Abstract

Object-oriented programming (OOP) is one of the most popular paradigms used for building software systems\(^1\). However, despite its industrial and academic popularity, OOP is still missing a formal apparatus similar to \(\lambda\)-calculus, which functional programming is based on. There were a number of attempts to formalize OOP, but none of them managed to cover all the features available in modern OO programming languages, such as C++ or Java. We have made yet another attempt and created EOLANG (also called EO), an experimental programming language based on \(\varphi\)-calculus.

Keywords: Object-Oriented Programming, Object Calculus

1 Introduction

It is difficult to define what exactly is OOP, as “the term has been used to mean different things,” according to Stefik and Bobrow [83]. Madsen and Møller-Pedersen [64] claimed that “there are as many definitions of OOP as there papers and books on the topic.” Armstrong [4] made a noticeable observation: “When reviewing the body of work on OO development, most authors simply suggest a set of concepts that characterize OO, and move on with their research or discussion. Thus, they are either taking for granted that the concepts are known or implicitly acknowledging that a universal set of concepts does not exist.”

1.1 Lack of Formal Model

The term OOP was coined by Kay [59] in 1966 and since then was never introduced formally [58]. Back in 1982, Rentsch [77] predicted: “Everyone will be in a favor of OOP. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is.”

There is a fundamental problem in OOP—the lack of a rigorous formal model, as was recapped by Eden and Hirshfeld [35]: “Unfortunately, architectural formalisms have largely ignored the OO idiosyncrasies. Few works recognized the elementary building blocks of design and architecture patterns. As a result of this oversight, any attempt to use formalisms for the specification of OO architectures is destined to neglect key regularities in their organization.”

There is no uniformity or an agreement on the set of features and mechanisms that belong in an OO language as “the paradigm itself is far too general,” as was concluded by Nierstrasz [72] in his survey.

OO and semi-OO programming languages treat OOP differently and have variety of different features to follow the concept of object-orientedness. For example, Java has classes and types (interfaces) but doesn’t have multiple inheritance [3], C++ has multiple inheritance but doesn’t directly support mixins [19], Ruby and PHP don’t have generics and types, but have traits [11], JavaScript doesn’t have classes, but has prototypes [78], and so on.

It was noted by Danforth and Tomlinson [27] that object-oriented programming, like functional programming or logic programming, incorporates a metaphor in which computation is viewed in terms divorced from the details of actual computation. However, in the case of OOP, this metaphor is rarely introduced with the mathematical precision available to the functional or logic programming models. Rather, OOP is generally expressed in philosophical terms, resulting in a natural proliferation of opinions concerning exactly what OOP really is.

1.2 Complaints of Programmers

Although the history of OOP goes back for more than 50 years to the development of Simula [26], OOP is under heavy criticism since the beginning to nowadays, mostly for its inability to solve the problem of software complexity.

According to Graham [43], “somehow the idea of reusability got attached to OOP in the 1980s, and no amount of evidence to the contrary seems to be able to shake it free,” while “OOP offers a sustainable way to write spaghetti code.” West [87] argues that the contemporary mainstream understanding of objects (which is not behavioral) is “but a pale shadow of the original idea” and “anti-ethical to the original intent.” Gosling and McGilton [42] notes that “unfortunately, ‘object oriented’ remains misunderstood and over-marketed as the silver bullet that will solve all our software ills.”
1.3 High Complexity

Nierstrasz [74] said that "OOP is about taming complexity through modeling, but we have not mastered this yet." Readability and complexity issues of OO code remain unsolved till today. Shelly [82] claimed that "Reading an OO code you can’t see the big picture and it is often impossible to review all the small functions that call the one function that you modified." Khanam [60] in a like manner affirmed: "Object oriented programming promotes ease in designing reusable software but the long coded methods makes it unreadable and enhances the complexity of the methods."

The complexity of OO software is higher than the industry would expect, taking into account the amount of efforts invested into the development of OO languages. As was concluded by Bosch [16], "OO frameworks have number of problems that complicate development, usage, composition and maintenance of software."

For example, the infamous legacy code has its additional overhead associated with OO languages—inheritance mechanism, which "allows you to write less code at the cost of less readability," as explained by Carter [20]. It is not infrequently when "inheritance is overused and misused," which leads to "increased complexity of the code and its maintenance," as noted by Bernstein [8].

The lack of formalism encouraged OOP language creators to invent and implement language features, often known as "syntax sugar," which are convenient for some of them in some special cases but jeopardize the consistency of design when being used too often and by less mature programmers. The most obvious outcome of design inconsistencies is high complexity due to low readability, which negatively affects the quality and leads to functionality defects.

1.4 Solution Proposed

EO\(^2\) was created in order to eliminate the problem of complexity of OOP code, providing 1) a formal object calculus and 2) a programming language with a reduced set of features. The proposed $\psi$-calculus represents an object model through data and objects, while operations with them are possible through formation, application, and decoration. The calculus introduces a formal apparatus for manipulations with objects.

EO, the proposed programming language, fully implements all elements of the calculus and enables implementation of an object model on any computational platform. Being an OO programming language, EO enables four key principles of OOP: abstraction, inheritance, polymorphism, and encapsulation.

The following four principles stay behind the apparatus we introduce:

- An object is a collection of attributes, which are uniquely named bindings to objects. An object is an atom if its implementation is provided by the runtime.
- An object is abstract if at least one of its attributes is void—isn’t attached to any object. An object is closed otherwise. Formation is the process of creating a objects. Application is the process of making a copy of existing object, specifying some or all of its void attributes with objects known as arguments. Application may lead to the creation of a closed object, or an abstract one, if not all void attributes are specified with arguments.
- An object may decorate another object by binding it to the $\varphi$ attribute of itself. A decorator has its own attributes and attached attributes of its decoratee.
- A special attribute $\Delta$ may be attached to data, which is a computation platform dependable entity not decomposable any further. Dataization is a process of retrieving data from an object, by taking what the $\Delta$ attribute is attached to. The dataization of an object at the highest level of composition leads to the execution of a program.

The rest of the paper is dedicated to the discussion of the syntax of the language that we created based on the calculus, the calculus itself, its semantics, and pragmatics. In order to make it easier to understand, we start the discussion with the syntax of the language, while the calculus is derived from it. Then, we discuss the key features of EO and the differences between it and other programming languages. We also discuss how the absence of traditional OOP features, such as mutability or inheritance, affect the complexity of code. At the end of the paper we overview the work done by others in the area of formalization of OOP.

2 Syntax

The entire syntax of EO language in BNF is available on the first page of the objectionary/\(eo\) Github repository\(^3\). Similar to Python [63], indentation in EO is part of the syntax: the scope of a code block is determined by its horizontal position in relation to other blocks, which is also known as "off-side rule" [61].

There are no keywords in EO but only a few special symbols denoting grammar constructs: $>$ for the attributes naming, . for the dot notation, [ ] for the specification of parameters of formations, ( ) for scope limitations, ! for turning objects into constants, : for naming arguments, " " (double quotes) for string literals, $@$ for the decoratee, ’ (apostroph) for explicit copying, $<$ for object identity, $^\_\_$ for referring to the parent object, & for referring to the home object, and $\$ for referring to the current object. Attributes, which are the only identifiers that exist in EO, may have any Unicode symbols in their names, as long as they start with a small English

\(^3\)https://github.com/objectionary/\(eo\)
letter and don’t contain spaces, line breaks, or special symbols mentioned above: test-File and 文件 are valid identifiers. Identifiers are case-sensitive: car and Car are two different identifiers. Java-notation is used for numbers and strings.

2.1 Identity, State, and Behavior

According to Booch et al. [15], an object in OOP has state, behavior, and identity: “The state of an object encompasses all of the properties of the object plus the current values of each of these properties. Behavior is how an object acts and reacts, in terms of its state changes and message passing. Identity is that property of an object which distinguishes it from all other objects.” The syntax of EO makes a difference between these three categories.

This is a formation of a new object book, which has a single identity attribute isbn:

| isbn > book |

To make another object with a specific ISBN, the `book` has to be copied, with the data as an argument:

```
book "978-1519166913" > b1
```

Here, b1 is a new object created. Its only attribute is accessible as b1.isbn.

A similar abstract object, but with two new state attributes, would look like:

```
isbn > book2
"Object Thinking" > title
memory 0 > price
```

The attribute title is a constant, while the price represents a mutable chunk of bytes in computing memory. They both are accessible similar to the isbn, via book2.title and book2.price. It is legal to access them in the abstract object, since they are attached to objects. However, accessing book2.isbn will lead to an error, since the attribute isbn is void in the abstract object book2.

A behavior may be added to an object with a new inner abstract object set-price:

```
isbn > book3
"Object Thinking" > title
memory 0 > price
[p] > set-price
~.price.write p > 0
```

The price of the book may be changed with this one-liner:

```
book3.set-price 19.99
```

2.2 Indentation

This is an example of an abstract object vector, where spaces are replaced with the "_" symbol in order to demonstrate the importance of their presence in specific quantity (for example, there has to be exactly one space after the closing square bracket at the second line and the > symbol, while two spaces will break the syntax):

```
_This_is_a_vector_in_2D_space_
[dx_dy] > _vector_
_dx_.sqrt > _length_
_____plus_
_disciplinary_dx.times_dx
_disciplinary_dy.timex_dy
```

The code at the line no. 12 is a comment. Two void attributes dx and dy are listed in square brackets at the line no. 13. The name of the object goes after the > symbol. The code at the line no. 14 defines an attached attribute length. Anywhere when an object has to get a name, the > symbol can be added after the object.

The declaration of the attribute length at the lines 14–17 can be written in one line, using dot notation:

```
((dx.times dx).plus (dy.timex dy)).sqrt > length
```

An inverse dot notation is used in order to simplify the syntax. The identifier that goes after the dot is written first, the dot follows, and the next line contains the part that is supposed to stay before the dot. It is also possible to rewrite this expression in multiple lines without the usage of inverse notation, but it will look less readable:

```
dx.times dx
   .plus
   dy.timex dy
   .sqrt > length
```

Here, the line no. 19 is the application of the object `dx.times` with a new argument `dx`. Then, the next line is the object `plus` taken from the object created at the first line, using the dot notation. Then, the line no. 21 is the argument passed to the object `plus`. The code at the line no. 22 takes the object `sqrt` from the object constructed at the previous line, and gives it the name `length`.

Indentation is used for two purposes: either to define attributes of an abstract object or to specify arguments for object application, also known as making a copy. A definition of an abstract object starts with a list of void attributes in square brackets on one line, followed by a list of attached attributes each in its own line. For example, this is an abstract anonymous object (it doesn’t have a name) with one void attribute x and two attached attributes succ and prev:

```
x]
x.plus 1 > succ
x.minus 1 > prev
```

The arguments of `plus` and `minus` are provided in a horizontal mode, without the use of indentation. It is possible to rewrite this code in a vertical mode, where indentation will be required:
This abstract object can also be written in a horizontal mode, because it is anonymous:

\[ [x] (x .plus 1 > succ) (x .minus 1 > prev) \]

### 2.3 EO to XML

Due to the nesting nature of EO, its program can be transformed to an XML document. The abstract object `vector` would produce this XML tree of elements and attributes:

```
<o name="vector">
  <o name="dx"/>
  <o name="dy"/>
  <o name="length" base=".sqrt">
    <o base=".plus">
      <o base=".times">
        <o base="dx"/>
      </o>
      <o base="dy"/>
    </o>
  </o>
</o>
```

Each object is represented by an `<o>` XML element with a few optional attributes, such as `name` and `base`. Each attribute is either a named reference to an object (if the attribute is attached, such as `length`), or a name without a reference (if it is void, such as `dx` and `dy`).

### 2.4 Data Objects and Tuples

There are a few abstract objects which can’t be directly copied, such as `float` and `int`. They are created by the compiler when it meets a special syntax for data, for example:

\[ [r] > circle \\
  r .times 3 .14 > circumference \]

This syntax would be translated to XMIR (XML based data format):

```
<o name="circle">
  <o name="r"/>
  <o base=".times" name="circumference">
    <o base="float">
      <o base="bytes">40-09-1E-B8-51-EB-85-1F</o>
    </o>
  </o>
</o>
```

### 2.5 Scope Brackets

Brackets can be used to group object arguments in horizontal mode:

```
sum (div 45 5) 10
```

The `(div 45 5)` is a copy of the abstract object `div` with two arguments `45` and `5`. This object is itself the first argument of the copy of the object `sum`. Its second argument is `10`. Without brackets the syntax would read differently:

```
sum div 45 5 10
```

This expression denotes a copy of `sum` with four arguments.

### 2.6 Inner Objects

An abstract object may have other abstract objects as its attributes, for example:

```
# A point on a 2D canvas
[x y] > point \\
[to] > distance
```

| Data    | Example                          | Size |
|---------|----------------------------------|------|
| bytes   | 1F-E5-77-A6                     | 4    |
| string  | "Hello, друг!"                   | 16   |
|         | "\u5BB6" or "й"                 | 2    |
| int     | 1024, 0x1A7E, or -42            | 8    |
| float   | 3.1415926 or 2.4e-34            | 8    |
| bool    | TRUE or FALSE                   | 1    |

Table 1. The syntax of all data with examples. The “Size” column denotes the number of bytes in the `as-bytes` attribute. UTF-8 is the encoding used in `string` object.
EOLANG and $\phi$-calculus

69 length. > len
vector
to x minus (^ . x)
to y minus (^ . y)

The object point has two void attributes \( x \) and \( y \) and the attribute distance, which is attached to an abstract object with one void attribute to and one attached attribute len. The inner abstract object distance may only be copied with a reference to its parent object point, via a special attribute denoted by the \(^ \) symbol:

distance.
point
  5 : x
  -3 : y
point: to
  13 : x
  3.9 : y

The parent object is (point 5 -3), while the object (point 13 3.9) is the argument for the void attribute to of the object distance. Suffixes \( : x \), \( : y \), and \( : to \) are optional and may be used to denote the exact name of the void attribute to be attached to the provided argument.

2.7 Decorators

An object may extend another object by decorating it:

80 [center radius] > circle
center > @
[p] > is-inside
lte. > @
  ^. @. distance $. p
  ^. radius

The object circle has a special attribute @ at the line no. 81, which denotes the decoratee: an object to be extended, also referred to as “component” by Gamma et al. [37].

The decorator circle has the same attributes as its decoratee center, but also its own attribute is-inside. The attribute @ may be used the same way as other attributes, including in dot notation, as it is done at the line no. 84. However, this line may be re-written in a more compact way, omitting the explicit reference to the @ attribute, because all attributes of the center are present in the circle; and omitting the reference to $\$ because the default scope of visibility of \( p \) is the object is-inside:

85 ^. distance \( p \)

The inner object is-inside also has the @ attribute: it decorates the object lte (stands for “less than equal”). The expression at the line no. 84 means: take the parent object of is-inside, take the attribute @ from it, then take the inner object distance from there, and then make a copy of it with the attribute \( p \) taken from the current object (denoted by the $\$ symbol).

2.8 Anonymous Formations

A formation may be used as an argument of another object while making a copy of it, for example:

93 (dir "/tmp"). walk
  * ([f] (f.is-dir > @))

An anonymous formation may have multiple attributes:

95 [x] (x.plus 1 > succ) (x.minus 1 > prev)

This object has two attributes succ and prev, and doesn’t have a name.

The parent of each copy of the abstract object will be set by the object walk and will point to the walk object itself.

2.9 Constants

EO is a declarative language with lazy evaluations. This means that this code would read the input stream two times:

96 [] > hello
stdout > say
sprintf
  "The length of %s is %d"
stdin.next-line > x!
x.length

The object sprintf may be used like this, to understand whether the \((0, 0)\) point is inside the circle at \((-3, 9)\) with the radius 40:

circle (point -3 9) 40 > c
  c.is-inside (point 0 0) > i

Here, \( i \) will be a copy of bool behaving like TRUE because lte decorates bool.

It is also possible to make decoratee void, similar to other void attributes, specifying it in the list of void attributes in square brackets.
the name \( x \) a constant. This means that all attributes of \( x \) are cached. Important to notice that the cache is not deep: the attributes of attributes are not cached.

Here, \( x \) is an attribute of the object hello, even though it is not defined as explicitly as `say`. Anywhere a new name shows up after the `>` symbol, it is a declaration of a new attribute in the nearest object abstraction.

### 2.10 Explicit Shallow Copies

There may be a need to make a copy of an object without giving any parameters to it. This may be done with an apostrophe suffix:

\[
\text{p} 3 5 > \text{p1}
\]

Here, two objects will be created, \( p \) and \( p1 \), where the former is an abstract one, a copy of copy, while the later is a closed one with two parameters specified. The apostrophe suffix may be used anywhere after the name of an object, for example:

\[
\text{point}' > \text{p}
\]

Making a copy of `circle` will not lead to making a copy of `point`, which is encapsulated by `circle`.

### 2.11 Object Identity

Each object has a special attribute `<`, which is an integer referring to a unique identifier of an object in the entire runtime scope of a program. All of the following expressions are true:

\[
\text{TRUE.<}.eq (\text{TRUE.<})
\]

\[
42.<.eq (42.<)
\]

\[
\text{point.<}.eq (\text{point.<})
\]

All of the following expressions are false:

\[
42.<.eq (7.<)
\]

\[
(2.+ 2).<.eq (4.<)
\]

\[
(\text{point 3 5}).<.eq ((\text{point 3 5}).<)
\]

\[
(* 1 2).<.eq ((* 1 2).<)
\]

### 2.12 Metas and License

A program may have a comment at the beginning of the file, which is called a license. The license may be followed by an optional list of meta statements, which are passed to the compiler as is. The meaning of them depends on the compiler and may vary between target platforms. This program instructs the compiler to put all objects from the file into the package `org.example` and helps it resolve the name `stdout`, which is external to the file:

\[
+package org.example
+alias org.eolang.io.stdout
\]

\[
[args] > \text{app}
\]

\[
\text{stdout} > @
\]

\[
"Hello, world!\n"
\]

### 2.13 Atoms

Some objects in EO programs may need to be platform specific and can’t be composed from other existing objects—they are called atoms. For example, the object `app` uses the object `stdout`, which is an atom. Its implementation would be provided by the runtime. This is how the object may be defined:

\[
+rt jvm org.eolang:eo-runtime:0.7.0
+rt ruby eolang:0.1.0
\]

\[
[text] > \text{stdout} /\text{bool}
\]

The `/bool` suffix informs the compiler that this object must not be compiled from EO to the target language. The object with this suffix already exists in the target language and most probably could be found in the library specified by the `rt` meta. The exact library to import has to be selected by the compiler. In the example above, there are two libraries specified: for JVM and for Ruby.

The `bool` part after the `/` is the name of object, which `stdout` decorates. The name may be replaced by a question mark, if uncertain about the object being decorated.

Atoms in EO are similar to “native” methods in Java and “extern” methods in C#: this mechanism is also known as foreign function interface (FFI).

### 2.14 Home Object

An instance of an abstact object may need to have access to the object where the abstract was defined, for example this is how object `tuple.map` is implemented in Objectionary:

\[
[] > \text{list}
\]

\[
[f] > \text{mapi} /\text{list}
\]

\[
[f] > \text{map}
\]

\[
.\text{mapi} > @
\]

\[
[x i]
\]

\[
&.f x > @
\]

The object `mapi` at the line no. 128 is an atom: it iterates through list of items and makes a copy of the provided two-arguments abstract object `f` applying the next item to it and the index in the tuple (that is why the name with the “i” suffix).

The object `map` does exactly the same, but doesn’t provide the index of each element to the inner abstract object. The anonymous inner abstract object at the line no. 131 has to get access to the attribute `f` of `map`. However, `.f` won’t
work, because the parent of it is a copy of map, and the 
parent of map is the object list. Thus, there is no way 
to get access to map using parent attributes.

The home attribute & helps here. Once an abstract object 
at the line no. 131 is created, its home attribute is set to the 
abstract object list at the line no. 127. Its parent attribute 
^ is also set to the object list, but is later changed by the 
atom map when a copy of it is being made. However, the 
home attribute remains the same.

3 Calculus

The proposed -calculus is based on set theory and 
lambda calculus, representing objects as sets of pairs and 
their internals as -terms. The rest of the section contains 
formal definitions of data, objects, attributes, formation, ap-
lication, decoration, and dataization.

3.1 Objects and Data

Definition 3.1. An object is a set of ordered pairs (a, v) 
such that a is an identifier, all a are different, and v is an 
object.

An identifier is either , , , or, by convention, a text 
without spaces starting with a small-case English letter in 
typewriter font.

The object at the line no. 1 may be represented as 
book = ((isbn, )) ,
where isbn is an identifier and is an empty set, which is 
a proper object, according to Definition 3.1.

Definition 3.2. An object may have properties of data, 
which is a computation platform dependable entity and is 
not decomposable any further within the scope of -calculus.

What exactly is data may depend on the implementation 
platform, but most certainly would include byte arrays, in-
tegers, floating-point numbers, string literals, and boolean 
values.

The object at the lines 3–5 may be represented as 
book2 = ((isbn, title, "Object Thinking"), (price, memory),
where isbn, title, and price are identifiers, memory 
is an object defined somewhere else, and the text in double 
quotes is data.

3.2 Attributes

Definition 3.3. In an object x, a is a void attribute with 
the name a iff (a, ) ∈ x; it is an attached attribute with the 
value v iff ∈ x and v ≠ ;

In Eq. (2), identifiers isbn, title, and price are the 
attributes of the object book2. The attribute isbn is void, 
while the other two are attached.

Definition 3.4. If x is an object and ∈ x, then v may 
be referenced as x.a; this referencing mechanism is called 
dot notation.

Both void and attached attributes of an object are accessi-
ble using the dot notation. There is no such thing as visibility 
restriction in -calculus: all attributes are visible to all objects 
outside of the one they belong to.

It is possible to chain attribute references using dot no-
tation, for example book2.price ± is a valid expression, 
which means “taking the attribute price from the object 
book2, and then taking the attribute neg from it.”

Definition 3.5. If (a, ) is an object, then x, a set consisting 
of all a, is its scope and the cardinality of |x| is the arity 
of x.

For example, the scope of the object at Eq. (2) consists of 
three identifiers: isbn, title, and price.

3.3 Formation

Definition 3.6. An object x is abstract iff at least one of its 
attributes is void, i.e. .

An alternative “arrow notation” may be used to denote an 
object x in a more compact way, where void attributes stay 
in the parentheses on the left side of the mapping symbol → 
and pairs, which represent attached attributes, stay on the 
right side, in double-square brackets. Equation (2) may be 
written as

book2(isbn) → [[
title → "Object Thinking", 
price → memory
]]

3.4 Application

Definition 3.7. If x is an abstract object and y is an object 
where y ⊆ x, then an application of y to x is a copy of x, a 
new object that consists of pairs (a ∈ x, v) such that v = y.a 
if x.a = and v = x.a otherwise.

Application makes some void attributes of x attached—by 
binding objects to them. The produced object has exactly the 
same set of attributes, but some of them, which were void 
before, become attached.

It is not expected that all void attributes turn into attached 
ones during application. Some of them may remain void, 
which will lead to creating a new abstract object. To the 
contrary, if all void attributes are substituted with arguments 
during copying, a newly created object will be closed.

Once set, attached attributes may not be reset. This may 
be interpreted as immutability property of objects.

Arrow notation may also be used to denote object copying, 
where the names of the attributes, which remain void, stay 
in the brackets on the left side of the mapping symbol →,
in the brackets. For example, the object at the line no. 73 may be written as
\[
\text{point}(x \mapsto 5, y \mapsto -3).\text{distance}(t \mapsto \text{point}(x \mapsto 13, y \mapsto 3.9)),
\]
and may further be simplified since the order of parameters is obvious:
\[
\text{point}(5, -3).\text{distance}(\text{point}(13, 3.9)).
\]
An application without arguments is a copy of an object. For example, in these expressions the attribute \( p_1 \) is attached to the same object as the attribute \( p \), while the attribute \( p_2 \) is attached to a new object, a copy of \( p \):
\[
\begin{align*}
 p & \mapsto \text{point}(5, -3), \\
 p_1 & \mapsto p, \\
 p_2 & \mapsto p().
\end{align*}
\]

### 3.5 Formation

**Definition 3.8.** The process of creating an object that is not a copy of another object is called **formation**.

Syntactically, object formation is denoted by double square brackets, as in Eq. (3), to the contrary of object application, which is denoted by round brackets, as in Eq. (4). Object abstraction is a special case of object formation. The following expression is a formation of the object \( x \):
\[
x \mapsto [y \mapsto [z \mapsto t]].
\]

### 3.6 Parent and Home

**Definition 3.9.** If \( x \) is an object, then \( x.p \) is its **parent**, which is the object that created \( x \).

An object may be created either by abstraction or application. In case of abstraction, an object is created by another abstract object, for example:
\[
\begin{align*}
 x & \mapsto [y \mapsto [z \mapsto t]] \\
 x.y.p & = x \\
 x.y.z.p & = y.
\end{align*}
\]
In case of application, an object is created by the prepending object:
\[
\begin{align*}
 a & \mapsto [b \mapsto c, d \mapsto e, f \mapsto h.i(j)] \\
 a.b.p & = a \\
 a.d.p & = e \\
 a.g.p & = h.
\end{align*}
\]
**Definition 3.10.** If \( x \) is an object, then \( x.a \) is its **home** object, which is the \( \rho \) of the abstract object \( x \) is a copy of.

For example:
\[
\begin{align*}
 x & \mapsto [y(f) \mapsto []] \\
 x.y.p & = x \\
 z & \mapsto [t \mapsto x.y(42)] \\
 z.t.p & = z \\
 z.t.a & = x.
\end{align*}
\]

### 3.7 Decoration

**Definition 3.11.** If \( x \) and \( y \) are objects and \( x.\varphi = y \), then \( \forall a(x.a = y.a) \) if \( a \notin \hat{x} \); this means that \( x \) is **decorating** \( y \).

Here, \( \varphi \) is a special identifier denoting the object, known as a **decoratee**, being decorated within the scope of the decorator.

For example, the object at the lines 80–85 would be denoted by this formula:
\[
\begin{align*}
 \text{circle}(\text{center}, \text{radius}) \mapsto & [] \\
 \varphi & \mapsto \text{center}, \\
 \text{is-inside}(p) & \mapsto [ \\
 \varphi & \mapsto \rho.\varphi.\text{distance}(p).\text{lte}(\text{radius})]
\end{align*}
\]
while the application of it would look like:
\[
c \mapsto \text{circle}(\text{point}(-3, 40), \text{radius}).
\]
producing:
\[
\begin{align*}
 c & \mapsto [] \\
 \text{center} & \mapsto \text{point}(-3, 40), \\
 \text{radius} & \mapsto 40, \\
 \varphi & \mapsto \text{center}, \\
 \text{is-inside}(p) & \mapsto [ \\
 \varphi & \mapsto \rho.\varphi.\text{distance}(p).\text{lte}(\text{radius})]
\end{align*}
\]
Because of decoration, the expression \( \rho.\varphi.\text{distance} \) in Eq. (9) is semantically equivalent to a shorter expression \( \rho.\text{distance} \) in Eq. (11).

The following expression makes a new object \( \hat{x} \), which represents a sequence of object applications ending with a copy of \( \text{lte} \):
\[
\text{is} \mapsto c.\text{is-inside}(\text{point}(1, 7)),
\]
producing:
\[
\begin{align*}
 c & \mapsto [] \\
 \text{center} & \mapsto \text{point}(-3, 40), \\
 \text{radius} & \mapsto 40, \\
 \varphi & \mapsto \text{center}, \\
 \text{is-inside} & \mapsto [ \\
 p & \mapsto \text{point}(1, 7), \\
 \varphi & \mapsto \rho.\text{distance}(p).\text{lte}(\text{radius})]
\end{align*}
\]
It is important to notice that attributes of a decoratee don’t belong to the scope of its decorator.

### 3.8 Atoms

**Definition 3.12.** If \( \lambda s.M \) is a function of one argument \( s \) returning an object, then it is an abstract object called an **atom**, \( M \) is its \( \lambda \)-term, and \( s \) is its void attribute.
Atoms may have their $\lambda$-terms defined outside of $\varphi$-calculus formal scope. For example, the object at the line no. 126 would be denoted as

$$\text{stdout}(\text{text}) \mapsto \lambda s. M_{\text{stdout}},$$

(14)

where $M_{\text{stdout}}$ is a $\lambda$-term defined externally.

In atoms, $\lambda$-terms are attached to $\lambda$ attribute. Thus, a more formal form of the Eq. (14) is:

$$\text{stdout} \mapsto [\text{text} \mapsto \emptyset, \lambda \mapsto M_{\text{stdout}}].$$

3.9 Constant

Definition 3.13. If $x \mapsto [y \mapsto z]$ is an object then $y$ is a constant attribute, meaning that the result of dataization of $x,y$ always equals to itself.

3.10 Identity

Definition 3.14. If $x$ is an object then $x.y$ is a positive integer data object with a unique identity of $x$ in the entire runtime scope.

For example, without any other objects in scope it is safe to assume the following, however there is no guarantee that the actual numbers will be the same in all implementations:

$$x(y) \mapsto []$$

$$z \mapsto x(y \mapsto 42)$$

$$\Phi, v = 0, x.v = 1, 42, v = 2, z.v = 3.$$  

(15)

4 Key Features

There are a few features that distinguish EO and $\varphi$-calculus from other existing OO languages and object theories, while some of them are similar to what other languages have to offer. The Section is not intended to present the features formally, which was done earlier in Sections 2 and 3, but to compare EO with other programming languages and informally identify similarities.

No Classes. EO is similar to other delegation-based languages like Self [85], where objects are not created by a class as in class-based languages like C++ or Java, but from another object, inheriting properties from the original. However, while in such languages, according to Fisher and Mitchell [36], “an object may be created, and then have new methods added or existing methods redefined,” in EO such object alteration is not allowed.

No Types. Even though there are no types in EO, compatibility between objects may be inferred in compile-time and validated strictly, which other typeless languages such as Python, Julia [10], Lua [50], or Erlang [1] can’t guarantee. Also, there is no type casting or reflection on types in EO.

No Inheritance. It is impossible to inherit attributes from another object in EO. The only two possible ways to re-use functionality are either via object composition or decorators. There are OO languages without implementation inheritance, for example Go [30], but only Kotlin [56] has decorators as a language feature. In all other languages, the Decorator pattern [37] has to be implemented manually [9].

No Methods. An object in EO is a composition of other objects and atoms: there are no methods or functions similar to Java or C++ ones. Execution control is given to a program when atoms’ attributes are referred to. Atoms are implemented by EO runtime similar to Java native objects. To our knowledge, there are no other OO languages without methods.

No Constructors. Unlike Java or C++, EO doesn’t allow programmers to alter the process of object construction or suggest alternative paths of object instantiation via additional constructions. Instead, all arguments are attached to attributes “as is” and can’t be modified.

No Static Entities. Unlike Java and C#, EO objects may consist only of other objects, represented by attributes, while class methods, also known as static methods, as well as static literals, and static blocks—don’t exist in EO. Considering modern programming languages, Go has no static methods either, but only objects and “structs” [80].

No Primitive Data Types. There are no primitive data types in EO, which exist in Java and C++, for example. As in Ruby, Smalltalk [39], Squeak, Self, and Pharo, integers, floating point numbers, boolean values, and strings are objects in EO: “everything is an object” is the key design principle, which, according to West [87, p.66], is an “obviously necessary prerequisite to object thinking.”

No Operators. There are no operators like + or / in EO. Instead, numeric objects have built-in atoms that represent math operations. The same is true for all other manipulations with objects: they are provided only by their encapsulated objects, not by external language constructs, as in Java or C#.

Here EO is similar to Ruby, Smalltalk and Eiffel, where operators are syntax sugar, while implementation is encapsulated in the objects.

No NULL References. Unlike C++ and Java, there is no concept of NULL in EO, which was called a “billion dollar mistake” by Hoare [44] and is one of the key threats for design consistency [18]. Haskell, Rust, OCaml, Standard ML, and Swift also don’t have NULL references.

No Empty Objects. Unlike Java, C++ and all other OO languages, empty objects with no attributes are forbidden in EO in order to guarantee the presence of object composition and enable separation of concerns [29]: larger objects must always encapsulate smaller ones.

No Private Attributes. Similar to Python [63] and Smalltalk [48], EO makes all object attributes publicly visible. There are no protected ones, because there is no implementation inheritance, which is considered harmful [49]. There are no private attributes either, because information hiding can anyway easily be violated via getters, and usually is, making the code longer and less readable, as explained by Holub [46].
No Global Scope. All objects in EO are attached to some attributes. Objects constructed in the global scope of visibility are attached to attributes of the $\Phi$ object of the highest level of abstraction. Newspeak and Eiffel are two programming languages that does not have global scope as well.

No Mutability. Similar to Erlang [5], there are only immutable objects in EO, meaning that their attributes may not be changed after the object is constructed or copied. Java, C#, and C++, have modifiers like `final`, `readonly`, or `const` to make attributes immutable, which don’t mean constants though. While the latter will always expose the same functionality, the former may represent mutable entities, being known as read-only references [12]. For example, an attribute $r$ may have an object `random.pseudo` attached to it, which is a random number generator. EO won’t allow assigning another object to the attribute $r$. However, every time the attribute is dataized, its value will be different. There are number of OOP languages that also prioritize immutability of objects. In Rust [67], for example, all variables are immutable by default, but can be made mutable via the `mut` modifier. Similarly, D [17] has qualifier `immutable`, which expresses transitive immutability of data.

No Exceptions. In most OO languages exception handling [40]: happens through an imperative error-throwing statement. Instead, EO has a declarative mechanism for it, which is similar to Null Object design pattern [66]: returning an abstract object causes program execution to stop once the returned object is dealt with.

No Functions. There are no lambda objects or functions in EO, which exist in Java 8+, for example. However, objects in EO have “bodies,” which make it possible to interpret objects as functions. Strictly speaking, if objects in EO would only have bodies and no other attributes, they would be functions. It is legit to say that EO extends lambda calculus, but in a different way comparing to previous attempts made by Mitchell et al. [71] and Di Gianantonio et al. [28]: methods and attributes in EO are not new concepts, but lower-level objects.

No mixins. There are no “traits” or “mixins” in EO, which exist in Ruby and PHP to enable code reuse from other objects without inheritance and composition.

5 Four Principles of OOP

In order to answer the question, whether the proposed object calculus is sufficient to express any object model, in this section we demonstrate how four fundamental principles of OOP are realized by $\varphi$-calculus: encapsulation, abstraction, inheritance, and polymorphism.

5.1 Abstraction

Abstraction, which is called “modularity” by Booch et al. [15], is, according to West [87, p.203], “the act of separating characteristics into the relevant and irrelevant to facilitate focusing on the relevant without distraction or undue complexity.” While Stroustrup [84] suggests C++ classes as instruments of abstraction, the ultimate goal of abstraction is decomposition, according to West [87, p.73]: “composition is accomplished by applying abstraction—the ‘knife’ used to carve our domain into discrete objects.”

In $\varphi$-calculus objects are the elements the problem domain is decomposed into. This goes along the claim of West [87, p.24]: “objects, as abstractions of entities in the real world, represent a particularly dense and cohesive clustering of information.”

5.2 Inheritance

Inheritance, according to Booch et al. [15], is “a relationship among classes wherein one class shares the structure and/or behavior defined in one (single inheritance) or more (multiple inheritance) other classes,” where “a subclass typically augments or restricts the existing structure and behavior of its superclasses.” The purpose of inheritance, according to Meyer [69], is “to control the resulting potential complexity” of the design by enabling code reuse.

Consider a classic case of behaviour extension, suggested by Stroustrup [84, p.38] to illustrate inheritance. C++ class `Shape` represents a graphic object on the canvas (a simplified version of the original code):

```cpp
class Shape {
  Point center;
  public:
    void move(Point to) { center = to; draw(); }
    virtual void draw() = 0;
};

The method `draw()` is “virtual,” meaning that it is not implemented in the class `Shape` but may be implemented in sub-classes, for example in the class `Circle`:

```cpp
class Circle : public Shape {
  int radius;
  public:
    void draw() { /* To draw a circle */ }
};
```

The class `Circle` inherits the behavior of the class `Shape` and extends it with its own feature in the method `draw()`. Now, when the method `Circle.move()` is called, its implementation from the class `Shape` will call the virtual method `Shape.draw()`, and the call will be dispatched to the overridden method `Circle.draw()` through the “virtual table” in the class `Shape`. The creator of the class `Shape` is now aware of sub-classes which may be created long after, for example `Triangle`, `Rectangle`, and so on.

Even though implementation inheritance and method overriding seem to be powerful mechanisms, they have been criticized. According to Holub [45], the main problem with implementation inheritance is that it introduces unnecessary
coupling in the form of the “fragile base class problem,” as was also formally demonstrated by Mikhajlov and Sekerinski [70].

The fragile base class problem is one of the reasons why there is no implementation inheritance in \( \varphi \)-calculus. Nevertheless, object hierarchies to enable code reuse in \( \varphi \)-calculus may be created using decorators. This mechanism is also known as "delegation" and, according to Booch et al. [15, p.98], is "an alternate approach to inheritance, in which objects delegate their behavior to related objects." As noted by West [87, p.139], delegation is "a way to extend or restrict the behavior of objects by composition rather than by inheritance." Seiter et al. [81] said that "inheritance breaks encapsulation" and suggested that delegation, which they called "dynamic inheritance," is a better way to add behavior to an object, but not to override existing behavior.

The absence of inheritance mechanism in \( \varphi \)-calculus doesn’t make it any weaker, since object hierarchies are available. Booch et al. [15] while naming four fundamental elements of object model mentioned “abstraction, encapsulation, modularity, and hierarchy” (instead of inheritance, like some other authors).

5.3 Polymorphism

According to Meyer [69, p.467], polymorphism means "the ability to take several forms," specifically a variable "at run time having the ability to become attached to objects of different types, all controlled by the static declaration." Booch et al. [15, p.67] explains polymorphism as an ability of a single name (such as a variable declaration) "to denote objects of many different classes that are related by some common superclass," and calls it "the most powerful feature of object-oriented programming languages."

Consider an example C++ class, which is used by Strous-trup [84, p.310] to demonstrate polymorphism (the original code was simplified):

```cpp
class Employee {
    string name;
    public:
        Employee(const string& name);
        virtual void print() { cout << name; };
}
```

Then, a sub-class of Employee is created, overriding the method \( print() \) with its own implementation:

```cpp
class Manager : public Employee {
    int level;
    public:
        Employee(int lvl) :
            Employee(name), level(lvl);
        void print() {
            Employee::print();
            cout << lvl;
}
```

Now, it is possible to define a function, which accepts a set of instances of class Employee and prints them one by one, calling their method \( print() \):

```cpp
void print_list(set<Employee*> &emps) {
    for (set<Employee*>::const_iterator p =
        emps.begin(); p != emps.end(); ++p) {
        (*p)->print();
    }
}
```

The information of whether elements of the set \( \emps \) are instances of Employee or Manager is not available for the \( print \_list \) function in compile-time. As explained by Booch et al. [15, p.103], "polymorphism and late binding go hand in hand; in the presence of polymorphism, the binding of a method to a name is not determined until execution."

Even though there are no explicitly defined types in \( \varphi \)-calculus, the conformance between objects is derived and "strongly" checked in compile time. In the example above, it would not be possible to compile the code that adds elements to the set \( \emps \), if any of them lacks the attribute \( print \). Since in EO, there is no reflection on types or any other mechanisms of alternative object instantiation, it is always known where objects are constructed or copied and what is the structure of them. Having this information in compile-time it is possible to guarantee strong compliance of all objects and their users. To our knowledge, this feature is not available in any other OOP languages.

5.4 Encapsulation

Encapsulation is considered the most important principle of OOP and, according to Booch et al. [15, p.51], “is most often achieved through information hiding, which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods." Encapsulation in C++ and Java is achieved through access modifiers like public or protected, while in some other languages, like JavaScript or Python, there are no mechanisms of enforcing information hiding.

However, even though Booch et al. [15, p.51] believe that “encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns,” in reality the barriers are not so explicit: they can be easily violated. West [87, p.141] noted that “in most ways, encapsulation is a discipline more than a real barrier; seldom is the integrity of an object protected in any absolute sense, and this is especially true of software objects, so it is up to the user of an object to respect that object’s encapsulation.” There are even programming “best practices,” which encourage programmers to compromise encapsulation: getters and
setters are the most notable example, as was demonstrated by Holub [46].

The inability to make the encapsulation barrier explicit is one of the main reasons why there is no information hiding in $\varphi$-calculus. Instead, all attributes of all objects in $\varphi$-calculus are visible to any other object.

In EO the primary goal of encapsulation is achieved differently. The goal is to reduce coupling between objects: the less they know about each other the thinner the the connection between them, which is one of the virtues of software design, according to Yourdon and Constantine [88].

In EO the tightness of coupling between objects should be controlled during the build, similar to how the threshold of test code coverage is usually controlled. At compile-time the compiler collects the information about the relationships between objects and calculates the coupling depth of each connection. For example, the object garage is referring to the object car.engine.size. This means that the depth of this connection between objects garage and car is two, because the object garage is using two dots to access the object size. Then, all collected depths from all object connections are analyzes and the build is rejected if the numbers are higher than a predefined threshold. How exactly the numbers are analyzed and what are the possible values of the threshold is a subject for future researches.

6 Complexity
One of the most critical factors affecting software maintainability is its complexity. The design of a programming language may either encourage programmers to write code with lower complexity, or do the opposite and provoke the creation of code with higher complexity. The following design patterns, also known as anti-patterns, increase complexity, especially if being used by less experienced programmers (most critical are at the top of the list):

- P1: Returning NULL references in case of error [44]
- P2: Implementation inheritance (esp. multiple) [45]
- P3: Mutable objects with side-effects [13]
- P4: Type casting [38, 68]
- P5: Utility classes with only static methods [18]
- P6: Runtime reflection on types [18]
- P7: Setters to modify object’s data [46]
- P8: Accepting NULL as function arguments [44]
- P9: Global variables and functions [68]
- P10: Singletons [18, 75]
- P11: Factory methods instead of constructors [18]
- P12: Exception swallowing [79]
- P13: Getters to retrieve object’s data [46]
- P14: Code reuse via mixins (we can think of this as a special case of workaround for the lack of multiple inheritance) [68]
- P15: Explanation of logic via comments [65, 68]
- P16: Temporal coupling between statements [18]
- P17: Frivolous inconsistent code formatting [65, 68]

In Java, C++, Ruby, Python, Smalltalk, JavaScript, PHP, C#, Eiffel, Kotlin, Erlang, and other languages most of the design patterns listed above are possible and may be utilized by programmers in their code, letting them write code with higher complexity. To the contrary, they are not permitted in EO by design:

- P1, P8 → There are no NULLs in EO
- P5, P11, P10 → There are no static methods
- P2 → There is no inheritance
- P3, P7 → There are no mutable objects
- P4, P6 → There are no types
- P9 → There is no global scope
- P12 → There are no exceptions
- P14 → There are no mixins
- P15 → Inline comments are prohibited
- P16 → There are no statements
- P17 → The syntax explicitly defines style

Thus, since in EO all patterns listed above are not permitted by the language design, EO programs will have lower complexity while being written by the same group of programmers.

7 Related Work
Attempts were made to formalize OOP and introduce object calculus, similar to lambda calculus [7] used in functional programming. For example, Abadi and Cardelli [2] suggested an imperative calculus of objects, which was extended by Bono and Fisher [14] to support classes, by Gordon and Hankin [41] to support concurrency and synchronisation, and by Jeffrey [55] to support distributed programming.

Earlier, Honda and Tokoro [47] combined OOP and $\pi$-calculus in order to introduce object calculus for asynchronous communication, which was further referenced by Jones [57] in their work on object-based design notation.
A few attempts were made to reduce existing OOP languages and formalize what is left. Featherweight Java is the most notable example proposed by Igarashi et al. [51], which is omitting almost all features of the full language (including interfaces and even assignment) to obtain a small calculus. Later it was extended by Jagannathan et al. [52] to support nested and multi-threaded transactions. Featherweight Java is used in formal languages such as Obsidian [25] and SJF [86].

Another example is Larch/C++ [21], which is a formal algebraic interface specification language tailored to C++. It allows interfaces of C++ classes and functions to be documented in a way that is unambiguous and concise.

Several attempts to formalize OOP were made by extensions of the most popular formal notations and methods, such as Object-Z [32] and VDM++ [33]. In Object-Z, state and operation schemes are encapsulated into classes. The formal model is based upon the idea of a class history [31]. Although, all these OO extensions do not have comprehensive refinement rules that can be used to transform specifications into implemented code in an actual OO programming language, as was noted by Paige and Ostroff [76].

Bancilhon and Khoshafian [6] suggested an object calculus as an extension to relational calculus. Jankowska [53] further developed these ideas and related them to a Boolean algebra. Lee et al. [62] developed an algorithm to transform an object calculus into an object algebra.

However, all these theoretical attempts to formalize OO languages were not able to fully describe their features, as was noted by Nierstrasz [73]: “The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundations for defining their semantics.” In addition, when describing the attempts of formalization, Eden [34] summarized: “Not one of the notations is defined formally, nor provided with denotational semantics, nor founded on axiomatic semantics.” Moreover, despite these efforts, Ciaffaglione et al. [22, 23, 24] noted in their series of works that a relatively little formal work has been carried out on object-based languages and it remains true to this day.

8 Acknowledgments

Many thanks to (in alphabetic order of last names) Fabrice Cabral, Kirill Chernyavskiy, Piotr Chmielowski, Danilo Danko, Konstantin Gukov, Andrew Graur, Ali-Sultan Ki-gizbaev, Nikolai Kudasov, Alexander Legalov, Tymur Lysenko, Alexander Naumchev, Alonso A. Ortega, John Page, Alex Panov, Alexander Pushkarev, Marcos Douglas B. Santos, Alex Semenyuk, Violetta Sim, Sergei Sklar, Stian Soiland-Reyes, Viacheslav Tradunskyi, Maxim Trunnikov, Ilya Trub, César Soto Valero, David West, and Vladimir Zakharov for their contribution to the development of EO and \( \phi \)-calculus.

References

[1] 2020. Erlang/OTP System Documentation.
[2] Martin Abadi and Luca Cardelli. 1995. An Impressive Object Calculus. Theory and Practice of Object Systems 1, 3 (1995).
[3] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. 2001. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications.
[4] Deborah J. Armstrong. 2006. The Quarks of Object-Oriented Development. Communications of the ACM 49, 2 (2006).
[5] Joe Armstrong. 2010. Erlang. Communications of the ACM 53, 9 (2010).
[6] Francois Bancilhon and Setrag Khoshafian. 1985. A Calculus for Complex Objects. In Proceedings of the Symposium on Principles of Database Systems.
[7] Hendrik P. Barendregt. 2012. The Lambda Calculus: Its Syntax and Semantics. College Publications.
[8] David Scott Bernstein. 2016. Beyond Legacy Code: Nine Practices to Extend the Life (and Value) of Your Software. Pragmatic Bookshelf.
[9] Lorenzo Bettini, Viviana Bono, and Betti Venneri. 2011. Delegation by Object Composition. Science of Computer Programming 76 (2011).
[10] Jeff Bezanon, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: a Fast Dynamic Language for Technical Computing.
[11] Xuan Bi and Bruno C. d S. Oliveira. 2018. Typed First-Class Traits. In Proceedings of the European Conference on Object-Oriented Programming.
[12] Adrian Birka and Michael D. Ernst. 2004. A Practical Type System and Language for Reference Immutability. ACM SIGPLAN Notices 39, 10 (2004).
[13] Joshua Bloch. 2016. Effective Java. Pearson Education India.
[14] Viviana Bono and Kathleen Fisher. 1998. An Impressive, First-Order Calculus With Object Extension. In Proceedings of the European Conference on Object-Oriented Programming.
[15] Grady Booch, Robert A. Maksimchuk, Michael Engle, Bobbi Young, Jim Comallen, and Kelli Houston. 2007. Object-Oriented Analysis and Design With Applications.
[16] Jan Bosch. 1997. Object-Oriented Frameworks: Problems & Experiences.
[17] Walter Bright, Alex, Andrei rescu, and Michael Parker. 2020. Origins of the D Programming Language. ACM on Programming Languages 4 (2020).
[18] Yegor Bugayenko. 2016. Elegant Objects. Amazon.
[19] Eden Burton and Emil Sekerinski. 2014. Using Dynamic Mixins to Implement Design Patterns. In Proceedings of the European Conference on Pattern Languages of Programs.
[20] Jeffrey Carter. 1997. OOP Vs. Readability. ACM SIGADA Ada Letters XVII (1997).
[21] Yoonsik Cheon and Gary T. Leavens. 1994. A Quick Overview of Larch/C++.
[22] Alberto Ciaffaglione, Luigi Liquori, and Mariano Miculan. 2003. Imperative Object-Based Calculi in Co-Inductive Type Theories. In Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning.
[23] Alberto Ciaffaglione, Luigi Liquori, and Mariano Miculan. 2003. Reasoning on an Imperative Object-Based Calculus in Higher Order Abstract Syntax. In Proceedings of the MERLIN’03: Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning About Languages With Variable Binding.
[24] Alberto Ciaffaglione, Luigi Liquori, and Mariano Miculan. 2007. Reasoning About Object-Based Calculi in (Co)Inductive Type Theory and the Theory of Contexts. Journal of Automated Reasoning 39 (2007).
[25] Michael J. Cohlenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and...
of Programming Languages and Tools.

[81] Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. 1998. Evolution of Object Behavior Using Context Relations. IEEE Transactions on Software Engineering 24, 1 (1998).

[82] Asaf Shelly. 2015. Flaws of Object Oriented Modeling. (2015).

[83] Mark Stefik and Daniel G. Bobrow. 1985. Object-Oriented Programming: Themes and Variations. AI Magazine 6, 4 (1985).

[84] Bjarne Stroustrup. 1997. The C++ Programming Language. Addison-Wesley Professional.

[85] David Ungar, R. Smith, and all B. 1987. Self: the Power of Simplicity. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications.

[86] Artem Usov and Prnela Dardha. 2020. SJF: an Implementation of Semantic Featherweight Java. In Proceedings of the Coordination Models and Languages. COORDINATION 2020, Vol. 12134.

[87] David West. 2004. Object Thinking. Pearson Education.

[88] Edward Yourdon and Larry L. Constantine. 1979. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall.