Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. XX, Num. XX (2018) 11-XX

RESEARCH ARTICLE

Formal Semantics for Java-like Languages and Research Opportunities

Semânticas Formais para Linguagens Similares ao Java e Oportunidades de Pesquisa

Samuel da Silva Feitosa[1]*, Rodrigo Geraldo Ribeiro[2], André Rauber Du Bois[1]

Abstract: Currently, Java is one of the most used programming languages, being adopted in many large projects, where applications reach a level of complexity for which manual testing and human inspection are not enough to guarantee quality in software development. Because of that, there is a growing research field that concerns the formalization of small subsets of Java-like languages aimed to conduct studies that were impossible to achieve through informal approaches. In this context, the objective of this paper is twofold: the discussion of the state-of-the-art on Java-like semantics and the presentation of research opportunities in this area. For the first goal, we present a research about Java-like formal semantics, filtering those that provide some insights in type-safety proofs, choosing the four most cited projects to be presented in details. We also briefly present some related studies that extended the originals aggregating useful features. Additionally, we provide a comparison between the most cited projects in order to show which functionalities are covered by each one of them. As for the second goal, we discuss possible future studies that can be performed by using the presented formal semantics.

Keywords: Java Semantics — Operational Semantics — Type Systems — Type Safety

Resumo: Atualmente Java é uma das linguagens de programação mais utilizadas, sendo adotada em muitos projetos de grande escala, onde aplicações alcançam um nível de complexidade no qual testes e inspeções manuais não são suficientes para garantir qualidade no desenvolvimento de software. Por conta disso, existe um crescente campo de pesquisa que diz respeito a formalização de pequenos fragmentos de linguagens similares ao Java, almejando a condução de estudos os quais eram impossíveis de realizar através de abordagens informais. Neste contexto, este artigo tem dois objetivos: a discussão do estado da arte sobre semânticas similares ao Java e a apresentação de oportunidades de pequisa nesta área. Para o primeiro objetivo, é proposta uma pequisa sobre semânticas formais da linguagem Java, filtrando aquelas que provêem provas de segurança de tipos, escolhendo os quatro projetos mais citados para serem apresentados em detalhes. Também são apresentados brevemente alguns estudos derivados que estendem os originais agregando funcionalidades. Adicionalmente, é apresentada uma comparação entre os projetos mais citados como forma de demonstrar quais funcionalidades são cobertas por cada um deles. Como segundo objetivo são discutidos possíveis trabalhos futuros que podem ser realizados através do uso das semânticas formais apresentadas.

Palavras-Chave: Semântica do Java — Semântica Operacional — Sistemas de Tipos — Segurança de Tipos

[1] *Programa de Pós-Graduação em Computação, Universidade Federal de Pelotas - UFPel, Brazil*
[2] *Programa de Pós-Graduação em Ciência da Computação, Universidade Federal de Ouro Preto - UFOP, Brazil*
***Corresponding author**: samuel.feitosa@inf.ufpel.edu.br

DOI: http://dx.doi.org/10.22456/2175-2745.XXXX • **Received:** dd/mm/yyyy • **Accepted:** dd/mm/yyyy

CC BY-NC-ND 4.0 - This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

1. Introduction

Nowadays, Java is one of the most popular programming languages [1, 2]. It is a general-purpose, concurrent, strongly typed, class-based object-oriented language. Since its release in 1995 by Sun Microsystems, and currently owned by Oracle Corporation, Java has been evolving over the time, adding features and programming facilities to its new versions. In a recent release of Java (JDK 8), the addition of new features such as lambda expressions, method references, and functional interfaces offer a programming model that fuses the object-oriented and functional styles [3].

Considering the growth in adoption of Java language for large projects, many applications have reached a level of complexity for which testing, code reviews, and human inspection are no longer sufficient quality-assurance guarantees. This problem increases the need for tools that employ static analysis techniques, aiming to explore all possibilities in an application, in order to guarantee the absence of unexpected behaviors [4]. Normally, developing and using these tools is a difficult task to be undertaken considering certain sizes of problems. To overcome this situation it is possible to model formal subsets of the problem applying a certain degree of abstraction, using only properties of interest, facilitating the understanding of the problem and also allowing the use of automatic tools [5].

Therefore, an important research area focuses on the formal semantics of languages and type-system specification, which helps the comprehension of a problem, allows to make formal proofs, and provides means to the establishment of fundamental properties of systems. Moreover, solutions can be machine checked through proof assistants providing a degree of confidence that cannot be reached using informal approaches. We should note that without a formal semantics it is impossible to state or prove anything about a language with certainty. For example, we cannot state that a program meets its specification, a type system is sound, or that a compiler or an interpreter is correct [6].

Some programming languages, such as ML [7], already come with a formal specification. Nevertheless, the official specification of languages are usually made in English prose with varying degrees of rigor and precision [5]. On the other hand, researchers are making efforts to formally study such languages. Indeed, there is a large body of literature on formal models of Java-like languages [8, 9, 10, 11], where each of these models are designed to treat certain features of Java in depth, abstracting other features. Also, recently a complete executable semantics for an older version of Java was proposed [6], but without contemplating new important features.

In this context, this work intends to provide a description of the most popular formalisms of Java according to Google Scholar database [12], comparing their features against each other, presenting significant projects that somehow use the previously presented formalisms, and discussing possible future studies. The criterion to define the most popular formalisms was the number of citations of each project found. Our search was filtered by those formalisms that described the Java semantics using structural operational semantics (both small-step or big-step), and by those that presented some insights on proofs of type-safety. These standard formalisms for Java-like languages greatly simplify investigations of novel programming constructs and help the identification of errors and misunderstandings. We are aware that our text does not cover all Java-like specifications, and that it only summarizes each of the presented approaches, but it can be useful as a starting point for future studies in this field.

Specifically, we claim the following contributions:

- We provide a catalog containing the description of the four most popular Java-like formalisms, presenting their main characteristics, and discussing related projects.
- We present a comparison among the features of each formalism, which can be useful when choosing one of them.
- We produce a list of topics for future work, which can be explored by others researchers.

The rest of this text is organized as follows: Section 2 presents the state-of-the-art in formal semantics for Java-like languages, reviewing the most used projects, and making a comparison between them. Section 3 discusses opportunities to research in this field, presenting studies found in the bibliography and possible future projects. Finally, we present the final remarks in Section 4.

2. Formal Semantics for Java

Java is a statically, strong typed, object-oriented, multi-threaded language. Except for threads, it is completely deterministic. The official specification of Java language is the JLS [13]. JLS has 755 pages and 19 chapters; more than 650 pages were used to describe the language and its behavior. Java is distributed as part of the Java Development Kit (JDK) and currently is in the version 10. At the imperative level, this language has 38 operators (JLS §3.12), 18 statements (§14), and some dozens of expressions (§15), among other features, and is evolving over time [6]. A Java program can be represented by a combination of several of its features. Considering that, the formalization (and update) of the whole language becomes an almost impossible task, justifying the need for definitions of formal subsets for Java.

Indeed, there exist several studies on the formalization of parts of the Java language [8, 9, 10, 11, 6, 14, 15, 16], and we have defined some criteria to select some of them to be presented in this text. Initially, we looked up for projects that describe the semantics of Java, particularly by structural operational semantics, filtering those that presented proofs of type-safety, both in formal or informal (non-mechanized) ways. From these, we selected the four most popular formalisms, i.e., those with the higher number of citations according to Google Scholar [12] database. Using this criterion, Featherweight Java could be considered the most popular, with almost 900 citations, followed by Classic Java, with approximately 500 quotes. Java$_S$ and Jinja currently present between 300 and 400 citations. The remainder of this section summarizes the selected formalizations, discussing their completeness and conformance with the official specification of Java, and comparing them with each other.

Featherweight Java

Featherweight Java (FJ) [10], proposed by Igarashi, Pierce, and Wadler, is a minimal core calculus for Java, in the sense that as many features of Java as possible are omitted while maintaining the essential flavor of the language and its type system. However, this fragment is large enough to include many useful programs. A program in FJ consists of a declaration of a set of classes and an expression to be evaluated, that corresponds to the *public static void main* method of Java.

FJ is related to Java, as λ-Calculus is to Haskell. It offers similar operations, providing classes, methods, attributes, inheritance and dynamic casts with semantics close to Java's. Featherweight Java project favors simplicity over expressivity and offers only five ways to create terms: object creation, method invocation, attribute access, casting, and variables. The following example shows how classes can be modeled in FJ. There are three classes, `A`, `B`, and `Pair`, with constructor and method declarations.

```
class A extends Object {
  A() { super(); }
}
class B extends Object {
  B() { super(); }
}
class Pair extends Object {
  A fst; B snd;
  Pair(A fst, B snd) {
    super();
    this.fst=fst;
    this.snd=snd;
  }
  Pair setfst(A newfst) {
    return new Pair(newfst, this.snd);
  }
}
```

FJ semantics applies a purely functional view without side effects. In other words, attributes in memory are not affected by object operations [17]. Furthermore, interfaces, overloading, call to base class methods, null pointers, base types, abstract methods, statements, access control, and exceptions are not present in the language [10].

Because the language does not allow side effects, it is possible to formalize the evaluation just using the FJ syntax, without the need for auxiliary mechanisms to model the heap.

Figure 1 presents the syntactic definitions originally proposed for FJ, where L refers to the classes list, K and M to constructors and methods respectively, and finally, e represents the expressions of that language. It is assumed that the set of variables includes the special variable *this* and *super* is a reserved keyword. Throughout this document, we write $\overline{f}$ as shorthand for a possibly empty sequence f_1,...,f_n (similarly for $\overline{C}$, $\overline{x}$, $\overline{e}$, etc.).

Figure 2 presents the evaluation rules originally proposed for FJ, formalizing how to evaluate *attribute access* (R-Field), *method invocation* (R-Invk), and *casts* (R-Cast) [10], the only three possible terms to be used in the *main program*. The presented functions *fields* and *mbody*, are also formalized in the original paper, representing respectively a way to obtain a list of attributes of some class C, and the term inside a method m that belongs to a given class C. In the *method invocation* rule, it is written $[\bar{x} \mapsto \bar{u}, \text{this} \mapsto \text{new } C(\bar{v})]t_0$ for the result of replacing x_1 by u_1,...,x_n by u_n, and *this* by "new $C(\bar{v})$" in expression t_0. In the *cast* rule, the symbol `<:` is used to express the sub-typing relation between C and D, stating that C is a subtype of D. These symbols are also used throughout the document.

Syntax

$L ::=$	class declarations	
$\quad class\ C\ extends\ C\ \{\overline{C}\ \overline{f}; K\ \overline{M}\}$		
$K ::=$	constructor declarations	
$\quad C(\overline{C}\ \overline{f})\ \{super(\overline{f});\ this.\overline{f} = \overline{f};\}$		
$M ::=$	method declarations	
$\quad C\ m(\overline{C}\ \overline{x})\ \{return\ e;\}$		
$e ::=$	expressions	
$\quad x$		variable
$\quad e.f$		field access
$\quad e.m(\overline{e})$		method invocation
$\quad new\ C(\overline{e})$		object creation
$\quad (C)\ e$		cast

Figure 1. Syntactic definitions for FJ.

Evaluation Rules

$$\frac{fields(\mathrm{C}) = \bar{\mathrm{C}}\ \bar{\mathrm{f}}}{\text{new } C(\bar{v}).\mathrm{f}_i \rightarrow v_i}\ \text{(R-Field)}$$

$$\frac{mbody(\mathrm{m}, \mathrm{C}) = (\bar{\mathrm{x}}, \mathrm{t}_0)}{\text{new } C(\bar{v}).\mathrm{m}(\bar{\mathrm{u}}) \rightarrow [\bar{\mathrm{x}} \mapsto \bar{\mathrm{u}}, \text{this} \mapsto \text{new } \mathrm{C}(\bar{\mathrm{v}})]\mathrm{t}_0}\ \text{(R-Invk)}$$

$$\frac{\mathrm{C} <: \mathrm{D}}{(\mathrm{D})\ (\text{new } \mathrm{C}(\bar{\mathrm{v}})) \rightarrow \text{new } \mathrm{C}(\bar{\mathrm{v}})}\ \text{(R-Cast)}$$

Figure 2. Evaluation rules for FJ.

The typing rules for expressions are in Figure 3. There we can note the use of an environment Γ, which represents a finite mapping from variables to types, written $\bar{x}$: $\bar{C}$. We let $\Gamma(x)$ denote the type C such that $x : C \in \Gamma$. The typing judgment for expressions has the form $\Gamma \vdash e : C$, read as "in the environment Γ, the expression e has type C". Some abbreviations are also used like in the reduction rules. The typing rules are syntax directed, with one rule for each form of expression, except for casts. Most of the typing rules are straightforward adaptations of the rules in Java: the rule (T-Var) checks if the variable x is in the Γ context and gets its type; rule (T-Field) uses the function *fields* to obtain the field type; the rules for method invocations (T-Invk) and for constructors (T-New) check that each actual parameter has a type that is subtype of the corresponding formal parameter type; the last three rules are related to casts, for each *upcasts*, *downcasts* and unrelated objects, respectively. The latter was added to allow proofs of type soundness.

For short, the formalization of sub-typing relation, auxiliary definitions, congruence and sanity checks for methods

Expression Typing

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \text{(T-Var)} \qquad \frac{\Gamma \vdash \mathrm{e}_0 : \mathrm{C}_0 \quad \mathit{fields}(\mathrm{C}_0) = \overline{\mathrm{C}}\,\overline{\mathrm{f}}}{\Gamma \vdash \mathrm{e}_0.\mathrm{f}_i : \mathrm{C}_i}\ \text{(T-Field)}$$

$$\frac{\Gamma \vdash \mathrm{e}_0 : \mathrm{C}_0 \quad \mathit{mtype}(\mathrm{m}, \mathrm{C}_0) = \overline{\mathrm{D}} \to \mathrm{C} \quad \Gamma \vdash \overline{\mathrm{e}} : \overline{\mathrm{C}} \quad \overline{\mathrm{C}} <: \overline{\mathrm{D}}}{\Gamma \vdash \mathrm{e}_0.\mathrm{m}(\overline{\mathrm{e}}) : \mathrm{C}}\ \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\mathrm{C}) = \overline{\mathrm{D}}\,\overline{\mathrm{f}} \quad \Gamma \vdash \overline{\mathrm{e}} : \overline{\mathrm{C}} \quad \overline{\mathrm{C}} <: \overline{\mathrm{D}}}{\Gamma \vdash \text{new } \mathrm{C}(\overline{\mathrm{e}}) : \mathrm{C}}\ \text{(T-New)} \qquad \frac{\Gamma \vdash \mathrm{e}_0 : \mathrm{D} \quad \mathrm{D} <: \mathrm{C}}{\Gamma \vdash (\mathrm{C})\ \mathrm{e}_0 : \mathrm{C}}\ \text{(T-UCast)}$$

$$\frac{\Gamma \vdash \mathrm{e}_0 : \mathrm{D} \quad \mathrm{C} <: \mathrm{D} \quad \mathrm{C} \neq \mathrm{D}}{\Gamma \vdash (\mathrm{C})\ \mathrm{e}_0 : \mathrm{C}}\ \text{(T-DCast)} \qquad \frac{\Gamma \vdash \mathrm{e}_0 : \mathrm{D} \quad \mathrm{C} \not<: \mathrm{D} \quad \mathrm{D} \not<: \mathrm{C}}{\Gamma \vdash (\mathrm{C})\ \mathrm{e}_0 : \mathrm{C}}\ \text{(T-SCast)}$$

Figure 3. Typing rules for FJ.

and classes were omitted here, but can be found in the original FJ paper [10].

An important contribution of FJ is the soundness proofs for the proposed type system. We show the *Theorem 1* as an example, to show the way proofs were modeled by the authors.

Theorem 1 (FJ Type Soundness) *If $\emptyset \vdash e : C$ and $e \rightarrow^* e'$ with e' a normal form, then e' is either a value v with $\emptyset \vdash v : D$ and $D <: C$, or an expression containing (D) new C($\overline{e}$) where $C <: D$.*

Proof. Immediate from Subject Reduction (Theorem 2.4.1) and Progress (Theorem 2.4.2) theorems found in the original paper [10].

Projects Derived from FJ

FJ is the currently most popular Java formalism and is both simple and concise due to careful selection of features. Several projects were presented extending FJ definitions by adding important features or novel programming constructors. Middleweight Java [18] and Welterweight Java [19] are new calculus based on FJ. The first remains compact and can be seen as an extension of FJ big enough to include the essential imperative features of Java, modeling object identity, field assignment, *null* pointers, constructor methods and block structure. The second presents an alternative core calculus to FJ, also modeling imperative features and supporting Java-style threads and Java-style concurrency control, with aliasing and thread-local stack frames.

Being a small subset of Java, novel constructions for object-oriented languages can be embedded in this language, allowing checking for type-safety. There exist several examples, and here we cite just a few. A proposal for closures [20] was presented years before the release of Java 8, quantum investigations in the OO context [21], a Coq formalization of FJ for studying product lines of theorems [22], and more recently a study addressing compositional and incremental type checking for object-oriented programming languages through co-contextual typing rules [23].

FJ is intended to be a starting point for the study of various operational features of object-oriented programming in Java-like languages, being compact enough to make rigorous proof feasible. Its operational semantics seems to be easier to understand than the others formalizations that follow this section.

ClassicJava

ClassicJava [8, 24] is a small subset of sequential Java proposed by Flatt, Krishnamurthi, and Felleisen. To model its type structure, the authors use type elaborations [25], where it is verified that a program defines a static tree of classes and a directed acyclic graph (DAG) of interfaces. For the semantics, rewriting techniques were used, where evaluation is modeled as a reduction on expression-store pairs in the context of a static type graph. The class model relies on as few implementation details as possible.

In ClassicJava, a program *P* is represented by a sequence of classes and interfaces followed by an expression. Each class definition consists of a sequence of field declarations and a sequence of method declarations. In the interfaces, the difference is that there are only methods. A method body in a class can be *abstract* when the method should be overridden in a subclass or can be an expression. In the case of interfaces, the method body must be always *abstract*. Similarly to Java, the objects are created with the *new* operator, but the constructors are omitted in the proposed specification. Thus, instance variables are initialized to *null*. There are also constructors that represents casts (*view* operator) and assignments (*let* operator). Figure 4 shows the formal syntax of ClassicJava.

Syntax

$P ::=$	program specification
$defn^*\ e$	
$defn ::=$	class and interface declarations
$class\ c\ extends\ c\ implements\ i^*\ \{field^*\ meth^*\}$	
$field ::=$	field statement
$t\ fd$	
$meth ::=$	method declarations
$t\ md(arg^*)\ \{body\}$	
$arg ::=$	argument list
$tvar$	
$body ::=$	method body declarations
$e \mid abstract$	
$e ::=$	expressions
$new\ c$	instancing a class
var	a variable name or *this*
$null$	*null* value
$e:\ c.fd$	field access
$e:\ c.fd = e$	field assignment
$e.md(e^*)$	method invocation
$super \equiv this:\ c.md(e^*)$	method invocation
$view\ t\ e$	cast
$let\ var = e\ in\ e$	assignment

Figure 4. Syntactic definitions for ClassicJava.

To be considered valid, a program should satisfy a number of simple predicates and relations, for example: *ClassOnce* indicates that a class name is declared only once, *FieldOncePerClass* checks if field names in each class are unique, *MethodOncePerClass* checks oneness for method names, *InterfacesAbstract* verifies that methods in interfaces are *abstract*, relation $\prec_P^c$ associates each class name in P to the class it extends, relation $\in_P^c$ (overloaded) capture the field and method declarations of P, and so on. The complete list of auxiliary definitions can be found in the original paper [8].

The operational semantics for ClassicJava is defined as a contextual rewriting system on pairs of expressions and stores. A store $\mathscr{S}$ is a mapping from *objects* to class-tagged field records. A field record $\mathscr{F}$ is a mapping from elaborated field names to values.

Figure 5 shows the operational semantics for ClassicJava. By looking at the *get* rule, for example, it is possible to note that a search for an attribute *fd* in a class c' is performed by using the *field record* $\mathscr{F}$, resulting in a value v. For the case of the *call* rule one can note that it invokes a method by rewriting the method call expression to the body of the invoked method, syntactically replacing argument variables in this expression with the supplied argument values and the special variable *this*. The other rules can be understood in a similar way.

The type elaboration rules translate expressions that access a field or call a method into annotated expressions. For instance, when a field is used, the annotation contains the compile-time type of the instance expression, which determines the class containing the declaration of the accessed field. The complete typing rules can be found in the original paper [8]. There the authors show that a program is well-typed if its classes definitions and final expressions are well-typed. A definition, in turn, is well-typed when its fields and method declarations use legal types and the method body expressions are well-typed. Finally, expressions are typed and elaborated in the context of an environment that binds free variables to types.

The authors also have presented formal proofs, which aim to guarantee the safety of this calculus, i.e., an evaluation cannot get stuck. This property was formulated through the *type soundness* theorem, where an evaluation step yields one of two possible configurations: either a well-defined error state or a new expression-store pair. In the latter case, there exists a new type environment that is consistent with the new store, and it establishes that the new expression has a type below t. The complete proof is available in an extended version of the original ClassicJava paper [24].

Projects Derived from ClassicJava

ClassicJava was meant to be an intuitive model of an essential Java subset. It was modeled originally to demonstrate an extension that develops a model of class-to-class functions referred as *mixins* - a *mixin* function maps a class to an extended class by adding or overriding fields and methods - and to state the type soundness theorems for the language. However, this calculus was used for many others purposes.

Several projects based on ClassicJava were proposed to work with threads and concurrency. One of them presents a static race detection for multi-threaded Java programs, through a small multi-threaded subset of Java, which extends the mentioned formalization [26]. Another similar project proposes a new static type system for multi-threaded programs, where well-typed programs in this system are guaranteed to be free of data races and deadlocks [27]. Also in this research area, among others, a core language to examine notions of safety with respect to transactions and mutual exclusion was proposed [28].

This formalism also was extended in different areas, such as contracts for OO languages, ownership types, and aspect-oriented languages. A study on the problem of contract enforcement in the object-oriented world from a foundational perspective was made generating Contract Java [29]. In another project, it was proposed a type system to work with ownership types, which provides a statically enforceable way of specifying object encapsulation and enables local reasoning about correctness in object-oriented languages [30]. And recently, some studies for aspect-oriented languages have been developed [31, 32].

Java$_S$, Java$_{SE}$ and Java$_R$

Another formal semantics for a subset of Java was developed by Drossopoulou and Eisenbach, where they have presented an operational semantics, a formal type system, and sketched[1]

[1] The authors provided informal (and incomplete) proofs to argue that the type system of Java is sound. The work of Syme [33] complemented these

Evaluation Rules

$$
\begin{array}{ll}
e = ... \mid \mathit{object} & E = [\,] \mid E : c.\mathit{fd} \mid E : c.\mathit{fd} = e \mid v : c.\mathit{fd} = E \\
v = \mathit{object} \mid \mathit{null} & \quad \mid E.\mathit{md}(e...) \mid v.\mathit{md}(v...E\ e...) \\
 & \quad \mid \mathit{super} \equiv v :\ c.\mathit{md}(v...E\ e...) \\
 & \quad \mid \mathit{view}\ t\ E \mid \mathit{let}\ \mathit{var} = E\ \mathit{in}\ e
\end{array}
$$

$$P \vdash \langle E[\mathit{new}\ c], \mathscr{S}\rangle \hookrightarrow \langle E[\mathit{object}], \mathscr{S}[\mathit{object}] \mapsto \langle c, \mathscr{F}\rangle]\rangle \ \mathit{where}\ \mathit{object} \notin \mathit{dom}(\mathscr{S})\ \mathit{and}\ \mathscr{F} = \{c'.\mathit{fd} \mapsto \mathit{null} \mid c \leq_P^c c'\ \mathit{and}\ \exists t\ s.t.\ \langle c'.\mathit{fd},\ t\rangle \in_P^c c'\} \qquad [\mathit{new}]$$

$$P \vdash \langle E[\mathit{object}:\ c'.\mathit{fd}], \mathscr{S}\rangle \hookrightarrow \langle E[v], \mathscr{S}\rangle\ \mathit{where}\ \mathscr{S}(\mathit{object}) = \langle c,\ \mathscr{F}\rangle\ \mathit{and}\ \mathscr{F}(c'.\mathit{fd}) = v \qquad [\mathit{get}]$$

$$P \vdash \langle E[\mathit{object}:\ c'.\mathit{fd} = v], \mathscr{S}\rangle \hookrightarrow \langle E[v], \mathscr{S}[\mathit{object} \mapsto \langle c,\ \mathscr{F}[c'.\mathit{fd} \mapsto v]\rangle]\rangle\ \mathit{where}\ \mathscr{S}(\mathit{object}) = \langle c,\ \mathscr{F}\rangle \qquad [\mathit{set}]$$

$$P \vdash \langle E[\mathit{object}.\mathit{md}(v_1,...,v_n)], \mathscr{S}\rangle \hookrightarrow \langle E[e[\mathit{object}/\mathit{this}, v_1/\mathit{var}_1,...,v_n/\mathit{var}_n]], \mathscr{S}\rangle\ \mathit{where}\ \mathscr{S}(\mathit{object}) = \langle c, \mathscr{F}\rangle\ \mathit{and}\ \langle \mathit{md}, (t_1...t_n) \longrightarrow t), (\mathit{var}_1...\mathit{var}_n), e\rangle \in_P^c\ c \qquad [\mathit{call}]$$

$$P \vdash \langle E[\mathit{super} \equiv \mathit{object} : c'.\mathit{md}(v_1,...,v_n)], \mathscr{S}\rangle \hookrightarrow \langle E[e[\mathit{object}/\mathit{this}, v_1/\mathit{var}_1,...,v_n/\mathit{var}_n]], \mathscr{S}\rangle\ \mathit{where}\ \langle \mathit{md}, (t_1...t_n) \longrightarrow t), (\mathit{var}_1...\mathit{var}_n), e\rangle \in_P^c\ c' \qquad [\mathit{super}]$$

$$P \vdash \langle E[\mathit{view}\ t'\ \mathit{object}], \mathscr{S}\rangle \hookrightarrow \langle E[\mathit{object}], \mathscr{S}\rangle\ \mathit{where}\ \mathscr{S}(\mathit{object}) = \langle c,\ \mathscr{F}\rangle\ \mathit{and}\ \leq_P^c t' \qquad [\mathit{cast}]$$

$$P \vdash \langle E[\mathit{let}\ \mathit{var} = v\ \mathit{in}\ e], \mathscr{S}\rangle \hookrightarrow \langle E[e[v/\mathit{var}]], \mathscr{S}\rangle \qquad [\mathit{let}]$$

$$P \vdash \langle E[\mathit{view}\ t'\ \mathit{object}], \mathscr{S}\rangle \hookrightarrow \langle \mathit{error}:\ \mathit{bad\ cast}, \mathscr{S}\rangle\ \mathit{where}\ \mathscr{S}(\mathit{object}) = \langle c,\ \mathscr{F}\rangle\ \mathit{and}\ \not\leq_P^c t' \qquad [\mathit{xcast}]$$

$$P \vdash \langle E[\mathit{null}:\ c.\mathit{fd}], \mathscr{S}\rangle \hookrightarrow \langle \mathit{error}:\ \mathit{derreferenced\ null}, \mathscr{S}\rangle \qquad [\mathit{nget}]$$

$$P \vdash \langle E[\mathit{null}:\ c.\mathit{fd} = v], \mathscr{S}\rangle \hookrightarrow \langle \mathit{error}:\ \mathit{derreferenced\ null}, \mathscr{S}\rangle \qquad [\mathit{nset}]$$

$$P \vdash \langle E[\mathit{null}.\mathit{md}(v_1,...,v_n)], \mathscr{S}\rangle \hookrightarrow \langle \mathit{error}:\ \mathit{derreferenced\ null}, \mathscr{S}\rangle \qquad [\mathit{ncall}]$$

Figure 5. Evaluation rules for ClassicJava.

an outline of the type soundness proof [34, 35, 9]. This subset includes primitive types, classes with inheritance, instance variables, and instance methods, interfaces, shadowing of instance variables, dynamic method binding, statically resolvable overloading of methods, object creation, null pointers, arrays and a minimal treatment of exceptions.

The author's approach was to define Java$_S$, which is a provably *safe* subset of Java containing the features listed above, a term rewrite system to describe the operational semantics and a type inference to describe compile-time type checking. They also prove that program execution preserves the type up to the subclass/subinterface relationship [9]. Furthermore, the type system was described in terms of an inference system.

This formal calculus was designed as a series of components, where Java$_S$ is a formal representation of the subset of Java semantics, Java$_{SE}$ is an enriched version of Java$_S$ containing compile-time type information, and Java$_R$ that extends Java$_{SE}$ and describes the run-time terms. Figure 6 shows the syntax of Java$_S$.

In Java$_S$ a program consists of a sequence of class bodies. Class bodies consist of a sequence of method bodies. Method bodies consist of the method identifier, the names and the types of the arguments, and a statement sequence. It is required exactly one *return* statement in each method body, and that it is the last statement. Also, it is considered only conditional statements, assignments, method calls, *try* and *throw* statements. This was done because iteration and others constructors can be achieved in terms of conditionals and recursion.

The calculus considers values, method calls, and instance variable access. The values are primitives (such as *true*, 4, 'c', etc.), references or arrays. References are *null*, or pointers to

proofs and provided a machine-checked version of them in the *Declare* proof assistant.

Syntax

$Program ::=$	program specification
$(ClassBody)^*$	
$ClassBody ::=$	class declarations
$ClassId\ ext\ ClassName\ \{(MethBody)^*\}$	
$MethBody ::=$	method declarations
$MethId\ is\ (\lambda\ ParId : VarType.)^*\ \{Stmts;\ return\ [Expr]\}$	
$Stmts ::=$	statement list
$\varepsilon \mid Stmts\ ;\ Stmt$	
$Stmt ::=$	statement declarations
$if\ Expr\ then\ Stmts\ else\ Stmts$	if statement
$Var := Expr$	assignment
$Expr.MethName(Expr^*)$	method invocation
$throw\ Expr$	exception throw
$try\ Stmts(catch\ ClassName\ Id\ Stmts)^*\ finally\ Stmts$	
$try\ Stmts(catch\ ClassName\ Id\ Stmts)^+$	
$Expr ::=$	expressions
$Value$	primitive values
Var	variable names
$Expr.MethName(Expr^*)$	method invocation
$new\ ClassName$	instancing a class
$new\ SimpleType([Expr])^+([])^*$	array instantiation

Figure 6. Syntactic definitions for Java$_S$.

objects. The expression *new C* creates a new object of class C, whereas the expression new T$[e_n]^+$ creates a n dimensional array. Also, pointers to objects are implicit.

As the others Java calculus, this proposal also models the class hierarchy, proposing the $\sqsubseteq$ relationship. Moreover, they also describe the environment, usually denoted by a Γ, using the *BNF* notation and containing both the subclass and interface hierarchies and variable type declarations. The environment also holds the type definitions of all variables and methods of a class and its interface. For short, this grammar [9] was omitted from here.

The following piece of code serves to demonstrate the Java$_S$ syntax and some of the features tacked by the authors.

```
ps = Phil ext Object {
  think is λy:Phil.{...}
  think is λy:FrPhil.{...}
}
FrPhil ext Phil {
  think is λy:
    Phil.{this.like := oyster;...}
}
```

Considering the presented program, the environment Γ is:

```
Γ = Phil ext Object {
like : Truth,
think : Phil → Phil
think : FrPhil → Book},
FrPhil ext Phil { like: Food,
think : Phil → Phil},
aPhil : Phil,pascal : FrPhil }
```

The operational semantics for this language was defined as a ternary rewrite relationship between configurations, programs, and configurations. Configurations are tuples of Java$_R$ terms and states. The terms represent the part of the original program remaining to be executed. The method calls evaluation were described as textual substitutions [9]. There are three relations for specifying the reduction of terms, one for each syntax category: $\overset{exp}{\rightsquigarrow}_{(\Gamma,p)}$, $\overset{var}{\rightsquigarrow}_{(\Gamma,p)}$, $\overset{stmt}{\rightsquigarrow}_{(\Gamma,p)}$. Global parameters are an environment Γ (containing the class and interface hierarchies, needed for runtime type checking) and the program p being executed [33].

The proposed rewrite system has 36 rules in total, where 15 of them are "redex" rules that specify the reduction of expressions in the cases where sub-expressions have reductions. A sample of this rules is:

$$\frac{stmt_0, s_0 \overset{stmt}{\rightsquigarrow}_{(\Gamma,p)} stmt_1, s_1}{\{stmt_0, stmts\}, s_0 \overset{stmt}{\rightsquigarrow}_{(\Gamma,p)} \{stmt_1, stmts\}, s_1}$$

There are 11 rules for dealing with the generation of exceptions: 5 for *null* pointers dereferences, 4 for bad array index bounds, one for bad size when creating a new array and one for runtime type checking when assigning to arrays. A simple example is:

$$\frac{\text{ground}(exp) \quad \text{ground}(val)}{\text{null}[exp] := val, s_0 \overset{stmt}{\rightsquigarrow}_{(\Gamma,p)} \text{NullPointExc}, s_0}$$

In this calculus, a term is *ground* if it is in normal form, i.e. no further reduction can be made. For short, several rules were omitted from this text, such as for field dereferencing, variable lookup, class creation, field assignment, local variable assignment, conditional statements, method call and rules for dealing with arrays. All of them are covered in the original paper [34]. This presentation also omits the type system rules and auxiliary definitions.

By proving subject reduction and soundness, the authors argue that the type system of Java$_S$ is sound, in the sense that unless an exception is raised, the evaluation of any expression will produce a value of a type "compatible" with the type assigned to it by the type system.

Projects Derived from Java$_S$

The formalization of Java$_S$ was a pioneer in this area, proposing an operational semantics, defining a formal type system and sketching an outline of the type soundness proof, becoming an inspiration to several projects. A machine-checked proof of the whole calculus was done, using a tool called *Declare*, complementing the written semantics and proofs by correcting and clarifying significant details [33].

The language Java$_{light}$ is directly related to Java$_S$, although it uses a different approach in the representation of programs and an evaluation semantics (aka "big-step") instead of a transition semantics (aka "small-step"). Java$_{light}$ language was also machine-checked, but in this case with the theorem prover Isabelle/HOL [36].

The operational semantics of Java$_S$ was also used to help in the development of proofs for JFlow, a new programming language that extends the Java language and permits static checking of flow annotations [37]. There are also some mechanized specifications for other languages, such as JavaScript and PHP, that were inspired by this work [38, 39, 40].

Java$_{light}$ and Jinja

Jinja [41, 11] is a Java-like programming language with a formal semantics designed to exhibit core features of Java, proposed by Nipkow and improved in conjunction with Klein. According to the authors, the language is a compromise between the realism of the language and tractability and clarity of the formal semantics. It is also an improvement of Java$_{light}$ [36], enhancing the treatment of exceptions.

In contrast to others formalizations, they presented a big and a small step semantics, which are independent of the type system, showing their equivalence. They also presented the type system rules, a definite initialization analysis, and the type safety proofs of the small step semantics. Additionally, the whole development has been carried out in the theorem prover Isabelle/HOL [42].

The abstract syntax of programs is given by the type definitions in Figure 7. A program is a list of *class declarations*. A class declaration consists of the name of the class and the class itself. A *class* consists of the name of its direct superclass, a list of field declarations, and a list of method declarations. A *field declaration* is a pair consisting of a field name and its type. A *method declaration* consists of the method name, the parameter types, the result type, and the method body. A *method body* is a pair of formal parameter names and an expression [11].

Syntax

$prog$ ::= $cdecl\ list$	program declaration
$cdecl$ ::= $cname \times class$	class declarations
$class$ ::= $cname \times fdecl\ list \times mdecl\ list$	class definition
$fdecl$ ::= $vname \times ty$	field declarations
$mdecl$ ::= $mname \times ty\ list \times ty$	method declarations
$J - mb$ ::= $vname\ list \times expr$	method body

Figure 7. Syntactic definitions for FJ.

Jinja is an imperative language, where all the expressions evaluate to certain values. Values in this language can be primitive, references, null values or the dummy value *Unit*. As an expression-based language, the statements are expressions that evaluate to *Unit*. The following expressions are supported by Jinja: the creation of new objects, casting, values, variable access, binary operations, variable assignment, field access, field assignment, method call, block with locally declared variables, sequential composition, conditionals, loops, and exception throwing and catching. The following example shows a program source-code using this language.

```
class B extends A {field F:TB
method M:TBs->T1 = (pB,bB)}
class C extends B {field F:TC
method M:TCs->T2 = (pC,bC)}
```

In this example, the field F in *class* C hides the one in *class* B. The same occurs with the method M. This differs from Java, where methods can also be *overloaded*, which means that multiple declarations of M can be visible simultaneously since they are distinguished by their argument types.

In this language, everything (expression evaluation, type checking, etc.) is performed in the context of a program P. Thus, there are some auxiliary definitions, omitted from here, like *is-class*, *subclass*, *sees-method*, *sees-field*, *has-field*, etc., that can be used to obtain information that are inside the abstract syntax tree of a program to assist on the evaluation.

The evaluation rules were presented in two parts: first, the authors introduce a big step or evaluation semantics, and then a small step or reduction semantics. The big step semantics was used in the compiler proof, and the small step semantics in the type safety proof. As this language deals with effects, it was necessary to define a *state*, that is represented by a pair, which models a *heap* and a *store*. A store is a map from variable names to values and a heap is a map from address to objects.

For the big-step semantics, the evaluation judgment is of the form $P \vdash \langle e, s\rangle \Rightarrow \langle e', s'\rangle$, where e and s are the initial expression and state, and e' and s' the final expression and state. Figure 8 show some of the rules for Jinja big-step semantics.

The first rule (R-New) first allocates a new address: function *new-Addr* returns a "new" address, that is, *new-Addr* h = $\lfloor a \rfloor$ implies $h\ a$ = *None*. Then predicate *has-fields* computes the list of all field declarations in and above class C, and *init-fields* creates the default field table. The second (R-Field) evaluates e to an address, looks up the object at the address, indexes its field table with (F, D), and evaluates to the value found in the field table. The lengthiest rule presented here (R-Method) is the one for a method call. It evaluates e to an address a and the parameter list ps to a list of values vs, looks up the class C of the object in the heap at a, looks up the parameter names pns and the *body* of the method M visible from C, and evaluates the body in a store that maps *this* to *Addr a* and the formal parameter names to the actual parameter values. The final store is the one obtained from the

Evaluation Rules

$$\frac{\textit{new-Addr } h = \lfloor a \rfloor \quad P \vdash C \textit{ has-fields FDTs} \quad h' = h(a \mapsto (C, \textit{init-fields FDTs}))}{P \vdash \langle \textit{new } C, (h, l)\rangle \Rightarrow \langle \text{addr a, (h', l)} \rangle} \quad \text{(R-New)}$$

$$\frac{P \vdash \langle e, s_0 \rangle \Rightarrow \langle \textit{addr } a, (h,l) \rangle \quad h\ a = \lfloor (C, fs) \rfloor \quad fs(F, D) = \lfloor v \rfloor}{P \vdash \langle e.F\{D\}, s_0 \rangle \Rightarrow \langle \text{Val } v, \text{(h,l)} \rangle} \quad \text{(R-Field)}$$

$$\frac{\begin{array}{c} P \vdash \langle e, s_0 \rangle \Rightarrow \langle \textit{addr } a,\ s_1 \rangle \quad P \vdash \langle ps, s_1 \rangle [\Rightarrow] \langle \textit{map Val vs}, (h_2, l_2) \rangle \\ h_2\ a = \lfloor (C, fs) \rfloor \quad P \vdash C \textit{ sees } M\text{: } Ts \rightarrow T = (\textit{pns, body}) \textit{ in } D \quad |vs| = |pns| \\ l_2' = [\textit{this} \mapsto \textit{Addr a, pns } [\mapsto] \textit{ vs}] \quad P \vdash \langle \textit{body}, (h_2, l_2') \rangle \Rightarrow \langle e', (h_3, l_3) \rangle \end{array}}{P \vdash \langle e.M(ps), s_0 \rangle \Rightarrow \langle e', (h_3, l_2) \rangle} \quad \text{(R-Method)}$$

Figure 8. Partial big-step semantics for Jinja.

evaluation of the parameters. The complete set of rules can be found at the original papers [41, 11].

According to the authors, the small-step semantics was provided because the big step semantics has several drawbacks, for example, it cannot accommodate parallelism, and the type safety proof needs a fine-grained semantics. The main difference between the two proposed semantics is that in the small-step they present *subexpression reduction*, which essentially describe the order that subexpressions are evaluated. Having the subexpressions sufficiently reduced, they describe the *expression reduction*. Most of that rules are fairly intuitive and many resemble their big-step conterparts [11]. These rules were omitted from this text.

In their papers, the authors relate the big-step with the small-step semantics, show the type-system rules and then prove its type-safety by showing the progress and preservation theorem for the proposed language. Additionally, the whole development of this project runs to 20000 lines of Isabelle/HOL text, which can be found online [42].

Projects Derived from Java$_{light}$ and Jinja

Jinja has become the basis for further investigations when concerning to Java-like languages. For example, it was possible to work on a framework to transform a single-threaded operational semantics into a semantics with an interleaved execution of threads, as an extension of Jinja [43]. Jinja was also used as starting point for a formalization of multiple inheritance in C++ [44]. Proof-synthesis algorithms for flow chart languages were also derived from this formalization [45].

More recently, among others projects that are based on Jinja or make use of it for some purpose, it was proposed a tool for the automated termination analysis of Java Bytecode programs, allowing verification of runtime complexity of existing Java programs [46]. In this same research area, it was revisited some known transformations from Jinja bytecode to rewrite systems from the viewpoint of runtime complexity, where the authors have been proposed an alternative representation of Jinja bytecode executions as computational graphs obtaining a novel representation [47]. Furthermore, there exist projects that make use of more precise formalizations in some parts of the project, as the hybrid approach for proving noninterference of Java programs [48]. Another interesting project, concerns about a Coq formal specification of the Siaam model [49], built over the Jinja specification.

Because these languages include bytecode-verification, virtual machine, compiler and a machine-checked formalization using the Isabelle/HOL theorem prover, they can be used in different kinds of projects and can be explored in future studies.

A Comparison Between the Most Used Semantics

The use of formal modeling can offer significant advances to the design of a complex system. The introduction of lightweight versions of a programming language, where complex features are dropped to enable rigorous arguments about key properties, allow a better understanding of the language characteristics, facilitate the investigation of novel constructions, and can be a useful tool for studying the consequences of extensions and variations.

However, choosing a formal model for a programming language when starting a research can be a difficult task, since there is a large number of projects in this area and several factors to be considered. For example, one project may need a more complete semantics, where a big number of features are included, while another project could prefer a more compact language, in order to study some specific extension that does not depend on a complete approach. A look at the related projects can be a good starting point to choose among the variety of formalisms, because it is possible to observe the pattern applied to these projects, and the similarities with the intended project. Considering this, in this section, we provide a comparison between the models presented in previous sections.

Table 1 shows which features are modeled in each of the discussed formal languages. Features of original Java that are not modeled in any of the presented languages (for example packages, access modifiers, λ-expressions, concurrency, reflection, among others) were suppressed from this table. For clarity, we split the table into categories, and we group some similar features for space optimization. We use three different kinds of support level, where ● means that the category or feature is fully supported by the presented formalism, ◐ stands for a partial support, and ○ when the feature is not supported at all. For presentation purposes we abbreviate Featherweight Java as FJ, ClassicJava as CJ, Java$_{\mathcal{S}}$ as JS, and Jinja as JJ in the first line of the table. We intended to present the main functionalities for each studied language, thus some minor features may not appear in this table.

Feature	**FJ**	**CJ**	**JS**	**JJ**
Primitive Types and Values	○	○	●	◐
Integer	○	○	●	●
Boolean	○	○	●	●
String literal	○	○	●	○
Basic Statements	○	◐	●	●
Null values and Assignment	○	●	●	●
Conditionals	○	○	●	●
Loops and Sequences	○	○	●	●
Try-catch-finally	○	○	●	●
Math Operators	○	○	○	●
Basic OOP	◐	◐	◐	◐
Classes and inheritance	●	●	●	●
Interfaces	○	●	●	○
Casts	●	●	○	●
Constructors, super and this	●	◐	○	◐
Polymorphism and overriding	●	●	○	●
Overloading and static methods	○	○	○	○
Generics	●	○	○	○
Arrays	○	○	●	○
Formal Specifications	◐	◐	◐	●
Big-step semantics	○	○	○	●
Small-step semantics	●	●	●	●
Progress and preservation	●	●	●	●
Use of proof assistant	○	○	◐	●

Support level: ● = Full ◐ = Partial ○ = None
FJ is Feath. Java, CJ is ClassicJava, JS is Java$_{\mathcal{S}}$, JJ is Jinja.

Table 1. Comparison between the most used Java-like semantics.

When looking at this table, we can identify two different patterns. The first, represented by Featherweight Java and ClassicJava, is compactness. The second, represented by Java$_{\mathcal{S}}$ and Jinja, is completeness. While the first ones are concentrated on a minimal object-oriented formalism, the others are formalizing a larger subset of Java language. These patterns can be useful to choose between one or another approach, so next, we discuss each pattern separately.

Featherweight Java and ClassicJava offer similar functionalities, where the goal of these projects was to define a core calculus that is as small as possible, capturing just the features of Java that are relevant for some particular task. In the case of FJ, the task was to analyze extensions of the core type system. The task of CJ was to analyze an extension of Java with mixins, a feature of Common Lisp language. The approach demonstrated by the authors of FJ is somewhat smaller than CJ, where the syntax, typing rules, and operational semantics of FJ take approximately three times less space than the other. Consequently, the soundness proofs are also correspondingly smaller. This fact can be a criterion for choosing one instead of the other. By the related projects presented previously, we can note that FJ was applied mostly in Java extensions or novel constructions in the object-oriented context, while CJ was applied in several projects to work with threads and concurrency. It is obvious that these projects can be (and were) used in different areas, but the applications in related projects can be useful to decide which one is best for new projects.

The functionalities offered by Java$_{\mathcal{S}}$ and Jinja are also similar. The goal of the first was to show that Java's type system is sound, while the goal of the second was to provide a formal semantics of the core features of Java, emphasizing on a unified model of the source language, the virtual machine, and the compiler. JS was one of the first steps toward a formal semantics for Java, and hence, it was an inspiration for later projects. However, because it explored a larger subset of an older version of Java, it is not usually taken as the basis for new projects, but it can be used for study purposes. In contrast, JJ was widely used as the basis for investigations of object-oriented characteristics. As JJ is not properly a subset of Java, it was also used on formalizations of other object-oriented languages. Because JJ also offers proofs, a virtual machine, and a compiler verified in the theorem prover Isabelle/HOL, it seems to be a good formalism when a more complete formalism of Java is needed.

The comparison presented in this section does not intend to cover all aspects of the presented formalisms, but it can provide insights on useful criteria for when choosing a formalism to be applied in some project. With this at hand, the next section presents some ideas that can be explored by using this material.

3. Research Opportunities

This section is dedicated to providing some insights for future research in the area of formal semantics and type system of programming languages, specifically in the object-oriented context. We organize the ideas in some categories, as follows.

Functional concepts in object-oriented languages: Traditional programming languages (imperative) are incorporating functional features over the years, making the code writing more and more multi-paradigm. For example, the concept of lambda expressions was embedded in version 8 of Java language [3], which allows treating functions as arguments, making the language more expressive. In this sense, it is possible to investigate what kind of abstractions used in functional languages can be mapped to the object-oriented ones using these new constructions. Among several possibilities, one can cite: parallel evaluation for working with lists, explore the *lazy* evaluation strategy, using infinite streams, the concept of monads [50], etc. As far as it is known, there is no formalization of these new aspects exactly as Java language works, such as lambda expressions, method references, and functional interfaces, and it can also be explored in future work.

Property-based random testing and mechanized proofs: Usually, in large projects, there exist some mechanisms to automate the management of testing. Unfortunately, such tests rarely cover all interesting cases of code, which means that some bugs may never be discovered. The use of random testing in the context of Java seems to be a promising approach since it is already applied in several other occasions [51]. Also, when considering more formal aspects, the use of proof assistant become a powerful tool for specification and verification of programming language semantics, although proof development is time-consuming and difficult to learn, in general.

Cost semantics: Another promising research area consists on cost estimation for programs using functional concepts, like higher-order functions. A study relating asymptotic analysis with *cost semantics* [52] can be explored for further investigations. In a cost semantics, it is possible to specify an abstract cost for a program which can be validated by a provable implementation that transfers the abstract cost to a precise concrete cost on a particular platform [53]. There are several projects dealing with functional languages. It is an opportunity to research on this subject for object-oriented languages for both sequential and parallel executions.

Dependent types: The study of dependent types has already been applied on the object-oriented context. For example, there exist some research about index refinements, or dependent types over a restricted domain, which is combined with the notion of pre- and post-type, giving the programmers the ability to reason about effective computations [54]. In this same area it is possible to investigate about *Liquid Types* (Logically Qualified Data Types), which is precise enough to prove a variety of safety properties [55].

Different extensions: More generally, it is also possible to study about several extensions for Java-like languages formally, such as studies on transactional memory, quantum computation, wild-cards and pattern matching, automatic parallelism, and many others. This kind of research is very important for providing a new well-tested feature with formal proofs that it behaves correctly.

4. Final Remarks

This document has been presented a study on different semantics for Java-like languages, comparing them with each other and showing several projects that were inspired by the proposed semantics. This study was motivated by the need for formalisms in the object-oriented context and the lack of formalization for some new characteristics of the Java language.

Based on this study, it is possible to make a choice among the most used Java-like semantics, considering the needs for different projects. An indirect consequence of this text is a better understanding of distinct ways to formalize the semantics and the type system of programming languages. In addition, the section that discusses research opportunities has pointed out several paths that can be followed by future projects, with help of the provided bibliography.

Author contributions

This paper shows a comparison between the most cited formal semantics for Java, developed by the authors as a result of the last year's research on this field. The Ph.D. student Samuel da Silva Feitosa worked on text writing and literature review. Professor Rodrigo Geraldo Ribeiro contributed to the literature review organization, and Professor André Rauber Du Bois was the research advisor and contributed to text review.

References

[1] TIOBE.COM. *TIOBE Index*. 2017. ⟨https://www.tiobe.com/tiobe-index/⟩. Accessed: 2017-08-28.

[2] LANGPOP.COM. *Programming Language Popularity Index*. 2013. ⟨http://langpop.corger.nl/⟩. Accessed: 2017-08-28.

[3] ORACLE.COM. *The Java Language Specification*. 2015. ⟨http://docs.oracle.com/javase/specs/jls/se8/html/⟩. Accessed: 2017-08-28.

[4] DEBBABI, M.; FOURATI, M. A formal type system for Java. *Journal of Object Technology*, v. 6, n. 8, p. 117–184, 2007.

[5] FILARETTI, D.; MAFFEIS, S. An executable formal semantics of PHP. In: *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. New York, NY, USA: Springer-Verlag New York, Inc., 2014. p. 567–592. ISBN 978-3-662-44201-2. Disponível em: ⟨http://dx.doi.org/10.1007/978-3-662-44202-9_23⟩.

[6] BOGDANAS, D.; ROSU, G. K-Java: A complete semantics of Java. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 50, n. 1, p. 445–456, jan. 2015. ISSN 0362-1340. Disponível em: ⟨http://doi.acm.org/10.1145/2775051.2676982⟩.

[7] MILNER, R. *The definition of standard ML: revised*. [S.l.]: MIT press, 1997.

[8] FLATT, M.; KRISHNAMURTHI, S.; FELLEISEN, M. Classes and mixins. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1998. (POPL '98), p. 171–183. ISBN 0-89791-979-3. Disponível em: ⟨http://doi.acm.org/10.1145/268946.268961⟩.

[9] DROSSOPOULOU, S.; EISENBACH, S. Describing the semantics of java and proving type soundness. In: *Formal*

Syntax and Semantics of Java. London, UK, UK: Springer-Verlag, 1999. p. 41–82. ISBN 3-540-66158-1. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=645580.658817⟩.

[10] IGARASHI, A.; PIERCE, B. C.; WADLER, P. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 23, n. 3, p. 396–450, maio 2001. ISSN 0164-0925. Disponível em: ⟨http://doi.acm.org/10.1145/503502.503505⟩.

[11] KLEIN, G.; NIPKOW, T. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 28, n. 4, p. 619–695, jul. 2006. ISSN 0164-0925. Disponível em: ⟨http://doi.acm.org/10.1145/1146809.1146811⟩.

[12] INC., G. *Google Scholar*. 2018. ⟨https://scholar.google.com.br/⟩. Accessed: 2018-01-12.

[13] ORACLE.COM. *The Java Language Specification*. 2018. ⟨https://docs.oracle.com/javase/specs/jls/se10/html/⟩. Accessed: 2018-05-03.

[14] CIANCARINI, S. C. C. L. P. A reduction semantics for java. 1998.

[15] FARZAN, A.; CHEN, F.; MESEGUER, J. Formal analysis of java programs in javafan. In: *In Proceedings of CAV*. [S.l.: s.n.], 2004. p. 501–505.

[16] STARK, R. F.; BORGER, E.; SCHMID, J. *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001. ISBN 3540420886.

[17] PIERCE, B. C. *Types and Programming Languages*. 1st. ed. [S.l.]: The MIT Press, 2002. ISBN 0262162091, 9780262162098.

[18] BIERMAN, G. M.; PARKINSON, M.; PITTS, A. *MJ: An imperative core calculus for Java and Java with effects*. [S.l.], 2003.

[19] ÖSTLUND, J.; WRIGSTAD, T. Welterweight java. In: *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*. Berlin, Heidelberg: Springer-Verlag, 2010. (TOOLS'10), p. 97–116. ISBN 3-642-13952-3, 978-3-642-13952-9. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=1894386.1894392⟩.

[20] BELLIA, M.; OCCHIUTO, M. E. Properties of java simple closures. *Fundam. Inf.*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 109, n. 3, p. 237–253, ago. 2011. ISSN 0169-2968. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=2361335.2361338⟩.

[21] FEITOSA, S. d. S. et al. A monadic semantics for quantum computing in featherweight java. In: ____. *Programming Languages: 20th Brazilian Symposium, SBLP 2016, Maringá, Brazil, September 22-23, 2016, Proceedings*. Cham: Springer International Publishing, 2016. p. 31–45. ISBN 978-3-319-45279-1. Disponível em: ⟨https://doi.org/10.1007/978-3-319-45279-1_3⟩.

[22] DELAWARE, B.; COOK, W.; BATORY, D. Product lines of theorems. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2011. v. 46, n. 10, p. 595–608.

[23] KUCI, E. et al. A co-contextual type checker for featherweight java (incl. proofs). *arXiv preprint arXiv:1705.05828*, 2017.

[24] FLATT, M.; KRISHNAMURTHI, S.; FELLEISEN, M. A programmer's reduction semantics for classes and mixins. In: *Formal Syntax and Semantics of Java*. London, UK, UK: Springer-Verlag, 1999. p. 241–269. ISBN 3-540-66158-1. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=645580.658808⟩.

[25] POTTIER, F. Hindley-milner elaboration in applicative style: Functional pearl. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, 2014. (ICFP '14), p. 203–212. ISBN 978-1-4503-2873-9. Disponível em: ⟨http://doi.acm.org/10.1145/2628136.2628145⟩.

[26] FLANAGAN, C.; FREUND, S. N. Type-based race detection for java. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, n. 5, p. 219–232, maio 2000. ISSN 0362-1340. Disponível em: ⟨http://doi.acm.org/10.1145/358438.349328⟩.

[27] BOYAPATI, C.; LEE, R.; RINARD, M. A type system for preventing data races and deadlocks in java programs. In: *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. [S.l.: s.n.], 2002. p. 211–230.

[28] WELC, A.; HOSKING, A. L.; JAGANNATHAN, S. Transparently reconciling transactions with locking for java synchronization. In: *Proceedings of the 20th European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2006. (ECOOP'06), p. 148–173. ISBN 3-540-35726-2, 978-3-540-35726-1. Disponível em: ⟨http://dx.doi.org/10.1007/11785477_8⟩.

[29] FINDLER, R. B.; FELLEISEN, M. Contract soundness for object-oriented languages. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 36, n. 11, p. 1–15, out. 2001. ISSN 0362-1340. Disponível em: ⟨http://doi.acm.org/10.1145/504311.504283⟩.

[30] BOYAPATI, C.; LISKOV, B.; SHRIRA, L. Ownership types for object encapsulation. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2003. v. 38, n. 1, p. 213–223.

[31] HAMLEN, K. W.; JONES, M. M.; SRIDHAR, M. Aspect-oriented runtime monitor certification. In: *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer-Verlag, 2012. (TACAS'12), p. 126–140. ISBN 978-3-642-28755-8. Disponível em: ⟨http://dx.doi.org/10.1007/978-3-642-28756-5_10⟩.

[32] MOLDEREZ, T.; JANSSENS, D. Modular reasoning in aspect-oriented languages from a substitution perspective. In: *Transactions on Aspect-Oriented Software Development XII*. [S.l.]: Springer, 2015. p. 3–59.

[33] SYME, D. Proving java type soundness. In: *Formal Syntax and Semantics of Java*. London, UK, UK: Springer-Verlag, 1999. p. 83–118. ISBN 3-540-66158-1. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=645580.658814⟩.

[34] DROSSOPOULOU, S.; EISENBACH, S. Java is type safe - probably. In: *In European Conference On Object Oriented Programming*. [S.l.]: Springer-Verlag, 1997. p. 389–418.

[35] DROSSOPOULOU, S.; EISENBACH, S.; KHURSHID, S. Is the java type system sound? *Theor. Pract. Object Syst.*, John Wiley & Sons, Inc., New York, NY, USA, v. 5, n. 1, p. 3–24, jan. 1999. ISSN 1074-3227. Disponível em: ⟨http://dx.doi.org/10.1002/(SICI)1096-9942(199901/03)5:1⟨3::AID-TAPO2⟩3.0.CO;2-T⟩.

[36] NIPKOW, T.; OHEIMB, D. von. Javalight is type-safe&mdash;definitely. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1998. (POPL ’98), p. 161–170. ISBN 0-89791-979-3. Disponível em: ⟨http://doi.acm.org/10.1145/268946.268960⟩.

[37] MYERS, A. C. *Mostly-static decentralized information flow control*. Tese (Doutorado) — Massachusetts Institute of Technology, 1999.

[38] BODIN, M. et al. A trusted mechanised javascript specification. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2014. v. 49, n. 1, p. 87–100.

[39] BODIN, M. *Certified semantics and analysis of JavaScript*. Tese (Doutorado) — Université Rennes 1, 2016.

[40] FILARETTI, D. An executable formal semantics of php with applications to program analysis. Imperial College London, 2015.

[41] NIPKOW, T. Jinja: Towards a comprehensive formal semantics for a java-like language. In: *IN PROCEEDINGS OF THE MARKTOBERDORF SUMMER SCHOOL. NATO SCIENCE SERIES*. [S.l.]: Press, 2003.

[42] KLEIN, G.; NIPKOW, T. *Jinja is not Java - Archive of Formal Proofs*. 2017. ⟨https://www.isa-afp.org/entries/Jinja.html⟩. Accessed: 2017-09-06.

[43] LOCHBIHLER, A. Type safe nondeterminism-a formal semantics of java threads. *Foundations of Object-Oriented Languages (FOOL 2008)*, 2008.

[44] WASSERRAB, D. et al. An operational semantics and type safety prooffor multiple inheritance in c++. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2006. v. 41, n. 10, p. 345–362.

[45] CHAIEB, A. Proof-producing program analysis. In: *Proceedings of the Third International Conference on Theoretical Aspects of Computing*. Berlin, Heidelberg: Springer-Verlag, 2006. (ICTAC’06), p. 287–301. ISBN 3-540-48815-4, 978-3-540-48815-6. Disponível em: ⟨http://dx.doi.org/10.1007/11921240_20⟩.

[46] PIRKER, M.; SCHAPER, B. M. From jinja bytecode to computation graphs. 2012.

[47] MOSER, G.; SCHAPER, M. A complexity preserving transformation from jinja bytecode to rewrite systems. *arXiv preprint arXiv:1204.1568*, 2012.

[48] KÜSTERS, R. et al. A hybrid approach for proving noninterference of java programs. In: IEEE. *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*. [S.l.], 2015. p. 305–319.

[49] CLAUDEL, B.; SABAH, Q.; STEFANI, J.-B. Simple isolation for an actor abstract machine. In: SPRINGER. *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. [S.l.], 2015. p. 213–227.

[50] MOGGI, E. Computational lambda-calculus and monads. In: *Proceedings of the Fourth Annual Symposium on Logic in computer science*. [S.l.]: IEEE Press, 1989. p. 14–23. ISBN 0-8186-1954-6.

[51] CLAESSEN, K.; HUGHES, J. QuickCheck: A lightweight tool for random testing of Haskell programs. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, 2000. (ICFP ’00), p. 268–279. ISBN 1-58113-202-6. Disponível em: ⟨http://doi.acm.org/10.1145/351240.351266⟩.

[52] BLELLOCH, G. E.; HARPER, R. Cache and i/o efficent functional algorithms. In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2013. (POPL ’13), p. 39–50. ISBN 978-1-4503-1832-7. Disponível em: ⟨http://doi.acm.org/10.1145/2429069.2429077⟩.

[53] HARPER, R. *Yet Another Reason Not To Be Lazy Or Imperative*. 2012. *Available at ⟨https://existentialtype.wordpress.com/2012/08/26/yet-another-reason-not-to-be-lazy-or-imperative/⟩*. Disponível em: ⟨https://existentialtype.wordpress.com/2012/08/26/yet-another-reason-not-to-be-lazy-or-imperative/⟩.

[54] CAMPOS, J.; VASCONCELOS, V. T. Imperative objects with dependent types. In: *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*. New York, NY, USA: ACM, 2015. (FTfJP ’15), p. 2:1–2:6. ISBN 978-1-4503-3656-7. Disponível em: ⟨http://doi.acm.org/10.1145/2786536.2786538⟩.

[55] RONDON, P. M.; KAWAGUCI, M.; JHALA, R. Liquid types. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2008. (PLDI ’08), p. 159–169. ISBN 978-1-59593-860-2. Disponível em: ⟨http://doi.acm.org/10.1145/1375581.1375602⟩.