EOLANG and φ-calculus

Yegor Bugayenko
yegor256@gmail.com
Moscow, Russia

Abstract

Object-oriented programming (OOP) is one of the most popular paradigms used for building software systems. However, despite its industrial and academic popularity, OOP is still missing a formal apparatus similar to λ-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 φ-calculus. We also created EOLANG (also called EO), an experimental programming language based on φ-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 [89]. Madsen and Møller-Pedersen [68] claimed that "there are as many definitions of OOP as there papers and books on the topic." Armstrong [3] 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 [62] in 1966 [61] and since then was never introduced formally. Back in 1982, Rentsch [82] 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 [36]: "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 [77] 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 [2], 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 [83], and so on.

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 [46], "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 [93] 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 [45] 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 [79] 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 [88] 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 [63] 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 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 φ-calculus represents an object model through data and objects, while operations with them are possible through abstraction, 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 *free*—isn't *bound* to any object. An object is *closed* otherwise. *Abstraction* is the process of creating an abstract object. *Application* is the process of making a *copy* of an abstract object, specifying some or all of its free attributes with objects known as *arguments*. Application may lead to the creation of a closed object, or an abstract one, if not all free attributes are specified with arguments.
- An object may *decorate* another object by binding it to the φ attribute of itself. A decorator has its own attributes and the attributes of its decoratee.
- A special attribute δ may be bound 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 δ attribute is bound 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 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

Fig. 1 demonstrates the entire syntax of EO language in BNF. Similar to Python [67], 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" [64].

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 abstract objects, `()` for scope limitations, `!` for turning objects into constants, `:` for naming arguments, `"` (double quotes) for string literals, `'` (single quotes) for one-character literals, `@` for the decoratee, `^` for referring to the parent 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 letter and don't contain spaces or line breaks: `test-File` and `i文件` are valid identifiers. Java-notation is used for numbers, strings, and character literals.

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."

```
program     ::= [ license ] [ metas ] objects
objects     ::= object EOL { object EOL }
license     ::= { comment } EOL
metas       ::= { meta } EOL
comment     ::= '#' { ANY } EOL
meta        ::= '+' name [ '␣' ANY { ANY } ] EOL
name        ::= /[a-z]/ { ANY }
object      ::= ( foreign | local )
foreign     ::= abstraction '␣ /' name
local       ::= ( abstraction | application ) details
details     ::= [ tail ] { vtail }
tail        ::= EOL TAB { object EOL } UNTAB
vtail       ::= EOL ref [ htail ] [ suffix ] [ tail ]
abstraction ::= attributes [ suffix ]
attributes  ::= '[' attribute { '␣' attribute } ']'
attribute   ::= '@' | name [ '...' ]
suffix      ::= '␣' '>' '␣' ( '@' | name ) [ '!' ]
ref         ::= '.' ( name | '^' )
application ::= head [ htail ]
htail       ::= application ref
              | '(' application ')'
              | application ':' name
              | application suffix
              | application '␣' application
head        ::= name | data | '@' | '$'
              | '^' | '*' | name '.'
data        ::= bytes | string | integer
              | char | float | regex
bytes       ::= byte { '-' byte }
byte        ::= /[\dA-F][\dA-F]/
string      ::= /"[^"]*"/
integer     ::= /[+-]?\d+|0x[a-f\d]+/
char        ::= /'([^']|\\\d+)'/
regex       ::= //.+/[a-z]*/
float       ::= /[+-]?\d+(\.\d+)?/ [ exp ]
exp         ::= /e(+|-)?\d+/
```

Figure 1. The full syntax of EO in BNF. EOL is a line ending that preserves the indentation of the previous line. TAB is a right-shift of the indentation, while UNTAB is a left-shift. ANY is any symbol excluding EOL and '␣'. The texts between forward slashes are Perl-style regular expressions.

The syntax of EO makes a difference between these three categories.

This is a declaration of an abstract object `book`, which has a single *identity* attribute `isbn`:

```
1 [isbn] > book
```

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

```
2 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:

```
3 [isbn] > book2
4   "Object Thinking" > title
5   memory > 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 bound to objects. However, accessing `book2.isbn` will lead to an error, since the attribute `isbn` is free in the abstract object `book2`.

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

```
6 [isbn] > book3
7   "Object Thinking" > title
8   memory > price
9   [p] > set-price
10    ^.price.write p > @
```

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

```
11 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):

```
12 #␣This is a vector in 2D space
13 [dx␣dy]␣>␣vector
14 ␣␣sqrt.␣>␣length
15 ␣␣␣␣add.
16 ␣␣␣␣␣␣dx.pow 2
17 ␣␣␣␣␣␣dy.pow 2
```

The code at the line no. 12 is a *comment*. Two *free 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 a *bound 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.pow 2).add (dy.pow 2)).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.pow 2
.add
  dy.pow 2
.sqrt > length
```

Here, the line no. 19 is the application of the object `dx.pow` with a new argument `2`. Then, the next line is the object `add` 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 `add`. 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 free attributes in square brackets on one line, followed by a list of bound attributes each in its own line. For example, this is an abstract *anonymous* object (it doesn't have a name) with one free attribute `x` and two bound attributes `succ` and `prev`:

```
[x]
  x.add 1 > succ
  x.sub 1 > prev
```

The arguments of `add` and `sub` 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:

```
[x]
  x.add > succ
    1
  x.sub > prev
    1
```

This abstract object can also be written in a horizontal mode, because it is anonymous:

```
[x] (x.add 1 > succ) (x.sub 1 > prev)
```

2.3 EO to XML

Due 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=".add">
      <o base=".pow">
        <o base="dx"/>
        <o base="int" data="int">2</>
      </o>
      <o base=".pow">
        <o base="dy"/>
        <o base="int" data="int">2</>
      </o>
    </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 bound, such as `length`), or a name without a reference (if it is free, such as `dx` and `dy`).

2.4 Data Objects and Arrays

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.mul 2 3.14 > circumference
```

This syntax would be translated to XMIR (XML based data format explained in Sec. 6):

```
<o name="circle">
  <o name="r"/>
  <o base=".mul" name="circumference">
    <o base="r"/>
    <o base="int" data="int">2</o>
    <o base="float" data="float">3.14</o>
  </o>
</o>
```

Each object, if it is not abstract, has a "base" attribute in XML, which contains that name of an abstract object to be copied. The objects `int` and `float` are abstracts, but their names can't be used directly in a program, similar to how `r` or `mul` are used. The only way to make a copy of the abstract object `int` is to use a numeric literal like `2`. The literal `3.14` is turned into a copy of the object `float`.

The abstract objects which can't be used directly and can only be created by the compiler through data—are called *data*. The examples of some possible data are in the Tab. 1.

Data	Example	Size
`bytes`	`1F-E5-77-A6`	4
`string`	`"Hello, друг!"`	16
`char`	`'家'` or `'\u5BB6'`	2
`int`	`-42`	4
`float`	`3.1415926` or `2.4e-34`	4
`bool`	`true` or `false`	1
`regex`	`/[a-z]+.+/m`	9

Table 1. The syntax of all data with examples. The "Size" column denotes the number of bytes to be used by the value in the column "Example". UTF-8 is the encoding used in `string`, `char`, and `regex` objects.

The `array` is yet another data, which can't be copied directly. There is a special syntax for making arrays, which looks similar to object copying:

```
58 * "Lucy" "Jeff" 3.14
59 * > a
60   (* 'a')
61   true
62 * > b
```

The code at the line no. 58 makes an array of three elements: two strings and one float. The code at the lines 59–61 makes an array named `a` with another array as its first element and `true` as the second item. The code at the line no. 62 is an empty array with the name `b`.

2.5 Varargs

The last free attribute in an abstract object may be a *vararg*, meaning that any number or zero arguments may be provided. All of them will be packaged in an array by the compiler, for example:

```
63 [x...] > sum
64 sum 8 13 -9
```

Here, at the first line the abstract object `sum` is defined with a free vararg attribute `x`. At the second line a copy of the abstract object is made with three arguments. The internals of the `sum` will see `x` as an `array` with three elements inside.

2.6 Scope Brackets

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

```
65 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:

```
66 sum div 45 5 10
```

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

2.7 Inner Objects

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

```
67 # A point on a 2D canvas
68 [x y] > point
69   [to] > distance
70     length. > len
71       vector
72         to.x.sub (^.x)
73         to.y.sub (^.y)
```

The object `point` has two free attributes `x` and `y` and the attribute `distance`, which is bound to an abstract object with one free attribute `to` and one bound attribute `len`. The *inner* abstract object `distance` may only be copied with a reference to its *parent* object `point`:

```
74 .distance
75   point
76     5:x
77     -3:y
78   point:to
79     13:x
80     3.9:y
```

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

Inner object may refer to the parent object by using the `^` symbol.

2.8 Decorators

An object may extend another object by *decorating* it:

```
81 [center radius] > circle
82   center > @
83   [p] > is-inside
84     leq. > @
85       ^.@.distance $.p
86       ^.radius
```

The object `circle` has a special attribute `@` at the line no. 82, which denotes the *decoratee*: an object to be extended, also referred to as "component" by Gamma et al. [40].

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. 85. 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`:

```
87 ^.distance p
```

The inner object `is-inside` also has the `@` attribute: it decorates the object `leq` (stands for "less than equal"). The expression at the line no. 85 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).

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

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

Here, `i` will be a copy of `bool` behaving like `true` because `leq` decorates `bool`.

It is also possible to make decoratee free, similar to other free attributes, specifying it in the list of free attributes in square brackets.

2.9 Anonymous Abstract Objects

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

```
90 files
91   "/tmp"
92   *
93     [f]
94       f.isDir > @
```

Here the object `files` is copied with two arguments, the string `"/tmp"` and the array with a single element, which is an abstract object with a single free attribute `f`. The `files` will use this abstract object, which doesn't have a name, in order to filter out files while traversing the tree of directories. It will make a copy of the abstract object and then treat it as a boolean value in order to make a decision about the file.

The syntax may be simplified and the abstract object may be inlined (the array is also inlined):

```
95 files
96   "/tmp"
97   * ([f] (f.isDir > @))
```

An anonymous abstract object may have multiple attributes:

```
98 [x] (x.add 1 > succ) (x.sub 1 > prev)
```

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

2.10 Constants

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

```
99  [] > hello
100   stdout > say
101     sprintf
102       "The length of %s is %d"
103       stdin.nextLine > x!
104       x.length
```

The `sprintf` object will go to the `x` two times. First time, in order to use it as a substitute for `%s` and the second time for `%d`. There will be two round-trips to the standard input stream, which obviously is not correct. The exclamation mark at the `x!` solves the problem, making the object by 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.

2.11 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:

```
105 # (c) John Doe, 2021
106 # All rights reserved.
107 # The license is MIT
108
109 +package org.example
110 +alias org.eolang.io.stdout
111
112 [args...] > app
113   stdout > @
114     "Hello, world!\n"
```

2.12 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*. 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:0.6.0
+rt ruby eolang:0.1.0

[text] > stdout /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.

Atoms in EO are similar to "native" methods in Java and "extern" methods in C#.

3 Calculus

The proposed φ-calculus is based on set theory [57] and lambda calculus, representing objects as sets of pairs and their internals as λ-terms. The rest of the section contains formal definitions of data, objects, attributes, abstraction, application, decoration, and dataization.

3.1 Objects and Data

Definition 1. An **object** is a set of ordered pairs (a_i, v_i) such that a_i is an identifier, all a_i are different, and v_i is an object.

An identifier is either φ, ρ, 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

$$\texttt{book} = \{(\texttt{isbn}, \varnothing)\}, \tag{1}$$

where `isbn` is an identifier and $\varnothing$ is an empty set, which is a proper object, according to the Def. 1.

Definition 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 φ-calculus.

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

The object at the lines 3–5 may be represented as

$$\texttt{book2} = \left\{ \begin{array}{l} (\texttt{isbn}, \varnothing) \\ (\texttt{title}, \texttt{"Object Thinking"}) \\ (\texttt{price}, \texttt{memory}) \end{array} \right\}, \tag{2}$$

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. In an object x, a is a free **attribute** iff $(a, \varnothing) \in x$; it is a bound attribute iff $\exists (a, v) \in x$ and $v \neq \varnothing$.

In Eq. 2, identifiers `isbn`, `title`, and `price` are the attributes of the object `book2`. The attribute `isbn` is free, while the other two are bound.

Definition 4. If x is an object and $\exists (a, v) \in x$, then v may be referenced as $x.a$; this referencing mechanism is called **dot notation**.

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

It is possible to chain attribute references using dot notation, for example `book2.price.neg` is a valid expression, which means "taking the attribute `price` from the object `book2` and then taking the attribute `neg` from it."

Definition 5. If $x(a_i, v_i)$ is an object, then $\hat{x}$, a set consisting of all a_i, is its **scope** and the cardinality of $|\hat{x}|$ 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 Abstraction

Definition 6. An object x is **abstract** iff at least one of its attributes is free, i.e. $\exists (a, \varnothing) \in x$; the process of creating such an object is called **abstraction**.

An alternative "arrow notation" may be used to denote an object x in a more compact way, where free attributes stay in the parentheses on the left side of the mapping symbol $\mapsto$ and pairs, which represent bound attributes, stay on the right side, in double-square brackets. Eq. 2 may be written as

$$\begin{array}{l} \texttt{book2}\,(\texttt{isbn}) \mapsto [\![\\ \quad \texttt{title} \mapsto \texttt{"Object Thinking"}, \\ \quad \texttt{price} \mapsto \texttt{memory} \\ ]\!]. \end{array} \tag{3}$$

3.4 Application

Definition 7. If x is an abstract object and y is an object where $\hat{y} \subseteq \hat{x}$, then an **application** of y to x is a **copy** of x, a new object that consists of pairs $(a \in \hat{x}, v)$ such that $v = y.a$ if $x.a = \varnothing$ and $v = x.a$ otherwise.

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

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

Once set, bound 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 free, stay in the brackets on the left side of the mapping symbol $\mapsto$, while objects P provided as arguments stay on the right side, in the brackets. For example, the object at the line no. 74 may be written as

```
point(x ↦ 5, y ↦ -3).distance(
  p ↦ point(x ↦ 13, y ↦ 3.9)
),
```
(4)

and may further be simplified since the order of parameters is obvious:

```
point(5, -3).distance(point(13, 3.9)).
```
(5)

3.5 Inner and Parent Objects

Definition 8. If an object x is bound to an attribute of an object y, then $x.\rho$ denotes y; the object x is **inner** object, while y is its **parent**; an object, which is not bound to any attributes, is called **anonymous**.

For example, the object at the lines 6–10 has three inner objects bound to attributes `title`, `price`, and `set-price`:

```
book3(isbn) ↦ ⟦
  title ↦ "Object Thinking",
  price ↦ memory,
  set-price(p) ↦ ⟦
    φ ↦ ρ.price.write(p)
  ⟧
⟧,
```
(6)

where ρ.`price` refers to the attribute `price` of the parent object `book3`. It is not always required to mention ρ explicitly, however it it may be present for the sake of disambiguation.

Since the same object may be bound to more then one attribute, the parent ρ may depend on where the object is bound.

3.6 Decoration

Definition 9. 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, φ is a special identifier denoting the object being decorated within the scope of the decorator.

For example, the object at the lines 81–86 would be denoted by this formula:

```
circle(center, radius) ↦ ⟦
  φ ↦ center,
  is-inside(p) ↦ ⟦
    φ ↦ ρ.φ.distance(p).leq(radius)
  ⟧
⟧,
```
(7)

while the application of it would look like:

```
c ↦ circle(point(-3, 9), 40),
```
(8)

producing:

```
c ↦ ⟦
  center ↦ point(-3, 9),
  radius ↦ 40,
  φ ↦ center,
  is-inside(p) ↦ ⟦
    φ ↦ ρ.distance(p).leq(radius)
  ⟧
⟧.
```
(9)

Because of decoration, the expression $\rho.\varphi.$`distance` in Eq. 7 is semantically equivalent to the expression $\rho.$`distance` in Eq. 9.

The following expression makes a new object `is`, which represents a sequence of object applications ending with a copy of `leq`:

```
is ↦ c.is-inside(point(1, 7)),
```
(10)

producing:

```
c ↦ ⟦
  center ↦ point(-3, 9),
  radius ↦ 40,
  φ ↦ center,
  is-inside ↦ ⟦
    p ↦ point(1, 7),
    φ ↦ ρ.distance(p).leq(radius)
  ⟧
⟧.
```
(11)

3.7 Atoms

Definition 10. 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 λ-term, and s is its free attribute.

For example, the atom at the line no. 63 would be represented as

$$\texttt{sum}(\texttt{x}) \mapsto \lambda s.\ \sum_{i=0}^{s[0].\texttt{x}.\texttt{size}-1} s[0].\texttt{x}.\texttt{get}(i), \tag{12}$$

where the function calculates an arithmetic sum of all items in the array `x` and returns the result as a data. The argument of the function is a vector s where the

first element is the object under consideration, the second element is its parent object, the third element is the parent of the parent, and so on. Thus, $s[0]$ is the object `sum` itself, while $s[0].$`x` is its inner object `x`, and $s[0].$`x`.`get`(0) is the first element of it, if it is an array. It is expected that the array has an attribute `size` representing the total number of elements in the array.

Atoms may have their λ-terms defined outside of φ-calculus formal scope. For example, the object at the line no. 118 would be denoted as

$$\texttt{stdout}\,(\,\texttt{text}\,) \mapsto \lambda s.M_{\text{stdout}}, \tag{13}$$

where M_{stdout} is a λ-term defined externally.

3.8 Locators

Definition 11. Object **locator** is a unique dot-separated not-empty collection of identifiers prepended by either ξ, ρ, or Φ.

Locators are used to avoid ambiguity when referencing objects. For example, Eq. 9 may be refined as

```
c ↦ ⟦
  center ↦ Φ.point(Φ.-3, Φ.9),
  radius ↦ Φ.40,
  φ ↦ ξ.center,
  is-inside(p) ↦ ⟦
    φ ↦ ρ.distance(ξ.p).leq(
      ρ.radius
    )
  ⟧
⟧,
```
(14)

where ξ denotes the current abstract object and Φ refers to the anonymous abstract "root" object. Defining an object in a global scope means binding it to the object Φ, unless it is an anonymous object, as the one at the lines 26–30.

The most precise and complete formula for the object in the Eq. 11 would also include attribute names for the object application:

```
c ↦ ⟦
  center ↦ Φ.point(
    x ↦ Φ.-3,
    y ↦ Φ.9
  ),
  radius ↦ Φ.40,
  φ ↦ ξ.center,
  is-inside(p) ↦ ⟦
    φ ↦ ξ.ρ.distance(to ↦ ξ.p).leq(
      other ↦ ξ.ρ.radius
    )
  ⟧
⟧.
```
(15)

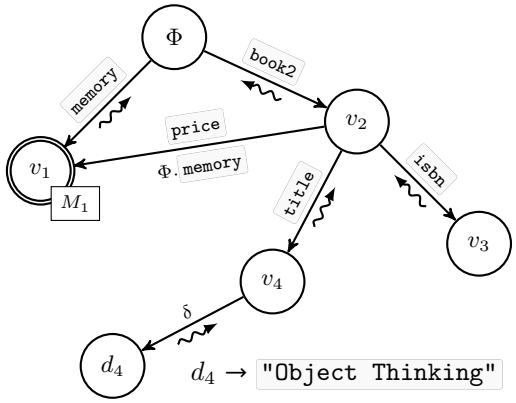


Figure 2. The object graph with a few objects from Eq. 2, where d_4 is `"Object Thinking"` data and M_1 is a lambda expression defined in the runtime.

4 Semantics

In order to explain how declarative expressions of φ-calculus can be translated into imperative instructions of a target computing platform, we 1) represent object model as *object graph*, 2) introduce a set of *graph modifying instructions* (GMI), 3) define *transformation rules* between φ-calculus expressions and GMIs, 4) suggest *dataization algorithm* turning object graph into function composition.

4.1 Object Graph

Consider the object from lines the lines 3–5, which is also represented by the expression in Eq. 1. Fig. 2 represents it as a graph.

The vertice at the top of the graph is the "root" object (see Def. 11), where all other objects that are not anonymous (see Def. 8) are bound to. The vertice v_2 is the abstract object `book2`. The name of the object within the scope of Φ is the label on the edge from Φ to v_2. The labeled edge between v_2 and v_3 makes the object v_3 an attribute of v_2 with the identifier `isbn`. Even though the object v_3 is $\varnothing$, the graph depicts it as any other object.

The rectangle attached to the vertice v_1 makes it an atom (see Def. 10) and M_1, the content of the rectangle, is its λ-term. Atoms are depicted with double-lined circles. The data d_4 attached to the vertice v_4 by the named edge δ is the text `"Object Thinking"`.

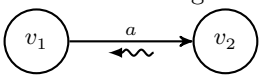
There are six graphical elements that may be present on an object graph: A *circle* with a name inside it is an object. A *named edge* from a circle to another circle is an attribute of the departing object. A *snake* edge is the ρ attribute. A *dotted* edge connects a copy with the origin. A *double-bordered* circle is an atom. A *rectangle* attached to a circle contains the λ-term of the atom.

4.2 GMI

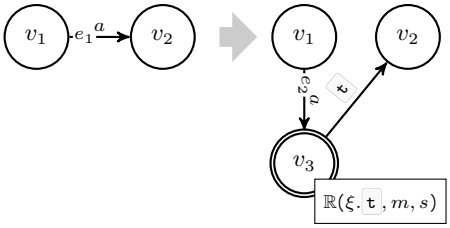
In order to formalize the process of drawing an object graph, we introduced a few GMIs:

`ADD`(v_1) Adds a new vertice v_1 to the graph:

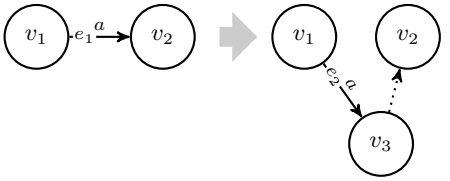
`BIND`(v_1, v_2, a) Adds a solid labeled uni-directed edge a from an existing vertice v_1 to an existing vertice v_2, making a snake edge if a equals to ρ and adding a reverse snake edge otherwise:



`DOT`(e_1, m, v_3, e_2) Breaks the edge e_1 going from v_1 to v_2, adding a new atom vertice v_3, connecting v_1 to v_3 with an e_2 labeled the same way as e_1, connecting v_3 and v_2 with an edge labeled as `t`, and attaching a rectangle with a special lambda expression to v_3:

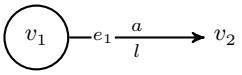


`COPY`(e_1, v_3, e_2) Breaks the edge e_1 going from v_1 to v_2, adding a new vertice v_3, connecting v_1 and v_3 with an edge e_2 labed the same way as e_1, and connecting v_3 and v_2 with a dotted edge:



`ATOM`(v_1, M_1) Attaches a rectangle to an existing vertice v_1 with a lambda expression M_1 inside and adds the second border to v_1:

`REF`(e_1, v_1, l, a) Starting from the vertice v_1, finds a vertice v_2 by the locator l and links them with an edge e_1 named as a with a supplementary label l (omitting the circle around the vertice v_2 is a visual trick that helps avoid a long arrow, which would need to be drawn from v_1 to the found v_2 otherwise):



All GMIs are idempotent, meaning that they have no additional effect if they are called more than once with the same input parameters. The object graph at Fig. 2 may be generated with the following ordered sequence of GMIs:

```
ADD(Φ)
ADD(v1);
ATOM(v1, M1);
BIND(Φ, v1, memory);
ADD(v2);
BIND(Φ, v2, book2);
ADD(v3);
BIND(v2, v3, isbn);
ADD(v4);
BIND(v2, v4, title);
REF(e, v2, Φ.memory, price);
ADD(d4);
BIND(v4, d4, δ);
```

4.3 Transformation Rules

In order to formalize the mechanism of turning φ-calculus formulas into an object graph, we introduced a number of transformation rules. R1 explains how an abstract object gets transformed to a sequence of GMIs:

$$\frac{v_i | x(a_1, a_2, \ldots, a_n) \mapsto [\![E]\!]}{\begin{array}{c}\mathtt{ADD}(v_{i\triangleright x}) \quad \mathtt{BIND}(v_i, v_{i\triangleright x}, x) \\ \forall j \in [1; n] \big(\mathtt{ADD}(v_{i\triangleright x\triangleright j}) \quad \mathtt{BIND}(v_{i\triangleright x}, v_{i\triangleright x\triangleright j}, a_j)\big) \\ v_{i\triangleright x} | E\end{array}} \textbf{R1}$$

The $v|E$ notation at the premise part of the rule means "E stands while the focus is at v," where E is an expression and v is an element of the graph, for example a vertice or an edge.

The hierarchical vertice indexing notation is used in order to avoid duplication of indexes. Thus, the index of the vertice $v_{i\triangleright x\triangleright 1}$ is unique on the graph. The symbol "$\triangleright$" is used as a delimiter between parts of the index. We decided to use this symbol instead of a more traditional dot because the semantic of the dot is already occupied by the dot notation in φ-calculus.

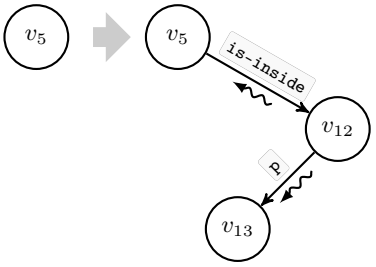
For the sake of simplicity of the graphs, the hierarchical notation won't be used in practical examples below.

Instead, single integer indexes will be used to denote vertices and edges, being incremented sequentially in order to avoid duplication.

Consider for example the abstract object bound to the attribute `is-inside` in Eq. 7. The premise $v_5|E$ will stand when the focus is at the vertice representing the object `circle`, where v_5 would be the vertice of it (the numbers 5 and 12 don't mean anything and are just placeholders):

$$\frac{v_5|\ \overbrace{\texttt{is-inside}}^{x}\ (\overbrace{\texttt{p}}^{a_1})\mapsto \overbrace{[\![\varphi\mapsto\ldots]\!]}^{E}}{\begin{array}{c}\texttt{ADD}(v_{12})\quad \texttt{BIND}(v_5,v_{12},\texttt{is-inside})\\ \texttt{ADD}(v_{13})\quad \texttt{BIND}(v_{12},v_{13},\texttt{p})\\ v_{12}|\varphi\mapsto\ldots\end{array}}$$

The effect of all GMIs generated by this rule would be the following on an object graph:



The E part of the premise is the internals of the abstract object `is-inside`. It will be processed by one of the rules, while looking at v_{12}. R2 explains how a comma-separated series of expressions break into individual rules (since the expression inside `is-inside` is the only one, this rule is not applicable):

$$\frac{v_i|E_1,E_2,\ldots,E_n}{v_i|E_1\quad v_i|E_2\quad\ldots\quad v_i|E_n}\textbf{R2}$$

R3 turns an attribute into an edge on the graph and then continues processing the expression that goes after the object being referred, by looking at the created edge:

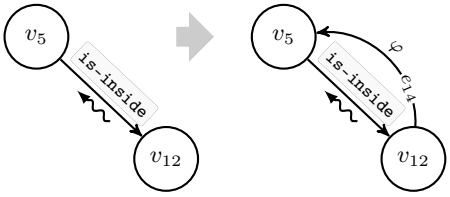
$$\frac{v_i|a\mapsto x\ E}{\texttt{REF}(e_{i\triangleright a},v_i,x,a)\quad v_i|x\quad e_{i\triangleright a}|E}\textbf{R3}$$

The notation "$x\ E$" in the premise of R3 splits the expression under consideration into two parts: the "head" of a single identifier x and the "tail" of the expression as E. In the conclusion part of the rule a vertice is found using the locator x and then a new edge is added, starting from the current vertice and arriving to the vertice found. Strictly, x must be a single identifier. However, in a more relaxed mode it is possible to have a longer locator as the head of the expression. For example, the expression $\rho.\rho.\texttt{p}$ can be split strictly on ρ as the head and $\rho.\texttt{p}$ as the tail; but it also can be split on $\rho.\rho$ as the head and `p` as the tail. Longer locators in the head part of the expression are only allowed if the vertice they refer to already exists on the graph. R3 also processes x in the conclusion part, providing other rules the opportunity to deal with it. In particular, R6 may process x if it is data.

The tranformation of the internals of `is-inside` with R3 would look like the following:

$$\frac{v_{12}|\ \overbrace{\varphi}^{a}\mapsto\overbrace{\rho}^{x}\ .\texttt{distance}(\texttt{p}).\texttt{leq}(\texttt{radius})}{\texttt{REF}(e_{14},v_{12},v_5,\varphi)\quad e_{14}|.\texttt{distance}(\texttt{p}).\texttt{leq}(\texttt{radius})}$$

Here ρ represents the x part of the premise and the expression that starts with a dot represents the E part. At the conclusion, x is being replaced with v_5, because ρ from the vertice v_{12} points to it: it is the parent object of v_{12}. The edge e_{14} created by the `REF` is used in the expression that finishes the conclusion, triggering the processing of the tail part of the formula: the head is the ρ, while the tail is the dot and everything that goes after it. Visually, the execution of R3 would produce the following changes on the object graph (the vertice v_{13} is not shown for the sake of brevity):



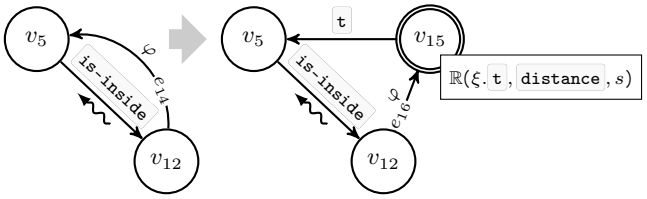
The dot notation is resolved by R4, which unlike previously seen rules, deals with an edge instead of a vertice:

$$\frac{e_i|.x\ E}{\texttt{DOT}(e_i,x,v_{i\triangleright x},e_{i\triangleright x\triangleright 1})\quad e_{i\triangleright x\triangleright 1}|E}\textbf{R4}$$

Here x is the identifier that goes after the dot and E is everything else, the tail of the expression. In this example, the instance of the rule would look like this:

$$\frac{e_{14}|\ \overbrace{.\texttt{distance}}^{.x}\ \overbrace{(\texttt{p}).\texttt{leq}(\texttt{radius})}^{E}}{\texttt{DOT}(e_{14},\texttt{distance},v_{15},e_{16})\quad e_{16}|(\texttt{p}).\texttt{leq}(\texttt{radius})}$$

Visually, the execution of this rule would lead to the following modifications on the object graph:



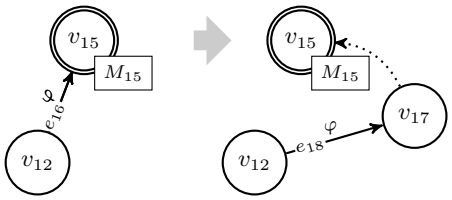
The application of arguments to abstract objects is transformed to the object graph by R5, which also deals with an edge instead of a vertice:

$$\frac{e_i|(E_1)\ E_2}{\texttt{COPY}(e_i,v_{i\triangleright 1},e_{i\triangleright 2})\quad v_{i\triangleright 1}|E_1\quad e_{i\triangleright 2}|E_2}\textbf{R5}$$

To continue the processing of the expression inside the abstract object `is-inside` the rule may be applied as the following:

$$\frac{e_{16}|\,(\texttt{to} \mapsto \xi.\texttt{p})\,.\texttt{leq}(\texttt{radius})}{\texttt{COPY}(e_{16}, v_{17}, e_{18}) \quad v_{17}|\texttt{to} \mapsto \xi.\texttt{p} \quad e_{18}|.\texttt{leq}(\texttt{radius})}$$

Visually, this rule would produce the following modifications on the graph:



The last rule deals with data, such as integers, string literals, and so on (together referred to as Γ):

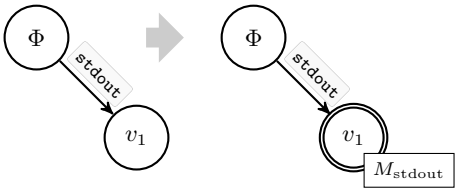
$$\frac{v_i|\Gamma}{\texttt{ADD}(d_i) \quad \texttt{BIND}(v_i, d_i, \delta) \quad d_i \to \Gamma}\textbf{R6}$$

Here the mapping $d_i \to \Gamma$ is added to the graph as a comment.

There is also one rule, which transforms atoms to the object graph by R7 attaching lambda expressions to vertices:

$$\frac{v|\lambda s.M}{\texttt{ATOM}(v, M)}\textbf{R7}$$

Visually, the rule would produce the following modifications on the graph for Eq. 13:

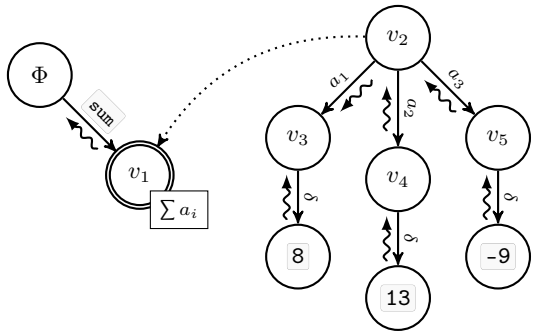


In order to demonstrate a larger example, Fig. 3 shows an object graph, which the described rules would generate by transforming the object `is` from Eq. 10.

4.4 Dataization

We define "dataization" as a process of turning an object into data, which said object *represents*. For example, the object at the line no. 64 represents an algebraic sum of three integers. The process of dataization expects each object to know what data it represents and if it doesn't know it, the object must know where to get the data. The object `sum` is not data, but it knows how to calculate it. Once being asked to turn itself into data it will ask all its three inner object the same question: "What data you represent?" They are integers and will return the data they have attached to their attributes δ. Then, the object `sum`, using its λ-term, will calculate the arithmetic sum of the numbers returned by its inner objects.

Visually, the object `sum` from the line no. 64 may be represented by the following object graph:



The dataization of v_2, which is an anonymous copy of `sum` with three arguments v_3, v_4, and v_5, would produce an arithmetic sum of three integers calculated by the λ-term of v_1.

We suggest the following recursive object discovery algorithm, which finds a vertice in a graph by its locator l and returns a vertice connected to it as an attribute a:

```
function ℝ(l, a, S)
  v ← l
  if l is a locator with a dot inside
    a' ← after the last dot in l
    l' ← before a' in l
    v ← ℝ(l', a', S)
  end if
  if v = ξ then v ← S[0]
  if v = ρ then v ← S[1]
  if v has a-edge to τ then return τ
  if v has φ-edge to τ then return ℝ(τ, a, τ + S)
  if v has a dotted edge to τ then return ℝ(τ, ξ.a, τ + S)
  if v has M then return ℝ((λs.M v + S), a, S)
  return ⊥
end
```

Here, S is a vector of vertices, while $v + S$ produces a new vector where v stays at the first position and all other elements of S follow. The notation $S[i]$ denotes the i-th element of the vector, while counting starts with zero.

The notation $(\lambda s.M\ v + S)$ means creating a function from the λ-term M with one parameter s and then calculating it with the argument $v + S$. It is expected that the function returns a locator of a vertice or the vertice itself, where the locator can be derived as the shortest path from Φ to the given vertice. The vector s, provided to the function as its parameter, is used in M when it is necessary to use $\mathbb{R}$ in order to find some object.

If a function returns data, which doesn't have a locator, a new vertice v_i is created implicitly with the locator $\Phi.v_i$,

and a single attribute δ connected to the data returned, where i is the next available index in the graph.

For example, $\mathbb{R}(\Phi.\texttt{c}.\texttt{center}.\texttt{y}, \delta, \varnothing)$ being executed on the graph presented at the Fig. 3 would return the vertice d_{21}, which is +9.

We also define a dataization function, which turns an object into data:

function $\mathbb{D}(l)$
  return $\mathbb{R}(l, \delta, \varnothing)$
end

The execution of the function $\mathbb{D}(x)$, where x is the "program" object, leads to the execution of the entire program. Program terminates with an error message when $\mathbb{D}(x)$ is $\bot$.

5 Pragmatics

First, the source code of EO is parsed by ANTLR4-powered parser and an intermediate representation is built in XML, as was demonstrated in the Section 2.3. The output format is called XMIR. One `.eo` file with the source code in EO produces one XMIR file with `.xml` extension.

XMIR is then refined via a *pipeline* of XSLT stylesheets. For example, XMIR at the lines 50–57 contains a reference to the object `r` at the line no.53. An XSL transformation adds an attribute `ref` to the XML element `<o/>`, referring it to the line inside XML document, where the object `r` is defined:

```
132 <o name="circle">
133   <o name="r"/>  <!-- Line no.133 -->
134   <o base=".mul" name="square">
135     <o base="r" ref="133"/>
136     <o base="int" data="int">2</o>
137     <o base="float" data="float">3.14</o>
138   </o>
139 </o>
```

There are over two dozens XSL transformations in the pipeline, which are applied to the XMIR in a specific order. New transformations can be added to the pipeline for example in order to detect inconsistencies in XMIR, enforce new semantic rules, or optimize object structures.

Then, XMIR can be translated to machine code, bytecode, C++ source code, or any other target platform language. We implemented a translator to Java source code, which represents XMIR objects as Java classes and attributes as pairs in encapsulated `java.util.HashMap` instances.

Then, Java source code is compiled to bytecode by OpenJDK Java compiler. Then, runtime dependencies with atoms are taken from Maven Central and placed to the Java classpath.

Finally, JRE runs the program through `Main.java` class together with all `.jar` dependencies in the classpath.

6 XMIR

Consider this sample EO snippet:

```
140 [x y...] > app
141   42 > n!
142   [t] > read
143     sub.
144       n.neg
145       t
```

It will be compiled to the following XMIR:

```
146 <o line="1" name="app">
147   <o line="1" name="x"/>
148   <o line="1" name="y" vararg=""/>
149   <o base="int" const="" data="int"
150     line="2" name="n">42</o>
151   <o line="3" name="read">
152     <o line="3" name="t"/>
153     <o base=".sub" line="4">
154       <o base="n" line="5"/>
155       <o base=".neg" line="5" method=""/>
156       <o base="t" line="6"/>
157     </o>
158   </o>
159 </o>
```

Each object is translated to XML element `<o>`, which has a number of optional attributes and a mandatory one: `line`. The attribute `line` always contains a number of the line where this object was seen in the source code.

The attribute `name` contains the name of the object, if it was specified with the `>` syntax construct.

The attribute `base` contains the name of the object, which is being copied. If the name starts with a dot, this means that it refers to the first `<o>` child.

The attribute `method` may be temporarily present, suggesting further steps of XMIR transformations to modify the code at the lines 154–155 to the following:

```
160 <o base=".neg" line="5"/>
161   <o base="n" line="5"/>
162 </o>
```

The semantics of the element `.sub` at the line no. 153 and the element `.neg` at the line no. 160 are similar: prefix notation for child-parent relationships. The XML element at the lines 160–162 means '.neg(n,t)', which in EO means 'n.neg t' or 'neg. n t'. The leading dot at the attribute `base` means that the first child of this XML element is the EO parent of it.

The attribute `data` is used when the object is data. In this case, the data itself is in the text of the object, as in the line no. 150.

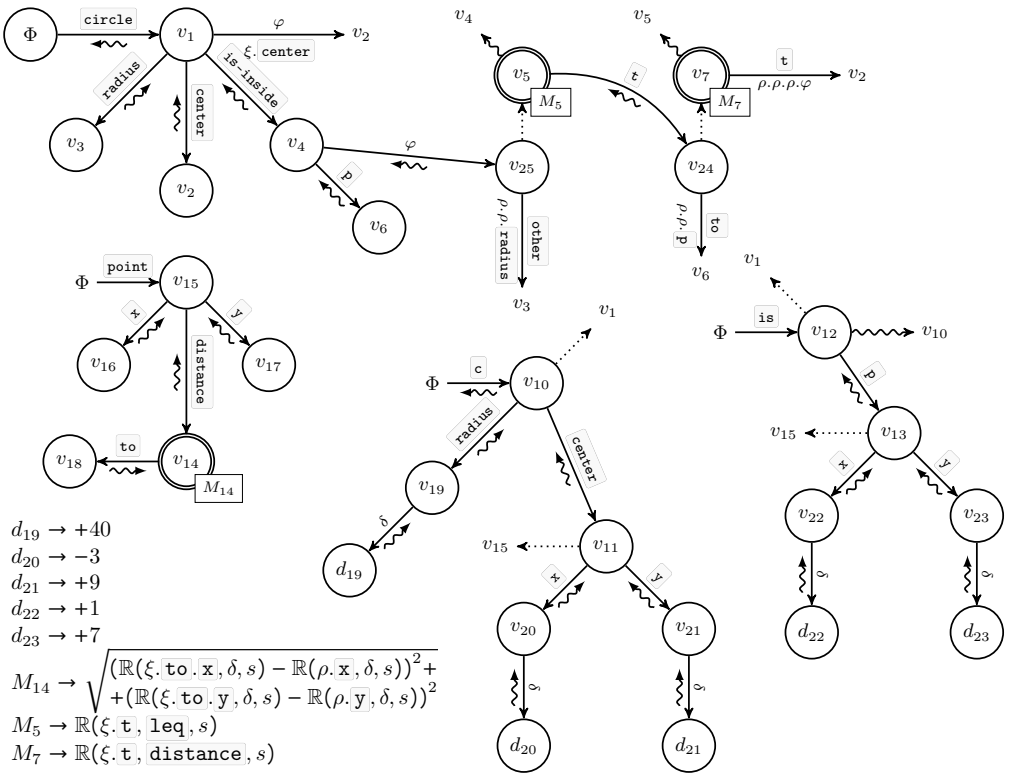


Figure 3. The graph of the object `is` from Eq. 10.

The attribute `vararg` denotes a vararg attribute. The attribute `const` denotes a constant.

7 Examples

The following examples demonstrate a few Java programs and their alternatives in EO.

7.1 Fibonacci Number

Fibonacci sequence is a sequence of positive integers such that each number is the sum of the two preceding ones, starting from 0 and 1, where:

$$F_n = F_{n-1} + F_{n-2}.$$

The formula can be implemented in Java using recursion, as suggested by Deitel and Deitel [27, p.743] (code style is modified):

```
public class FibonacciCalculator {
  public long fibonacci(long n) {
    if (n < 2) {
      return n;
    } else {
      return fibonacci(n-1) +
        ↪ fibonacci(n-2);
    }
  }
}
```

The same functionality would look in EO like the following:

```
[n] > fibo
  if. > @
    n.less 2
    n
    add.
      fibo (n.sub 1)
      fibo (n.sub 2)
```

7.2 Determining Leap Year

Consider a program to determine whether the year, provided by the user as console input, is leap or not. The Java code, as suggested by Liang [66, pp.105–106], would look like this (the code style was slightly modified):

```
import java.util.Scanner;
public class LeapYear {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a year: ");
    int year = input.nextInt();
    boolean isLeapYear =
      (year % 4 == 0 && year % 100 != 0) ||
      (year % 400 == 0);
    System.out.println(year +
      " is a leap year? " + isLeapYear);
```

```
  }
}
```

The same functionality would require the following code in EO:

```
+alias org.eolang.*
+alias org.eolang.io.stdout
+alias org.eolang.io.stdin
+alias org.eolang.txt.scanner

[args] > main
  seq > @
    stdout
      "Enter a year:"
    stdout
      concat
        scanner > year
          stdin
        .nextInt
        " is a leap year?"
        or.
          and.
            eq. (mod. year 4) 0
            not. (eq. (mod. year 100) 0)
          eq. (mod. year 400) 0
```

7.3 Division by Zero

As was explained by Eckel [34, p.314], since division by zero leads to a runtime exception, it is recommended to throw a more meaningful exception to notify the user about the exceptional situation. This is how it would be done in Java:

```
class Balance {
  // Calculate how much each user should
  // get, if we have this amount of users
  float share(int users) {
    if (users == 0) {
      throw new RuntimeException(
        "The number of users can't be zero"
      );
    }
    // Do the math and return the number
  }
}
```

This is how this functionality would look in EO:

```
[] > balance
  [users] > share
    if. > @
      eq. users 0
      []
        "The number can't be zero" > msg
        "InvalidInput" > type
      # Do the math and return
```

If the `users` argument is zero, an abstract object will be returned, with a free body and two bound attributes `msg` and `type`:

```
[]
  "The number of users can't be zero" > msg
  "InvalidInput" > type
```

Once this object will be touched by the runtime, it will cause the entire program to halt. This behavior is similar to what is happening in Java with exceptions.

7.4 Date Builder

Creating a date/time object is a common task for most programs, which is resolved in JDK8 [85] through the `Calendar.Builder` class, which suggests method cascading [7], also known as fluent interface [39], for its users (an innacurate and simplified example):

```
Calendar c = new Calendar.Builder()
  .setYear(2013)
  .setMonth(4)
  .setDay(6)
  .build();
```

The implementation of an immutable version of the `Calendar.Builder` class would look like this in Java:

```
class Builder {
  private final int year;
  private final int month;
  private final int day;
  Builder(int y, int m, int d) {
    this.year = y;
    this.month = m;
    this.day = d;
  }
  Builder setYear(int y) {
    return new Builder(
      y, this.month, this.day
    );
  }
  Builder setMonth(int m) {
    return new Builder(
      this.year, m, this.day
    );
  }
  Builder setDay(int d) {
    return new Builder(
      this.year, this.month, d
    );
  }
  Calendar build() {
    return new Calendar(
      this.year, this.month, this.day
    );
  }
}
```

This is how this functionality would look in EO, combining the builder and the calendar in one object:

```
[year month day] > calendar
  [y] > setYear
    calendar y month day > @
  [m] > setMonth
    calendar year m day > @
  [d] > setDay
    calendar year month d > @
  # The functionality of the calendar
  # goes in here...
```

This is how it would be used in EO:

```
calendar
.setYear 2013
.setMonth 4
.setDay 6
```

7.5 Streams

Working with a flow of binary or text data requires the use of stream objects, as explained by Metsker [73, p.226]. A non-canonical Java stream may be presented by a two-methods interface and a sample implementation of it:

```
interface Stream {
  void print(String text);
  void close();
}
class ConsoleStream implements Stream {
  @Override
  void print(String text) {
    System.out.println(text);
  }
  @Override
  void close() {
    // Maybe something else
  }
}
```

Then, it may be required to prepend all lines with a prefix. In order to do this a decorator design pattern may be used, as explained by Gamma et al. [40, p.196]:

```
class PrefixedStream implements Stream {
  private final Stream origin;
  PrefixedStream(Stream s) {
    this.origin = s;
  }
  @Override
  void print(String text) {
    this.origin.print("DEBUG: " + text);
  }
  @Override
  void close() {
    this.origin.close();
  }
}
```

The `PrefixedStream` encapsulates an object of the same type it implements. The decorator modifies the behavior of some methods (e.g., `print()`), while remain others untouched (e.g., `close()`). This is how the same design would look in EO:

```
[] > console_stream
  [text] > print
    stdout > @
      text
  [] > close
    # Do something here
```

Then, the decorator would look like this:

```
[@] > prefixed_stream
  [text] > print
    ^.@.print
      concat
        "DEBUG: "
        text
```

Here, the `^.@` attribute is the one that is being decorated. The object `prefixed_stream` has attribute `close` even though it is not declared explicitly.

If the object is used like this, where `stdout` is another stream, printing texts to the console:

```
prefixed_stream(stdout).print("Hello,
 ↪ world!")
```

Then the console will print:

```
DEBUG: Hello, world!
```

8 Mappings

There are language features in modern object-oriented programming languages, which do not exist in EO, for example: multiple inheritance, annotations, encapsulation and information hiding, mixins and traits, constructors, classes, assertions, static blocks, aspects, NULL references, generics, lambda functions, exception handling, reflection, type casting, and so on. We assume that all of them may be represented with the primitive language features of EO. There is no complete mapping mechanism implemented yet, but there are a few examples in this section that demonstrate how some features may be mapped from Java to EO.

8.1 Inheritance

This Java code utilizes inheritance in order to reuse the functionality provided by the parent class `Shape` in the child class `Circle`:

```
abstract class Shape {
  private float height;
  Shape(float h) {
    height = h;
```

```
  }
  float volume() {
    return square() * height;
  }
  abstract float square();
}
final class Circle extends Shape {
  private float radius;
  Circle(float h, float r) {
    super(h);
    radius = r;
  }
  float square() {
    return 3.14 * radius * radius;
  }
};
```

The method `volume` relies on the functionality provided by the abstract method `square`, which is not implemented in the parent class `Shape`: this is why the class is declared as `abstract` and the method `square` also has a modifier `abstract`. It is impossible to make an instance of the class `Shape`. A child class has to be define, which will inherit the functionality of `Shape` and implement the missing abstract method.

The class `Circle` does exactly that: it `extends` the class `Shape` and implements the method `square` with the functionality that calculates the square of the circle using the radius. The method `volume` is present in the `Circle` class, even though it is implemented in the parent class.

This code would be represented in EO as the following:

```
[child height] > shape
  [] > volume
    child.square.mul ^.height
[height radius] > circle
  shape $ height > @
  [] > square
    3.14.mul
      radius.mul
        radius
```

There is not mechanism of inheritance in EO, but decorating replaces it with a slight modification of the structure of objects: the parent object `shape` has an additional attribute `child`, which was not explicitly present in Java. This attribute is the link to the object that inherits `shape`. Once the `volume` is used, the attribute refers to the child object and the functionality from `circle` is used. The same mechanism is implemented in Java "under the hood": EO makes it explicitly visible.

8.2 Classes and Constructors

There are no classes in EO but only objects. Java, on the other hand, is a class-oriented language. In the snippet at the lines the lines 325–344, `Shape` is a class and a better way of mapping it to EO would be the following:

```
[] > shapes
  [c h] > new
    # Some extra functionality here, which
    # stays in the class constructor in Java
    []
      c > child
      h > height
      [] > volume
        child.square.mul ^.height
```

Here, `shapes` is the representation of Java class `Shape`. It technically is a factory of objects. In order to make a new, its attribute `new` must be used, which is similar to the operator `new` in Java. The functionality of a Java constructor may also be implemented in the attribute `new`, such as a validation of inputs or an initialization of local variables not passed through the constructor.

8.3 Mutability

All objects in EO are immutable, which means that their attributes can't be changed after an object is created. Java, on the other hand, enables mutability. For example, both `height` and `radius` in the lines the lines 325–344 are mutable attributes, which can be modified after an object is instantiated. However, the attribute `radius` of the EO object `circle` at the lines the lines 348–353 can't be modified. This may be fixed by using the object `memory`:

```
[height r] > circle
  memory r > radius
  shape $ height > @
  [] > square
    3.14.mul
      radius.mul
        radius
```

An instance of the object `memory` is created when the object `circle` is created, with the initial value of `r`. Then, replacing the object stored in the `memory` is possible through its attribute `write`:

```
circle 1.5 42.0 > c
c.radius.write 45.0
```

This code makes an instance of `circle` with the radius of `42.0`. Then, the radius is replaced with `45.0`.

8.4 Type Reflection

There are no types in EO, while Java not only have at least one type for each object, but also enable the

retrieval of this information in runtime. For example, it is possible to detect the type of the shape with this code:

```
if (s instanceof Circle) {
  System.out.println("It's a circle!");
}
```

In EO this meta-information about objects must be stored explicitly in object attribute, in order to enable similar reflection on types:

```
[height radius] > circle
  "circle" > type
  # The rest of the object
```

Now, checking the type of the object is as easy as reading the value of its attribute `type`. The mechanism can be extended with more additional information during the transition from Java to EO, such as information about attributes, decoratee, etc.

8.5 Exception Handling

There are no exceptions in EO, but there are objects that can't be dataized. A traditional Java `try/catch/finally` statements may be represented by an object `try` provided by EO runtime. For example, consider this Java code:

```
try {
  Files.write(file, data);
} catch (IOException e) {
  System.out.println("Can't write to file");
} finally {
  System.out.println("This happens anyway");
}
```

It may be translated to EO:

```
try
  []
    files.write > @
      file
      data
  []
    stdout > @
      "Can't write to file"
  []
    stdout > @
      "This happens anyway"
```

Now, throwing an exception is returning an object that can't be dataized and handling the exception is checking for whether the object has φ attribute or not. All of this is done by the object `try`.

8.6 Control Flow Statements

Java has a few control flow statements, such as `for`, `while`, `do`, `if`, `continue`, `break`. They don't exist in EO. However, EO may have objects that implement the required functionality in runtime, often with the help of mutable objects:

```
while (i < 100) {
  if (i % 2 == 0) {
    System.out.prinln("even!");
  }
  i++;
}
```

This code may be translated to EO as the following:

```
[]
  memory > i
  while. > @
    i.less 100
    seq
      if.
        eq. (i.mod 2 0)
        stdout "even!"
      i.write (i.add 1)
```

Here, `while` and `if` are the objects referred to as attributes of an object `bool`, while `i` is a mutable object.

9 Key Features

There are a few features that distinguish EO and φ-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 3 and 2, but to compare EO with other programming languages and informally identify similarities.

No Classes. EO is similar to other delegation-based languages like Self [91], 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 [38], "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 is inferred in compile-time and validated strictly, which other typeless languages such as Python, Julia [10], Lua [53], or Erlang [37] 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 is via object composition and decorators. There are OO languages without implementation inheritance, for example Go [30], but only Kotlin [59] has decorators as a language feature. In all other languages, the Decorator pattern [40] 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 assigned 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" [86].

No Primitive Data Types. There are no primitive data types in EO, which exist in Java and C++, for example. As in Ruby, Smalltalk [42], 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 [93, 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 [47] 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 [67] and Smalltalk [51], EO makes all object attributes publicly visible. There are no protected ones, because there is no implementation inheritance, which is considered harmful [52]. 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 [49].

No Global Scope. All objects in EO are assigned to some attributes. Objects constructed in the global scope of visibility are assigned to attributes of the `system` 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 [4], 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]. E.g., an attribute `r` may have an object `random` assigned to it, which is a random number generator. EO won't allow assigning another object to the attribute `r`, but every time the attribute is read its value will be different. There are number of OOP languages that also prioritize immutability of objects. In Rust [71], 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 [43]: happens through an imperative error-throwing statement. Instead, EO has a declarative mechanism for it, which is similar to Null Object design pattern [70]: 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. [76] 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.

10 Four Principles of OOP

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

10.1 Abstraction

Abstraction, which is called "modularity" by Booch et al. [15], is, according to West [93, 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 [90] suggests C++ classes as instruments of abstraction, the ultimate goal of abstraction is decomposition, according to West [93, p.73]: "composition is accomplished by applying abstraction—the 'knife' used to carve our domain into discrete objects."

In φ-calculus objects are the elements the problem domain is decomposed into, which goes along the claim of West [93, p.24]: "objects, as abstractions of entities in the real world, represent a particularly dense and cohesive clustering of information."

10.2 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 [74], is "to control the resulting potential complexity" of the design by enabling code reuse.

Consider a classic case of behaviour extension, suggested by Stroustrup [90, p.38] to illustrate inheritance. C++ class `Shape` represents a graphic object on the canvas (a simplified version of the original code):

```
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`:

```
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 [48], 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 [75].

The fragile base class problem is one of the reasons why there is no implementation inheritance in φ-calculus. Nevertheless, object hierarchies to enable code reuse in φ-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 [93, p.139], delegation is "a way to extend or restrict the behavior of objects by composition rather than by inheritance." Seiter et al. [87] 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 φ-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).

10.3 Polymorphism

According to Meyer [74, 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 Stroustrup [90, p.310] to demonstrate polymorphism (the original code was simplified):

```
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:

```
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()`.:

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

The information of wether 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 φ-calculus, the comformance 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.

10.4 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 [93, 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 [49].

The inability to make the encapsulation barrier explicit is one of the main reasons why there is no information hiding in φ-calculus. Instead, all attributes of all objects in φ-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 [94].

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.

11 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 [47]
- P2: Implementation inheritance (esp. multiple) [48]
- P3: Mutable objects with side-effects [13]
- P4: Type casting [41, 72]
- P5: Utility classes with only static methods [18]
- P6: Runtime reflection on types [18]
- P7: Setters to modify object's data [49]
- P8: Accepting NULL as function arguments [47]
- P9: Global variables and functions [72]
- P10: Singletons [18, 80]
- P11: Factory methods instead of constructors [18]
- P12: Exception swallowing [84]
- P13: Getters to retrieve object's data [49]
- P14: Code reuse via mixins (we can think of this as a special case of workaround for the lack of multiple inheritance) [72]
- P15: Explanation of logic via comments [69, 72]
- P16: Temporal coupling between statements [18]
- P17: Frivolous inconsistent code formatting [69, 72]

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.

12 Related Works

Attempts were made to formalize OOP and introduce object calculus, similar to lambda calculus [6] used in functional programming. For example, Abadi and Cardelli [1] suggested an imperative calculus of objects, which was extended by Bono and Fisher [14] to support classes, by Gordon and Hankin [44] to support concurrency and synchronisation, and by Jeffrey [58] to support distributed programming.

Earlier, Honda and Tokoro [50] combined OOP and π-calculus in order to introduce object calculus for asynchronous communication, which was further referenced by Jones [60] 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. [54], 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. [55] to support nested and multi-threaded transactions. Featherweight Java is used in formal languages such Obsidian [25] and SJF [92].

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 [81].

Bancilhon and Khoshafian [5] suggested an object calculus as an extension to relational calculus. Jankowska [56] further developed these ideas and related them to a Boolean algebra. Lee et al. [65] 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 [78]: "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 [35] 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.

13 Acknowledgments

Many thanks to (in alphabetic order of last names) Fabricio Cabral, Kirill Chernyavskiy, Piotr Chmielowski, Danilo Danko, Konstantin Gukov, Ali-Sultan Kirgizbaev,

Nikolai Kudasov, Alexander Legalov, Tymur λysenko, Alexandr Naumchev, Alonso A. Ortega, John Page, Alex Panov, Alexander Pushkarev, Marcos Douglas B. Santos, Alex Semenyuk, Violetta Sim, Sergei Skliar, Stian Soiland-Reyes, Viacheslav Tradunskyi, Ilya Trub, and César Soto Valero for their contribution to the development of EO and φ-calculus.

References

[1] Martín Abadi and Luca Cardelli. 1995. An Imperative Object Calculus. *Theory and Practice of Object Systems* 1, 3 (1995).

[2] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. 2001. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[3] Deborah J Armstrong. 2006. The Quarks of Object-Oriented Development. *Commun. ACM* 49, 2 (2006).

[4] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (2010).

[5] Francois Bancilhon and Setrag Khoshafian. 1985. A Calculus for Complex Objects. In *Symposium on Principles of Database Systems*.

[6] Hendrik P Barendregt. 2012. *The Lambda Calculus: Its Syntax and Semantics*. College Publications.

[7] Kent Beck. 1997. *Smalltalk Best Practice Patterns*. Prentice Hall.

[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 (Nov. 2011).

[10] Jeff Bezanson, 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 *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 Imperative, First-Order Calculus with Object Extension. In *European Conference on Object-Oriented Programming*.

[15] Grady Booch, Robert A Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. 2007. Object-Oriented Analysis and Design with Applications.

[16] Jan Bosch. 1997. Object-Oriented Frameworks: Problems & Experiences.

[17] Walter Bright, Andrei Alexandrescu, and Michael Parker. 2020. Origins of the D programming language. *ACM on Programming Languages* 4 (2020).

[18] Yegor Bugayenko. 2016. *Elegant Objects*. Vol. 1. Amazon.

[19] Eden Burton and Emil Sekerinski. 2014. Using Dynamic Mixins to Implement Design Patterns. In *European Conference on Pattern Languages of Programs*.

[20] Jeffrey Carter. 1997. OOP vs. Readability. *ACM SIGADA Ada Letters* XVII (March 1997).

[21] Yoonsik Cheon and Gary T Leavens. 1994. A Quick Overview of Larch/C++.

[22] Alberto Ciaffaglione, Luigi Liquori, and Marino Miculan. 2003. Imperative Object-Based Calculi in Co-inductive Type Theories. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*.

[23] Alberto Ciaffaglione, Luigi Liquori, and Marino Miculan. 2003. Reasoning on an Imperative Object-based Calculus in Higher Order Abstract Syntax. In *MERLIN'03: Proceedings of the 2003 ACM SIGPLAN Workshop on Mechanized Reasoning About Languages with Variable Binding*.

[24] Alberto Ciaffaglione, Luigi Liquori, and Marino 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. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. 2019. Obsidian: Typestate and Assets for Safer Blockchain Programming. *CoRR* 1 (2019).

[26] Ole-Johan Dahl and Kristen Nygaard. 1966. SIMULA: an ALGOL-based simulation language. *Commun. ACM* 9, 9 (1966).

[27] Harvey M. Deitel and Paul J. Deitel. 2007. *Java How to Program* (7th ed.). Prentice Hall.

[28] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. 1998. A Lambda Calculus of Objects with Self-Inflicted Extension. In *Conference on Object-Oriented programming, Systems, Languages, and Applications*.

[29] Edsger W Dijkstra. 1982. On the Role of Scientific Thought. In *Selected Writings on Computing: a Personal Perspective*. Springer Verlag.

[30] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language*. Addison-Wesley Professional.

[31] David Duke and Roger Duke. 1990. Towards a Semantics for Object-Z. In *International Symposium of VDM Europe*.

[32] Roger Duke, Paul King, Gordon Rose, and Graeme Smith. 2000. *The Object-Z Specification Language*. Citeseer.

[33] Eugene Durr and Jan Van Katwijk. 1992. VDM++, A Formal Specification Language for Object-Oriented Designs. In *Proceedings Computer Systems and Software Engineering*.

[34] Bruce Eckel. 2006. *Thinking in Java* (3 ed.). Prentice Hall.

[35] Amnon Eden. 2002. A Visual Formalism for Object-Oriented Architecture.

[36] Amnon H Eden and Yoram Hirshfeld. 2001. Principles in Formal Specification of Object-Oriented Architectures. In *CASCON'01: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*.

[37] Ericsson AB 2020. *Erlang/OTP System Documentation*. Ericsson AB.

[38] Kathleen Fisher and John C Mitchell. 1995. A Delegation-Based Object Calculus with Subtyping. In *International Symposium on Fundamentals of Computation Theory*.

[39] Martin Fowler. 2005. https://www.martinfowler.com/bliki/FluentInterface.html

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

[41] Gannimo. 2017. Type Confusion: Discovery, Abuse, and Protection. Chaos Computer Club e.V..

[42] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley.

[43] John B Goodenough. 1975. Exception Handling: Issues and a Proposed Notation. *Commun. ACM* 18, 12 (1975).

[44] Andy Gordon and Paul D. Hankin. 1998. A Concurrent Object Calculus: Reduction and Typing.

[45] James Gosling and Henry McGilton. 1995. The Java Language Environment. *Sun Microsystems Computer Company* 2550 (1995).

[46] Paul Graham. 2004. *Hackers & Painters: Big Ideas from the Computer Age*. O'Reilly Media.

[47] Tony Hoare. 2009. Null References: The Billion Dollar Mistake.

[48] Allen Holub. 2003. Why Extends is Evil.

[49] Allen Holub. 2004. More on Getters and Setters.

[50] Kohei Honda and Mario Tokoro. 1991. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming*.

[51] John Hunt. 1997. *Smalltalk and Object Orientation: an Introduction*. Springer Science & Business Media.

[52] John Hunt. 2000. Inheritance Considered Harmful. In *The Unified Process for Practitioners*. Springer.

[53] Roberto Ierusalimschy. 2016. *Programming in Lua, Fourth Edition*. Lua.Org.

[54] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. 2001. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001).

[55] Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. 2005. A Transactional Object Calculus. *Science of Computer Programming* 57, 2 (2005).

[56] Beata Jankowska. 2003. Yet Another Object-oriented Data Model and Its Application. *Control and Cybernetics* 32 (Jan. 2003).

[57] Thomas Jech. 2013. *Set Theory*. Springer Science & Business Media.

[58] Alan Jeffrey. 1999. A Distributed Object Calculus. In *Proceedings of FOOL*.

[59] Dmitry Jemerov and Svetlana Isakova. 2017. *Kotlin in Action*. Manning Publications Company.

[60] Cliff B. Jones. 1993. A Pi-calculus Semantics for an Object-Based Design Notation. In *International Conference on Concurrency Theory*.

[61] Alan Kay. 1986. *FLEX—A Flexible Extendable Language*. Master's thesis. University of Utah.

[62] Alan Kay. 1997. The Computer Revolution Hasn't Happened Yet.

[63] Zeba Khanam. 2018. Barriers to Refactoring: Issues and Solutions. *International Journal on Future Revolution in Computer Science & Communication Engineering* 4 (2018).

[64] Peter J Landin. 1966. The Next 700 Programming Languages. *Commun. ACM* 9, 3 (1966).

[65] Kwak Lee, Hoon-Sung Ryu, Hong-Ro, and Keun-Ho. 1996. A Translation of an Object Calculus into an Object Algebra. *The Transactions of the Korea Information Processing Society* 3 (1996).

[66] Y Daniel Liang. 2012. *Introduction to Java Programming: Brief Version* (9 ed.). Pearson Education, Inc.

[67] Mark Lutz. 2013. *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media.

[68] Ole Lehrmann Madsen and Birger Møller-Pedersen. 1988. What Object-Oriented Programming May Be-and What It Does Not Have to Be. In *European Conference on Object-Oriented Programming*.

[69] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1 ed.). Prentice Hall PTR, USA.

[70] Robert C Martin, Dirk Riehle, and Frank Buschmann. 1997. *Pattern Languages of Program Design 3*. Addison-Wesley.

[71] Nicholas D Matsakis and Felix S Klock. 2014. The Rust Language. *ACM SIGADA Ada Letters* 34, 3 (2014).

[72] Steve McConnell. 2004. *Code Complete, Second Edition*. Microsoft Press, USA.

[73] Steven John Metsker. 2002. *Design Patterns Java Workbook*. Addison-Wesley.

[74] Bertrand Meyer. 1997. *Object-Oriented Software Construction*. Prentice Hall.

[75] Leonid Mikhajlov and Emil Sekerinski. 1998. A Study of the Fragile Base Class Problem. In *European Conference on Object-Oriented Programming*.

[76] John C Mitchell, Furio Honsell, and Kathleen Fisher. 1993. A Lambda Calculus of Objects and Method Specialization. In *IEEE Symposium on Logic in Computer Science*.

[77] Oscar Nierstrasz. 1989. A Survey of Object-Oriented Concepts.

[78] Oscar Nierstrasz. 1991. Towards an Object Calculus. In *European Conference on Object-Oriented Programming*.

[79] Oscar Marius Nierstrasz. 2010. Ten Things I Hate About Object-Oriented Programming. *Journal of Object Technology* 9, 5 (2010).

[80] R. Nystrom. 2014. *Game Programming Patterns*. Genever Benning.

[81] Richard F Paige and Jonathan S Ostroff. 1999. An Object-Oriented Refinement Calculus.

[82] Tim Rentsch. 1982. Object Oriented Programming. *ACM SIGPLAN Notices* 17, 9 (1982).

[83] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Conference on Programming Language Design and Implementation*.

[84] Jonathan Rocha, Hugo Melo, Roberta Coelho, and Bruno Sena. 2018. Towards a Catalogue of Java Exception Handling Bad Smells and Refactorings. In *Proceedings of the 25th Conference on Pattern Languages of Programs (PLoP '18)*. The Hillside Group, USA.

[85] Herbert Schildt. 2018. *Java: The Complete Reference, Eleventh Edition* (11th ed.). McGraw-Hill Education.

[86] Frank Schmager, Nicholas Cameron, and James Noble. 2010. GoHotDraw: Evaluating The Go Programming Language With Design Patterns. In *PLATEAU'10: Evaluation and Usability of Programming Languages and Tools*.

[87] 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).

[88] Asaf Shelly. 2015. Flaws of Object Oriented Modeling. https://software.intel.com/content/www/us/en/develop/blogs/flaws-of-object-oriented-modeling.html

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

[90] Bjarne Stroustrup. 1997. *The C++ Programming Language* (3 ed.). Addison-Wesley Professional.

[91] David Ungar and Randall B Smith. 1987. Self: The Power of Simplicity. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*.

[92] Artem Usov and Prnela Dardha. 2020. SJF: An Implementation of Semantic Featherweight Java, In Coordination Models and Languages. COORDINATION 2020. *Lecture Notes in Computer Science* 12134.

[93] David West. 2004. *Object Thinking*. Pearson Education.

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