What Is a Metaclass?

A "metaclass" approach to the foundations of object-oriented programming is presented. The core structure of object technology as established in [135] [136] [138] is used to provide a rigorous answer to the title question in various contexts. It is shown that the metaclass question is a good linguistic device since it generates four fundamental questions about the object model core: what are the *objects*, what are the *classes*, what is *instance-of* and what is the *inheritance* relation. This constitutes a basic method for the investigation of object-oriented environments.

Author

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

Document date

Initial release February 16, 2016
Last major release February 16, 2016
Last update August 21, 2016

HTML version

An HTML version of this document is available at http://www.atalon.cz/om/what-is-a-metaclass/.

Warning

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

Table of contents

Introduction

For more than two decades, object-oriented programming (OOP) has been the dominant programming methodology. Despite the ubiquitousness of the methodology in software industry and despite the formal nature of programming languages, there is no common consensus about theoretical foundations of OOP. Quoting from wikipedia [𝕎₃]:

> *Attempts to find a consensus definition or theory behind objects have not proven very successful (…), and often diverge widely.*

The sentence appeared in December 2004 in the *Formal definition* section of the wikipedia article titled *Object-oriented programming* [𝕎₁]. Until at least 2015, the wording remained unchanged except that in 2007, a note has been inserted into the parentheses with a reference to the book *A Theory of Objects* (AToO) by M. Abadi and L. Cardelli [1].

In April 2013, a paragraph was added to the same section [𝕎₅], providing links to the PhD thesis by M. AbdelGawad. In his thesis, titled *NOOP: A Mathematical Model of Object-Oriented Programming* [2], the author points out that the theoretical underpinnings described by Abadi and Cardelli are not relevant to mainstream object-oriented languages like Java. This is because of the fundamental characteristics of the type system. While the type system developed in AToO is structural, Java's type system is nominal. Similar objections have been raised to the book *Foundations of Object-Oriented Programming Languages: Types and Semantics* (FoOOPL) by Kim B. Bruce [25]. In fact, AbdelGawad regards his thesis as antithesis of Bruce's work.

In July 2013, the paragraph was judged by wikipedia contributors to be self-promotional and subsequently deleted. Although not mentioned in the talk which preceded the deletion [𝕎₇], the edit action could have been justified purely with respect to the notability criteria: a Google Scholar search for articles citing AToO returns 1500 results, as of 2012 [1c], while a search for citations of the thesis returns just 3 results, (even) as of 2015 [2a], all of them self-citations.

Yet the *Formal definition* section (later renamed to *Formal semantics*) will probably provide a more balanced view on the subject if a link to the thesis is preserved. The thesis *is* notable as a document that draws attention to the apparent mismatch between OOP research and software industry. For any theoretical model, the question of applicability or relevance to existing programming languages should be one of the primary concerns. AbdelGawad lists Java, C#, Smalltalk, C++, Scala, and X10 as examples of "*nominal OO languages*". Subsequently, examples of "*structural OO languages*" are given: Strongtalk, Moby, PolyToil, and OCaml, followed by a remark that, for the most part, these languages "*are used only by OO programming languages researchers*".

Let us now once more quote from the above mentioned Wikipedia article. As of version from June 8, 2015 [𝕎₄], the introductory part contains the following sentence:

> *Significant object-oriented languages include Python, C++, Objective-C, Smalltalk, Delphi, Java, C#, Perl, Ruby and PHP.* (∗)

Now consider the coincidence of this list of 10 languages with the lists provided by AbdelGawad. In the "N"-case (where "N" stands for "nominal") there is a substantial common part shared by the lists (⁑). Java, C++ and C# are prominent languages in software industry. Much in contrast, there is no coincidence in the "S"-case (where "S" stands for "structural"). Indeed, none of the 10 languages is "structural". The books AToO and FoOOPL must have missed an important point. It is not clear how the books relate to the OOP world as known in 2015.

Notes:

(∗) In the subsequent versions of the page, *Swift* is inserted into the list, referencing a programming language introduced in 2014 as "Objective-C without the C" [𝕎₂₇].

(⁑) We should remark that despite using the NOOP acronym in the thesis title, AbdelGawad restricts his attention to "*nominally-typed OO languages*" by which he means those nominal OO languages that are statically typed. As a notable consequence, the Smalltalk programming language is ruled out. This reduces the list of mainstream programming languages that are said to be in the author's focus to Java, C#, C++ and Scala, just as stated in [𝕎₅].

The objective

We regard AbdelGawad's thesis as notable since it addresses the issue of relevance: **OOP foundations should relate to OOP** known from software industry. In establishing the foundations, prominent programming languages should take precedence over the less used ones. This is a crucial requirement which has not been sufficiently taken into account (or accepted) in most research in the theory behind OO languages. As a result, it appears questionable whether there are any OOP foundations at all.

Foundations need not be uniform, such a requirement would be unrealistic. It is to be expected that there are multiple foundational *models*. For each such model, a clear correspondence should be established between the model and the modelled features of existing programming languages. A model might be considered more foundational than another one if it relates to a larger part of OOP. (If there is no inclusion between the compared parts of OOP then the parts should be weighted according to the significance as witnessed by software industry.)

Another important requirement for the considered foundational system is what can be called by "abstractiveness", that is, stratification by abstraction. The system should preferably be built by a gradual process of abstraction refinement. That is, there should be a hierarchy of models with the most coarse models (the most abstract / simplified / basic / fundamental) around the top. These models, being the most fundamental ones, should provide a proper definition of the most fundamental notions. Finer models (that is, the more complex ones) should be based on the simpler models just by adding complexity. This is much in the spirit of object-oriented development.

The objective of this document is to contribute to the foundations of object-oriented programming by providing a rigorous answer to the title question: *What is a metaclass?* We do this by establishing a mathematical model (or a system of models) in which the notion is precisely defined. For the most part, such a model has already been presented in other documents by the present author (cf. [135] and [138]). This document provides a more elaborate alternative to [135]. The title question is used as a rhetoric vehicle to arrive at what is believed to be the core structure of object technology. (∗)

The notion of a metaclass appears to be very interesting. In particular, it can be used as *startpoint for the study of (the messiness of) computer science*, as we demonstrate in this document. The quest for a rigorous explanation of the term leads directly to the definition of the core structure of Python. The path is rather short. Following the wording of the classic definition of a metaclass, it is necessary to interpret the terms "class", "instance-of" and "instance of a class". Since the last term is synonymous to "object" (at least in Python), its interpretation provides the definition of objects. Now the only missing constituent that needs to be defined is *inheritance* – a fundamental notion of object technology (see [178] [20] [166] [113]). That's all to it, at least in Python. The resolution of the terms leads to a family of structures $(O, \in, \leq)$ where O is a finite set of abstract entities called *objects* and $\in$ and $\leq$ are relations between these entities, called *instance-of* and *inheritance*, respectively. In these structures, *metaclasses* as well as *classes* are just objects that satisfy certain properties expressed via $\in$ and $\leq$.

Given this, the notion of a metaclass appears as one of the most simple and elementary notions in object-oriented programming. Such a result might come as a surprise to the Python community. In the fourth edition of *Learning Python* [103], one of the most voluminous books about Python, metaclasses are dealt with in the very last chapter and stated to be "*perhaps the most advanced topic in this book*" (page 1054 [103a]). This statement is accompanied by the famous quote about the "magic" of metaclasses by Tim Peters [146].

Note: (∗) We use the term "object technology" for a generalization of "object-oriented programming" so that also other "OO"-terms are included, in particular object-oriented analysis and design (OOAD) and object-oriented database management systems (OODBMS) which are listed by the Journal of Object Technology [179]. Although we are predominantly concerned with foundations of OOP, it turns out that the core part is relevant to other fields as well.

The investigation method

The solution of the metaclass question in Python provides us with a method for the recognition of core features of a given object-oriented environment. The method consists of finding precise answers to the following questions:

(A) Which entities are objects?

(B) Which entities are classes?
(C) What relation is the counterpart of Python's $\in$?
(D) What relation is the counterpart of $\leq$?
(E) Which entities are metaclasses?

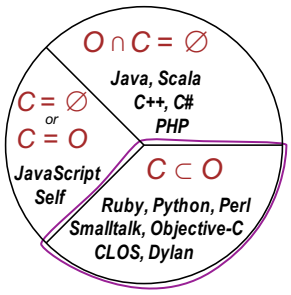
We will later extend the list of constituents which need to be recognized by *.ec* and *.ɛc* which are distinguished (possibly empty) subrelations of $\in$. Moreover, we consider environments in which (E) or even (B) can be answered by "none".

We will see that finding the answers to the above fundamental questions is often a difficult task, revealing inconsistencies and sloppiness in the terminology used in the literature. We can already give FoOOPL [25] as an example that demonstrates difficulties with (A). In chapter 10, "*the formal specification of a simple object-oriented programming language, SOOL*" is presented with no clear statement about which entities are considered to be objects. The most explicit statement in this respect appears in chapter 14, where objects are said to be "*represented as implicit references*".

Another example of a "foundational puzzle" in this respect is *Featherweight Java* (FJ) [83], currently being the most popular formal simplification of the world's most popular object-oriented programming language. Did you know that in FJ, objects are a special case of entities called "object creation"? What are classes in FJ? (cf. [143])

The main division of OO languages

We will mainly focus on the part of object technology in which the metaclass term is likely to appear. It turns out that in most cases this is where the "*classes are objects*" principle applies. The diagram on the right shows what we consider to be the main division of OO languages. There are three groups according to the following criteria:



- Does the language support the notion of a class?
- If so, are classes among objects?

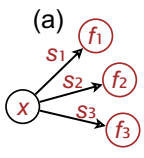
The first question alone provides a division into so called *class-based* languages (where $\underline{C} \neq \emptyset$, whenever $\underline{C}$ denotes the set of entities called classes) and *prototype-based* languages (where $\underline{C} = \emptyset$). We focus on the part in which $\emptyset \neq \underline{C} \subset \underline{O}$.

However, there is a pitfall. As of 2015, the most significant prototype-based programming language is JavaScript (this position lasting for more than one decade). It is not that clear whether JavaScript can be considered "classless". There is a strong indication (cf. [56], [53]) that JavaScript in fact *does* support the notion of a class. We therefore cannot rely on the $\underline{C} \neq \emptyset$ condition for the delimitation of (so called) class-based languages.

We address the problem by providing another justification for the above division. This is established by introducing the notion of a *method provider* as performed below. Unfortunately, the new approach does not work well with so called *multimethod* (or also *multiple dispatch*) languages like CLOS or Dylan. For these languages, we will rely on the definition of classes as presented in the literature. (See [58] and [169] for the statement of the "*classes are objects*" principle in Dylan and CLOS, respectively.)

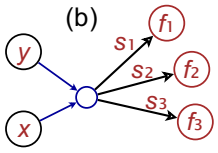
The (broken) canon of method association

Perhaps the most canonic feature of object-oriented programming is the association of objects with *methods*. Objects are said to "have" methods. Each association between an object and a method has a *name* (a "label") that identifies a method among all the methods that the object "has". Objects are receivers of *messages*. Each object responds to the messages sent to it by invoking methods with which the object is associated. Each message contains a name for the identification of the method that should be invoked. The diagram (a) on the right shows an object x that is associated to methods f_i via names s_i, $i = 1,2,3$.

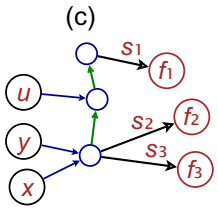


As with every OOP canon, the described method association is not canonic enough. The multimethod languages CLOS and Dylan proclaim themselves to be object-oriented yet are not subject to such a method association. In these languages, objects do not "have" methods or respond to messages sent to them. Instead, object *tuples* do.

Method providers



In a typical case, an object shares the same method association with many other objects. This is a consequence of methods typically having a counterpart in some piece of code written by a human programmer, but objects typically missing such a one-to-one counterpart. It is therefore natural to think about an *intermediate entity* that *provides* the association between an object and the methods, as shown by the (b) diagram. Here x and y are objects that both have the same method association as in the (a) diagram from the previous subsection. The intermediate entity is displayed by a blue circle.



It is also often the case that objects only *partially* share a method association. Some part of an association might be "inherited". Therefore, an intermediate entity might be split into several refined entities as shown by the (c) diagram. Here x and y are objects that both have the same method association $\{(s_1,f_1),(s_2,f_2),(s_3,f_3)\}$ as in (b) but this association has in addition a distinguished subset, a sub-association $\{(s_1,f_1)\}$ which is a method association of u. The u object is again one of the potentially many objects that have the same method association.

The derivation of method association for objects in (c) constitutes a phenomena that is (sometimes more than anything else) considered to be an essential feature of object-oriented programming. It is called by several names: *method lookup* or *method dispatch* [115], *dynamic dispatch* [𝕎₁₁] [41], *late binding* or *dynamic binding* or even *polymorphism* [51]. We should stress out that much in contrast to AToO [1], we do not consider the derivation be an implementation feature. On the contrary, we regard it as a *design concept*. The (c) structure is what is designed and the (a) structure is what is derived, not the other way round.

We call entities displayed in (c) by blue circles *method providers*. The term "method suite" from AToO can be thought of as being correspondent to "method provider". Note that there is an unnecessary split in the diagram which indicates that we consider each method provider having its own identity. An object can have several method providers which mediate the object's method association. There is a natural notion of *precedence* between the providers.

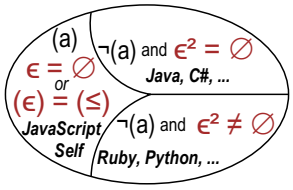
Now we provide the promised criteria that yield the same division of OOP (excluding CLOS and Dylan) as depicted at the beginning of this section but this time without a reference to the notion of a class. The questions are as follows:

Are method providers among objects?

Is every object its own method provider?

The prototype-based languages are those for which the **YES**+**YES** answer applies. The remaining languages are (called) class-based. The **YES**+**NO** answer corresponds to the "*classes are objects*" principle.

Partition by "*instance-of*"



The same division of OO languages into three groups is obtained using the concept of an *instance-of* relation. We can ask the following two questions:

- Does the language support the notion of an "instance-of" relationship?
- If so, is *instance-of-instance-of* an empty relation?

Languages that do support the concept are the class-based ones, the remaining are prototype-based. However, just like with the concept of a class, a terminologic problem appears for JavaScript. Due to the JavaScript built-in `instanceof` operator, there is a strong indication that the language in fact *does* support the instance-of relation.

In the diagram on the right, ϵ denotes the instance-of relation and ϵ^2 is the instance-of-instance-of composition. The degeneracy condition (a) that delimits prototype-based languages is again provided in two variants: $\epsilon = \varnothing$ stands for "classlessness" and $(\epsilon) = (\leq)$ stands for "classfulness", i.e. $C = O$. (Here $\leq$ is the inheritance relation. This new symbol can be avoided by writing $\epsilon = \epsilon^2$ instead.) In this document we support the latter variant for the Self programming language (see The classy Self).

Objects as instances

One of the most established canons of the (so called) class-based OO languages is that objects form the domain of *instance-of*. That is,

(a) each object is an instance of a class [20] [111] and
(b) each instance of a class is an object.

In yet another words, "object" and "instance" are synonymous, so that it does not make much sense to use the word "instance" for the characterization of an entity. Unfortunately, this rule is only partially followed in the literature. Most notably, Python publications tend to use the term "instance" for objects that are not classes [4] [103]. In contrast, the Ruby literature sticks to the rule so that the standalone term "instance" is almost non-existing here.

Metaclass pre- and anti- conditions

Since the "classes are objects" principle and the metaclass term usually appear together we express the principle and its opposite using the word metaclass. These are the two corresponding conditions:

metaclass pre-condition: classes are objects ($C \subseteq O$),
metaclass anti-condition: classes are not objects ($C \cap O = \varnothing$).

In both conditions, it is assumed that $C \neq \varnothing$. We repeat that it is not that clear whether the opposite condition of classlessness ($C = \varnothing$) applies to the so called prototype-based languages like Self.

The metaclass pre-condition turns out to be a necessary condition for the presence of metaclasses in given environment. It is however not sufficient as is shown for the case of the Perl programming language (see The Perl's `isa`).

This document naturally focuses on the part of object technology that is subject to the metaclass pre-condition. It should be noted that many (and presumably most) publications about OOP make the opposite assumption: the metaclass anti-condition holds. The books by Booch et al. [20] and Meyer [111] [W65] are two prominent examples.

The metaclass approach to OOP foundations

"Do without calculus" he told me.
I looked at him. He wasn't playing Dixie on the harmonica. The damn fool was serious.
Post Office, Charles Bukowski [W69] (adjusted)

This section describes the taxonomic position from which OOP foundations are approached. We have already specified the part of OOP which we mostly focus on. There are three another rules for the delimitation of our approach.

The *no-types* rule

This document provides a formal model of a specific part of object-oriented programming. This "specific part" is shown to be *central* for many significant languages. From the list of ten OO languages that were considered significant by Wikipedia in the middle of 2015, there are six whose core part is modelled using the "metaclass" investigation method: Python, Objective-C, Smalltalk, Java, Perl and Ruby.

Note that if PHP is added to the selection (and Java possibly removed) then the list will consist of exactly those of the 10 listed languages that are regarded as *dynamic*. This indicates that the models presented in the sequel focus on the dynamic part of OOP. For the non-dynamic part, the metaclass investigation method can be applied too (and we do apply it to the controversial border-case of Java), but the results tend to be (more) trivial and thus less useful.

Now let us use some magic in terminology. For the most part of OOP, a "dynamic language" is the same as a "dynamically-typed language". (The present author is not aware of any OO language that would constitute a counterexample and, simultaneously, could be regarded as significant.) Since we are interested in theoretic foundations of OOP, we interpret the term "dynamically-typed" in the context of (academic) computer science. According to StackOverflow [S3], the term reduces to "**untyped**". (Such a reduction can also be observed in the

field of object-oriented analysis and design, see [20c] where the term "typeless" is used.) Our approach to foundations of OOP therefore obeys the following rule:

Dispense with the notion of type.

That is, we do not hold for appropriate to include types at the very start while building the OOP foundations. A similar approach can be observed in the case of functional programming, whose foundational core is formed by the *untyped* lambda calculus.

Note: We will use the term "type" for a reference to particular concepts introduced by other parties (e.g. "types" as particular objects in Dylan, or "types" as non-object entities in VODAK).

The *no-calculus* rule

In addition to the "typed" vs "untyped" delimitation, there is another division line which we consider even more important, forming the main bisection. There are **two main approaches to OOP** given by two possible directions how to read the term "object-oriented programming":

(OO) Read the term from left to right so that "**object-oriented**" comes first and "programming" comes second.

(P) Read the term from right to left so that "**programming**" comes as first and "object-oriented" as second.

Accordingly, we will further speak about the **OO**-*approach* and the **P**-*approach*.

Analyzing the term linguistically, the "object-oriented" part is an adjective and "programming" is a substantive (noun). The P-approach just says that what is substantial on the term is – naturally – the substantive. It is therefore necessary to built OOP foundations by first establishing the "P" foundations, that is, the foundations of computer programming. This is achieved by developing a suitable model of computation, which in turn leads to the notion of a *calculus*, most notably the lambda calculus. We can therefore refer to the P-approach as to the ***calculus approach***. Since the lambda calculus forms the theoretical basis of functional programming, the P-approach attempts to establish OOP foundations by adding object-oriented features to (the foundations of) functional programming.

In contrast, the OO-approach to OOP regards the *adjective* as more important. This approach does not focus on code (computation) but focuses on data (state) instead. This is according to the object motto introduced by B. Meyer in his award-winning book *Object-Oriented Software Construction* [111] [𝕎65]. The motto reads:

Ask not first what the system does: Ask what it does it to!

Several other textbooks can be found in which this motto is considered to be a fundamental principle of object-oriented programming. Quoting from [117]: "The object-oriented way of thinking focuses on the data rather than on the procedures", or "[OOP] focuses on the objects rather than on procedures". Similarly, OOP "focuses on [the] data" according to [24] [17] [180], OOP "concentrate[s] on data" [94], in OOP, "emphasis is on the data" [159], OOP is characterized by the "data-first fashion" [153].

The present document applies the OO-approach. As a consequence, we will by default obey the following second rule:

Dispense with the notion of calculus.

Putting the two delimitation rules together, we provide a type-free and calculus-free view of foundations of object technology. This in particular means that the objective of this document is almost completely disjoint with the vast majority of OOP research performed so far. The *metaclass* term appears to be an exceptionally good indicator of the disjointness: there is no mention of metaclasses in either of AToO [1], FoOOPL [25] or NOOP [2]. The Metaclasses are taboo section provides even stronger evidence.

Object identity

We treat objects as abstract entities without any inherent structure. In the presented models, each object has its own *identity* independent on what can be called the object's "content" or "data" (or "state" or "attributes" or "value" …). We regard object identity as a crucial feature for building OOP foundations. We do not mean it to prevent the occurence of identityless objects altogether. Our approach just supports the view that such objects should only arise by *refinement of core* models. An example of such a refinement is presented in the incremental description of the Ruby object model [131] by the introduction of *immediate values*. For example, `Fixnum`s are given by their integer values. In the code on the right, objects *x* and *y* are identityless in the sense that they are uniquely given by their values (which are

```
class Fixnum
  attr_accessor :zz
end
x = 35; x.zz = '_X'
y = 45; y.zz = 'Y_'
puts (30+5).zz # _X
```

the integer numbers 35 and 45, respectively).

```
puts (40+5).zz # Y_
```

Booch et al. [20] list identity as one of the 3 characteristic properties of an object. This was also what Wikipedia said until 2013 [W8]. Many other publications regard object identity as an essential feature of object-orientedness [111] [90] [71] [93] [128] [67] [159] or even as "*the foundation of object oriented programming*" [121]. However, as with virtually any OOP concept, the essentiality of object identity remains disputable [41]. Research works on OOP foundations often do not mention the concept explicitly, leaving the reader guess whether a particular model supports object identity or not. This is also the case of AToO [1], FoOOPL [25] and NOOP [2]. In the impς calculus from AToO, objects are just "records of methods" (i.e. collections of certain labelled entities) and thus devoid of identity. For example, there is exactly one object without methods. The concept of identity is secluded from objects to entities called *store location*. In contrast, the *SOOL* language introduced in FoOOPL presumably supports object identity since objects are "*represented as […] references*", see the remarks at the end of the Investigation method subsection.

In the NOOP thesis, objects are defined by inductive construction as (certain) triples *(sc,fr,mr)* (where *sc* stands for a "signature closure", *fr* is a "field record" and *mr* is a "method record") and thus do not have identity in the above sense of independence of the object's content. This is rather in opposition to the thesis text which states that "*[class signature is] embedded inside objects as part of the identity of the objects*".

Summary of the rules

Here is the list of the four rules which characterize our approach. They are sort of Occam's razor for OOP foundations.

1. **Dispense with types.**
2. **Dispense with calculus.**
3. **Support object identity.**
4. **Assume metaclass pre-condition ("classes are objects").**

To the author's knowledge there is only one book about OOP foundations that follows all the rules: the book *Putting Metaclasses to Work* by I.R. Forman and S.H. Danforth [59]. In the sections to follow we take this book as a starting point for our investigations.

Classic definition of metaclasses

The classic definition of the term *metaclass* (less commonly written as *meta-class*) is expressed in the introductory sentence of the wikipedia article titled *Metaclass* [W30]. As of 2015 it reads:

> *In object-oriented programming, a metaclass is a class whose instances are classes.*

The sentence appeared first in the page version by Jason Orendorff in February 2006 [W31] and remained unchanged till at least 2015. Note the first part of the of the sentence: "*In object-oriented programming*". It indicates the term's default context as well as that there are other fields with a similar notion of metaclass. As of 2015, this can be observed on the wikipedia article titled *Metaclass (Semantic Web)* [W32] whose introductory sentence is the same as in [W30] except that "object-oriented programming" is replaced by "the Semantic Web". We will further regard the common main part of these sentences as the *classic definition* of the metaclass term, i.e.

(⊙) *A metaclass is a class whose instances are classes.*

This definition appeared in late 1970s in the field of knowledge representation [102]. Shortly after, the definition entered the world of object-oriented programming via the classic book *Smalltalk-80: The Language and Its Implementation* by Adele Goldberg and David Robson [66]. Quoting from page 76, "*A class whose instances are themselves classes is called a metaclass*". The same definition appears in the paper *Definition and Application of Metaclasses* [44] as the first sentence of the paper's abstract and in the book *Metaclasses and Their Application* ([93] page 12). The definition is also in accordance with the perhaps most authoritative reference to the concept of metaclasses – the book *Putting Metaclasses to Work* by Ira R. Forman and Scott H. Danforth from 1998 [59]. It is stated on page 15 as Definition 5: "*A metaclass is an object whose instances are*

classes". Here the word "object" is used instead of "class". We assume that this definition is meant to be equivalent with (⊙). The slight change in the wording has probably no other purpose than to stress out that metaclasses are subject to the uniformity principle *everything is an object*. Our assumption can be based on later publications by Ira R. Forman in which the metaclass concept is applied to the Java programming language [60] [61] as well as on his contributions to the Wikipedia talk page [𝕎₃₄].

In addition to (⊙) there is another classic definition which can be found in the Free On-Line Dictionary Of Computing [82a]:

(◈) *A metaclass is the class of a class.*

We will further refer to (⊙) as to the *primary* classic definition and to (◈) as to the *secondary* classic definition. If none of the two emphasized adjectives is used we mean *primary* by default.

Interestingly (and for reasons explained below), the secondary classic definition is preferred over (⊙) in the documentation of Python [150a]. This programming language might be nowadays seen as the default language for discussing metaclasses. As of 2015, more than 50% of the first 100 results returned by the Google search of *metaclass* are related to Python [𝔾₁].

Raw explanation

We can already provide a very informal explanation of the term. The notion of a metaclass as introduced by the classic definition above appears as a natural consequence of object-oriented environments that are (a) *class-based* (i.e. provide a strong support for explicit grouping of objects into classes) and (b) support the *uniformity principle* of objects and classes (i.e. classes are "first-class citizens" and are therefore among objects). In popular terms, it can be expressed by the following rules: [120]

i. Every object is an instance of a class.
ii. Classes are objects – being subject to the "everything is an object" principle.
iii. It follows that classes must also be instances of classes.
iv. A class whose instances are classes is called a *metaclass*.

(How simple!)

The F&D model via Python

Having introduced the wording(s) of the classic definition of metaclasses we can set out for deciphering its meaning. What does "*a class whose instances are classes*" mean in the context of object-oriented programming? We have already provided a raw explanation which refers to popular but vague phrases. In this section we provide a precise answer for the particular case of the Python programming language. Fortunately, we can perform this task by simultaneously describing the relevant parts of the object model as presented in the book [59] by Forman & Danforth. We will henceforth refer to the book as to the *F&D book*, or simply the *book*, and let *F&D model* be the (presumed) model which the book describes.

The correlation between the book and Python is no coincidence. Python's author(s) state the book as the main source of inspiration for the design of language's metaclass functionality [155] and call the book the "*bible of metaclasses*" [8]. (Considering the association of the metaclass term with Python as indicated by Google we can conclude that the book is the most authoritative reference to the metaclass concept.)

Instantiation graph

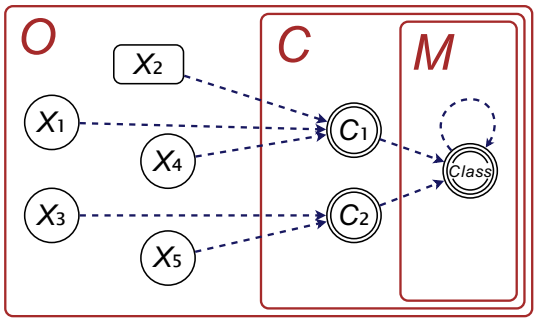
The metaclass framework which the book conveys is called "*monotonic reflective class-based model*". The model starts with a finite set O of *objects*. As the first structural constituent, a map between objects is introduced, denoted *class*, which assigns to each object x a unique object $class(x)$, called *the class of* x. The induced mathematic structure $(O, class)$ is a functional graph that is usually called the *instantiation graph* [27] [38] [118] (however, the book does not introduce this term). If x and y are objects such that $class(x) = y$ (i.e. y is the class of x) then x is said to be an *instance of* y. An object that appears as the class of an object is called a *class*.

The only constraint that is put to the functional graph $(O, class)$ (in addition to its finiteness) is that there is exactly one object, necessarily a class, that is involved in a cycle. Equivalently, the instantiantion graph forms an algebraic tree [131] and can thus be alternatively called an *instantiation tree* [28] [38] [93] (neither this term

occurs in the book). The root of the tree is denoted by *Class*. It follows that *Class* equals its class.

Having defined what it means for an object to be a class as well as what it means to be an instance of a given object we can apply the above mentioned Definition 5 from page 15: A *metaclass* is an object whose instances are classes.

The following picture is a transcription of Figure 2-6 from the same page. It shows an example of what has just been introduced (the book calls it a "*sample environment*"). There are eight objects, three of them are classes, and one of the classes is in addition a metaclass. The dashed arrows show the *class* map which, according to the introduced terminology, is coincident with the *instance-of* relation. The picture also shows the notation C and M for the sets of classes and metaclasses, respectively. Moreover, the book also introduces the term *ordinary object* (resp. *ordinary class*) for elements of the difference $O \setminus C$ (resp. $C \setminus M$) together with the drawing convention of single / double / triple boundary for ordinary objects / ordinary classes / metaclasses.



(Python 3)

```
Class = type
C1 = Class('',(),{})
C2 = Class('',(),{})
X1 = C1()
X2 = C1()
X4 = C1()
X3 = C2()
X5 = C2()
```

The correspondent Python code is shown on the right side. First, the built-in metaclass `type` is denoted by `Class`. Subsequently, the classes `C1` and `C2` are created by `Class` instantiation and the remaining five objects `X1` to `X5` are created by instantiating `C1` or `C2`. (In Python, class instantiation is performed by "calling" the class.) The *class* links between objects are established by the `__class__` attribute.

Explicit classes

In the above established definitions, "being a class" is a property of objects that is *derived* from the instantiation graph. Observe an undesired consequence: After executing the first three lines of the Python code (i.e. immediately after the creation of the `C2` class) both C_1 and C_2, having no instances yet, are recognized – by definition – as ordinary objects. It is therefore probable that we have misinterpreted the book's text and that the set C of classes is in fact defined explicitly. That is, the structure on which the Definition 5 is based is not just the instantiation graph *(O, class)* but a refined structure *(O, C, class)* such that O and *class* are as before and C is some given subset of O whose elements are called *classes* and such that *class(x)* belongs to C for every object x. In general, there can be classes without instances, as opposed to the previous definition.

The vacuous truth problem

Does the structure *(O, C, class)* establish correctness of Definition 5? Of course not. Observe first that it is not obvious how to interpret the Definition 5 regarding the logical quantifiers. Does "an object whose instances are classes" mean

(a) "an object whose *all* instances are classes", or
(b) "an object that has *at least one* instance that is a class", or
(c) the logical conjunction of (a) and (b)?

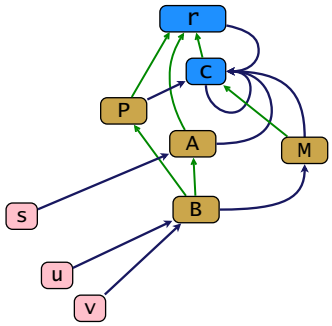
We can prevent ordinary objects from being recognized as metaclasses by vacuous truth according to (a) by replacing "object" with "class" (just like it occurs in the classic definition). But this is obviously not enough, the vacuous truth problem would remain for classes. An ordinary class would be recognized, immediately after its creation, as a metaclass according to (a). A metaclass that is not its own class would be first recognized as an ordinary class according to both (b) and (c).

We now came to the rather trivial conclusion: regardless of whether classes are defined explicitly and regardless of the exact quantification in "classes whose instances are classes"

> the instantiation graph is not sufficient to provide a rigorous definition of the metaclass term.

Note that this is contrary to what the classic definition of a metaclass suggests as well as contrary to what its raw explanation or the book suggest.

Inheritance graph



(Python 3)

```
r = object
c = type
M = c('', (c, ), {})
A = c('', (   ), {})
P = c('', (   ), {})
B = M('', (A,P), {})
s = A()
u = B()
v = B()
```

The apropriate structure for a rigorous definition of the notion of metaclass arises as a refinement of the instantiation graph by another relation between objects: the *inheritance* relation, which we denote by $\leq$. The resulting structure is therefore constituted by two relations between objects: the instantiation graph (the *class* map) and the inheritance relation $\leq$. As the symbol suggests, the $\leq$ relation is required to be a partial order. The reflexive transitive reduction of inheritance (which exists due to the finiteness of O and is unique due to antisymmetry of $\leq$) is a directed acyclic graph (DAG) called the *inheritance graph*. This is a subrelation of $\leq$ which we will refer to by $\prec$. The inheritance relation is in turn obtained from its inheritance graph as the reflexive transitive closure, so that there is a one-to-one correspondence between $\leq$ and $\prec$.

An example of such a structure is shown on the right, with blue arrows showing the instantiation graph and green arrows showing the inheritance graph. We abandoned the book's drawing conventions for distinguishing between objects from $O \setminus C$, $C \setminus M$ and M. Ordinary objects are drawn with pink background, blue color is used for "circular" classes – those that are involved in a cycle formed by a combination of blue and green links. The remaining classes are drawn with sandy brown background. There is no graphic distinction between metaclasses and ordinary classes.

Also the book introduces inheritance between objects, although with subtle differences. The book's definitory constituent for inheritance is a relation on the set C of classes refered to by *isSubclassOf* and termed *inheritance graph* (Postulate 6, page 20). The image of an object x under this relation is denoted by *parents(x)*. The reflexive transitive closure of *isSubclassOf* is denoted by *isDescendantOf*. (Presumably, the reflexive closure is taken just over the set C of classes.) It is required that *isSubclassOf* be a DAG but there is no requirement that this DAG be transitively reduced – there can be *transitivity edges* (cf. [50]). Equivalently, the set *parents(x)* can contain different objects that are comparable in *isDescendantOf*. The correspondence between ours and the book's relations is expressed as

$$(C, \prec) \subseteq \mathit{isSubclassOf} \subset \mathit{isDescendantOf} = (C, \leq).$$

That is, in the restriction to the set C of classes, (a) *isDescendantOf* equals the inheritance relation $\leq$ (however, the book does not introduce the term "inheritance relation"). And, (b) our inheritance graph equals the transitive reduction of *isSubclassOf*. This means that in general, *isSubclassOf* encodes more information than *isDescendantOf*. Interestingly, such a feature is also present in Python (where `x.__bases__` is the correspondent for *parents(x)*).

Python core structure

We now provide an axiomatic specification of the family of structures which we claim be suitable for a rigorous definition of metaclass in the context of the Python programming language. The axiomatization prescribes exactly which combinations of blue and links between objects (i.e. instantiation and inheritance graphs) are allowed. For convenience, we will from now on prefer the dot notation for the class map, i.e. we write *x.class* instead of *class(x)*. Such notation can also be observed in Python where the class of an object x can be referred to by `x.__class__`.

A *Python core structure* is a structure $(O, .class, \leq)$ where

- O is a set of *objects*,
- *.class* is the *class map* between objects (for an object x, *x.class* is *the class of x*),
- $\leq$ is the *inheritance* relation between objects.

If $x \leq y$ then x is said to be an *(inheritance) descendant* of y which in turn is an *(inheritance) ancestor* of x. We also let $<$ denote the reflexive reduction (the *strict inheritance*) of $\leq$, i.e. $x < y \leftrightarrow x \leq y$ and $x \neq y$. Similarly, symbols $\geq$, $>$, $\not\leq$, $\not<$, … have the obvious sense. For an object x we say that

- x is a *class* iff there is an object a such that $x.class \leq a.class.class$.

Objects that are not classes are called *ordinary* objects. The structure is subject to the following axioms:

(py~1) Inheritance, $\leq$, is a partial order on O.

(py~2) Ordinary objects have no strict descendants.

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

(py~4) There is a unique class, denoted r and called the *inheritance root*, that is a top class w.r.t. $\leq$.

(py~5) The one-element set $\{r.class\}$ is the only cycle of *.class*.

(py~6) The inheritance root r is the only strict ancestor of $r.class$.

(py~7) There are only finitely many objects.

The definition is self-sufficient – it contains all the terminology and notation that is required to state the axioms. Subsequently, we introduce further definitional extensions. As before, we let C be the set of classes. By definition, C equals the image of O under the composition $(.class) \circ (.class) \circ (\geq) \circ .class(-1)$ where *.class(-1)* denotes the inverse of *.class*. The first three axioms assert that C is subject to the following rules: (The first *four* axioms then assert that C is in fact *generated* by the rules.)

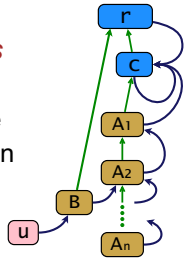
1. The class of every object is a class.
2. Every ancestor of a class is a class. (An equivalent of (py~2)).
3. Every descendant of a class is a class. (A consequence of (py~3)).

Axiom (py~1) states the reflexivity ($x \leq x$ for every object x), transitivity ($x \leq y \leq z$ implies $x \leq z$) and antisymmetry ($x \leq y \leq x$ implies $x = y$) conditions for inheritance. Note that the domain of $\leq$ is O not just C so that each object is its own descendant and ancestor. This approach is unusual. In most contexts, inheritance is only defined between classes. If (py~1) and (py~3) are assumed then axiom (py~2) just states that $(O, \leq)$ is just the reflexive closure of the restriction to C.

Axiom (py~3) asserts the condition that is stated as the last book's postulate (Postulate 10, page 42) and accounts for the "monotonic" adjective in "monotonic reflective class-based model". We will therefore refer to this axiom as to the *monotonicity condition*.

Axiom (py~4) is the standard condition of a single-rooted inheritance hierarchy of classes. Also the book introduces the inheritance root, naming it *Object*. However, there is a note that "*There is no postulate for introducing Object …*". In Python, the inheritance root is named `object`.

Assuming finiteness of O, axiom (py~5), asserts that the *.class* map forms a tree rooted at $r.class$. We call the distinguished object $r.class$ (necessarily a class) the *instantiation root* and denote it by c. The example on the right shows that there is no limit as to the depth of the tree. As already mentioned earlier, the tree structure of the instantiation graph is one of the first things the book assumes, with *Class* denoting the root. After the introduction of inheritance and *Object* as the top class, the book also asserts that *Class* is the class of *Object*.




```
r = object
c = type
A0 = c
A1 = A0('',(A0,),{})
A2 = A1('',(A1,),{})
A3 = A2('',(A2,),{})
B = A2('',(),{})
u = B()
```

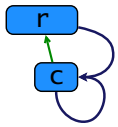
For the explanation of the remaining axioms we introduce some additional definitions:

- Let $\in$ be the composition of *.class* with $\leq$ and call this relation *(object) membership*.
- Let $\prec$ be the reflexive transitive reduction of $\leq$ (the *inheritance graph*).

We also let $\in^i$ denote the i-th composition of $\in$ with itself, for a positive natural i. Similarly, let *.class(i)* be the i-th composition of *.class* with itself so that e.g. $x.class(2)$ is a shorthand for $x.class.class$.

Let us call an object x *circular* if it appears in a cycle of $\in$, i.e. if $x \in^i x$ for some positive natural i. Proposition B2 below shows that the already described axioms together with the last axiom assert that circular objects are exactly the ancestors of c (including c itself – by our definition of "ancestors").

Axiom (py~6) postulates that there are exactly two circular objects, necessarily r and c. This asserts minimality and non-degeneracy of the Python's built-in circular structure so that it looks exactly like what is shown by the diagram on the right. The structure is simultaneously the minimum Python core structure. Using the notation for the inheritance graph, the axiom can be expressed as $c \prec r$.



The book imposes this condition too and calls the above 2-element circular structure the "minimal

environment". (To be precise, by not requiring the inheritance graph to be transitively reduced the book does not rule out additional ancestors of $\underline{c}$, but we assume that such objects are meant to be disallowed.) Similar diagram is provided in Figure 2-8, with the names *Class* and *Object* for $\underline{c}$ and $\underline{r}$ respectively. (The *Object* class is however erroneously depicted as a metaclass, with a triple boundary.)

Proposition A: Assume axioms (py~1)–(py~4). Then for every object x,

1. $x \le \underline{r} \quad \leftrightarrow \quad x \in \underline{c} \quad \leftrightarrow \quad x \in \underline{C}$ (where $\underline{c} = \underline{r}.class$),
2. $x \le \underline{c} \quad \leftrightarrow \quad x \le y.class$ for some $y \in \underline{C}$.

Proof:

1. Let us refer to the equivalences by (i)↔(ii)↔(iii). By monotonicity of $.class$, if $x \le \underline{r}$ then $x.class \le \underline{r}.class$ and also $x.class(2) \le \underline{r}.class(2)$. Since $\underline{r}.class = \underline{c}$ this shows (i)→(ii)→(iii). The closing implication (i)←(iii) follows by (py~4).
2. For the → direction put $y = \underline{r}$. The reverse direction follows by: $y \le \underline{r} \to y.class \le \underline{r}.class$. □

Proposition B: Assume axioms (py~1)–(py~5).

1. For every objects x, y and every natural $i > 0$,
   $x \in^i y \quad \leftrightarrow \quad x.class(i) \le y$.
2. Assume in addition finiteness of $\underline{O}$, i.e. (py~7). Then for every object x and every natural $i > 0$,
   $x \in^i x \quad \leftrightarrow \quad \underline{c} \le x$.

Proof:

1. The ← direction is trivial. The opposite direction follows by induction over i.
2. Let us make the proposition's assumption and let x be an object and i a positivie natural number. To prove the ← part of the equivalence, assume $\underline{c} \le x$. Then x is a class and thus $\underline{c} \le x \le \underline{r}$. By monotonicity, $\underline{c}.class \le x.class \le \underline{r}.class$. Since $\underline{c}.class = \underline{c}$ by (py~5) and $\underline{r}.class = \underline{c}$ by definition of $\underline{c}$ it follows that $x.class = \underline{c} \le x$ and thus $x \in x$ so that also $x \in^i x$.

   To prove the opposite direction, assume $x \in^i x$, i.e. $x.class(i) \le x$. By monotonicity of $.class$, there is a decreasing sequence

   $$x \ge x.class(i) \ge x.class(2 \cdot i) \ge x.class(3 \cdot i) \ge \cdots$$

   of objects. Since there are only finitely many objects it follows that $x.class(k \cdot i) = x.class(k \cdot i).class(i)$ for some natural k. That is, $x.class(k \cdot i)$ belongs to a cycle of the instantiation graph. By (py~5), there is exactly one cycle of $.class$, namely $\{\underline{c}\}$. It follows that $x.class(k \cdot i) = \underline{c}$ and therefore $x \ge \underline{c}$. □

Corollary: Let $\mathcal{S} = (\underline{O}, .class, \le)$ be a Python core structure.

1. The structure $(\underline{O} \setminus \{\underline{r}, \underline{c}\}, (.class) \cup (\prec))$ is a finite DAG.
2. The following are equivalent:
   a. $\underline{O} \ne \{\underline{r}, \underline{c}\}$.
   b. There is an object that is minimal in $(.class) \cup (\prec)$, i.e. such that it has no instances and no strict descendants.
3. For every object x that is minimal in $(.class) \cup (\prec)$, the restriction of $\mathcal{S}$ to $\underline{O} \setminus \{x\}$ is a Python core structure.

Disallowed structures

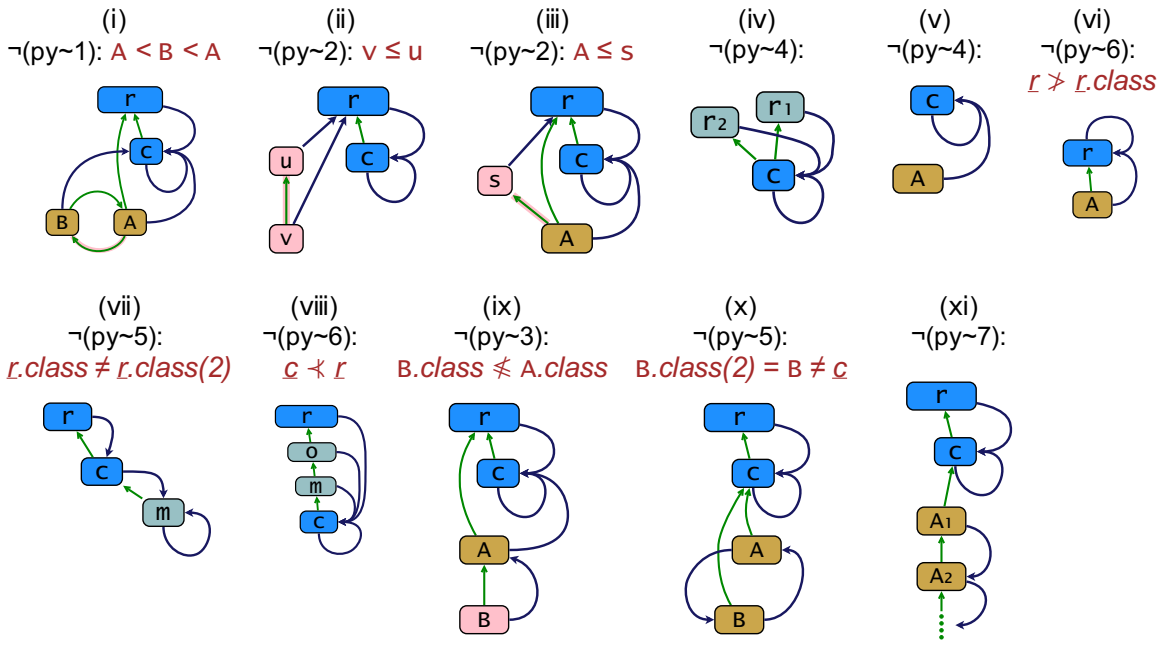
The following examples of disallowed structures show that there is no redundancy in the axioms. For the sake of rigorosity we first handle a dependency between the axiom wordings: both (py~5) and (py~6) refer to $\underline{r}$ whose existence is postulated by (py~4). Let us (temporarily) denote R the set of all classes that are *maximal* in $\le$. Then (py~5) and (py~6) can be formulated without a reference to $\underline{r}$ as follows:

(py~5') $R.class$ is a one-element set that is the only cycle of $.class$.

(py~6') $R \ne R.class$ and for every objects x, y, if $R.class \ni x < y$ then $y \in R$.

With this alteration, each example violates just the indicated condition. (Note that the alteration is only required to provide an "exclusive" violation of (py~4). Other axioms do not need that.)

The first example (i) shows a cycle in inheritance. (As a consequence, $\prec$ is not unique – we could have drawn a green arrow from B to $\underline{r}$ instead of from A to $\underline{r}$.) Examples (ii) and (iii) show ordinary objects involved in strict inheritance.



Example (vii) shows the built-in structure of the LOOPS programming language. Diagram (viii) shows the structure between the four built-in circular classes of the Ruby programming language (as of Ruby 1.9 and later).

Example (ix) demonstrates a violation of the monotonicity condition: $A \le B$ but $A.class \not\le B.class$. Python detects the monotonicity break upon the B class creation, as shown by the following code:

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

Another example of monotonicity break detection is provided in a later subsection. The last example (xi) shows that Proposition B2 would not hold without the assumption of finiteness ($A_k \in A_k \not\succ c$ for each k).

Metaclass

Now we have axiomatized Python core structures as finite mathematical structures that arise from the object model described in the book *Putting metaclasses to work*. The book is considered to be the "bible of metaclasses" according to prominents of the Python community. Among all formal languages, the Python programming language provides by far the strongest connection to the "metaclass" term, according to Google. It is therefore the right time to introduce the promised rigorous definition.

> Metaclasses are exactly the descendants of c.

That is, there is a built-in circular metaclass c (named `type` in Python, resp. *Class* in the book). An object is a metaclass if and only if it is an inheritance descendant of the built-in metaclass.

As we mentioned earlier, the book does not state that the boxed sentence is the definition of a metaclass. However, the implication $x \le c \to x \in M$ appears as Theorem 4 on page 30. In Python (using the language's terminology), *metaclasses are exactly the subclasses of the* `type` *class*, including the `type` class itself. Such a delimitation can be found in many publications [4] [13] [75] [103] [172] [188].

We can observe that the axioms assert that

metaclasses are classes all of whose instances are classes.

This statement should however be interpreted as an implication: If x is a metaclass then (a) x is a class and (b) all instances of x are classes. The converse does not hold due to the vacuous truth problem.

Remark: The phrase "***all*** *of whose instances are classes*" can be found in [170].

Substructures

The $(O, .class, \le)$ signature which we used for the axiomatization of Python core structures is a minimum one.

The *.class* map cannot be derived from the ≤ relation and vice versa. However, this conciseness has a drawback: using this signature, the family of Python core structures is not closed w.r.t. taking substructures. It can be observed that for a Python core structure $\mathcal{S} = (O, .class, \leq)$, a substructure $\mathcal{S}' = (O', .class, \leq)$ of $\mathcal{S}$ is a Python core structure if and only if $r \in O'$. That is, for a set X of objects, the (relational) restriction of $\mathcal{S}$ to X is a Python core structure iff

$$X.class \cup \{r\} \subseteq X$$

where $X.class$ is the image of X under *.class*. The desired closedness w.r.t. taking substructures is therefore achieved by adding r into the signature.

The additional condition of X being an upset in ≤ results in a stronger closure property which can be expressed as either of

(a) $X.class \cup X.\updownarrow \subseteq X$ or (b) $X.class \cup X.parents \subseteq X$

where $X.\updownarrow$ and $X.parents$ are the respective images of X under ≤ and $\prec$. (However, it is necessary to assume in addition that X is non-empty.) In terms of directed graphs, X is closed iff it is closed w.r.t. out-going edges in both the instantiation graph and the inheritance graph. Such a condition allows for a decomposition of Python core structures according to the following lemma.

Decomposition lemma: Let $\mathcal{S} = (O, .class, \leq)$ be a structure where *.class* is a map on a set O and ≤ is a relation on O. Let X, Y be subsets of O such that both are subject to the (a) condition above and $X \cup Y = O$. Then the following are equivalent:

i. The restrictions of $\mathcal{S}$ to X and Y are both Python core structures.
ii. $\mathcal{S}$ itself is a Python core structure.

(By induction, the lemma can be formulated with X_1, …, X_n instead of X, Y.) □

We can use this lemma for the investigation of object models in particular programming languages. We focus only on small *.class*- and *.parents*-closed portions of the underlying data structure. A violation of any axiom in a substructure yields a violation in the whole structure. If there are no violations and the tested portions provide a reasonably generic coverage then we can make positive judgments about the whole structure.

Potential instances

It can be observed that if $\mathcal{S}_1 = (O_1 .class, \leq)$ and $\mathcal{S}_2 = (O_2, .class, \leq)$ are Python core structures such that $\mathcal{S}_1$ is a substructure of $\mathcal{S}_2$ (we also say that $\mathcal{S}_2$ is an *extension* of $\mathcal{S}_2$) then for every object from O_1, "being a class" as well as "being a metaclass" is the same in $\mathcal{S}_1$ and $\mathcal{S}_2$. Moreover, for every $\mathcal{S}_1$ there exists an extension $\mathcal{S}_2$ in which every class has at least one instance. (For an instanceless class q, consider an attachment of a new object x such that $x.class = q$ and $x.parents = \{r\}$ in the case that q is a metaclass and $x.parents = \emptyset$ in the case that q is an ordinary class.)

The vacuous truth problem can therefore be alleviated by considering possible extensions of a given Python core structure $\mathcal{S} = (O, .class, \leq)$ by additional objects. An object x is a class iff it has *potential* instances, i.e. x has at least one instance in some extension $\mathcal{S}'$ of $\mathcal{S}$. Similarly, an object x is a metaclass iff its potential instances are classes. [126] This leads to an adjustment of the classic definition:

> Metaclasses are classes all of whose **potential** instances are classes.

Note however that even with this definition, the vacuous truth problem can still arise in the context of languages that support "static classes", i.e. classes that are explicitly declared to be uninstantiable.

Terminology and notation in a nutshell

The following list provides a summary of terminology and notation that have been introduced for Python core structures, together with new terminology and notation for images and preimages of ≤, ∈ and $\prec$. Each of x and y denote an object. The uppercase X symbol is used for a set of objects. The ⍟ symbol is used for relational composition, with the inscribed triangle indicating the left-to-right direction for the interpretation.

O	the set of (all) *objects*
.class	the *class map* between objects, $x.class$ is the *class of* x. As a relation, *.class* is called the *instantiation graph* or *instantiation tree* or also *direct membership*.
≤	the *inheritance* relation, a partial order on O

$<$	the *strict inheritance* relation, the reflexive reduction of $\leq$
$\prec$	the *inheritance graph*, the reflexive transitive reduction of $\leq$ to immediate pairs
$\in$	the *(object) membership* relation between objects, equals $(.class) \circ (\leq)$, i.e. $x \in y \leftrightarrow x.class \leq y$.
$x.\uparrow$	the set of *(inheritance) ancestors* of x, shorthand for $\{x\}.\uparrow$
$x.\downarrow$	the set of *(inheritance) descendants* of x, shorthand for $\{x\}.\downarrow$
$X.\uparrow$ ($X.\downarrow$)	the image (resp. pre-image) of X under $\leq$
$x.parents$	the set of *(inheritance) parents* of x, shorthand for $\{x\}.parents$
$X.parents$	the image of X under $\prec$
$x.\in$	the set of *containers* of x, shorthand for $\{x\}.\in$
$x.\ni$	the set of *members* of x, shorthand for $\{x\}.\ni$
$X.\in$ ($X.\ni$)	the image (resp. pre-image) of X under $\in$
$\underline{C}$	the set of *classes*, $\underline{C} = \underline{O}.\in.\downarrow = \underline{r}.\downarrow = \underline{c}.\ni$.
$\underline{O} \setminus \underline{C}$	the set of *ordinary objects*
$\underline{r}$	the *inheritance root*, the top class (named *Object* in the book, `object` in Python)
$\underline{c}$	the *instantiation root*, the top metaclass (named *Class* in the book, `type` in Python), the class of $\underline{r}$ and the only fixed point of $.class$
$\underline{c}.\downarrow$	the set of *metaclasses*. The book denotes the set by $\underline{M}$ but we (try to) save this symbol for other purposes. Classes that are not metaclasses are *ordinary classes*.

We also use $X.class$ to denote the image of X under $.class$. The reversed symbols $\geq$, $>$, $\succ$ and $\ni$ are used to denote the inverse of the respective relation. For a natural $i > 0$ we let $\in^i$ be the *i-th power* of $\in$, so that e.g. $\in^2$ equals $(\in) \circ (\in)$. (We let $\in^0$ be undefined yet.) Similarly, for every natural i, $.class(i)$ is the i-th power of $.class$, with $.class(0)$ being the identity map on $\underline{O}$. We also let $.class(-i)$ be the inverse of $.class(i)$ for every natural i. In particular, $.class(-1)$ is the inverse of $.class$.

The *instance-of* ambiguity

When we started the description of the object model by Forman & Danforth we have introduced three terms for the *class* constituent (later altered to *.class*). Even four terms: (a) the *class map*, (b)(1) the *instantiation graph* (b)(2) *instantiation tree* and (c) the *instance-of* relation. Subsequently, we preferred using (b) over (c). We mostly reserved the (c) term just for phrases like "*(all) instances of y*" in order to refer to the set of (all) objects x such that $x.class = y$. This is because in Python, "*instance-of*" is used for $\in$ (instead of for $.class$). A similar mismatch between the book's term and the name of the Python's introspection method occurs for the inheritance relation, as shown by the following table.

Our terminology		Our expression	Book's expression	Python expression
x is a *member of* y	x is an *instance of* y	$x \in y$	x *isA* y	`isinstance(x,y)`
x is a *descendant of* y		$x \leq y$	x *isDescendantOf* y	`issubclass(x,y)`
x is a *direct instance of* y		$x.class = y$	x *isInstanceOf* y	`x.__class__ == y`
x is a *direct (strict) descendant of* y		$x \prec y$	x *isSubclassOf* y	`x in {y}.__bases__`

The table also shows that we resolve the ambiguity in favor of Python. In particular, from now on by "*instance-of*" we mean the same as "*member-of*", i.e. the object membership relation $\in$. The $.class$ map is then the *direct instance-of* relation.

It remains to adjust appropriately all the previously made statements that used the term "instance". In fact, there are just three such statements which can be expressed as follows (x denotes an object):

- x is minimal in $(.class) \cup (\prec)$ $\leftrightarrow$ x has no instances and no strict descendants.
- x is a metaclass $\rightarrow$ all instances of x are classes.
- x is a metaclass $\leftrightarrow$ x is a class and all potential instances of x are classes.

Because "instance (of)" meant what we now call "direct instance (of)" we could adjust the above statements by adding the "direct" adjective. However, we can observe that this is not necessary: All the statements remain valid with the new meaning of "instance (of)" as "member (of)". In particular, the definition of metaclass via potential instances is independent on whether we mean just direct instances or allow indirect ones.

In what follows we will prefer using "member" instead of "instance".

Recursive definition

We now provide a second axiomatization of Python core structures, equivalent to the first one, but using different definitory constituents. The definition is recursive, showing how the structure can be incrementally constructed.

A *Python core structure* is a structure $\mathcal{S} = (O, .class, \prec, r)$ where

- O is set of *objects*,
- $.class$ is the *class map* between objects,
- $\prec$ is the *inheritance graph*, a relation between objects,
- r is the *inheritance root*, a distinguished object.

For a set X of objects, we write $X.parents$ for the image of X under $\prec$. Similarly, $X.class$ is the image of X under $.class$. Let $\preceq$ be the reflective transitive closure of $\prec$ called the *inheritance* relation using the same ancestor/descendant terminology as before. Descendants of r form the set C of *classes*, the remaining objects are *ordinary*. Let $c = r.class$ be the *instantiation root*. Descendants of c are *metaclasses*.

The structure is then subject to the following conditions:

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

(pÿ~2) All objects from $O.parents \cup O.class$ are classes.

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

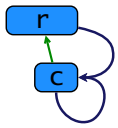
(pÿ~4) If $O = O.parents \cup O.class$ then $O = \{r, c\} \neq \{r\}$.

(pÿ~5) For every ordinary object x, $x.class$ is not a metaclass.

(pÿ~6) The restriction of $\mathcal{S}$ to $O \setminus \{x\}$ is a Python core structure for every $x \in O \setminus (O.parents \cup O.class)$.

(pÿ~7) The set O of objects is finite.

Observe that $O.parents \cup O.class$ is the set of objects that are not minimal w.r.t. $(\prec) \cup (.class)$. Condition (pÿ~4) axiomatizes the minimum structure. It is easily verified that such a structure exists and equals the Python's built-in circular structure as described before (for convenience also shown on the right). It is the only structure that cannot be further reduced since every object is a target of either a blue or a green arrow. Axioms (pÿ~6) and (pÿ~7) assert that any Python core structure can be built from the minimum 2-element structure by adding new objects, one by one.



Let us by (𝕒) refer to the definition of a Python core structure according to the first axiomatization (py~1)–(py~7) and by (𝕓) to the recursive definition axiomatized by (pÿ~1)–(pÿ~7).

Proposition: The (𝕒) and (𝕓) definitions of a Python core structure are equivalent.

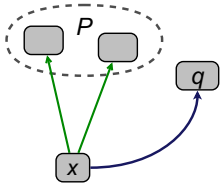
Proof:

Assume first that $\mathcal{S} = (O, .class, \preceq)$ is a Python core structure according to (𝕒) and let $\prec$, r, c, $.parents$ and C be derived from $\mathcal{S}$ according to (𝕒). We have to prove that $\mathcal{S}' = (O, .class, \prec, r)$ satisfies (pÿ~1)–(pÿ~7). Obviously, $\preceq$ in $\mathcal{S}'$ is the same as $\preceq$ in $\mathcal{S}$ and so are all of $.parents$, c and C (the $r.\mathrm{I} = C$ equality is stated by Proposition A1). Axioms (pÿ~1), (pÿ~3) and (pÿ~7) are the same as in (𝕒). The $O.parents \subseteq C$ inclusion follows from (py~2). The $O.class \subseteq C$ follows by definition of C in (𝕒). Condition (pÿ~4) follows by Corollary 2. Condition (pÿ~5) follows by $C.class.\mathrm{I} = c.\mathrm{I}$ (Proposition A2). Condition (pÿ~6) follows by Corollary 3 using induction.

To show the opposite direction, assume that $\mathcal{S} = (O, .class, \prec, r)$ satisfies (pÿ~1)–(pÿ~7) and let $\preceq$, c, $.parents$ and C be derived from $\mathcal{S}$ according to (𝕓). We have to prove that $\mathcal{S}' = (O, .class, \preceq)$ satisfies (py~1)–(py~7). Axioms (py~1), (py~3) and (py~7) follow by exact correspondence. If $O = O.parents \cup O.class$ then by (pÿ~4) $\mathcal{S}$ is the minimum 2-object structure and (py~∗) are easily checked. Otherwise, let $x \in O \setminus (O.parents \cup O.class)$ and let $\mathcal{S}_0$ (resp. $\mathcal{S}'_0$) be the restriction of $\mathcal{S}$ (resp. of $\mathcal{S}'$) to $O \setminus \{x\}$. By (pÿ~6) and the finiteness of O we can apply an induction argument and assume that $\mathcal{S}'_0$ is a Python core structure according to (𝕒). Now axioms (py~5) and (py~6) follow by minimality of x in $(.class) \cup (\prec)$. Axiom (py~2) follows by $x.parents \subseteq r.\mathrm{I}$ and the equivalence: y is a class in $\mathcal{S}'_0$ $\leftrightarrow$ y is a class in $\mathcal{S}'$ and $y \neq x$. Finally, to show (py~4) assume that x is a class in $\mathcal{S}'$, i.e. $x.class \preceq a.class(2)$ for some object a. Since a is necessarily different from x it follows by Proposition A2 that $x.class \preceq c$. Consequently, $x \preceq r$ by (pÿ~5). □

New object creation

The recursive definition says that any Python core structure can be incrementally built from its restriction to $\{r,c\}$ by attaching new objects, one by one. New object creation is usually called *class instantiation* although different terms are used for the case when the newly created object is a class. A new object is attached by one blue link to the class q that is being instantiated and by zero or more green links to the requested inheritance parents. Some sets of target objects are disallowed by axioms. Let an *attachment request* be a pair (q,P) such that



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

so that $q = x.class$ and $P = x.parents$ for the new object x. An attachment request (q,P) 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 $x \not\leq y$.

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

(3) Assert (pÿ~3): For every x from P, $q \leq x.class$. (The monotonicity condition.)

(4) Assert (pÿ~5): If P is empty then $q \not\leq \underline{c}$.

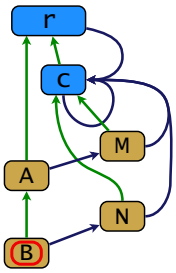
Axioms of a Python core structure are preserved after the attachment of x if and only if (1)–(4) are asserted.

Python provides several ways how to create new objects but for practical reasons there is no single built-in method capable of creating both ordinary objects as well as classes. The above described new object creation can be artifically assembled using the `object.__new__` and `type.__new__` built-in methods as shown by the `new` function defined in the following code.

```
import inspect; isclass = inspect.isclass; r = object; c = type
def new(q,P=()):
  if   not isclass(q):      assert(False)
  elif not issubclass(q,c): assert(not P); return r.__new__(q)
  else:                     assert(P);     return c.__new__(q,'',P,{})
```

The `if`/`elif`/`else` distinctions correspond to q being ordinary object / ordinary class / metaclass. The assertion of (1) is left out. For q being a metaclass, conditions (2) and (3) are checked by the Python's built-in method.

The following diagram demonstrates the incremental creation of a Python core structure. During the last execution of `new`, a violation of the monotonicity condition is detected (reported as a "metaclass conflict").



¬(pÿ~3): $B.class \not\leq A.class$

(Python 3)

```
r = object
c = type
M = new(c,(c,))
N = new(c,(c,))
A = new(M,(r,))
B = new(N,(A,)) # TypeError: metaclass conflict
```

Axiomatization via ϵ

By definition, any Python core structure on a set $\underline{O}$ of objects is uniquely given by the pair $(.class, \leq)$ of relations on $\underline{O}$ and also by $(.class, \prec)$ due to the $(\leq)\leftrightarrow(\prec)$ correspondence. But we can also replace $.class$ by ϵ in these pairs, as a consequence of the following observation: In a Python core structure,

$.class$ is the smallest relation R (w.r.t. set inclusion) such that $R \circledcirc (\leq) = (\epsilon)$.

Equivalently, for every object x, $x.class$ is the unique container of x that is least in inheritance, written as $x.\epsilon = x.class.\mathrm{I}$.

The box on the right shows an axiomatization of Python core structures in the $(\underline{O}, \epsilon, \leq, \underline{r}, \underline{c})$ signature. The two distinguished objects $\underline{r}$ and $\underline{c}$ are added for convenience. Axiom (pϵ~6) asserts the acyclicity of ϵ outside $\{\underline{r},\underline{c}\}$ ($\epsilon^+/\ni^+$ denote the transitive closure of $\epsilon/\ni$ and X^2 is a shorthand for the cartesian product $X \times X$). The existence of the least container $x.class$ for every object x is stated as axiom (pϵ~8).

(pϵ~1) $\leq$ is a partial order.

(pϵ~2) (a) $(\epsilon) \circledcirc (\leq) \subseteq (\epsilon)$. (subsumption of ϵ)
(b) $(\leq) \circledcirc (\epsilon) \subseteq (\epsilon)$. (monotonicity of ϵ)

(pϵ~3) $\underline{O}.\epsilon \leq \underline{r}$. (Every container is from $\underline{r}.\mathrm{I}\downarrow$.)

(pϵ~4) $\underline{O} = \underline{O}.\ni$. (Every object is a member.)

(pϵ~5) Objects from $\underline{O} \setminus \underline{r}.\mathrm{I}\downarrow$ are minimal.

This condition is also known as *single classification* [62]. (The absence of the condition results in a *multiple classification*.)

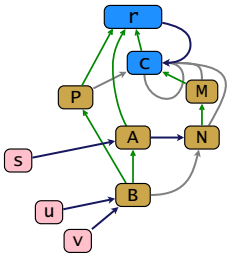
Documents [135] and [136] define a slight generalization, called *canonical primary structures*, allowing infinitely many ordinary objects and also allowing additional ancestors of *c*, just like in the (viii) example of disallowed structures.

(pϵ~6) $(\epsilon^+) \cap (\ni^+) = \{r,c\}^2 \neq \{r\}^2$.
(pϵ~7) $c.\ni \leq r$. (metalevel condition)
(pϵ~8) $(\epsilon) = (.class) \circledcirc (\leq)$ for a map *.class*.
(pϵ~9) *O* is finite.

Class map reduction

By observation, the class map *.class* can be further reduced to a submap that equals the smallest relation *R* such that

$(\leq) \circledcirc R \circledcirc (\leq) = (\epsilon)$.



Such a reduction is shown in the diagram on the right. Gray color indicates the difference *(.class) \ (.cl̇ass)* where *.cl̇ass* denotes the reduced submap. However, although this leads to drawing less arrows in a diagram, we will mostly prefer to show the full *.class* map explicitly. The diagram also demonstrates an asymmetry in the use of the "direct" adjective in "direct member". If *x* is a direct member of *y* (i.e. *x* is a direct instance of *y*) then *y* is a minimal container of *x* but *x* is not necessarily a maximal member of *y*. (Consider the ϵ relation between {A,B} and {M,N}. Then B is a direct member of N in spite of having a strict ancestor A that is also a member of N.)

Various expressions of *C* and *M*

In the definition of Python core structures we have introduced the set *C* of classes as the image of the set *O* of objects under the *(.class) ⊙ (.class) ⊙ (≥) ⊙ .class(-1)* relation. Using the subsequently introduced notation, the relation can be more concisely expressed as *.class(2) ⊙ (∋)*. The set *M* of metaclasses is by definition equal the pre-image of {*c*} (the only cycle of *.class*) under ≤. There are of course other possible expressions of *C* and *M* as shown in the following table which emphasizes the similarity between the definition of classes and metaclasses.

Terminology (The set of:)	Notation	Equal to the image of				Top object	Classifier
		O under		*C* under	*M* under		
classes	*C*	*.class(2) ⊙ (∋)*	*(ϵ) ⊙ (≥)*	*(≤) ⊙ (≥)*	*.class(-1)*	*r*	*c*
metaclasses	*M*	*.class(2) ⊙ (≥)*		*(.class) ⊙ (≥)*	*(≥)*	*c*	—

In the last column, the term *classifier* means an object *x* such that a given set *X* of objects contains exactly the members of *x*, i.e. $X = x.\ni$. The *c* class is the classifier for classes. If the name `Class` is used for *c* (as is the case of the book but not the case of Python) we could write "*classes are exactly the `Class`es*", using the classifier naming convention introduced later. There is no classifier for metaclasses in Python core structures – as opposed to the case of LOOPS [19], where "*metaclasses are exactly the `Metaclass`es*".

Using the notation for images and pre-images under ≤ and ϵ we can express the implicit equalities of the table as follows:

$C = O.class(2).\ni = O.\epsilon.\downarrow = C.\updownarrow.\downarrow = M.class(-1) = r.\downarrow = c.\ni$,
$M = O.class(2).\downarrow = C.class.\downarrow = M.\downarrow = c.\downarrow$.

Note that the $M = C.class.\downarrow$ equality can be read as "*metaclasses are the classes of classes or the descendants thereof*" which yields a refinement of the secondary classic definition (⟐).

The book's axiomatization

For the sake of completeness we also present what can be regarded as a distillation of the book's axiomatization of Python core structures.

An *fd-core structure* is a structure $\mathcal{S} = (O, .class, \prec, Class)$ where

- O is set of *objects*,
- $.class$ is a map between objects,
- $\prec$ is a relation between objects (*isSubclassOf*),
- $Class$ is a distinguished object.

Let $\preceq$ be the reflective transitive closure of $\prec$ and let isA be the composition of $.class$ with $\preceq$. For an object x we say that

$(*)$ x is a *class* iff $x\ isA\ Class$.

(fd~1) $\prec$ is a directed acyclic graph (DAG).
(fd~2) Only classes can be related by $\prec$.
(fd~3) The $.class$ map is monotone w.r.t. $\preceq$.
(fd~4) There is a class $Object$ that is a top class w.r.t. $\preceq$.
(fd~5) The set $\{Class\}$ is the only cycle of $.class$.
(fd~6) $Class \prec Object$.
(fd~7) There are only finitely many objects.

The structure is subject to the axioms in the box on the right. The additional condition of $\prec$ being transitively reduced results in an axiomatization that is equivalent to that of Python core structures (with the $c = Class$ and $r = Object$ identifications).

As already pointed out before, there is no single list in the book that would put the conditions together. Moreover, there is no direct statement of the $(*)$ equivalence for the delimitation of classes. Instead, the equivalence is established as a consequence of the following:

1. An object x "responds" to a given "method" "defined" exclusively in a class y iff $x\ isA\ y$.
2. An object x "responds" to the `makeInstance` "method" iff x is a class.
3. The $Class$ class is the only class that "defines" the `makeInstance` "method".

As the quotation marks indicate, the statements resort to a (yet undefined) finer structure which is however unnecessary for the delimitation of classes or metaclasses.

Summary

We have provided a rigorous answer to "what is a metaclass?" for the context of Python, according to the book *Putting metaclasses to work*, the "bible of metaclasses". We have followed the book's path by delimiting metaclasses from a given set O of objects according to an abstract structure between objects. We have shown that the $(O, .class)$ structure is not sufficient to rigorously define metaclasses due to the vacuous truth problem. The abstract structures which we regard as the right definitory setting for the metaclass term have *two* constituents, both being relations between objects:

- $.class$, the *instantiation* graph, and
- $\preceq$, the *inheritance* relation, reducible to $\prec$, the *inheritance graph*.

The structures can be simply visualized by diagrams with a 2-colored set of arrows between nodes. Blue arrows stand for instantiation and green arrows stand for direct inheritance. Axioms (py~1)–(py~7) specify exactly which combinations of blue and green arrows are allowed, yielding the family of *Python core structures* $(O, .class, \preceq)$.

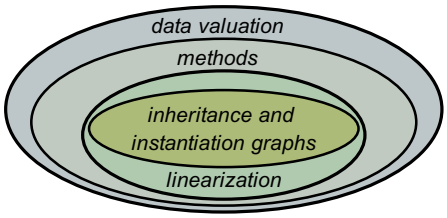
The set M of *metaclasses* arises by definitional extension, together with the set C of *classes*. There is also a distinguished class r and a distinguished metaclass c. These two objects form a circular substructure, $r.class = c.class = c \prec r$. We have established convenient notation $.\!\!\uparrow$ / $.\!\!\downarrow$ for images and pre-images under the inheritance relation $\preceq$. Moreover, we introduced a special symbol $\in$ for the composition $(.class) \circ (\preceq)$ and call this relation *(object) membership*. (In contrast to $\preceq$, we use the $\in$ symbol directly for the correspondent image map.)

We have defined metaclasses as the descendants of c, which we defined to be the class of r, which we defined to be the top of the set C of classes, which we defined to be the image of O under $(.class) \circ (.class) \circ (\ni)$. We could have defined the set M directly just like C – as the image of O under $(.class) \circ (.class) \circ (\geq)$. The primary classic definition $(\circledcirc)$ appears as a statement about metaclasses: *if an object x is a metaclass then all its instances are classes*. The reverse implication can be established by resorting to *potential* instances, i.e. considering the instances of x in any extension of the given Python core structure. This yields three equivalent definitions which are summarized in the following list:

1. *Metaclasses are the descendants of the built-in metaclass c.*
2. *Metaclasses are the classes of classes and their descendants.*
3. *Metaclasses are the classes all of whose potential instances are classes.*

The F&D model, part 2: Linearization

In the previous section we described the core part of the object model presented in the book *Putting metaclasses to work*. We focused on those aspects of the model that are necessary and sufficient for a rigorous definition of the *metaclass* term. This resulted in a family of structures $(O, .class, \le)$. In the sections to follow we want to explain the utility of these core structures. This is accomplished by describing further layers of the F&D model. Starting with Python core structures which axiomatize the possible combinations of inheritance and instantiation graphs, each new layer provides a refinement of the previous one. The diagram on the right provides a brief scheme. We will label the corresponding four families of structures by $\mathbf{FD}_1$ to $\mathbf{FD}_4$. In particular, an $\mathbf{FD}_1$ structure is just another name for a Python core structure.



The next family of structures just adds a single definitory constituent that induces a linear order on the set $x.\updownarrow$ of inheritance ancestors for each object x. That is, for each object x, there is a list $x.ancs$ of all ancestors of x, called the *linearization* of x (this term is introduced in the book). This order is in accordance with inheritance – it is a linear extension of $(x.\updownarrow, \le)$. As a consequence, in the case of *single inheritance* – when each object has at most one inheritance parent (i.e. $(C, \le)$ is a tree) – the linearization is uniquely given. A proper refinement of $\mathbf{FD}_1$ structures only takes place in the case of *multiple inheritance*.

$\mathbf{FD}_2$ structure

An ***FD_2 structure*** is an $\mathbf{FD}_1$ structure $(O, .class, \le)$ equipped with $\le_{mro}$ where

- $\le_{mro}$ is a relation between ordered pairs of objects called *method resolution order*, shortly *MRO*.

Thus, $\le_{mro}$ can be regarded as a quaternary relation on O. Since the set $O \times O$ of couples of objects is disjoint from the set O of objects (by a standard assumption of mathematical structures) we can drop the $_{mro}$ suffix and write $(x,y) \le (a,b)$ instead of $(x,y) \le_{mro} (a,b)$. We say that objects a and b are *MRO-compatible* if

$(a,x) \le (a,y) \leftrightarrow (b,x) \le (b,y)$ for every pair x, y of common ancestors of a and b.

The structure is subject to the following axioms:

(fd_2~1) The MRO relation, $\le_{mro}$, is a partial order on the inheritance relation $(O, \le)$.

(fd_2~2) Components of MRO correspond to linearly ordered ancestor sets, i.e.

(a) if $(a,b) \le (x,y)$ then $a = x$, and (b) either $(a,b) \le (a,c)$ or $(a,c) \le (a,b)$

for every (a,b), (a,c), (x,y) from $(O, \le)$.

(fd_2~3) For every objects a, b, if $a \le b$ then: (a) $(a,a) \le (a,b)$, (b) a and b are MRO-compatible.

Note that since $\le$ is the domain of $\le_{mro}$, an $\mathbf{FD}_2$ structure is given by $(O, .class, \le_{mro})$. The $\le_{mro}$ constituent is in a one-to-one correspondence with the linearization map $.ancs$ (which is a function from the set O of objects to the set O^* of finite lists of objects) given by

$(a,x) \le (a,y)$ ↔ there are natural i, j such that $a.ancs[i] = x$ and $a.ancs[j] = y$ and $i \le j$.

Condition (fd_2~3)(b) (i.e. MRO-compatibility of any pair of objects comparable in inheritance) can be expressed as:

if $a \le b$ then $b.ancs$ is a sublist of $a.ancs$.

This is known as the *linearization monotonicity* condition [48] [49] [50] [167].

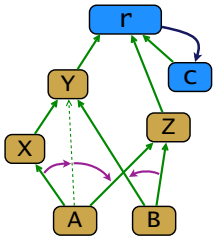
The diagram on the right shows an $\mathbf{FD}_2$ structure with the three magenta arrows indicating (a part of) MRO:

$(A,X) < (A,Y) < (A,Z)$ and $(B,Z) < (B,Y)$.

(The instantiation graph is shown in the reduction to $.class$ so that there is just one blue arrow). The whole MRO arises as the unique minimum extension of the shown reduction (such that the axioms are satisfied). In Python, the `.__mro__` attribute of classes can be used for the introspection of $.ancs$:

```
print(r.__mro__ == (r,)
  and c.__mro__ == (c,r)
```

```
    and Y.__mro__ == (Y,r)
    and Z.__mro__ == (Z,r)
    and X.__mro__ == (X,Y,r)
    and A.__mro__ == (A,X,Y,Z,r)
    and B.__mro__ == (B,Z,Y,r)) # True
```




```
r = object
c = type
Y = c('',(     ),{})
Z = c('',(     ),{})
X = c('',(Y,   ),{})
A = c('',(X,Y,Z),{})
B = c('',(  Z,Y),{})
```

For demonstration purpose, the sample structure contains a pair of classes `A`, `B` that are not MRO-compatible. These classes have no potential common descendants. In Python, an attempt to create such a descendant e.g. by `c('',(A,B),{})` or by `c('',(B,A),{})` results in an exception (`Cannot create a consistent method resolution order`).

Consistency with inheritance

An important consequence of (fd₂~3) is that MRO is consistent with the inheritance relation. That is, for every object a, the ancestor list $a.ancs$ encodes a linear extension of $(a.\updownarrow, \leq)$. Equivalently, for every objects a, x, y,

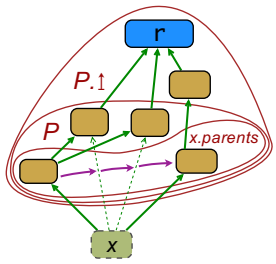
$(*)$ $\quad a \leq x \leq y \quad \rightarrow \quad (a,x) \leq (a,y)$.

Proof: Assume $a \leq x \leq y$. By (fd₂~3)(b), a and x are MRO-compatible and thus $(a,x) \leq (a,y) \leftrightarrow (x,x) \leq (x,y)$. The right side of "↔" is then asserted by (fd₂~3)(a). □

In case of single inheritance, $(a.\updownarrow, \leq)$ is a linear order for every a and thus equal to its linear extension. The $(*)$ implication turns into an equivalence. Observe also that $(*)$ can be used as a replacement of (fd₂~3)(a).

C3 linearization

Recall that in **FD**$_1$ structures, a new object x creation was determined by an attachment request (q,P) where q is the requested class of x and P is the requested *set* of inheritance parents of x. In an **FD**$_2$ structure, the transition request has to be refined to also encompass the requested extension of MRO (that is, the requested value of $x.ancs$). The refinement is established by ordering of P into a *list*, called *local precedence order*, which we denote ps. Simultaneously, ps is allowed to contain more than the set $x.parents$ of the requested direct strict ancestors. (Consequently, $x.parents = mins_{\leq}(P)$, i.e. the set of requested inheritance parents of x contains exactly the classes that are minimal in $(P, \leq)$.)



Given the requested local precedence order ps, let $\prec_{ps}$ be the relation on $P.\updownarrow$ defined by

$a \prec_{ps} b \quad \leftrightarrow \quad$ either $(p,a) \prec (p,b)$ for some $p \in P$ (i.e. $a = ps[i].ancs[j]$ and $b = ps[i].ancs[j+1]$ for some i, j)
or $a = ps[i]$ and $b = ps[i+1]$ for some i.

(By $\ell[i]$ we refer to the i-th member of the list ℓ, starting with the 0-th member. Moreover, $[x]$ refers to the list containing x as the only member, and $+$ is used for list concatenation.) The attachment request in an **FD**$_2$ structure can then be expressed as (q,ps). The request is acceptable iff $(q, mins_{\leq}(P))$ is an acceptable attachment request in **FD**$_1$ and, in addition, $\prec_{ps}$ is acyclic (i.e. a DAG). For an acceptable attachment request (q,ps) the linearization $x.ancs$ of the newly created object x equals $[x] + as$ where as is a list that encodes a total order of $P.\updownarrow$. This total order is a linear extension of $\prec_{ps}$, also known as a topological sort of $\prec_{ps}$.

We can of course specify as purely in terms of lists. Let n be the length of ps (i.e. n equals the number of elements of P). Then as is a list containing exactly the members of lists $ps[i].ancs$, $i < n$, without duplicates, (equivalently as is a *permutation* of $P.\updownarrow$), and such that each of the lists ps and $ps[i].ancs$, $i < n$, is a sublist of as.

In general, there are multiple linear extensions of $\prec_{ps}$ (i.e. there can be multiple lists as for a given list ps). A

deterministic prescription is provided below. Given an **FD$_2$** structure $\mathcal{S} = (O, .class, \leq, \leq_{mro})$, let $c3$ be a partial map between finite lists of objects such that $c3(ps)$ is a list that encodes a linear extension of $\prec_{ps}$, whenever it exists, as follows:

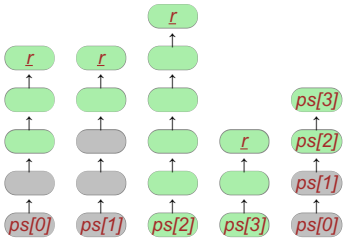
$c3(ps)$ is undefined if $\prec_{ps}$ is not a DAG (i.e. there is a cycle in $\prec_{ps}$), else
$c3(ps) = c3(ps, [\,])$ where the 2-argument version of $c3$ is defined recursively by
$c3(ps,as) = as$ if $\mathrm{rng}(as) = P.\!\uparrow$ (i.e. if as contains all elements of $P.\!\uparrow$), else
$c3(ps,as+[u])$ where u is the unique class such that

i. u is minimal in $(P.\!\uparrow \setminus as, \prec_{ps})$, and
ii. if v is another class minimal in $(P.\!\uparrow \setminus as, \prec_{ps})$ and i (resp. j) is the least index such that $ps[i] \leq u$ (resp. $ps[j] \leq v$) then $i \leq j$.

(We use $\mathrm{rng}(\ell)$ for the *range* of a list ℓ, i.e. the (unordered) set of all members of ℓ and abbreviate $X \setminus \mathrm{rng}(\ell)$ to $X \setminus \ell$.)

The above prescription can be expressed as an algorithm for incremental computation of $c3(ps)$. Given the ps list, let as be an initial part of the to-be-computed list $c3(ps)$. Start with as being an empty list. Then append repeatedly an element of $P.\!\uparrow$ at the end of as until it contains all elements of $P.\!\uparrow$. (The as list is then the resulting value of $c3(ps)$.) In each step, choose the appended member to be a minimal element u of the $(P.\!\uparrow \setminus as, \prec_{ps})$ digraph (i.e. the digraph is formed as the restriction of $\prec_{ps}$ to the set $P.\!\uparrow \setminus as$ of the not-yet-sorted classes). If there is no such u then $\prec_{ps}$ is not acyclic and therefore $c3(ps)$ is undefined. If there are more possible u's then choose that one from $ps[i].\!\uparrow$ (equivalently, from the $ps[i].ancs$ list) with i the smallest possible.

It can be observed that this is just a standard topological sort algorithm (see e.g. [72], Algorithm 3.2.1) endowed with a deterministic choice of minimum element in each step (the choice is given by condition (ii)). Alternatively, the $c3$ map can be expressed in terms of an operation on lists of lists. (This is in fact the standard description [50].) The minimality of u is expressed as the u's occurrence in a head of a list to be processed and, simultaneously, the u's non-occurrence in a tail of any list. The diagram on the right shows an example in which ps has four members, so that the input list of lists equals $[ps[0].ancs, \ldots, ps[3].ancs, ps]$. The classes to be processed (after a partial evaluation of $c3(ps)$) are displayed in green.



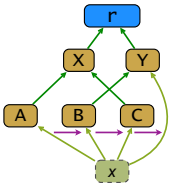
Why "C3"?

The linearization $[x] + c3(ps)$ of a new class x using the above described $c3$ map is called *C3 linearization*, according to the original paper by Barrett et al. [12] in which the map has been introduced. The "C3" name refers to the following three consistency properties which are guaranteed by the linearization:

(1) monotonicity,
(2) preservation of the requested local precedence order,
(3) consistency with the extended precedence graph.

(The claim that these are indeed the three properties that the authors had in mind is contained in the first paragraph of a subsection titled "C3 - A linearization consistent with the EPG". Note that the consistency with the inheritance graph is not listed since it is implied by monotonicity – this is in contrast to [50].) The first 2 properties are, in conjunction, equivalent to the statement that $c3(ps)$ encodes a linear extension of $\prec_{ps}$, whenever it exists. The third property states that $c3(ps)$ encodes also a linear extension of the acyclic part of $\lhd_{ps}$ where $\lhd_{ps}$ is another relation between classes, called the *extended precedence graph* (induced by ps) and defined by

$$a \lhd_{ps} b \quad \leftrightarrow \quad ps[i] \leq a \text{ and } ps[j] \leq b \text{ for some } i < j.$$



By the *acyclic part* of $\lhd_{ps}$ is meant the difference $(\lhd_{ps}) \setminus (\rhd_{ps})$ where $\rhd_{ps}$ is the inverse of $\lhd_{ps}$. Thus, (3) can be expressed as

$$a \lhd_{ps} b \not\lhd_{ps} a \quad \rightarrow \quad a = as[i] \text{ and } b = as[j] \text{ for some } i < j$$

where as is a list such that $x.ancs = [x] + as$. The diagram on the right (with $ps = [A,B,C,Y]$) shows that even condition (3) does not ensure uniqueness of as. In addition to $c3(ps) = [A,B,C,X,Y,r]$, also $[A,B,C,Y,X,r]$ is "C3". (Apply $X \lhd_{ps} Y \lhd_{ps} X$.)

The F&D model, part 3: Methods

Until now, the structures used to describe parts of the F&D model (namely $\mathbf{FD}_1$-structures and $\mathbf{FD}_2$-structures) were single-sorted. In each structure, there is a single sort O of objects and the structure is constituted by relations between objects. In contrast, the next refinement is multi-sorted.

$\mathbf{FD}_3$ structure

An ***FD**$_3$ structure* is an $\mathbf{FD}_2$ structure equipped with $(\Sigma, \Pi, .\pi, .m_\Theta())$, where

- Σ is the set of *names*,
- Π is the set of *(anonymous) methods*,
- $.\pi$ is a partial function from O to Π called *method valuation*,
- $.m_\Theta()$ is a partial function from $O \times \Sigma$ to Π called the *own method map*.

We regard methods and names as abstract entities, just like we do with objects. If (x,s) is in the domain of $.m_\Theta()$ then $x.m_\Theta(s)$ refers to the value of $.m_\Theta()$ at (x,s). The structure is subject to the following axioms:

Domains →

O	Σ	Π
owner	name	method

(fd$_3$~1) If $x.m_\Theta(s)$ is defined then x is a class. (That is, ordinary objects do not own methods.)

(fd$_3$~2) If $x.\pi$ is defined then x is an ordinary object. (Classes are not valuated by methods.)

(fd$_3$~3) The $.m_\Theta()$ map is finite. (Every object owns only finitely many named methods.)

(fd$_3$~4) The Π set is finite. (There are only finitely many methods.)

The whole signature of an $\mathbf{FD}_3$ structure is $(O, \Sigma, \Pi, .class, \leq, \leq_{mro}, .\pi, .m_\Theta())$. The last constituent can be regarded as a relational database table with three columns according to the diagram above right. Underline in owner and name indicates a primary key.

The $.\pi$ map is an auxiliary constituent which allows some ordinary objects to serve as *handles* of methods. In a potential further refinement of the F&D model, there can be a distinguished class `F` that serves as a classifier for the domain of $.\pi$, as does the class named "`function`" in Python.

Named method ownership, provision and respondence

We call elements of $\Sigma \times \Pi$ *named methods*. An $\mathbf{FD}_3$ structure can be regarded as an $\mathbf{FD}_2$ structure equipped with ownership of named methods. Subsequently, we define partial functions $.\omega()$, $.m_H()$ and $.m_R()$ on $O \times \Sigma$ by

$x.\omega(s) = y \quad \leftrightarrow \quad y = x.ancs[i]$ for the smallest i such that $x.ancs[i].m_\Theta(s)$ is defined,
$x.m_H(s) = x.\omega(s).m_\Theta(s)$,
$x.m_R(s) = x.class.m_H(s)$.

Note that whereas $.\omega()$ is a subrelation of $O \times \Sigma \times O$, all of $.m_\Theta()$, $.m_H()$ and $.m_R()$ are subrelations of $O \times \Sigma \times \Pi$. That is, for each $*$ from $\{\Theta,H,R\}$, $.m_*()$ is a relation between objects and named methods. We call these relations *(named method) ownership*, *provision* and *respondence*. The following table summarizes the terminology and also provides some alternatives.

Terminology ($.m_*()$ as a map $O \times \Sigma \rightharpoonup \Pi$)	Expression	Terminology ($.m_*()$ as a relation between O and $\Sigma \times \Pi$)	
own method map	$x.m_\Theta(s) = f$	x *owns* (s,f)	*(ownership)*
inherited / *providing* method map	$x.m_H(s) = f$	x *inherits* / *provides* (s,f)	*(provision)*
received / *responding* method map	$x.m_R(s) = f$	x *receives* / *responds* to $(*)$ (s,f)	*(respondence)*

We will also use the adjectives from the first column in combination with the later introduced *method arrow* term for the triples from the respectives sets $.m_*()$. For example, $x.m_\Theta(s) = f$ can be read as "(x,s,f) is an own method arrow". By *strict provision* we mean the difference between provision and ownership.

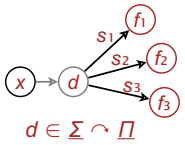
Note: $(*)$ The statement "x responds to (s,f)" should be understood as an abbreviation of "x responds to an s-named **message** by (invoking) f".

Observations:

1. If x owns (s,f) then it also inherits (s,f). (That is, $.m_\Theta() \subseteq .m_H()$ – ownership implies provision.)
2. $x.\omega(s) = x \quad \leftrightarrow \quad x$ owns (s,f) for some method f.
3. If x inherits (s,f) then $x.\omega(s)$ owns (s,f).
4. If x inherits (s,f) and $x \le y \le x.\omega(s)$ then y inherits (s,f).
5. All named methods owned by a class x are received by all x's direct instances.

Method dictionaries

By currying, each of $.m_*()$ can be viewed not only as a partial map from $\underline{O} \times \underline{\Sigma}$ to $\underline{\Pi}$ (i.e. as an element of $(\underline{O} \times \underline{\Sigma}) \rightharpoonup \underline{\Pi}$) but also as a total map from objects to partial maps from names to methods (i.e. as an element of $\underline{O} \to (\underline{\Sigma} \rightharpoonup \underline{\Pi})$). That is, each of $.m_*()$ assigns each object x a dictionary $(*)$ $d = x.m_*()$ whose keys are names and whose values are methods. The inherited method map, $.m_H()$, can then be expressed via dictionary merge. For every object x,



$x.m_H() = a_0 \mathbin{◐} a_1 \mathbin{◐} \cdots \mathbin{◐} a_{n-1}$ (the dictionary of inherited methods of x)

where n is the number of x's inheritance ancestors, $a_i = x.ancs[i].m_\Theta()$ (that is, a_i is a dictionary with own methods of x's i-th ancestor in the method resolution order), and $◐$ is the "left merge" operator on dictionaries defined by

$a \mathbin{◐} b = c$ where c is the unique dictionary such that $c[s] = a[s]$ if $a[s]$ is defined else
$= b[s]$ if $b[s]$ is defined else $c[s]$ is undefined.

(For a dictionary d, we let $d[s]$ refer to the value of d at s.)

Observations:

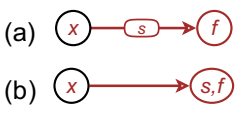
1. There is no need for parentheses in $a_0 \mathbin{◐} a_1 \mathbin{◐} \cdots \mathbin{◐} a_{n-1}$ since $◐$ is associative.
2. For every ordinary object x, both $x.m_\Theta()$ and $x.m_H()$ are empty. (A consequence of $(fd_3{\sim}1)$.)
3. If x and y are direct instances of the same class then the dictionaries $x.m_R()$ and $y.m_R()$ are identical.

Note: $(*)$ We use the term "dictionary" as a synonym to "finite functional relation " – a dictionary is a finite set of key-value pairs with unique keys. Exactly the same definition is introduced in the book (Definition 1, page 4). In contrast, Python's dictionaries are associative arrays. They are objects used to represent functional relations, but without a one-to-one correspondence. In the code on the right, after the execution of the first 2 lines, p and q are non-identical objects representing the same functional relations. After the p = q assignment, p and q become identical.

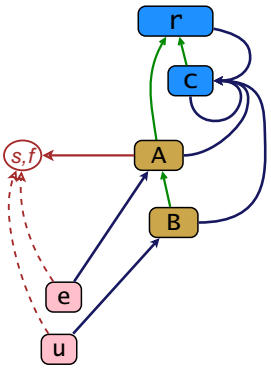
```
p = {'x' : 1, 'y' : 2}
q = {'x' : 1, 'y' : 2}
print (p is q) # False
print (p == q) # True
p = q
print (p is q) # True
```

Method arrows

Let us call the triples from $\underline{O} \times \underline{\Sigma} \times \underline{\Pi}$ *method arrows*. For a method arrow $p = (x,s,f)$, x is the *source object*, s is the *name* and f is the *target method* of p. Alternatively, we consider method arrows to be pairs from $\underline{O} \times (\underline{\Sigma} \times \underline{\Pi})$. In this case, if $p = (x,(s,f))$ is a method arrow then x is the *source object* and (s,f) is the *target named method*. These two alternatives pose two ways for the diagrammatization as shown on the right. In the (b) case the arrows can be used to display the relations of ownership, provision and respondence between objects and named methods. We will further prefer the (a) case for diagrams with only one source object x and the (b) case for diagrams of whole $\mathbf{FD}_3$ structures. (In the (a) case we also use the less comfort way of arrow labelling as already shown for a method dictionary in the previous subsection.)



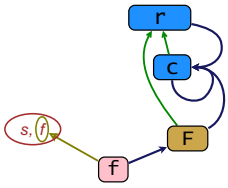
The diagram below shows a sample $\mathbf{FD}_3$ structure with just one own method arrow ($.m_\Theta() = \{(A,s,f)\}$). The induced relation of named method respondence is indicated by dashed lines (there are exactly 2 received method arrows). Since A has exactly one strict descendant, there is also one arrow in the strict provision relation $.m_H() \setminus .m_\Theta()$ (namely (B,s,f)) which is however not displayed.



(Python 3)

```
def f(self): print ('@-' + str(id(self)))
s = 'm'
class A:    pass
class B(A): pass
e = A()
u = B()
setattr(A,s,f)              # A.m = f
em = getattr(e,s); em()     # em = e.m; @-9828912
um = getattr(u,s); um()     # um = u.m; @-9829008
_m = getattr(e,s)           # _m = e.m
print (em)          # <bound method A.f of (...)
print (_m is em)            # False
print (em.__self__ is e)    # True
print (em.__func__ is f)    # True
```

The code on the right shows that Python performs automatic reification of received method arrows – `em` and `um` are objects that represent the received method arrows *(e,s,f)* and *(u,s,f)*, respectively. (∗) The `_m` object is another reification of *(e,s,f)*, with a different identity than `em`. The last two lines show introspection facilities for getting at the source and target of these arrows.



Observe also that in the Python code, `f` denotes an object, whereas in the diagram, *f* (in obligue font) denotes a method. Here we assume that $f = \mathtt{f}.\pi$, i.e. *f* is the method valuation of the `f` object. The diagram above could be refined by the diagram on the right to reflect this correspondence. The olive arrow indicates the $.\pi$ map, and `F` denotes the built-in class (named `'function'`) that is the class of `f`.

Note: (∗) To be precise, Python does not perform reification of *(x,s,f)* but just of *(x,f)* – the *s* name (equal to `'m'`) is lost. This is indicated by the textual representation of `em` which (erroneously) refers to `A.f` instead of to `A.m`.

Transitions

For the first two layers we only introduced *incremental* transitions. New structures arose from old by adding objects together with acceptable extensions of the definitory relations *.class*, ≤ and $\leq_{\mathrm{mro}}$. In contrast, we let transitions of $\mathbf{FD}_3$-structures be non-incremental. We refer to the correspondence between $.m_\Theta()$ and a database table as shown within the definition and allow arbitrary operation of upsert and deletion of a single method arrow.

The following table shows the corresponding Python and SQL expressions. We use "own_method_arrows" as the name for the database table.

	In Python	In SQL
upsert	`setattr(`*x*`,`*s*`,`*f*`)`	`MERGE INTO own_method_arrows VALUES (`*x*`,`*s*`,`*f*`)`
delete	`delattr(`*x*`,`*s*`)`	`DELETE FROM own_method_arrows WHERE owner = `*x*` AND name = `*s*

Python of course supports more convenient syntax for the modification of $.m_\Theta()$. Most programs only use incremental transitions – that is, typically there are no deletions and most upserts are inserts, not updates. Moreover, the inserts are mostly expressed as parts of "class definitions". The standard way to insert *(A,'m',f)* from our example looks like this:

```
class A:
  def m(self): print (…)
```

That is, the own method dictionary $\mathtt{A}.m_\Theta()$ is created directly after the `A` class creation.

As for the (auxiliary) method valuation map $.\pi$, we only allow incremental transitions – the value of $x.\pi$ (or its definedness) for an old object *x* cannot be changed.

The purpose of core structure

The sample structure for method arrows demonstrates the main utility of **FD**$_2$ structures:

> ***providing a uniform mechanism for the sharing of named methods***.

Each object is a potential receiver of named methods. Typically, methods are shared since most methods apply to more than one object. Sharing methods leads to grouping of methods which is performed via ownership by classes. The method resolution order $\leq_{mro}$ between classes (with the implied inheritance relation $\leq$ in particular) provides a structure for incremental specification of groups of named methods.

Alternatively, core structures $(O, .class, \leq_{mro})$ serve as ***generators*** of received method arrows. These implicit entities are generated from owned method arrows (the explicit entities) and are then used for method invocation.

Method invocation

A *method invocation* can be regarded as a transition specified by the code that is associated to a method. The transition request is a triple (x,s,ℓ) where

- x is the receiver object on which the method is invoked,
- s is the name of the method to be invoked, and
- $\ell = a_1,\ldots,a_n$ is the possibly empty list of arguments with which the method is invoked.

In the book, these three constituents are listed on page 59, although with a different terminology. As the canonical *expression* for a method invocation we will regard the form

$x.\langle s\rangle(a_1,\ldots,a_n)$

where $\langle s\rangle$ stands for the dereferenced value so that e.g. if s equals `'m'` then $\langle s\rangle$ stands for $x.m$. The method that is invoked by the above expression equals $x.m_R(s)$. The process of evaluation of $x.m_R(s)$ (i.e., given an object x and a name s, determine the method f such that x responds to (s,f)) is known as *method resolution*.

Returned value

In an analogy to HTTP requests or database queries there is a "response" in a narrower sense to transition requests (x,s,ℓ) from the previous subsection. The expression $x.\langle s\rangle(a_1,\ldots,a_n)$ is *evaluated* to an a object y that is said to be the *returned value* of the method invocation. The y object equals the value of the expression from the f's code that is evaluated as the last one. Transitions made by evaluation of $x.\langle s\rangle(a_1,\ldots,a_n)$ can be regarded as "side effects". In some cases, there are no side effects. Such methods define (partial) algebraic operations on the set O of objects, i.e. maps from $(n+1)$-tuples of objects to objects.

Methods that have no side effects and take no arguments have a prominent position. Every such method f defines a partial map between objects. If x responds to (s,f) and $x.\langle s\rangle()$ evaluates to y then the pair (s,y) can be regarded as an *attribute* of x. To support this semantics, we let $x.\langle s\rangle$ be a shorthand for $x.\langle s\rangle()$. As a canonical example of use, if f is a method that returns the class of the object upon which f is invoked and the inheritance root r owns the named method $('class',f)$ then

$x.class$ evaluates to $x.class$ for every object x,

provided that the method is not overridden, i.e. $.m_\Theta('class')$ is only defined for r.

Execution context

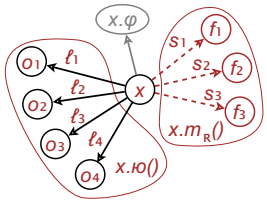
There is a refinement of **FD**$_3$ structures that is closely associated with method invocation: *execution context*. We will not introduce a formal structure, but assume that part of the refinement is a distinguished object, referred to by *self*. If a code is executed that is associated with a method invoked upon object x then *self* equals x. That is, if $x.m_R(s) = f$ then execution of $x.\langle s\rangle(a_1,\ldots,a_n)$ results in a composition of 3 transitions (we omit handling of $a_1,\ldots,a_n$):

1. Set *self* to x.
2. Evaluate the code of f.

3. Reset *self* to the value that it was assigned to before (1).

Interpretation of the *f* method code is dependent of *self*.

The F&D model, part 4: Data valuation



In the last layer of the introduced 4-layer model we let objects be equipped with "real" data. There will be two kinds of data: (a) an optional hidden link $x \mapsto x.\varphi$ to a value from some commonly understood domain like boolean values **TRUE** and **FALSE**, strings, integers, or even non-scalar values like e.g. pairs of integers, and (b) a dictionary *x.ю()* whose values are objects and whose keys are *instance variable names*. Together with the received method dictionary $x.m_R()$ this constitutes bundling of data with behaviour, as shown on the right.

Unlike with named methods, there is no support for sharing of named objects from *x.ю()*. That is, there is no distinction between ownership and respondence for *.ю()*. In contrast to the book, we simplify the situation and allow arbitrary dictionaries *x.ю()*, similarly to Python. In particular, we do not require compatibility of instance variables – *x.ю()* and *y.ю()* can have different domain (different set of keys) even if *x.class = y.class*.

The $(x.ю(), x.\varphi, x.m_R())$ structure shown by the diagram constitutes what can be called *perceived object properties*. Typically, the syntax of a programming language is maximally tuned for the access and manipulation of these properties. A relational database counterpart of the diagram looks as follows:

Instance variable valuation

O	Σ	O
owner	name	value

Object data valuation

O	Φ
owner	value

Named method respondence

O	Σ	Π
receiver	name	method

Note that whereas the first two tables are *base tables*, the third table is a *database-view*. Observe also that we have omitted the method valuation *.π* which could be defined for an ordinary object *x*. The diagram above should therefore be merged with the single arrow diagram on the right. (Such a merge is performed in the Object neighborhood subsection.) Accordingly, there should be one more base table which would store *.π*.

FD_4 structure

An ***FD_4 structure*** is an **FD_3 structure** equipped with $(\Phi, .\varphi, .ю())$, where

- Φ is the set of *values*, alternatively, the *value domain*,
- $.\varphi$ is a partial function from O to Φ called *object data valuation*,
- *.ю()* is a partial function from $O \times \Sigma$ to O called the *instance variable valuation*.

The structure is subject to the following axioms:

(fd4~1) The *.φ* map can only be defined for ordinary objects. (That is, classes are not *.φ*-valued.)

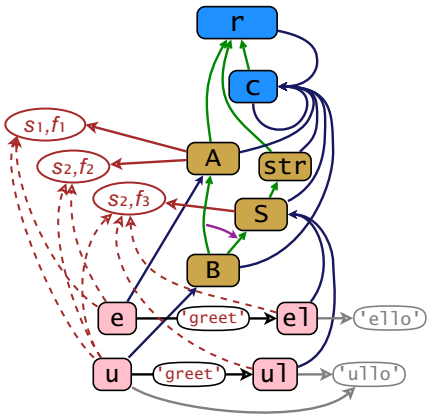
(fd4~2) The *.ю()* map is finite.

Pairs *(x,s)* from the domain of *.ю()* are called *instance variables*. If *p = (x,s)* is an instance variable then *s* is the *name of p* and *x.ю(s)* is the *value of p*. Moreover, *p* is said to be an instance variable *of x*.

Similarly to the method valuation *.π*, data valued objects *x* are meant to serve as handles of *x.φ*. In Python, there are built-in classes like `str` or `int` whose direct instances *x* are in a one-to-one correspondence with *x.φ*. The code on the right shows that Python 3 allows indirect instances of `str` which are obviously *.φ*-valuated by strings but are not subject to such a one-to-one correspondence.

```
class S(str): pass
s1 = 'abc'; t1 = S('abc')
s2 = 'abc'; t2 = S('abc')
print (s1 is s2) # True
print (t1 is t2) # False
```

The diagram below shows an example of an **FD**$_4$ structure. Note that the sample structure from the previous section arises as a substructure of the reduct to an **FD**$_3$ structure. (Here we must insist on the $\leq$ relation in the signature. Substructurality w.r.t. $\prec$ is not satisfied since there is an additional parent for `B`.)



(Python 3)

```
s1 = '__init__'
s2 = 'm'
def f1(self,greet): self.greet = greet
def f2(self): print ('h-' + self.greet)
def f3(self): print ('-h' + self)
class A:      pass
class S(str): pass
class B(A,S): pass
setattr (A,s1,f1)      # A.__init__ = f1
setattr (A,s2,f2)      # A.m = f2
setattr (S,s2,f3)      # S.m = f3
el = S('ello'); el.m() # -hello
ul = S('ullo'); ul.m() # -hullo
e = A(el); e.m()       # h-ello
u = B(ul); u.m()       # h-ullo
print (el,ul,u)        # ello ullo ullo
```

Instance variable valuation, $.\omega()$, is shown by labelled black arrows, whereas object valuation, $.\varphi$, is displayed by gray arrows. There are exactly 2 instance variables, $(e,\texttt{'greet'})$ and $(u,\texttt{'greet'})$, both created via the $(\texttt{'__init__'},f_1)$ named method which is owned by the `A` class. For the demonstration of object valuation, the built-in class `str` is utilized. The valuated objects in the sample structure are exactly the instances of the `str` class (all of them indirect).

Transitions

In **FD**$_4$ structures, the transitions of the instance variable valuation $.\omega()$ are like those of the own method map $.m_\Theta()$ in **FD**$_3$ structures: upsert of individual triples from $O \times \Sigma \times O$ to $.\omega()$ and their possible deletion. However there is a difference in a typical occurrence of inserts, updates and deletions. While a deletion of an instance variable is still rarely used in common programs, there are usually many updates (in addition to inserts). That is, in contrast to methods, the values of instance variables typically change during the program execution (hence the term "variable").

Instance variable access

Similarly to method invocation we introduce an expression for instance variable valuation. This time we make use of the *self* object that is known from the execution context: For every name s,

`@<`s`>` evaluates to $self.\omega(s)$ (whenever $self.\omega(s)$ is defined).

For simplicity, we assume here that "`@`" does not appear in any possible name. (Just like we have already implicitly assumed that there is no occurrence of "`.`" in any name.)

This way we support the well-known paradigm of object oriented programming: Instance variables of an object x can only be accessed via methods to which x responds.

FD$_1$ to FD$_4$ in a summary

We have presented an object model based on the book *Putting metaclasses to work* [59] by Forman & Danforth. Following the approach of *abstract state machines*, we described the model in four steps by abstraction refinement. In each step i, the model is expressed as a family of **FD**$_i$ structures together with supported transitions.

- **FD**$_1$ structures $(O, .class, \leq)$, also called Python core structures, are constituted by *instantiation* and *inheritance* graphs. These are the two most fundamental relations between objects and they are already sufficient for the definition of the *metaclass* term. In the diagrams, we use blue arrows for the instantiation graph and green arrows for the inheritance graph.
- **FD**$_2$ structures $(O, .class, \leq, \leq_{mro})$ provide an ordering between inheritance-related pairs of objects, known as *method resolution order (MRO)*. As a result, each object has defined a *linear* ordering of its inheritance ancestors. This ordering is consistent with the inheritance relation so that the $\leq_{mro}$ constituent brings

additional information only in the case of multiple inheritance. In the diagrams, we use magenta arrows to indicate the part of MRO that is not implied by inheritance. The source and target of a magenta arrow is a green arrow.

- **FD$_3$** structures $(O, \Sigma, \Pi, .class, \leq, \leq_{mro}, .\pi, .m_\Theta())$ introduce additional sorts of *names* (Σ) and *methods* (Π). The last constituent, $.m_\Theta()$, is a partial map from $O \times \Sigma$ to Π and can also be regarded as a relation of *ownership* between objects (O) and *named methods* ($\Sigma \times \Pi$). There is an induced partial map from $O \times \Sigma$ to Π, denoted $.m_R()$, which can be regarded as the relation of *respondence* between objects and named methods. For each object x, the dictionary $x.m_R()$ of named methods to which x responds equals the merge of the dictionaries $y.m_\Theta()$ of named methods owned by inheritance ancestors y of $x.class$. The precedence of the merged dictionaries is given by method resolution order. In the diagrams, we use solid brown arrows to indicate named method ownership and dashed brown arrows to indicate the respondence.

  The $.\pi$ constituent is an auxiliary partial map from O to Π that can be used to refer to methods via objects. We have reserved the olive color for the diagrammatization of $.\pi$.
- **FD$_4$** structures $(O, \Sigma, \Pi, \Phi, .class, \leq, \leq_{mro}, .\pi, .m_\Theta(), .\varphi, .\text{ю}())$ introduce the additional sort of *values* (Φ) which are assigned to some objects via the $.\varphi$ map. The Φ set is meant to comprise some commonly understood data domains like integers, strings or booleans. The $.\text{ю}()$ constituent is a partial map from $O \times \Sigma$ to O, called the *instance variable valuation*, which equips each object x with a dictionary $x.\text{ю}()$ of *named objects*. (That is, $x.\text{ю}()$ is the valuation of instance variables of x.) This dictionary, together with the (optional) $.\varphi$-valuation of x, constitutes the object x data (alternatively, *state*). According to a fundamental paradigm of object-oriented programming, $x.\text{ю}()$ can only be manipulated via elements of $x.m_R()$ – the named methods to which x responds.

We can summarize the above using the chosen coloring of arrows. The notion of a *metaclass* is defined purely in terms of blue and green arrows (the "core"). However, to understand the *meaning* of the blue and green arrows more arrows are necessary, i.e. more colors. First, magenta arrows are added which provide a supplemental ordering of green arrows. Subsequently, solid brown arrows are added which point to named methods. Given this, the meaning of blue and green arrows can already be explained as providing the derivation

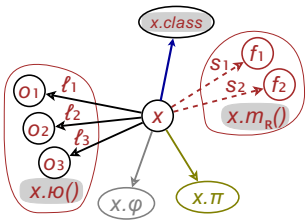
solid brown arrows → dashed brown arrows.

Dashed brown arrows bind objects to named methods to which they respond. To understand the meaning of brown arrows, black arrows are necessary which provide explicit named bindings between objects. Finally, to understand black arrows, gray arrows are introduced which associate some objects to values from some "external" domain. The domain is meant to contain values that are commonly understood. Optionally, there can also be olive arrows that provide direct association of objects with (unnamed) methods.

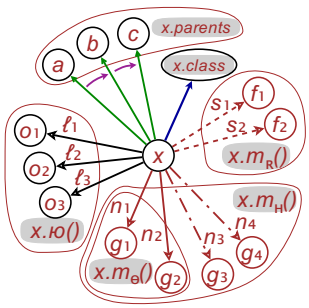
Object neighbourhood

A good picture of the hitherto defined structure(s) is established when we focus on the out-neighbourhood of an object x for all the introduced arrows. There are significant differences between ordinary objects and classes, as shown by the two diagrams below. As an exclusive feature that does not apply to classes, ordinary objects x can be $.\varphi$-valued as well as $.\pi$-valued. On the other hand, as an exclusive feature that does not apply to ordinary objects, classes can have inheritance parents (indeed, all except r have), and also can have own methods as well as inherited methods. All objects can have instance variables and received methods. The only out-arrow that is guaranteed to exist for every object x is the blue one – the link to $x.class$.

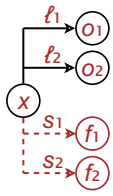
(a)
x is an ordinary object



(b)
x is a class

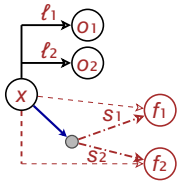


An informal simplification of **FD**$_4$ structures is described in the box below. The resulting structures provide an essential model of OOP with metaclasses, assuming single inheritance and omitting both the $.\varphi$- and $.\pi$-valuations. The description appears to be reversed to that of **FD**$_4$ structures since it starts with named method respondence as the requested goal of OOP.

1. **Bundle data and behavior via objects**.


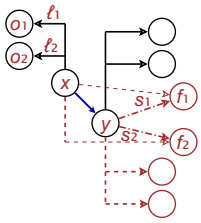
Objects are abstract entities that bundle data with behavior [20]. Data is a set of named objects and behavior is a set of named methods. A method is an abstract entity associated with code that consists of instructions for program execution. For an object x, we refer by $x.\text{ю}()$ to x's data (the valuation of instance variables of x) and by $x.m_R()$ to x's behavior (the named methods responded to by x). Names in $x.\text{ю}()$ and $x.m_R()$ are unique so that $x.\text{ю}()$ is a dictionary of objects and $x.m_R()$ is a dictionary of methods.

Objects have own identity, independent from data and/or behavior. That is, there can be different objects with identical data and identical behavior.

2. **Share behavior, particularize data**.


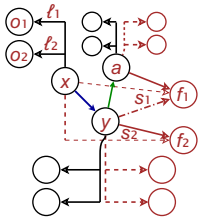
There is a fundamental difference between data and behavior. Typically, behavior is *shared* among multiple objects while data is not. If two different objects x and y contain the same data (i.e. $x.\text{ю}() = y.\text{ю}()$) then it is usually just by (temporary) coincidence. In contrast, if two objects x and y have the same behavior (i.e. $x.m_R() = y.m_R()$) then it is mostly by intention. Another way to view the difference is that data are usually generated by program execution but methods are not – they are typically written by a (human) programmer.

As a consequence, the binding mechanism used for data is different from that used for behavior. Each named object from an object's x data has an *explicit* binding from x. (That is, one "slot" per instance variable.) In contrast, behavior is only bound as a whole by a single common link to (the representation of) a *group* of methods. It follows that each named method to which an object x should respond only has an *implicit* binding from x, obtained by composition of the common link with an internal link within the group.

3. **Establish uniformity of receivers and providers**.


By the principle of uniformity, grouping of named methods is accomplished via objects. For every object x, there is an object y that serves as a designed *provider* of $x.m_R()$. Such a provider is called a *class*. Moreover, y is *the class* of x which in turn is a direct *instance* of y. This induces the $.class$ map between objects. For a class y, $y.m_H()$ is the set of named methods provided by y. Typically, a class y is fundamentally different from its instances so that it does not respond to methods that it provides.

If we suppress the concept of inheritance then there will be an *explicit* binding between y and each of the named methods from $x.m_R() = y.m_H()$. In general, the bindings are scattered through inheritance ancestors of y, according to the next step.

4. **Establish inheritance of providers**.


Make the method provision incrementally shared by establishing a hierarchy, denoted $\leq$, between providers (classes). Distinguish between method ownership and provision. For a class y, let $y.m_\Theta()$ be the set of *own* named methods. This dictionary forms a subset of $y.m_H()$. For each named method (s,f) from $y.m_\Theta()$ there is an *explicit* binding between y and (s,f). The whole dictionary $y.m_H()$ of provided methods arises as a merge of all dictionaries $a.m_\Theta()$ where a ranges through inheritance ancestors of y.

An even shorter description might look like this:

1. Objects are fundamental entities of OOP. Each object has data and responds to messages via methods that

manipulate the data.
2. Every object is a direct instance of a class – an entity that provides the methods with which the object responds to messages sent to it.
3. Classes are objects. (The metaclass pre-condition.)
4. Classes, as method providers, are built incrementally.

Python core structure in Python

We have defined Python core structures alias **FD**$_1$ structures as a part of object model by Forman & Danforth [59] – the part that is appropriate for a rigorous definition of the *metaclass* term. The first name which we have chosen indicates that we regard the family as relevant to the Python programming language. We have demonstrated the relevance several times using fragments of Python code. Moreover, Python was accompanying us not just in the initial family but all the way from **FD**$_1$ to **FD**$_4$. If we focus, for now, on **FD**$_1$-structures then we can say that we *have* already provided the correspondence between **FD**$_1$-structures and Python. First, we have mentioned (more than once) that `object` and `type` are the Python's correspondents to *r* and *c*, respectively. But more importantly, according to what we have written, there are even two independent ways how to express the correspondence between constituents of **FD**$_1$-structures and Python. For every objects *x*, *y*,

(1) $x.class = y \leftrightarrow$ `x.__class__ is y`
$x \leq y \leftrightarrow$ `(x is y) or (type in x.__class__.__mro__) and (y in x.__mro__)`

---

(2) $x \in y \leftrightarrow$ `isclass(y) and isinstance(x,y)`
$x \leq y \leftrightarrow$ `(x is y) or isclass(x) and isclass(y) and issubclass(x,y)`

The first correspondence is via the `__class__` attribute of objects and `__mro__` attribute of classes. Correspondence (2) uses the built-in `isinstance` and `issubclass` introspection methods. (This time the correspondence relates to $\in$ and $\leq$ as the definitory constituents, see Axiomatization via $\in$.) The `isclass(x)` expression is a shorthand for `isinstance(x,type)` as defined in the `inspect` module.

Let us assume that (1) or (2) is the Python's definition of possible combinations of the instantiation and inheritance graphs. Given this, does Python conform to the axiomatization of Python core structures? The answer is: no. We will see that conformance is asserted for neither of (1) or (2) and that the two definitions are in general not consistent. As for (1), Python allows explicit setting of both `__class__` and `__mro__` attributes of objects without making sufficient checks that all of (py~1)–(py~7) are satisfied.

As for (2), `isinstance` and `issubclass` are introspection methods that can be overridden: if a class *y* responds to a named method `('__instancecheck__',f)` (owned necessarily by a metaclass) then `isinstance(x,y)` will evaluate to *y*.*f*(*x*) for all objects *x* (that are not direct instances of *y* – another Python speciality). Similarly, `issubclass(x,y)` can be overridden by defining `__subclasscheck__` in *y.class*. Moreover, even overrides made by Python's standard library via abstract-base-classes do not respect all the axioms.

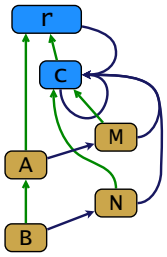
In the `issubclass(x,y)` expression, both arguments are required to be classes (otherwise an exception is raised). The `issubclass` method puts this constraint to just the second argument. However, according to the Python documentation [150b] both methods have an altered recognition of classes, shown by the code on the right: if *x*.`__bases__` evaluates to a tuple of instances of *c* then *x* is regarded as a class.

```
class A: pass
x = A()
try: issubclass(x,x)
except Exception as e: print(e)
# issubclass() arg 1 must be a class
x.__bases__ = ()
print (issubclass(x,x)) # True
isinstance(x,x)       # False but OK
```

Subverting `__class__`

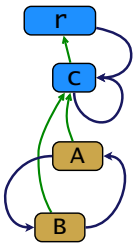
First of all, Python allows to change the class of an object, with some limitations. That is, transitions are not incremental w.r.t. *.class*. Moreover, some of these transitions allow for violation of some of the (py~∗) axioms. The examples below show that by manipulating the `__class__` attribute, it is possible to create structures with a non-monotonic *.class* map (the (a) case), as well as structures in which *{c}* is not the only cycle of *.class* (cases (b) and (c)).

(a)
¬(py~3):
B.*class* ⩬ A.*class*



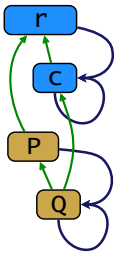

```
r = object
c = type
M = c('',(c,),{})
N = c('',(c,),{})
A = M('',(  ),{})
B = M('',(A,),{})
B.__class__ = N
```

(b)
¬(py~5):
B.*class(2)* = B ≠ *c*




```
r = object
c = type
dummy = c('',(c,),{})
A = dummy('',(c,),{})
B = dummy('',(c,),{})
B.__class__ = A
A.__class__ = B
```

(c)
¬(py~5):
Q.*class* = Q ≠ *c*

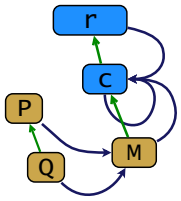



```
r = object
c = type
dummy = c('',(c, ),{})
P = dummy('',(   ),{})
Q = dummy('',(P,c),{})
Q.__class__ = Q
P.__class__ = Q
```

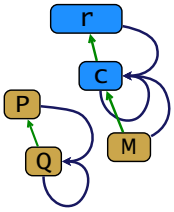
Subverting `__mro__`

Although the `__mro__` attribute of classes cannot be changed directly by assignment like `__class__`, it can be subverted too. To do this, use a metaclass that owns a named method `('mro',f)`. The example below shows how to create structures that violate (py~4) – classes `P` and `Q` are not descendants of *r*. (Observe that monotonicity of *.class* is still satisfied. Moreover, (py~5) and (py~6) can be reformulated similarly to what is presented in the in Disallowed structures subsection so that (py~4) is the only axiom that is violated.)

(a)



(b)



(Python 3)

```
r = object
c = type
def P_mro(self): return [self]
def Q_mro(self): return [self,P]
M = c('',(c,),{})
M.mro = P_mro; P = M('',(M,),{})
M.mro = Q_mro; Q = M('',(M,),{})
P.__class__ = Q
Q.__class__ = Q
```

The code on the right without the last 2 lines corresponds to (a), the whole code to (b). Note that in both cases, `P` and `Q` are classes (according to the original definition which defines the set *C* of classes as equal to *Q.class(2).*∋). In the (b) case, `P` and `Q` are not instances of *c* so that they are not reported as classes by `inspect.isclass`. However, Python still allows these two objects to appear in place of any argument of `issubclass` and `isinstance`. These introspection methods faithfully report the shown structure. (That is, except for `issubclass(P,Q)` both methods evaluate to `True` for any combination of arguments taken from *{P,Q}*.) Finally, observe that in the (b) case, the `Q` class, being the class of itself and having `P` and `Q` as the only instances, passes both the primary and secondary classic definitions of a metaclass.

The `__mro__` attribute can also be subverted to achieve a violation of (py~1). That is, Python does not guarantee that inheritance, as induced by `__mro__`, is a partial order. The code on the right demonstrates possible irreflexivity – the `A` class is not its own descendant. Similarly, there can be transitivity breaks. In contrast, antisymmetry seems to be guaranteed (or at least substantially harder to break).

```
class M(type):
  def mro(self): return []
A = M('',(),{})
A.xx = 41
print (A.__mro__)       # ()
print (issubclass(A,A)) # False
print (hasattr(A,'xx')) # False
```

Abstract base classes

As of version 2.6 and newer, Python contains a facility for a "virtual extension" of the inheritance relation, known as *abstract base classes* [154]. An abstract base class is a class that is an instance of the `ABCMeta` metaclass provided by Python Standard Library in the `abc` module. By calling `x.register(y)` where *x* is an abstract base class and *y* is an arbitrary class it is established that *y* is a *virtual subclass* of *x*. This relationship is then reflected by the `isinstance` and `issubclass` introspection methods, but it does not affect the respondence of objects to named methods. The underline in "arbitrary" indicates that there are no checks as to the consistency of the resulted relation – it is just considered to encode a "gentlemen's agreement". (Thus, depending on whether you are a gentleman or not, you can achieve cycles in inheritance.) The code on the right demonstrates a break of transitivity in the standard library. [81]

```
from collections import Hashable
print(issubclass(list, object))     # True
print(issubclass(object, Hashable)) # True
print(issubclass(list, Hashable))   # False
```

Superclass linearization

So far, only the core structures have been considered in this section – those constituted by inheritance and instantiation relationships. What about $\mathbf{FD}_i$ for $i > 1$? The case $i = 3,4$ is discussed in the next section. As for $\mathbf{FD}_2$, Python performs C3 linearization upon class creation. That is, for a class *x*, if `x.__mro__` is not overridden via a `mro` method as above then it is set to the list *[x] + c3(ps)* where *ps* is the list that encodes the requested precedence order of (the selected) inheritance ancestors of *x* and *c3* is the linearization map. (If *c3(ps)* is not defined then the class creation request is rejected.) The *ps* list is then stored in the `x.__bases__` attribute. This happens even if `__mro__` is overridden, so that `x.__bases__` stores just a part of the class creation request. The attribute can even be subsequently modified, as shown in the code on the right. In particular, `x.__bases__` is not guaranteed to be a sublist of `x.__mro__`, contrary to the conditions asserted by the C3 linearization.

```
c = type
class P: pass
class M(c):
  def mro(self): return (c,c)
A = M('',(P,M),{})
print (A.__bases__ == (P,M)
  and  A.__mro__   == (c,c)) # True
A.__bases__ = (M,M)
print (A.__bases__ == (M,M)) # True
```

Presumably, the `__bases__` attribute is of little importance to the Python data model. In particular, the inheritance relation ≤ is based on `__mro__` and disregards `__bases__`.

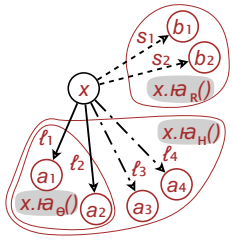
Python attributes

We have seen that there are discrepancies between the family of $\mathbf{FD}_1$ structures (and consequently $\mathbf{FD}_2$ structures) defined by our object model, and the Python's counterpart, as observed at execution of simple Python scripts. However, all the demonstrations involved some form of hacking: either `__class__` or `__mro__` have been explicitly set. Without it, one can still hope for an exact correspondence.

Things change substantially when going to $\mathbf{FD}_3$ and $\mathbf{FD}_4$ structures. In contrast to our model which keeps methods separate from instance variables and only applies the $(O, .class, \leq_{mro})$ structure for the sharing of named methods, Python blends the two things together into *attributes*. This is in turn based on blending inheritance with object membership.

Blending ≤ with ∈

Let us define an *attribute* in Python to be a named object, i.e. it is a pair *(s,o)* from $\Sigma \times O$. Similarly to named method ownership, $.m_\Theta()$, there is a relation of *attribute ownership* between objects and attributes, which is simultaneously a partial map from $O \times \Sigma$ to O. Let us denote this map by $.ha_\Theta()$, so that $x.ha_\Theta(s) = y$ means that *x* is an owner of the *(s,y)* attribute and $x.ha_\Theta()$ refers to the dictionary of all own attributes of *x*. Like named method ownership in $\mathbf{FD}_3$ structures, the relation of attribute ownership in Python is definitory and can be directly manipulated by upserts (`setattr`) and deletions (`delattr`). There are, by the same analogy, derived relations $.ha_H()$ and $.ha_R()$ of attribute provision and respondence, respectively, although these terms are generally not so fitting as with named methods. However, in contrast to our model (an the F&D model too), there is one more derived



relation between objects and attributes which can be called *attribute resolvability*. Let us denote it simply by dropping the subscript, $.ʁa()$. The relation is given as the merge of provision and respondence, with the provision taking precedence. That is, using the ◐ operator introduced with method dictionaries, for every object x,

$x.ʁa() = x.ʁa_H() \,◐\, x.ʁa_R()$.

In particular, attribute resolvability restricted to $\{x\}$ is given by attribute ownership restricted to $x.↕ \cup x.∈$. Since $x.ʁa_R()$ is by definition equal to $x.class.ʁa_H()$, the above equality can be read as follows:

The resolved attributes of x are those inherited by $x.class$ updated by those inherited by x.

For the purpose of further description, we introduce terminology which reflects this merge. If $x.ʁa(s) = y$ then we say that (x,s) is *resolved to* y. If, in addition, $x.ʁa_H(s) = y$ then (x,s) is resolved to y *via* ≤. Otherwise, i.e. if $x.ʁa_R(s) = y$ and $x.ʁa_H(s)$ is undefined then (x,s) is resolved to y *via* ∈. Note that "resolving via ≤" is another name for $.ʁa_H()$ whereas "resolving via ∈" is a name for the difference $.ʁa() \setminus .ʁa_H()$.

Default semantics of $x.<s>$

The attribute resolvability relation $.ʁa()$ defined in the previous subsection provides default semantics for (dot-) qualified name resolution in Python. For an object x and a name s,

$x.<s>$ is evaluated to $x.ʁa(s)$.

If $x.ʁa(s)$ is undefined then evaluation of $x.<s>$ results in an `AttributeError` exception. In the code on the right, `B.n` is resolved via ≤ and both `B.m` and `b.n` are resolved via ∈. An attempt to resolve `b.m` leads to an exception, in spite of `b` ∈ `B` ∈ `M` and `M` being an owner of `('m',int(41))`, showing that $∈^2$ is not taken into account during the attribute resolution.

```
class M(type): pass
class N(M): pass
class A(metaclass=N): pass
class B(A): pass
b = B()
M.m = 41; M.n = 30; A.n = 26
print (B.m,B.n,b.n) # 41 26 26
try: b.m
except Exception as e: print(e)
#'B' object has no attribute 'm'
```

Method resolution

Let us recall that in **FD$_3$** structures we introduced methods as abstract entities which, when equipped with a name, can be *received* by objects. An object x can only have a chance to be a receiver of a named method (s,f) if $x ∈ y$ for some object y (necessarily a class) that is an owner of (s,f). Methods can only be invoked upon their receivers. If a named method (s,f) is received by an object x then $x.<s>(a_1,…,a_n)$ is the canonical expression for the invocation of f upon x with $a_1,…,a_n$ as arguments. If there are zero arguments, we can use $x.<s>$ as a shorthand for $x.<s>()$. The invocation of f changes the execution context: the x object becomes the "current object".

Python is at odds with the above described mechanism, mainly due to the blending of inheritance (≤) with membership (∈). In Python, some objects x are *callable*. Let us regard them as $.π$-valued, but use the term "function" for $x.π$ rather than "method". In a typical program, most Python's callable objects are instances of the built-in class referred to by `types.FunctionType` and named `function`. (In contrast to `object` or `type` the class is not by default available under its name but we will use the name to refer to the class.)

Instances of `function` play a special role in attribute resolution. Let y be such an instance. If (x,s) resolves to y via ∈ then the evaluation of $x.<s>$ results in a new object p that is a reification of (x,y). Such reifications are called *bound methods* and are instances of the built-in class `types.MethodType` which reports itself as `method`. The expression $x.<s>(a_1,…,a_n)$ is evaluated in two steps as

$p = x.<s>; \quad p(a_1,…,a_n)$

Subsequently, $p(a_1,…,a_n)$ is interpreted as $y(x,a_1,…,a_n)$ – the y object is called with x as the first argument, the remaining arguments, if any, are shifted to the right.

```
def y(self=0): print (self)
class A: pass
class B(A): pass
A.m = y
x = B  ; x.m()                  # 0
x      ; print (x.m is y)       # True
x = B(); print (x.m is y)       # False
x      ; x.m() # <__main__.B object ...
print (y.__class__.__name__) # function
print (x.m.__class__.__name__) # method
```

In contrast, if (x,s) resolves to y via ≤ then evaluation of $x.<s>$ results just in y, i.e. it has the default semantics according to the previous subsection. As of Python 3, no reification takes place. (This differs from

previous versions of Python [4].) As a consequence, $x.\langle s\rangle(a_1,\ldots,a_n)$ is interpreted as $y(a_1,\ldots,a_n)$ – the context of the x object is lost.

Faking `__class__` and `__mro__`

We have seen that the default semantics of $x.\langle s\rangle$ is by default overridden when (x,s) resolves via $\in$ to an instance of `function`. Python provides interception facilities for further overrides: the `__getattribute__` and `__getattr__` methods as well as per-attribute access via descriptors. In addition to interception there is a subtle refinement regarding attribute ownership. For an object x there are in general 2 (abstract) dictionaries of attributes of which x can be considered to be an owner. The first dictionary consists of *special* attributes whose names start and end with double underscore. The optional second dictionary is reified into an object d whose class is either `mappingproxy` (if x is a class) or `dict` (if x is an ordinary object). If present, then d appears in the first dictionary under the `'__dict__'` key – i.e. d is usually referred to by x.`__dict__`.

Unfortunately, the two dictionaries can share some keys – that is, attributes in the second dictionary can have special names. In some cases, such attributes have impact to attribute resolution. The code on the right shows that both `__class__` and `__mro__` can be faked this way. This is different from the subversion demonstrated before since the "real" attributes remain unchanged. They are just shadowed. The last line indicates that for the introspection of the class of x, `type(`x`)` is more reliable than x.`__class__`. There is no counterpart to `__mro__` (known to the author), although in the code, `A.mro()` would evaluate to `[A,object]`. However, this just invokes the (recomputation of) C3 linearization and relies on `A.__bases__` which can be subverted.

```
class M(type):          __mro__   = 55
class A(metaclass=M): __class__ = 33
a = A()
print (A.__mro__)               # 55
print (a.__class__)             # 33
print (type(a) is A)            # True
```

The *is-a* misnomer

One of the most remarkable points in the F&D book is the name introduced for the object membership relation, $\in$. As we have already mentioned before, the name is: *isA*. Just like *isDescendantOf* is a camel case concatenation of "is descendant of", *isA* comes from "is a", often written with a hyphen as *is-a*. That is, in Python core structures, the following statements about objects x and y are equivalent:

- x is a y,
- x is a member of y, ($x \in y$)
- x is an instance of y. (Recall that we follow the Python's terminology here and allow indirect instances.)

This equivalence reflects faithfully the meaning of the indefinite article. Quoting from wikipedia: [𝕎₇₀] "English uses *a/an*, from the Old English forms of the number 'one', as its primary indefinite article." That is, "x is a y" means "x is one y". Let us recognize this semantics in two phrases borrowed from [63]:

1. "John is a student" is a case of "x is a y" where x equals to "John" and y equals to "student".
2. "a student is a person" is a case of "x is a y" where x equals to "a student" and y equals to "person".

Note in particular that in the second phrase, the substitution for x is not "student" but "**a** student". That is, it is wrong (a misinterpretation of the indefinite article) to infer "is-a" relationship between "student" and "person", since "student" is not the same as "a student". The word "student" denotes a concept of which "John" is one possible instance, whenever "John" is a unique identifier in given context. Being such a concept, "student" is not a "person".

Unfortunately, exactly this mistake happened in the early days of information technology and has not yet been rectified. As can be observed on the wikipedia page titled Is-a, in most publications, *is-a* refers to $\leq$ (instead of $\in$). This is despite the fact that in the field of knowledge representation, the problem of confusion of $\in$ with $\leq$ has already been recognized in the 1970s (!), see [109] or [21] where "*infamous "ISA" links*" are mentioned.

The cause

What is the probable cause of the misnomer? We can speculate that it has something to do with the metaclass anti-condition. In object-oriented modelling, the main model is constituted by the class diagram. In the presence of the anti-condition, there are no objects in such a model. With a slight exaggeration, this can be paraphrased as:

> By default, an object model is prohibited from containing objects.

Since there are no objects, there are no instantiation links. That is, by the default meaning of the indefinite article, the *is-a* relation between the entities (the nodes) of the diagram is empty. This vacant place poses an opportunity to override the semantics.

The opportunity has been seized in favor of inheritance, ≤. The reasoning goes as follows. If `B` and `T` are classes such that `B ≤ T` (think of the names as of abbreviations for `Beech` and `Tree`) then one can say that

a `B` *is a* `T`,

meaning that an instance of `B` is also an instance of `T`. The sentence structure can be expressed as "a-B is a-T" (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 `B` and `T`.

This is in turn abbreviated to "`B` *is-a* `T`" (so that the `B`'s article is dropped and the `T`'s article is secluded from `T`), meaning that the `(B,T)` pair belongs to the *is-a* relation. If the hyphenation is finally removed then the whole "process" can be schematized as

a `B` *is a* `T` ⇢ `B` *is a* `T`,

that is, we just dropped the `B`'s article. In contrast, the `T`'s article has been preserved.

The correct *is-a*

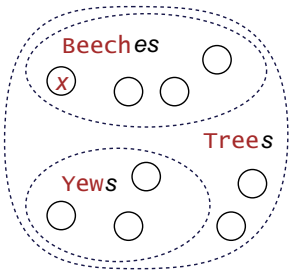
Let us state explicitly that the correct semantics of *is-a*, as implied by the default meaning of the indefinite article, is that of object membership, ∈. That is, for every objects *x*, *y*,

$$x \text{ is-a } y \quad \leftrightarrow \quad x \in y.$$

There are a few notable cases in object technology in which the "is-a" phrase is interpreted correctly. All the occurrences appear exclusively hand in hand with the acceptance of the metaclass pre-condition. We have already mentioned the F&D book's definition of *isA*. In addition, and in particular, there is as an `is_a?` introspection method in Ruby, and an `isa` pointer in Objective-C [35]. Another important contexts where the term *is-a* or *isa* is used correctly are F-logic (see *isa-F-atom* in [6] and *is-a assertion* in [26]) and the KL-ONE frame language [181] [106]. See also [120] [151] [32] [14] [158] and [159] for further occurrences of the correct use of *is-a*.

∈ *Note:* The *lunate epsilon* symbol '∈' [98] [𝕎39] which we have introduced for object membership was used by Giuseppe Peano in 1889 for set membership [144]. According to Peano, "*a* ∈ *b*" is read as "*a* est quoddam *b*" (in Latin) which means "*a* is a *b*" (in English) [𝕎40]. This demonstrates a correct use of *is-a*. The image on the left displays the assumed rendering of the unicode character U+03F5.

Classifier naming convention



Since we follow the metaclass pre-condition, namely that classes are objects, we need to interpret the *is-a* phrase correctly as having the semantics of ∈. That is, *x* ∈ `B` means *x* is a `B`. Equivalently, being "a `B`" is the same as being "a member of `B`", which we (still) hold for equivalent to being "an instance of `B`". In accordance, we let the set `B.э` of members of `B` be referred to as `B`s.

This allows us a convenient expression of object classification. In the diagram on the right, `Beech`es are the instances of the `Beech` class, `Tree`s are the instances of `Tree`. We assume that `Tree` is a common ancestor of `Beech` and `Yew`, so that e.g. all `Beech`es are `Tree`s.

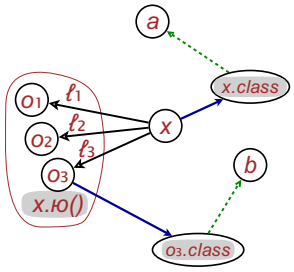
In a Python core structure, objects are exactly the `Object`s and classes are exactly the `Class`es whenever `Object` is the name of the inheritance root *r* and `Class` is the name of the instantiation root *c*, just like proposed by the F&D model. In a potential generalization of Python core structures, the naming convention can be used for a subtle differentiation. This has already been shown for the case of Java core structures, where classes are (among) `Class`es but the reverse inclusion does not hold.

is-a versus *has-a*

The "is a" catch phrase, as known from object technology, has an established counterpart: "has a". The *has-a* relationship is a term associated with *object composition*. (A car has a steering wheel.)

The following equivalences show the definition of *has-a* as it corresponds to our definition of *is-a*. In an **FD$_4$** structure, for every objects *x*, *y*,

x is y	↔	*x* = *y*,
x is a y	↔	*x* ∈ *y*,
x has y	↔	*x*.ю(*s*) = *y* for some name *s*,
x has a y	↔	*x*.ю(*s*) ∈ *y* for some name *s*.



x is-a a
x has o_i
o_3 *is-a b*
x has-a b

That is, *has-a* is the composition of *has* with *a*, where *a* is synonymous to *is-a* (since *is* stands for identity) and *has* is the projection of *.ю()* to *O* × *O*, i.e. *x has y* iff *x* has an instance variable whose value is *y*.

In the previous subsections, we have recognized ≤ as the overridden definition of *is-a* used in class diagrams. Now we have the non-overridden definition of *has-a*. What is the correspondent relation used in class diagrams? Let us refer to this relation by *HAS*. For convenience, let *SUB* be an additional term for inheritance (an abbreviation of *subsumption*). The answer can be then expressed as follows. In an **FD$_4$** structure that is "sufficiently saturated", for every classes *a*, *b*,

a SUB b ↔ *a*.э ⊆ *b*.э,
a HAS b ↔ *a*.э.ю(*s*) ⊆ *b*.э for some name *s*.

By "sufficiently saturated" we mean that the inclusions on the right side are supposed to be preserved by transitions. In case of *SUB* this has been established by the introduction of ≤ as a definitory constituent. The *HAS* relation, however, goes beyond **FD$_4$** structures – it would be necessary to equip classes with the "types" of the instance variables of their instances. Moreover, the *has* relation on which *HAS* is based implies no ownership and thus expresses *aggregation* rather than *composition*.

Finally, the *HAS* relation is usually considered in a more strict sense where there is equality instead of inclusion, so that *a HAS b* iff *b* is the classifier for *a*.э.ю(*s*) for some *s*. (That is, even though a car has a wheel and a wheel is an object, it is not usually stated that a car has an object.)

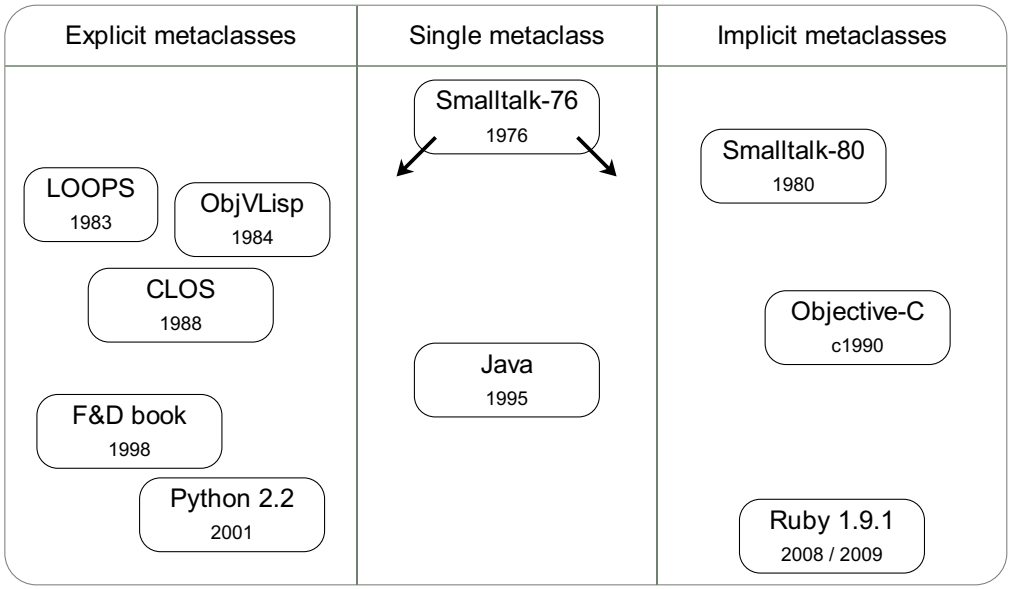
The OOP landscape of metaclasses

Until now we have investigated the metaclass term almost exclusively in the context of Python and the F&D

book. We might say that by the introduction of Python core structures we have established an initial definitory context. Now we will look at other contexts in which the term occurs and under the specific conditions there, try again to provide the answer to the title question. We make efforts to develop a universal model that is applicable in most contexts. We will usually proceed as follows:

1. Recognize the core structure: Figure out what the blue and green links are in given circumstances.
2. Make judgements about what combinations of blue and green links are allowed.
3. Establish the definition of a metaclass.

The following table provides a partial outline of the considered OOP environments in a chronological order.



The table also provides a division of the environments into three groups according to a fundamental characteristics of their core structure. In the case of the middle group, there is exactly one metaclass. The distinction between explicit and implicit metaclasses can be approximately described as follows.

- An *implicit metaclass* y is a metaclass that has a top member x – all potential members of y are descendants of x, written as $x.\mathfrak{l} = y.\mathfrak{d}$. In particular, $y.\mathfrak{d} \neq \varnothing$.
- An *explicit metaclass* y is a metaclass that is not implicit, i.e. $y.\mathfrak{d}$ is potentially topless, allowing $y.\mathfrak{d} = \varnothing$.

We allow the built-in (circular) metaclasses to be exempt from this description. Under this assumption all metaclasses in a Python core structure are explicit. Note that structures with implicit metaclasses cannot in general be created by attaching new objects one-by-one.

Smalltalk-76

Smalltalk-76 is the first programming language that is subject to the metaclass pre-condition: classes are objects. The introduction of metaclasses into programming can therefore be attributed to Smalltalk-76. According to Pierre Cointe [39] (see also [28])

> " Metaclasses were invented by Smalltalk-76 in order to express the behavior of classes as true objects able to handle message passing. "

A similar claim can be found in [46]: "*… Smalltalk-76 introduced metaclasses, classes of classes.*" However, there is no mention of metaclasses in the language documentation [84] and Smalltalk-76 authors even use the "No metaclasses" characterization for the language [85]. This is because the metaclass pre-condition is satisfied in a minimum possible way: there is only one class in the range of ϵ^2, the built-in metaclass `Class`, whose subclassing is disallowed.

The structure of instantiation and inheritance in Smalltalk-76 arises as a simplification of Python core structures. Let *Smalltalk-76 core structure* be a structure $(O, .class, \leq, r)$ where

- O is a set of *objects*,
- *.class* is the *class* map $O \to O$,
- $\leq$ is the *inheritance* relation between objects, and

(s76~1) $(O, \leq)$ is a partial order.
(s76~2) $C.class = \{c\} = c.\mathfrak{l}$.
(s76~3) Objects from $O \setminus C$ are minimal in $\leq$.
(s76~4) $O.class \subseteq C$.
(s76~5) The set $c.\mathfrak{l}$ contains exactly 2 objects.
(s76~6) $(C, \leq)$ is a tree.

- r is a distinguished object.

The structure is subject to the axioms in the box on the right, with C denoting the set $r.\mathrm{I}$ of *classes* and c denoting $r.class$.

(s76~7) There are only finitely many objects.

Observe that a Smalltalk-76 core structure is a Python core structure satisfying the following two constraints:

(i) Single metaclass: c has no strict descendants.

(ii) Single inheritance: $(C, \leq)$ is a tree.

Smalltalk-76 also established the `Object` and `Class` nomenclature for the circular classes r and c, respectively.

ObjVLisp

ObjVLisp [27] [28] [38] is an experimental object-oriented extension of Lisp developed by Pierre Cointe and Jean-Pierre Briot in the mid-1980s. The main purpose of ObjVLisp is to propose a uniform object model with metaclass support. The core structure of blue and green links arises from that of Smalltalk-76 by removing the two above mentioned constraints (i) and (ii). In contrast to the F&D model, the *.class* map monotonicity is not asserted. As a result, Python core structures can be thought of as ObjVLisp core structures constrained by the monotonicity condition.

In some aspects, papers [27] [28] can be regarded as precursors to the F&D book although no such claim appears in the book. There are several points in the papers that can be used to enhance the description of the core structure. We already did so by adopting the term *instantiation graph* as the missing counterpart to the *inheritance graph*. Yet more remarkable is the term *terminal instances* which is correspondent to "ordinary objects" from the F&D book. We will futher adopt the adjective "terminal" and introduce the following new terminology and notation. In a Python core structure, let

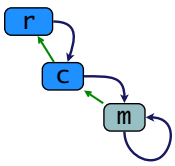
T be the set $O \setminus O.\in.\mathrm{I}$ and call elements of T *terminal(s)*.

That is, we use "terminal" both as a noun and an adjective. Furthermore, we use *non-terminal* as the negation of *terminal*. In a Python core structure, classes are the non-terminals. We will further encounter structures in which classes are defined to form a strict subset of non-terminals.

Moreover, "terminal" can be used to overcome the ambiguity of "ordinary". (Recall that by the F&D book, "ordinary object" is an object that is not a class and "ordinary class" is a class that is not a metaclass. Assume that x is an ordinary class. Being a class, x is an object. Is x ordinary?)

Note: The term "*terminal objects*" for objects that have "*no instantiation semantics*" is used in [122].

LOOPS



LOOPS [19] is another object-oriented extension of Lisp that explicitly uses the metaclass concept. In contrast to most models encountered in object-oriented programming, LOOPS contains an explicit classifier for metaclasses, a class named `Metaclass` which becomes the instantiation root. This leads to the "*golden braid*" of circular classes `Object`, `Class` and `Metaclass` as the respective classifiers for objects, classes and metaclasses. The structure of blue and green links between them is shown by the diagram on the right (see also [38] or [87], Figure 7.4).

Metaclasses in CLOS

The *Common Lisp Object System* (CLOS) is an extension of the Common Lisp programming language that is strongly associated with the metaclass term, especially in academic circles. As of 2015, Google Scholar [⧉] reports more results for "clos metaclass" than for "python metaclass". We will regard CLISP 2.49 [74] as the reference implementation.

The CLISP built-in core structure

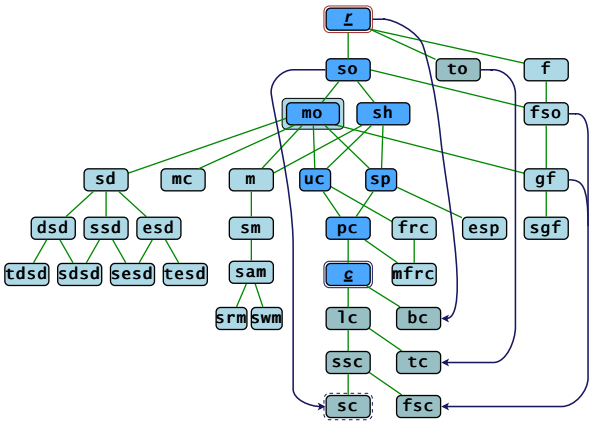
The diagram below shows a part of the built-in core structure of CLOS as implemented by CLISP. Every node in the diagram corresponds to an object. For convenience, each node is labelled by a short identifier, like with diagrams presented before. However, each corresponding object *x* also has a full name which is available by evaluation of `(class-name x)`. If you are viewing this document in Mozilla Firefox or Google Chrome you can get the full name of each object by hovering the cursor over the label. Alternatively, see [136] for the full legend. The green links have an implicit upward direction. Moreover, the *.class* map (blue links) is shown in the reduction (*.class*) introduced for Python core structures: For each object *x*, there is a least ancestor of *x* that is the source of a (unique) blue link. The target of the link then equals *x.class*. The structure can be verified by introspection methods according to the box on the right.

x.class ... `(class-of x)`
x.ancs ... `(class-precedence-list x)`
x ∈ *y* ... `(typep x y)`
x ≤ *y* ... `(subtypep x y)`

The set *X* of selected objects is (presumably) the smallest one such that

- the *r* object (named `t` or also `⊤` to resemble the ⊤ symbol for top) is in *X*,
- *X* is closed w.r.t *.class* and *.↕* (see Substructures),
- every descendant *x* of *r* for which *x.class* is defined is in *X*, (*x* is one of `r`, `so`, `to`, `fso` or `gf`)
- every descendant of `mo` (which is named `metaobject`) is in *X*.

The discovery of all `mo`'s descendants has been accomplished by a recursive application of `class-direct-subclasses`.



A smaller part of the built-in structure is shown in the next subsection, this time with unreduced *.class* map. The diagram is aligned according to the reachability of `sc` (`standard-class`) to show that the *.class* map forms a tree.

What is a class?

Note that we avoided using the term "class" in the previous subsection. The word only appears as part of CLOS introspection method names like e.g. `class-of` (in addition to the inferred *.class* map). This has two reasons.

(A) You can't be sure whether classes are subject to "everything is an object" principle in CLOS. This problem is caused by inconsistent CLOS documentation discussed later.

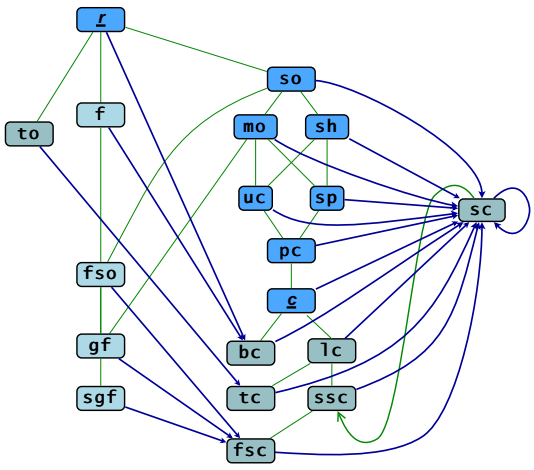
(B) Even if every class is considered to be an object like in Python or in the F&D model, the definition of the set *C* of classes as being equal to *O.class(2).∋* does not apply to CLOS. (Consider e.g. the `f` object, named `function`.) Instead, we use the *C* = *O.∈.↕* definition.

The reason for (B) is (necessarily) that the above structure is not a Python core structure. It can be observed that neither of (py~3), (py~5) or (py~6) is satisfied. In particular, there are breaks of monotonicity of *.class*. (For example `so` ≤ *r* but `so`.*class* ≰ *r.class*.)

However, the presumed family of CLOS core structures is still close to Python core structures:

- The *.class* map forms a **tree** according to the diagram on the right. The root of the tree is *r.class(2)* rather than *r.class*. (The root is the `sc` class, named `standard-class`. Observe that although *{sc}* is the only cycle of *.class*, it cannot be used as a classifier for classes – not every class is an instance of `sc`.)
- Although there are built-in monotonicity breaks, the CLISP interpreter performs checks of *.class* monotonicity for user-created classes. [74a]

A possible axiomatization of CLOS core structures can be



found in [136].

What the literature says

To demonstrate the (A) problem from the previous subsection, let us consider the following three authoritative sources about CLOS:

- the book titled *The Art of the Metaobject Protocol* (AMOP) [91],
- the book titled *Common Lisp the Language* [169], and
- the html document titled *Common Lisp HyperSpec* [147].

Each of the documents has its own wikipedia page [𝕎63] [𝕎64] [𝕎62]. According to AMOP, classes are not objects. They are only *represented* by objects, called *class metaobjects*:

> *The term class metaobject is used for the backstage structure that represents the classes …* (Page 18)

The book is fairly consistent in the distinction of classes from objects. For example, according to page 28, the `class-of` function, when applied to an object *x*, does not return the class of *x* but just the metaobject of *x*'s class.

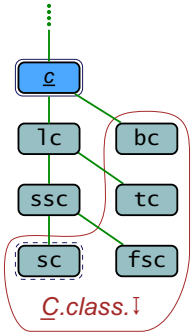
In contrast, the book [169] considers classes to be (among) objects just like in Python or the F&D model: "A *class* is an object that determines the structure and behavior of a set of other objects, which are called its *instances*". Similarly, "The function `class-of` returns the class of which the given object is an instance". The same approach is taken in the Common Lisp HyperSpec document [147] except for one occurrence of "Classes are represented by objects" [147a]. However, the CLISP document [74] which builds upon both AMOP and HyperSpec again oscillates between the two approaches.

What is a metaclass?

According to AMOP, the term "metaclass" is not appropriate for the description of CLOS:

> *We refrain from using the term "metaclass" for classes like `standard-class`, choosing to use the more explicit phrase: class metaobject class. The sole exception is an option we will add to `defclass` called `:metaclass`, which has been retained for historical reasons.* (Page 75)

The book [169] mentions `standard-class`, `built-in-class` and `structure-class` as pre-defined metaclasses. We presume that subclasses of these classes are meant to be metaclasses too, although there is no such statement in the book. The HyperSpec glossary provides the classic definition [147c]. Moreover, the description of the `class` class [147b] resembles the definition of the classifier for classes: an object *x* is a class if and only if *x* is an instance of `class`. We can therefore assume that `class` is the top metaclass and that metaclasses in CLOS are exactly the descendants of `class`. We can even use the $\underline{c}$ symbol for `class` provided that $\underline{c}$ is defined by



$$\underline{c} = \bigvee \underline{C}.class,$$

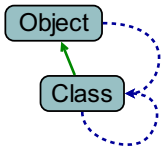
i.e. the top metaclass $\underline{c}$ equals the least common ancestor of classes of classes. Note that this equality holds also in Python core structures. The three CLOS classes named `class`, `clos::slotted-clas` and `clos::semi-standard-class` which form the difference $\underline{c}.\downarrow \setminus \underline{C}.class.\downarrow$ in the built-in structure are (presumably) "abstract" – they are disallowed from having any (potential) direct instances.

S. Koide and H. Takeda define CLOS metaclasses as the (non-strict) subclasses of `standard-class` (`sc`) [96] [97]. In a parallel to RDF Schema, they assume that being a fixpoint of *.class* is the definitory characteristic for the top metaclass, disregarding built-in breaks of monotonicity of *.class*.

Is `java.lang.Class` a metaclass?

As of 2015, the Java programming language can be considered as the most popular object-oriented language, this status unchanged for more than one decade. It is therefore worthwhile

to investigate whether or how the metaclass term applies to Java. What are metaclasses in Java? We can immediately discover a similarity between the F&D model and the Java hierarchy of built-in classes: the `java.lang` package contains classes named `Object` and `Class` which are in an exact correspondence to *r* and *c*, i.e. the classes named *Object* and *Class* in the F&D model, respectively.



The code on the right uses the introspection methods `getSuperclass` (defined in `Class`) and `getClass` (defined in `Object`) to demonstrate that

(a) `Object` is the top class,

(b) `Class` is a direct strict subclass of `Object`,

(c) both `Object` and `Class` are direct instances of `Class` (since both `r.getClass()` and `c.getClass()` evaluate to `c`).

```
class X {public static void main(String[] x){
  Class
  r = Object.class,
  c = Class.class; System.out.println (
  r.getSuperclass() == null &&
  c.getSuperclass() == r    &&
  r.getClass()      == c    &&
  c.getClass()      == c );     // true
}}
```

Furthermore, the `Class` class, being declared final, cannot have strict subclasses. We can therefore conjecture that `java.lang.Class` is the only metaclass in Java.

Indeed, this is exactly what the F&D book [59] states. Quoting from page 42: "Java has a single metaclass `Class` of which all classes are instances." This statement is confirmed in [60] and in particular in another book by Ira R. Forman, this time co-authored by Nate Forman, and titled *Java Reflection in Action* [61]. The book appeared in 2005, seven years after the F&D book, with praiseful notes from Doug Lea [𝕎66] and John Vlissides [𝕎67]. Quoting from page 23 of [61]:

> "`Class` is an example of a *metaclass*, which is a term used to describe classes whose instances are classes. `Class` is Java's only metaclass."

Given this, an unexperienced computer scientist would tend to take for granted that `java.lang.Class` is a metaclass and the only metaclass in Java. (Whereas experienced computer scientists just *pre*tend to take it for granted.)

Are classes objects?

But can instances of a Java class be classes? Since "class instances" is synonymous to "objects" the question can be stated as:

> Are Java classes (among) objects?

Unfortunately, the book *Java reflection in action* [61] gives two different answers. According to page 23, "*classes are objects*". According to page 268, there is a "*distinction between class and class object*". Both attitudes can be (separately) observed in other Java-related publications (cf. [110] [176] versus [130] [80] [15]). However, the prevailing view is that in Java, classes are NOT objects. This is in particular supported by the Java Language Specification [70]:

> "The method `getClass` returns the `Class` object that represents the class of the object." (§4.3.2 [70a]).

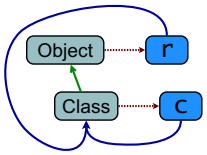
Similarly, the word "represent" is used in the API specification of `java.lang.Class` [127a]: "Instances of the class `Class` represent classes and interfaces in a running Java application." Exactly these arguments were presented by Jason Orendorff when discussing the section's title question with Ira R. Forman [𝕎34]. It follows that instances of `java.lang.Class` are not classes, they just *represent* classes. As a consequence, `java.lang.Class` does not meet the classic definition of a metaclass. This fact has eventually been recognized also in the book [61]. On page 259, an adjusted definition is provided according to which a *metaclass* is "*a class or class object whose instances are class objects*". Interestingly, this is a glossary definition which does not appear in the regular text.

Class reification

Let us share the common view that in Java, classes are not objects, they are just represented by objects. That is, classes are *reified* by objects. For a class *x*, let us denote *x.reif* the *reification* of *x*. (The object *x.reif* is usually called the *class object* of the class *x*.)

Java uses the very strange syntax `<s>.class` for the reification of the *s*-named class. In

the code above, assignments `r = Object.class` and `c = Class.class` let `r` be the reification of the `Object` class and `c` be the reification of the `Class` class. Note also that the distinction between class and its class object, although quite consistently adhered to by Java documentation, is not reflected by the names of introspection methods: for an object *x*, `x.getClass()` does not evaluate to the class of *x* but to the reification of the class of *x*.



Now we can attempt to further adjust the classic definition so that it works even with respect to a reification:

Metaclasses are classes all of whose potential instances are reifications of classes.

Unfortunately, the `java.lang.Class` does not meet this definition either, at least not if we wish to be consistent with established Java terminology. There are instances of the Java's `Class` class that are not the reification of a class. Instead, they are the reification of an *interface*, like e.g. the `Serializable` interface from the `java.io` package. Java uses the same syntax for the reification of interfaces so that `java.io.Serializable.class` is an instance of `java.lang.Class` – this establishes a counterexample to the adjusted definition.

Java core structure

In this subsection we present what can be regarded as the Java's counterpart to Python core structures. We are only interested in objects as runtime entities and want to describe the structure of blue and green links between them. For the sake of compatibility with the terminology introduced in the F&D model we have to part with the Java's terminological duality for classes (interfaces) and their runtime counterparts. That is, from now on we let classes and interfaces be objects, and consider the `<s>.class` expression to be just an inconvenient way to refer to the class or interface named *s*.

For simplicity, we exclude the nine distinguished instances of the `Class` class that represent primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`) or `void`. To handle these objects properly (i.e. as classes and thus descendants of *r*) it would be necessary to introduce an additional ancestor of the `Object` class – just like in the case of the Scala programming language which rectifies these Java's conceptual shortcomings by introducing the `Any` class as the new inheritance root.

The definition below is a modification of that of Smalltalk-76 core structures. The set C is again defined as $r.\mathrm{I}$, but the term "class" is used only for objects from a distinguished subset $\dot{C}$ of C. The complementary subset is that of interfaces. To establish such a distinction an additional definitory constituent is required. This constituent could be directly set to $\dot{C}$. However, it turns out that $\dot{C}$ forms a closure system in $(C, \leq)$ – for every interface x there is a least class y that is an ancestor of x. This defines a closure operator, $.c$, that is actually used in the signature. (In an analogy to $\leq$, we let $.c$ be a total map on O so that ordinary objects become fixpoints of $.c$, in addition to classes.)

A *Java core structure* is a structure $(O, .class, \leq, r, .c)$ where

- O is a set of *objects*,
- $.class$ is the *class map* between objects
- $\leq$ is the *inheritance* relation between objects,
- r is the *inheritance root*, a distinguished object,
- $.c$ is a map between objects.

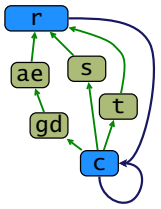
Denote C the set $r.\mathrm{I}$ of all descendants of r, and $c = r.class$ (the `Class` class). Let $\dot{C}$ be the set $C \cap O.c$ of *classes*. Objects from $C \setminus O.c$ are *interfaces*. The structure is subject to the axioms in the box on the right.

(ja~1) $(O, \leq)$ is a partial order.
(ja~2) $\dot{C}.class = \{c\} = c.\mathrm{I}$.
(ja~3) Objects from $O \setminus C$ are minimal in $\leq$.
(ja~4) $O.class \subseteq \dot{C}$.
(ja~5) The set $c.\mathrm{I} \cap \dot{C}$ contains exactly 2 objects.
(ja~6) $(\dot{C}, \leq)$ is a tree.
(ja~7) There are only finitely many objects.
(ja~8) The $.c$ map is a closure operator w.r.t. $\leq$.
(ja~9) $(C \setminus O.c).c = \{r\}$.

The last axiom says that in Java, the non-identity part of $.c$ is simply a constant map to $\{r\}$. That is, interfaces are not interleaved with classes. This is in contrast to Scala, where a *trait* (the Scala's counterpart to Java's interface) can in general inherit from any class that is not final.

It can be observed that Smalltalk-76 core structures are a special case of Java core structures – those without interfaces (i.e. such that *.c* is identity, equivalently, *C* = *C*).

The diagram on the right shows the built-in circular structure of Java 8. The six objects, four of which are interfaces, are exactly the objects *x* such that *x* ∈ *x* (that is, `x.isInstance(x)` evaluates to `true`). The *.class* map is shown in the restriction to classes.




```
Class
r  = Object.class,
c  = Class.class,
ae = AnnotatedElement.class,
gd = GenericDeclaration.class,
s  = Serializable.class,
t  = Type.class;
```

What is a metaclass?

Metaclasses in a Java core structure $\mathcal{S} = (O, .class, \leq, .c)$ are delimited the same way as metaclasses in a Python core structure: as the descendants of *c*. Since axiom (ja~2) asserts that there are no descendants of *c* other than *c* itself, it follows that *c* is the only metaclass in any Java core structure. If the reification map *.reif* is regarded as isomorphism between Java core structures, then it could be said that `java.lang.Class` is the only metaclass in Java.

The definition of metaclasses as descendants of classes of classes applies as well. However, the classic definition has to be adjusted. To do this, we apply the classifier naming convention for the `Class` class: instances of this class are referred to as `Class`es. The proper adjustment of the classic definition is then as follows:

> Metaclasses are classes all of whose potential instances are `Class`es.

This definition applies both to Python and Java core structures, provided that the *c* class is named `Class`.

Note: In [122], the built-in Java classes `Method` and `Field` from the `java.lang.reflect` package are given as examples of metaclasses. However, in subsequent considerations applied within a similar model for a class named `Property` the author concedes that "the term *metaentity* might be more appropriate".

The Perl's `isa`

Let us try to recognize the core structure of instantiation and inheritance in Perl. Unfortunately, the Perl programming language (as of both Perl 5 and Perl 6) makes the distillation of blue and green links much harder than Python, CLOS or Java. Perl does not seem to properly distinguish between the two fundamental relations, which can be seen as a possible consequence of the *is-a* misnomer.

In what follows we only consider classes and their instances as they are (presumably) understood in Perl. That is, in Perl 5, "class" is synonymous to "package" and can be created using the `package` keyword. An instance of a class *x* is an entity that is "*blessed into*" *x* using the built-in `bless` subroutine. In Perl 6, a class *x* is an entity created using the `class` keyword. Instances of *x* are created by invocation of the `new` method on them: *x*`.new` evaluates to a new instance of *x*. We regard Perl's classes and their instances as objects. This uniformity is based on a substantial common property of these entities, namely that both are potential *invocants*, with a uniform method invocation semantics. If an *s*-named method is invoked upon *x* (by *x*`-><`*s*`>(…)` in Perl 5 or by *x*`.<`*s*`>(…)` in Perl 6) then *x* is passed to the body of <*s*> in a uniform way (either as `$_[0]` in Perl 5 or as `self` in Perl 6).

It is therefore possible to make judgements about the ∈ relation based on responsiveness to named methods. This leads to the revelation of circularity of Perl's classes:

Every class responds to its own named methods.

In contrast to Python, there is true responsiveness because the invocant is passed to the method. Since every class *x* responds to a named method *(s,f)* owned by *x*, even if there is a named method *(s,g)* owned by any other class *y* (potentially, *y* can be such that *x* ∈ *y*), it follows that every class *x* equals its own class. In Perl 6, this statement is confirmed by the idempotency of the `WHAT` introspection method: For every object *x*,

x`.WHAT.WHAT === `*x*`.WHAT`.

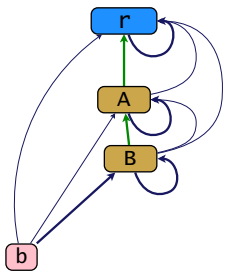
Finally, since the *.class* map is idempotent, it follows that in the restriction to classes, ∈ is coincident with ≤.

The diagram below shows a sample structure based on the above reasoning. The blue arrows indicate the ∈ relation, thickness of arrows distinguishes the *.class* map. In the case of Perl 6, the `Mu` class (which is the

inheritance root) should also be displayed to make the structure closed w.r.t. $.\updownarrow$.

(Perl 5)

```
package UNIVERSAL
          { sub new { bless {},$_[0] }}
package A { sub me { $_[0] } }
package B { our @ISA = A }
my
$r = UNIVERSAL,
$b = B->new;
print ($b->me == $b &&
       B->me == B  &&
       A->me == A);   # 1
```



(Perl 6)

```
class A { method me { self } }
class B is A {}
my $r = Any;
my $b = B.new;
say ($b.me === $b &&
     B.me === B  &&
     A.me === A);     # True
say ($b.WHAT === B &&
     B.WHAT === B &&
     A.WHAT === A);   # True
```

The `isa` method

The $\in$ relation can be detected by the `isa` introspection method which is present in both Perl 5 and Perl 6. (However, as of Rakudo Star Release 2015.06, the method does not work properly in Perl 6 for ordinary objects x. It seems that the check refers to $x.class$ instead of to x so that e.g. `1.isa(0)` evaluates to `True`.) A thorough description of Perl's 5 `isa` can found in [177]:

> "`isa()` returns a true value if its invocant is or derives from the named class, or if the invocant is a blessed reference to the given type."

That is, `isa` is meant to serve two different purposes: (a) *instance-of* (the correct "is-a"), (b) *descendant-of* (the misnamed "is-a"). Due to the Perl's circularity, (b) becomes a special case of (a) so that one can think that there is no misnomer. But it can hardly be considered as intentional since no documentation states that Perl classes are instances of themselves. (Possibly except for the mysterious first paragraph of the *Metaclasses* subsection of [175a] which seems to suggest something in that sense.)

Perl core structure

The box on the right shows a formalization of core structures that corresponds to the above informal description. Note in particular that there is just one definitory constituent: $\in$. By definitional extension, we let

- $\underline{C} = \underline{O}.\in$ be the set of *classes*,
- $(\underline{O}, \leq) = (\underline{C}, \in) \cup (\underline{O}, =)$ be the *inheritance* relation,
- $.class$ be the unique map such that $(.class) \circledcirc (\leq) = (\in)$.

In the restriction to the set $\underline{C}$ of classes, $\in$ is coincident with $\leq$. As a consequence, every class x is circular: $x \in x$ and even $x.class = x$. Let us apply the Python's terminology and say that every class is its own instance.

A *Perl core structure* is a structure $(\underline{O}, \in)$ where

- $\underline{O}$ is a set of *objects*,
- $\in$ the *is-a* relation between objects,

such that the following are satisfied:

(pl~1) $(\underline{O}.\in, \in)$ is a partial order.

(pl~2) There is a unique object $\underline{r}$ such that $\underline{O} = \underline{r}.\ni$.

(pl~3) For every object x, $x.\in$ has a bottom.

(pl~4) There are only finitely many objects.

Observation: Perl core structures can be axiomatized in the signature $(\underline{O}, .class, \leq)$ using a slight alteration of (py~1)–(py~7):

- The set $\underline{C}$ is defined differently as $\underline{O}.\in.\updownarrow$.
- Axioms (py~5) and (py~6) are replaced by the requirement of idempotency od $.class$.

Metaclasses in Perl 5

Recall that in Python core structures, there are 3 equivalent definitions of metaclasses as

(1) *classes all of whose potential instances are classes,*
(2) *descendants of $\underline{r}.class$,*
(3) *classes of classes and their descendants.*

In Perl, these definitions become different. According to (1), there are no metaclasses since nothing prevents the creation of an ordinary object that is a direct instance of an arbitrary given class. According to (2) and (3), all classes are metaclasses. Here we regard the (adjusted) primary definition (1) as the authoritative one. The "no metaclasses" characteristics can be found in [40].

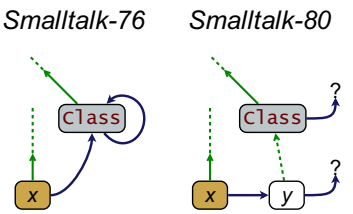
Metaclasses in Perl 6

Since we consider the Perl core structure to be applicable to both Perl 5 and Perl 6 there are no metaclasses in the sense of this structure, both in Perl 5 and Perl 6. However, Perl 6 introduces its own concept of a metaclass [175] which can presumably be described by referring to *metaobjects*:

> *A metaclass is the class of a metaobject.*
> *A metaclass is a class all of whose potential instances are metaobjects.*

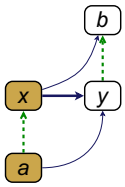
Depending on the further development of the Perl 6 specification, we might provide a more specific description of the term in a next version of this document.

The Smalltalk-80 dialectic



Smalltalk-80 is the first popular (if not the first ever) programming language that uses the *metaclass* term for a distinguished set of objects. It is therefore of fundamental importance to our investigation. According to James Althoff, the inventor of metaclasses in Smalltalk-80, the purpose of the facility is to overcome the limitation of Smalltalk-76 of each class responding to exactly the same methods – a consequence of each class being a direct instance of the same class, the `Class` class [5]. The solution was to equip each class x with an individual container y, a descendant of `Class` termed "metaclass", which allows to define individual methods to which x responds, just like x allows to define methods that are only received by instances of x. As a result, metaclasses as known from Smalltalk-80 are *implicit*: each metaclass y appears as an implicit container of its unique direct member x.

Powerclass link



Smalltalk-80 establishes a one-to-one correspondence for every pair (x,y) that is subject to the above considerations. The correspondence is given by the following equalities:

$x.\mathfrak{l} = y.\ni$ $\quad (a \le x \leftrightarrow a \in y,$ i.e. x is the highest member of y), and
$x.\in = y.\mathfrak{l}$ $\quad (x \in b \leftrightarrow y \le b,$ i.e. y is the least container of x).

As for Pharo or Squeak, the bidirectional links given by the correspondence can be expressed as x `==` y `thisClass` and x `class ==` y. (See the next subsection for the basic introspection methods.)

The $x.\mathfrak{l} = y.\ni$ equality can be read as: *members of y are exactly the subclasses of x* (i.e. descendants of x). This exposes an analogy to the relationship between a set X and its powerset $\mathbb{P}(X)$: *members of $\mathbb{P}(X)$ are exactly the subsets of X*. (That is, $A \subseteq X \leftrightarrow A \in \mathbb{P}(X)$.) We can therefore call the above pair (x,y) of objects a *powerclass link* and say that y is the *powerclass* of x. We should point out that these two terms do not appear in any publication about Smalltalk-80.

In set theory, the powerset operator $\mathbb{P}$ can be applied repeatedly, forming an infinite chain of sets

$X \in \mathbb{P}(X) \in \mathbb{P}(\mathbb{P}(X)) \in \cdots$.

As a conceptual counterpart, there is an infinite regress of powerclass links. However, Smalltalk-80 only supports a *fixed evaluation of this infinite regress* – exactly one step is implemented for each Smalltalk-76-correspondent class. Unfortunately, this has not been reflected in the Smalltalk-80 documentation. Instead, the specific implementation has been mistaken for a concept. This in turn resulted in inconsistencies in Smalltalk-80 terminology which also affect the definition of metaclasses.

Recognition of ∈ and ≤

The structure of blue and green links in Smalltalk-80 can be easily recognized via the `class` and `superclass` introspection methods. For every objects x, y, there is

- a blue link from x to y iff x `class ==` y (iff x `isMemberOf:` y),
- a green link from x to y iff x `superclass ==` y.

In addition, the membership relation ∈ is introspected by `isKindOf` [120].

The key question

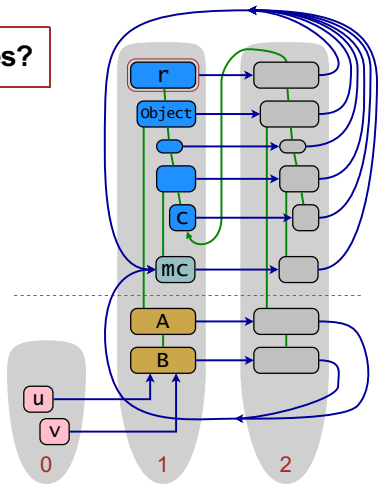
The main problem with the blue and green links lies in the terminology by which they are described. The key question sounds: ***Which objects are classes?*** There is also a companion question: *Which objects are metaclasses?*

The diagram below shows a sample structure with just 18 objects so that each object can be visually tested against the "being a class" predicate. The set of objects is partitioned into *metalevels* 0, 1 and 2 according to the reachability of the object denoted `mc` (and named `Metaclass`) via blue links. Metalevel 2 consists of direct members of `mc`, metalevel 1 consists of direct members of objects from metalevel 1, the remaining objects are from metalevel 0. The dashed horizontal line indicates a division into the built-in and user-created part. Let us call the built-in part the *circular core*. Each of the 12 objects from this substructure appears in a cycle formed by a combination of blue and green links.

Which of the 18 objects are classes?

(Pharo 1.3 / Squeak 4.2)

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



Unfortunately, there is no publication about Smalltalk that would provide a clear answer. Instead, the following two different answers are supported simultaneously:

> (A) Classes are the objects from metalevels 1 and 2.
> (B) Classes are the objects from metalevel 1.

Similarly, also the companion question has two different answers, dependent on the answer to the key question.

> (A) Metaclasses are the objects from metalevel 2 together with the `Metaclass` class.
> (B) Metaclasses are the objects from metalevel 2.

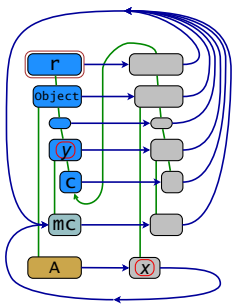
Typically, the (A) answer is provided first. It appears as a consequence of the uniformity principles stated in the raw explanation of the classic definition of metaclasses: (i) Every object is an instance of a class. (ii) Classes are objects too. (iii) It follows that classes are instances of classes. Since the "instance-of" relation is understood in the strict sense as "direct instance-of" it follows that everything pointed to by a blue arrow must be a class.

Once the (A) answer is established it is usually forgotten in favor of the more elegant description provided by the (B) answer. First, it is ignored that `Metaclass` meets the definition of a class whose instances are classes according to the (A) answer. Subsequently, the correspondence between metalevels 1 and 2 is formulated as e.g.: *Inheritance of metaclasses parallels inheritance of classes.* This has in particular disastrous consequences for the semantics of the `class` introspection method:

- if *x* is a class (i.e. belongs to metalevel 1) then *x* `class` is NOT a class (since it belongs to metalevel 2).

The monotonicity break

Let us further stick to the sample structure from the previous subsection with a hope that the structure captures the general properties of Smalltalk-80 blue and green links. Observe a consequence of the (B) answer to the key question: classes and metaclasses form disjoint sets. This disjointness is not only expressed in the documentation (where – as pointed out above – an inclusion relationship is usually stated first) but is in fact hard-wired into the core structure:



- The `Class` class (`c`) is the classifier for classes (i.e. objects from metalevel 1),
- the `Metaclass` class (`mc`) is the classifier for metaclasses (i.e. objects from metalevel 2).

That is, classes are exactly the `Class`es and metaclasses are exactly the `Metaclass`es. This is only possible by breaking the monotonicity of blue links. The implication

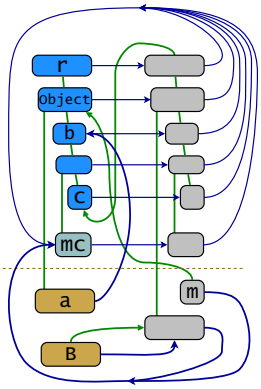
$$x \le y \quad \rightarrow \quad (x \text{ class}) \le (y \text{ class})$$

does not hold if y is an ancestor of `Class` and x is from metalevel 2. (In the sample structure, the implication is satisfied exactly for all the remaining pairs (x,y).)

Jungle structures

What combinations of blue and green links are allowed in Smalltalk-80? Let us provide an honest answer:

Nobody knows.



(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).
```

Both Pharo and Squeak, the two major Smalltalk-80 implementations, provide several ways to create structures fundamentally different from the sample. The diagram on the right shows that (1) instantiating the class named `Behavior` (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 so that it would point to (presumably) arbitrary non-terminal object. As a result, cycles in inheritance can arise.

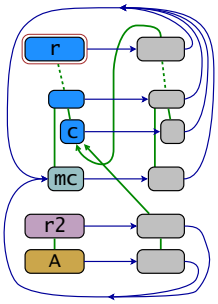
There are no guidelines in the Smalltalk-80 literature that would specify which structures should be regarded as standard. One has to make private assumptions. To rule out the anomalies, we only consider structures created

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

However, it should be noted that these assumptions can appear as too strict, disallowing some existing Smalltalk-80 extensions (see e.g. [112] [113], figure 5.3).

Subsidiary root

We of course want to take into account the built-in structure. It turns out that the major implementations of Smalltalk-80 have a quirk that cannot be considered to be a standard extension of the circular core. Each of Pharo and Squeak contains a "subsidiary" root – a built-in parentless class r_2 other than the inheritance root r. (The r_2 class is named `PseudoContext` in Pharo and `ObjectTracer` in Squeak). Being in metalevel 1, r_2 comes equipped with its powerclass from metalevel 2. This object is a direct inheritance descendant of the `Class` class, just like in the case of the powerclass of r.



(Pharo 1.3)

```
r  := ProtoObject.
c  := Class.
mc := Metaclass.
r2 := PseudoContext.
r2 subclass: #A.
```

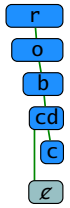
Core structures cultivated

At this point we have established, at least informally, sufficient conditions for a consistent description of Smalltalk-80 core structures. We allow (a) built-in monotonicity breaks and (b) subsidiary roots, but rule out the "jungle structures". In particular, there is an order-isomorphism between metalevels 1 and 2. For every objects x, y from metalevel 1,

$$x \leq y \quad \leftrightarrow \quad (x\ \texttt{class}) \leq (y\ \texttt{class}).$$

A formal description is provided below.

A *Smalltalk-80 core structure* is a structure $(O, .a\mathfrak{class}, \leq, \mathfrak{C})$ where O is a set of *objects*, $\leq$ is the *inheritance* relation between objects, $.a\mathfrak{class}$ is a map between objects (the blue links), and $\mathfrak{C}$ is a distinguished object (the `Metaclass` class). Objects from $\mathfrak{C}.a\mathfrak{class}(-1)$ constitute *metalevel 2*, the set $c.a\mathfrak{class}(-2)$ is *metalevel 1*. The remaining objects constitute *metalevel 0*. Let $\in$ be the *membership* relation, equal to $(.a\mathfrak{class}) \circ (\leq)$. Objects involved in a cycle of $\in$ are *circular*. The structure is subject to the following conditions.



(smt~1) $(O, \leq)$ is a finite forest – a finite partial order such that $x.\uparrow$ is a chain for every x.

(smt~2) Metalevel 0 is an antichain w.r.t $\leq$.

(smt~3) In a restriction, $.a\mathfrak{class}$ is an order-isomorphism between metalevels 1 and 2 w.r.t. $\leq$.

(smt~4) Inheritance between circular objects from metalevel 1 is given by the diagram on the right.

(smt~5) Every object from metalevel 1 is a member of `Class` (c).

(smt~6) Every descendant of `Behavior` (b) that is from metalevel 1 is circular.

(smt~7) Every member of `Behavior` is from metalevel 1 or 2.

Note that (smt~5) can be stated as: Every top of metalevel 2 is a direct strict descendant of `Class`.

Retrofit of terminology and notation

We now rectify the Smalltalk-80 definitions so that the correspondence to Python core structures (and thus to Smalltalk-76 core structures) can be established. The clue is to adhere to the (B) answer to the key question, that is:

- classes are the objects from metalevel 1.

Accordingly, the class of an object x is the least container of x that is a class. As a result, the class of every class is constantly the `Class` class just like in Smalltalk-76. This allows us to view Smalltalk-80 core structures as a definitional refinement of Smalltalk-76 core structures. As in Python core structures, we let C be the set of classes, $.class$ be the class map, r be the inheritance root and c be equal to $r.class$. We do not make any adjustments to the membership relation, $\in$ (so that it is what can be introspected by `isKindOf`). However, the *instance-of* relation is now the range-restriction of $\in$ to classes, i.e.

- x is an instance of $y \quad \leftrightarrow \quad x \in y$ and y is a class.

What is a metaclass?

There are several possibilities how to delimit metaclasses in Smalltalk-80:

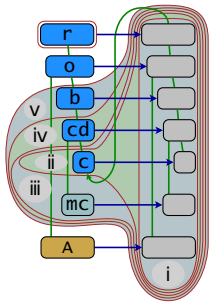
i. Metaclasses are the objects from metalevel 2.

ii. Metaclasses form the set $c.\downarrow$ of descendants of c and thus the set $C.class.\downarrow$.

iii. Metaclasses form the set $c.\downarrow.class.\downarrow$.

iv. Metaclasses form the set $(\bigvee c.\downarrow.class.\downarrow).\downarrow$.

v. Metaclasses are the non-terminal objects all of whose potential members are non-terminal.



As of Pharo or Squeak, the corresponding sets form a strict inclusion chain according to the diagram on the right. We distinguish between

- *implicit metaclasses* – the objects from metalevel 2, and
- *explicit metaclasses* – the classes named `Class`, `Metaclass`, `ClassDescription` and `Behavior` (the

amount is given by the choice of i–v).

The (v) delimitation appears in [112] [113]. (However, there are additional descendants of *c* from metalevel 1, which we disallowed.) In these documents, "metaclasses" is the term used for implicit metaclasses and "meta-classes" are the explicit metaclasses.

Resistant definition of metaclasses

The last proposed delimitation of metaclasses in Smalltalk-80 can be regarded as a resistant form of the classic definition. Let us state it once more:

> Metaclasses are the non-terminal objects all of whose potential members are non-terminal.

The definition is resistant against violation of the following conditions:

(a) Non-terminal objects are the `Class`es.
(b) Non-terminal objects are the classes.

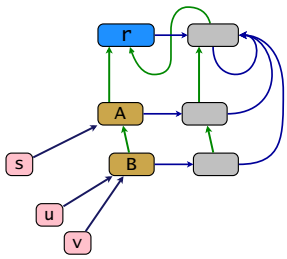
If (a) is satisfied, i.e. there is a class named `Class` that is a classifier for the set of all non-terminals, then we can (at least) replace "non-terminal objects" with "`Class`es":

- Metaclasses are the `Class`es all of whose potential members are `Class`es.

If (b) is satisfied then we can replace "non-terminal objects" with "classes" and, since ∈ coincides with *instance-of* in this case, also "members" with "instances":

- Metaclasses are the classes all of whose potential instances are classes.

Next Step: Objective-C



The Objective-C programming language [34] [35] adopted the Smalltalk's concept of a parallel hierarchy of implicit metaclasses. Simultaneously, the first step towards the purification of the Smalltalk-80 core structure has been accomplished by removing the monotonicity break. There is no `Metaclass` class in Objective-C. Instead, every implicit metaclass is a direct member of a top implicit metaclass. As a consequence, blue links form again a tree like in a Python core structure.

There are two specifics of Objective-C core structure that complicate the description. Nevertheless, they are only minor obstacles in comparison to non-monotonicity.

- There are multiple inheritance roots, one for each component of ∈. As of GNUstep, there are (at least) 3 built-in inheritance roots, named `Object`, `NSObject` and `NSProxy`.
- Inheritance roots are the only circular classes. Each inheritance root *r* is a direct ancestor of its correspondent implicit metaclass, so that

  $$r.class = r$$

  where *.class* is the corrected class map (such that *x.class* is the least container of *x* that is a class). As a consequence, there is no class that would correspond to `Class`.

The blue and green links are implemented as the `isa` and `super_class` pointers. There are corresponding introspection methods named `object_getClass` and `class_getSuperclass`. (However, a discrepancy may be introduced by some implementations [35b].) Interestingly, `NSObject` provides an introspection method `isKindOfClass` for the detection of the instance-of relation, rather than that of ∈. (Recall that *instance-of* is the range-restriction of ∈ to classes.)

What is a metaclass?

Due to the degeneracy of inheritance roots, all classes in Objective-C can have terminal objects as potential instances. This results in the following equivalent delimitations of metaclasses:

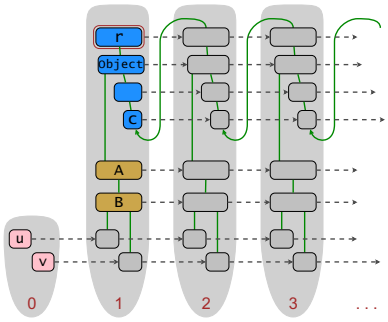
- Metaclasses are the objects from metalevel 2.
- Metaclasses are the non-terminal objects all of whose potential members are non-terminal.

The Rubification

The Ruby programming language [57] accomplishes a total purification of the Smalltalk-80 core structure. As of MRI (Matz's Ruby Interpreter) 1.9.1 and newer, the concept of powerclass links is applied universally: every object has a powerclass. This means that

- every class has a powerclass,
- every terminal object has a powerclass, and
- every powerclass has a powerclass.

As a consequence, there is an infinite regress of powerclass links as indicated by the gray dashed arrows (with a horizontal left-to-right direction) shown in the diagram on the right. The silver gray nodes indicate the *powerclasses* which the powerclass links point to.



Note: Versions 1.6 – 1.9.0 contain monotonicity breaks for powerclasses of powerclasses. Versions prior to 1.6 presumably do not support explicitly referenced powerclasses.

Core constituents

This subsection provides a brief overview of the Ruby core structure. A more detailed description can be found in [135]. See also [131] and [133] for alternatives.

Powerclass links form the *powerclass map* which is a total map between objects that we denote $.ec$. The core structure is then given by $(\underline{O}, .ec, \leq)$. That is, there is again an inheritance graph (the green links) but the instantiation graph (the blue links) has been replaced by powerclass links, a finer relation. The set $\underline{C}$ of *classes* equals $\underline{r}.\mathfrak{I} \setminus \underline{O}.ec$ so that objects are either terminals, classes or powerclasses:

$$\underline{O} = \underline{T} \uplus \underline{C} \uplus \underline{O}.ec.$$

In a correspondence, the *.class* map is a coarsement of the *.ec* map *derived* by

$$x.class = y \quad \leftrightarrow \quad x.ec.\mathfrak{I} \cap \underline{C} = y.\mathfrak{I},$$

that is, *x.class* is the least ancestor of *x.ec* that is a class. Informally, *x.class* is evaluated by following the superclass links from *x.ec* until a class is reached (cf. `rb_class_real` [7a]). It turns out that the whole structure $(\underline{O}, .ec, \leq)$ is given as the *powerclass completion* of the "front" structure $(\underline{O} \setminus \underline{O}.ec, .class, \leq)$: For every a, b from $\underline{O} \setminus \underline{O}.ec$ and every natural i, j,

$$a.ec(i) \leq b.ec(j) \quad \leftrightarrow \quad i \geq j \text{ and } a.class(i-j) \leq b.$$

These "front" structures are exactly the Smalltalk-76 core structures (up to the number of circular classes), see Smalltalk-76 ↔ Ruby correspondence. In particular, $\underline{C}.class = \{\underline{c}\}$ where $\underline{c} = \underline{r}.class$. (The class of every class is `Class`.)

The *object membership* relation $\in$ is defined as the composition of *.ec* with $\leq$. Totality of *.ec* then allows for a short expression of powerclass link equalities:

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

The *instance-of* relation equals $(.class) \circledcirc (\leq)$ and is thus a coarsement of $\in$.

There are infinitely many *metalevels*. For every object x, the powerclass of x is one metalevel higher than x. Similarly to Smalltalk-80, terminal objects form the metalevel 0, and classes belong to metalevel 1. However, the whole metalevel 1 equals $\underline{C} \uplus \underline{T}.ec$. For every natural i, the i-th metalevel equals $t.\ni \setminus t.\mathfrak{I}$ where $t = \underline{r}.ec(i)$ is the top of the $(i+1)$-th metalevel.

Terminology for powerclasses

There are several terms used for a powerclass within Ruby community.

- ***Eigenclass*** seems to be the public term officially supported by the author of Ruby [57] and also appeared

in a preliminary version of the Ruby Specification [86a]. The term arose in 2005 from a discussion among Ruby community.

- ***Singleton class*** is the term supported by the Ruby Specification [86] and in particular by the name of the introspection method: `x.singleton_class` evaluates to *x.ec*. Moreover, this term is also favored by most authors of books about Ruby (cf. [16] [166] [145]).
- The term ***metaclass*** has also been used as a synonym for the above terms, especially in the article titled *Seeing metaclasses clearly* [65] published in April 2005. This document initiated the above mentioned discussion about the correct name for Ruby's powerclasses [157a]. The documentation of the standard Ruby implementation [7a], gives the *metaclass* term a narrower meaning – it relates to powerclasses of `Class`es so that powerclasses of terminal objects are not metaclasses (see also [166]).

Generally, we allow "eigenclass of" *x* be synonymous to "powerclass of" *x* whenever it is asserted that the powerclass of *x* is the least container of *x*. That is, in the context of the monotonicity condition $(\leq) \circledcirc (\in) \subseteq (\in)$ (satisfied in Ruby), "eigenclasses" are the same as "powerclasses". This also explains the "e" and "c" symbols used in the *.ec* notation – the notation has first been used in the description of the Ruby object model [131]. (As a backronym, it can also be interpreted as "exponent class".)

We favor "eigenclass" over "singleton class" since the latter suggests that if *x* is a singleton class then *x* is a class – this is what we do not support by our definitions since it would establish an equality between *.class* and *.ec* (and be thus in a contradiction with the `class` introspection method).

Ruby core structure

The box below contains a rather concise axiomatization of the core structure of the Ruby object model (cf. [133] [131] [136] for less elegant alternatives).

A *Ruby core structure* is a structure $(O, .ec, \leq, r)$ where

- O is a set of *objects*,
- *.ec* is the *powerclass* map $O \to O$, (for an object *x*, *x.ec* is the *powerclass* of *x*)
- $\leq$ is the *inheritance* relation between objects, and
- r is a distinguished object.

The usual terminology and notation for $\leq$ applies. Objects from $O.ec$ are *powerclasses*, the remaining ones are *primary*. Objects that are not from $r.\downarrow$ are *terminal(s)*. The structure is subject to the following axioms.

(rb~1) $(O, \leq)$ is a partial order.

(rb~2) The powerclass map, *.ec*, is an order-embedding of $(O, \leq)$ into itself.

(rb~3) Terminal objects are minimal in $\leq$.

(rb~4) Every powerclass is a descendant of r.

(rb~5) Every object equals $a.ec(i)$ for a primary object a and a natural i.

(rb~6) There are only finitely many primary objects.

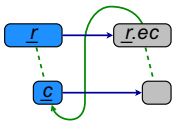
(rb~7) Descendants of powerclasses are powerclasses.

(rb~8) There is a (unique) *superclass* partial map *.sc* on O such that $(<) = (.sc) \circledcirc (\leq)$.

(rb~9) The set $c.\downarrow$ equals $\{c\} \cup r.ec.\downarrow$ where c denotes $r.ec.sc$.

(rb~10) The set $c.\updownarrow$ contains exactly 4 objects.

Axiom (rb~9) might be called the "twist link equivalence". It asserts that $r.ec$ is the only inheritance child of c. As a consequence, $r.ec$ and c have the same set of (potential) members so that c can be considered a "duplicate" of $r.ec$. The number 4 in the last axiom (rb~10) refers to number of circular classes. As of Ruby 1.9 (and newer), there is a following chain:



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

In contrast to Python and very much in contrast to Smalltalk-80, Ruby disallows any transition that would violate the above conditions. The code on the right shows that a powerclass cannot be instantiated or subclassed and that subclassing of `Class` is disallowed too.

```
ec = Object.singleton_class
ec.new            rescue p $!
Class.new(ec)     rescue p $!
Class.new(Class) rescue p $!
```

The Smalltalk-76 ↔ Ruby correspondence

In Smalltalk-76, there are exactly two circular classes while in Ruby 1.9, there are four of them. These numbers (2 or 4) appear in our axiomatization of respective core structures as the cardinality of $\underline{c}.\updownarrow$. Assume that the corresponding axioms are unified to: "$\underline{c}.\updownarrow$ contains at least two objects". Then there is a one-to-one correspondence between Smalltalk-76 core structures and Ruby core structures, as described by the following propositions.

1. Let $\mathcal{S}_0 = (\underline{O}_0, .class, \leq, \underline{r})$ be a Smalltalk-76 core structure and $\mathcal{S} = (\underline{O}, .ec, \leq, \underline{r})$ be a structure such that
   i. $(\underline{O}, \leq, \underline{r})$ is an extension of $(\underline{O}_0, \leq, \underline{r})$,
   ii. $.ec$ is an injective and well-founded map on $\underline{O}$,
   iii. $\underline{O}_0 = \underline{O} \setminus \underline{O}.ec$, and
   iv. for every a, b from $\underline{O}_0$ and every natural i, j, $\quad a.ec(i) \leq b.ec(j) \quad \leftrightarrow \quad i \geq j$ and $a.class(i\text{-}j) \leq b$.

   Then $\mathcal{S}$ is a Ruby core structure, up to the cardinality of $\underline{c}.\updownarrow$.
2. Let $\mathcal{S} = (\underline{O}, .ec, \leq, \underline{r})$ be a Ruby core structure, $.class$ be the class map in $\mathcal{S}$ and $\underline{O}_0$ be the set $\underline{O} \setminus \underline{O}.ec$ of primary objects. Then $(\underline{O}_0, .class, \leq, \underline{r})$ is a Smalltalk-76 core structure, up to the cardinality of $\underline{c}.\updownarrow$.

Introspection

Constituents of a Ruby core structure can be introspected in Ruby as follows.

```
x.class  =  x.class
   x.sc  =  x.superclass       if Class === x
   x.ec  =  x.singleton_class unless x.immediate_value?
 x < y   ↔  x < y              if Class === x && Class === y
 x ∈ y   ↔  x.is_a? y          if Class === y
```

It is assumed that the x object is an instance of `Object` which can be checked by `Object === x`. (The `===` introspection method, owned by the `Module` class, detects ∋). Introspection of powerclasses is not possible for *immediate values* which are the `Fixnum`s, `Symbol`s and the three objects `nil`, `false` and `true`. In the latter case, if x is one of `nil`, `false` or `true`, the expression (`y =`) `x.singleton_class` evaluates to the class of x. (As a consequence, the `singleton_class?` method, introduced in Ruby 2.1, reports `false` when invoked upon such y.)

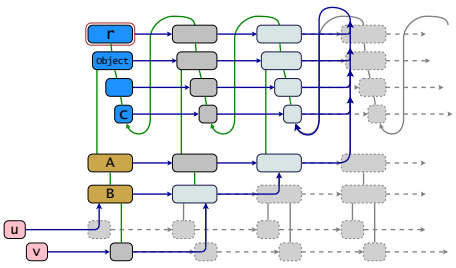
```
class Object; def immediate_value?
  [NilClass,
   FalseClass, TrueClass,
   Fixnum,
   Symbol].any? { |x| x === self }
end end
```

Finally, there is the `instance_of?` introspection method that detects the *direct-instance-of* relation and is thus a predicate version of the `class` introspection method. As a particular consequence, if y is the eigenclass of x then `x.instance_of?(y)` always evaluates to `false`. An object is never an instance of its powerclass.

Where are the blue links?

Naturally, the infinite regress of powerclass links must be lazily evaluated. There can only be finitely many objects that are actually represented (allocated). Let us call such objects *actual*. If A is the (finite) set of all actual objects then it is natural to assume that $\underline{T} \uplus \underline{C}$ is a subset of A. We can therefore define another coarsement of $.ec$, denoted $.aclass$, and called the *actualclass map*. Use

$$x.aclass = y \quad \leftrightarrow \quad x.ec.\updownarrow \cap A = y.\updownarrow.$$



That is, $x.aclass$, the *actualclass* of x, is the least ancestor of $x.ec$ that is actual. It follows by $\underline{C} \subseteq A \subset \underline{O}$ that for every object x,

$$x.ec \leq x.aclass \leq x.class.$$

We recognize $.aclass$ as the blue arrows in Ruby. Formally, this map must be introduced via an additional definitory constituent to Ruby core structures. While $.ec$ is a conceptual refinement of $.class$, the actualclass map is an implementation-oriented refinement. As a consequence, there is no introspection method for $.aclass$ in Ruby, much in contrast to Smalltalk-80 or Objective-C, in which $.aclass$ is mistaken for $.class$. As of MRI/YARV 1.9, the $.aclass$ links between allocated objects are maintained via the `klass` field in the C source.

What is a metaclass?

We have already mentioned that the documentation of Ruby's source code considers an object x to be a metaclass if it equals the eigenclass $x.ec$ of an object x that is not terminal. It has been objected in the 2005 discussion that this terminology does not meet the definition from the F&D book [157b].

To establish a compatibility with Python core structures we regard also the $\underline{c}$ class (named `Class`) to be a metaclass. This results in the following equivalent delimitations of metaclasses in Ruby.

- Metaclasses are the descendants of $\underline{c}$.
- Metaclasses are the classes of classes and their descendants.
- Metaclasses are the `Class`es all of whose potential members are `Class`es.

As with Smalltalk-80, we let *explicit metaclasses* to be those that are classes, the remaining ones are *implicit*. That is, implicit metaclasses are exactly the objects from metalevel 2 or higher. The `Class` class is the Ruby's only explicit metaclass.

Ruby's full *is-a*

Ruby allows to refine the core structure of superclass and eigenclass links by *module inclusion* which provides the facility of multiple inheritance. *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. Let $\underline{\mathrm{M}}$ denote the reflexive closure of this relation (i.e. $\underline{\mathrm{M}}$ is *self-or-own-includer-of*). Then the $\dot{\in}$ relation (the "full" membership or also *weak* membership) is given by

$$(\dot{\in}) = (\in) \circ \underline{\mathrm{M}}.$$

This extended membership corresponds to the `is_a?` introspection method. The "canonical" $\in$ relation equals the restriction of $\dot{\in}$ to `Class`es. Similarly, the "canonical" inheritance, $\leq$, is extended to $\dot{\leq}$ by

$$(\dot{\leq}) = (\leq) \circ \underline{\mathrm{M}}$$

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, $\dot{\leq}$ can have anomalies (as to transitivity and/or antisymmetry), the problem known as the double/dynamic inclusion problem. [54]

Metamodules

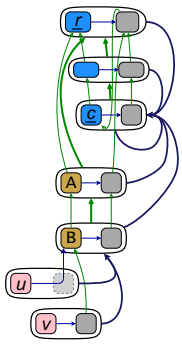
Just like `Class` descendants are metaclasses, descendants of the `Module` class can be regarded as *metamodules* – they are "metaclasses" w.r.t. $\dot{\in}$. The classic definition of a metaclass is translated to:

- A *metamodule* is a `Module` all of whose potential weak members are `Module`s.

Hidden powerclasses

Ruby supports the treatment of powerclasses as hidden entities. For an object x,

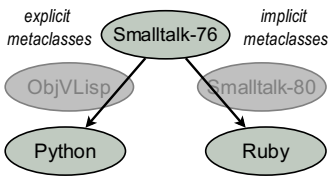
- methods owned by $x.ec$ can be defined (upserted) via x (use `def x.<s>(…); … end`),
- methods provided by $x.ec$ can be introspected via x (use `x.singleton_methods`),
- modules can be included into $x.ec$ via x (use `x.extend(y)` to include y into $x.ec$).



See [131] for details. As a result, most Ruby scripts do not need to use explicit references to powerclasses. (Early Ruby versions even disallowed such references. Presumably, referenceable powerclasses appeared first in version 1.6 which introduced the `class << x; … end` construct to "open" the powerclass of x.) The core structure of blue and green links can be considered to be defined between pairs x–$x.ec$ where x is a primary object (a terminal or a class) according to the diagram on the right. According to the Smalltalk-76↔Ruby correspondence, thick links between the pairs form a Smalltalk-76 core structure, up to the cardinality of $\underline{c}.\updownarrow$.

Methods owned by a class are termed *instance methods*, methods owned by a powerclass are *singleton methods*. In addition, methods owned by a module are of either of the two groups depending on what the module is included into.

The general monotonic core



Let us now return to the table that served as a schedule for the exploration of the metaclass term in object-oriented programming. The table indicated that there are two main directions in the evolution of object models in which the notion of a metaclass appears:

- *explicit* metaclasses, and
- *implicit* metaclasses.

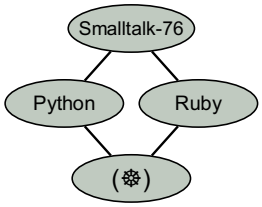
The diagram on the right provides a simplification by showing just the startpoint (Smalltalk-76) and the two respective endpoints (Python and Ruby), together with the first step for each direction.
We can now observe that both Python core structures and Ruby core structures are subject to monotonicity of ϵ:

$$(\leq) \circledcirc (\epsilon) \subseteq (\epsilon),$$

a consequence of the monotonicity of *.class* and *.ec* w.r.t. $\leq$. Also recall that in contrast to Python core structures a Ruby core structure is uniquely given by that of Smalltalk-76 via powerclass completion (up to the number of circular classes): For every primary objects a, b and every natural i, j,

$$a.ec(i) \leq b.ec(j) \quad \leftrightarrow \quad i \geq j \text{ and } a.class(i\text{-}j) \leq b.$$

Python and Ruby cores composed



The above construction can be generalized for Python core structures by the following prescription [135] [136]. For every primary objects a, b and every natural i, j,

$$a.ec(i) \leq b.ec(j) \quad \leftrightarrow \quad i \geq j \text{ and } a.class(i\text{-}j) \leq b \quad \text{or} \quad i{+}1 = j \text{ and } a < \underline{c} \text{ and } b = \underline{r}.$$

The resulting structures form the family of *tight canonical eigenclass structures* (this term is used in [135]) which are in a one-to-one correspondence to Python core structures. (Again, the family of Python core structures is considered in a slight generalization that allows additional classes between $\underline{c}$ and $\underline{r}$.) The "tight" adjective means $\underline{C}.\updownarrow \subseteq \underline{C} \cup \{\underline{r}\}.ec$, that is, $\underline{r}.ec$ is the only powerclass that is allowed to be an ancestor of a class. As a consequence, all classes reside on metalevels 1 and 2. Since this condition might appear as too restrictive, documents [135] [136] define canonical structures as a slight generalization of structures defined by the above powerclass completion, which accounts for the removal of "tight". We refer to the corresponding family by (☸).

The (☸) family can be axiomatized in the $(\underline{O}, .ec, \leq, \underline{r})$ signature. Ruby core structures form a subfamily given by additional constraints of single inheritance and a single explicit metaclass. Python core structures are obtained as the "front" substructures.

Monotonic powerclass structure

The essential structure of infinite powerclass regress is captured by the general family described below. (See also [$\mathbb{W}_{36}$] where the term "essential structure of ϵ" is used.)

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

- $\underline{O}$ is a set of *objects*,
- *.ec* is the *powerclass* map $\underline{O} \to \underline{O}$,
- $\leq$ is the *inheritance* relation between objects,
- $\underline{r}$ is the *inheritance root*, a distinguished object.

Let $\underline{T}$ be the set $\underline{O} \setminus \underline{r}.\downarrow$ of objects that are not descendants of $\underline{r}$ and let $.ec^*$ denote the reflexive transitive closure of *.ec*. The structure is subject to the five axioms on the right.

(e~1) $(\underline{O}, \leq)$ is a partial order.
(e~2) *.ec* is an order-embedding of $(\underline{O}, \leq)$ into itself.
(e~3) Objects from $\underline{T}.ec^*$ are minimal in $\leq$.
(e~4) Every powerclass is a descendant of $\underline{r}$.
(e~5) The set $\underline{r}.ec^*$ has no lower bound in $\leq$.

The (☸) family from the previous subsection forms a subfamily given by additional constraints:

- Simplicity and non-degeneracy of the circular substructure.
- Single classification, that is, the existence of a total *.class* map.

- Assertion that $\underline{r}.ec$ has no siblings and therefore $\underline{c}$ is a "duplicate" of $\underline{r}.ec$.
- The "front" substructure being a generator for (almost) the whole substructure via powerclass completion.
- Finiteness.

See [135] or [136] for the formal axioms.

Note: (∗) We also use *monotonic .ec-complete* structure or *monotonic eigenclass structure* equivalently.

Monotonic .ec-based structure

A finer family of structures is obtained by allowing $.ec$ to be partial. In particular, the current state of the evaluation of Ruby's powerclass regress can be expressed.

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

- $\underline{O}$ is a set of *objects*,
- $\in$ is the *(object) membership* relation,
- $\leq$ is the *inheritance* relation between objects,
- $\underline{r}$ is the *inheritance root*, a distinguished object, and
- $.ec$ is the partial *powerclass* map $\underline{O} \curvearrowright \underline{O}$.

Let $.ec^*$ be the reflexive transitive closure of $.ec$, let $.ce$ be the inverse of $.ec$. Let $\underline{T} = \underline{O} \setminus \underline{O}.\in.\mathrm{I}$ be the set of *terminal* objects. The structure is subject to the axioms on the right, with the $.mli$ function introduced below by further definitions. Let

$$(\in^{-1}) = (\leq) \circledcirc (.ce) \circledcirc (\leq)$$

be the *anti-membership* relation. For every natural k, let $\in^{-k}$ be the k-th relational composition of $\in^{-1}$ with itself. Moreover, let $\in^0$ be equal to $\leq$, so that $\in^i$ is defined for every integer i. We can then define the *metalevel index* function, $.mli$, from $\underline{O}$ to the set $\omega{+}1 = \omega \cup \{\omega\}$ (the natural numbers plus the first infinite ordinal) by

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

That is, the metalevel index of an object x equals either (a) the greatest natural number i such that $x \in^{1-i} \underline{r}$ or (b) ω if no such greatest i exists (i.e. if x is related to $\underline{r}$ by infinitely many negative powers of $\in$). Axiom (eb~7) just disallows the (b) case. For a natural i, the i-th *metalevel* consists of objects whose metalevel index is i. If $\underline{r}.ec(i)$ is defined then it equals the top of the $(i{+}1)$-st metalevel. $\underline{T}$ is the (possibly empty) 0-th metalevel. For each natural i, the set $\underline{r}.\ni^{-i}$ consists of objects x such that $x.mli > i$.

(eb~1) $(\underline{O}, \leq)$ is a partial order.

(eb~2) (a) $(\in) \circledcirc (\leq) \subseteq (\in)$. (subsumption of $\in$)
(b) $(\leq) \circledcirc (\in) \subseteq (\in)$. (monotonicity of $\in$)

(eb~3) (a) $(\in) \circledcirc (.ce) \subseteq (\leq)$. (If $x.ec = y$ then (a) $x.\mathrm{I} = y.\ni$,
(b) $(.ce) \circledcirc (\in) \subseteq (\leq)$. (b) $x.\in = y.\mathrm{I}$.)

(eb~4) The inheritance root $\underline{r}$ is the top of $\underline{O}.\in$, w.r.t. $\leq$.

(eb~5) Every object has a container, $\underline{O} = \underline{O}.\ni$.

(eb~6) Objects from $\underline{T}.ec^*$ are minimal in $\leq$.

(eb~7) Every object x has a finite metalevel index $x.mli$.

Proposition:

1. A monotonic $.ec$-complete structure is a monotonic $.ec$-based structure in which $.ec$ is total.
2. Every monotonic $.ec$-based structure $(\underline{O}_0, \in, \ldots)$ is embedded into a monotonic $.ec$-complete structure $(\underline{O}, \in, \leq, \ldots)$ by powerclass completion according to the following prescription for the inheritance:

   $$a.ec(i) \leq b.ec(j) \quad \leftrightarrow \quad a \in^{i-j} b.$$

   The embedding preserves the metalevel index.
3. If $\mathcal{S} = (A, \in, \leq, \underline{r}, .ec)$ is the restriction of a Ruby core structure to a set A of *actual* objects (with $.ec$ restricted as a relation) then $\mathcal{S}$ is a monotonic $.ec$-complete structure.
4. The is a one-to-one correspondence between Python core structures and monotonic $.ec$-based structures $(\underline{O}, \in, \leq, \underline{r}, .ec)$ such that (a) $\underline{O}$ is finite, (b) $\underline{O}.ce = \{\underline{r}\}$, (c) $\underline{r}.\in$ has exactly 3 objects: $\underline{r}.ec < \underline{c} < \underline{r}$, (d) $\underline{c}.\mathrm{I} = \{\underline{c}\} \cup \underline{r}.\ni^{-1}$, (e) $\in$ is acyclic outside $\underline{r}.\in$ and (f) $(\in) = (.aclass) \circledcirc (\leq)$ for a map $.aclass$ (coincident with $.class$ except for direct instances of $\underline{c}$).

What is a metaclass?

In the general families of monotonic $.ec$-based or $.ec$-complete structures we let metaclasses form exactly the set $\underline{r}.\ni^{-1}$. That is,

metaclasses are the objects from metalevel 2 or higher.

According to the terminology for $\in^{-1}$, metaclasses are the anti-members of the inheritance root. In a subfamily that asserts the existence of $\underline{r}.ec$ as well as the existence of a designed "duplicate" $\underline{c}$ of $\underline{r}.ec$ – i.e. $\underline{c}$ is a unique object such that $\underline{c}.\mathrm{I} = \{\underline{c}\} \uplus \underline{r}.ec.\mathrm{I}$ – we consider $\underline{c}$ to be a metaclass too:

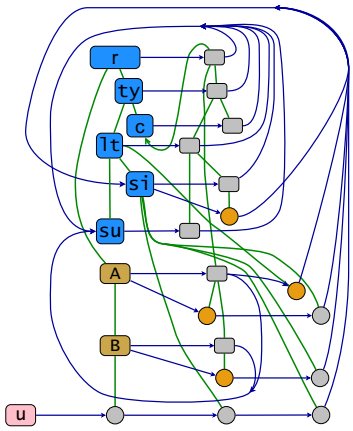
Metaclasses are the descendants of *r.ec* together with a "duplicate" *c* of *r.ec* if it exists.

If *c* exists (as is the case of the (☸) family) and is named `Class` then the three definitions stated for Ruby core structures apply. However, in contrast to Ruby core structures, the set $r.ec.\mathfrak{l} \cap C$ can be nonempty – that is, there can be explicit metaclasses other than *c*.

The singletons of Dylan

The Dylan programming language fully conforms to the metaclass pre-condition: classes are objects [58]. Indeed, the F&D book lists Dylan among languages that "*have single metaclasses in their cycles*" (page 15). However, the *metaclass* term appears only rarely in Dylan documentation. A single occurrence of "meta-class" can be found in [43] where this term is used for the `<type>` class.

The diagram on the right shows a sample core structure that can be created in Open Dylan (version 2013.2). Inheritance between non-terminal objects can be detected by the `subtype?` method. The composition of blue arrows with inheritance – the object membership ∈ in Dylan – is exactly what is detected by the `instance?` method. Labelled nodes indicate either classes (built-in classes in blue and user-created ones in sandy brown) or terminal objects (in pink). There are two kinds of unlabelled nodes.



- The unlabelled rounded rectangles are used to indicate the so called "*subclass types*", a facility introduced to Dylan in 1995 [148]. For a class *x*, the (rather horrendous) expression `subclass(x)` " *returns a type which describes all the objects representing subclasses of the given class* ". The blue arrows pointing to these nodes are horizontal and show the $x \mapsto$ `subclass(x)` mapping. All "subclass types" are direct instances of the class named `<subclass>` (`su` in the diagram).
- The circles indicate "*singleton types*" or shortly, *singletons*. For an object *x*, the expression `singleton(x)` "*creates a type whose only member is*" *x* [58]. The blue arrows pointing to circles show the $x \mapsto$ `singleton(x)` mapping. Orange color is used for the distinction of singletons of classes or "subclass types". (As a second way of the same distinction, the remaining circles are exactly those pointed to by *horizontal* blue arrows.) All singletons are direct instances of the class named `<singleton>` (`si`).

The built-in classes labelled by `r`, `ty` and `c` are named `<object>`, `<type>` and `<class>`, respectively. We can immediately observe a correspondence between Dylan's "subclass types" and Smalltalk-80 implicit metaclasses. The `<subclass>` class plays the same role as the `Metaclass` class in Smalltalk-80, and thus the same monotonicity break is introduced. However, expectedly enough, in contrast to Smalltalk-80, `subclass(x)` is not reported as the class of *x*. Instead, the class of every class is constantly the `<class>` class. This is because in Dylan, "subclass types" and singletons are consistently regarded as "non-class types". The `object-class` introspection method then correctly implements the *.class* map: For every object *x*,

- `object-class(x)` is the least container of *x* that is a class.

The infinite regress of singletons

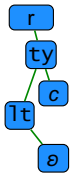
We now consider Dylan core structures without "subclass types" and focus on the properties of the singleton map.

A *Dylan core structure* is a structure $(O, \leq, r, .class, .\varepsilon c)$ where *O*, ≤, *r* and *.class* are the same as in Smalltalk-76 core structures (without axioms) and

- *.εc* is the *singleton* map between objects.

Objects from $O.\varepsilon c$ are *singletons*. Let $.\varepsilon c^*$ be the reflexive transitive closure of *.εc*, let *C* be the set $r.\mathfrak{l} \setminus O.\varepsilon c$ of *classes*, and let $c = r.class$ and $\mathfrak{e} = r.\varepsilon c.class$

(dy~1) $(O, \leq)$ is a partial order.
(dy~2) $C.class = \{c\} = c.\mathfrak{l} \cap C$.
(dy~3) Objects from $O \setminus C$ are minimal in ≤.
(dy~4) $O.class \subseteq C$.
(dy~5) $(O \setminus O.\varepsilon c).\varepsilon c^* = O$.
(dy~6) For every object *x*, $x.\varepsilon c < x.class$.
(dy~7) The singleton map, *.εc*, is injective.
(dy~8) The set $O \setminus O.\varepsilon c$ is finite.



be distinguished objects (named <class> and <singleton>, respectively). The structure is subject to the axioms in the box on the right.

(dy~9) $\underline{O}.\varepsilon c.class = \{\underline{\mathfrak{o}}\} = \underline{\mathfrak{o}}.\mathrm{I} \cap \underline{C}$.
(dy~10) $(\underline{c}.\mathrm{I} \cup \underline{\mathfrak{o}}.\mathrm{I}, \prec)$ is given by the diagram.

Similarities between $.\varepsilon c$ and the powerclass map $.ec$ from Ruby core structures can be observed. In particular,

- $\underline{O} = \underline{T} \uplus \underline{C} \uplus \underline{O}.\varepsilon c$ where $\underline{T}$ is defined as $\underline{O} \setminus \underline{r}.\mathrm{I}$,
- $.\varepsilon c$ is an injective well-founded map (and thus forms right-infinite chains),
- $\in$ equals $(.\varepsilon c) \circledcirc (\leq)$,
- $x.class = y \quad \leftrightarrow \quad x.\in \cap \underline{C} = y.\mathrm{I}$,
- the whole structure is given by *singleton completion* of the "front" structure $(\underline{O} \setminus \underline{O}.\varepsilon c, .class, \leq)$.

The prescription for the $.\varepsilon c$-completion is however rather dissimilar to that used for $.ec$-completion: For every a, b from $\underline{O} \setminus \underline{O}.\varepsilon c$ and every natural i, j,

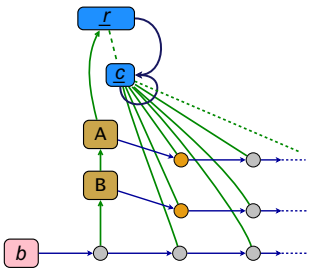
$$a.\varepsilon c(i) \leq b.\varepsilon c(j) \quad \leftrightarrow \quad i = 1 \text{ and } j = 0 \text{ and } a.class \leq b \quad \text{or} \quad i > 1 \text{ and } j = 0 \text{ and } b = \underline{\mathfrak{o}} \quad \text{or} \quad i = j \text{ and } a = b.$$

This is because the set $\underline{O}.\varepsilon c$ of all singletons has $\underline{\mathfrak{o}}$ (the class named <singleton>) as an *imposed* classifier, just like <subclass> is an imposed classifier for "subclass types". As a consequence, all singletons of singletons are descendants of $\underline{\mathfrak{o}}$ and thus (incorrectly) reside on metalevel 1 (which can be expressed as $\underline{O}.\varepsilon c(2) \cap \underline{c}.\mathrm{I} = \varnothing$).

Corrected structure

Let us specify corrections to the above structure that allow to establish a rigorous connection to set theory.

1. Declare $.\varepsilon c$ as a *partial* map between objects.
2. Let $\underline{\mathfrak{o}}$ be equal to $\underline{c}$ so that (dy~2) and (dy~9) can be expressed as
   $\underline{r}.\mathrm{I}.class = \{\underline{c}\} = \underline{c}.\mathrm{I} \cap \underline{C}$.
3. Instead of (dy~10), require that $\underline{c}$ contains at least 2 objects.
4. Add the following requirement: For every object x,
   $\underline{c} \leq x \quad \leftrightarrow \quad x.\varepsilon c$ is undefined.



The object membership relation, $\in$, then equals $((.class) \cup (.\varepsilon c)) \circledcirc (\leq)$. It can be observed that $x.\varepsilon c$ is defined if and only if x is a circular object. In the restriction to non-circular objects x, the $.class$ map can be defined implicitly:

$$x.class = y \quad \leftrightarrow \quad x.\varepsilon c \prec y.$$

Singleton completion is given by the following prescription: For every a, b from $\underline{O} \setminus \underline{O}.\varepsilon c$ and every natural i, j such that $a.\varepsilon c(i)$ and $b.\varepsilon c(j)$ are defined,

$$a.\varepsilon c(i) \leq b.\varepsilon c(j) \quad \leftrightarrow \quad j = 0 \text{ and } a.class(i) \leq b \quad \text{or} \quad i = j \text{ and } a = b.$$

What is a metaclass?

In the corrected structure, we let metaclasses form the set $\underline{c}.\mathrm{I}$. In an (uncorrected) Dylan core structure, we require that both $\underline{c}$ and $\underline{\mathfrak{o}}$ are metaclasses which leads to the <type> class (as the least common ancestor of $\underline{c}$ and $\underline{\mathfrak{o}}$) as a candidate for the top metaclass. This choice is in accordance with [43] and also seems to satisfy the resistant definition of metaclasses.

Metaclasses in Newspeak

The Newspeak programming language [23] adjusts the Smalltalk-80 core structure by flattening the metaclass hierarchy. According to the specification draft, all metaclasses are direct subclasses of `Class`. This way, every metaclass becomes a singleton. However, the word "singleton" does not appear in the specification, which makes Newspeak's terminology antipodal to that of Dylan in this respect.

The MCJ core

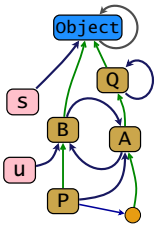
As of 2015, there are very few publications in which the term *metaclass* is put in connection with the notion of

calculus. Perhaps the most notable representant of this small group is the 2005 article titled *A Core Calculus of Metaclasses* [78]. Unfortunately, the occurrence of "metaclass" in the article's title is a consequence of what we call the *metaclass misnomer*. From this point of view the article does not show how (or prove that) "metaclass" and "calculus" (or "type") can be put in combination.

In the article, the authors present an adaptation of Featherweight Java [83], named MCJ, in which "*each class is an instance of a metaclass*" (section 2.1 of [78]). In addition to the "subclass" hierarchy as known from Java, there is a "kind" hierarchy: each class *x* has an "immediate containing class" also called *the kind of x*. (∗)
The declaration

```
class X kind Y extends Z { … }
```

is used to define a class `X` such that *X.class* = `Y` and *X.parents* = *{Z}*. Each class serves as a potential method provider to its instances. Since classes are instances (⁑), each class potentially responds to methods provided by containing classes. In addition, each class *x* can have "static" methods which take precedence over the instance methods provided by *x*'s containing classes. The "static" methods of *x* can be thought of as being provided by a virtual singleton of *x*, referred to by `typeOf[x]`. Similar resolution applies to "fields".




```
class Q kind Q {}
class A kind B extends Q {}
class B kind A {}
class P kind A extends B {}
s = new Object
u = new B
```

As shown by the diagram on the right, there can be (almost) arbitrary cycles in *.class*. Monotonicity of *.class* is not asserted. There is no `Class` class, and `Object` remains the only built-in class.

Notes:

1. (∗) In a contradiction to the quoted statement from article's section 2.1, the `Object` class is said to have no kind (section 4.1). We remedy this by assuming that `Object` is its own kind – see (mcj~2) below.
2. (⁑) We have to write "classes are instances" instead of the usual "classes are objects" since the latter phrase does not appear anywhere in the article's text. In fact, we can conjecture that the authors do not follow what can be thought of as a universally accepted OOP canon, namely that if something is an instance of a class then it is necessarily an object. (See Objects as instances.)

MCJ core structure

The family of core structures defined below is meant to provide an abstraction of MCJ. We deviate from the MCJ formalization at least in two respects. Firstly, we allow classes with undefined singletons. Secondly, we ignore the fact that in MCJ, just like in Featherweight Java, terminal objects are uniquely given by class declarations. (As one of the consequences, MCJ prevents `Object` from having multiple direct terminal instances.)

Let an *MCJ core structure* be a structure $(O, \leq, r, .class, .\varepsilon c)$ where

- O is a set of *objects*,
- $\leq$ is the *inheritance* relation between objects,
- r is the *inheritance root*, a distinguished object,
- $.class$ is the *class* map $O \to O$, and
- $.\varepsilon c$ is the *singleton* map $O \rightharpoonup O$.

Objects from $O.\varepsilon c$ are *singletons* and objects from $r.\mathrm{I} \setminus O.\varepsilon c$ form the set C of *classes*. The structure is subject to the axioms on the right. As with Dylan core structures, we let ϵ be equal to $((.class) \cup (.\varepsilon c)) \circledcirc (\leq)$.

(mcj~1) $(O, \leq)$ is a partial order.
(mcj~2) $r.class = r$.
(mcj~3) Objects from $O \setminus C$ are minimal in $\leq$.
(mcj~4) $O.class \cup O.\varepsilon c \subseteq r.\mathrm{I}$.
(mcj~5) $(.\varepsilon c) \circledcirc (<) \;\subseteq\; (.class) \circledcirc (\leq)$
(mcj~6) $.\varepsilon c$ is injective.
(mcj~7) $.\varepsilon c$ can only be defined for classes.
(mcj~8) $.\varepsilon c.class \subseteq .class.class$.
(mcj~9) $(C, \leq)$ is a tree.
(mcj~10) There are only finitely many objects.

In MCJ, r is a class named `Object`. Descendants of r are called "types". The $.\varepsilon c$ map is realized via the `typeOf` operator. Singletons are compile time entities (the "`typeOf` types"), classes are "non-`typeOf` types". The object membership relation, ϵ, corresponds to typing, "`:`", inheritance, $\leq$, is the subtyping, "`<:`". Conditions (mcj~2) and (mcj~8) form a completion of *.class* – in MCJ, $\{r\} \cup O.\varepsilon c$ are exactly the "types" x for which $x.class$ is undefined.

We also introduce an additional condition that ensures embeddability of MCJ core structures into metaobject structures introduced later.

(mcj~11) For every object x that is non-well founded in ϵ,
(a) $x.\varepsilon c$ is undefined, (b) $x.\mathrm{I} \subseteq y.\ni$ whenever $x \epsilon y$.

What is a metaclass?

In MCJ, every class can potentially have terminal instances, and thus no class is a metaclass according to the resistant definition. The only objects that meet the definition are the singletons (that is, the "`typeOf`-types" in the article's terminology).

Presumably, the MCJ authors regard an object x to be a metaclass iff it belongs to $C.class.\updownarrow$ – i.e. metaclasses are the classes of classes and the ancestors thereof. The following comparison can be drawn:

> Adjusted classic definition: *Metaclasses are classes all of whose potential instances are classes.*
> Presumed MCJ definition: *Metaclasses are classes some of whose instances are classes.*

Note that in the MCJ definition, the "potential" adjective is missing since it becomes redundant.

The metaclass misnomer

We regard the use of the "some" quantifier from the previous subsection as incorrect and call it the *metaclass misnomer*. The metaclass misnomer can be simply described as the assumption that **classes of classes** are necessarily metaclasses even if one of (or both) the following conditions are NOT asserted:

- non-degeneracy: $r.\in \neq \{r\}$ (i.e. $r.class \neq r$ – the inheritance root is not the class of itself),
- monotonicity: $(\leq) \circ (\in) \subseteq (\in)$ (the *.class* map is monotonic w.r.t. $\leq$).

The non-degeneracy condition should more precisely be expressed as the existence of two distinct classes r and c such that r is the top class (usually named `Object`) and c is such that

$$r.\downarrow = c.\ni,$$

i.e. c is the powerclass of r or its duplicate, thus becoming the top metaclass (usually named `Class`).

In MCJ, both conditions are violated.(∗) In particular, there is no `Class` class in MCJ. There is no need for the `Class` class exactly because the authors do not require that ***all*** instances of a metaclass should be classes. According to the authors, "*the `Class` class play[s] no special role, and […] thus the class and metaclass hierarchies could both be rooted at `Object`*".

In Python, both conditions are ensured so that there is no misnomer. In Perl, each class equals the class of itself which ensures monotonicity of *.class* but causes degeneracy at the same time – there is no `Class` class and thus no metaclass. In the field of knowledge representation it is usually the case that non-degeneracy is asserted and just the monotonicity condition is absent. In RDF Schema, there is a `rdfs:Class` class which is identified by some authors as the top metaclass to avoid the metaclass misnomer (see What is a metaclass in RDFS).

Notes:

1. (∗) We can express the non-degeneracy condition by $r.\in \not\subseteq \{r\}$ so that it remains unsatisfied even without the additional *.class* loop for `Object`.
2. Using the general set-theoretic model described below the metaclass misnomer can be characterized as confusing *metalevel* with *rank*, see the "all" versus "some" comparison.

The classy Self

Prototype-based programming languages (PBL), such as Self [173], are known to be "classless". Typically, such characteristics is based on statements of the following kind:

- In a PBL, methods are "*directly stored in objects*". [42] (Instead of being associated with classes.)
- "*Classes describe objects*" whereas "*Prototypes describe objects and are objects*" [30].
- In a PBL, " *there is one kind of objects equip[p]ed with attributes and methods, three primitives to create objects (…) and one control structure: message sending together with a delegation mechanism* " [47].

Apparently, this "classlessness" is based on the assumption that classes are not objects. If, on the contrary, classes are allowed to be objects, then prototype-based languages become "classful". More specifically, prototype-based languages can be characterized as those object-oriented programming languages in which

$$(\in) = (\leq),$$

i.e. object membership is coincident with the inheritance relation. Equivalently, every object equals its least

container. As a particular consequence, there are no terminal objects – every object is *non-terminal*. Recall that by default, if there is no finer structure which would yield distinguished subsets of non-terminal objects (e.g. by *.ec* in Ruby or by *.c* in Java) then non-terminal objects are exactly the classes. Therefore, prototype-based languages should by default be characterized by "anti-classlessness":

> Every object equals the class of itself.

In particular, every object is a class. New object creation (often called "cloning") can be considered to be a dynamic (as well as anonymous and generalized to encompass multiple inheritance) version of circular class declaration in MCJ: `class X kind X extends Y {}` is interpreted as "let *X* be a clone of *Y*".

Self core structure

In Self, the inheritance relation is derived from a distinguished part of what we call *instance variable valuation* and denote *.ιο()* (see **FD**$_4$ structures). However, the Self language documentation uses the term *slot*: if $x.\iota o(s) = y$ then x is an owner of the (s,y) slot where s is the name and y is the value. If, in addition, s has an asterisk (*) suffix then $x \prec y$ – i.e. y is considered to be an inheritance parent of x. There are no constraints put to *.ιο()* so that $\prec$ can be any relation between objects. In particular, cycles in $\prec$ are possible and are handled dynamically at method lookup [3].

In order to get a connection to families of core structures defined before, we introduce additional tameness conditions. The box on the right says that "tame Self core structure" is synonymous to "finite partial order with a top". There is just one definitory constituent: $\leq$. By definitional extension, we let

- $\in$ be identical to $\leq$,
- *.class* be identity on O.

Note that there is a one-to-one correspondence between tame Self core structures and Perl core structures without terminal objects.

> A *tame Self core structure* is a structure $(O, \leq)$ where
>
> - O is a set of *objects*,
> - $\leq$ the *inheritance* relation between objects,
>
> such that the following are satisfied:
>
> (ts~1) $(O, \leq)$ is a partial order.
>
> (ts~2) There is a top object, r.
>
> (ts~3) There are only finitely many objects.

Metaclasses in Self

Viewing tame Self core structures as a special case of Perl core structures, we can conclude that there are no metaclasses in Self. That is, in Self, everything is a class but nothing is a metaclass. Unfortunately, such a conclusion does not arise by the resistant definition of metaclasses. The definition, although being resistant enough to cope with the degeneracy of Perl, is not resistant enough to cope with the (yet greater) degeneracy of Self.

The Io classification

A refinement of tamed Self core structures is obtained by distinguishing a closure system w.r.t. inheritance. The corresponding closure operator, *.c*, can be regarded as a classification map. Since *.c* is idempotent, the classification hierarchy has just 2 levels (cf. idempotency of `.WHAT` in Perl 6). Such a distinction is established in the Io programming language [45], where objects with a slot named `type` are considered to be "types". Objects obtain this slot upon assignments to capitalized slots, as shown by the code on the right. By this semantics, the "type" of a "type" x equals x. In general, the "type" of an object x is the least ancestor of x that is a "type". (The x object in the example becomes a "type" after the `x A := x` assignment.)

```
x := Object clone
x type println  # Object
x y := x
x type println  # Object
x A := x
x type println  # A
x B := x
x type println  # A
```

The $(\leq) = (\in)$ semantics is suggested by the name of the correspondent introspection method: `isKindOf`.

Prototypes in JavaScript

The JavaScript programming language [56] supports the *(∈) = (≤)* equality as far as the semantics of method resolution is concerned. Every object responds to its own methods. However, there is a limited native support for the perception of another core structure, where there is an *instance-of* relation detectable by the `instanceof` operator. Such a structure is established by 3 (partial) maps between objects, *.sc*, *.class*, and *.ce*, given by object properties named `__proto__`, `constructor` and `prototype`, respectively. Simultaneously, there are three basic kinds of objects: ordinary objects (terminals), classes and prototypes. 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 strict inheritance < which arises as the reflexive closure of *.sc* (i.e. by following the `.__proto__` links) can be detected via the `isPrototypeOf` method. A one-to-one correspondence between classes and prototypes is established by the `constructor` and `prototype` properties. The `constructor` property is owned by prototypes and inherited by other objects. In contrast, `__proto__` and `prototype` are never (strictly) inherited. The `__proto__` property has been standardized in ECMAScript 6 [53]. The `instanceof` operator is given by

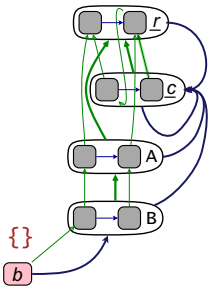
x `instanceof` *y* ↔ *y*`.prototype.isPrototypeOf(`*x*`)`

so that *y*`.prototype` is not reported as an instance of *y*.

The diagram on the right shows a sample core structure for ECMAScript 6. Thin green arrows indicate the `__proto__` links (*.sc*) and the horizontal thin blue arrows stand for the `constructor` links (*.class*) between prototypes (*O.ce*) and classes (*O.class*). The restriction of *.class* and ≤ to *O* \ *O.ce* (objects without prototypes) is indicated by thick arrows. Simultaneously, the diagram suggests viewing each prototype-class pair as a single object. The `__proto__` link between *c* and *r* is established artificially. (By default, the parent of *c* equals *c.ce*.)

(ECMAScript 6)

```
var
r = Object,
c = Function, b
c.__proto__ = r
class A {}
class B extends A {}
b = new B
```



Observe that the "coarse" structure of thick links is a Smalltalk-76 core structure. Moreover, the diagram is very similar to that of *x*–*x.ec* pairs considered for Ruby core structures.

In the following subsection, we let ES6 core structures be exactly the prototype completions of Smalltalk-76 core structures. Such structures can be recognized in ECMAScript 6 under several circumstances. In particular:

- Classes are: `Object` (as *r*), `Function` (as *c*) and objects created by subclassing using the `class` keyword.
- Terminal objects are those created by instantiation of classes other than `Function`.
- Prototypes are objects referenced by *x*`.prototype` from classes *x*.
- The `__proto__` link from `Function` to `Object` is established (e.g. by `Function.__proto__ = Object`).
- Only classes other than `Function` can be subclassed.
- Neither of `__proto__`, `prototype` or `constructor` links are explicitly manipulated.

ES6 core structure

Let an *ES6 core structure* be a structure $\mathcal{S} = (O, .class, \leq, r, .ce)$ where

- *O* is a set of *objects*,
- *.class* is the *class* map $O \to O$,
- ≤ is the *inheritance* relation,
- *r* is a distinguished object,
- *.ce* is the *prototype* partial map $O \curvearrowright O$.

(es6~1) *(O* \ *O.ce, .class, ≤, r)* is a Smalltalk-76 core structure.
(es6~2) *(≤)* ⨾ *(.class) = (≤)* ⨾ *(.ec)*.

Objects from *O.ce* are called *prototypes*, objects from *O.class* are *classes*. The structure is subject to the axioms in the box on the right, with *.ec* denoting the inverse of *.ce*.

Observations:

1. Prototypes are defined exactly for classes. Classes are exactly the descendants of *r*.
2. Every object is a descendant of *r.ce*.
3. The *.ce* map establishes an order-isomorphism between classes and prototypes.

4. The $.ec$ map establishes powerclass links between prototypes and classes.
5. (Prototype completion.) Up to isomorphism, $\mathcal{S}$ is uniquely given by $\mathcal{S}_0 = (O \setminus O.ce, .class, \leq, r)$. To construct $\mathcal{S}$ from $\mathcal{S}_0$, define $a.ce$ for each class a to be a new object, extend $.class$ by $a.ce.class = a$, and extend $\leq$ by

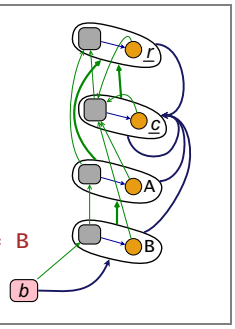
$$\begin{aligned} a.ce \leq b.ce &\leftrightarrow a \leq b, \\ a \leq b.ce &\leftrightarrow a.class \leq b, \\ a.ce \leq b &\leftrightarrow \text{false}. \end{aligned}$$

ES5 core structure

The box on the right shows vanilla subclassing in ECMAScript 5 (and also in ECMAScript 3). In contrast to ES6, there are no parallel hierarchies of classes and prototypes. Instead, classes can be regarded as singletons, as indicated by the diagram. This makes ES5 core structures similar to core structures of the Newspeak programming language.

(ECMAScript 5)

```
var
r = Object,
c = Function,
A = new c,
B = new c
B.prototype = new A
B.prototype.constructor = B
var
b = new B
```



What is a metaclass?

In the $(\in) = (\leq)$ semantics, there are no metaclasses. However, in the `instanceof` operator semantics, the `Function` object is a metaclass. This can be justified by the following identity:

```
(new new Function).constructor.constructor === Function.
```

By disallowing the creation of descendants of `Function` and assuming a single global object, the `Function` object is the only metaclass in JavaScript (both in ES5 and ES6).

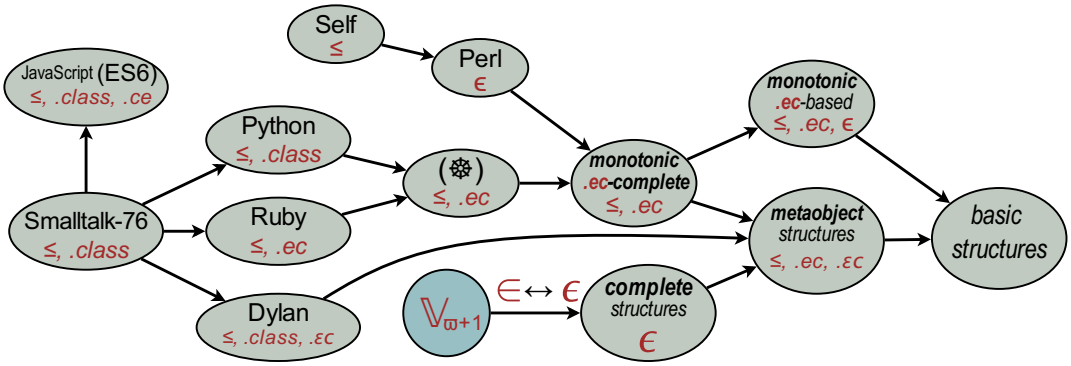
The general core

We have already defined monotonic powerclass structures as the common "essential" family for Python and Ruby. This family (or the family of monotonic $.ec$-based structures) constitutes common generalization for core structures that are subject to monotonicity of $\in$:

$(\leq) \circ (\in) \subseteq (\in)$.

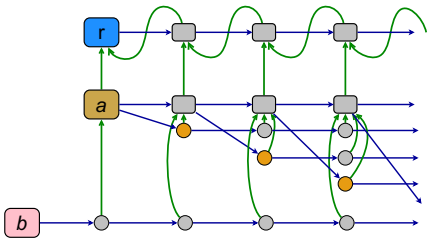
In this section we provide the ultimate general core in which monotonicity of $\in$ is abandoned. The clue constituent (that was not present in monotonic structures) is the singleton map $.\varepsilon c$ introduced by Dylan. As a result, the interpretation of blue and green links in terms of set theory is established.

The diagram below shows that the Python-Ruby-Dylan triad leads to the family of *metaobject structures* with $\leq$, $.ec$ and $.\varepsilon c$ as definitory constituents. In analogy to the corrected Dylan core structures, $x.\varepsilon c$ is defined exactly for *bounded* objects where boundedness is a generalization of non-circularity. This gives rise to the *bounded membership* relation, ϵ, as the composition of $.\varepsilon c$ with $\leq$. This relation is then in a direct correspondence to set membership, $\in$, between well-founded sets in some partial universe of sets.



An arrow from node A to node B indicates that the B family arises from the A family by generalization and/or completion (meaning an *.ec* or *.εc* or *.ce* -completion).

Metaobject structure



The diagram on the right shows the first few metalevels of a *metaobject structure*. The structure is constituted by the powerclass map *.ec* (shown by horizontal blue arrows), the singleton map *.εc* (shown by blue arrows pointing to a circle which indicates a singleton) and the inheritance relation ≤ (shown by green links for the ≺ reduction). Observe that there is a partial coincidence of *.ec* and *.εc* :

$x.ec = x.\varepsilon c \quad \leftrightarrow \quad x$ is terminal or a singleton.

There are several membership relations derived from the definitory constituents:

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

(mo~6) The singleton map, *.εc*, is injective.
(mo~7) Objects from $O.\varepsilon c.ec^{*}$ are minimal in ≤.
(mo~8) For every objects *x*, *y* such that *x.εc* is defined,
$x.\varepsilon c \leq y.ec \quad \leftrightarrow \quad x \leq y$.
(mo~9) For every object *x*, *x.εc* is defined $\quad \leftrightarrow \quad x.d < \varpi$.

Formally, a *metaobject structure* is a structure $(O, \leq, r, .ec, .\varepsilon c)$ such that the reduct $(O, \leq, r, .ec)$ is a monotonic powerclass structure (thus (e~1)–(e~5) apply),

- *.εc* is a partial map between objects,

and the additional axioms are satisfied as shown in the box on the right.

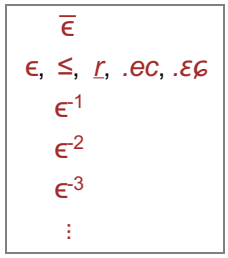
In the last axiom, ϖ is a fixed limit ordinal and *.d* is the *rank* function $O \to \varpi+1$ defined recursively as follows:

$x.d = \varpi$ if *x* is non-well-founded in ϵ,
$x.d = \varpi \wedge (\sup \{a.d + 1 \mid a \epsilon x\} \vee \sup \{a.mli + i\text{-}j \mid a \in x.\ni^{i}.\ni^{\text{-}j}, i,j \in \mathbb{N}\})$ 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. The expression *a.mli* refers to the metalevel index of *a* which is defined the same way as for monotonic *.ec*-based structures. Finiteness of *a.mli* is asserted by (e~5).

Objects are either *unbounded* or *bounded* according to whether their rank is maximal or not, respectively. By (mo~9), the set of bounded objects equals $O.\ni$ – i.e. an object is bounded iff it appears on the left side of ϵ. (This way the *bounded* vs. *unbounded* terminology becomes compatible with that used in the Knowledge Interchange Format (KIF)[64].) By definition of *.d*, all non-well founded objects are unbounded (but not necessarily vice versa). As a fundamental consequence, ϵ is a well-founded relation. See [138] for details.

Basic structure



Basic structures form the most general family. They arise from metaobject structures by allowing $.ec$ and $.\varepsilon c$ to be incomplete, possibly empty. This is compensated for by using ϵ, $\overline{\epsilon}$ and ϵ^{-k} for every natural k as additional definitory constituents. The whole signature is shown on the right. The last constituent, $.\varepsilon\not c$, stands for the difference $(.\varepsilon c) \setminus (.ec)$.

Except for the ES6 core structures ($*$), every family of structures depicted in the diagram above is definitionally equivalent to a subfamily of basic structures. This also applies to the "actual" Ruby core structures $(O, \preceq, .aclass, .ec)$ where O are the actual objects – the finite part of the infinite regress that is actually evaluated.

The axiomatization of the family of basic structures is rather complicated, see [138] for details.

Note: ($*$) ES6 core structures can be made a subfamily by "shifting" them one metalevel higher.

Complete structure of ϵ

Complete structures of ϵ form a subfamily of metaobject structures. In addition to the completeness of $.ec$ and $.\varepsilon c$ it is asserted that there is a one-to-one correspondence between objects and non-empty subsets of the set $O.\ni$ of bounded objects, and that inclusion between these subsets corresponds with inheritance:

(A) *Extensional consistency*: For every objects x, y, $x \preceq y \;\leftrightarrow\; x = y$ or $\varnothing \neq x.\ni \subseteq y.\ni$.

(B) *Extensional completeness*: For every subset X of $O.\ni$ there is an object x such that $x.\ni = X$.

Together with an additional condition for the ϵ-rank of objects it is established that a complete structure is definitionally equivalent to an $(\varpi{+}1)$-*superstructure* cumulatively built from the ground stage of terminal objects (the *urelements*). In particular, a complete structure of ϵ is fully determined by ϵ. Every basic structure can be faithfully embedded into a complete structure, with $.ec$-completion and $.\varepsilon c$-completion being two distinguished steps in the gradual embedding process. See [135] [138] for details.

Embedding into the von Neumann universe

Let $\mathbb{V}$ denote the von Neumann universe of all well-founded sets and for an ordinal α, let $\mathbb{V}_\alpha$ be the α-th stage, also called the partial von Neumann universe of rank α. Then $(\mathbb{V}_{\varpi+1}, \in)$ is an $(\varpi{+}1)$-*superstructure* and thus a complete structure of ϵ (having $\varnothing$ as the only terminal object). Every basic structure can be faithfully embedded into $\mathbb{V}$, with the following main correspondences:

$$\epsilon \leftrightarrow \in, \quad \preceq \leftrightarrow \subseteq, \quad x.ec \leftrightarrow r \cap \mathbb{P}(x), \quad x.\varepsilon c \leftrightarrow \{x\}.$$

The embedding can be chosen is such a way that all terminal objects are represented by singleton sets of the same rank. (Note that for $\mathbb{V}_{\varpi+1}$, the identity map does not constitute a faithful embedding since the $\preceq \leftrightarrow \subseteq$ correspondence fails due to $\varnothing \subseteq r$. This is ruled out by terminal objects being mapped to singleton sets.) See [135] [138] for details.

What is a metaclass?

The definition introduced for the monotonic case applies to the general case. That is, up to a possible "duplicate" of $r.ec$, **metaclasses are the objects from metalevel 2 or higher.** Since every basic structure $\mathcal{S} = (O, \ldots)$ can be embedded into a complete structure which is definitionally equivalent to an $(\varpi{+}1)$-superstructure $\mathcal{V} = (V, \epsilon)$, we can regard $\mathcal{V}$ as an ultimate extension of $\mathcal{S}$ in which all objects have all their "potential" members. As a consequence, we can remove the adjective "potential" from the resistant definition of metaclasses and say that

- metaclasses are the non-terminal objects all of whose members are non-terminal,

where "members" refers to object membership ϵ in $\mathcal{V}$. Moreover, "members" can also refer to bounded membership, ϵ, since $x.\ni \preceq r \leftrightarrow x.\ni \preceq r$.

Further proximity to the classic definition of metaclasses can be achieved if we adopt the terminology from Morse–Kelley set theory with urelements (atoms) as presented in [156]. In this system, which we refer to by MKU, "atoms" are the terminal objects and "classes" are the non-terminal ones. As a result,

- metaclasses are classes all of whose members are classes.

A correspondence between MKU and $\mathcal{V}$ is established for ϖ equal to a strongly inaccessible cardinal. In this case the set V can be seen as the universe of MKU devoid of the empty set. (That is, $\varnothing$ is removed from the universe as well as all classes that contain $\varnothing$ in their transitive closure. As a result, all classes are "purely non-pure".)

"all" versus "some"

An exact semantical comparison can be drawn for the two possible quantifiers for "instances" in the classic definition of metaclasses. Metaclasses are … "all of whose instances are …" or "some of whose instances are …"?

Let x be a non-terminal object in a complete structure of $\in$. Then

all members of x are non-terminal $\leftrightarrow$ $x.mli \geq 2$,
some members of x are non-terminal $\leftrightarrow$ $x.d \geq 2$.

That is, "all" refers to the metalevel index whereas "some" refers to the rank. We support the former case.

Metaclasses are taboo

We have shown that the notion of *metaclass* appears in several significant object-oriented programming languages such as Python, Ruby, Smalltalk, Objective-C or CLOS. These languages are adherent representants of object-orientedness, with a strong support for the "everything is an object" paradigm. As a consequence, the languages can be considered to form an important (or even central) part of the field of object-oriented programming. We have also shown that the metaclass term is tightly connected to the two most fundamental relations between objects – the blue and green links that form the core structure $(O, \in, \leq)$. We have provided a precise description for many families of core structures and even established a connection to set theory, a foundational system for mathematics.

Given this, it seems natural to assume that the notion of metaclass is an established constituent of foundations of object-oriented programming. However, it turns out to be a false assumption. As of 2015, if you ask a computer scientist about literature on mathematical modelling of object-oriented programming languages, you typically obtain the following references [$\mathbb{S}_1$] [$\mathbb{S}_2$]:

- ***A Theory of Objects*** by Martin Abadi and Luca Cardelli [1],
- ***Foundations of Object-Oriented Programming Languages: Types and Semantics*** by Kim B. Bruce [25],
- ***Types and programming languages*** by Benjamin C. Pierce [149],
- ***Object-Oriented Programming A Unified Foundation*** by Giuseppe Castagna [32],
- ***Practical foundations for programming languages*** by Robert Harper [76],
- ***Featherweight Java*** by Atsushi Igarashi, Benjamin C. Pierce and Philip Wadler [83].

Note that the first two books, AToO and FoOOPL, have already been discussed in the introductory sections. Subsequently, you can use Google Book Search [1b] [25b] [149a] [32a] [76a] or a PDF reader [83a] to search for the occurrences of "metaclass" or "meta-class" (or the plurals thereof) in these books or papers. In total, you get the following number of results:

0. (There are no metaclasses.)

Alternatively, you can try Google Scholar to search for the *metaclass* term in all articles published by any author of the above documents. The search for articles published up to 2011 [$\mathbb{\Gamma}_3$] returns no results. (This holds if just the single term "metaclass" is searched for. Allowing the hyphenated form and the plurals yields 3 papers by Luca Cardelli, each just mentioning the existence of metaclasses in Smalltalk [$\mathbb{\Gamma}_4$].) A search with the date limit increased to 2014 [$\mathbb{\Gamma}_5$] yields exactly one article [15] which is co-authored by K.B. Bruce. In this article, having no metaclasses – a consequence of classes not being objects – is considered to be the strength of the Java design.

Two viewpoints of foundations of OOP

The reason behind these results lies in the different viewpoint that is taken by the majority of OOP research. As

of 2015, the prevailing approach to foundations of object-oriented programming is to invent an appropriate counterpart to foundations of functional programming. With perhaps a slight exaggeration, the approach can be formulated as a belief that OOP foundations arise by a suitable adjustment of typed lambda calculus. All the above books espouse this viewpoint.

Let us recall that we presented 3 bisection and 1 trisection criteria for the delimitation of our metaclass approach. The main criterium is given by deciding which part of the term "object-oriented programming" is thought to be more essential. The *calculus* approach emphasizes "programming" over "object-oriented". We have provided the following characterization of the bisection:

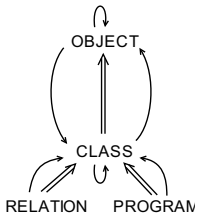
Calculus approach:	Focus on code / computation.
Non-calculus approach:	Focus on data / state.

This single criterium is already sufficient to distinguish the present document from the above listed references. Formal models described in this document do not use the notion of a calculus. The opposite is the case of all the above literature.

The calculus approach takes the view that programming (including OOP) is about writing programs. Programs are just code expressed in a programming language. It is therefore necessary (or at least natural) to focus on the code in the first place. Given this, the task is to invent an "essential object-oriented language" – a simple formal language which would capture the essence of OOP. Following the success of lambda calculus as a foundation for functional programming, it is believed that a similar formal system is appropriate for object-oriented programming.

Metaclasses in knowledge representation

We have established mathematically precise model for the fundamental relations of instantiation ($\in$) and inheritance ($\leq$) as they appear in many programming languages. However, the $\in$ and $\leq$ relations go beyond OOP – they can be observed virtually in any object-oriented data model. Similarly, the metaclass question is relevant also in other contexts than just OO programming languages.



According to Google Scholar [$\mathbb{\Gamma}_7$], the inception of a data model with metaclasses (and thus the introduction of the "metaclass" term into information science) took place in late 1970s in the field of knowledge representation. The invention can be attributed to Hector Levesque and John Mylopoulos. In their 1979 article titled *A procedural semantics for semantic networks* [102], the authors presented a system that contained all the constituents of what we call the core structure: objects, classes and metaclasses together with the $\in$ and $\leq$ relations. (The $\in$ relation is called "*instance hierarchy*" or *instance-of* in a predicate. The $<$ relation is termed, sadly, "*IS-A hierarchy*" or *subclass-of* in a predicate.) The diagram on the right is a direct transcription of article's Fig.17 which shows that the Python's built-in circular structure appears to be part of the system. Here the single-line arrows stand for $\in$ and the double-line arrows indicate direct inheritance, $\prec$. Similarly to Smalltalk-76 or the F&D model, there are two circular classes: $\mathfrak{r}$ (named `OBJECT`) and $\mathfrak{c}$ (named `CLASS`). Classes are exactly the instances of $\mathfrak{c}$ and simultaneously the descendants $\mathfrak{r}$. Metaclasses are introduced as "*classes whose instances are also classes*". The built-in `CLASS` class is the top metaclass, so that "*every metaclass is a subclass of `CLASS`*". As can be observed on the diagram, there are other built-in metaclasses (but they are not circular).

Today's most popular knowledge representation systems are based on the ontology languages designed for the Semantic Web. There are two distinguished ones in which the metaclass pre-condition is satisfied: RDF Schema and OWL Full. The documents [137] and [135] provide a detailed description of what can be considered the core structure of these languages (the "ontological structure of $\in$"). The structures are different from (most of) the hitherto described families of core structures of OOP in the following features:

- Distinction of *properties* as instances of a special class, $\mathfrak{p}$. 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.
- Absence of the monotonicity condition $(\leq) \circ (\in) \subseteq (\in)$.

Note that the latter two features can still be established by specialization of the general core.

The core of RDF Schema

In RDF Schema (RDFS) [183] [184] objects are called *resources* – every object is an instance of the `rdfs:Resource` class (r). Classes are instances of `rdfs:Class` (c), properties are instances of `rdf:Property` (p). Unlike classes, properties do not have a common built-in ancestor.

Formally, we define an *RDFS core structure* as a structure $(O, \in, \leq_C, \leq_P, r, c, p)$ where O is the set of *objects* or *resources*, $\in$, $\leq_C$, and $\leq_P$ are the *instance-of*, *subclass-of* and *subproperty-of* relations on O, respectively, and r, c, and p are distinguished objects. Instances of c form the set C of *classes*, instances of p form the set P of *properties*. The structure is subject to the conditions in the box on the right.

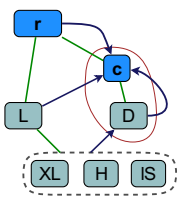
(r~1) $\leq_C$ is a preorder on C. (rdfs10)&(rdfs11)
(r~2) $\leq_P$ is a preorder on P. (rdfs5)&(rdfs6)
(r~3) $(\in) \circ (\leq_C) \subseteq (\in)$. (rdfs9)
(r~4) Only classes can have instances.
(r~5) Every object is an instance of r.
(r~6) Every class is a subclass of r. (rdfs8)
(r~7) r, c and p are pairwise distinct.
(r~8) $C \setminus P$ is finite.
(r~9) p is a class.

RDF Schema ensures that all of (r~∗) are satisfied. Some conditions have direct correspondence to *entailment rules* (referenced in parenthesis). On the other hand, the above axiomatization only defines a coarsement of RDFS. The *built-in* RDFS structure (if reduced to the core according to our axiomatization) is subject to axioms of a Python core structure provided that inheritance, $\leq$, is identified with $\leq_C$, and only finitely many properties `rdf:_1`, `rdf:_2`, … are allowed. Moreover, the built-in structure even possesses single inheritance. See [137] and [135] for details.

What is a metaclass?

By the (partial) correspondence to Python core structures, we define the RDFS metaclasses as the (non-strict) descendants of c (`rdfs:Class`). Indeed, the subsumption rule $(\in) \circ (\leq) \subseteq (\in)$ (rdfs9) asserts that all potential instances of subclasses of c are guaranteed to be classes, so that these subclasses meet the resistant definition of metaclasses. On the other hand, every RDFS class not from $c.\mathrm{I}$ can potentially have a direct instance that is not a class. However, since the monotonicity condition $(\leq) \circ (\in) \subseteq (\in)$ is not asserted in RDF Schema, it is also not ensured that classes of classes ($O.class(2)$) are descendants of c. The "all"/"some" quantification makes a difference – there can be classes that have both classes and non-classes (terminal objects) as direct instances. This establishes conditions for the metaclass misnomer.

Fortunately, research papers can be found in which the misnomer is avoided. In [104], metaclasses are delimited in accordance to our definition. The authors make a claim that classes of classes should not be considered metaclasses unless they are descendants of `rdfs:Class`.



Similarly, S. Koide and H. Takeda require that "*all metaclasses must hold `rdfs:Class` as superclass*" [95]. In [96], the class `UnitOfMeasure` from the SUMO ontology is given as an example of an "*ill-structured metaclass*" that should be "remedied" by making it a subclass of c.

The diagram on the left shows the restriction of the built-in RDFS core structure to $(c.\mathrm{I} \setminus \{c\}).\ni^*.\mathrm{I}$ according to RDF 1.1 ($\ni^*$ is the reflexive transitive closure of $\ni$). In addition to c, there is just one another built-in metaclass, named `rdfs:Datatype` (labelled `D`).

The OWL 2 core

OWL 2 Full [182] is an ontology language defined as an extension of RDF Schema. The built-in structure contains additional 58 classes and 62 properties [137a]. The two circular classes that are adopted from RDF

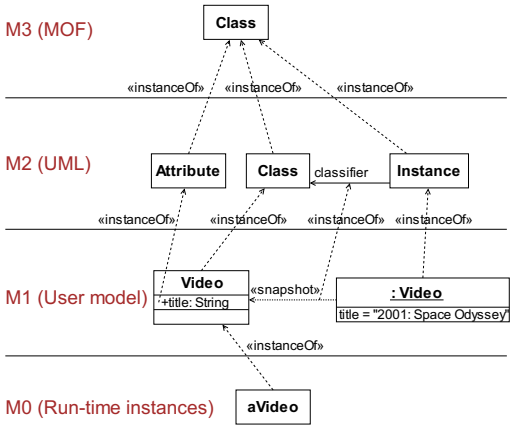
Schema, rdfs:Resource and rdfs:Class, are doubled by owl:Thing and owl:Class as their respective OWL equivalents. Similarly, owl:ObjectProperty is an equivalent of rdf:Property, and (the metaclass) owl:DataRange is equvivalent to rdfs:Datatype. There is still single inheritance between non-equivalent 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 RDFS objects (resources) 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 (labelled L in the diagram from the previous subsection) is made an instance of rdfs:Datatype.

owl:Nothing

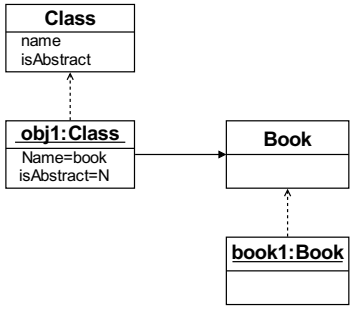
There are exactly 7 built-in OWL classes that are (non-strict) inheritance descendants of rdfs:Class. However, one of them, owl:Nothing, is special since it is asserted to (a) be a descendant of all classes and (b) to have no instances. That is, owl:Nothing is meant to be an abstraction of the empty set. But we did not include such a bottom object in our set-theoretic model (see Complete structure of $\in$ and [138] for details). Consequently, owl:Nothing goes beyond our universe (of objects) and as such it is not a metaclass.

Metamodelling kernel



One of the most difficult contexts for the recognition of the $\in$ and $\leq$ relations is the field of metamodelling. Traditionally, a *layered architecture* is presumed dividing the system entities into layers (levels) according to "metaness". The Object Management Group (OMG) standard [124] defines four layers: M0 is the *data layer*, M1 is the *model*, M2 is the *meta-model* and M3 is the *meta-meta-model*. The top layer M3 is coincident with the Meta-Object Facility (MOF), whereas the M2 layer includes the Unified Modeling Language (UML). There is an *instance-of* relationship, denoted «instanceOf», which relates elements of consecutive layers according to the **strict metamodelling paradigm**: elements in any of the layers must be instances of elements in the layer immediately above, bar the top one. Moreover, «instanceOf» is functional so that it provides a layer-increasing mapping ($*$). The diagram on the right is a verbatim transcription of an example from [124] (also provided by [116]). In the opposition to instantiation, the inheritance relation, $\leq$, which is usually called *specialization* (with *generalization* being the term for the inverse), can only relate elements from the same layer. *Note:* ($*$) The OMG standard [124] states right-uniqueness of «instanceOf» explicitly for the relationship between UML and MOF elements, see also [116].

Clabject split



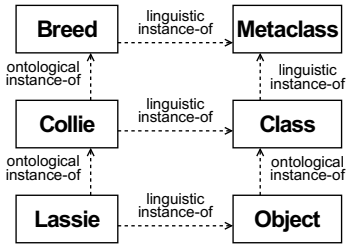
Until recently, composition of the instance-of relation was considered illegal by some prominent authors, most notably Gonzales-Peres and Henderson-Sellers [𝕎68]. These authors made a presupposition of the metaclass anti-condition – classes are not objects. In [161] a conclusion was drawn that the metaclass concept "*is self-contradictory and should therefore [be] discarded*". In a subsequent paper [69] the authors proposed what can be called a "clabject split". Here "clabject" is a portmanteau of "class" and "object" that refers to elements from the intermediate layers – i.e. elements that do not reside on either of the bottom (M0) or top (M3) layers. Every clabject should be treated as two distinct entities ("de-clabjectized" into the object facet and class facet) according to the diagram on the right (transcribed from the book [162]). The links between the facets form an "isotypical interpretive mapping". This is in order to support the "*conventional object-oriented paradigm*". Although not stated explicitly in either of [69] or [162] we can hypothesize analogy with Java class

reification.

Linguistic vs. ontological instantiation

Other authors regard the instance-of relation given by the «instanceOf» links in the layered architecture as too coarse. In [9] a proposal is made to distinguish between *linguistic* and *ontological* instantiation. The linguistic instance-of is roughly correspondent to the inter-level «instanceOf» as specified by OMG while the ontological instance-of is an orthogonal intra-level relation. The diagram on the right is adopted from [162].

Another intra-level instantiation is proposed in [14].



The *power-type* misnomer

In addition to problems with $\in$ and $\leq$, the metamodelling context also establishes obstacles to the recognition of the powerclass map, *.ec*. In an analogy to the *is-a* misnomer one can speak about a "power-type misnomer". The notion of *power type* has been introduced ($*$) to the field of metamodelling in a 1994 article by J. Odell [123]. Quoting from the paper, " *A power type is an object type whose instances are subtypes of another object type.* ". A closer inspection of the paper makes us hypothetize that "power type of" is meant to be an abstraction of "non-trivial set-partition of". That is, if x and y are "object types" that are abstractions of sets X and Y, respectively, then by "y being a power type of x" is meant that elements of Y are non-empty mutually disjoint proper subsets of X whose union is X. In other words, Y is a non-trivial partition of X. Particular consequences:

(i) An object type can have more than one power type.

(ii) An object type is never an instance of its power type, only (some of) its *strict* subtypes are.

The notion has subsequently been adopted to UML. The OMG standard [125] provides a description similar to that of the original paper, but, in addition, makes use of the *metaclass* term: "*Power types […] are metaclasses with an extra twist: the instances can also be subclasses*". Note that this poses a difficulty for the differentiation between "power type" and "metaclass" since e.g. in the F&D model and Python, the predicate of "being a subclass" is implicitly satisfied for every class. (Every class is a subclass of itself and also a subclass of the inheritance root r.)

Several (probably most) metamodelling authors [100] [119] [31] regard the Odell's concept of a power type to be an adoption of the synonymous notion introduced by L. Cardelli in type theoretic setting [29]. However, the Cardelli's power types are abstraction of powersets and are thus roughly correspondent to powerclasses (see [142] for details). In this interpretation, neither of the above conditions (i) and (ii) is satisfied. Recent publications in metamodelling [31] [162] [163] therefore consider the Odell's interpretation of "power type" to be a misnomer. The OMG standard [125] is ambiguous in this respect. It mostly follows the Odell's interpretation but also suggests a correspondence between "power type" and "power set": "*The notion of power type was inspired by the notion of power set*" [$\mathbb{W}_{19}$].

Notes:

1. ($*$) The Odell's (mis)interpretation of Cardelli's power types can be thought of as having at least one intermediate step. A similar interpretation of the same notion already appeared in the 1993 article titled *Expressiveness in Conceptual Data Modelling* [79]. Here the same formulation for "object types" x and y is presumably applicable as shown above except that mutual disjointess is not requested.
2. If "power type of" is meant as the abstraction of "non-empty subset of the powerset of" then "power type of" is the same as "anti-member of". See the definition of the anti-membership relation $\in^{-1}$.

Back to ObjVLisp uniformity

In 2013, Henderson-Sellers and his colleagues came to a conclusion that the (core part of) metamodelling has become incoherent [163]. They suspect that insisting on the strict metamodelling paradigm constitutes a "Ptolemaic" approach which makes metamodelling foundations unnecessary complex. A "*Copernican-style revolution*" is needed. In the same year the authors presented a "*new modelling approach in which 'everything is an object'*" [164]. In particular, the metaclass anti-condition that was presupposed in the 2005 paper [161] has been upturned to the metaclass pre-condition: classes are objects. This "Copernican" shift leads in substance

to the ObjVLisp model. There is a "commencing entity", an object called `Object`. Since any object is of a specific class, the class of `Object` is introduced, named `Class`, which is simultaneously a subclass of `Object`. Metaclasses are no longer "self-contradictory" as in [161]. On the contrary, they "*are the basis for language definition*". Moreover, they are defined as "*those distinguished classes that directly or indirectly inherit from the class* `Class`" – which is exactly the definition established in Python core structures.

In the subsequent paper titled *A Foundation for Multi-Level Modelling* [37] the authors describe a minimalistic metamodelling language called *Kernel* which provides an implementation for the proposal.

Metaclasses in VODAK

As of 2015, there are exactly two books which contain the word *metaclass(es)* in their title. In addition to the F&D book [59] there is a book by W. Klas and M. Schrefl titled *Metaclasses and Their Application: Data Model Tailoring and Database Integration* [93]. The book provides a description of the object-oriented data model of a database management system called *VODAK*. As the title suggests, metaclasses are regarded as an essential model constituent.

The next subsection provides a distillation of the core part of VODAK's data model. In contrast to Python core structures, there are *two* sorts of entities: *objects* and *types*. Objects are involved in the instantiation graph (which is just the *.class* map), whereas types are involved in the inheritance graph. Metaclasses and classes form subsets of objects and are delimited according to their instantiation depth. There are three partial mappings from objects to types. These mappings when combined with the *.class* map establish the provision of methods (which are owned by types).

VODAK core structure

For a fixed natural number $d \geq 2$ let a *VODAK core structure* of *instantiation depth* d be a two-sorted structure $(O, \mathbb{T}, .class, \leq, m, .types)$ where

- O is a set of *objects*,
- $\mathbb{T}$ is a set of *types* (disjoint from O),
- *.class* is the *class* map $O \to O$,
- $\leq$ is the *inheritance* relation between types,
- m is the *instantiation root*, a distinguished object,
- *.types* is a map $O \to \mathbb{T}^*$ (from objects to finite lists of types).

(vo~1) $(\mathbb{T}, \leq)$ is a partial order.
(vo~2) $\{m\}$ is the only cycle of *.class*.
(vo~3) For every object x, $x.class(d) = m$.
(vo~4) For every object x, $x.types.length \leq x.mli+1$.
(vo~5) $O \cup \mathbb{T}$ is a finite set.

The structure is subject to the axioms in the box on the right. In (vo~4), *x.mli* denotes the *metalevel index* of x which is defined as the greatest natural i such that $i \leq d$ and $x.class(d\text{-}i) = m$. (We use the standard notation for powers of *.class*: for a natural $i > 0$, *.class(i)* is the i-th composition of *.class* with itself, *.class(0)* is the identity on O and *.class(-i)* is the inverse of *.class(i)*.)

Note that there is no inheritance between objects, only between types. The *instance-of* relation between objects is coincident with the *.class* map. Condition (vo~3) asserts that the depth of the instantiation tree is at most d. For an object x,

- x is (a) *terminal* $\leftrightarrow$ $x.class(d\text{-}1) \neq m$ $\leftrightarrow$ $x.mli = 0$,
- x is a *class* $\leftrightarrow$ $x.class(d\text{-}1) = m$ $\leftrightarrow$ $x.mli \geq 1$,
- x is a ***metaclass*** $\leftrightarrow$ $x.class(d\text{-}2) = m$ $\leftrightarrow$ $x.mli \geq 2$.

The instantiation root m is the only object from the d-th metalevel (the highest metalevel).

In VODAK, $d = 4$ and the instantiation root m is named `VML-CLASS`. There is a distinguished object on the metalevel 3, named `METACLASS`, that is supposed to be the "user instantiation root". (∗) Moreover, the length of *x.types* is further restricted to at most three. The following terminology is used:

- *x.types[0]* is the *own-type* of x,
- *x.types[1]*, if defined, is the *instance-type* for instances of x,
- *x.types[2]*, if defined, is the *instance-instance-type* for instances of instances of x.

Further restrictions to *.types* are imposed, being based on the built-in hierarchy of types (cf. [92a]). For example, instance-types of user-defined metaclasses are supposed to be descendants of

`Metaclass_InstType`.

Note: (∗) In [93] [92] the metalevel numbering is 1-based so that `METACLASS` appers on "Level 4".

Method resolution

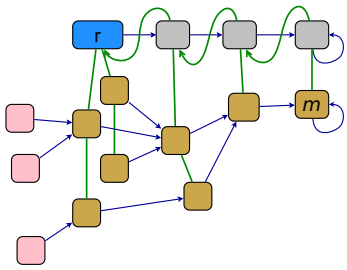
In VODAK, methods are provided by types. For an object x, the set of all types that are providers of methods to which x responds is the union of all the sets

$$x.class(i).types[i].\updownarrow$$

where i is a natural number such that $x.class(i).types[i]$ is defined. Moreover, for $i < j$ the set $x.class(i).types[i].\updownarrow$ takes precedence over $x.class(j).types[j].\updownarrow$. In particular, the own-type of x takes precedence over the instance-type of $x.class$ which in turn takes precedence over the instance-instance-type of $x.class.class$.

Strict metalevelling

A common subfamily of VODAK core structures and basic structures of ϵ can be obtained by equipping each metalevel with a top as follows. Let d be a fixed natural number, $d \geq 2$, and let $\mathcal{S}$ be a single-sorted structure $(O, \leq, .class, .ec, r, m)$ where O is the set of objects, $\leq$ is a relation, $.class$ is a total map, $.ec$ is a partial map, and r and m are distinguished objects. The structure is subject to conditions below. In the conditions, R denotes the set $r.ec^*$, and for every object x, $x.mli$ is a natural number equal to (a) i where i is greatest such that $i \leq d$ and $x.class(d\text{-}i) = m$ if such i exists, or (b) $i+1$ where i is the smallest natural number such that $r.ec(i) = x$.



1. $(O, \leq)$ is a finite partial order.
2. $R.\updownarrow = R$.
3. $R.class = \{r\}$.
4. $(O \setminus R).ec = \emptyset$.
5. $(O \setminus R).class(d+i) = \{m\}$ for every natural i.
6. For every $x \in O$ and every natural i, $\quad x \leq r.ec(i) \leftrightarrow x.mli > i$.
7. For every $x,y \in O \setminus R$, $\quad x < y \rightarrow x.mli = y.mli$.

The last condition asserts that $\leq$ is "intra-level" on $O \setminus R$. Object membership is defined by $(\epsilon) = ((.class) \cup (.ec) \cup \{r.ec(d\text{-}1)\}^2) \circledcirc (\leq)$.

Apparently, $\mathcal{S}$ is uniquely given by $(O \setminus R, \leq, .class, m)$ whose reduct $(O \setminus R, .class, m)$ is a reduct of a VODAK core structure. The inheritance relation on $O \setminus R$ can be established by appropriately defining just "instance-types" (i.e. $.types[1]$). This unfortunately means that the presented common subfamily does not use the VODAK's metaclass facility (which is based on $.types[2]$).

Summary

Pursuing the title question, we have thoroughly investigated core constituents of the object model in many contexts of object technology. The list of environments (which were mostly programming languages) that we took into consideration contains at least 20 items: F&D model, Python, Smalltalk-76, ObjVLisp, LOOPs, CLOS, Java, Perl, Smalltalk-80, Objective-C, Ruby, Dylan, Newspeak, MCJ, Self, Io, JavaScript (ES5, ES6), RDF Schema, OWL 2, MOF + UML, VODAK.

We started with the book *Putting metaclasses to work* by Ira Forman and Scott Danforth [59], which we refer to as the F&D book, and investigated the core part of the object model described there. As a result, we obtained the axiomatic family of *Python core structures* which are mathematical structures constituted by a pair of relations between *objects*. The two relations are *instance-of* (ϵ) and *inheritance* ($\leq$). The relations can be recovered from their respective reductions $.class$ (*instantiation graph*) and $\prec$ (*inheritance graph*). We have used the reductions for the diagrammatization in which objects are depicted as nodes whereby $.class$ and $\prec$ are displayed as blue and green links between the nodes. Every structure contains a unique circular core which consists of two distinguished objects: the *inheritance root* r (named `Object` in the F&D book) and the *instantiation root* c (named `Class`). Similarly, in every structure there are two distinguished sets of objects:

classes are objects x such that $x \leq \underline{r}$ and ***metaclasses*** are objects x such that $x \leq \underline{c}$.

We have provided several equivalent axiomatizations. The axioms just specify exactly which combinations of blue and green links are allowed. The most interesting axiom is that of *monotonicity of* $\in$ which can be stated as either the order-theoretic monotonicity of *.class* w.r.t. $\leq$ (as postulated in the F&D book) or as the inclusion $(\leq) \circ (\in) \subseteq (\in)$. The latter form can be seen as the reversed counterpart of the *subsumption rule* $(\in) \circ (\leq) \subseteq (\in)$ (cf. [97]).

We have used the word *Python* for the name of the family since the structures are observable in the Python programming language. Unless some form of hacking is performed, the binary relations between objects that are introspected by `isinstance` and `issubclass` are subject to the same constraints as $\in$ and $\leq$ in Python core structures. This is no coincidence as the development of Python 2.2 has been influenced by the F&D book. Since Python has the strongest association with the metaclass term according to general Google search [$\mathbb{G}_1$], the circumstances let us feel confident that the family of Python core structures is the right initial setting for the metaclass approach to OOP foundations.

The established family of Python core structures has further been refined into a four-layered object model (mostly) according to the F&D book. The resulting family of $\mathbf{FD}_4$ structures provides an essential model of OOP with metaclasses. The family is built incrementally by abstraction refinement. Python core structures stand for the initial abstraction and are alternatively named $\mathbf{FD}_1$ structures. The next family of $\mathbf{FD}_2$ structures introduces ordering between the arrows of the inheritance graph known as *method resolution order (MRO)*. Using this ordering, each class has *linearized* ancestors. The family of $\mathbf{FD}_3$ structures adds *methods* as well as labelled links from classes to methods. In accordance to the F&D book, methods are abstract entities different from objects. After this step, each object is equipped with named methods to which it responds. (We should more precisely write "by which it responds to messages carrying the name".) The binding of objects with respondent methods is implicit and is derived from the explicit ownership of methods by classes using the MRO. This derivation is called *method resolution* and it is the main purpose of core structure. The essence of method resolution is given by what can be called $\in$*-logic*: an object x is a potential receiver of a named method owned by an object y iff $x \in y$. The assumption that methods are resolved via $\in$-logic can further be used for the recognition of $\in$ in a given environment.

In the (last) refinement step to $\mathbf{FD}_4$ structures, *instance variable* valuation is established, formed by labelled links between objects. These links constitute the "data part" of objects so that an object is an abstract entity that bundles data with behavior. There is no implicit binding of data, they are accessed by the object methods via $=$*-logic*. The difference in the logic used for the resolution of methods and data ($\in$-logic vs. $=$-logic) follows from the difference between methods and data: methods are typically shared but data are particularized.

Further on, we sought after a counterpart of $\mathbf{FD}_4$ structures in Python. The main difference between the Python object model and our F&D model is in the resolution of methods and data. In Python, methods and instance variables are unified into *attributes* which are in turn resolved via the "blended" $((\leq) \cup (\in))$-logic. However, a "de-blending" takes place for methods. For an object x, only the methods resolved via $\in$ are *invoked upon* x so that x is passed to the method body as the *invocant*. If a method is resolved via $\leq$ then the context of x is forgotten. We can therefore still feel confident that methods are meant to be resolved via $\in$-logic.

We have also thoroughly tested Python's conformance with the axiomatization of Python core structures. As a result, we found out that Python is not resistant against specifically crafted code and thus the axioms are generally not asserted. We have to guess that our tests are just hacks which – according to a "gentlemen's agreement" – are not meant to be used. In Python, such a guess is still relatively easy.

This way we have answered the metaclass question in the most important (or at least most popular) context of the Python programming language and its associated context of the F&D book. In the rest of the document various other relevant environments are considered. Before this investigation, an unpleasant phenomena is discussed called the *is-a misnomer*, a far-reaching terminological mistake of object technology. The mistake lies in using the *is-a* catchphrase as a name for $\leq$ instead of for $\in$, based on an asymmetric modification of the phrase "a `B` is a `T`". The misnomer does not appear in all environments and we in fact provided as much as a dozen of examples with a correct use of *is-a* including the F&D book or even a book by Guiseppe Peano. We also mentioned the (similar) *has-a misnomer* which is based on an asymmetric modification of "a `B` has a `T`".

Historically, the metaclass approach to OOP has been initiated by the Smalltalk-76 programming language. It is the first programming language that is subject to the metaclass pre-condition: classes are objects. However, only the built-in metaclass `Class` was allowed so that there is no mention of metaclasses in the language documentation. The family of Smalltalk-76 core structures is precisely defined as a subfamily of Python core structures constrained by the conditions of single metaclass and single inheritance. Moreover, the same names `Object` and `Class` are used for the two classes of the built-in circular core like in the F&D book.

In 1980s, the development of the metaclass approach split into two directions. The first direction was initiated by Smalltalk-80 which introduced *implicit* metaclasses. This metaclass model evolved into the Ruby core structures which we consider to be the (contemporary) endpoint of this evolution line. In the second, later direction, *explicit* metaclasses have been introduced, leading to the family of Python core structures, which we regard as the second (contemporary) endpoint. This development line was initiated by LOOPS and ObjVLisp – a pair of object-oriented extensions of the Lisp programming language. The ObjVLisp model [27] can be regarded as just the Python core model without the monotonicity condition. Because object models with explicit metaclasses lead to the Python's model and are also easier to describe we focused on them first.

The *Common Lisp Object System* (CLOS) has the strongest association with the metaclass term in academic circles. Paradoxically, the well known book *The Art of the Metaobject Protocol* (AMOP) [91] refrains from using the term and also casts doubts as to whether the "classes are object" principle applies to CLOS. We therefore had to cope with ambivalencies in CLOS literature. After ignoring AMOP, there were no problems with the recognition of objects, classes, $\in$ and $\leq$. The only problem was the delimitation of metaclasses – a consequence of built-in *monotonicity breaks* of the *.class* map. Interestingly, the CLISP interpreter of CLOS asserts (presumably with limitations) that *.class* monotonicity is preserved for *user*-defined classes.

The *Java* programming language is mentioned in the F&D book as an example of a single-metaclass language: "Java has a single metaclass `Class` of which all classes are instances."[59b] Indeed, one can recognize the `Object`–`Class` circular core $\{\underline{r}, \underline{c}\}$ with $\underline{r}.class = \underline{c}.class = \underline{c} < \underline{r}$ also in Java just like in the F&D model, Python, Smalltalk-76 or ObjVLisp. Similarly to Smalltalk-76, subclassing of `java.lang.Class` is disallowed. The same view is confirmed in *Java Reflection in Action* [61] which is another book co-authored by Ira Forman. However, the F&D model assumes the "classes are objects" principle to be a prerequisite for the definition of a metaclass. This metaclass pre-condition is mostly disapproved in the Java literature (paradoxically, it is both approved and disapproved in [61]). Similarly to AMOP, there is a distinction between a class and the *class object* that is the *reification* of the class. In order to get the Java's counterpart of Python core structures, it is necessary to remove the distinction. The presented family of *Java core structures* is a refinement of Smalltalk-76 core structures. In Java, the term *class* is not used for all descendants of $\underline{r}$ but just for a distinguished subset that must be specified explicitly. The complementary subset is that of *interfaces*.

In *Perl* (both Perl 5 and Perl 6), the "classes are objects" principle is established using the term *invocant* as a synonym for "object". Objects are just potential invocants of methods. Assuming the $\in$-logic for the resolution of methods, we have recognized the $\in$ relation as being correspondent to the Perl's built-in `isa` introspection method. Subsequently, the Perl's circularity has been detected: every class is the class of itself, classes are just the circular objects, and $\leq$ is the restriction of $\in$ to classes (in particular, $(\leq) \subseteq (\in)$). Due to this degeneracy, there is no counterpart of the `Class` class and thus there are no metaclasses. This makes Perl an example of a programming language in which classes of classes are not metaclasses.

Until after investigating Perl, we have pursued only those object models which only have *explicit* objects in their core structure. Apart from the built-in circular substructure, such core structures can be built by attaching objects *one-by-one*. These structures are simpler to describe which was one of the reasons why we pursued them first. Now we set out for the investigation of the other metaclass evolution line, historically the first one, namely that of *implicit metaclasses*. This way we arrived at definitely the biggest obstacle to the description of the metaclass approach to OOP: Smalltalk-80.

The *Smalltalk-80* programming language introduced the metaclass term into OOP. Unfortunately, the established terminology does not reflect the fact that metaclasses in Smalltalk-80 are *implicit* and only appear with one-to-one links to other objects, unlike in Python. Consequently, there is a problem with delimitation of classes. The Smalltalk-80 literature is ambiguous as to whether metaclasses are classes or not. That is, it is usually stated that metaclasses *are* classes but subsequently contradicted by statements about "parallel hierarchies" between classes and metaclasses. We have settled the situation by *not regarding* the Smalltalk's implicit metaclasses as classes. This in turn creates a need to distinguish between $\in$ and *instance-of*: x is an instance of y iff $x \in y$ and y is a class. Simultaneously, the *.class* map has to be defined such that $(.class) \circledcirc (\leq)$ is the instance-of relation and thus the map only partially corresponds to the built-in introspection method named `class`. If x is a class then $x.class$ equals `Class` just like in Smalltalk-76.

We have established an ad-hoc concept of a *metalevel* according to the reachability of the `Metaclass` class via "blue" links. Classes are then delimited as the metalevel 1, whereas implicit metaclasses form the metalevel 2. Due to built-in monotonicity breaks there are several possibilities how to delimit *explicit* metaclasses (similarly to CLOS). More importantly, we have introduced the concept of *powerclass links* for the one-to-one correspondence between metalevels 1 and 2. Such links from x to y are characterized by the condition $x.\downarrow = y.\ni$ and $x.\in = y.\updownarrow$. The first of the two equalities offers a correspondence with the *powerset* operator, an observation

not found in Smalltalk-80 literature.

The *Objective-C* programming language adopted the Smalltalk's concept of a parallel hierarchy of implicit metaclasses but without the `Metaclass` class. As a result, there are no monotonicity breaks – the $(\leq) \circledcirc (\in) \subseteq (\in)$ rule is asserted. However, there are multiple inheritance roots which are at the same time the only circular classes, so that there is no metalevel 1 correspondent to the `Class` class.

A conceptually clean object model with implicit metaclasses has been established in the *Ruby* programming language. As of version 1.9.1 and newer, the concept of powerclass links is applied universally: every object x has a ***powerclass*** (also called *eigenclass* or "singleton class"), denoted $x.ec$. A *Ruby core structure* is given by $.ec$ and $\leq$ just like a Python core structure is given by $.class$ and $\leq$. The object *membership* relation $\in$ equals $(.ec) \circledcirc (\leq)$. In contrast to the $.class$ map, the powerclass map $.ec$ constitutes *infinite regress* that has to be lazily evaluated. More specifically, the $.ec$ map is an order embedding w.r.t. $\leq$ which also means that monotonicity of $\in$ is asserted. Up to the number of circular classes there is a one-to-one correspondence between Smalltalk-76 core structures and Ruby core structures. The latter arise from the former by powerclass *completion*, and the former are obtained from the latter by removing powerclasses. We have delimited *classes* as those `Class`es that are not powerclasses. *Metaclasses* are the descendants of the `Class` class and this class is also the only explicit metaclass. Otherwise, all metaclasses are implicit (i.e. powerclasses).

Having formalized the two main contemporary metaclass models – the Python's with *explicit* metaclasses and the Ruby's with *implicit* metaclasses, we continued with a formalization of the general core constrained by the *monotonicity* condition. A common super-family of Python and Ruby core structures is obtained by powerclass completion of Python core structures. A distillation of five essential axioms results in obtaining the general family of *monotonic powerclass structures*. A finer family is obtained by allowing $.ec$ to be partial which allows for the expression of the current state of evaluation of (Ruby's) powerclass regress. In these structures, we have introduced the *anti-membership* relation $\in^{-1}$ and defined the *metalevel index* $x.mli$ of an object x. ***Metaclasses* are** the anti-members of r, i.e. **objects from metalevel 2 or higher**. This can be regarded as the definitive answer for the title question except that in families that support the existence of the top *explicit* metaclass $r.class$ (usually named `Class`) – as a unique "duplicate" of the top *implicit* metaclass $r.ec$ – the delimitation should be extended to include the duplicate too.

At this point we set out to discover the ultimate set-theoretic semantics of $\in$, $\leq$ and $.ec$. Apparently, should correspondences $\in \leftrightarrow \in$ and $\leq \leftrightarrow \subseteq$ be established then the monotonicity condition $(\leq) \circledcirc (\in) \subseteq (\in)$ has to be dropped and $.ec$ in the $(\in) = (.ec) \circledcirc (\leq)$ equality has to be replaced by an abstraction of the singleton-set mapping $x \mapsto \{x\}$. The solution is provided by the family of *metaobject structures* which arise by expansion of monotonic powerclass structures. In addition to $.ec$ and $\leq$, there is one more definitory constituent – the *singleton* map $.\varepsilon c$. In these structures the *object membership* relation $\in$ is defined as the union $(\in) \cup (\overline{\in})$ where $(\in) = (.\varepsilon c) \circledcirc (\leq)$ is the *bounded membership* and $(\overline{\in}) = (.ec) \circledcirc (\leq)$ is the *power membership*. The precise correspondence between object membership and set membership is then expressed as $\in \leftrightarrow \in$.

In contrast to the powerclass map $.ec$, the singleton map $.\varepsilon c$ is not total but only defined for *bounded* objects. Boundedness of an object x is given by its *rank*, denoted $x.d$, which is a recursively defined ordinal number, at most equal to a fixed limit ordinal ϖ. An object x is bounded iff $x.d < \varpi$. Since objects that are non-well founded in $\in$ have by definition the maximum rank, it follows that $\in$ is a well-founded relation.

Just like there is a refinement of monotonic powerclass structures in which $.ec$ is partial, it is also possible to generalize metaobject structures so that both $.ec$ and $(.\varepsilon c) \setminus (.ec)$ are arbitrarily partial, possibly empty. The resulting family is that of *basic structures* which are elaborated in a special document [138]. This family forms a common generalization of many families provided in the present document, see the diagram. Set-theoretic representation of a basic structure (and thus also e.g. of a Python core structure) is provided by a gradual completion into an $(\varpi+1)$*-superstructure* which is embedded into the von Neumann universe of well-founded sets.

The *Dylan* programming language is probably the first language that provides an implementation of the singleton map $.\varepsilon c$. There is an infinite regress of singletons, similarly to Ruby's infinite regress of powerclasses. In fact, the introduction of Ruby's term "singleton class" can be attributed to the fact that for a terminal object x, $x.ec = x.\varepsilon c$. In contrast to our formalizations, Dylan supports singletons for *all* objects, including circular classes. Moreover, Dylan also supports lazily evaluated powerclasses of classes. In Dylan, a powerclass is termed "*subclass type*" and is an instance of the built-in class named `<subclass>`. This class plays the same role as the `Metaclass` class in Smalltalk-80 and thus the same monotonicity break is introduced. This also offers several posibilities for the delimitation of explicit metaclasses. Dylan's implicit metaclasses are the "subclass types" together with singletons except for the singletons of terminal objects. Such a terminological

identification is of course nowhere to be found in Dylan's literature. Unlike in Smalltalk-80, there is no ambiguity in the delimitation of classes – "subclass types" and singletons are consistently regarded as "non-class types".

Next, we explored a formal model named *MCJ*, described in the article *A Core Calculus of Metaclasses* [78]. By our considerations, the model exhibits a terminologic inadequacy somewhat opposite to that of Dylan's: the term "metaclass" is used in the wrong place. In MCJ, metaclasses are delimited as classes of classes (presumably together with the ancestors) without imposing the monotonicity condition. This allows for dispension with the `Class` class. Apparently, the authors do not require that *all* instances of a metaclass must be classes but use the "*some*" quantifier instead. This can also be formulated as using the rank *.d* ("some") instead of the metalevel index *.mli* ("all"). We call such an interpretation the *metaclass misnomer*. By our definitions (and in a correspondence to some authors from the field of knowledge representation [104] [95]), there are no metaclasses in MCJ. As a correlated issue, in MCJ, each class is an instance of a class, but classes are not considered to be objects.

The next environment to investigate was *Self*, a canonic representant of prototype-based programming languages (PBLs). In these languages, named methods (or arbitrarily valued "slots" in general) are resolved via ≤-logic. This is usually thought to be in contrast with "class-based" languages where ϵ-logic is used instead. We have proposed a formalization in which $(\epsilon) = (\leq)$. That is, every object is the class of itself – "object" and "class" are synonyms. Every object resides on metalevel 1 so that there are no metaclasses. This can also be seen as a consequence of Self core structures forming a subfamily of Perl core structures.

The *JavaScript* programming language performs resolution of methods (or "properties" in general) via ≤-logic. Simultaneously, there is a built-in support for a finer core structure with implicit objects called *prototypes*. Similarly to the correspondence between classes and implicit metaclasses in Smalltalk-80 or Objective-C, there is a one-to-one correspondence between prototypes and classes in JavaScript, especially in the ECMAScript 6 version. Moreover, this correspondence is that of powerclass links: if *y* is a class and *x* is the prototype object that corresponds to *y* (so that *x* equals *y*`.prototype`) then *x* is the common inheritance ancestor of all instances of *y* – written as $x.\downarrow = y.\ni$. The *instance-of* relation between objects is suggested by the built-in `instanceof` operator. Since only single inheritance is allowed and there are exaclty two built-in circular classes (named `Object` and `Function`, the latter corresponding to the Smalltalk's `Class` class) a JavaScript core structure arises by *prototype completion* of a Smalltalk-76 core structure. This introduces inheritance to metalevel 0, with `Object.prototype` being the top object.

Before we abandoned the context of programming languages, we made an observation of the anathema of metaclasses in literature on mathematical modelling of OOP. Apparently, the metaclass approach has very little support in OO research so that standard literature does not mention the term "metaclass" at all.

In the rest of the document the metaclass approach is applied beyond OOP. The field of *knowledge representation* is where the metaclass term appeared first. In the presumably seminal work [102], the same circular core $\{\underline{r},\underline{c}\}$ as in Smalltalk-76 has been introduced, with $\underline{r}.\downarrow$ being the set of *classes* and $\underline{c}.\downarrow$ the *metaclasses*. Moreover, the $\underline{r}.\downarrow = \underline{c}.\ni$ equality is asserted. In contrast to programming languages, there is typically no monotonicity of ϵ and no *.class* map (the latter due to multiple classification). Moreover, there can be a distinguished subset of objects called *properties* which are not classes but can still be related by inheritance.

The core part of *RDF Schema* shares several features with the model presented in [102], including the delimitation of classes and metaclasses. Some authors stress the difference between $\underline{O}.\epsilon^2$ and $\underline{c}.\downarrow$ with the latter being the correct delimitation of metaclasses. The $\underline{r}.\downarrow \supseteq \underline{c}.\ni$ and $(\epsilon) \circledcirc (\leq) \subseteq (\epsilon)$ inclusions are expressed by the rdfs8 and rdfs9 rules, respectively. Moreover, the built-in structure can almost be regarded as a Python core structure and is thus compatible with the F&D model.

The *OWL 2 Full* ontology language is based on RDF Schema. There is a distinguished built-in class, `owl:Nothing`, that is asserted to be a descendant (a subclass) of every class and thus be an abstraction of the empty set. We do not regard this class as a metaclass, despite its inheritance relation to $\underline{c}$.

We have also investigated the *metamodelling* context. In contrast to most other enviroments, we only provided an informal exposition. There are several different concepts of the *instance-of* relation in metamodelling, even with different views as to whether the *instance-of-instance-of* composition is legal. A recent trend can be observed towards a unification based on the ObjVLisp model. This also affects the metaclass concept which was considered self-contradictory in 2005 but has been rehabilitated in 2013 as the "*basis for [a new metamodelling] language definition*".

Finally, we paid attention to (presumably) the first ever book that bears the metaclass term in its title. The book's main title reads *Metaclasses and their application* [93]. The subject is an object-oriented data model of a

database management system called *VODAK*. At the core of the model there is a decoupling of instantiation and inheritance. Instantiation is defined between *objects* and is just the *.class* map, whereas inheritance occurs between *types*. Each object *x* is linked to a list *x.types* of either 1, 2 or 3 types according to its *x*'s metalevel. The types *x.class(i).types[i]* (*i = 0,1,2*) are then consecutive startpoints of a 3-phase method lookup.

Obstacles to OOP foundations

We have encountered several obstacles to foundations of object technology.

1. **The code-first approach**. This is just a pejorative name for the *calculus* approach taken by the majority of OOP research. The calculus approach attempts to imitate the construction of the foundations of functional programming. A counterpart of lambda calculus is sought that would be suitable for modelling object-oriented features. The underlying (data) abstractions of ϵ and $\leq$ are unfocused and available only implicitly if at all. The approach is at odds with OO principles since it focuses on computation and not on the data.
2. **The type-first approach**. This approach, usually coupled with the calculus approach, takes the view that a foundational object-model should only be introduced together with a sophisticated facility, called (static) type system, that would restrict possible state transitions. A type system in particular restricts instance variable valuation and method invocation. The former necessitates refinement of classes by instance variable declarations, the latter requires refinement of methods by argument signatures.

   Although it might indeed be the case that statically typed object-oriented languages are superior to dynamic object-oriented languages (w.r.t. large-scale software development), the dynamic languages like Ruby, Python or JavaScript became so popular that they cannot be ignored. That is, there is a significant part of OOP world where there are no types (in the sense of computer science). This not only makes type systems at least partially dispensable but also creates a need to build type-free foundational core of OOP. We have established such a core by focusing on it, i.e. by focusing on ϵ and $\leq$.
3. **Classlessness of PBLs**. Prototype-based programming languages (PBLs) are said to be "classless". The main reason behind this viewpoint appears to be the metaclass anti-condition. In PBLs, methods (including shared methods) are bound to objects and this binding can be modified at run-time. Because classes are not considered to be objects (runtime entities) they cannot be modified dynamically. Interestingly, the wikipedia page titled *Prototype-based programming*, version February 2006 [$\mathbb{W}_9$], lists Smalltalk as an exceptional example of "*very few class-based object-oriented systems (…) [that] allow classes to be altered during the execution of [a] program*". Ten years later, the same sentence is present but the list of the "very few" consists of CLOS, Dylan, Objective-C, Perl, Python, Ruby and Smalltalk [$\mathbb{W}_{10}$].

   A much more comprehensible view of PBLs will be obtained if they are said to be "classful", being subject to the degeneracy condition $(\epsilon) = (\leq)$. Such a view is presented in [77a] where PBLs are considered to form a "*1 level system*" in which "*all objects can be viewed as classes and all classes can be viewed as objects*".
4. **Missing class entities**. The Java programming language and its documentation suggest metaclass anti-condition: there is a distinction between classes and their reification. However, this distinction has to be guessed since there is no definition of what a class is supposed to be. That is, there is no model in which classes are clearly defined entities. The Java Language Specification [70] introduces classes in a chapter titled *Classes* whose first sentence refers to *class declarations*. The second sentence already assumes that the notion of a class is defined. A similar trick is used in Featherweight Java, see [143].

   To a lesser extent, a similar problem appears in CLOS where some literature (in particular the AMOP book [91]) makes a distinction between "class" and "class metaobject".
5. **Smalltalk-80 core features**. Due to its historical significance, the Smalltalk-80 programming language is a language that poses the biggest obstacle to the description of ϵ and $\leq$. While introducing many revolutionary concepts including the metaclass pre-condition, Smalltalk-80 unfortunately also gave rise to what can be called a *paradigm of sloppiness*. There are several conceptually flawed features which tend to mask each other.
   - **Unrecognized powerclass links**. It has not been properly recognized (neither in Smalltalk-80 literature nor in the language introspection facilities) that there is a fundamental semantic difference between the the two groups of blue links: (a) the many-to-one links between terminal objects and their classes and (b) the one-to-one links between classes and their implicit metaclasses. The (b) links are abstraction of the powerset operator and thus form a one-step evaluation of infinite regress. The support of exactly one evaluation step is an implementation feature that was mistaken for a concept.
   - **Ambiguos delimitation of classes**. Classes are unanimously said to form a subset of objects, but the

subset is specified ambiguously. There are two delimitations: (A) classes are exactly the non-terminal objects, and (B) classes are just those non-terminal objects that are not `Metaclasses`. The (A) variant supports uniformity of blue links, the (B) variant allows statements about "parallel hierarchies" between classes and metaclasses.

- **Unrecognized monotonicity breaks**. Despite the correlation of terms *monotonic map* and *(least) fixed point* in computer science, the occurrence of a 2-element cycle in the instantiation graph of Smalltalk-80 has not been put into connection with non-monotonicity of the "`class`" map.

  Moreover, the built-in monotonicity breaks complicate the delimitation of metaclasses.
- **Jungle structures**. Instantiation or subclassing of some built-in classes or using the `superclass:` method allows for the creation of "jungle structures". This might be desired for experimentation but also creates a need to specify which structures should be regarded as standard. Unfortunately, there are no such guidelines in the Smalltalk-80 literature.

Some of these features apply to other programming languages, in particular CLOS and JavaScript.

6. **The *is-a* misnomer**. This is probably the biggest misnomer of object technology. An inheritance (subclass) relationship between classes (or concepts) `B` and `T` is named by the *is-a* catchphrase (alternatively, *isa* or *ISA*), based on the following sentence:

   If `B` ≤ `T` then **a** `B` *is a* `T`. (If `B` is a subclass of `T` then an instance of `B` is also an instance of `T`.)

   The resulting phrase "`B` *is-a* `T`" arises by asymmetric modication of "**a** `B` *is a* `T`" – the `B`'s article is dropped while the `T`'s article is preserved.
7. **The *metaclass* misnomer**. This can be understood as inserting "*some of*" instead of "*all of*" before "*whose*" in the classic definition of metaclasses. The misnomer appears when classes of classes are considered metaclasses in "non-pythonic" context, i.e. in environments that are not subject to non-degeneracy and monotonicity conditions asserted for Python core structures. *A Core Calculus of Metaclasses* [78] is a notable article that uses the metaclass misnomer in its terminology.
8. **The *power-type* misnomer**. This misnomer arose in metamodelling by confusing "powerset of" with "partition of". The (wrong) "partition of" interpretation is supported by the OMG standard [125].
9. **The *instance-of* ambiguity**. In general, there are at least 3 different interpretations of "*x* is an instance of *y*":

   (a) $x.class = y$, (b) $x.class \leq y$, (c) $x \in y$.

   The (b) case should more precisely be formulated as: $x \in y$ and y is class. This is what we consider to be the right definition of *instance-of*. The (a) case stands for *direct instance-of*, while (c) states the *(object) membership* relationship between x and y, respectively. If non-terminal objects are exactly the classes as in Python then (b) and (c) are coincident.

   In Python core structures, *direct* instances of a given class x are either all classes or all terminal. This makes the distinction between "all of" and "some of" in the definition of a metaclass unnecessary – provided that the "*direct*" adjective is not omitted. Without the adjective and assuming the "some of" quantification, the inheritance root *r* is incorrectly recognized as a metaclass.

Conclusion

(*This dark part of the document is yet to be written.*)

Epilogue

In our opinion, formalization of practical languages is a task which nobody is able and wants to do. […] Moreover, such formalization is below the level of mathematical aesthetic and elegance and is not intellectually interesting; though, these are the main driving forces of the mathematical research.

Kazimierz Subieta et al. [171]

References

[1] Martin Abadi, Luca Cardelli, ***A Theory of Objects***, Springer Science & Business Media 1996, http://lucacardelli.name/theoryofobjects.html
[1a] Preface , [1b] search: metaclass , [1c] Citations (2015)

[2] Moez A. AbdelGawad, ***NOOP: A Mathematical Model of Object-Oriented Programming***, PhD thesis, Rice University 2012,
[2a] Citations (2015)

[3] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, Mario Wolczko, ***The SELF 4.1 programmer's reference manual***, Tech. rep., Sun Microsystems, Inc. and Stanford University 2000,

[4] Marty Alchin, ***Pro Python***, Apress 2010,

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

[6] Jürgen Angele, Michael Kifer, Georg Lausen, ***Ontologies in F-logic***, Handbook on Ontologies, Springer Berlin Heidelberg 2009,
[6a] isa-F-atom

[7] Minero Aoki, ***Ruby Hacking Guide***, 2004, (translated by Vincent Isambart, translations and additions by C.E. Thornton, 2008, https://ruby-hacking-guide.github.io/)
[7a] Classes and modules

[8] David Ascher, Alex Martelli, Anna Ravenscroft, ***Python Cookbook***, O'Reilly 2005
[8a] bible of metaclasses

[9] Colin Atkinson, Thomas Kühne, ***Model Driven Development: A Metamodeling Foundation***, Software, IEEE 20.5 2003

[10] 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

[11] Jacobus W. de Bakker, Willem-Paul de Roever, Grzegorz Rozenberg, ***Foundations of Object-Oriented Languages: REX School/Workshop 1990***, Springer Science & Business Media 1991

[12] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, P. Tucker Withington, ***A Monotonic Superclass Linearization for Dylan***, Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications , ACM New York, 1996, http://www.webcom.com/haahr/dylan/linearization-oopsla96.html
[12a] C3 - A linearization consistent with the EPG

[13] David M. Beazley, ***Python essential reference***, Addison-Wesley Professional 2009

[14] Jean Bézivin, Richard Lemesle, ***Ontology-based layered semantics for precise OA&D modeling***, ECOOP'97 1998
[14a] instantiation link (isA)

[15] Andrew P. Black, Kim. B. Bruce, Michael Homer, James Noble, ***Grace: the absence of (inessential) difficulty***, Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software 2012

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

[17] Günter Blaschek, ***Object-oriented Programming: With Prototypes***, Springer Science & Business Media 2012
[17a] [OOP] focuses on the data

[18] 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,

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

[20] 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,
[20a] An object is not a class , [20b] state + behavior + identity , [20c] Smalltalk is a typeless language ,

[20d] object is an instance

[21] Ronald J. Brachman, ***A Structural Paradigm for Representing Knowledge***, Report No. 3605, Bolt, Beranek and Newman, Inc., Cambridge Mass. 1978,

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

[23] Gilad Bracha, ***Newspeak Programming Language Draft Specification, Version 0.095***, 2014, //bracha.org/newspeak-spec.pdf

[24] Keith Bradnam, Ian Korf, ***UNIX and Perl to the rescue!: a field guide for the life sciences***, Cambridge University Press 2012,
[24a] [OOP] focuses on data

[25] Kim B. Bruce, ***Foundations of Object-Oriented Programming Languages: Types and Semantics***, MIT Press 2002,
[25a] Overview from the publisher , [25b] search: metaclass

[26] Jos de Bruijn, ***Logics for the Semantic Web***, Semantic Web Services: Theory, Tools and Applications Idea Group Inc 2007,
[26a] is-a assertion

[27] 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,

[28] 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,

[29] 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

[30] Luca Cardelli, ***Class-based vs Object-based Languages***, PLDI'96 Tutorial, 1996,
[30a] Prototypes

[31] Victorio A. de Carvalho, João Paulo A. Almeida, ***Towards a Well-Founded Theory for Multi-Level Conceptual Modelling***, 2015,

[32] Giuseppe Castagna, ***Object-Oriented Programming A Unified Foundation***, Springer Science & Business Media, 2012,
[32a] search: metaclass , [32b] *isa* pointer

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

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

[35] David Chisnall, ***Objective-C Phrasebook***, Addison Wesley Professional 2012,
[35a] Understanding the `isa` pointer , [35b] prototype-based object orientation

[36] David Chisnall, ***A new Objective-C runtime: from research to production***, Communications of the ACM 55.9 2012, http://queue.acm.org/detail.cfm?id=2331170&ref=fullrss,

[37] Tony Clark, Cesar Gonzalez-Perez, Brian Henderson-Sellers, ***A foundation for multi-level modelling***, MULTI@ MoDELS 2014,

[38] 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,

[39] Pierre Cointe, ***A Tutorial Introduction to Metaclass Architecture as provides by Class Oriented Languages***, FGCS 1988,

[40] Damian Conway, ***Object Oriented Perl***, Manning Publications, 2000,
[40a] no metaclasses

[41] William Cook, ***A Proposal for Simplified, Modern Definitions of "Object" and "Object Oriented"***, William Cook's Fusings 2012, http://wcook.blogspot.com/2012/07/proposal-for-simplified-modern.html

[42] Iain D. Craig, ***The interpretation of object-oriented programming languages***, Springer Science & Business Media, 2012,
[42a] Slots and Methods

[43] Iain D. Craig, ***Programming in Dylan***, Springer Science & Business Media, 2012,
[43a] meta-class

[44] 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

[45] Steve Dekorte, ***Io: a small programming language***, Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM, 2005

[46] François-Nicola Demers, Jacques Malenfant, ***Reflection in logic, functional and object-oriented programming: a Short Comparative Study***, Proceedings of the IJCAI. Vol. 95, 1995

[47] Christophe Dony, Jacques Malenfant, Daniel Bardou, ***Classifying Prototype-based Programming Languages***, Prototype-based Programming: Concepts, Languages and Applications 86, 1998

[48] Roland Ducournau, et al., ***Monotonic conflict resolution mechanisms for inheritance***, ACM SIGPLAN Notices, Vol. 27. No. 10, ACM 1992

[49] Roland Ducournau, et al., ***Proposal for a Monotonic Multiple Inheritance Linearization*** , ACM SIGPLAN Notices. Vol. 29. No. 10, ACM 1994

[50] Roland Ducournau, Jean Privat, ***Metamodeling semantics of multiple inheritance***, Science of Computer Programming 76.7, 2011

[51] Bruce Eckel, ***Thinking in Java***, Prentice Hall Professional, 2003
[51a] late binding

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

[53] Ecma International, ***ECMA-262 6th Edition, The ECMAScript 2015 Language Specification***, http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf
[53a] Let *constructorParent* be *superclass*.

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

[55] Erik Ernst, ***Safe Dynamic Multiple Inheritance***, DAIMI Report Series 31.556, 2002

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

[57] David Flanagan, Yukihiro Matsumoto, ***The Ruby Programming Language***, O'Reilly 2008
[57a] eigenclass

[58] 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/
[58a] Nonclass Types , [58b] classes are objects

[59] Ira R. Forman, Scott H. Danforth, ***Putting Metaclasses to Work***, Addison Wesley 1998
[59a] *class* is monotonic , [59b] Java has a single metaclass

[60] Ira R. Forman, ***Declarable modifiers: A proposal to increase the efficacy of metaclasses***, Reflection and Software Engineering, Springer Berlin Heidelberg 2000
[60a] In Java, `Class` is the only metaclass

[61] Ira R. Forman, Nate Forman, ***Java Reflection in Action***, Manning Publications 2005
[61a] Figure 1.6 (circular core) , [61b] in Java, classes are objects ,
[61c] distinction between class and class object ,
[61d] metaclasses are [...] classes that produce classes when instantiated, [61e] glossary: metaclass

[62] Martin Fowler, ***UML distilled: a brief guide to the standard object modeling language***, Addison-Wesley 2004

[63] Ulrich Frank, ***Thoughts on classification / instantiation and generalisation / specialisation*** , ICB-

Research Report, No. 53, 2012

[64] Michael R. Genesereth, Richard E. Fikes, ***Knowledge Interchange Format Version 3.0: Reference Manual***, 1992,

[65] Jonathan Gillette (_why), ***Seeing Metaclasses Clearly***, 2005, http://whytheluckystiff.net/articles/seeingMetaclassesClearly.html

[66] Adele Goldberg, David Robson, ***Smalltalk-80: The Language and Its Implementation***, Addison Wesley 1983, http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf

[67] Stephen J Goldsack, Stuart JH Kent, eds., ***Formal methods and object technology***, Springer Science & Business Media 2012 (1996),
[67a] object identity

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

[69] Cesar Gonzalez-Perez, Brian Henderson-Sellers, ***Modelling software development methodologies: A conceptual foundation***, Journal of Systems and Software 80.11 2007

[70] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, ***The Java Language Specification***, 2015 http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf
[70a] §4.3.2 , [70b] Classes

[71] Peter Grogono, Mark Gargul, ***A graph model for object oriented programming***, ACM SIGPLAN Notices, 1994

[72] Jonathan L. Gross, Jay Yellen, Ping Zhang, Eds., ***Handbook of Graph Theory***, CRC Press, 2013

[73] Carl A. Gunter, ***Theoretical aspects of object-oriented programming: types, semantics, and language design***, Mit Press, 1994

[74] Bruno Haible, Michael Stoll, Sam Steingold, ***Implementation Notes for GNU CLISP***, 2010 http://www.clisp.org/impnotes.html
[74a] CLOS:VALIDATE-SUPERCLASS

[75] Daryl D. Harms, Kenneth McDonald, ***The Quick Python Book***, Manning, 2000

[76] Robert Harper, ***Practical foundations for programming languages***, Cambridge University Press, 2012
[76a] search: metaclass

[77] Robert J. Hathaway, ***Object-Orientation FAQ***, Object Magazine Online, 1995 http://www.ipipan.gda.pl/~marek/objects/faq/
[77a] What Is A Meta-Class?

[78] Sam Tobin-Hochstadt, Eric Allen, ***A Core Calculus of Metaclasses***, Fundamentals of Object-Oriented Languages (FOOL), 2005

[79] Arthur H.M ter Hofstede, Th.P. van der Weide, ***Expressiveness in Conceptual Data Modelling***, Data & Knowledge Engineering 10.1 1993
[79a] *Instances of power types are […] not necessarily all […]*

[80] John Hunt, ***Java and Object Orientation: An Introduction***, Springer Science & Business Media 2002

[81] Naftali Harris, ***Python Subclass Relationships Aren't Transitive***, 2014 http://www.naftaliharris.com/blog/python-subclass-intransitivity/

[82] Denis Howe, ***The Free On-line Dictionary of Computing***, http://foldoc.org/
[82a] Metaclass , [82b] Metaclass (as of 4 May 2001)

[83] Atsushi Igarashi, Benjamin C. Pierce, Philip Wadler, ***Featherweight Java: a minimal core calculus for Java and GJ***, ACM SIGPLAN Notices. Vol. 34. No. 10, ACM, 1999
[83a] search: metaclass

[84] 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

[85] Dan Ingalls, Bert Freudenberg, Ted Kaehler, Yoshiki Ohshima, Alan Kay, ***Reviving Smalltalk-78***, 2014

[86] IPA Ruby Standardization WG, ***Ruby Draft Specification***, 2010,
https://www.ipa.go.jp/osc/english/ruby/
[86a] Preliminary version (December 1, 2009)

[87] Peter Jackson, ***Introduction to Expert Systems***, Addison-Wesley 1999

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

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

[90] Michael Kifer, Georg Lausen, James Wu, ***Logical foundations of object-oriented and frame-based languages***, Journal of the ACM 1995

[91] Gregor Kiczales, Jim D. Rivieres, Daniel G. Bobrow, ***The Art of the Metaobject Protocol***, MIT press 1991
[91a] class metaobject represents the class , [91b] refrain from using "metaclass"

[92] Wolfgang Klas, Karl Aberer, Erich Neuhold, ***Object-Oriented Modelling for Hypermedia Systems Using the VODAK Model Language***, Advances in object-oriented database systems Springer 1994
[92a] initial subtype hierarchy

[93] Wolfgang Klas, Michael Schrefl, ***Metaclasses and Their Application: Data Model Tailoring and Database Integration***, Springer 1995
[93a] object identity

[94] Scott Knaster, Mark Dalrymple, Waqar Malik, ***Learn Objective-C on the Mac: For OS X and iOS***, Apress 2012,
[94a] concentrate on data

[95] 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

[96] Seiji Koide, Hideaki Takeda, ***Meta-Circularity and MOP in Common Lisp for OWL Full***, Proceedings of the 6th European Lisp Workshop ACM 2009 http://www-kasm.nii.ac.jp/~koide/SWCLOS2/KoideELW.pdf

[97] Seiji Koide, ***Theory and Implementation of Object Oriented Semantic Web Language*** , diss., Sokendai 2010 http://www.nii.jp/graduate/thesis/pdf/201103/koide_Dr_thesis.pdf

[98] Jukka K. Korpela, ***Mathematical Expressions***, Suomen E-painos Oy 2014
[98a] lunate epsilon

[99] Thomas Kühne, ***Matters of (meta-) modeling***, Software & Systems Modeling 5.4 2006

[100] Thomas Kühne, Daniel Schreiber, ***Can Programming be Liberated from the Two-Level Style? Multi-Level Programming with DeepJava***, ACM SIGPLAN Notices. Vol. 42. No. 10 2007

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

[102] Hector Levesque, John Mylopoulos, ***A procedural semantics for semantic networks***, Associative networks: Representation and use of knowledge by computers 1979
[102a] metaclass , [102b] Fig.17 , [102c] top of the hierarchy [of metaclasses]

[103] Mark Lutz, ***Learning Python***, O'Reilly 2009
[103a] Metaclasses are […] the most advanced topic

[104] Aimilia Magkanaraki, Sofia Alexaki, Vassilis Christophides, and Dimitris Plexousakis, ***Benchmarking RDF Schemas for the Semantic Web***, ISWC, 2002,
[104a] subclass of `rdfs:Class` ↔ metaclass

[105] Ionel C. Mărieș, ***Understanding Python metaclasses***, 2015
http://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/

[106] Gérald Masini et al., ***Object-Oriented Languages***, Academic Press 1991
[106a] isa

[107] 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

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

[109] Drew McDermott, ***Artificial intelligence meets natural stupidity***, ACM SIGART Bulletin 57 1976

[110] Wolfgang De Meuter, Theo D'Hondt, Jessie Dedecker, ***Intersecting Classes and Prototypes***, Perspectives of System Informatics, Springer Berlin Heidelberg, 2003

[111] Bertrand Meyer, ***Object-oriented Software Construction***, Prentice Hall, 1997
[111a] OBJECT MOTTO , [111b] inheritance , [111c] object identity , [111d] object is an instance

[112] Mira Mezini, ***Towards variational object-oriented programming: The rondo model***, Tech. Rep. TUD-ST-2002-02, Software Technology Group, Darmstadt University of Technology, 2002

[113] Mira Mezini, ***Variational Object-Oriented Programming Beyond Classes and Inheritance***, Springer Science & Business Media, 2013
[113a] Figure 5.3

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

[115] John C. Mitchell, ***Concepts in programming languages***, Cambridge University Press, 2003
[115a] Dynamic lookup

[116] Daniel Minoli, ***Enterprise architecture A to Z: frameworks, business process modeling, SOA, and infrastructure technology***, CRC Press, 2008
[116a] An example of the four-layer metamodel hierarchy , [116b] instance-of exactly one

[117] Hanspeter Mössenböck, ***Object-oriented programming in Oberon-2***, Springer Science & Business Media 2012
[117a] [OOP] focuses on the data , [117b] [OOP] focuses on the objects rather than on procedures , [117c] concentration on the data , [117d] inheritance

[118] Paulo J. L. de Moura, ***Logtalk: Design of an object-oriented logic programming language***, 2003
http://logtalk.org/papers/thesis.pdf

[119] Bernd Neumayr, Michael Schrefl, Bernhard Thalheim, ***Modeling techniques for multi-level abstraction***, The evolution of conceptual modeling Springer Berlin Heidelberg 2011

[120] Oscar Nierstrasz, Stéphane Ducasse, Damien Pollet, ***Pharo by Example***, Square Bracket Associates 2009, http://pharobyexample.org
[120a] classes must also be instances of classes , [120b] essence of *is-a*

[121] James Noble, Jan Vitek, John Potter, ***Flexible alias protection***, Springer Berlin Heidelberg, 1998
[121a] object identity

[122] Jan Pettersen Nytun, ***Consistency Modeling in a Multi-Model Architecture***, Diss. University of Oslo 2010

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

[124] OMG, ***OMG UML Infrastructure 2.4.1***, Object Management Group 2011
http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/

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

[126] Piet van Oostrum, ***Metaclasses in object oriented programming languages***, Liber Amicorum voor Doaitse Swierstra 2013, http://www.staff.science.uu.nl/~hage0101/liberdoaitseswierstra.pdf

[127] Oracle Inc., ***API specification for the Java™ Platform, Standard Edition***, 2015,
http://docs.oracle.com/javase/8/docs/api/overview-summary.html
[127a] `java.lang.Class`

[128] Meilir Page-Jones, , Larry L. Constantine, ***Fundamentals of object-oriented design in UML***, Addison-Wesley Professional 2000
[128a] object identity

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

Conceptual Modelling, 2012

[130] Tobias Pape, Arian Treffer, Robert Hirschfeld, Michael Haupt , ***Extending a Java Virtual Machine to Dynamic Object-oriented Languages***, Universitätsverlag Potsdam, Vol. 82, 2014,
[130a] `java.lang.Class`

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

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

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

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

[135] Ondřej Pavlata, ***Object Membership: The Core Structure of Object Technology***, 2015, http://www.atalon.cz/om/object-membership/

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

[137] Ondřej Pavlata, ***Object Membership: The ontological structure***, 2012, http://www.atalon.cz/om/object-membership/ontology/
[137a] OWL built-in structure

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

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

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

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

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

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

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

[145] Paolo Perrotta, ***Metaprogramming Ruby 2***, Pragmatic Bookshelf 2014

[146] Tim Peters, ***Acrimony in c.l.p.***, 2002, https://mail.python.org/pipermail/python-list/2002-December/134521.html

[147] Kent Pitman, ***Common List HyperSpec***, 2005, http://www.lispworks.com/documentation/common-lisp.html
[147a] 4.3.1 Introduction to Classes , [147b] `class` , [147c] metaclass

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

[149] Benjamin C. Pierce, ***Types and programming languages***, MIT press 2002,
[149a] search: metaclass

[150] Python Software Foundation, ***Python Documentation***, https://docs.python.org
[150a] Metaclass , [150b] PyObject_IsSubclass

[151] Christian Queinnec, ***Lisp in Small Pieces***, Cambridge University Press 2003,
[151a] `is-a?`

[152] 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,

[153] Eberhardt Rechtin, Mark W. Maier, ***The Art of Systems Architecting***, CRC Press 2010,
[153a] data-first fashion

[154] Quido van Rossum, Talin, ***PEP 3119 -- Introducing Abstract Base Classes***, 2007,
https://www.python.org/dev/peps/pep-3119/

[155] Quido van Rossum, ***Origin of metaclasses in Python***, http://python-history.blogspot.com/2013/10/origin-of-metaclasses-in-python.html

[156] Jean E. Rubin, ***Set theory for the mathematician***, Holden-Day, 1967,

[157] ***Ruby Talk***, http://blade.nagaokaut.ac.jp/ruby/ruby-talk/index.shtml
[157a] Article: Seeing Metaclasses Clearly [157b] Zeroton classes

[158] David M. Russinoff, ***Proteus: A frame-based nonmonotonic inference system***, Object-Oriented Concepts, Databases and Applications 1987,
[158a] *x* is a *y*

[159] SAS Institute, ***SAS Component Language 9.3: Reference***, SAS Institute 2011
[159a] emphasis is on the data , [159b] object identity , [159c] *is-a* relationship ,
[159d] the `Class` class is a metaclass

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

[161] Brian Henderson-Sellers, Cesar Gonzalez-Perez, ***The rationale of powertype-based metamodelling to underpin software development methodologies***, Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling-Volume 43 Australian Computer Society, Inc. 2005,
http://crpit.com/confpapers/CRPITV43HendersonSellers.pdf

[162] Brian Henderson-Sellers, ***On the mathematics of modelling, metamodelling, ontologies and modelling languages***, Springer Science & Business Media 2012,
[162a] isotypical mapping , [162b] powertype

[163] Brian Henderson-Sellers, Owen Eriksson, Cesar Gonzalez-Perez, Pär J. Ågerfalk, ***Ptolemaic Metamodelling? The Need for a Paradigm Shift***, Progressions and Innovations in Model-Driven Software Engineering IGI Global, 2013 2013,
[163a] inappropriately 'powertype' , [163b] Ptolemaic fixes

[164] Brian Henderson-Sellers, Tony Clark, Cesar Gonzalez-Perez, ***On the search for a level-agnostic modelling language***, Advanced Information Systems Engineering Springer Berlin Heidelberg 2013,
[164a] metaclasses are those classes that inherit from the class `Class`

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

[166] Pat Shaughnessy, ***Ruby Under a Microscope: Learning Ruby Internals Through Experiment***, No Starch Press 2013,

[167] Michele Simionato, ***The Python 2.3 Method Resolution Order***, Python Software Foundation 2003,
https://www.python.org/download/releases/2.3/mro/

[168] Michael A. Smith, ***Embedding an object calculus in the unifying theories of programming***, Diss. University of Oxford 2010,

[169] Guy L. Steele, ***Common LISP: the language***, Digital press 1990,
[169a] class is an object , [169b] `class-of` returns the class , [169c] predefined metaclasses

[170] Mark Stefik, Daniel Bobrow, ***Object-oriented programming: Themes and variations***, AI magazine 6.4 1985,
[170a] metaclass

[171] Kazimierz Subieta, Yahiko Kambayashi, Jacek Leszczylowski, Kazumasa Yokota, ***A Critique of Object Algebras***, 1995, http://www.ipipan.waw.pl/~subieta/artykuly/CritiqObjAlg.html

[172] Mark Summerfield, ***Programming in Python 3***, Pearson Education India, 2009,
[173] David Ungar, Randal B. Smith, ***Self: The power of simplicity***, Vol. 22. No. 12. ACM, 1987,

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

[175] Larry Wall, ***Synopsis 12: Objects***, Perl 6 Design Documents, 2004, http://design.perl6.org/S12.html
[175a] Metaclasses , [175b] Classes (2006) ,

[176] Paul S. Wang, ***Java with Object-Oriented Programming***, Thomson Brooks/Cole, 2002,

[177] Shane Warden, ***Modern Perl***, Onyx Neon Press, 2014,
[177a] The `isa()` method

[178] Peter Wegner, ***Dimensions of Object-Based Language Design***, ACM 1987,

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

[180] Richard S. Wiener, Lewis J. Pinson, ***An introduction to object-oriented programming and C++***,
Reading: Addison-Wesley, 1988,
[180a] [OOP] focuses on the data

[181] William A. Woods, James G. Schmolze, ***The KL-ONE Family***, Computers & Mathematics with Applications 23.2, 1992,

[182] World Wide Web Consortium, ***OWL 2 RDF-Based Semantics***, 2009, http://www.w3.org/TR/owl2-rdf-based-semantics/
[182a] Additional axiomatic triples for RDFS

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

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

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

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

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

[188] Tarek Ziadé, ***Expert Python Programming***, Packt Publishing Ltd, 2008,

---

[𝔾] ***Google***, https://google.com
[𝔾1] metaclass

[𝕀] ***Google Scholar***, https://scholar.google.com
[𝕀1] "python" "metaclass" , [𝕀2] "clos" "metaclass" , [𝕀3] metaclass (selected authors, 2011) ,
[𝕀4] meta[-]class[es] (selected authors, 2011) , [𝕀5] metaclass (selected authors, 2014) ,
[𝕀6] meta[-]class[es] (selected authors, 2014) , [𝕀7] meta[-]class[es] prior to 1980

[𝕊] ***StackExchange***, http://stackexchange.com, ***StackOverflow***, http://stackoverflow.com
OOP foundations resources: [𝕊1] by Andrej Bauer , [𝕊2] by Dave Clarke ,
[𝕊3] *untyped vs dynamically typed*

[𝕎] ***Wikipedia: The Free Encyclopedia***, http://wikipedia.org
[𝕎1] Formal definition of OOP (2004) , [𝕎2] (2007) , [𝕎3] Formal semantics of OOP (2015) ,
[𝕎4] OOP (2015) , [𝕎5] NOOP (added April 2013) , [𝕎6] (removed July 2013) ,
[𝕎7] Talk: References to NOOP , [𝕎8] Properties of an object (2013) ,
[𝕎9] Prototype-based programming (February 2006) ,
[𝕎10] Prototype-based programming (February 2016) , [𝕎11] Dynamic dispatch , [𝕎12] OOAD ,
[𝕎13] OODBMS , [𝕎14] Knowledge representation , [𝕎15] Metamodelling , [𝕎16] OMG , [𝕎17] UML ,
[𝕎18] MOF , [𝕎19] Powertype (UML) , [𝕎20] Multiple dispatch , [𝕎21] OCaml , [𝕎22] Go , [𝕎23] Java ,
[𝕎24] C# , [𝕎25] Python , [𝕎26] Ruby , [𝕎27] Swift Features , [𝕎28] Programming ,

[𝕎29] Dynamic programming language , [𝕎30] Metaclass (current version) ,
[𝕎31] Metaclass (as of February 2006) , [𝕎32] Metaclass (Semantic Web) ,
[𝕎33] Metaclass (Semantic Web) – April 2015 , [𝕎34] Talk:Metaclass#java.lang.Class ,
[𝕎35] Prototype-based programming , [𝕎36] Eigenclass model (January 2013),
[𝕎37] Functional graph , [𝕎38] Functional relation , [𝕎39] Lunate epsilon , [𝕎40] Peano ϵ ,
[𝕎41] Preorder , [𝕎42] Partition , [𝕎43] Transitive closure , [𝕎44] Vacuous truth ,
[𝕎45] Transitive reduction , [𝕎46] DAG , [𝕎47] Substructure , [𝕎48] Closure operator ,
[𝕎49] Upper set , [𝕎50] Abstract state machines , [𝕎51] Linear extension , [𝕎52] Topological sort ,
[𝕎53] Directed graph , [𝕎54] Currying , [𝕎55] Associative array , [𝕎56] Reification ,
[𝕎57] Lazy evaluation , [𝕎58] Literal , [𝕎59] Primary key , [𝕎60] Database view , [𝕎61] Upsert ,
[𝕎62] Common Lisp HyperSpec , [𝕎63] AMOP , [𝕎64] Common Lisp the Language ,
[𝕎65] Object-Oriented Software Construction , [𝕎66] Doug Lea , [𝕎67] John Vlissides ,
[𝕎68] Henderson-Sellers , [𝕎69] Post Office , [𝕎70] Indefinite article , [𝕎71] F-logic , [𝕎72] KL-ONE ,
[𝕎73] SUMO , [𝕎74] Ontology language , [𝕎75] Semantic Web , [𝕎76] Functional programming ,
[𝕎77] Typed lambda calculus , [𝕎78] Nominal type system , [𝕎79] Structural type system , [𝕎80] ~~Is-a~~ ,
[𝕎81] ~~Has-a~~ , [𝕎82] Object composition , [𝕎83] Aggregation , [𝕎84] Class diagram ,
[𝕎85] Morse–Kelley set theory , [𝕎86] Urelement , [𝕎87] Ruby MRI

License