ec*, the *powerclass map*, is an auxiliary, possibly empty, one-to-one subrelation of ϵ.

Since abstraction is one of the most fundamental principles of object technology, we can characterize our approach as

applying object technology to itsef.

Introductory sample

The diagram below shows an example of a core structure of object technology. Objects are displayed as nodes. The inheritance relation, $\leq$, is shown by green arrows in the reduction to immediate pairs. Object membership, ϵ, is the composition of blue arrows with inheritance. The powerclass map, *.ec*, is indicated by horizontal blue arrows.



(Ruby 1.9)

```
r = BasicObject
c = Class
A = c.new(r)
B = c.new(A)
s = A.new
u = B.new
v = B.new

class << s; end
class << v; end
```

The code on the right shows how the structure can be obtained in the Ruby programming language as a part of data structure of a running program.

Notes: (See Ruby core sample.)

1. c and r are only immediate in the partial structure, restricted to chosen objects. In the complete Ruby built-in structure, there are 2 objects between c and r.
2. In order to demonstrate the possible partiality of the *.ec* map, we deviate from the canonical interpretation supported further in the document. In Ruby, we consider *.ec* to be a total map.

The objective

The objective of this document is to rigorously describe structures that arise from ϵ, $\leq$ and $.ec$. That is, using the conventions in the diagram above, the document is preoccupied with the following question:

What can be said about the blue and green arrows?

For instance, it can be said that there cannot be an additional green arrow from r to s in the sample structure. Therefore, what the document essentially contains is an axiomatic description of families of structures that arise from the three core relations. Since the relations are fundamental, their description should contribute to foundations of object technology. Presumably, the above question can be stated as

What is the mathematical structure for the
- most fundamental part
- utmost simplification
- highest abstraction

of object technology?

Naturally, there is no universal description for all specific cases. We have to define idealizations on which the specific cases are based.

The $\epsilon^2 \neq \emptyset$ condition and circularity

Structures in which ϵ^2 is empty (that is, the composition of ϵ with itself is an empty relation) have a simple description which is provided in a single introductory section. The complementary condition $\epsilon^2 \neq \emptyset$ can be expressed as the *classes are objects* principle. (By convention, the phrase also indicates that the structure is not "classless".)

This document is predominantly concerned with structures satisfying $\epsilon^2 \neq \emptyset$. That is, $x \epsilon y \epsilon z$ for some (not necessarily distinct) objects x, y and z. It turns out that in all the introduced structures, the following stronger condition holds:

There exists an object x such that $x \epsilon x$.

This circularity condition is perhaps the best indicator for the issues handled in this document. Another good indicator is the presence of the notion of *metaclass*. Being a metaclass mostly implies being in the image of ϵ^2. Therefore, support for metaclasses is dependent on the $\epsilon^2 \neq \emptyset$ condition. In the affiliated document [51] the metaclass term is used as a label for our approach to object technology.

The non-objective

The focus being on the core structure of the object model, many notions of object-oriented programming occur outside the scope of this document. This should be in particular emphasized for the notion of ***type***. This document provides no explicit explanation as to what a type is. Maybe the document provides an implicit explanation but the author is unaware of it.

The concept of type is regarded as something more complex (an therefore less fundamental) than things described in this document. We use the term "type" exclusively as a mean of quotation (reference) of particular concepts by other parties (e.g. "types" as particular objects in Dylan, or "abstract power type system" for the abstract structure of Cardelli's power types).

As a consequence, this document becomes virtually disjoint with the following two books, both of which claim to provide foundations of object-oriented programing. (The "metaclass" term can be used as an indicator for the disjointness. There is no occurrence of this word in either of the books. Similarly for "meta-class", cf. [51].)

- ***A Theory of Objects*** by Martin Abadi and Luca Cardelli [1].

  Excerpt from the Preface: [1a] *This book develops a theory of objects as a foundation for object-oriented languages and programming. Our theory provides explanations for object-oriented notions in terms of a few basic primitives, and can be useful for the design and understanding of programming languages.*

- ***Foundations of Object-Oriented Programming Languages: Types and Semantics*** by Kim B. Bruce [8]

Excerpt from the publisher's overview: [8a] *This text explores the formal underpinnings of object-oriented languages to help the reader understand the fundamental concepts of these languages and the design decisions behind them.*

Composition of ϵ and $\leq$

It will be convenient for further reference to already introduce the following composition rules for ϵ and $\leq$:

Rule name	Short expression	Using variables	Set-theoretic counterpart	RDFS rule(s)
Transitivity of $\leq$	$(\leq) \odot (\leq) \subseteq (\leq)$	if $x \leq y \leq z$ then $x \leq z$	$(\subseteq) \odot (\subseteq) \subseteq (\subseteq)$	rdfs11 (and rdfs5)
Subsumption of ϵ	$(\epsilon) \odot (\leq) \subseteq (\epsilon)$	if $x \mathrel{\epsilon} y \leq z$ then $x \mathrel{\epsilon} z$	$(\in) \odot (\subseteq) \subseteq (\in)$	rdfs9
Monotonicity of ϵ	$(\leq) \odot (\epsilon) \subseteq (\epsilon)$	if $x \leq y \mathrel{\epsilon} z$ then $x \mathrel{\epsilon} z$		

These rules can be observed to be satisfied by the introductory sample. We even have used the first two conditions to reduce the set of displayed arrows: The set of green arrows is the transitive reduction of $\leq$ and the set R of blue arrows in the diagram is minimum such that $R \odot (\leq) = (\epsilon)$. We could even have had further reduced the set of displayed blue arrows by applying the mononicity rule. The resulting subrelation of ϵ would be the minimum relation S such that $(\epsilon) = (\leq) \odot S \odot (\leq)$. (In the case of the sample, $S = (.ec) \cup \{(u,\mathrm{B})\}$.)

The table also indicates that $\subseteq$ (set inclusion) and $\in$ (set membership) are the respective set-theoretic counterparts in of $\leq$ and ϵ. However, the latter correspondence is indirect – we will further introduce $\mathit{\epsilon}$, a restriction of ϵ, to be the direct correspondent of $\in$, see Set-theoretic interpretation.

In contrast to subsumption, the monotonicity rule has no set-theoretic counterpart. In set theory, it is not true that for every sets x, y, z,   if $x \subseteq y \in z$ then $x \in z$. (For example, if $x \subset y \in \{y\}$ then $x \notin \{y\}$.) Nevertheless, monotonicity of ϵ is satisfied in the major part of object technology. The "monotonicity" term comes from the following equivalent formulation: for every objects x, y,

$$x \leq y \rightarrow x.\epsilon \supseteq y.\epsilon$$

where $u.\epsilon$ is the set of all containers of u (the image of $\{u\}$ under ϵ). In particular, in *canonical primary structures*, the condition can be expressed as monotonicity of the $.class$ map, i.e. for every objects x, y,

$x \leq y \rightarrow x.class \leq y.class$.   (If x is a subclass of y then $x.class$ is a subclass of $y.class$.)

This is known as the *metaclass compatibility* condition [21].

Main results

This document builds a mathematical model that is presumably relevant to the core part of object technology. The following fundamental notions of computer science can be rigorously described in terms of the model:

- class
- metaclass
- eigenclass
- instance-of
- inheritance
- isa
- class-of
- eigenclass-of
- singleton-of

It is shown how the model applies to class-based programming languages (Ruby, Python, Java, Scala, Smalltalk, Objective-C, CLOS, Perl and Dylan), prototypal languages (JavaScript), and ontology languages (RDF Schema, OWL Full). The following characteristics can be observed:

- The Ruby programming language is the definitive sample language for the description of the object model. The Ruby object model (ROM) provides a clean and robust implementation of the core structure. In fact, the work on this document originated in discovering the exquisite properties of ROM.
- The Python programming language can be thought of as complementary to Ruby w.r.t. core structure. The most part of the core structure of object technology can be described by combining Ruby and Python.
- The Smalltalk-80 programming language is the biggest obstacle to understanding the above fundamental notions of object technology.
- The JavaScript programming language can be thought of as a class-based language with prototypes as inverse eigenclasses.

On the other hand, it is shown how the model can be embedded into the universe of sets. This results in a set-theoretical foundation of the core part of object technology.

Basic structure

$\overline{\in}$

∊, ≤, $\underline{r}$, .ec, .εȼ

$\in^{-1}$

$\in^{-2}$

$\in^{-3}$

⋮

The general mathematical model of "The Core" is provided by the family of *basic structures*. The diagram on the right shows the signature of these structures. In addition to the ∊, ≤ and .ec relations mentioned in the introduction there are additional constituents:

- $\overline{\in}$, the *power membership* relation, is an abstraction of the composition (.ec) ⊚ (≤).
- $\in^{-i}$, for every natural *i > 0*, are the *negative powers of membership*, which are abstractions of (≤) ⊚ .ec(-i), where .ec(-i) is the *i*-th relational composition of the inverse of .ec.
- .εȼ is the *primary singleton* map, an auxiliary subrelation of ∊ that is an abstraction of set membership between non-singleton sets and singleton sets.

Objects are abstractions of sets in a suitable partial universe of well-founded sets. The universe is limited by a rank – both from above and below. In particular, the empty set is not considered to belong to the universe.

- $\underline{r}$, the *inheritance root*, is a distinguished object, that is an abstraction of the largest set within the partial universe.

The axioms of basic structures are provided in a dedicated section. Moreover, there are two distinguished derived subrelations of ∊:

- .εc is the *singleton* map – an abstraction of set membership between (arbitrary) sets and singleton sets,
- ϵ is the *bounded membership* relation, which is an abstraction of (.εc) ⊚ (≤) and thus an abstraction of set membership within the partial universe.

Finally, the most fundamental three relations ∊, ≤ and .ec have the following semantics:

- ≤ is an abstraction of set inclusion.
- .ec is an abstraction of a relativized powerset operator. That is, if *x* is an object that is an abstraction of the set *a*, then *x.ec* is an abstraction of the set of all such subsets *b* of *a* such that both *b* and *{b}* belong to the partial universe. Since the empty set does not belong to the universe, .ec and .εc are coincident on singletons.
- ∊ equals (ϵ) ∪ ($\overline{\in}$).

Distinguished subfamilies

The following distinguished families of basic structures can be singled out:

- *Metaobject structures* are those in which .ec is total and .εc is defined for every object whose rank is not maximal. They can be presented in the signature ($\underline{O}$, ≤, $\underline{r}$, .ec, .εc).
- *Complete structures* are partial universes with urelements, devoid of the empty set. They are superstructures cumulatively built from the ground stage of memberless objects. They can be expressed as ($\underline{O}$, ϵ) – that is, ϵ is the only definitory constituent, just like ∈ in set theory.
- *Monotonic structures* arise from the (abstract) monotonicity condition (∊) = ($\overline{\in}$) which is present virtually in all object-oriented programming. Though not imposed in ontological languages such as RDF Schema, it can be assumed that most real datasets are subject to monotonicity. (In particular, the built-in RDFS structure is monotonic.)
  - In monotonic structures, the term "powerclass" is interchangeable with "eigenclass".
- *Monotonic eigenclass structures* are monotonic structures in which .ec is total. This family can serve as the "essential mathematical model" for the core structure of object technology. This is for the following reasons:
  - The monotonicity condition poses no restriction in most parts of object technology.
  - The .ec map can be completed for any basic structure.
  - The structures are axiomatized with 5 simple conditions which can be stated with only a few preliminary definitions.
  - The structures are fully determined by ∊: *x* ≤ *y* ↔ *x.*∊ ⊇ *y.*∊ and .ec is the unique map such that (∊) = (.ec) ⊚ (≤) (therefore, ∊ arises as the composition of infinite eigenclass regress with inheritance).
- *Canonical primary structures* are based on the Python object model. They describe the instance-inheritance structure that is thought to be canonical in the world of OOP.
- *Canonical eigenclass structures* are based on the Ruby object model. In this document, these structures are regarded as the standard core structures of OOP.

Up to a minor difference, there is a duality between the two families of canonical structures: Canonical primary structures are obtained from canonical eigenclass structures by omitting eigenclasses. Conversely, canonical eigenclass structures are obtained from canonical primary structures by eigenclass completion.

Connection between ϵ and $\leq$

The powerclass ($.ec$) and singleton ($.\varepsilon c$) maps provide a connection between membership (ϵ) and inheritance ($\leq$):

For every objects x, y,	in a **monotonic eigenclass structure**:	whereas in a **metaobject structure**:
	$x \in y.ec \leftrightarrow x \leq y$, $x \in y \leftrightarrow x.ec \leq y$,	$x \in y.ec \leftrightarrow x \leq y$, $x \in y \leftrightarrow x.\varepsilon c \leq y$ or $x.ec \leq y$.

Using the concise notation of relational composition,

$(.ec) \odot (\ni) = (\geq)$ and $(\epsilon) = (.ec) \odot (\leq)$ in a monotonic eigenclass structure,
$(.ec) \odot (\ni) = (\geq)$ and $(\epsilon) = ((.\varepsilon c) \cup (.ec)) \odot (\leq)$ in a metaobject structure.

In monotonic structures, the singleton map $.\varepsilon c$ is a submap of $.ec$. The singletons are exactly the powerclasses from powerclass chains that start in *terminal* objects – which are objects that do not have any members or strict inheritance ancestors. (In the sample structure, s, u and v are terminals and $s.ec = s.\varepsilon c$ and $v.ec = v.\varepsilon c$ are singletons.)

Set-theoretic interpretation

Set-theoretic interpretation of basic structures is summarized by the table below. Any basic structure $\mathcal{S}$ can be identified with a set $\underline{O}$ in the von Neumann universe of well-founded sets. Let $\mathbb{P}$ denote the standard *powerset* operator, that is, for a set x, $\mathbb{P}(x)$ is the set of all subsets of x. Moreover, let $\mathbb{P}_1(x)$ be the set of singleton subsets of x. The constituents of $\mathcal{S}$ are then obtained as follows:

Terminal objects	$\underline{T}$	$=$	$\underline{O} \cap \mathbb{P}_1(\bigcup \underline{O} \setminus \underline{O})$
Inheritance root	$\underline{r}$	$=$	$\bigcup \underline{O} \setminus \bigcup \underline{T}$
For every x, y from $\underline{O}$:			
Bounded membership	$x \epsilon y$	$\leftrightarrow$	$x \in y$
Inheritance	$x \leq y$	$\leftrightarrow$	$x \subseteq y$
Singleton map	$x.\varepsilon c = y$	$\leftrightarrow$	$\{x\} = y$
Powerclass map	$x.ec = y$	$\leftrightarrow$	$\underline{r} \cap \mathbb{P}(x) = y$
Power membership	$x \overline{\epsilon} y$	$\leftrightarrow$	$\underline{r} \cap \mathbb{P}(x) \subseteq y$
Object membership	$x \epsilon y$	$\leftrightarrow$	$\underline{r} \cap \mathbb{P}(x) \subseteq y$ or $x \in y$

Simple cases of ϵ

To proceed further with the description of ϵ, $\leq$ and $.ec$ (even before giving an account on history of implementation and documentation of these structures) we first do away with simple cases:

I. $\epsilon = \varnothing$ $(*)$ (purely prototypal structures),
II. $\epsilon \neq \varnothing$ and $\epsilon^2 = \varnothing$ (bipartite structures).

For convenience, we also present the condition for the remaining (i.e. non-simple) cases:

III. $\epsilon^2 \neq \varnothing$ (basic structures of ϵ and their specializations).

Note: $(*)$ This case relates to prototype-based programming languages and is more carefully handled in [51]. An alternative interpretation is proposed, in which ϵ is equal to $\leq$.

Purely prototypal structures

Purely prototypal object models are those which deny to institutionalize any concept of a classification. In accordance, a *pp-core structure* is a structure $(O, \le)$ such that

(pp~1) $\le$ is a partial order on O.

By not presenting $\in$ and $.ec$ in the signature it is implicitly understood that these relations are both empty.

We do not consider all prototype-based programming languages to have a purely prototypal core structure. In particular, in JavaScript [17], $\in \neq \emptyset$. We base this judgement on the following observations:

- The expression *x* `instanceof` *y* can evaluate to `true` for some *x* and *y*.
- The notion of a *class* is supported in authoritative books about JavaScript [19].

Bipartite structures

Bipartite structures represent a two-sorted view of $\in$ which is characteristic for static class-based languages like C++ or Eiffel.

By a *bipartite primary structure* we mean a structure $(O, \in, \le)$ where

- O is a set,
- $\in$ is the *instance-of* relation on O,
- $\le$ is the *inheritance* relation on O.

Let us denote $T = O.\ni$, the pre-image of O under $\in$, and call elements of T *instances*. Elements of the complementary set $C = O \setminus T$ are *classes*. The structure is subject to the following conditions.

(bp~1) Inheritance, $\le$, is a partial order. In particular, its reflexive reduction $<$ is acyclic.

(bp~2) (a) $(<) \circ (\in) = \emptyset$. (b) $(\in) \circ (\in) = \emptyset$.
That is, instances have (a) no strict descendants and (b) no instances.

(bp~3) $(\in) \circ (\le) \subseteq (\in)$. (The subsumption rule.)
That is, every class has ("inherits") all instances of its ancestors.

(bp~4) Every instance x has a least container, $x.class$.
As a consequence, there is a unique map $.class$ from T to C such that $(\in) = (.class) \circ (\le)$.
(Observe that this equality implies the subsumption rule.)

Similarly to the previous case, it is assumed that $.ec$ is empty. (This is also indicated by the adjective *primary* since in theory, there can be bipartite structures that support $.ec$ on T. However no such occurrence in object technology is known to the author.) We might consider the additional condition of a single-rooted inheritance (which is not satisfied in C++). On the other hand, (bp~4) is not to be imposed on ontology languages.

Note that we did not mention the word "object" anywhere in the definition. This is because we have disobeyed the two-sortedness – we unified the sorts T and C into one set O. In the two-sorted view of $\in$ the term "object" is exclusively reserved for elements of T. This is expressed by:

- *Classes are not objects*. (See e.g. [6], page 93.)

This is in opposition to our approach. We consider the core structure to be single-sorted – there is a uniform domain O of objects on which the relations $\in$, $\le$ and $.ec$ are defined. In particular,

- *Classes are objects*.

The following table shows a division of significant representants of object technology according to the composability of $\in$. Note that there is a middle group which takes a neutral approach:

- For every class x, the *reificiation* of x ($x.reif$) is an object.

The point of it is that "reification" can be overridden by "identity". This approach seems to be supported in particular by the Java programming language. In Java, the names of introspection methods like `getClass()` or `getSuperclass()` suggest that it is a class what is returned, rather than its reification. In the book *Java reflection in action* [22], both agreement and disagreement to the uniformity principle can be observed. (According to page 23, "*classes are objects*". According to page 268, there is a "*distinction between class and class object*".)

Group	Composability of ∊	Programming Languages		Ontology Languages
(A)	$\varnothing = (∊) \circ (∊)$	C++, Eiffel	PHP	OWL 2 DL
(B)	$\varnothing \neq (∊) \circ (.reif) \circ (∊)$	Java, Scala, C#		—
(C)	$\varnothing \neq (∊) \circ (∊)$	Ruby, Python, Smalltalk, Objective-C, CLOS, JavaScript		RDF Schema, OWL 2 Full

Overview of atalon.cz and related documents

There are several documents at atalon.cz that are concerned with the core structure of object technology.

- Object Membership – *The Core Structure of Object Technology* (this document).

  This is the main document that provides an introduction to the subject as well as synthesis of other documents.

- Object Membership – Basic Structure. [46]

  The document describes the ultimate mathematical model of "The Core" and its connection to set theory. In particular, it is shown that object membership, ∊, arises by the following fundamental equality:

  $$(∊) = (\epsilon) \cup (\overline{∊})$$

  where ϵ is the "bounded" part, a well-founded relation, and $\overline{∊}$ is the "monotonic" part. The main correspondence between (the core structure of) object technology and set theory can be then expressed as

  $$\epsilon \quad \leftrightarrow \quad \in.$$

  If objects have sufficiently many bounded members and $.ec$ is total, then all of ∊, $\leq$ and $.ec$ are determined by ϵ so that the core structure can be expressed as $(\underline{O}, \epsilon)$. In a complete structure, every non-empty subset of $\underline{O}.\ni$ equals the extension $x.\ni$ of a unique object x. Such a structure looks like a partial universe of sets, given by the pair $(\underline{\varpi}+1, \kappa)$, where $\underline{\varpi}$ is a limit ordinal that is the rank of the inheritance root $\underline{r}$, and κ is the cardinality of the ground stage of urelement-like sets. Every basic structure can be embedded into such a partial universe.

- Object Membership: *Simplified Structure*. [47]

  The assumption of monotonicity ($(∊) = (\overline{∊})$) and of totality of $.ec$ results in the family *monotonic eigenclass structures*. This family has a simple description (simpler than the general case of basic structures) which is provided in the referred document. The restrictions can be considered acceptable:

  - The monotonicity condition holds for the majority of object-oriented programming and modelling.
  - Every basic structure has a powerclass completion.

  A complete monotonic structure $(\underline{O}, \epsilon)$ is obtained from a complete basic structure $(\underline{V}, \epsilon)$ by the restriction to *hereditarily* ↧*-complete* objects, i.e. $\underline{O} = \{ x \mid x.\ni = x.\ni.↧ \subseteq \underline{O} \}$.

- Object Membership: *The core structure of object-oriented programming*. [44]

  Historically, this document introduced the term *object membership* (together with the ∊ symbol) and originally was the main document. The (canonical) core structure of OOP arises by unifying core structures of Ruby and Python. It can be viewed as either the Ruby core structure with multiple inheritance and user-created explicit metaclasses allowed, or the Python core structure equipped with eigenclasses.

  The core structures of Java, Scala, Smalltalk, Objective-C, CLOS or Perl are viewed as modifications of the canonical structure. (See also Specializations of ∊.) Additionally, refinement structures are described which provide linearization of object's ancestors.

- Object Membership: *The ontological structure*. [45]

  The document provides a generalization of object membership as it applies to ontological structures based on RDF Schema. In contrast to object-oriented programming the following features are present:

  - An object x can have multiple minimum classes of which x is an instance.
  - There is a distinguished "sort" of objects, called *properties*. Like terminals, properties are not descendants of the inheritance root. Unlike terminals, properties can have descendants.
  - Inheritance does not have to be antisymmetric – classes and properties can have distinct equivalents.

  A summary is provided in the Ontological structure of ∊ section.

- Object Membership with Prototypes. [49]

  As already observed on the example of the JavaScript programming language, prototype-based languages cannot be, in general, declared to be "classless" or to be devoid of the instance-of relation (despite the traditional view [68a]). Moreover, it can be observed that in the JavaScript core structure $(\underline{O}, \epsilon, \leq, \underline{r}, .ec)$, not only ϵ but also the $.ec$ constituent is non-empty. The following correspondence holds:

$x.ec = y \ \leftrightarrow\ x$ `==` y`.prototype`.

That is, prototypes can be thought of as eigenclass predecessors of classes. A summary of the referred document is provided in the Prototypes section.

- Object Membership and Powertypes. [50]

  The document provides an explanation of the notion of a powertype in terms of the core structure. In particular, it is shown that, in a suitable axiomatic restriction, the Cardelli's $Power()$ operator coincides with $.ec$. That is, Cardelli's power types are powerclasses in the abstract setting. The only difference is that $Power(x)$ is only defined for `Class`es (called "types").

  Subsequently, the adoption of powertypes in metamodelling is considered, as well as powertype relationship in RDF Schema.

- The Dialectic of Classes and Metaclasses in Smalltalk-80. [48]

  Historically, the Smalltalk programming language is of fundamental importance to the core structure of object technology:

  - Smalltalk-76 is the first language in which $\epsilon^2 \neq \emptyset$,
  - Smalltalk-80 is the first language in which $.ec \neq \emptyset$.

  Unfortunately, the $.ec \neq \emptyset$ condition has not been reflected in the terminology or documentation of Smalltalk-80. As a consequence, inconsistencies have been established and so Smalltalk-80 became the biggest obstacle for the description of the core structure of OOP, at least for the author of this document. The referred document collects evidence about the inconsistencies. A consistent resolution of the terminology is provided in a special section.

- What Is a Metaclass?. [51]

  The document provides a more elaborate alternative to the present document. The title question is used as a linguistic device for the exploration of the object model core in various contexts. There are about 20 investigated environments. The *metaclass* term is used as a label for an approach to OOP foundations. The "metaclass approach" is based on four rules in the spirit of Occam's razor: (1) dispense with types, (2) dispense with calculus, (3) support object identity, (4) assume metaclass pre-condition ("classes are objects").

- Featherweight Java Axiomatically. [52]

  Featherweight Java is the most popular formalization of the most popular object-oriented programming language. In the referenced document, the formalization is adjusted to provide a clear definition of the core structure of the underlying data model.

- The Ruby Object Model: *Data Structure in Detail*. [40]

  The document provides a detailed description of the Ruby Object Model (as of version 1.9) via abstraction refinement. The object model is incrementally specified in the series of abstract structures together with their possible transitions. The most abstract, initial structure, denoted **S0**, just introduces the basic nomenclature of objects: *terminals*, *classes* and *eigenclasses*. The next structure, **S1**, is the core structure of Ruby in the sense introduced at the beginning of this document. The structure is (exactly what is) induced by *superclass* and *eigenclass* links between objects. The yet subsequent structure, **S2**, equips **S1** with module inclusion lists so that the complete inheritance structure is established (referred to by MRO, which stands for method resolution order). There is an in-between structure, denoted $\mathbf{S1}_r$ in [44], which encompasses just the extension of the Ruby's (canonical) object membership, ϵ, to the "full" membership, $\boldsymbol{\epsilon}$, which can be introspected by the `is_a?` built-in method (see also Specializations of ϵ).

  There are more than 20 structures in the gradual description. The resulting structure (still providing just an abstraction of the Ruby object model, though far more detailed than **S1**) can be viewed as a naming multidigraph, consisting of nodes and uniquely labelled arrows between them.

- Ruby Object Model – S1 superstructure representation. [41]

  The document provides a set-theoretic representation of the core structure of Ruby (again referred to as the **S1** structure) using the *bottom stratum* map $.\varrho$ between objects in an $(\omega+1)$-superstructure $(\underline{V}, \in)$. In contrast to the embedding presented in [46], the embedding from [41] maps all objects to elements of *finite* stages. As a result, for objects x, y whose metalevel index is lower than a predefined number k,

  - $x \leq y \ \leftrightarrow x.\varrho \subseteq y$ and $x \epsilon y \ \leftrightarrow x.\varrho \in y$.

- Ruby Object Model – The S1 structure. [42]
  The document focuses on the description of the **S1** structure using the material from the previous two documents. In all these three documents, object membership is expressed in the composition *(.ec)* ⊙ *(≤)*, the ϵ symbol is missing. Moreover, ≤ is considered to be defined only between non-terminal objects.
- Ruby Object Model: *Comparison with Smalltalk-80*. [43]
  Historically, this document is the first description (available at atalon.cz) of the Smalltalk-80 correspondent of the Ruby's **S1** structure. The document can be regarded as a precursor to the dedicated section.

Eigenclass model (Wikipedia article)

The archived Wikipedia article titled Eigenclass model [68b] provides a synopsis of monotonic object membership. The family of canonical eigenclass structures is introduced in the minimum signature (O, ϵ). The general family of monotonic eigenclass structures is presented under the name "*essential structure of* ϵ".

The Wikipedia page en.wikipedia.org/wiki/Eigenclass_model containing the article has been created in October 2012 by the author of this document. Several major edits followed until January 2013. In October 2013, the page has been deleted (changed to a redirect to Ruby (programming language)#Semantics) by Wikipedia administrators on the following grounds:

- The article violates the NOR principle (No original research).
- The term *eigenclass model* is not an established topic in computer science.

For those interested in the article there is a short list of corrections which I would apply if the page was not deleted:

c(i)	replace by	*c.ec(i)*
"other that"		"other than"
(p~4)	extend with "and instances of each other".	

History

The following table outlines the history of object models whose core structures satisfy $\epsilon^2 \neq \emptyset$. A more detailed treatment is provided in [51] which focuses on the *metaclass* term.

Year	Author(s)	Introduced (or applied) concept	Language(s)
1976	Daniel Ingalls [24]	Classes are objects ($\epsilon^2 \neq \emptyset$)	Smalltalk-76
1979	H. Levesque, J. Mylopoulos	Metaclasses in knowledge representation [30]	
1980	James Althoff [2]	Implicit metaclasses (*.ec* defined for classes)	Smalltalk-80
1983	D. Bobrow, M. Stefik [5]	Explicit metaclasses	LOOPS
1988	D. Bobrow, G. Kiczales [4]	Explicit metaclasses with a monotonicity check	CLOS
1988	Luca Cardelli [11]	Power types	Smalltalk-80
1992	Apple Computers, Inc.	Universal singletons	Dylan
1994	James Odell [37]	Power types in metamodelling	UML
1995	James Gosling	The `Class` class in mainstream programming	Java
1995	**Yukihiro Matsumoto**	Hidden eigenclasses, not referenceable	**Ruby**
1995	Keith Playford [54]	Combining powerclasses with singletons	Dylan
1995	Brendan Eich	Prototypes as inverse eigenclasses	JavaScript
1998	I. Forman and S. Danforth	A book about metaclasses	Java, Smalltalk, CLOS, Dylan
2001	Guido van Rossum	Fully monotonic explicit metaclasses	Python 2.2
2002	World Wide Web Consortium	Classes are resources ($\epsilon^2 \neq \emptyset$ in semantic web)	RDF Schema, OWL 2 Full
2004	Martin Odersky	Metalevel-1-only eigenclasses/singletons	Scala
		Universal eigenclasses, fully monotonic	

2008	**Yukihiro Matsumoto et al.**	($.ec$ defined for all objects, lazily evaluated)	**Ruby 1.9.1**
2012	Ondřej Pavlata (∗)	Formalization of monotonic ϵ with a total $.ec$	(most of the languages above and some others)
2015		Formalization of basic structure of ϵ	

(∗) *Show me a list of documents (articles, books), possibly hundreds of items long, that are not listed in the previous section. If, in total, these documents say about the abstract structure of ϵ, $\leq$ and $.ec$ more than **one tenth** of what documents inside of atalon.cz say, then I remove my name from the table.*

O.P.

Notes:

1. Bold font indicates that the introduction of Ruby is regarded in this document to be the most significant contribution to the development of the core structure of object technology. We can guess that the Ruby core structure has been created by rectification of Smalltalk-80 adopting the concept of universality of singletons in Dylan. This is best illustrated by samples from Ruby, Smalltalk-80 and Dylan.

The core structure by Forman & Danforth

In 1998, a book by Ira R. Forman and Scott Danforth has been published, titled *Putting Metaclasses to Work* [21]. The book provides an object model of class-based reflective programming languages using the concept of a metaclass. With some effort, a reduction of this model can be made to the core structure, which concerns just the ϵ and $\leq$ relations. A detailed description is provided in [51].

Preliminaries

Some familiarity with elementary algebra, order theory and set theory is assumed. See [46] for details.

Well-foundedness

For a relation ϵ on a set X, an element $x \in X$ is *well-founded in* ϵ if x is not a member of an infinite descending chain in ϵ, i.e. if there is no infinite chain of the form

$\dots x_2 \epsilon x_1 \epsilon x_0 = x.$

A relation ϵ on a set X is *well-founded* if all elements $x \in X$ are well-founded in ϵ. Assuming the axiom of choice, this is equivalent to the condition that every non-empty subset $Y \subseteq X$ contains an element y that is minimal in (Y,ϵ), i.e. there is no u from Y such that $u \epsilon y$.

Rank

For a well-founded relation ϵ on a set X, the *rank* function of ϵ (alternatively, the ϵ-rank) is a map $r()$ from X to ordinal numbers such that for every $x \in X$,

- $r(x) = \sup \{r(a) + 1 \mid a \epsilon x\}$.

By well-founded recursion, there is exactly one such map. Obviously, $r(x) = 0 \leftrightarrow x$ is minimal in ϵ. Moreover,

- let the ϵ-rank of a subset Y of X be $\sup \{r(a) + 1 \mid a \in Y\}$,
- let the rank of ϵ be the ϵ-rank of X.

The *instance-of* relation

In class-based programming languages, the *instance-of* relation is a fundamental mean to express similarity between objects. The relation links objects to *classes*. Objects that are instances of a given class share behavior (and structure) described by that class.

Let us write $x \epsilon y$ for "the object x is an instance of the class y". By using the ϵ symbol (lunate epsilon) for the instance-of relation we suggest the correspondence:

$$\epsilon \quad \leftrightsquigarrow \quad \in,$$

i.e ϵ has the semantics of set membership, $\in$. The class y represents the set of all its instances. Just like sets can share some elements, we allow classes to share instances. That is, we consider ϵ in the weak sense where an object can belong to several classes. We might also speak about *has-a-class* as a synonym for *instance-of*.

There is a second point that should be noted in the expression $x \mathrel{\epsilon} y$: both the object x and the class y are denoted by a lowercase letter. This suggests that we are focused on programming languages that support the following uniformity principle of ϵ:

Classes are objects.

As a consequence, ϵ is a relation between objects. If $x \mathrel{\epsilon} y$ then the class y is a meta-object of x in the sense that y is one of objects that provide description for x. If $x \mathrel{\epsilon} y$ and x is itself a class (and so is every other potential instance of y) then y is a *metaclass*. In fact, the occurrence of the notion of a "metaclass" in publications about a particular programming language is a good indicator that the language supports handling of classes as objects.

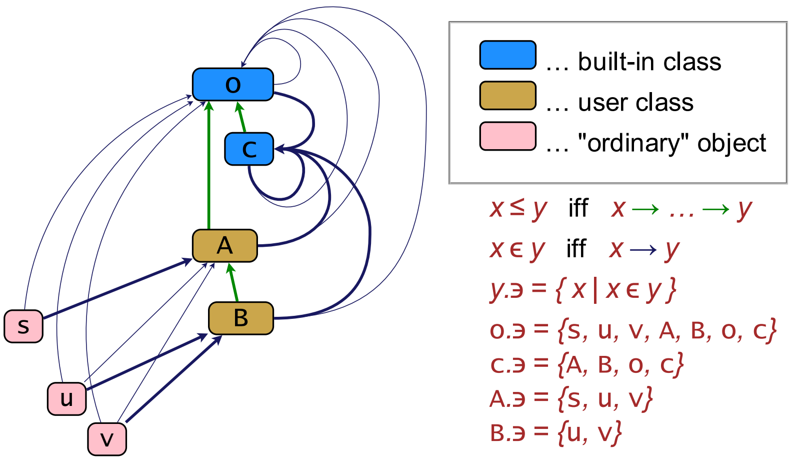
Sample structure

The following diagram shows a sample structure of the instance-of relation. The structure contains 7 objects: 4 classes and 3 "ordinary" objects. The instance-of relation is displayed by dark blue arrows: $x \mathrel{\epsilon} y$ iff there is a blue arrow from x to y.

For an object x we denote $x.\ni$ the set of all instances of x and call it the *extension* of x. Ordinary objects have empty extension. The sample structure is "saturated" in the sense that classes are distinguishable by their instances: different classes have different extensions. This allows to derive a partial order, denoted $\leq$, and called *inheritance*, between the 7 objects by:

$$x \leq y \quad \text{iff} \quad x = y \text{ or } \emptyset \neq x.\ni \subset y.\ni.$$

If $x \leq y$ then x is said to be an *(inheritance) descendant* of y which is in turn an *(inheritance) ancestor* of x. We also let $<$ denote the reflexive reduction (the *strict inheritance*) of $\leq$ in the obvious sense. The inheritance relation is shown as a Hasse diagram in the transitive reduction of $<$ by green arrows between classes. Ordinary objects are not involved in $<$. The sample structure only has *single inheritance* — every object has at most one inheritance *parent*.



$x \leq y \quad \text{iff} \quad x \to \ldots \to y$

$x \mathrel{\epsilon} y \quad \text{iff} \quad x \to y$

$y.\ni = \{\, x \mid x \mathrel{\epsilon} y \,\}$

$\text{o}.\ni = \{\text{s, u, v, A, B, o, c}\}$

$\text{c}.\ni = \{\text{A, B, o, c}\}$

$\text{A}.\ni = \{\text{s, u, v}\}$

$\text{B}.\ni = \{\text{u, v}\}$

Note that every object has a class that is least in inheritance. For an object x let us denote $x.class$ such a class and call it *the class* of x. If $x.class = y$ then we also say that x is a *direct* instance of y. Thus, the $.class$ map forms a subset of ϵ, distinguished by thick blue arrows in the diagram. The instance-of relation can be recovered from the $.class$ map and inheritance by their composition, i.e. $(\epsilon) = (.class) \odot (\leq)$ where the composition symbol "$\odot$" is interpreted left-to-right. The equality can be read as: x is an instance of y iff the class of x is an inheritance descendant of y.

Observe that classes o and c are circular – they are instances of itself. Moreover, every object is an instance of o and classes are exactly the instances of c.

Structure creation

In a programming language, `o` and `c` are distinguished *built-in* classes, usually named `Object` and `Class`, respectively. The remaining part of the structure is built incrementally, by adding "user" classes `A` and `B` and their instances `s`, `u` and `v`, one by one. Each of the addition of these five user objects x can be formally expressed as class *instantiation*:

$$x = q.\mathtt{new}(p_1, \ldots, p_n)$$

where q is the requested class of x and $p_1, \ldots p_n$ are the requested *parents* (direct inheritance ancestors) of x. For ordinary objects, the assignment reduces to $x = q$`.new()` which we further abbreviate to $x = q$`.new`. Since the sample structure only has single inheritance, new class creation can be expressed by $x =$ `c.new(`p`)` with the single requested parent p. As a result, the structure is built by the following five assignments:

```
A = c.new(o); B = c.new(A); s = A.new; u = B.new; v = B.new
```

This line of pseudocode (with "`;`" used as a delimiter between the assignments), if preceded by `o = Object` and `c = Class`, is a valid line of code in the Ruby programming language for the creation of the sample structure.

Adding inheritance to the signature

Let us denote $O_0 = \{o,c\}$, $O_1 = O_0 \cup \{A\}$, …, $O_5 = O_4 \cup \{v\} = \bar{O}$ so that O_0 is the set of built-in objects of the sample structure and for each $i = 1, \ldots, 5$, the set O_i corresponds to an addition of a single object into the structure. Note that most of the intermediate structures $(O_i, \in)$ do not satisfy the extensionality principle that allowed us to distinguish classes as objects with non-empty extension and to derive inheritance via inclusion of class extensions. To capture also the intermediate structures, the inheritance relation has to be prescribed explicitly. Therefore, we extend the signature from $(\bar{O}, \in)$ to $(\bar{O}, \in, \leq)$. We call this instance-inheritance structure the *primary structure* of $\in$.

Note that since $(\in) = (.class) \circledcirc (\leq)$ we can use the signature $(\bar{O}, .class, \leq)$ as well.

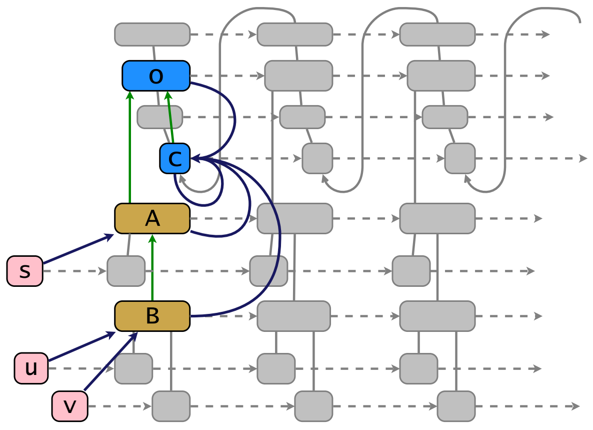
The sample expressed in relevant languages

This section demonstrates the presence of the instance-of relation in object oriented programming and modelling. It is shown how the sample structure can be created and introspected in various prominent programming languages. An example of an ontological language is provided too.

Where applicable, a diagram is provided showing how the sample structure is obtained as a restriction of a finer structure.

In Ruby

The Ruby programming language allows a uniform expression of the sample structure. Every (user) object can be created via the `new` method. The instance-of relation between the 7 objects can be detected via the `is_a?` method (aliased as `kind_of?`). In the restriction to classes, the `<=` method corresponds to $\leq$. The *.class* map is provided by the `class` introspection method.



```
o = Object
c = Class
A = c.new      # class A; end
B = c.new(A)   # class B < A; end
s = A.new
u = B.new; v = B.new
```

In Python

```
o = object
c = type
A = c('A', (), dict())    # class A(): pass
B = c('B', (A,), dict())  # class B(A): pass
s = A()
u = B(); v = B()
```

In Python, new instances are created by "calling" classes. Also the classes A and B can be created this way, though not so transparently as in Ruby. The instance-of relation between *x* and *y* is detected by `isinstance(`*x*`,`*y*`)`. Inheritance between classes *x*, *y* is detected by: $x < y$ iff `issubclass(`*x*`,`*y*`)`. For every object *x*, the expression `type(`*x*`)` returns the class of *x*.

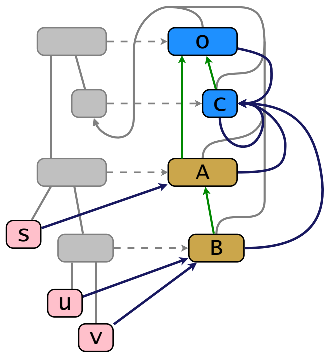
In JavaScript

Traditionally, *prototype-based* programming languages are regarded to be "classless" and thus devoid of the instance-of relation. However, as we have already mentioned before, we do not consider this characteristics to be applicable to JavaScript. The diagram below demonstrates that JavaScript allows for the creation of our sample structure. (So that all of o, c, A and B are examples of JavaScript classes.) Every (user) object can be created via the new operator. The instance-of relation between the 7 objects can be detected via the `instanceof` operator. The *.class* map is provided by the `constructor` property. Inheritance between classes can be detected via the `isPrototypeOf` method as follows:

$x < y$ iff *y*`.prototype.isPrototypeOf(`*x*`.prototype)`.

Observe that in contrast to the Ruby sample, the additional objects of the finer structure are drawn on the opposite (i.e. left) side. This is because Ruby supports implicit containers (eigenclasses), whereas JavaScript supports implicit members (prototypes).

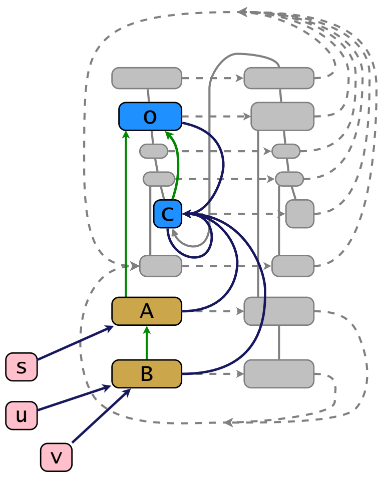
Note: To simplify the code, inheritance between B and A is achieved using the non-standard property `__proto__`.



```
o = Object; c = Function
A = new c
B = new c; B.prototype.__proto__ = A.prototype
s = new A
u = new B; v = new B
```

In Smalltalk

Smalltalk-80 (as of Pharo or Squeak) only allows creation of classes via the `subclass:` method. The instance-of relation between the 7 objects corresponds to the `isKindOf:` introspection method. The introspection method named `class` only partially corresponds to the *.class* map: while it agrees on ordinary objects, it disagrees on classes.



```
o := Object. c := Class.
o subclass: #A.
A subclass: #B.
s := A new.
u := B new. v := B new.
```

In Java

In the Java programming language, classes can be regarded as objects due to the *reification* facility. For a class named `A`, the "class object" of `A` is referred to by `A.class`. Note that in contrast to Ruby, `A.class` does not refer to the class of the class named `A`. As a consequence, there is a notational duality between (non-reified) classes and their object counterparts. The instance-of relation between objects *x* and *y* where *y* is known to be a reified class named `Y` can be detected either via

- *x* `instanceof Y`, or
- *y*`.isInstance(`*x*`)`.

The (reified) class of an object *x* equals *x*`.getClass()`. If *x* is a (reified) class other than `o`, then *x*`.getSuperclass()` is the (reified) inheritance parent of *x*. Note that the method names suggest that it is a class what is returned, rather than a "class object".

```
class A {}
class B extends A {}

class Xx {
  public static void main (String[] xx) {
    Object
    o = Object.class,
    c = Class.class,
    A = A.class,
    B = B.class,
    s = new A(),
    u = new B(), v = new B();
  }
}
```

In Scala

Scala adopts the Java's concept of reification of classes. For a class named `A`, the expression `classOf[A]` refers to the "class object". Note again that `classOf[A]` is not the class of `A`. Introspection facilities for the instance-of relation and inheritance between classes are similar to those of Java.

```
var o = classOf[Object]
var c = classOf[Class[_]]
class A
class B extends A
var s = new A
var u = new B; var v = new B
```

In Dylan

The Dylan programming language fully supports the "classes are objects" principle [20]. Every user object can be created via the `make` method. Inheritance between classes is detected via `subtype?`, the instance-of relation between objects and classes is detected via `instance?`. Finally, the `object-class` method is the exact correspondent to the *.class* map.

```
let o = <object>;
let c = <class>;
let A = make(c);
let B = make(c, superclasses: list(A));
let s = make(A);
let u = make(B); let v = make(B);
```

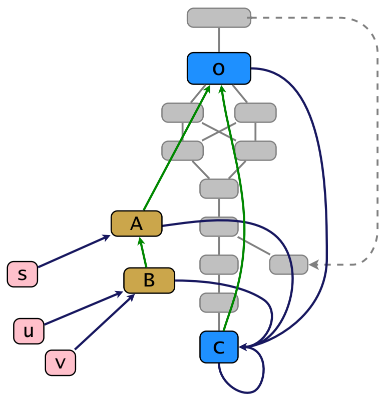
In CLOS

According to authoritative publications [58] [35], the Common Lisp Object System (CLOS) fully conforms to the "classes are objects" principle. (However, there is still some sort of notational distinction for classes.) The instance-of and inheritance relations between objects can be detected via the built-in `typep` and `subtypep` methods, respectively.

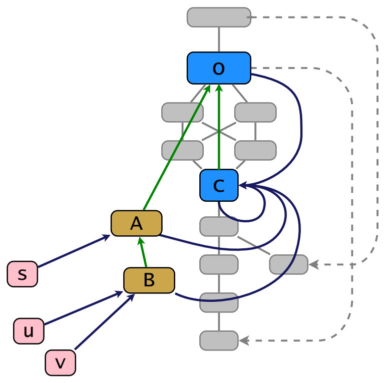
```
(defvar o (find-class 'standard-object))
(defvar c (find-class 'standard-class))
(defvar A (make-instance c :name 'A))
(defvar B (make-instance c :name 'B :direct-superclasses (list A)))
(defvar s (make-instance A))
(defvar u (make-instance B)) (defvar v (make-instance B))
```

Gray color in the code indicates that we consider two possible interpretations of `c`, shown in the diagrams below. In the (a) case, the class map between the 7 objects is provided by the `class-of` method.

(a)
`c` corresponds to `standard-class`



(b)
`c` corresponds to `class`



In RDFS

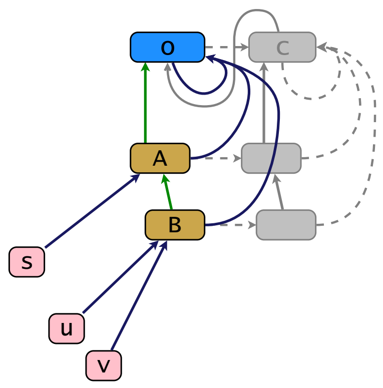
In RDF Schema,[63] [64] objects are called *resources*. Our sample structure is obtained from the RDF graph entailed by the following set of RDF triples expressed in Turtle syntax [67]. The "example" prefix `ex:` is used for the user-created objects.

```
ex:B rdfs:subClassOf ex:A .
ex:s rdf:type ex:A .
ex:u rdf:type ex:B .
ex:v rdf:type ex:B .
```

The sample built-in classes o and c are those named rdfs:Resource and rdfs:Class, respectively. The rdf:type property stands for the instance-of relation, ϵ. Inheritance between classes is expressed via the rdfs:subClassOf property.

In Objective-C

The diagram below shows how the sample structure can be interpreted in Objective-C. The o and c classes are coincident, so that the sample structure contains one object less. The diagram also indicates that there is a "non-degenerate" interpretation, with c set to the gray object labelled by c. In this document, we prefer the first interpretation, since we do not consider the gray object to be a class.



```
@interface A: NSObject; @end
@interface B: A;        @end
@implementation A; @end
@implementation B; @end
int
main(int argc, const char *argv[]) {
  id
  o = [NSObject class],
  s = [A new],
  u = [B new], v = [B new];
  return 0;
}
```

The instance-of relation can be detected via the isKindOfClass: method. The class_getSuperclass method can be used for the introspection of inheritance between classes. Similarly to Smalltalk-80, the class introspection method only agrees with the *.class* map on ordinary objects. For a class *x*, the expression [*x* class] evaluates to *x*. (This is in contrast to Smalltalk-80, where *x* class is always different from *x*.)

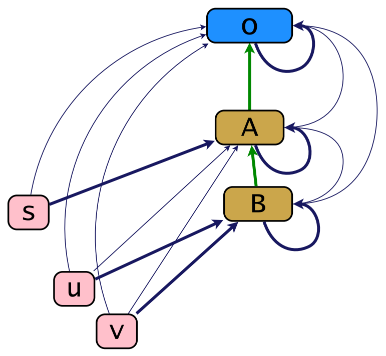
In Perl

In Perl 5, the instance-of relation, ϵ, deviates from the standard (represented by the sample structure) in a substantial way. In the restriction to classes, ϵ coincides with inheritance, ≤. As a consequence, there is a total circularity between classes:

- every class is an instance of itself, and even
- every class is the class of itself.

The instance-of relation can be detected by the isa introspection method. The semantics of ϵ is then adhered to by method lookup. [15]

The diagram below shows the sample structure in the correspondent modification.



```
package UNIVERSAL { sub new { bless {}, $_[0] } }
package A         { }
package B         { our @ISA = A }
my
$o = UNIVERSAL,
$s = A->new,
$u = B->new, $v = B->new;
```

Refinement by implicit containers

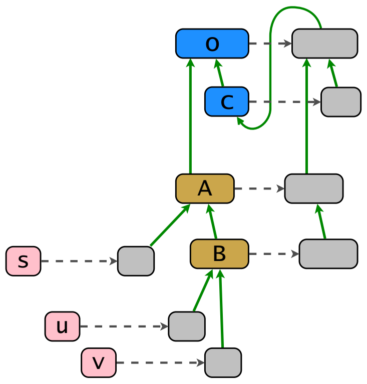
In the primary structure $(\bar{O}, \in, \leq)$ introduced above, every non-built-in object is explicitly created by class instantiation. (As shown by the Ruby or Python code, classes themselves are created by instantiation of c.) As a consequence, every class is the most specific descriptor for its direct instances. Equivalently, objects instantiated by the same class have the same description. In particular, every class has the same set {c, o} of "describing" classes in $(\bar{O}, \in, \leq)$.

Eigenclasses

The instance-of relation can be refined by adding implicit containers. In the following diagram, the sample structure $(\bar{O}, \in, \leq)$ is extended to $(\bar{O}', \in, \leq)$ in such a way such that every object x from $\bar{O}$ has its own implicit meta-object, the *eigenclass* of x, denoted $x.ec$. Therefore,

$\bar{O}' = \bar{O} \uplus \bar{O}.ec$,

where $\bar{O}$ are the *primary objects*, and $\bar{O}.ec$ are *eigenclasses*. The $x \mapsto x.ec$ assignment is displayed by gray left-to-right arrows.



$x \in y$ iff $x \rightarrow \dots \rightarrow \dashrightarrow \rightarrow \dots \rightarrow y$

The inheritance relation is extended to $\bar{O}'$ as follows. For every x, y from $\bar{O}$,

- $x.ec \leq y.ec$ iff $x \leq y$, (That is, $.ec$ is an order embedding of $(\bar{O}, \leq)$ into $(\bar{O}', \leq)$.)
- $x.ec \leq y$ iff $x \in y$,
- $x \nleq y.ec$.

Object membership

The $\in$ relation, extended to $\bar{O}'$, equals $(\leq) \circ (.ec) \circ (\leq)$. In contrast to inheritance, we do not preserve the "instance-of" term for $\in$. Instead, we call the $\in$ relation *object membership*. If $x \in y$ then x is said to be a *member of* y. We can also say that y is a *container of* x.

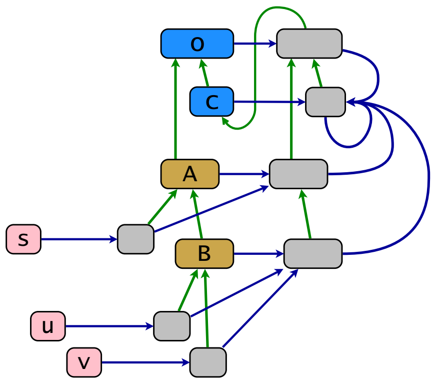
Proposition: For every x, y from $\bar{O}$,

◦ *x* ∊ *y* iff *x.ec* ≤ *y*.

That is, *x.ec* is the least container of *x*. In particular, *x* ∊ *x.ec* and *x.ec* < *x.class*.

The refined "class" map

Just like every object *x* in the primary structure has *x.class* as the unique least class that has *x* as its instance, in the refined structure *(Ō', ∊, ≤)*, every object has a unique least container. For an object *x* we denote *x.aclass* such a container, and call it the *actualclass* of *x*. In the diagram below, the *.aclass* map is shown by blue links.



$x \in y$ iff $x \rightarrow \rightarrow \ldots \rightarrow y$

Observations:

1. The *.aclass* map coincides with *.ec* on primary objects and equals the the composition $.ec^{-1}.class.ec$ on eigenclasses.
2. *(∊) = (.aclass) ⦾ (≤)*.

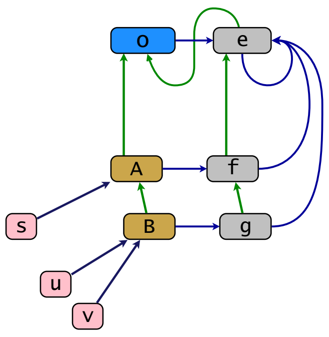
Support in programming languages

Of the languages mentioned above, only Ruby supports refinement presented in the previous subsection. There are some languages that provide partial support.

- In Smalltalk-80 and Objective-C, each class has its own *implicit metaclass*.
- In Scala, the `object` construct can be used to create ordinary objects that have its own meta-object.
- In Dylan, all the 7 eigenclasses have a correspondent but the *.aclass* map is different (see the sample structure).

In Objective-C

The refinement of the sample primary structure for Objective-C is shown on the following diagram. Blue links correspond to `isa` pointers, green links to `super_class` pointers. [13]

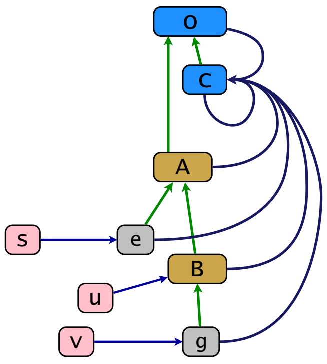


```
id
e = object_getClass(o),
f = object_getClass(A),
g = object_getClass(B);
```

The implicit metaclasses `e`, `f` and `g` are created together with classes `o` (the built-in `NSObject` class), `A` and `B`, respectively.

In Scala

The structure on the following diagram can be created using the `object` construct for `s` and `v` instead of the `new` operator used in the primary sample.



```
object s with A
val    u = new B
object v with B
val    e = s.getClass
val    g = v.getClass
```

Eigenclass completion

The structure $(\underline{\bar{O}}', \in, \leq)$ still provides only partial refinement since the pattern has only been applied to primary objects. A complete refinement arises when eigenclasses themselves have eigenclasses. We can extend $\underline{\bar{O}}'$ to $\underline{\bar{O}}''$ by eigenclasses of $\underline{\bar{O}}'$, then extend $\leq$ and $\in$ accordingly, then extend $\underline{\bar{O}}''$ to $\underline{\bar{O}}'''$ by eigenclasses of $\underline{\bar{O}}''$, and so on. This leads to infinite eigenclass chains. The resulting structure $(\underline{O}, \in, \leq)$ has infinitely many objects and can be characterized as an embedding of inifinite regress into inheritance.

Full uniformity of the structure provides the following simplifications:

- Object membership, $\in$, equals the composition $(.ec) \circledcirc (\leq)$, i.e. an object x is a member of an object y iff $x.ec$ is an inheritance descendant of y. (However, the equality $(\in) = (\leq) \circledcirc (.ec) \circledcirc (\leq)$ is still valid.)
- Inheritance is obtained from $\in$ as inclusion of sets of containers. Therefore, the structure can be expressed in the simple signature $(\underline{O}, \in)$.

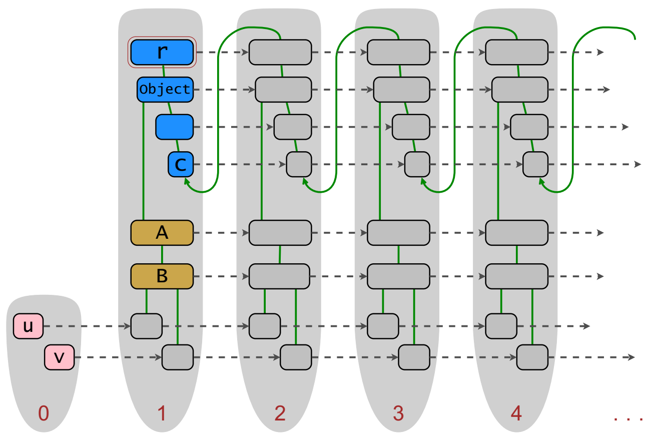
The core structure of Ruby

In this section we describe the completely refined structure $(\underline{O}, \in, \leq)$ anew, using Ruby as the definitive sample language. The recapitulation allows us to establish consistent notation and terminology.

Sample structure

The following diagram shows a sample for the description of the core structure of Ruby. Note that the structure is just the eigenclass completion of the sample primary structure except that

- the *s* object is removed to simplify the diagram, and
- all four circular classes of Ruby 1.9 are contained in the structure, not just `Object` and `Class`.



(Ruby 1.9)

```
r = BasicObject
c = Class
class A; end
class B < A; end
u = B.new
v = B.new
```

There are infinitely many metalevels, indexed 0, 1, 2, ... The structure is built using two sorts of links. Green links correspond to the `superclass` introspection method. A green line going from an object x upwards to an object y indicates that y is an (the) inheritance parent of x. For curved lines (those starting at the top of metalevel 2 and higher) the upward direction is expressed explicitly by an arrow. All the other superclass links are drawn as a Hasse diagram. Gray arrows (directed left-to-right) show the infinite regress of meta-objects. Each object x has its own meta-object, called the *eigenclass* of x, which is one metalevel higher than x.

The prefix *eigen* stands for *own* – different objects have different eigenclass. In the Ruby comunity, the term "eigenclass" appeared around the year 2005. Previously, the term "singleton class" has been used. As a (historical) consequence, the corresponding introspection method is named `singleton_class`.

We use the notation $x.ec$ for the eigenclass of x, so that $x.ec$ corresponds to `x.singleton_class`. The $.ec$ map is then the *eigenclass map* shown by gray arrows. Furthermore, let $x.ec(i)$ be the i-th application of $.ec$ to x. Components of the $.ec$ map are *eigenclass chains* of the form x, $x.ec$, $x.ec(2)$, $x.ec(3)$, ... The inverse of $.ec$ is denoted $.ce$. For x from $O.ec$, $x.ce$ is the *eigenclass predecessor* of x. Let $.ec*$ be the reflexive transitive closure of $.ec$. For an object y, let $y.pr$ be the *primary object* of y, that is, the first object of the eigenclass chain $y.pr.ec*$ to which y belongs.

Inheritance and membership

The reflexive transitive closure of green links is the *inheritance* relation between objects, denoted $\le$. Since the superclass links are acyclic, $\le$ is a partial order. We also let $<$ denote the *strict* inheritance with the obvious meaning. It can be observed that the eigenclass map is an order-embedding with respect to inheritance. For every objects x, y,

$x \le y$ iff $x.ec \le y.ec$.

Object membership, $\in$, equals the composition $(.ec) \odot (\le)$, i.e. an object x is a *member of* an object y iff $x.ec$ is an inheritance descendant of y. For a positive natural n, let $\in^n$ denote the n-th composition of $\in$ with itself, that is, $\in^1$ equals $\in$ and for objects x, y

$x \in^n y \;\leftrightarrow\; x = x_0 \in x_1 \in \cdots \in x_{n-1} \in x_n = y$ for some objects $x_1, \ldots, x_{n-1}$.

For now, we consider $\in^0$ to be equal to the identity relation between objects. Later in this document, we redefine $\in^0$ to $\le$ according to the definitions for basic structures. The reflexive transitive closure of $\in$ is denoted $\in*$. We also introduce a notation for images and preimages under $\le$ and $\in$.

- $x.{\uparrow}$ (resp. $x.{\downarrow}$) denotes the set of (non-strict) ancestors (resp. descendants) of an object x.
- Similarly, for a set X of objects, $X.{\uparrow}$ (resp. $X.{\downarrow}$) is the upset (resp. downset) of X.
- $x.\in$ (resp. $x.\ni$) is the set of all containers (resp. members) of an object x.
- $x.\in*$ is the set $x.\in^0 \cup x.\in^1 \cup x.\in^2 \cup \ldots$.

Observe that for every objects x, y,

(1) $x.{\downarrow} = x.ec.\ni$ (descendants of x are exactly the members of $x.ec$),
(2) $x.\in = x.ec.{\uparrow}$ (containers of x are exactly the ancestors of $x.ec$),
(3) $x \le y$ iff $x.\in \supseteq y.\in$ (x is a descendant of y iff every container of y is also a container of x),
(4) $x.ec = y$ iff $x.\in = y.{\uparrow}$ (the eigenclass of x is the least container of x).

Conditions (1) and (2) are the $(\geq) = (.ec) \circ (\ni)$ and $(\in) = (.ec) \circ (\leq)$ equalities, respectively. As a consequence of (3) and (4), the whole structure $(\underline{O}, .ec, \leq)$ is given just by object membership, $(\underline{O}, \in)$. Note that the first equivalence implies that $\in$ is co-extensional – objects are distinguished by their containers. In contrast, $\in$ is typically not extensional.

Direct inheritance

Green links in the diagram correspond to *direct (strict) inheritance* – the reflexive transitive reduction of $\leq$. We might sometimes use the $\prec$ symbol for this relation. If $x \prec y$ then x is an *(inheritance) child* of y and y is an *(inheritance) parent* of x. We will mostly refer to direct inheritance through its image map, *.parents*. For an object x, $x.parents$ (as an abbreviation of $\{x\}.parents$) denotes the set of inheritance parents of x.

The notation is adapted to multiple inheritance. If single inheritance is asserted, we can use the *.sc* notation for $\prec$ so that $x.sc = y$ iff $x \prec y$. For an object x, the object $x.sc$, if defined, is the *superclass* of x. The core structure of Ruby can be then expressed as $(\underline{O}, .ec, .sc)$. [42]

Membership as is-a

The object membership relation, $\in$, between the objects of the sample structure can be detected in Ruby by the `is_a?` introspection method. The method is aliased as `kind_of?`. (There is also the `Class#===` introspection method, which detects the inverse relation, $\ni$.)

If x is a member of an object named `B` then one can say that x *is-a* `B`. The set `B.∋` of all members of `B` can be referred to as `B`s. This convention can be used for nomenclature of objects. Moreover, there can be a distinction between the names that are built using this convention (these names are capitalized and styled) and those names that are not. In Ruby, the built-in classes `Object`, `Module` and `Class` provide such a distinction.

- `Object`s are objects that are not "blank slate" objects.
- `Module`s are modules together with `Class`es.
- `Class`es are classes together with eigenclasses.

Note in particular that we regard *is-a* as another name for object membership, $\in$, not for inheritance, $\leq$. Unfortunately, in object modeling, and in most publications about object oriented programming, "is-a" corresponds to inheritance. The reasoning goes as follows. If `A` and `B` are classes such that `A ≤ B` (`A` is a subclass of `B`) then one can say that

an `A` *is a* `B`,

meaning that an instance of `A` is also an instance of `B`. The sentence structure can be expressed as "an-A is a-B" (two pairs of article-noun and a verb between). However, the longest characteristic subsequence is "is a" which leads to another hyphenation and to saying that

there is an *is-a* relationship between `A` and `B`.

This is in turn abbreviated to "`A` *is-a* `B`" (so that the `A`'s article is dropped and the `B`'s article is secluded from `B`), meaning that the `(A,B)` pair belongs to the *is-a* relation. This abbreviated statement then becomes ambiguous since one way to interpret it is

the ~~A *is a* B~~

which is fundamentally different from the original statement.

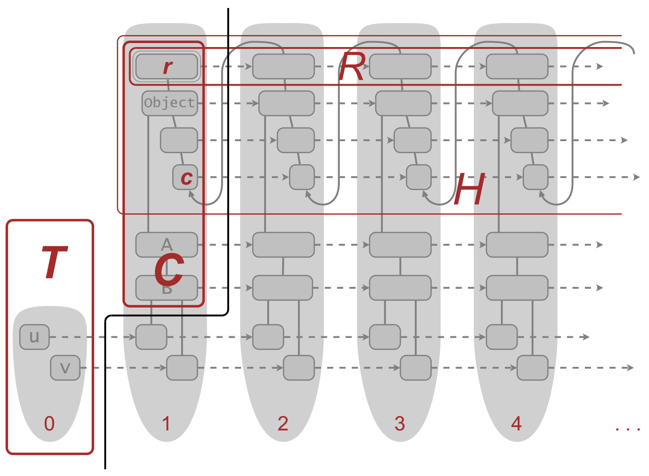
Note: The two different meanings of "*is-a*" have been analyzed by R. Brachman [7] [39].

- The $\in$ meaning is called *individual/generic* and is exemplified by "Socrates is a man".
- The $\leq$ meaning is called *generic/generic* and is exemplified by "a cat is a mammal".

The ambiguity of "*is-a*" can be resolved according to the adherence to the uniformity principle stated in the introduction. If classes are not considered to be objects (individuals) then "*is-a*" is resolved in favor of $\leq$. Otherwise, if classes are among objects (as is our assumption) it is reasonable to interpret "*is-a*" as $\in$. At present, this second interpretation is supported only rarely. The Ruby introspection method `is_a?` and the Objective-C `isa` pointers are two notable examples. Others can be found in publications about Smalltalk [36] or Self [59]. Another example is the book by Forman and Danforth [21] which uses *isA* for $\in$. A more detailed discussion can be found in [51].

Basic nomenclature of objects

We introduce the basic nomenclature of objects according to the following diagram.



O	… objects	= *T* ⊎ *C* ⊎ *O.ec*
O.ec	… eigenclasses	
O.pr	… primary objects	= *O.c* = *T* ⊎ *C*
T	… terminals	= *O* \ *r*.↧
C	… classes	= *O.pr* \ *T*
H	… helix objects	= *r*.ϵ* = *r*.↧.*he* = *R*.↥
R	… reduced helix	= *r.ec** = *r*.↧.*re*
r	… inheritance root	
c	… metaclass root / instance root	
r.↧	… `Class`es	= *c*.϶ = *O*.ϵ.↧
c.↧	… metaclasses	= *C.class*.↧
r.ec(i).϶ \ *r.ec(i)*.↧	… *i*-th metalevel	

We denote *O* the (infinite) set of all objects.

- *O.ec* (the image of the eigenclass map, shown right to the black line) is the set of *eigenclasses*.
- *O* \ *O.ec* (objects that are NOT eigenclasses, to the left of the black line) is the set of *primary* objects.
- *T* denotes the set of *terminal* objects or just *terminals* – objects without members and without strict ancestors.
- *C* is the set of *classes*, the primary non-terminal objects.
- *r* is the *inheritance root*. It is the highest non-terminal object, w.r.t. inheritance.
- *c* is the *instance root* and is also called the *metaclass root* and named `Class`. It is the unique inheritance parent of *r.ec*.
- *R* is the set of objects from the eigenclass chain of *r*, the *reduced helix*.
- *H* is the set of *helix* objects – objects *x* such that *x* ϵ *x*. Note that *H* = *R*.↥.

Previously, we used the term "ordinary objects" for elements of *T*. The new terminology is similar to that introduced in the ObjVlisp model [9] [10] [14] where the term "terminal instances" is used. Note that for an object *x* the following are equivalent:

- *x* is non-terminal.
- *x* is a `Class` (i.e. *x* is a member of the *c* object which is named `Class`).
- *x* is an inheritance descendant of *r* (i.e. *x* ≤ *r*).

In particular, non-terminal objects can be referred to as `Class`es.

The terminology for the set *H* = *r*.ϵ* is based on the similarity of the diagrammatization of *(H, ≤)* with a right-infinite helical curve.

Classification systems

By a *classification system* we mean a closure system *Y* on `Class`es. I.e. it is a set of distinguished non-terminal objects such that every non-terminal *x* has a least ancestor from *Y*. Equivalently, *Y* is such that every object *x* has a least container from *Y*. If *.y* is the corresponding closure operator (that maps a non-terminal object to its least ancestor from *Y*), then the corresponding *classification map* (that maps any object to its least container from *Y*), equals *.ec.y*. Observe that since both *.ec* and *.y* are monotone, any classification map is monotone, i.e. for every objects *a*, *b*,

- if *a* ≤ *b* then *a.ec.y* ≤ *b.ec.y*.

Note that the sets *C*, *R* and *H* are all classification systems. (In fact, since in Ruby the inheritance on `Class`es forms a tree, any subset of *O* \ *T* containing *r* is a classification system.) We denote the corresponding closure operators by *.c*, *.re* and *.he*, respectively.

In the case of *.c*, we extend the definition to all objects: we let *.c* to be the closure operator corresponding to the closure system *C* ⊎ *T* in *(O, ≤)*. This allows us to conveniently express the set of all primary objects as *O.c*.

Metalevels

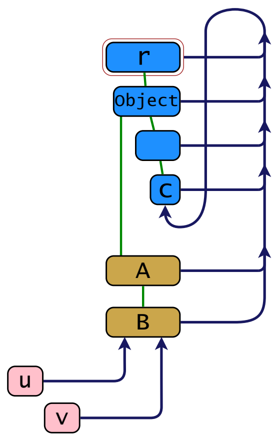
The set *R* is the classification system for *metalevels*. An object *x* belongs to the *i*-th metalevel (and therefore has the *metalevel index* *x.mli* equal to *i*) iff *x.ec.re* = *r.ec(i)*.
Further observations:

- The set *T* of terminal objects is the metalevel 0.
- For *i* > *0*, the *i*-th metalevel has a top, equal to *r.ec(i-1)*.
- The *i*-th metalevel, *i* ≥ *0*, equals the set *r.ec(i)*.∋ \ *r.ec(i)*.↧.
- For an object *x*, the metalevel index *x.mli* equals the cardinality of *x*.↥ ∩ *R*.
- The last metalevel containing a primary object is isomorphic to any higher metalevel (via *.ec(i)* for a suitable *i*).

Classes and the *.class* map

The set of *all* `Class`es forms a trivial classification system, with *.ec* the corresponding classification map. However, the eigenclass map is not useful for classifying objects since different objects have different eigenclass, so that each object is classified as different from every other object.

The most important classification system is the set *C* of primary non-terminal objects. Objects from *C* are called *classes*. The corresponding classification map is denoted *.class*. (So that *.class* = *.ec.c*.) For an object *x*, the object *x.class* is said to be the *class of x*. Note that the just introduced meaning of "class" and "class-of" possesses the following fundamental consistency:



The class of *x* is the least container of *x* that is a class.

Further observations:

- Classes are primary `Class`es.
- `Class`es are classes together with eigenclasses.
- *x.class* ≠ *x.ec* for every object *x*.
- The *.class* map is derived from the eigenclass map and the inheritance relation.
- The distinguished objects *r* and *c* are classes.
- The *.class* map forms a tree rooted at *c*. In particular, *c* is the only object that is the class of itself.
- Unlike eigenclasses, classes cannot be in general expressed as *O.class* (the image of the class map).

The diagram on the right shows the class map for the sample structure, in the restriction to primary objects. To illustrate the *.ec.c* composition, the arrows are drawn along eigenclass paths.

In Ruby, all classes reside in the metalevel 1. As a consequence, *.class.class* maps constantly to *c*. The Ruby introspection method named `class` is in 100% correspondence with the *.class* map.

The ***instance-of*** relation

Just like the *.class* map is a coarsement of the eigenclass map, the composition *(∊)* ⊙ *(.c)* is a coarsement of object membership. This relation is called *instance-of* and can be equivalently defined as the range-restriction of ∊ to classes, i.e. for objects *x*, *y*,

- *x* is an *instance of y*   iff   *x* ∊ *y* and *y* is a class.

The *.class* map, taken as a relation, is called *direct-instance-of*. Note that the term "instance" as such does not distinguish any objects. Every object is an instance of some class. In particular, every object is an instance of the inheritance root *r* and every object is a direct instance of its class.

By allowing indirect instances our terminology follows the semantics of the `isinstance()` method of Python or the `instanceof` operator from Java. In Ruby, the introspection method `instance_of?` corresponds to direct-instance-of. (However, the Ruby Specification [25] allows indirect instances.) Note that

if `y` is the eigenclass of `x` then `x.instance_of?(y)` always evaluates to `false`.

An object is never an instance of its eigenclass.

Metaclasses

In order to establish a correspondence with other programming languages, we introduce the term "metaclass":

Metaclasses are exactly the inheritance descendants of *c* (including *c* itself), so that the set of all metaclasses can be expressed as *c*.↧.

This in particular ensures that classes of classes (*C.class*) are metaclasses, as well as the classes of eigenclasses (*O.ec.class*).

Observations:

- The *c* class is the only metaclass from metalevel 1.
- The set of all metaclasses other than *c* can be expressed as *r.ec.*↧ – it consists of all objects from metalevel 2 or higher.
- Suppose that our sample structure has been extended by `w = Module.new` so that the `Module` class (which is the parent of *c*) has a terminal member `w`. Then
  - metaclasses are exactly the `Class`es that do not have terminal members.

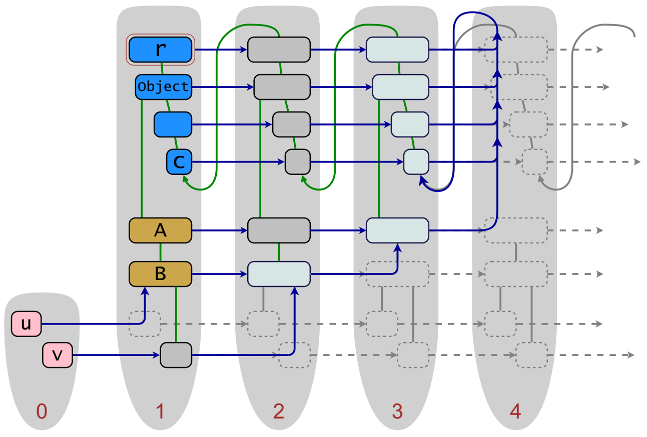
Metaclasses are said to be *explicit* or *implicit* according to whether they are classes or eigenclasses, respectively. Note that by naming the eigenclass *c.ec* by `Metaclass` we could express the set *c*.↧ (= *c.ec.*∍) as `Metaclass`es. This suggests to define metaclasses as the primary `Metaclass`es just like classes are primary `Class`es. However, this would require an additional term for non-primary `Metaclass`es. Moreover, the "explicit" / "implicit" adjectives seem to be an established convention. [21] [16] [55]

Eigenclass actuality

Since eigenclass chains are infinite they must be lazily evaluated. There can be only finitely many objects that are actually represented (allocated). We call such objects *actual*. A natural direction for evaluation of eigenclass chains is that of the eigenclass map: if *x.ec* is actual then *x* must be actual. Let all primary objects be actual, so that the set of actual objects forms a closure system in *(O, ≤)*.

We denote *.a* the corresponding closure operator and let *.aclass = .ec.a* be the classification map. For an object *x*, *x.aclass* is the *actualclass of x* – the least ancestor of *x.ec* that is actual. The set *O.a* of all actual objects is the *actuality extent*. We can view the *(O, ∊, .a)* structure as an implementation-oriented refinement of *(O, ∊)*. In particular, *x.aclass* can be regarded as the actual startpoint of method lookup for *x* – non-actual eigenclasses between *x.ec* and *x.aclass* are skipped.

The diagram below shows the actuality extent of allocated objects for the sample structure. The extent arises after opening the eigenclasses of `A` and `v`. The actualclass map (in the restriction to actual objects) is shown by blue arrows. Note that the map forms a tree rooted at *c.ec(2)*.



(Ruby 1.9)

```
r = BasicObject
c = Class
class A; end
class B < A; end
u = B.new
v = B.new

class << A; end
class << v; end
```

Note that the eigenclasses `A`*.ec(2)* and `B`*.ec* have been allocated but not opened. This is an implementation feature of Ruby. Every newly created class *x* has its eigenclass allocated. When the eigenclass *x.ec(i)* is evaluated (e.g. by `class << x; end` for *i = 1*) then MRI/YARV makes sure that *x.ec(i+1)* is allocated. As a consequence, we can distinguish 2 actuality extents: one for actually *referenced* (evaluated) objects and a larger one for *allocated* objects.

As already mentioned, the diagram shows the structure for the larger extent. Objects that have been allocated but not referenced are shown in light blue. The actualclass map shown by blue links could be denoted *.klass*, since it is maintained by the implementation via the `klass` field in the C source. Note that there cannot be an introspection method for *.klass* due to the *probe effect*. There could only be an introspection method for the actualclass map for the smaller extent, but even this is not supported since it is considered to be an implementation property.

The reason for having an extra object allocated in each eigenclass chain of classes is an efficient update of the actualclass map for inheritance descendants. This does not affect terminal objects (or their eigenclasses). Since they cannot have descendants, they do not need to have their eigenclass allocated.

The core structure of Smalltalk-80

The Smalltalk-80 programming language is the first language which introduced the refinement of the instance-of relation by implicit objects. Simultaneously, Smalltalk is one of the first proponents of the "classes are objects" pattern.

Unfortunately, the Smalltalk-80 core structure possesses uniformity deficiencies which were compensated by equivocal terminology in the Smalltalk's literature. [48]

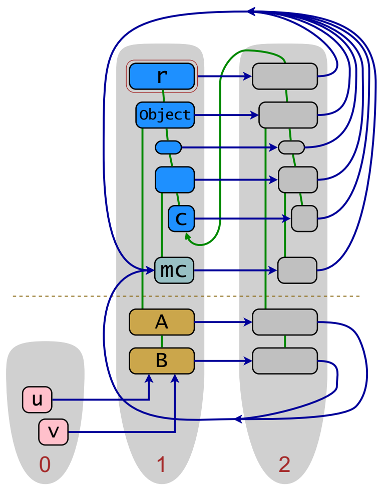
To describe the Smalltalk-80 counterpart of the Ruby core structure we use a specially restrained terminology in which the following terms are (at first) avoided:

- **Class** — we do not say which objects are *classes* and which not.
- **Metaclass** — we do not say which objects are *metaclasses* and which not.
- **Class-of**. We do not say what is *the class of* an object. In particular, even if the expression `x class` evaluates to `y` we do not say that `y` is *the class of* `x`.
- **Instance-of**. We do not say what it means for objects *x* and *y* that *x* is *an instance of y*.

Sample structure

The following diagram shows a sample core structure of Smalltalk-80. The structure consists of 18 objects with oriented links between them. There are two sorts of links. The green links correspond to the `superclass` introspection method. There is a single line for which the upward direction is expressed explicitly by an arrow. All the other superclass links are drawn as a Hasse diagram.

The blue links correspond to the `class` introspection method. A blue line starting at `x` and ending in `y` (with an arrow head pointing to `y`) indicates that the expression `x class` evaluates to `y`.



(Pharo 1.3 / Squeak 4.2)

```
 r := ProtoObject.
 c := Class.
mc := Metaclass.
Object subclass: #A.
A      subclass: #B.
u := B new.
v := B new.
```

The horizontal dashed line indicates the division of the structure into the built-in part (above the line) and the user-created part. In the code right to the diagram, first distinguished built-in objects are denoted (by `r`, `c` and `mc`) and subsequently the user-part of the structure is created.

The built-in part forms a substructure — it contains exactly the objects reachable from the `mc` object via a combination of blue and green links. (Note that the substructure is generated by any single object it contains, not just `mc`.) Moreover, it is a minimum (nonempty) substructure, since the `mc` object is reachable from any object (via at most 3 blue links).

Metalevels

The sample diagram shows 3 types of blue links. The *curved* blue links point to the `mc` object. The *straight* left-to-right links precede the curved links. Each curved link is preceded by exactly one straight link. The remaining, *broken*-line links precede the straight links, but without a one-to-one correspondence.

The division of blue links induces a division of objects into 3 *metalevels*.

- The metalevel **2** consists of all objects `x` such that `x class == mc`.
- The metalevel **1** consists of all objects `x` such that `x class class == mc`.
- The metalevel **0** consists of all remaining objects. For every `x` from the metalevel 0,

  `x class class class == mc`.

  However, this equality cannot be used to distinguish the metalevel 0 since it also holds for every object from the metalevel 2.

The sample structure contains 8 objects from metalevel 2 (shown in gray), 8 objects from metalevel 1, and 2 objects from metalevel 0 (denoted `u` and `v`).

Uniformity deficiency No. 1

Observe that the blue links provide the following mappings between metalevels:

- Objects of metalevel 0 are mapped to objects of metalevel 1, possibly many-to-one.
- Objects of metalevel 1 are mapped to objects of metalevel 2 in a one-to-one correspondence.
- Objects of metalevel 2 are constantly mapped to the distinguished object `mc` from metalevel 1.

This reveals a defect of uniformity of the Smalltalk-80 object model:

- Objects from the metalevel 1 have each an individual (meta)object from the next metalevel pointed to by a blue link. Objects from other metalevels do not have such an own meta-object.

Inheritance

We denote ≤ the reflexive transitive closure of green links, i.e. for objects x, y

$x \leq y$ iff y is reachable from x via zero or more green links.

This relation forms a partial order, called *inheritance*.

Further observations:

1. Objects from metalevel 0 are exactly the singletons in inheritance: they have no (strict) descendants or ancestors.
2. The object `r` is the top of the metalevel 1 as well as of the union of metalevels 1 and 2, w.r.t. inheritance.
3. The object `r class` is the top of the metalevel 2, w.r.t. inheritance.
4. The parent of `r class` is the object `c` from metalevel 1.
5. In the restriction to a metalevel, the blue links constitute a map that is *monotone* w.r.t. inheritance, i.e. if x and y are from the same metalevel, then

   $x \leq y$ implies `(`x` class)` ≤ `(`y` class)`.
6. Since blue links constitute bijection between metalevels 1 and 2, they constitute an order isomorphism between these metalevels, w.r.t. inheritance. If `x` is an object from metalevel 1 different from `r`, then

   `x class superclass == x superclass class`.

Uniformity deficiency No. 2

The following observation reveals another defect of uniformity of Smalltalk-80:

- Without the restriction to a particular metalevel, the blue links do NOT constitue a monotone map.

For example, if `x := r class` and `y := c`, then `x ≤ y` (we even have `x superclass == y`) but `x class` equals `mc` which is not an inheritance descendant of `y class`.

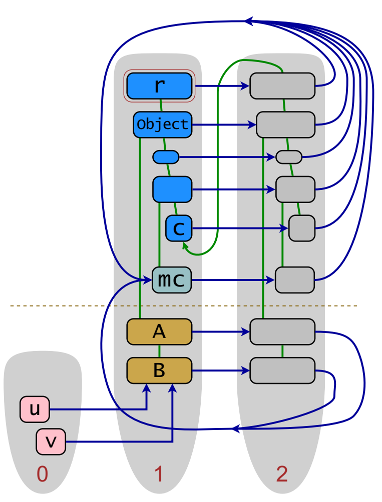
The Ruby-Smalltalk core correspondence

The correspondence between the core structures of Smalltalk-80 and Ruby can be observed on the following pair of diagrams. The Ruby structure has the actuality extent $\underline{O}$*.a* = $\underline{T}$ ⊎ $\underline{C}$ ⊎ $\underline{C}$*.ec*, so that

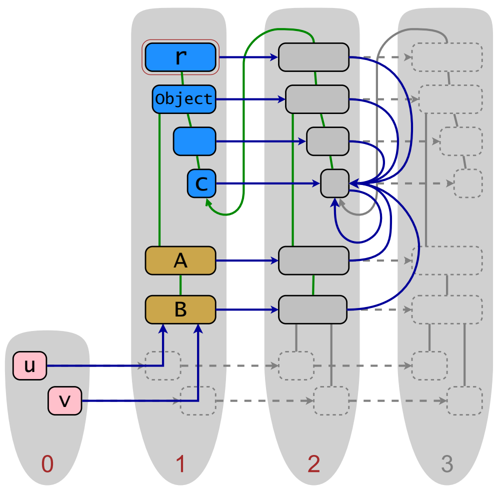
- $\underline{T}$ is the set of actual objects on metalevel 0,
- $\underline{C}$ is the set of actual objects on metalevel 1,
- $\underline{C}$*.ec* is the set of actual objects on metalevel 2,

and there are no more metalevels with actual objects.

Smalltalk-80



Ruby



We now resolve the terms "class", "class-of", "metaclass" and "instance-of" by applying Ruby-based terminology and notation to Smalltalk. Let

- *terminals* ($\underline{T}$) be the objects from metalevel 0,
- *classes* ($\underline{C}$) be exactly the objects on metalevel 1, and
- *implicit metaclasses* ($\underline{C}$*.ec*) be the objects from metalevel 2,

In particular, the metalevel 1–2 correspondence is a correspondence between classes and implicit metaclasses.

The differences between the core samples can be listed as follows:

1. The Smalltalk-80 core contains two more built-in classes, of which only the `mc` class (named `Metaclass`) constitutes a structural change as a "receiver" of blue arrows.
2. The blue arrows starting at metalevel 2 are "redirected" to `mc`.

Since blue arrows in the Ruby sample display the actualclass map, *.aclass*, the blue arrows in the Smalltalk sample indicate the *imposed* actualclass map, which we denote *.aȼlass*. Therefore,

- *x.aȼlass* = *x.aclass* if *x* is a class or terminal,
- *x.aȼlass* = `mc` otherwise (i.e. if *x* is an implicit metaclass).

The *imposed class map*, *.ȼlass*, equals the composition *.aȼlass.c*, i.e. *x.ȼlass* is the least ancestor of *x.aȼlass* that is a class. The *imposed (object) membership* equals the composition *(.aȼlass)* ⊙ *(≤)*. This relation can be detected via the `isKindOf:` introspection method. The *imposed instance-of* relation is the range restriction of imposed membership to classes, so that it equals the composition *(.ȼlass)* ⊙ *(≤)*. By using imposed membership for the *is-a* naming convention, classes can be expressed as `Class`es and implicit metaclasses as `Metaclass`es. Moreover, $\underline{C}$*.ȼlass.*↧ can be considered to be the set of *all* metaclasses. The classes `Class` and `Metaclass` are then the only explicit metaclasses.

Notes:

1. Note in particular, that the `class` introspection method in Smalltalk-80 does not correspond to the synonymous method of Ruby.
   - In Ruby, the `class` introspection method corresponds to the class map, *.class*.
   - In Smalltalk-80, the `class` method corresponds to the imposed actualclass map, *.aȼlass*. As a consequence, `x class` is not necessarily a class.

   The following table describes the differences:

For x from	Smalltalk-80 `x class` evaluates to	Ruby `x.class` evaluates to
$\underline{T}$ (terminals)	the class of x	
$\underline{C}$ (classes)	the eigenclass of x	the class of x which is constantly `Class`
$\underline{C}.ec$ (implicit metaclasses)	the imposed class of x which is constantly `Metaclass`	

2. There is no support for the above or similar definitions in the Smalltalk literature. Instead, publications about Smalltalk-80 use equivocal terminology which suggests two different delimitations of classes and metaclasses: [48]

   (A) Classes are the objects from metalevels 1 and 2.
   Metaclasses are the objects from metalevel 2 plus the class named `Metaclass`.

   (B) Classes are the objects from metalevel 1.
   Metaclasses are the objects from metalevel 2.

The primary structure of ϵ according to Python

We now set out for a formal description of object membership. This section provides axiomatizations of the canonical primary structure. The axioms are general enough to allow two features that have not yet been captured by the provided samples:

- multiple inheritance (that is, an object can have multiple inheritance parents), and
- explicit metaclasses other than the "built-in" $\underline{c}$ class.

It turns out that under some minor assumptions the canonical primary structures of ϵ are exactly the structures that are allowed in the Python programming language [31] (of course, up to the number of helix classes). The correspondence can be expressed as

$(.class, .\!\!\uparrow) \quad \leftrightarrow \quad$ `(.__class__, .__mro__)`

where `__class__` and `__mro__` are (built-in) attributes of Python objects. The `__mro__` attribute is only applicable to classes. It stands for *method resolution order* and stores the list of inheritance ancestors in the order in which they are looked up during method resolution. The inheritance relation, $\leq$, between classes is obtained by

- $x \leq y$ iff $y \in x.$`__mro__`

so that the order of classes in the `__mro__` list is disregarded. We assume that a potential explicit manipulation of the attribute is a permutation (changing just the order of classes in the list).

Note: In addition to the `__mro__` attribute, Python classes also contain another inheritance related attribute: the `__bases__` list. Apart from the order within the list, the intended semantics of `.__bases__` is that of the *.parents* map. However, the correspondence is not asserted. In particular, the `__mro__` and `__bases__` attributes are not kept in sync if they are explicitly manipulated. We therefore think of *.parents* to be derived from `.__mro__` (satisfying the assumption above) rather than from `.__bases__`.

Recursive definition

We first provide a recursive definition which shows how the structure can be incrementally constructed. A *canonical primary structure* (of ϵ) is a structure $\mathcal{S} = (\underline{\bar{O}}, .class, .parents, \underline{r})$ where

- $\underline{\bar{O}}$ is set of *objects*,
- *.class* is the *class map* between objects,
- *.parents* is the *inheritance parent set* map from $\underline{\bar{O}}$ to the powerset of $\underline{\bar{O}}$,
- $\underline{r}$ is the *inheritance root*, a distinguished object.

We denote $\prec$ the relation on $\underline{\bar{O}}$ defined by $x \prec y$ iff $y \in x.parents$. For a set X of objects, we write $X.parents$ for the image of X under $\prec$. Therefore, for an object x, $x.parents$ is just an abbreviation for $\{x\}.parents$. We can also write $.parents(i)$ for the i-th application of $.parents$. Furthermore, let $\leq$ be the reflective transitive closure of $\prec$. The usual terminology and notation is used for $\leq$. Descendants of $\underline{r}$ form the set $\underline{C}$ of *classes*, the remaining objects are *terminal(s)*. Let $\underline{c} = \underline{r}.class$ be the *metaclass root*. Descendants of $\underline{c}$ are *metaclasses*.

The structure is then subject to the following conditions:

(py~1) $\prec$ is irreflexive and transitively reduced.

(py~2) All objects from $\bar{O}.parents \cup \bar{O}.class$ are classes.

(py~3) The $.class$ map is monotone w.r.t. $\leq$, i.e. $x \leq y$ implies $x.class \leq y.class$ for every objects x, y.

(py~4) If $\bar{O} = \bar{O}.parents \cup \bar{O}.class$ then (a) $\leq$ is a linear order, (b) $.class$ is constant, and (c) $r \neq c$.

(py~5) Classes of terminal objects are not metaclasses.

(py~6) The restriction of $\mathcal{S}$ to $\bar{O} \setminus \{x\}$ is a canonical primary structure for every $x \in \bar{O} \setminus (\bar{O}.parents \cup \bar{O}.class)$.

(py~7) The set C of classes is finite.

Observations:

- $\bar{O}.parents \cup \bar{O}.class$ is the set of objects that are not minimal w.r.t. $(\prec) \cup (.class)$.
- The restriction of the structure to $c.\updownarrow$ is described by (py~4). It is a "built-in" substructure containing no objects that are minimal w.r.t. $(\prec) \cup (.class)$.
- Condition (py~1) could be stated just for the built-in structure. It asserts that $\prec$ is a reflexive transitive reduction of $\leq$ so that there is a one-to-one correspondence between $.parents$ and $\leq$.
- (py~6) is the only recursive condition. It says that removing an object minimal w.r.t. $(\prec) \cup (.class)$ preserves the structure.
- We avoided recursive definitions of classes and metaclasses by introducing $\leq$.

New object attachment

The recursive definition says that any finite canonical primary structure can be incrementally built from its restriction to $c.\updownarrow$ by attaching new objects, one by one. Let an *attachment request* be a pair (P,q) such that

- P is the requested (possibly empty) set of inheritance parents, and
- q is the requested class,

so that $P = x.parents$ and $q = x.class$ for the new object x. An attachment request (P,q) is *acceptable* iff the following are satisfied:

(1) P is an antichain in $\leq$, i.e. if x and y are different objects from P, then $x \nleq y$.

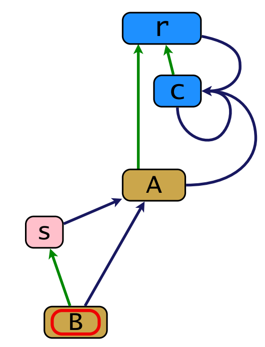
(2) Assert (py~2): Every object from $P \cup \{q\}$ is a class.

(3) Assert (py~3): For every x from P, $q \leq x.class$.

(4) Assert (py~5): If P is empty then q is not a metaclass.

Conditions (py~1)–(py~7) are preserved after the attachment of x if and only if (1)–(4) are asserted. Condition (1) can be left out if $x.parents$ is set to $mins_{\leq}(P)$ instead of to P. The following diagrams shows assertions of (2)–(4) in Python.

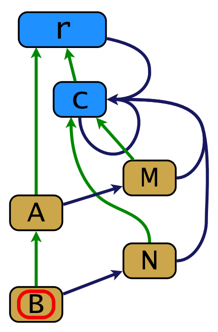
(a)
¬(py~2): `s` is not a class



```
class A():
  def __new__(a): pass
s = A()

# TypeError:
class B(s): pass
```

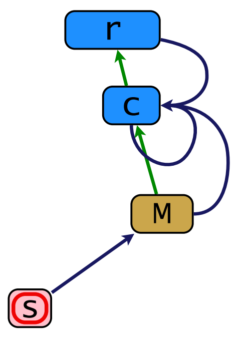
(b)
¬(py~3): `B`$.class \nleq$ `A`$.class$



```
class M(type): pass
class N(type): pass
class A(metaclass=M): pass

#  metaclass conflict
class B(A,metaclass=N): pass
```

(c)
¬(py~5): `M` is a metaclass



```
class M(type): pass
class X: pass
s = X()

# TypeError:
s.__class__ = M
```

Canonical primary structure

We now provide a "default" axiomatization of canonical primary structures. In the description, a preparation is already made for the extension by implicit objects. The set of objects in a primary structure is denoted $O.pr$. This expression can be later resolved to an application of $.pr$ to the extended set O of objects. Similarly, some sentences contain the adjective "primary" to be also applicable in the context of the extended structure.

By a *canonical primary structure* of ϵ we mean a structure $(O.pr, \epsilon, \leq, r)$ where

- $O.pr$ is set of *primary objects*,
- ϵ is the *instance-of* relation between primary objects,
- $\leq$ is the *inheritance* relation between primary objects,
- r is the *inheritance root*, a distinguished object.

The usual terminology and notation is used for $\leq$ and ϵ. Objects that are not descendants of r form the set T of *terminal(s)*, the remaining primary objects form the set C of *classes*. *Helix* objects x are such that $r \epsilon^i x$ for some $i \geq 0$. *Metaclasses* are the lower bounds of helix classes, i.e. they are objects x such that all helix classes are among ancestors of x. An object x is well-founded w.r.t. ϵ if there is no infinite sequence of objects $x_1, x_2, \dots$ of objects such that $x \ni x_1 \ni x_2 \ni \dots$. The structure is then subject to the following axioms:

(p~1) Inheritance, $\leq$, is a partial order.

(p~2) $(\epsilon) \odot (\leq) = (\epsilon) = (\leq) \odot (\epsilon)$. (That is, (a) $(\epsilon) \odot (\leq) \subseteq (\epsilon)$ and (b) $(\leq) \odot (\epsilon) \subseteq (\epsilon)$.)

(p~3) Terminals have no instances and no strict descendants.

(p~4) Helix classes are (a) totally ordered by $\leq$, (b) instances of each other, and (c) at least two in number.

(p~5) Metaclasses can only have classes as instances.

(p~6) Every non-helix object is well-founded w.r.t. ϵ.

(p~7) Every object x has a least primary container, $x.class$.

(p~8) The set C of classes is finite.

Notes and observations:

1. In RDF Schema, condition (p~2)(a) is asserted by the rdfs9 entailment rule. Using type theoretic terminology, this can be called the *subsumption rule*. [28]
2. Conditions (p~2)(b) and (p~6) can be conveniently expressed using the $.class$ map.
   - The $.class$ map is monotone. (Therefore, (p~2)(b) is the metaclass compatibility condition.)
   - The $.class$ map forms a tree.
3. Condition (p~2) can be equivalently written as a single equality: $(\leq) \odot (\epsilon) \odot (\leq) = (\epsilon)$.
4. Condition (p~3) means that terminals are minimal both in ϵ and $\leq$.

Proposition: Axiomatizations (py~1)–(py~7) and (p~1)–(p~8) are equivalent.

The canonical eigenclass structure of ϵ

Roughly speaking, the canonical refinement of the instance-of relation is established by combining Ruby with Python. This can be viewed in two ways:

(A) Equipping Python's core structure with eigenclasses.
(B) Relaxing Ruby's core structure by allowing multiple inheritance and classes on higher metalevels.

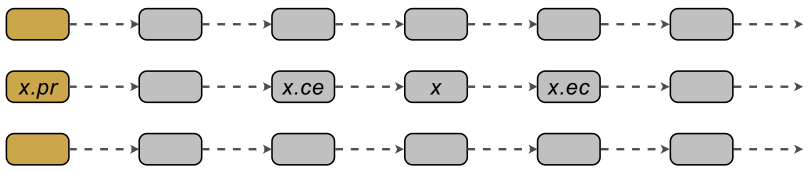
We provide precise interpretation of (A) and (B). The interpretation of (A) will be "tight" in the sense that every class has at most one eigenclass ancestor (necessarily $r.ec$). This seems to be the simplest way of eigenclass completion. However, since it also disallows classes on metalevels higher than 2, the interpretation of (B) defines a "slightly" larger family than that of (A). Moreover, some axioms for (B) are singled out for further generalization of object membership.

Eigenclass regress alone

The structure of infinite regress of eigenclasses can be simply characterized as a monounary algebra $(O, .ec)$ such that

- *.ec* is an injective well-founded map.

Elements of $\underline{O}$ are *objects*, *.ec* is the *eigenclass map* between objects. For an object *x*, *x.ec* is the *eigenclass of x*. We denote *x.ec(i)* the *i*-th application of *.ec* to *x*. (Let *x.ec(0) = x*.) Since the *.ec* map is injective, it has a partial inverse which we denote *.ce*. For an eigenclass *x*, *x.ce* is the *(direct) eigenclass predecessor* of *x*. Objects from $\underline{O}$*.ec* are *eigenclasses*, the remaining (i.e. those without an eigenclass predecessor) are *primary*. Components of $(\underline{O}, .ec)$ are *eigenclass chains*.



Since *.ec* is well-founded, every eigenclass chain starts in a primary object. For an object *x*, let *x.pr* denote the *primary object* of *x*, (i.e. the primary object of the eigenclass chain to which *x* belongs). The unique natural *i* such that *x.pr.ec(i) = x* is denoted *x.eci* and called the *eigenclass index* of *i*.

Observations:

1. For each eigenclass chain *X*, *(X, .ec)* is isomorphic (via *.eci*) to the structure $(\mathbb{N}, succ)$ of natural numbers where *succ* is the successor operator.
2. $\underline{O} = \underline{O}.pr \uplus \underline{O}.ec$.

Notes:

- As of version 2.0 or older, Ruby does not provide any introspection methods for *.ce*, *.pr* or *.eci*. Internally, the *.ce* links are implemented using the `__attached__` instance variable.
- ABCL/R is another example of a programming language that supports infinite regress. In the documents [33] [34], the term *metaobject* (or *meta-object*) is used for "eigenclass". An eigenclass chain is a *metaobject tower*. As the term suggests, the regression is diagrammatized vertically. The code expressions `[meta x]` and `[den x]` correspond to *x.ec* and *x.ce*, respectively.

Tight canonical structure by eigenclass completion

By a *tight canonical eigenclass* structure of $\in$ we mean a structure $(\underline{O}, .ec, \leq, \underline{r})$ where $\underline{O}$ is a set of *objects*, *.ec* is the *eigenclass* map $\underline{O} \to \underline{O}$, $\leq$ is the *inheritance* relation between objects, and $\underline{r}$ is a distinguished object. The structure is subject to conditions (ec~1)–(ec~3) below. Let the *object membership* relation, $\in$, be the composition $(.ec) \circ (\leq)$. Additional terminology and notation is induced by the first two conditions. In particular, *.pr* is the *primary object* map (obtained from *.ec*) and $\underline{c}$ is the *metaclass root*.

(ec~1) *.ec* is an injective well-founded map.

(ec~2) $(\underline{O}.pr, \in, \leq, \underline{r})$ is a primary structure of $\in$.

(ec~3) For every primary objects *a*, *b* and every natural $i, j > 0$, the following are satisfied:

(A) $a.ec(i) \leq b$ iff $a \dot{\in}^{i} b$ where $\dot{\in}$ denotes the restriction of $\in$ to primary objects.
(B) $a \leq b.ec(j)$ iff $a < \underline{c}$, $b = \underline{r}$ and $j = 1$.
(C) $a.ec(i) \leq b.ec(j)$ iff $a.ec(i\text{-}1) \leq b.ec(j\text{-}1)$.

The definition is in fact a prescription for the eigenclass completion of a primary structure. Given a canonical primary structure $(\underline{O}.pr, \in, \leq, \underline{r})$, the construction steps are as follows:

1. Equip each primary object with an eigenclass chain.
2. Extend $\leq$ according to (ec~3).
3. Extend $\in$ by setting $(\in) = (.ec) \circ (\leq)$.

Canonical eigenclass structure

By a *canonical eigenclass* structure of $\in$ we mean a structure $(\underline{O}, .ec, \leq, \underline{r})$ where $\underline{O}$ is a set of *objects*, *.ec* is the *eigenclass* map $\underline{O} \to \underline{O}$, $\leq$ is the *inheritance* relation between objects, and $\underline{r}$ is a distinguished object. The following additional terminology and notation applies:

The *object membership* relation, $\in$, is the composition $(.ec) \circ (\leq)$, *.ec** is the reflexive transitive closure of

$.ec$, $\epsilon*$ is the transitive closure of $(\leq) \cup (\epsilon)$. Let $.ce$, $.ce*$, $\ni$ and $\ni*$ be inverses of $.ec$, $.ec*$, ϵ and $\epsilon*$, respectively.
For an object x, the set $x.ec*$ is the *eigenclass chain* of x, the set $x.↧$ / $x.↥$ / $x.\ni$ / $x.\epsilon$ is the set of *descendants* / *ancestors* / *members* / *containers* of x. The $.pr$ map is defined by: $x = y.pr \;\leftrightarrow\; \{x\} = y.ce* \setminus \underline{O}.ec$.
Let $\underline{O}.ec$ be the set of *eigenclasses*, the remaining object being *primary*, $\underline{T} = \underline{O} \setminus \underline{r}.↧$ be the set of *terminal objects*, $\underline{C} = \underline{r}.↧ \setminus \underline{O}.ec$ be the set of *classes*, $\underline{R} = \underline{r}.ec*$ be the *reduced helix*, $\underline{H} = \underline{r}.\epsilon*$ be the set of *helix* objects.

A canonical eigenclass structure of ϵ is subject to the following axioms (the separator delimits the 5 axioms of monotonic eigenclass structures):

- •(e~1) Inheritance, $\leq$, is a partial order.
- (e~2) The eigenclass map, $.ec$, is an order-embedding of $(\underline{O}, \leq)$ into itself.
- •(e~3) Objects from eigenclass chains of terminals are minimal in inheritance.
- (e~4) Every eigenclass is a descendant of the inheritance root.
- (e~5) $\underline{R}$ has no lower bound in $\leq$.

- •(e~6) Helix classes are (a) totally ordered by $\leq$, (b) an upset in $\leq$, and (c) at least two in number.
- •(e~7) Objects from $\underline{R}$ have no siblings in $\leq$.
- •(e~8) Every non-helix object is well-founded w.r.t. ϵ.
- (e~9) Descendants of non-helix eigenclasses are eigenclasses.
- (e~10) For every object x and every class y such that $x.ec \;\epsilon\; y$, there is a class a such that $x \;\epsilon\; a \;\epsilon\; y$.
- •(e~11) Every object x has a least primary container, $x.class$.
- •(e~12) The set $\underline{C}$ of classes is finite.

Conditions marked with "•" have a direct counterpart in the axiomatization of the primary structure.
Notes and observations:

- Axiom (e~2) is the essential axiom. It can be equivalently expressed as $(\leq) = (.ec) \odot (\leq) \odot (.ce)$.
- Axiom (e~3) asserts that terminals are unrelated by $<$ to any object and that this property is preserved after a "free instantiation" of any class.
- Axiom (e~4) asserts that $\underline{r}$ is a "universal object": $\underline{O} = \underline{r}.\ni$. Since $\underline{r}$ is asserted to be a class by e~(1)(2)(4)(5), every object is an *instance of* $\underline{r}$.
- Axiom (e~5) asserts that $.ec$ is a well-founded map. Due to the injectivity asserted by (e~2), $.ec$ establishes the infinite regress of eigenclasses.

- Axiom (e~6) asserts that the restriction to $\underline{H}$ is (a)(b) as simple as possible but (c) non-degenerate: $\underline{H} \neq \underline{R}$. Using the finiteness condition (e~12), $\underline{H}$ is asserted to form a closure system in $(\underline{O} \setminus \underline{T}, \leq)$. In addition, the existence of a least helix class, $\underline{c}$, is ensured.
- Axiom (e~7) asserts that the eigenclass chain of $\underline{c}$ is not used for the connection to the helix. Using $.he$ as the closure operator for $\underline{H}$ (i.e. such that $\underline{r}.↧.he = \underline{H}$), (e~7) can be expressed as $\underline{c} \notin \underline{C}.he.pr$.
- Axiom (e~8) asserts that $\underline{H}$ is exactly the non-well-founded part of the structure. Moreover, $x \in \underline{H}$ iff $x \;\epsilon\; x$.
- Axioms (e~9) and (e~10) assert a one-to-one correspondence between the well-founded part $(\underline{O} \setminus \underline{H}, \ldots)$ and its restriction $(\underline{O}.pr \setminus \underline{H}, \ldots)$ to primary objects.
- Axiom (e~10) can be equivalently stated as any of the following:
  - the *member-of-instance-of* relation is equal to the *instance-of-instance-of* relation,
  - $.ec.class = .class.class$ (using (e~11)),
  - $.c$ preserves ϵ (using (e~11))
- Axiom (e~11) asserts that the set $\underline{O}.pr$ of primary objects forms a closure system in inheritance. We denote $.c$ the corresponding closure operator so that $\underline{O}.pr = \underline{O}.c$ and $.class = .ec.c$.
  Moreover, this axioms makes (e~4) redundant. (This is because if e~(1)(2) is assumed then: (e~4) $\leftrightarrow$ every object has a primary container.)

Notes and observations:

1. A canonical eigenclass structure of ϵ is expressible in the $(\underline{O}, \epsilon)$ signature.
2. Since (e~4) asserts that every object has a primary container, it becomes redundant within canonical structures due to (e~11).

Correspondence between primary and eigenclass structures

The correspondence between the primary structures and the eigenclass structures is described as follows.
Proposition:

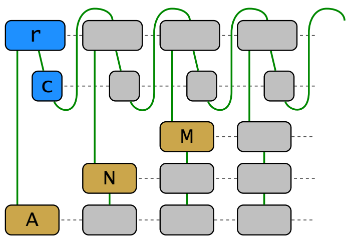
1. The restriction of a canonical eigenclass structure to primary objects is a canonical primary structure.
2. The *.c* map is a *homomorphic projection* of *(O, ∈, ≤)* onto *(O.pr, ∈, ≤)*.
3. A tight canonical eigenclass structure is a canonical eigenclass structure satisfying the following stronger version of axiom (e~9):

   (e~9$^+$) Descendants of eigenclasses-other-than-*r.ec* are eigenclasses.

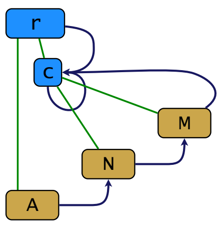
Note that (e~9$^+$) restrains the helix entry *x.he* of explicit metaclasses x other than c to *r.ec*. As a consequence, if x, y are explicit metaclasses different from c such that $x \in y$ then $\leq$ cannot be a single inheritance. (Ancestors of *x.ec* cannot be linearly ordered in $\leq$ since y and *x.ec.re* are incomparable.) A non-tight canonical structure (so that (e~9$^+$) is not imposed) allows single inheritance for arbitrary depth of instantiation.

The diagrams below show the correspondence between a canonical eigenclass structure, its restriction to primary objects and the subsequent "tight" eigenclass completion. Note that the tree structure of $\leq$ is not preserved.

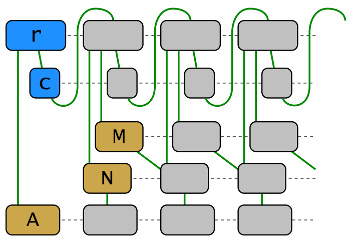
(a)
Canonical eigenclass structure
(O, .ec, ≤)



(b)
The primary structure of (a)
(O.pr, .class, ≤)



(c)
The "tight"
eigenclass completion of (b)



Ruby's additional constraints

The Ruby programming language imposes single inheritance and a single metalevel for classes. These additional conditions can be respectively written as follows.

(e~13) For every object x, the set *x.↥* is linearly ordered by $\leq$.

(e~9^{++}) Descendants of eigenclasses are eigenclasses. (Equivalently, every class is on metalevel 1.)

The conditions make some of the previous ones redundant. In particular, (e~9) ← (e~9$^+$) ← (e~9^{++}) is an obvious implication chain.

Monotonic eigenclass structure of ∈

The family of structures given by the first five axioms of canonical eigenclass structures can be regarded as the "essential mathematical model" for the core structure of object technology. This is for the following reasons:

- Simplicity. There are five simple conditions that can be stated with only a few preliminary definitions. Moreover, the structures are fully determined solely by ∈.
- Generality. The monotonicity condition seems to be predominant in object technology so that it can be regarded as an acceptable (or even desirable) restriction. Similarly, the requirement of every object having an eigenclass only means that the structures are expressed in their eigenclass completion (which is unique, up to isomorphism). General monotonic structures (in which *.ec* is partial) can be thought of as implementation-oriented refinement that records the "actuality" state of eigenclasses.
- Connection to algebraic set theory. Object membership, ∈, is formed by the composition of *.ec* and $\leq$ just

like the membership relation ε in *Zermelo-Fraenkel algebras*.[26] The following table shows the notational correspondence:

Monotonic eigenclass structure	$x \in y$	$\leftrightarrow$	$x.ec \leq y$
Zermelo-Fraenkel algebra	$x\ \varepsilon\ y$	$\leftrightarrow$	$s(x) \leq y$

The next subsection provides a concise and rather self-contained definition of monotonic eigenclass structures.

Monotonic eigenclass structure

By a *monotonic eigenclass structure* we mean a structure $\mathcal{S} = (\underline{O}, .ec, \leq, \underline{r})$ where

- $\underline{O}$ is a set of *objects*,
- $.ec$ is the *eigenclass* map $\underline{O} \rightarrow \underline{O}$, (objects from $\underline{O}.ec$ are *eigenclasses*)
- $\leq$ is the *inheritance* relation between objects,
- $\underline{r}$ is the *inheritance root*, a distinguished object.

The usual *ancestor* / *descendant* terminology is used for inheritance. Objects that are not descendants of $\underline{r}$ are *terminal(s)*. Let $.ec*$ denote the reflexive transitive closure of $.ec$. For an object x, the set $x.ec*$ (the image of $\{x\}$ under $.ec*$) is the *eigenclass chain* of x.

The structure is subject to the following axioms:

(e~1) Inheritance, $\leq$, is a partial order.	
(e~2) The eigenclass map, $.ec$, is an order-embedding of $(\underline{O}, \leq)$ into itself.	$(\leq) = (.ec) \circ (\leq) \circ (.ce)$
(e~3) Objects from eigenclass chains of terminals are minimal in inheritance.	$(\underline{O} \times \underline{T}.ec*) \cap (<) = \emptyset$
(e~4) Every eigenclass is a descendant of the inheritance root.	$\underline{O}.ec \leq \underline{r}$
(e~5) The eigenclass chain of $\underline{r}$ has no lower bound in $\leq$.	$\underline{R}.\triangledown = \emptyset$

The definitions introduced before the axioms of canonical eigenclass structures apply. See also notes to the axioms.

Decomposition of ϵ

As already shown for the Ruby core sample, any monotonic eigenclass structure can be expressed in the minimum signature $(\underline{O}, \in)$. The constituents of the original signature are obtained as follows:

$$x \leq y \quad \leftrightarrow \quad x.\in \supseteq y.\in,$$

$x.ec = y \quad \leftrightarrow \quad x.\in = y.\uparrow$.   (In addition, $x.ec = y \rightarrow x.\downarrow = y.\ni$ but "$\leftarrow$" does not hold in general.)

Moreover, $\underline{r}$ is the unique top of $\underline{O}.\in$.

Monotonic primary structure of ϵ

For the sake of completeness we introduce a generalization of canonical primary structures according to the following table.

	Special	General
$.ec$ is total	Canonical eigenclass structure	Monotonic eigenclass structure
$.ec$ is empty	Canonical primary structure	Monotonic primary structure

Note: For simplification, we introduce slight discrepancy with the more precise document [46]. The family of structures defined in the following subsection should be more correctly called *membership-based monotonic structures* since they rely on the prescription

(mp-γ) $x \in^{-k} y \quad \leftrightarrow \quad x < \underline{r}.\in^{k+i}$ and $\underline{r}.\in^{k+i} \neq \underline{r}.\in^{k+i-1}$ and $\underline{r} \in^{i} y$ for some natural i

for negative powers of ϵ ($\in^{-k}$ where $k > 0$) which we introduce later for basic structures. The correctly defined family of monotonic primary structures arises simply as a subfamily of monotonic basic structures in which $.ec = \emptyset$ (so that every object is primary).

Monotonic primary structure

By a *monotonic primary structure* of ϵ we mean a structure $(\underline{O}, \epsilon, \leq, \underline{r})$ where $\underline{O}$ is a set of *objects*, ϵ is the *membership* relation between objects, $\leq$ is the *inheritance* relation between objects, and $\underline{r}$ is the *inheritance root*, a distinguished object.

The usual terminology and notation is used for $\leq$ and ϵ. In addition, the $\leq$ and $<$ symbols are also used in the "polar" sense for relations between sets of objects, so that e.g $X < Y$ means that every object of X is less than every object of Y. Let $\underline{T} = \underline{O} \setminus \underline{O}.\epsilon.$↧ be the set of *terminal* objects and $\underline{H} = \underline{r}.\epsilon{*}$ the set of *helix* objects where $\epsilon{*}$ is the transitive closure of $(\leq) \cup (\epsilon)$. For a natural $i > 0$, let ϵ^i be the i-th composition of ϵ with itself and let ϵ^0 be equal to $\leq$.

The structure is subject to the following axioms:

(mp-γ~1) $\leq$ is a partial order.

(mp-γ~2) $(\epsilon) \odot (\leq) = (\epsilon) = (\leq) \odot (\epsilon)$.   (That is, (a) $(\epsilon) \odot (\leq) \subseteq (\epsilon)$ and (b) $(\leq) \odot (\epsilon) \subseteq (\epsilon)$.)

(mp-γ~3) The inheritance root $\underline{r}$ is the top of $\underline{O}.\epsilon$ w.r.t. $\leq$.

(mp-γ~4) Every object has a container, $\underline{O} = \underline{O}.\ni$.

(mp-γ~5) Terminal objects are minimal in $\leq$.

(mp-γ~6) If $\underline{H} \neq \underline{r}.\epsilon^i$ for every natural i then $\underline{H}$ has no lower bound in $\leq$.

(mp-γ~7) For every natural i such that $\underline{H} \neq \underline{r}.\epsilon^i$,   if $x < \underline{r}.\epsilon^{i+1}$ then $x.\ni < \underline{r}.\epsilon^i$.

Observation: For a structure $\mathcal{S} = (\underline{O}, \epsilon, \leq, \underline{r})$ the following are equivalent:

i. $\mathcal{S}$ is a canonical primary structure.
ii. $\mathcal{S}$ is a monotonic primary structure satisfying (p~4)–(p~8).

Moreover, the (p~4) condition (Metaclasses can only have classes as instances.) can only be stated for the metaclass root since other metaclasses are strict descendants of $\underline{r}.\epsilon$ and thus are subject to (mp-γ~7).

Basic structure of ϵ

In this section we introduce a common generalization of the hitherto introduced structures that can be regarded as an abstract set-theoretical model of the core structure of object technology. The generalization can be roughly characterized by the following steps:

- Remove (relax) the monotonicity condition. As already observed in the introduction, in contrast to the subsumption rule, the monotonicity condition $(\leq) \odot (\epsilon) \subseteq (\epsilon)$ is not universally satisfied when $\leq$ and ϵ are interpreted as $\subseteq$ and $\in$, respectively.
- Allow partial definition of $.ec$. Until now we have formally described structures that either had no eigenclasses (primary structures) or in which the eigenclass map was total (eigenclass structures).
- Provide a non-monotonic counterpart to $.ec$. Until now, $x.ec$, if defined, was the least container of x. The $(\geq) = (.ec) \odot (\ni)$ equality indicates that the $.ec$ map is an abstraction of the powerset operator. However, in set theory, the least set that contains a given set x as an element is $\{x\}$, the singleton of x, which is different from the powerset of a set x (except when x is empty).

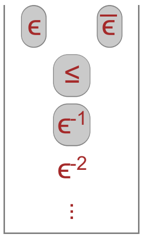
This is established by the family of *basic structures* [46].

Basic structure

By a *basic structure* of ϵ we mean a structure $\mathcal{S} = (\underline{O}, \epsilon^{(*)}, \overline{\epsilon}^{(*)}, \underline{r}, .ec, .\varepsilon\text{ȼ})$ where

- $\underline{O}$ is a set of *objects*,

⋮ ⋮
ϵ^2 $\overline{\epsilon}^2$



- $\epsilon^{(*)}$ and $\overline{\epsilon}^{(*)}$ are bi-infinite sequences of relations between objects such that (a) $(\epsilon^{i}) = (\overline{\epsilon}^{i})$ for $i \leq 0$, (b) $(\epsilon^{j+1}) = (\epsilon^{j}) \odot (\epsilon)$ and (c) $(\overline{\epsilon}^{j+1}) = (\overline{\epsilon}^{j}) \odot (\overline{\epsilon})$ for $j > 0$, and where
  - $(\epsilon) = (\epsilon^{1})$ is the *(object) membership* relation,
  - $(\overline{\epsilon}) = (\overline{\epsilon}^{1})$ is the *power membership* relation,
  - $(\leq) = (\epsilon^{0})$ is the *inheritance* relation (with $.\downarrow$ / $.\uparrow$ used for preimages / images under $\leq$),
  - (ϵ^{-1}) is the *anti-membership* relation,
- $\underline{r}$ is the *inheritance root*, a distinguished object,

and the remaining two definitory constituents are partial maps $\underline{O} \curvearrowright \underline{O}$ (i.e. functional relations between objects):

- $.ec$ is the *powerclass* map   (objects from $\underline{O}.ec$ are *powerclasses*),
- $.\varepsilon\text{ɕ}$ is the *primary singleton* map   (objects from $\underline{O}.\varepsilon\text{ɕ}$ are *primary singletons*).

Note that due conditions (a–c) the two bi-infinite sequences are given by ϵ and the left-infinite sequence $\{\overline{\epsilon}^{i} \mid i \in \mathbb{Z}, i \leq 1\}$. In particular, we can drop the "wildcard" superscript from $\epsilon^{(*)}$ in the signature. Before stating the axioms some preliminary definitions need to be introduced.

For every integer i, $\backepsilon^{i}$ (resp. $\overline{\backepsilon}^{i}$) denotes the inverse of ϵ^{i} (resp. of $\overline{\epsilon}^{i}$). For a natural i, let $.ec(i)$ be the i-th composition of $.ec$ with itself, with $.ec(0)$ being the identity on $\underline{O}$. Let $.ec(-i)$ be the inverse of $.ec(i)$. Let $\underline{T} = \underline{O} \setminus \underline{O}.\epsilon.\downarrow$ be the set of *terminal* objects (or *terminals*). The *metalevel index*, $x.mli$, of an object x is defined by

$$x.mli = sup\ \{\, i \mid x\ \epsilon^{1-i}\ \underline{r},\ i \in \mathbb{N}\,\}.$$

Finally, the definition of the *rank* function, $.d$, assumes a fixed limit ordinal $\underline{\varpi}$ in the context. The rank of an object x is then recursively defined by

$x.d = \underline{\varpi}$ if x is non-well-founded in ϵ,
$x.d = \underline{\varpi} \wedge (sup\ \{a.d + 1 \mid a\ \epsilon\ x\} \vee sup\ \{a.mli + i\text{-}j \mid a \in x.\backepsilon^{i}.\backepsilon^{-j},\ i,j \in \mathbb{N}\})$ if x is well-founded in ϵ.

(We use $\alpha \wedge \beta$ (resp. $\alpha \vee \beta$) to refer to the minimum (resp. maximum) of ordinal numbers α and β. To be correct, the prescription requires finiteness of $a.mli$ which is asserted by (b~10).)

The structure is then subject to the following axioms:

The axioms

(b~1) $(\overline{\epsilon}) \subseteq (\epsilon)$.
(b~2) $(\overline{\epsilon}^{i}) \odot (\overline{\epsilon}^{j}) \subseteq (\overline{\epsilon}^{i+j})$ for every integer i, j.
(b~3) $(\epsilon) \odot (\epsilon^{i}) \subseteq (\epsilon^{1+i})$ for every integer i.
(b~4) $(\overline{\epsilon}^{i}) \cap (\backepsilon^{-i}) = .ec(i)$ for every integer i.
(b~5) The inheritance root $\underline{r}$ is the top of $\underline{O}.\epsilon$ w.r.t. $\leq$.
(b~6) Every object x has a container, $x.\epsilon \neq \emptyset$.
(b~7) For every object x from $\underline{T} \cup \underline{O}.\varepsilon\text{ɕ}$ and every natural i, (a) $x.\backepsilon^{-i} = \{x\}.ec(i)$, (b) $x.\backepsilon^{-i}.\epsilon = x.\backepsilon^{-i}.\overline{\epsilon}$.
(b~8) If $x.\varepsilon\text{ɕ} = y$ then: (a) $\{x\} = y.\backepsilon$, (b) $x.\epsilon^{i} = y.\epsilon^{i-1}$ for every $i \leq 1$, (c) $(x,y) \notin (\overline{\epsilon})$.
(b~9) Reserved for the non-member union map.
(b~10) For every object x, the metalevel index $x.mli$ is finite.
(b~11) For every object x, $x.d = \underline{\varpi} \ \rightarrow\ x.\epsilon = x.\overline{\epsilon}$. (That is, every unbounded object is a power member.)

Observations:

1. For a suitable choice of (i,j) in (b~2) we obtain transitivity of $\leq$, subsumption of $\overline{\epsilon}$, and the monotonicity of $\overline{\epsilon}$.
2. For $i = 0$ in (b~3) we obtain the subsumption of ϵ: $(\epsilon) \odot (\leq) \subseteq (\epsilon)$. In contrast, monotonicity is not asserted.
3. For $i = 0$ in (b~4) we obtain the antisymmetry and reflexivity of $\leq$ so that $\leq$ is a partial order on $\underline{O}$.
4. (b~5) asserts that $\underline{O} = \underline{T} \uplus \underline{r}.\downarrow$. Since $\underline{r}\ \epsilon\ x \leq \underline{r}$ for some object x it follows by subsumption of ϵ that $\underline{r}\ \epsilon\ \underline{r}$. Since $\underline{r}$ is non-well-founded in ϵ it follows by (b~11) that $\underline{r}\ \overline{\epsilon}\ \underline{r}$. Consequently, for every non-terminal x, $x\ \overline{\epsilon}\ \underline{r}$ by monotonicity of $\overline{\epsilon}$ and $x\ \epsilon\ \underline{r}$ since $(\overline{\epsilon}) \subseteq (\epsilon)$.
5. As a particular consequence of (b~6) and (b~7)(b) we obtain that $x\ \overline{\epsilon}\ \underline{r}$ for every terminal x. It follows that $\underline{r}$ is a universal (power) container: $\underline{O} = \underline{r}.\overline{\backepsilon} = \underline{r}.\backepsilon$.
6. As a particular consequence of (b~7)(a) we obtain that every object from $\underline{T}.ec*$ is minimal in $\leq$ (cf. (e~3)).

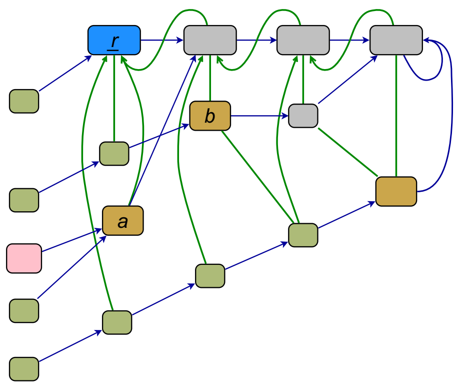
Rank

If every object is ∊-*grounded*, i.e. connected with a terminal object via ∊ (equivalently, $\underline{T}.∊^* = \underline{O}$) then the prescription for the rank function is simplified as follows. Let W be the set of objects that are well-founded in ∊. Then

$x.d = \underline{\varpi}$ if $x \notin W$,
$x.d = \underline{\varpi} \wedge \sup\{a.d + 1 \mid a ∊ x\}$ if $x \in W$.

That is, $x.d = \underline{\varpi}$ for all objects x that are either non-well-founded in ∊ or are well-founded and their rank in the well-founded relation $(W,∊)$ is greater or equal to $\underline{\varpi}$. For the remaining objects x, let $x.d$ be their rank in $(W,∊)$.

The following diagram shows an ad-hoc completion (∗) making each object of the original structure ∊-grounded. Added objects are displayed in khaki color. See [46] for the precise definition.



Notes and observations:

1. (∗) As indicated by the case of the a object, new member chains are added even for objects from $\underline{T}.∊^*$ so that the term "completion" is not quite adequate.
2. It is asserted that $x.mli \le x.d$ for every object x. However, even memberless objects can have their rank strictly higher than the metalevel index. This is the case of the b object ($2 = b.mli < b.d = 3$).

Another simplification arises in the case $\underline{\varpi} = \omega$. That is, the "default" value of $\underline{\varpi}$ is the first infinite ordinal so that the rank of an object is either finite of equal to ω. In this case, $.d$ can be defined by

- $x.d = \sup\{a.mli + i\text{-}j \mid a \in x.\ni^i.\ni^{-j},\ i,j \in \mathbb{N}\}$.

Bounded membership

The rank function $.d$ is used for distinction between two kinds of objects:

- Objects x such that $x.d < \underline{\varpi}$ are *bounded*.
- Objects x such that $x.d = \underline{\varpi}$ are *unbounded*.

$$\begin{tikzcd} & ∊ \arrow[dl, dash] \arrow[dr, dash] & \\ \epsilon \arrow[dr, dash] & & \overline{∊} \arrow[dl, dash] \\ & \overline{\epsilon} & \end{tikzcd}$$

The *bounded membership* relation ϵ is then the domain-restriction of ∊ to bounded objects, i.e.

- $x \,\epsilon\, y$ iff $x ∊ y$ and x is bounded.

The set of bounded objects can be referred to by $\underline{O}.\ni$ or by $\underline{r}.\ni$. Every bounded object is well-founded in ∊ (i.e. ϵ is a well-founded relation) but not necessarily vice versa. We also introduce the $\overline{\epsilon}$ symbol for $(\overline{∊}) \cap (\epsilon)$ (the *bounded power membership*).

Note that axioms (b~1) and (b~11) can be stated as a single condition $(∊) = (\epsilon) \cup (\overline{∊})$. This establishes an inclusion lattice between ∊, ϵ, $\overline{∊}$ and $\overline{\epsilon}$ according to the above diagram on the right.

Singletons

Singletons are the objects from $(\underline{T}.ec \cup \underline{O}.\varepsilon\varsigma).ec^*$ (where $.ec^*$ is the reflexive transitive closure of $.ec$). The range-restriction of ∊ to singletons is denoted $.\varepsilon c$ and called the *singleton map*. Axioms (b~7)(a) and (b~8)(a) assert that $.\varepsilon c$ is a partial map between objects, so that

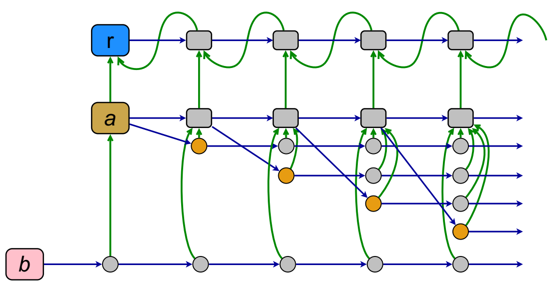
$x.\varepsilon c = y \;\leftrightarrow\; \{x\} = y.\ni$ and y is a singleton.

Another consequence of the axioms is $.\varepsilon c.\varepsilon c = .\varepsilon c.ec$ – powerclasses of singletons are singletons. The primary

singleton map $.\varepsilon\not{c}$ used in the axiomatization equals the difference $(.\varepsilon c) \setminus (.ec)$.
The following table shows similarities and differences between $.ec$ and $.\varepsilon c$.

	y is the **powerclass** of x, $x.ec = y$	y is the **singleton** of x, $x.\varepsilon c = y$
Members of y	$x.\updownarrow = y.\ni$	$\{x\} = y.\ni$
Ancestors of y	$x.\overline{\in} = y.\updownarrow$	$x.\in = y.\updownarrow$
Metalevel index increment ($y.mli - x.mli$)	1	1
Rank increment ($y.d - x.d$) on $\underline{O}.\ni$	1	1
x (or y) can be unbounded	YES	NO

Metaobject structure



The diagram on the right shows a basic structure that is *metaobject complete* – the powerclass map $.ec$ (shown by horizontal blue arrows) is total and the singleton map $.\varepsilon c$ (shown by blue arrows pointing to a circle which indicates a singleton) is defined on the set $\underline{O}.\ni$ of bounded objects. This subfamily of basic structures can be axiomatized as *metaobject structures* in the signature $(\underline{O}, \leq, \underline{r}, .ec, .\varepsilon c)$. The membership constituents of a basic structure are obtained as follows:

$(\in) = (.\varepsilon c) \odot (\leq)$ (*bounded membership*),
$(\overline{\in}) = (.ec) \odot (\leq)$ (*power membership*),
$(\in) = (\in) \cup (\overline{\in})$ ((*object*) *membership*),
$(\in^{-k}) = (\leq) \odot .ec(-k)$ (*anti-membership* and its powers, $k > 0$).

A metaobject structure is subject to the following axioms:

(mo~1)–(mo~5)	The same as axioms (e~1)–(e~5) of monotonic eigenclass structures.
(mo~6)	The singleton map, $.\varepsilon c$, is injective.
(mo~7)	Objects from $\underline{O}.\varepsilon c.ec*$ are minimal in $\leq$.
(mo~8)	For every objects x, y such that $x.\varepsilon c$ is defined, $x.\varepsilon c \leq y.ec \quad \leftrightarrow \quad x \leq y$.
(mo~9)	For every object x, $x.\varepsilon c$ is defined $\leftrightarrow$ $x.d < \underline{\varpi}$.

The rank function $.d$ is defined by the same prescription as in basic structures. See [46] for details.

Monotonic structures

$$\in \quad \leq \quad \in^{-1} \quad \in^{-2} \quad \vdots$$

Monotonic basic structures are such that $(\overline{\in}) = (\in)$. They can be axiomatized in the signature $(\underline{O}, \in^{(*)}, \underline{r}, .ec)$ where $\in^{(*)}$ is a left-infinite (or down-infinite) sequence according to the diagram on the right. The axioms are as follows (using the terminology and notation established for basic structures):

(m~1) $(\in^{i}) \odot (\in^{j}) \subseteq (\in^{i+j})$ for every integer i, j.
(m~2) $(\in^{i}) \cap (\ni^{-i}) = .ec(i)$ for every integer i.
(m~3) $\underline{O}.\in \leq \underline{r}$.
(m~4) $\underline{O} = \underline{O}.\ni$.
(m~5) For every object x from $\underline{T}$ and every natural i, $\{x\}.ec(i) = x.\ni^{-i}$.
(m~6) For every object x, the metalevel index $x.mli$, defined as $sup\ \{\, i \mid x \in^{1-i} \underline{r},\ i \in \mathbb{N} \,\}$, is finite.

Structures that have been introduced before the basic structures in this document can be considered to form subfamilies of monotonic basic structures:

- Monotonic eigenclass structures are definitionally equivalent to monotonic basic structures that are powerclass complete.
- Monotonic primary structures (see the note about terminological imprecision before the subsection) are

obtained from monotonic basic structures by imposing the following condition to negative powers of ϵ: For every natural $k > 0$,

$x \epsilon^{-k} y$ iff there is a natural i such that (a) $\underline{r}\, \epsilon^{i}\, y$, (b) $x < \underline{r}.\epsilon^{i+k}$, and (c) $\underline{r}.\epsilon^{i+k} \neq \underline{r}.\epsilon^{i+k-1}$.

This yields $.ec = \emptyset$ as a particular consequence. See [46] for details.

The union map

The (b~9) label is reserved for the axiomatization of the *non-member union map*, $.\text{ɷ}$. This is considered to be the first candidate for a possible expansion of basic structures. The $.\text{ɷ}$ map forms the explicit part of $.\cup$, the *union map*, which is a partial map between objects that is an abstraction of the least anti-container and thus of set union. The $.\cup$ map is obtained from $.\text{ɷ}$ by

$$(.\cup) \;=\; (.\text{ɷ}) \uplus ((\epsilon^{-1}) \cap (\ni)).$$

Axioms of ("non-expanded") basic structures assert that $(\epsilon^{-1}) \cap (\ni)$ is a partial map. Moreover, the inverses of $.ec$ and $.\varepsilon c$ are distinguished submaps. The presumed axiomatization of $.\text{ɷ}$ is shown below together with the similar axiom (b~8).

(b~8) If $x.\varepsilon\text{ɕ} = y$ then: (a) $\{x\} = y.\ni$, (b) $x.\epsilon^{i} = y.\epsilon^{i-1}$ for every $i \leq 1$, (c) $(x,y) \notin (\overline{\epsilon})$.

(b~9) If $x = y.\text{ɷ}$ then: (a_1) $x.\overline{\ni} = y.\ni.\overline{\ni}$, ($a_2$) $x.\ni = y.\ni^{2}$, (b) $x.\epsilon^{i} = y.\epsilon^{i-1}$ for every $i \leq 0$, (c) $(x,y) \notin (\epsilon)$.

Unfortunately, the $.\text{ɷ}$ map also requires an adjustment to the definition of the rank function $.d$ (see [46] for details).

Complete structure of ϵ

A basic structure $\mathcal{S} = (\underline{O}, \epsilon, \overline{\epsilon}^{(*)}, \underline{r}, .ec, .\varepsilon\text{ɕ})$ is said to be *complete* if it satisfies the following conditions:

(A) $\mathcal{S}$ is *extensionally consistent*: For every objects x, y, $x \leq y \;\leftrightarrow\; x = y$ or $\emptyset \neq x.\backepsilon \subseteq y.\backepsilon$.

(B) $\mathcal{S}$ is metaobject complete: (a) $\mathcal{S}$ is powerclass complete and (b) $\mathcal{S}$ is singleton complete.

(C) $\mathcal{S}$ is *extensionally complete*: For every subset X of $\underline{O}.\backepsilon$ there is an object x such that $x.\backepsilon = X$.

(D) $\mathcal{S}$ is ϵ*-ranked*: For every object x, $r_{\epsilon}(x)$ (the ϵ-rank of x) equals $x.d$.

We also say that $\mathcal{S}$ is a *complete structure* (of ϵ). Note that due (B), $\mathcal{S}$ can be considered a special case of a metaobject structure. Every complete structure $\mathcal{S}$ is uniquely given by the bounded membership ϵ according to the following table:

Inheritance	$x \leq y$	$\leftrightarrow$	$x = y$ or $\emptyset \neq x.\backepsilon \subseteq y.\backepsilon$
Inheritance root	$\underline{r}.\backepsilon$	$=$	$\underline{O}.\backepsilon$
Bounded inheritance	$x\, \epsilon^{0}\, y$	$\leftrightarrow$	$x \leq y$ and $x \in \underline{O}.\backepsilon$
Singleton map	$x.\varepsilon c = y$	$\leftrightarrow$	$\{x\} = y.\backepsilon$
Powerclass map	$x.ec = y$	$\leftrightarrow$	$x.\backepsilon^{0} = y.\backepsilon$
Power membership	$x\, \overline{\epsilon}\, y$	$\leftrightarrow$	$x.\backepsilon^{0} \subseteq y.\backepsilon$
Object membership	$x \epsilon y$	$\leftrightarrow$	$x\, \epsilon\, y$ or $x\, \overline{\epsilon}\, y$

The *bounded inheritance* relation is the zeroth power of ϵ, similarly to the $(\leq) = (\epsilon^{0})$ correspondence. By definition, $x.\backepsilon^{0} = x.\downarrow \cap \underline{O}.\backepsilon$. There are other simplifications via ϵ:

Terminal objects	$\underline{T}$	$=$	$\underline{O} \setminus \underline{O}.\epsilon$
Metalevel index	$x.mli$	$=$	$min\, \{i \mid x \in \underline{T}.\epsilon^{i},\, i \in \mathbb{N}\}$
Rank	$x.d$	$=$	$sup\, \{a.d + 1 \mid a\, \epsilon\, x\}$

The equalities for $.mli$ and $.d$ can be described as follows.

- $\mathcal{S}$ is ϵ-*levelled*, that is, every non-terminal object has a bounded member with a lesser metalevel index.
- $\mathcal{S}$ is ϵ-*ranked*, that is, $x.d$ equals the ϵ-rank of x for every object x.

Moreover, there is a simple and transparent axiomatization of complete structures via $(\underline{O}, \epsilon)$. Such an axiomatization can be provided in a generalization for arbitrary rank of ϵ, as shown in the next subsection. (Realize that $\underline{r}.d$ equals by definition a fixed limit ordinal $\underline{\varpi}$. Since $\underline{r}.d$ is also the ϵ-rank of $\underline{r}$ and $\underline{r}.\ni = \underline{O}.\ni$ it follows that the rank of $(\underline{O}, \epsilon)$ equals $\underline{\varpi}+1$.)

Superstructure

By an *(abstract) superstructure* we mean a structure $(\underline{V}, \epsilon)$ (where $\underline{V}$ is the set of *objects*, and ϵ is a relation between objects) such that the following conditions hold.

(1) ϵ is well-founded. For a subset X of $\underline{V}$ let us denote $r(X)$ the ϵ-rank of X.

(2) For every non-empty set X of objects such that $r(X) < r(\underline{V})$ there is a unique object x such that $x.\ni = X$.

The existence and uniqueness of the x object in (2) can be stated separately:

(2a) For every set X of objects such that $r(X) < r(\underline{V})$ there is an object x such that $x.\ni = X$.

(2b) For every objects x, y from $\underline{V}.\epsilon$, if $x.\ni = y.\ni$ then $x = y$. (Weak extensionality of ϵ)

It can be shown that up to isomorphism, every non-empty superstructure $(\underline{V}, \epsilon)$ is uniquely given by the pair (α, κ) of non-zero ordinal numbers where α is the ϵ-rank of $\underline{V}$ and κ is the cardinality of the *ground stage* (which is the set $\underline{V} \setminus \underline{V}.\epsilon$ of objects with zero rank, i.e. the set of terminal objects).

For convenience, we let the term α-*superstructure* mean a superstructure $(\underline{V}, \epsilon)$ whose ϵ-rank equals α. The axiomatization of complete structures via ϵ can be then expressed as follows:

$$\boxed{(\underline{O}, \epsilon, \overline{\epsilon}^{(*)}, \underline{r}, .ec, .\varepsilon\underline{c}) \text{ is a complete structure of } \epsilon \quad \leftrightarrow \quad (\underline{O}, \epsilon) \text{ is an } (\underline{\varpi}+1)\text{-superstructure.}}$$

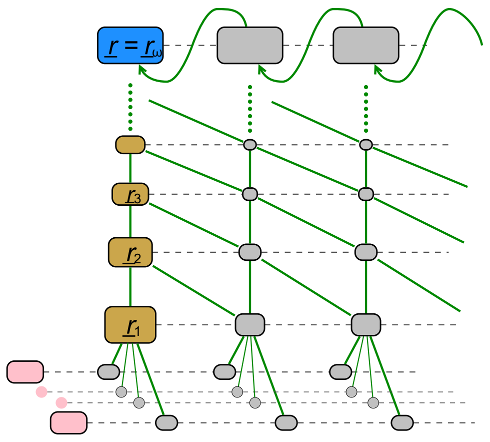
That is, the bounded membership ϵ in a complete structure forms an $(\underline{\varpi}+1)$-superstructure, and, conversely, an $(\underline{\varpi}+1)$-superstructure $(\underline{O}, \epsilon)$ induces a complete structure $\mathcal{S}$ according to the table above. Bounded membership in $\mathcal{S}$ coincides with the original ϵ relation.

Stages

Any superstructure can be thought to be built in *stages*. For an ordinal number i, the i-th stage, $\underline{V}_i$, consists of objects whose rank is strictly less than i. In the case of an $(\underline{\varpi}+1)$-superstructure $(\underline{O}, \epsilon)$ we obtain the following sequence:

- $\underline{V}_0 = \varnothing$,
- $\underline{V}_1 = \underline{T} = \underline{O} \setminus \underline{O}.\epsilon$ (the ground stage),
- $\underline{V}_{i+1} = \{\, x \mid x.\ni \subseteq \underline{V}_i \,\}$,
- $\underline{V}_i = \bigcup \{\, \underline{V}_j \mid j < i \,\}$ for a limit ordinal i,
- $\underline{V}_{\underline{\varpi}} = \underline{O}.\ni = \underline{r}.\ni$ (bounded objects),
- $\underline{V}_{\underline{\varpi}+1} = \underline{O}$.

Except for $\underline{V}_0$ and $\underline{V}_{\underline{\varpi}+1}$ each stage $\underline{V}_i$ is an ϵ-preimage of a unique *stage object*, $\underline{r}_i$ (see the diagram on the right, note that singletons not from $\underline{T}.ec*$ are not shown).



Kinds of superstructures

With some adjustments, the definitional extension of an $(\underline{\varpi}+1)$-superstructure $(\underline{O}, \epsilon)$ as specified above can be applied to arbitrary superstructures $(\underline{V}, \epsilon)$. Apart from the degenerate cases of empty ϵ, the adjustments to be made are the following:

- Instead of referring to $\underline{r}$ as the unique object whose existence is guaranteed, introduce a predicate "x *being an inheritance root*": x is such that $x.\ni = \underline{V}.\ni$.

- Add the $\emptyset \neq x.\ni^0$ condition in the definitions of $.ec$ and $\overline{\epsilon}$.

We can then classify superstructures according to the properties of their definitional extensions:

(Let ϖ be a limit ordinal and α an arbitrary ordinal.)	ZFC	**MK**		Degenerate cases	
	$\underline{V} = \underline{V}_\varpi$	$\underline{V} = \underline{V}_{\varpi+1}$	$\underline{V} = \underline{V}_{\alpha+2}$	$\underline{V} = \underline{V}_1$	$\underline{V} = \underline{V}_0$
Rank of $\underline{V}$	ϖ	$\varpi + 1$	$\alpha + 2$	1	0
Bounded objects $\underline{V}.\ni$	$\underline{V}$	$\underline{V}_\varpi$	$\underline{V}_{\alpha+1}$	$\emptyset$	$\emptyset$
Existence of the (an) inheritance root $\underline{r}$ ($\underline{r}.\ni = \underline{V}.\ni$)	NO	YES	YES	NO, unless $\underline{V} = \{\underline{r}\}$	NO
Powerclass map $.ec$ is total	YES	YES	NO		YES
Singleton map $.\varepsilon c$ is total	YES	NO	NO	NO	YES
Boundedness preserved by $.ec$ / $.\varepsilon c$	YES	YES	NO	YES	YES

The ZFC and MK labels indicate a connection to set theory. For an inaccessible cardinal ϖ, a superstructure $(\underline{V}, \epsilon)$ whose ground stage contains exactly one object (i.e. ϵ is extensional) is a model of set theory:

- ZFC (Zermelo-Frankel set theory with the Axiom of Choice) if the rank of ϵ equals ϖ,
- MK (Morse-Kelley set theory) if the rank of ϵ equals $\varpi + 1$.

Complete structures belong to the MK-group.

Note: In [27], the singleton map $.\varepsilon c$ is total – unbounded objects have their singleton constantly set to $\underline{r}$ (considering $(\underline{V}, \epsilon)$ as a model of MK).

Completion

Every basic structure can be faithfully embedded into a complete structure. That is, if $\mathcal{S}_0 = (\underline{O}_0, \epsilon, \overline{\epsilon}^{(*)}, \underline{r}, .ec, .\varepsilon c)$ is a basic structure then there is a complete structure $\mathcal{V} = (\underline{V}, \epsilon, \overline{\epsilon}^{(*)}, \underline{r}_\varpi, .ec, .\varepsilon c)$ and an embedding map $.v$ from $\underline{O}_0$ to $\underline{V}$ such that the following conditions are satisfied:

i. $.v$ is embedding w.r.t. ϵ^i and $\overline{\epsilon}^i$ for every integer i,
ii. $\underline{r}.v = \underline{r}_\varpi$,
iii. $.v$ is embedding w.r.t. $.ec$ and $.\varepsilon c$,
iv. $.v$ preserves being a primary object, i.e. $\underline{O}_0.pr.v \subseteq \underline{V}.pr$,
v. $.v$ preserves the rank, i.e. $x.d = x.v.d$ for every $x \in \underline{O}_0$.

Note: If R denotes a relation both in $\mathcal{S}_0$ and $\mathcal{V}$ then $.v$ is an *embedding* w.r.t. R if the following equivalence holds for every objects x, y from $\underline{O}_0$:

$$(x,y) \in R \quad\leftrightarrow\quad (x.v, y.v) \in R.$$

The embedding is established in the following steps:

1. Rank pre-completion. This step can be omitted for $\varpi = \omega$. Otherwise, attach a set X of ϖ new members to each primary non-well-founded object x that is not ϵ-ranked. The members are attached in such a way that $(X, \epsilon, \leq)$ is isomorphic to $(\varpi, \in, \subseteq)$.
2. Powerclass completion. Append an infinite powerclass chain to each object for which the powerclass is not defined. The resulting structure is powerclass complete.
3. Singleton completion. Append an infinite singleton chain to each bounded object for which the singleton is not defined. Technically, this can be done in two steps by first adding just the missing primary singletons and subsequently performing the powerclass completion.
4. Extensional pre-completion. $(*)$ To every object x that is not extensionally consistent, i.e. for which there exists y such that the following equivalence is not satisfied,

   $$x \leq y \quad\leftrightarrow\quad x = y \text{ or } \emptyset \neq x.\ni \subseteq y.\ni,$$

   attach two powerclass chains each of which starts in a terminal object. The resulting structure is *pre-complete*, that is,

   - extensionally consistent (the above equivalence holds for every objects x, y),
   - powerclass consistent (that is, if x is *powerclass-like* then x is a powerclass – see [46] for details),
   - powerclass complete,
   - singleton complete (every bounded object x has a singleton $x.\varepsilon c$), and
   - ϵ-ranked.

In particular, the structure is fully determined by bounded membership ϵ according to the same prescription as with complete structures.

Note: $(*)$ A simplified description is provided here, see [46] for the detailed description.

5. Cumulative embedding into an $(\varpi+1)$-superstructure. Let $\mathcal{S} = (\underline{O}, \epsilon)$ be the pre-complete structure to be embedded. Choose an $(\varpi+1)$-superstructure $\mathcal{V} = (\underline{V}, \epsilon)$ so that its ground stage $\underline{V}_1$ has the same cardinality as the set $\underline{T}$ of terminal objects of $\mathcal{S}$. Then the requested embedding map $.v$ is obtained as a limit of a transfinite sequence

   $.v_0, .v_1, \ldots, .v_\varpi = .v$

   of maps from $\underline{O}$ to $\underline{V}$ defined as follows:

   I. The restriction of $.v_i$ to terminals is for every i identical and forms a bijection between $\underline{T}$ and $\underline{V}_1$.
   II. The restriction of $.v_i$ to the set $\underline{O}.\epsilon$ of non-terminal objects x is recursively defined by

   $x.v_0.\ni \;=\; x.\overline{\ni}.v_0.ec.\ni \cup x.\ni.v_0,$

   $x.v_i.\ni \;=\; x.\overline{\ni}.v_{i-1}.ec.\ni \cup x.\ni.v_0$ if i is a successor ordinal,

   $x.v_i.\ni \;=\; \bigcup\{\, x.v_k.\ni \mid k < i \,\}$ if i is a limit ordinal.

   Note that the definition of $.v_0$ is by the well-founded recursion on $(\underline{O}, \epsilon)$, whereas the definition of $.v_i$ for $i > 0$ uses transfinite recursion over i.

Powerclass completion

Powerclass completion deserves particular attention since it is actually implemented in Ruby. For canonical primary structures, the completion has already been described, see the (ec~3) condition. (Since $(\epsilon) = (\overline{\epsilon})$, the term "eigenclass" is used for "powerclass".)

In general, for a basic structure $\mathcal{S}_0 = (\underline{O}_0, \epsilon, \overline{\epsilon}^{(*)}, \ldots)$, its powerclass completion $\mathcal{S} = (\underline{O}, \epsilon, \overline{\epsilon}^{(*)}, \underline{r}, .ec, .\varepsilon\mathfrak{c})$ is created in the following steps:

1. Prolongate powerclass chains to infinity, that is, extend $(\underline{O}_0, .ec)$ to $(\underline{O}, .ec)$ so that
   - $.ec$ is an injective well-founded map on $\underline{O}$ and $\underline{O}_0 \setminus \underline{O}_0.ec = \underline{O} \setminus \underline{O}.ec$ (new objects are powerclasses).
2. Extend membership and power membership powers to new objects.
   For every primary objects a, b and every natural i, j such that at least one of $a.ec(i)$ or $b.ec(j)$ is new,

   $a.ec(i) \epsilon b.ec(j)$ iff $a \,\overline{\epsilon}^{1+i-j}\, b$,

   $a.ec(i) \,\overline{\epsilon}^{k}\, b.ec(j)$ iff $a \,\overline{\epsilon}^{k+i-j}\, b$ for every integer k.

Representation by sets

As already foreshadowed in the context of superstructures, the main correspondence between objects and well-founded sets can be expressed by

$$\epsilon \quad \leftrightarrow \quad \in \,.$$

That is, bounded membership, ϵ, is an abstraction of set membership, $\in$. Assuming powerclass completeness for simplicity, the set-theoretic interpretation of the 3 main constituents of the core structure of object technology can be informally described via the following correspondences:

$\leq \leftrightarrow \subseteq$,	that is, inheritance is an abstraction of set inclusion,
$x.ec \leftrightarrow \underline{r} \cap \mathbb{P}(x)$,	that is, the powerclass map is an abstraction of relativized powerset operator,
$\epsilon \leftrightarrow \in$ plus $.ec.\updownarrow$	that is, object membership is like set membership augmented with the inclusion of the powerclass.

The von Neumann universe

Let $\mathbb{V}$ be the von Neumann universe of all well-founded sets. Recall that $\mathbb{V}$ is obtained from the empty set by iterative application of the powerset operator $\mathbb{P}$. For every ordinal α,

$\mathbb{V}_\alpha = \bigcup\{\, \mathbb{P}(\mathbb{V}_\beta) \mid \beta < \alpha \,\}$, that is, $\mathbb{V}_0 = \emptyset$,

$\mathbb{V}_\alpha = \mathbb{P}(\mathbb{V}_{\alpha-1})$ if α is a successor ordinal,
$\mathbb{V}_\alpha = \bigcup \{ \mathbb{V}_\beta \mid \beta < \alpha \}$ if α is a limit ordinal,

$\mathbb{V} = \bigcup \{ \mathbb{V}_\alpha \mid \alpha \in \mathrm{On} \}$. ($\alpha \in \mathrm{On}$ means that α is an ordinal.)

The *rank* function $r()$ on $\mathbb{V}$ is defined by

- $r(x) = \alpha \quad \leftrightarrow \quad x \in \mathbb{V}_{\alpha+1} \setminus \mathbb{V}_\alpha$.

For every ordinal α, $\mathbb{V}_\alpha$ is a set whose members are exactly the sets x such that $r(x) < \alpha$. The axiom of foundation says that every set belongs to $\mathbb{V}$.

$(\varpi+1)$-superstructure in $\in$

The $(\varpi+1)$-superstructure $\mathcal{V} = (\underline{V}, \epsilon)$ referred to in the last step of the completion of object membership can be chosen as a restriction of the set membership structure in the von Neumann universe. Given the requested cardinality κ of the ground stage, one can proceed as follows.

1. Determine the ground stage. Choose $\underline{V}_1$ to be a set such that
   a. $\underline{V}_1 \subseteq \mathbb{V}_{\alpha+1} \setminus \mathbb{V}_\alpha$ for some ordinal α,
   b. every element of $\underline{V}_1$ is a singleton set (so that $\underline{V}_1$ is an antichain w.r.t. $\subseteq$),
   c. the cardinality of $\underline{V}_1$ equals κ.

   Since for every ordinal i, the set $\mathbb{V}_{i+1} \setminus \mathbb{V}_i$ has cardinality at least i, we can put $\alpha = \kappa+1$.
2. Cumulate the remaining stages. Similarly as with the von Neumann hierarchy, construct a transfinite sequence of stages up to the $(\varpi+1)$-th stage.

   $\underline{V}_0 = \varnothing$,
   $\underline{V}_i = \underline{V}_1 \cup (\mathbb{P}(\underline{V}_{i-1}) \setminus \{\varnothing\})$ if i is a successor ordinal, (in contrast, $\mathbb{V}_i = \mathbb{P}(\mathbb{V}_{i-1})$)
   $\underline{V}_i = \bigcup \{ \underline{V}_j \mid j < i \}$ if i is a limit ordinal,
   $\underline{V} = \underline{V}_{\varpi+1}$.

As a result, $(\underline{V}, \in)$ is an $(\varpi+1)$-superstructure such that for every x, y from $\underline{V}$,

- $r(x) = \alpha + x.d$,
- $x \leq y \quad \leftrightarrow \quad x \subseteq y$.

Basic structure as a set

Since every basic structure can be embedded into an $(\varpi+1)$-superstructure which in turn can be embedded into the von Neumann universe, it follows that every basic structure $\mathcal{S}$ can be represented by a well-founded set $\underline{O}$. The following table shows how the main constituents of $\mathcal{S}$ can be expressed in set-theoretic terms. (Recall that $\mathbb{P}_1(x)$ is the set of singleton subsets of x.)

Terminal objects	$\underline{T}$	=	$\underline{O} \cap \mathbb{P}_1(\bigcup \underline{O} \setminus \underline{O})$
Inheritance root	$\underline{r}$	=	$\bigcup \underline{O} \setminus \bigcup \underline{T}$
Complete extension	$\underline{V}$	=	$\underline{r} \cup (\mathbb{P}(\underline{r}) \setminus \{\varnothing\})$
For every x, y from $\underline{O}$:			
Bounded membership	$x \epsilon y$	$\leftrightarrow$	$x \in y$
Inheritance	$x \leq y$	$\leftrightarrow$	$x \subseteq y$
Singleton map	$x.\varepsilon c = y$	$\leftrightarrow$	$\{x\} = y$
Powerclass map	$x.ec = y$	$\leftrightarrow$	$\underline{r} \cap \mathbb{P}(x) = y$
Power membership	$x \bar{\epsilon} y$	$\leftrightarrow$	$\underline{r} \cap \mathbb{P}(x) \subseteq y$
Object membership	$x \epsilon y$	$\leftrightarrow$	$\underline{r} \cap \mathbb{P}(x) \subseteq y$ or $x \in y$

The $\underline{V}$ set represents a complete basic structure $\mathcal{V}$ that is a faithful extension of $\mathcal{S}$.

Specializations of ϵ

This section describes how the canonical structure of object membership can be applied to object models of particular languages. We focus on 8 class-based programming languages Ruby, Python, Smalltalk-80, Java, Scala, CLOS, Objective-C, and Perl which have already been considered for the sample structure of the instance-of relation. Prototypal languages (JavaScript), ontology languages (RDFS) and the Dylan programming language (as the only representant of non-monotonic membership) are discussed separately. Similarly, the support of actual eigenclasses is described in another section.

In a particular programming language, object membership appears in a specialized form. Such a "specialization" can be obtained by adjustments to the canonical structure of ϵ. In most cases (5 of 8), it is necessary to refine the structure by additional constituents so that the minimum signature $(\underline{O}, \epsilon)$ needs to be extended.

In Ruby

We have already described the Ruby's additional constraints for the canonical structure: single inheritance and single explicit metaclass ($\underline{C} \cap \underline{c}.\!\downarrow = \{\underline{c}\}$). As of Ruby 1.9 (and newer), there are 4 helix classes:

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

As a consequence of single inheritance, the structure is determined by superclass and eigenclass links, corresponding to the `superclass` and `singleton_class` introspection methods, respectively.

However, this is only the (canonical) reduct of what is understood by object membership in Ruby. The "full" membership, $\boldsymbol{\epsilon}$, results from a finer structure which takes module inclusion into account. *Modules* are terminal instances of the `Module` class, i.e. modules are `Module`s that are not `Class`es. The additional structure is given by the *own-includer-of* relation between `Module`s (classes, eigenclasses, modules) and modules. If $\underline{\mathrm{M}}$ denotes the reflexive closure (i.e. $\underline{\mathrm{M}}$ is *self-or-own-includer-of*) then the $\boldsymbol{\epsilon}$ relation is given by

$(\boldsymbol{\epsilon}) = (\epsilon) \odot \underline{\mathrm{M}}$.

This extended membership corresponds to the `is_a?` introspection method (aliased by `kind_of?` and, in the inverse, also by `Class#===`). You can check that previously we stated the `is_a?`$\leftrightarrow\epsilon$ correspondence just for the provided sample structures. Similarly, the "canonical" inheritance, $\leq$, is extended to $\boldsymbol{\leq}$ by

$(\boldsymbol{\leq}) = (\leq) \odot \underline{\mathrm{M}}$

which, in the restriction to `Module`s, corresponds to the `<=` introspection method. This extended inheritance is a multiple inheritance. In addition, since Ruby supports dynamic module inclusion, $\boldsymbol{\leq}$ can have anomalies (as to transitivity and/or antisymmetry), the problem known as the double/dynamic inclusion problem. [18]

In Python

Assuming consistent setting of the `__mro__` attribute of classes, Python core structures are exactly the primary structures $(\underline{O}.pr, \epsilon, \leq)$ with two helix classes. (Therefore, the Python object model conforms to the tight canonical structure.) Helix classes are named as follows:

$$\underline{c} = \texttt{type} < \texttt{object} = \underline{r}$$

In Scala and Java

In Scala and Java, the term "*classes*" is used just for a subset of the set $\underline{C}$ of primary descendants of $\underline{r}$. The subset forms a closure system in $(\underline{r}.\!\downarrow, \leq)$, so that it can be expressed as $\underline{C}.c$ where $.c$ is an explicit closure operator added to the canonical structure. The generalized structure is then of the form $(\underline{O}, \epsilon, .c)$. This way the set $\underline{C}$ is split into *classes* and *non-terminal mixins*. In Scala, mixins are called *traits*, in Java they are *interfaces*. Mixins are not allowed to be metaclasses.

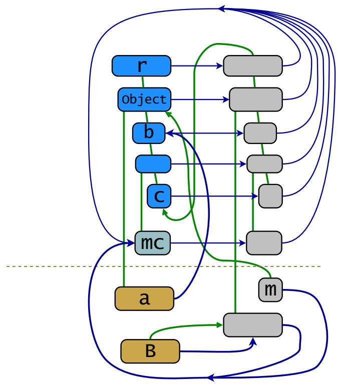
The structure is then subject to additional constraints. There are no explicit metaclasses other than $\underline{c}$, and, more importantly, a single inheritance between classes applies (but not between traits / interfaces). The helix contains 3 classes:

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

For Java, the `Any` class can be thought of as a fictitious root allowing primitive values to be objects – they become objects that are not `Object`s. Mixins (traits / interfaces) are among descendants of the `Object` class. Java in addition disallows interleaving interfaces with classes, so that $x.c$ = `Object` for every interface x.

In Smalltalk-80

Both Pharo and Squeak, the two major Smalltalk-80 implementations, provide several ways to break fundamental characteristics of the core structure such as the one-to-one correspondence between classes and implicit metaclasses or acyclicity of inheritance. The diagram below shows that (1) instantiating the class named `Behaviour` (or `Class` or `ClassDescription`) creates a "dangling class", (2) instantiating the class named `Metaclass` creates a "dangling metaclass", (3) a class can be made a direct inheritance descendant of its metaclass. The `superclass:` method allows to change the `superclass` link to point to (presumably) arbitrary non-terminal object so that cycles can arise.



(Pharo 1.3 / Squeak 4.2)

```
r  := ProtoObject.
b  := Behavior.
c  := Class.
mc := Metaclass.
```

(1) `a := Behavior new.`
(2) `m := Metaclass new.`
(3) `Object subclass: #B.`
`B superclass: (B class).`

We assume that the above anomalies are only allowed due to negligence of implementations, not by design. To rule out the anomalies, we only consider structures created

- without instantiation or subclassing of `Behavior` or of its descendants,
- without using the `superclass:` setter.

As already demonstrated on the sample core structure, the Smalltalk-80 object model does not conform to the canonical structure of ϵ due to the *metaclass redirection*. This is expressed via an imposed metaclass root $\underline{ȼ}$, named `Metaclass`, which induces the corresponding imposed class map, *.ȼlass*. This map coincides with the standard *.class* map except for implicit metaclasses, where it "redirects" the value from the `Class` class to the `Metaclass` class, introducing monotonicity breaks with respect to inheritance. Note that the `class` introspection method does not correspond to *.ȼlass* but to *.aȼlass* (the *imposed actualclass map*) which takes object actuality into account.

Another quirk that can be captured by an appropriate generalization of the canonical structure is formed by *additional twist links*. These are inheritance child-parent pairs $(x.ec, \underline{c})$ where x is a "subsidiary" inheritance root – a built-in parentless class other than $\underline{r}$. Each of Pharo and Squeak contains one such class, named `PseudoContext` and `ObjectTracer`, respectively.

The helix chain contains 5 classes:

$$\underline{c} = \texttt{Class} < \texttt{ClassDescription} < \texttt{Behavior} < \texttt{Object} < \texttt{ProtoObject} = \underline{r}$$

The `Metaclass` class is a sibling of the `Class` class. Metaclasses can be defined as *C.ȼlass.↧* – this makes `Class` and `Metaclass` the only explicit metaclasses. Finally, the Ruby conditions apply: single inheritance and single metalevel for classes.

Traits

Similarly to Ruby module inclusion, Smalltalk-80 supports extension of inheritance via inclusion of terminal objects, called traits.[56] Traits are `Trait`s – instances of the built-in `Trait` class. Unlike in Ruby, the semantics of extended object membership, ϵ, is not reflected by the `isKindOf:` introspection method (as of

Pharo 1.3).

In Objective-C

In Objective-C, the eigenclass model has to be generalized to allow multiplicity and degeneracy of inheritance roots. There are several components of object membership, each with its own inheritance root. In each component, the inheritance root r is the only helix class so that $r.class = r$. Equivalently, $(r.ec, r)$ is the twist link.

As of GNUstep, there are (at least) 3 built-in inheritance roots, named `Object`, `NSObject` and `NSProxy`. Like in Smalltalk-80, the Ruby conditions apply: single inheritance and single metalevel for classes. As a consequence of degeneracy, metaclasses cannot be expressed as $\underline{C}.class.\downarrow$ since this set contains all classes. One possible solution is to simply define a metaclass as an object on the metalevel 2 or higher (so that there are no explicit metaclasses).

In CLOS

The Common Lisp Object System deviates from the canonical structure in two features. It introduces non-linear inheritance between helix classes as well as an imposed class map with monotonicity breaks w.r.t. inheritance. Unlike in Smalltalk, this imposed class map cannot be expressed via a constant "redirection" target. As of CLISP 2.49, there are 8 helix classes:

$$\underline{c} = \texttt{class} < \texttt{clos::potential-class} < \ldots < \texttt{standard-object} < \texttt{T} = \underline{r}$$

The missing 4 classes do not form a chain in inheritance. Like in Python, there are no additional constraints: multiple inheritance is supported as well as creation of explicit metaclasses.

In Perl 5

As already shown on the sample structure, the Perl object model is distinguished by total circularity of classes: Every class is the class of itself. This is equivalent to say that Perl establishes the $(\underline{C}, \leq) = (\underline{C}, \epsilon)$ equality: the instance-of relation, in its restriction to classes, coincides with the inheritance relation. This way Perl merges the two different meanings of *is-a*. Consequently, the `isa` introspection method can be used both to detect membership as well as inheritance.

As for metaclasses, we can apply the solution proposed to Objective-C and define metaclasses to be the objects on metalevel 2 or higher. Since Perl does not have any actual objects on these metalevels, there are no actual metaclasses.

In Perl, multiple inheritance is allowed. The built-in class named `UNIVERSAL` stands for the inheritance-root $\underline{r}$. However, the assumption is needed that the `@ISA` variable of this class is not changed.

Eigenclass actuality

The concept of *eigenclass actuality* has already been introduced for the Ruby core structure. Any in-memory representation of object membership can only store a finite "front" part of the structure. We call objects from this substructure *actual(s)*. In languages that do not support ϵ refined by implicit objects, the actual objects are exactly the primary ones. In general, some eigenclasses may also be actual.

Given a monotonic eigenclass structure $(\underline{O}, \epsilon)$, possible subsets A of actual objects can be axiomatized as follows. (For a set X of objects, we let $X.\wedge$ be the set of all *strict upper bounds* of X.)

(a~1) A is finite.

(a~2) $A \supseteq \underline{O}.pr$. (Every primary object is actual.)

(a~3) $A \supseteq A.ce$. (If $x.ec$ is actual then so is x.)

(a~4) A is a closure system in $\leq$. (Every object has a least actual ancestor.)

(a~5) $A \cap \underline{H} = (\underline{R} \setminus A).\wedge$. (There exist a natural k such that for every helix object x, $x \in A \leftrightarrow x.mli < k$.)

Conditions (a~4) and (a~5) assert that there is a "twist" pair of objects t, u such that $u \succ t \in \underline{R}$ and $u.\updownarrow = A \cap$

H. (That is, *u* is the unique inheritance parent of a metalevel top *t*, and *u* is the bottom of all actual helix objects.) Since *A* is a closure system in inheritance, there is a corresponding closure operator, *.a*, and the corresponding *actualclass* map, *.aclass* = *.ec.a*. For an object *x*, the actualclass of *x* is the least actual container of *x*.

In canonical structures, the *.aclass* map forms a tree (similarly to *.class*). The *u* object that is the bottom of *A* ∩ H belongs to the eigenclass chain of c (the instance / metaclass root) and is the unique fixpoint of *.aclass* (that is, *u* is the root of the actualclass tree). Observe also that

x.ec ≤ *x.aclass* ≤ *x.class*.

While *.ec* is a conceptual refinement of *.class*, the actualclass map is an implementation-oriented refinement. The *.aclass* map is identical to the *.class* map iff no eigenclass is actual.

Note: In most cases (e.g. in the introductory sample), the blue arrows contained in the diagrams show the actualclass map. (In general, blue arrows have been used to display *.aclass* or *.class* or *.class.*↿ or *.aȼlass*.)

Specializations

In Python, Java, CLOS, and Perl, eigenclasses are purely fictitious – they are never actual, so that the actualclass map coincides with the class map. Scala allows eigenclasses of terminal objects, e.g. via the `object` definition. In Smalltalk-80 and Objective-C, the set of actual eigenclasses equals C*.ec*.

In Ruby

In Ruby, all objects that are not *immediate values* can have actual eigenclasses. As of MRI/YARV 1.9 inheritance ancestors of actual objects are actual, i.e. for the set *A* of actual objects

(a~6) *A* is an up-set in ≤.

This condition is thus satisfied for implementations of all languages with (e~9⁺⁺) considered in this document. (In general, *A* needs to be replaced with *A* ∪ {r*.ec*}.)

Ruby is the only language that supports dynamic eigenclass allocation. Typically, for an object *x*, the eigenclass *x.ec* becomes actual by defining "singleton" methods for *x*. For efficiency, there are more allocated eigenclasses than those accessed from the user code. This induces 2 different extents of actuality, as described before.

In Smalltalk-80

In Smalltalk, the imposed class map, *.ȼlass*, has the corresponding imposed actualclass map, *.aȼlass*, which corresponds to the introspection method named `class`. Due to the metaclass redirection, *.aȼlass* does not form a tree but just a (directed) pseudotree with a 2-element cycle containing the `Metaclass` class and its eigenclass.

The Dylan core structure

The Dylan programming language is presumably the first language that introduced singleton objects and probably the only language (as of 2014) that supports – at least experimentally – both powerclasses and universal singletons. The singleton support makes Dylan the only non-monotonic language considered in this document.

Non-terminal objects in Dylan are called *types* [20a]. The following table shows 3 supported kinds of types that have correspondence to basic structures.

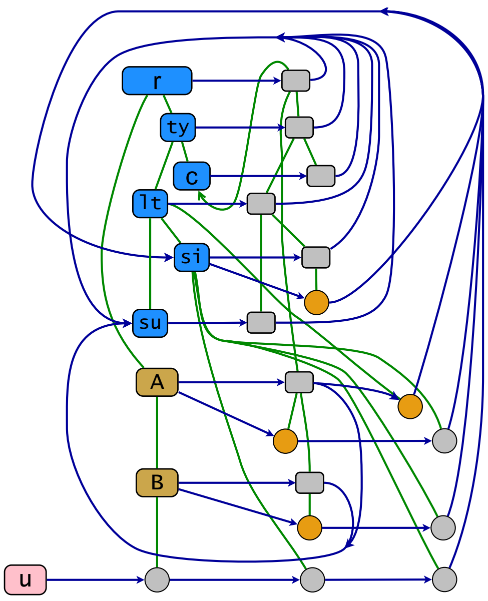
Kind of Dylan "types"	Our terminology	Evaluation	Instances of (via `instance?`)
Classes			`<class>`
Singletons		*x.εɕ* ↔ `singleton(x)`	`<singleton>`
Subclass types [54]	Powerclasses outside O*.εc*	*x.ec* ↔ `subclass(x)`	`<subclass>` (∗)

Notes:

1. (∗) As of Open Dylan 2013.2, the `<subclass>` class is not referenced by its name.
2. Evaluation of `subclass(x)` is supported for every *x* that is a class.
3. Evaluation of `singleton(x)` is supported for all objects *x*. As a consequence, there cannot be a perfect correspondence between singletons in Dylan and singletons in basic structures since the latter ones are only defined for *bounded* objects.
4. Every evaluation of `singleton(x)` or `subclass(x)` (even when performed repeatedly with the same argument *x*) returns a new object. Such objects can be thought of as equivalent representants of a given singleton or powerclass.

Sample

The following diagram shows a sample core structure of Dylan. Inheritance between non-terminal objects can be detected by the `subtype?` method. The composition of blue arrows with inheritance – object membership in Dylan – is exactly what is detected by the `instance?` method. All powerclasses of classes (i.e. all the supported powerclasses w.r.t. given set of classes) are displayed. In contrast, supported singletons are only shown for some objects.



(Open Dylan 2013.2)

```
let r  = <object>;
let ty = <type>;
let c  = <class>;
let si = <singleton>;
let lt = last(direct-superclasses(si));
let su = object-class(subclass(r));
let A = make(c);
let B = make(c, superclasses: list(A));
let u = make(B);
```

`lt` … `<limited-type>`

`su` … `<subclass>`

Observations:

1. There is a correspondence between "subclass types" in Dylan and metaclasses in Smalltalk-80.

	Dylan	Smalltalk-80
Powerclasses of classes (*C.ec*) are termed:	Subclass types	(Implicit) metaclasses
The (imposed) metaclass root *Ȼ* is named:	`<subclass>`	`Metaclass`
For a class *x*, the powerclass *x.ec* is evaluated by:	`subclass(x)`	`x class`
The imposed class map *.ȼlass* is introspected by method:	`object-class`	—
The imposed actualclass map *.aȼlass* is introspected by:	—	`class`

2. There is a cycle in membership caused by singletons. According to the `instance?` method,
   - `<singleton>` and `singleton(<singleton>)` are members of each other.
3. Powerclasses (Dylan's "subclass types") can only have classes as members. As a consequence, *(≤) ⊚ (.ec) ⊆ (∊)* is not satisfied in general. For example, `u`*.εc* (the singleton of the terminal object `u`) is from `B.↧` but not from `B`*.ec.∋*.

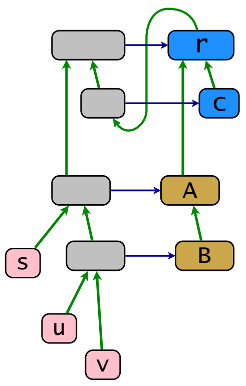
Prototypes

It is also possible to refine the instance-of relation the opposite way by eigenclass *predecessors* of objects, rather than successors. Such implicit objects are called *(instance) prototypes*. The prototype of a class y, denoted $y.ce$, is the highest instance of y, i.e. every instance of y is an inheritance descendant of $y.ce$. In contrast to eigenclasses, we only consider prototypes of classes. First, it does not (seem to) make sense to have instance prototypes of terminal objects. Second, having no prototypes of prototypes simplifies the description.

Formally, *object membership with prototypes* can be defined by combining eigenclasses and prototypes. In the specialized document [49], $\mathbf{S1}_J$ structures are introduced as structures $(\underline{\Theta}, \underline{O}, \epsilon)$ such that $(\underline{\Theta}, \epsilon)$ is a canonical eigenclass structure without terminal objects, and $\underline{O}$ is a subset of $\underline{\Theta}$ such that the substructure $(\underline{O}, \epsilon)$ is a canonical eigenclass structure whose non-terminal objects form the set $\underline{\Theta}.ec.\downarrow$. Objects from $\underline{\Theta} \setminus \underline{O}$ are the instance prototypes. The inclusion $\underline{\Theta}.ec.\downarrow \subseteq \underline{O}$ establishes a one-to-one correspondence between the i-th metalevel of $(\underline{\Theta}, \epsilon)$ and the $(i+1)$-th metalevel of $(\underline{O}, \epsilon)$ for each natural i.

Sample

The following diagram shows the prototypal completion of the instance-of sample. Blue links display the restriction of the class map to the set $\underline{C}.ce$ of prototypes, let it be denoted $.\mathfrak{c}lass$. The (unrestricted) $.class$ map is inherited from $.\mathfrak{c}lass$ The instance-of relation is given by $(\epsilon) = (\leq) \circledcirc (.class)$, or by $(\epsilon) = (\leq) \circledcirc (.\mathfrak{c}lass)$, or also by $(\ni) = (.ce) \circledcirc (\geq)$. (In this particular case we do not need to distinguish between member-of and and instance-of.)



(JavaScript)

```
var r, c, A, B, s, u, v;
r = Object
c = Function
A = new c
B = new c; B.prototype.__proto__ = A.prototype
s = new A
u = new B; v = new B
```

The missing inheritance between classes:

```
A.__proto__ = c.__proto__ = r; B.__proto__ = A
```

Note that $\underline{r}.ce$ becomes the "new" inheritance root – the unique common ancestor of all objects (in contrast to $\underline{r}$ which is just a common root for non-zero metalevels).

In JavaScript

As of ECMAScript, 5th Edition, [17] the JavaScript programming language provides a limited native support for object membership. The (ideal) structure is established by 3 (partial) maps between objects, $.sc'$, $.class$, and $.ce$, given by object properties named `__proto__`, `constructor` and `prototype`, respectively. For an object x and a class y,

- `x.__proto__` is the inheritance parent of x,
- `x.constructor` is the class of x,
- `y.prototype` is the instance prototype of y.

The `constructor` property is owned by prototypes and inherited by other objects. In contrast, `__proto__` and `prototype` are never (strictly) inherited. The `__proto__` property is non-standard. When not supported, `Object.getPrototypeOf(x)` can be used for introspection. The instance-of relation can be introspected via the `instanceof` operator, except that `y.prototype` is not reported as an instance of y.

The Ruby conditions apply: single inheritance between objects and single metalevel for classes. In contrast to Ruby, Smalltalk or Objective-C, JavaScript does not support parallel metalevel hierarchies. Only inheritance between prototypes is supported, not between classes. This is why we wrote $.sc'$ instead of just $.sc$. According to $.sc'$, the inheritance parent of every class is $\underline{c}.ce$ (`Function.prototype`). However, the modification $(\leq) \rightarrow (\leq') = (\leq) \setminus (\underline{C}, <)$ has no impact to the instance-of relation: we still have $(\epsilon) = (\leq') \circledcirc (.class) = (\leq') \circledcirc (.\mathfrak{c}lass)$. There are two helix classes:

$$\underline{c} = \texttt{Function} < \texttt{Object} = \underline{r}.$$

Note that we wrote $<$, not $<'$ which does not hold.

As the names `constructor` or `Function` suggest, JavaScript uses another terminology for the class map or the class set. In JavaScript, classes are *constructors* and also *functions*. Every constructor is a function. Functions are the `Function`s. Except for $c.ce$ and for native built-in functions like `eval` or `parseInt`, every function is also a constructor. Being terminal instances of c, the native built-in functions account for the relaxation of (p~5). Another deviation from the canonical structure is caused by the built-in `Function.prototype.bind` method. A constructor x created using this method shares its instance prototype $x.ce$ with the constructor to which x is bound (although $x.ce$ is not obtained via `x.prototype`.)

The above description of the JavaScript native core structure is only valid under ideal circumstances which are not guaranteed by the ECMAScript standard. For example, the `constructor` or `prototype` properties can be almost arbitrarily manipulated. To ensure validity, additional restrictions have to be imposed on state transitions.

Ontological structure of ∊

A generalization of the canonical structure of $\in$ allows for a description of the core structure of ontologies in which classes are (among) individuals. We introduce a two-step generalization motivated by the RDF Schema.[63] [64] The "narrow" definition can be characterized by the following features:

- Distinction of *properties* as instances of a special class, p. Properties, like terminal objects, are not descendants of the inheritance root. In contrast to terminals, properties can have (other properties as) ancestors / descendants.
- Inheritance does not have to be antisymmetric – distinct classes or properties can be descendants of each other and thus become equivalent.
- *Multiple classification* – an object x can have multiple minimum classes of which x is an instance, so that the existence of $x.class$ is not guaranteed.

Disallowing the above features leads to primary (canonical) structures. The "broad" definition further generalizes the "narrow" definition by only using those constraints that are imposed by RDFS axioms and entailments rules.

Narrow definition

By a *primary ontological structure* of $\in$ we mean a structure $(\bar{O}, \in, \leq, r, c, p)$ with a similar terminology, notation and axioms as canonical primary structures. The differences are as follows. p is a distinguished object whose instances are called *properties*. We denote P the set of all properties. The set T of *terminal* objects equals $\bar{O} \setminus (C \cup P)$. The structure is then subject to the following axioms:

(o~1) Inheritance, $\leq$, is a preorder.

(o~2) (a) $(\in) \circ (\leq) \subseteq (\in)$ and (b) $(\leq) \circ (\in) \subseteq (\in)$.

(o~3) (a) Only classes can have instances. (b) Terminals have no strict descendants.

(o~4) Helix classes are (a) totally pre-ordered by $\leq$, and (b) instances of each other.

(o~5) Metaclasses can only have classes as instances.

(o~6) Every non-helix object is well-founded w.r.t. $\in$.

(o~7) (a) $\bar{O} = r.\ni$. (b) $C.\in \subseteq c.{↥} \cup c.{↧}$.

(o~8) The set C of classes is finite.

(o~9) c is a least helix class.

(o~10) p is a class not from $c.{↥} \cup c.{↧}$.

Notes and observations:

1. Non-equivalence of r and c is asserted by (o~10).
2. Condition (o~7) is a weakening of the class map axiom (p~7). It asserts that (a) every object has a container, and (b) only helix classes or metaclasses can have classes as instances.
3. Condition (o~10) asserts disjointness of classes and properties so that $\bar{O} = T \uplus P \uplus C$.
4. Primary (canonical) structures (equipped with a "pointed" class p) are the primary ontological structures such that

- $\leq$ is antisymmetric,
- $\underline{p}$ has no instances, and
- $x.class$ exists for every object x.

Broad definition

By the "broad" definition, an *RDFS core structure* is a structure $(\underline{\bar{O}}, \epsilon, \leq_C, \leq_P, \underline{r}, \underline{c}, \underline{p})$ where $\underline{\bar{O}}$ is the set of *objects* or *resources*, ϵ, $\leq_C$, and $\leq_P$ are the *instance-of*, *subclass-of* and *subproperty-of* relations on $\underline{\bar{O}}$, respectively, and $\underline{r}$, $\underline{c}$, and $\underline{p}$ are distinguished objects. Instances of $\underline{c}$ form the set $\underline{C}$ of *classes*, instances of $\underline{p}$ form the set $\underline{P}$ of *properties*. The structure is subject to the following conditions:

(r~1) The subclass-of relation, $\leq_C$, is a preorder on $\underline{C}$.   (rdfs10) and (rdfs11)

(r~2) The subproperty-of relation, $\leq_P$, is a preorder on $\underline{P}$.   (rdfs5) and (rdfs6)

(r~3) The subsumption rule applies: $(\epsilon) \odot (\leq_C) \subseteq (\epsilon)$.   (rdfs9)

(r~4) Only classes can have instances.

(r~5) Every object is an instance of $\underline{r}$.

(r~6) Every class is a subclass of $\underline{r}$.   (rdfs8)

(r~7) The objects $\underline{r}$, $\underline{c}$ and $\underline{p}$ are pairwise distinct.

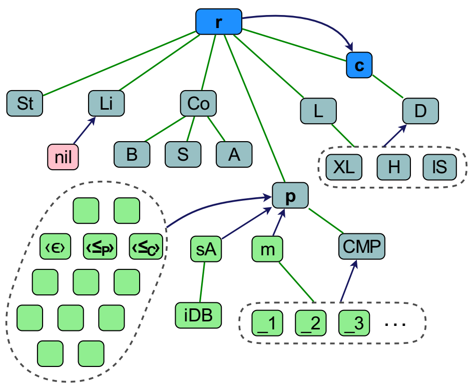
(r~8) The set $\underline{C} \setminus \underline{P}$ is finite.

(r~9) $\underline{p}$ is a class.

The "narrow" definition is obtained from the "broad" one by defining the inheritance relation $\leq$ as the reflexive closure of $(\leq_C) \cup (\leq_P)$ and making the following assertions: (o~2)(b): $(\leq) \odot (\epsilon) \subseteq (\epsilon)$, (o~4): helix classes are …, (o~6): well-foundedness of ϵ on non-helix objects, (o~7)(b): $\underline{C}.\epsilon \subseteq \underline{c}.↥ \cup \underline{c}.↧$, (o~9): $\underline{c}$ is a least helix class, and (o~10): $\underline{p}$ is not from $\underline{c}.↥ \cup \underline{c}.↧$.

Built-in RDFS core

The *built-in RDFS core structure* is an RDFS core structure $(\underline{\bar{O}}, \epsilon, \leq_C, \leq_P, \underline{r}, \underline{c}, \underline{p})$ shown by the following diagram. We consider the set $\underline{\bar{O}}$ to be equal to the *RDF(S) vocabulary* [66], so that objects are *URI names*. Let $\leq$ be the reflexive transitive closure of green links (implicitly directed upwards). The ϵ relation is obtained as $(\leq) \odot R \odot (\leq)$ where R is the relation shown by blue links. Subsequently, $\leq_P$ and $\leq_C$ are the restrictions of $\leq$ to $\underline{p}.\ni$ and $\underline{c}.\ni$, respectively.



‹∈›	… `rdf:type`
‹≤P›	… `rdfs:subPropertyOf`
‹≤C›	… `rdfs:subClassOf`
$\underline{r}$	… `rdfs:Resource`
$\underline{c}$	… `rdfs:Class`
$\underline{p}$	… `rdf:Property`

Note that the structure is fairly regular. It has single inheritance and single classification, yielding the *.class* map. The only feature not allowed in canonical structures is strict inheritance between objects that are not descendants of $\underline{r}$. Similarly to the Python programming language, there is a minimal non-degenerate chain of 2 helix classes:

$$\underline{c} = \texttt{rdfs:Class} < \texttt{rdfs:Resource} = \underline{r}.$$

In addition to $\underline{c}$, there is a second built-in metaclass, `rdfs:Datatype`.

Note: The diagram shows the built-in structure according to RDF 1.1 Semantics [65] which adds `rdf:HTML` and `rdf:langString` as 2 new built-in instances of `rdfs:Datatype` to the previous 2004 version [64].

Interpretation by RDF graphs

In RDF Schema, data structures are encoded via RDF graphs. By an *RDF graph* we mean a structure $(\underline{\bar{O}}, \underline{\Psi})$ such that $\underline{\bar{O}}$ is set of *objects* and $\underline{\Psi}$ is a subset of $\underline{\bar{O}}^3 = \underline{\bar{O}} \times \underline{\bar{O}} \times \underline{\bar{O}}$. Elements of $\underline{\bar{O}}^3$ are *triples* of objects. For a triple $t = (s,p,o)$, s is the *subject*, p is the *predicate*, and o is the *object* of t. The *relational extent* of $p \in \underline{\bar{O}}$ is denoted $p.rel$ and defined as the set $\{(s,o) \mid (s,p,o) \in \underline{\Psi}\}$.

We can now define an *RDFS core graph* as an RDF graph $(\underline{\bar{O}}, \underline{\Psi})$ such that the following holds:

(1) $\underline{\bar{O}}$ includes the RDF(S) vocabulary.

In particular, there are six distinguished objects $\langle\epsilon\rangle$, $\langle\leq_C\rangle$, $\langle\leq_P\rangle$, $\underline{r}$, $\underline{c}$ and $\underline{p}$. We also denote ϵ, $\leq_C$, $\leq_P$ the relational extents of $\langle\epsilon\rangle$, $\langle\leq_C\rangle$, and $\langle\leq_P\rangle$, respectively.

(2) The structure $\mathcal{S} = (\underline{\bar{O}}, \epsilon, \leq_C, \leq_P, \underline{r}, \underline{c}, \underline{p})$ is an RDFS core structure.

(3) The built-in RDFS core structure is a weak substructure of $\mathcal{S}$.

Since every object appears as a subject in some triple, an RDFS core graph is completely given by its set of triples. The *built-in RDFS core graph* is the minimum set of triples. Any RDFS core graph is obtained by (explicitly) adding new triples and subsequently applying *entailment rules* – the presence of some triples *entails* the presence of other triples.

There are entailment rules that refer to distinguished objects not considered in the above description. (In particular, the `rdfs:domain` and `rdfs:range` properties.) As a consequence, some of our RDFS core graphs are in fact not allowed by RDF Schema. However, RDF Schema still only guarantees the conformance to the "broad" definition of ontological structure of ϵ. None of (o~2)(b), (o~4), (o~6), (o~7)(b)[32], (o~9), or (o~10) is asserted.

OWL 2

In OWL 2 Full [62], the structure of RDF Schema is extended by additional axiomatic triples. The extension is then subject to the (additional) OWL 2 RL/RDF rules [61]. However, no conditions from the difference between "narrow" and "broad" structures are asserted.

Built-in structure

The built-in structure contains additional 58 classes and 62 properties [45a]. There are four helix classes, preordered by

$$\{\texttt{owl:Class}, \texttt{rdfs:Class}\} < \{\texttt{owl:Thing}, \texttt{rdfs:Resource}\}.$$

which indicates that RDFS helix classes have their OWL equivalents. Similarly, `rdf:Property` is equivalent to `owl:ObjectProperty` and the metaclass `rdfs:Datatype` is equivalent to (the metaclass) `owl:DataRange`. In total, there are 7 built-in descendants of `rdfs:Class`. However, one of them, `owl:Nothing`, is special since it is asserted to (a) be a descendant of all classes and (b) to have no instances. That is, `owl:Nothing` is meant to be an abstraction of the empty set. As a consequence, `owl:Nothing` is not regarded as a metaclass (cf. [51]). There is still single inheritance between non-equivalent objects other than `owl:Nothing`. Single classification is not preserved (not even up to equivalence) due to common instances of `owl:AnnotationProperty` and `owl:OntologyProperty`.

The restriction of inheritance to RDF(S) vocabulary is the same as inheritance in the built-in RDFS core structure. The restriction of instance-of is almost the same: the `rdfs:Literal` class is made an instance of `rdfs:Datatype` by a single additional axiomatic triple [62a].

Powertypes

The interplay between inheritance, $\leq$, and membership, ϵ, provided by the powerclass map, $.ec$, and expressed by

$(\geq) = (.ec) \circledcirc (\ni)$

has been investigated in type-theoretic setting by Luca Cardelli [11]. Let us focus just on the structure induced by the typing relation and the power type operator. Then, if we denote $\underline{c} = \underline{r}.ec$, the following correspondence can be established:

This document		Type-theoretic setting [11]	
Terminology	Notation	Notation	Terminology
Inheritance	≤	≤ / ⊆ / <:	Subtyping
Object membership	ϵ	:	Typing
Powerclass map	.ec	*Power()*	Power type operator
Terminal objects	$\underline{T}$		Values
Non-terminal objects	$\underline{c}$.∋		Types
Top of metalevel 2	$\underline{c}$	*Type*	Type of types

Notes:

1. In [11], the ⊆ symbol is used for subtyping. The table shows also two other symbols that are common in the literature.
2. The correspondence between inheritance (as defined in this document) and subtyping should be taken in the restriction to non-terminal objects. A value is usually not considered to be a subtype of itself.

In the specialized document [50], Cardelli's six typing rules are distinguished (those that can be expressed by just *:*, *Power()* and *Type*) to form an *abstract power type system* $(\underline{O}, \epsilon, .ec, \underline{c})$. Subsequently, additional conditions are provided so that the resulting family of structures is definitionally equivalent to basic structures of ϵ such that *x.ec* is defined exactly for non-terminal objects *x*.

Powertypes in metamodelling

The Cardelli's notion of power types has been adopted by J. Odell in the field of metamodelling [37]. However, whereas Cardelli's power type is an abstraction of powerset, Odell's power type is an abstraction of a *non-trivial partition*. In particular, in metamodelling,

- a type can have more than one power type, and
- there is no typing relation (instance-of) between a type and its power type.

Since for every sets *x*, *y*, "*y* being a non-trivial partition of *x*" is a special case of "*x* being a non-member union of *y*", the semantic shift can be diagrammatized by

$$(.ec) \dashrightarrow .\varpi'(-1)$$

where *.ϖ'* is a distinguished subrelation of the non-member union map *.ϖ*.

References

[1] Martin Abadi, Luca Cardelli, ***A Theory of Objects***, Springer Science & Business Media 1996, http://lucacardelli.name/theoryofobjects.html
[1a] Preface

[2] James Althoff, ***[Python-Dev] Classes and Metaclasses in Smalltalk***, 2001, https://mail.python.org/pipermail/python-dev/2001-May/014508.html

[3] Giuseppe Attardi, Cinzia Bonini, Maria Rosario Boscotrecase, Tito Flagella, and Mauro Gaspari, ***Metalevel Programming in CLOS***, In ECOOP, vol. 89, 1989, http://www.ifs.uni-linz.ac.at/~ecoop/cd/papers/ec89/ec890243.pdf

[4] Daniel Bobrow, Gregor Kiczales, ***The common Lisp object system metaobject kernel: a status report***, Proceedings of the 1988 ACM conference on LISP and functional programming, ACM 1988,

[5] Daniel Bobrow, Mark Stefik, ***The LOOPS Manual***, Xerox Corporation 1983,

[6] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston, ***Object-oriented analysis and design with applications***, Vol. 3, Addison-Wesley 2008,

[7] Ronald J. Brachman, ***What IS-A is and isn't: An analysis of taxonomic links in semantic networks***, Computer 16.10, North-Holland 1983,

[8] Kim B. Bruce, ***Foundations of Object-Oriented Programming Languages: Types and Semantics***, MIT Press 2002,
[8a] Overview from the publisher

[9] Jean-Pierre Briot, Pierre Cointe, ***The OBJVLISP Model: Definition of a Uniform, Reflexive and Extensible Object Oriented Language***, European Conference on Artificial Intelligence (ECAI'86), Advances in Artificial Intelligence-II, North-Holland 1986,

[10] Jean-Pierre Briot, Pierre Cointe, ***A Uniform Model for Object-Oriented Languages Using the Class Abstraction***, Tenth International Joint Conference on Artificial Intelligence (IJCAI'87), 1987,

[11] Luca Cardelli, ***Structural Subtyping and the Notion of Power Type***, Proc. of the 15th ACM Symp. on Principles of Programming Languages, POPL'88, 1988,
http://www.daimi.au.dk/~madst/tool/tool2004/papers/structural.pdf

[12] C. C. Chang, H. Jerome Keisler, ***Model Theory***, Studies in Logic and the Foundations of Mathematics (3rd ed.), Elsevier, 1990,

[13] David Chisnall, ***Cocoa Programming Developer's Handbook***, Addison Wesley 2009,

[14] Pierre Cointe, ***Metaclasses are First Class: the ObjVlisp Model*** , Proceeding OOPSLA '87 Conference proceedings on Object-oriented programming systems, languages and applications , North-Holland 1987,

[15] Damian Conway, ***Object Oriented Perl***, Manning Publications, 2000,

[16] Mohamed Dahchour, Alain Pirotte, Esteban Zimányi, ***Definition and Application of Metaclasses***, Proceedings of the 12th International Conference on Database and Expert Systems Applications, Springer-Verlag 2001, http://cs.ulb.ac.be/publications/P-01-01.pdf

[17] Ecma International, ***ECMAScript Language Specification, Edition 5.1***, http://www.ecma-international.org/ecma-262/5.1/

[18] (eigenclass.org admin), ***The double inclusion problem***, 2005,
http://eigenclass.org/hiki/The+double+inclusion+problem

[19] David Flanagan, ***JavaScript: The Definitive Guide, Sixth Edition***, O'Reilly 2011

[20] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington, ***Dylan Programming: An Object-oriented and Dynamic Language***, Addison Wesley 1996 opendylan.org/books/dpg/
[20a] Nonclass Types

[21] Ira R. Forman, Scott H. Danforth, ***Putting Metaclasses to Work***, Addison Wesley 1998

[22] Ira R. Forman, Nate Forman, ***Java Reflection in Action***, Manning Publications 2005

[23] Cesar Gonzalez-Perez, Brian Henderson-Sellers, ***A powertype-based metamodelling framework***, Software & Systems Modeling 5.1 2006

[24] Daniel Ingalls, ***The Smalltalk-76 programming system design and implementation***, Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM, 1978

[25] IPA Ruby Standardization WG, ***Ruby Draft Specification***, 2010,
https://www.ipa.go.jp/osc/english/ruby/

[26] Andri Joyal, Ieke Moerdijk, ***Algebraic set theory***, Cambridge University Press 1995

[27] John L. Kelley, ***General Topology***, Springer 1975

[28] Seiji Koide, Hideaki Takeda, ***OWL-Full Reasoning from an Object Oriented Perspective***, The Semantic Web–ASWC 2006 Springer 2006 http://www-kasm.nii.ac.jp/papers/takeda/06/koide06aswc.pdf

[29] Timothy C. Lethbridge, Robert Laganiere, ***Object-oriented software engineering***, McGrawhill Education 2005

[30] Hector Levesque, John Mylopoulos, ***A procedural semantics for semantic networks***, Associative networks: Representation and use of knowledge by computers 1979

[31] Mark Lutz, ***Learning Python***, O'Reilly 2009

[32] Aimilia Magkanaraki, Sofia Alexaki, Vassilis Christophides, and Dimitris Plexousakis, ***Benchmarking RDF Schemas for the Semantic Web***, ISWC, 2002, http://www.ics.forth.gr/isl/publications/paperlink/iswc02.pdf

[33] Satoshi Matsuoka, Akinori Yonezava, ***Metalevel solution to inheritance anomaly in concurrent object-oriented languages***, Proceedings of the ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming 1990

[34] Satoshi Matsuoka, ***Language Features for Re-Use and Extensibility in Concurrent Object-Oriented Programming Languages***, 1993

[35] Linda G. DeMichiel, Richard P. Gabriel, ***The common lisp object system: An overview***, ECOOP'87 European Conference on Object-Oriented Programming 1987

[36] Oscar Nierstrasz, Stéphane Ducasse, Damien Pollet, ***Pharo by Example***, Square Bracket Associates 2009, http://pharobyexample.org

[37] James J. Odell, ***Power Types***, Journal of Object-Oriented Programming 7.2 1994

[38] OMG, ***OMG UML Superstructure 2.4.1***, Object Management Group 2011

[39] Jari Palomäki, Hannu Kangassalo, ***That IS-IN Isn't IS-A: A Further Analysis of Taxonomic Links in Conceptual Modelling***, 2012

[40] Ondřej Pavlata, ***The Ruby Object Model: Data Structure in Detail***, 2012, http://www.atalon.cz/rb-om/ruby-object-model

[41] Ondřej Pavlata, ***The Ruby Object Model: S1 superstructure representation***, 2012, http://www.atalon.cz/rb-om/ruby-object-model/s1-rep/

[42] Ondřej Pavlata, ***Ruby Object Model – The S1 structure***, 2012, http://www.atalon.cz/rb-om/ruby-object-model/rb-om-s1.pdf

[43] Ondřej Pavlata, ***The Ruby Object Model: Comparison with Smalltalk-80***, 2012, http://www.atalon.cz/rb-om/ruby-object-model/co-smalltalk/

[44] Ondřej Pavlata, ***Object Membership: The Core Structure of Object-Oriented Programming***, 2012–2015, http://www.atalon.cz/om/object-membership/oop/

[45] Ondřej Pavlata, ***Object Membership: The ontological structure***, 2012, http://www.atalon.cz/om/object-membership/ontology/
[45a] OWL built-in structure

[46] Ondřej Pavlata, ***Object Membership – Basic Structure***, 2015, http://www.atalon.cz/om/object-membership/basic/

[47] Ondřej Pavlata, ***Object Membership: Simplified Structure***, 2015, http://www.atalon.cz/om/object-membership/simple/

[48] Ondřej Pavlata, ***The Dialectic of Classes and Metaclasses in Smalltalk-80***, 2015, http://www.atalon.cz/om/smalltalk/dialectic/

[49] Ondřej Pavlata, ***Object Membership with Prototypes***, 2015, http://www.atalon.cz/om/object-membership/prototypes/

[50] Ondřej Pavlata, ***Object Membership and Powertypes***, 2015, http://www.atalon.cz/om/object-membership/powertypes/

[51] Ondřej Pavlata, ***What Is a Metaclass?***, 2016, http://www.atalon.cz/om/what-is-a-metaclass/

[52] Ondřej Pavlata, ***Featherweight Java Axiomatically***, 2016, http://www.atalon.cz/om/featherweight-java-axiomatically/

[53] Guiseppe Peano, ***Arithmetices principia: nova methodo***, Fratres Bocca 1889, https://archive.org/details/arithmeticespri00peangoog

[54] Keith Playford, ***Dylan Enhancement Proposal: Subclass***, Dylan Hackers 1995, http://opendylan.org/proposals/dep-0005.html

[55] Reza Razavi, Noury Bouraqadi, Joseph Yoder, Jean-François Perrot, Ralph Johnson, ***Language***

support for Adaptive Object-Models using Metaclasses , Computer Languages, Systems & Structures, 31(3) 2005,

[56] Nathanael Schärli, ***Traits: Composing Classes from Behavioral Building Blocks***, 2005, http://scg.unibe.ch/archive/phd/schaerli-phd.pdf

[57] Andrew Shalit, Jeffrey Piazza and David Moon, ***Dylan, an object-oriented dynamic language***, Apple Computer Inc 1992,

[58] Guy L. Steele, ***Common LISP: the language***, Digital press 1990,

[59] David Ungar, Randal B. Smith, ***Self: The power of simplicity***, Vol. 22. No. 12. ACM, 1987,

[60] Richard Wiener, ***Editorial***, Journal of Object Technology 2002, http://www.jot.fm/issues/issue_2002_05/editorial/index.html

[61] World Wide Web Consortium, ***OWL 2 Profiles***, 2009, http://www.w3.org/TR/owl2-profiles/

[62] World Wide Web Consortium, ***OWL 2 RDF-Based Semantics***, 2009, http://www.w3.org/TR/owl2-rdf-based-semantics/
[62a] Additional axiomatic triples for RDFS

[63] World Wide Web Consortium, ***RDF Schema***, 2004, http://www.w3.org/TR/rdf-schema/

[64] World Wide Web Consortium, ***RDF Semantics***, 2004, http://www.w3.org/TR/2004/REC-rdf-mt-20040210/

[65] World Wide Web Consortium, ***RDF 1.1 Semantics***, 2014, http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/

[66] World Wide Web Consortium, ***RDF Vocabulary Description Language 1.0: RDF Schema***, 2004, http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

[67] World Wide Web Consortium, ***Terse RDF Triple Language***, 2014, http://www.w3.org/TR/turtle/

[68] ***Wikipedia: The Free Encyclopedia***, http://wikipedia.org
[68a] Prototype-based programming , [68b] Eigenclass model (January 2013)

License