Formal Semantics for Java-like Languages and Research Opportunities

PhD Student: Samuel da Silva Feitosa[1]
Advisor: Andre Rauber Du Bois[1]
Co-advisor: Rodrigo Geraldo Ribeiro [2]

[1]Universidade Federal de Pelotas (UFPEL)

{samuel.feitosa,dubois} [at] inf.ufpel.edu.br

[2]Universidade Federal de Ouro Preto (UFOP)

{rodrigo} [at] decsi.ufop.br

Abstract. *The objective of this document is twofold: first we discuss the state of art on Java-like semantics, focusing on those that provide formal specification using operational semantics (big-step or small-step), studying in detail the most cited projects and presenting some derived works that extend the originals aggregating useful features. Also, we filter our research for those that provide some insights in type-safety proofs. Furthermore, we provide a comparison between the most used projects in order to show which functionalities are covered in such projects. Second, our effort is focused towards the research opportunities in this area, showing possible ideas that can be implemented using the previously presented projects for studying features of object oriented languages, and pointing for some possibilities to explore in future researches.*

1. Introduction

Nowadays, Java is one of the most popular programming languages [tiobe.com 2017, langpop.com 2013]. 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 time, adding features and programming facilities in its new versions. The latest version is Java 8, which brings new features as lambda expressions, method references, and functional interfaces offering a programming model that fuses the object-oriented and functional styles [oracle.com 2015].

Considering the growth in adoption of the 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 [Debbabi and Fourati 2007]. Normally, this task is hard to be accomplished due to computability issues considering certain problem sizes. For overcoming this situation it is possible to model formal subsets of the problem applying certain degree of abstraction, using only properties of interest, facilitating the understanding of the problem and also allowing the use of automatic tools [Filaretti and Maffeis 2014].

Therefore, an important research area concerns the formal semantics of languages and type-system specification, which enables the verification of a problem consistency, allowing making formal proofs, and establishing fundamental properties on systems. Besides, 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 can't state that a program meets its specification, a type system is sound, or that a compiler or an interpreter is correct [Bogdanas and Roşu 2015].

Some programming languages, such as ML [Milner 1997], already come with a formal specification. Nevertheless, official specification of languages are usually made in English prose with varying degrees of rigor and precision [Filaretti and Maffeis 2014]. On the other hand, researchers are making efforts to study formally such languages. Indeed, there is a large body of literature on formal models of Java-like languages [Flatt et al. 1998, Drossopoulou and Eisenbach 1999, Igarashi et al. 2001, Klein and Nipkow 2006], 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 [Bogdanas and Roşu 2015], but without contemplating newest important features. It is important to note that standard formalisms for Java-like languages greatly simplify investigations of novel programming constructs, and helps on identification of errors and misunderstandings. Thus, these formalizations can be explored for future researches.

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

2. Formal Semantics for Java

There exist several works on the formalization of the Java language, in particular on the proof of soundness of the Java type system. This section presents some of them, discussing their completeness and compliance with the official specification of Java, and comparing them with each other.

2.1. Featherweight Java

Featherweight Java (FJ) [Igarashi et al. 2001], 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 *main* method of Java.

FJ has a similar relation with Java, like λ-Calculus has with Haskell. It offers similar operations, providing classes, methods, attributes, inheritance and dynamic casts with semantics close to Java's. The Featheweight 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 [Pierce 2002]. 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 [Igarashi et al. 2001].

Because the language does not allow side effects, it is possible to formalize the evaluation just using the FJ syntax, without the need of 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 expresses 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.).

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.

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) [Igarashi et al. 2001], 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 $[\overline{\mathrm{x}} \mapsto \overline{\mathrm{u}}, \mathrm{this} \mapsto \mathrm{new\ C}(\overline{\mathrm{v}})]\mathrm{t}_0$ for the result of substituting x_1 by u_1,...,x_n by u_n, and the keyword *this* by "new C($\overline{\mathrm{v}}$)" in 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.

Evaluation Rules

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

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

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

Figure 2. Evaluation rules for FJ.

The typing rules for expressions are in Figure 3. The authors use the meta-variable Γ to denote environments,i.e. finite mappings from variables to types. Sometimes it is used notation $\overline{\mathrm{x}}$: $\overline{\mathrm{C}}$ to denote environments. The typing judgment for expressions has the form $\Gamma \vdash e : C$, read as "in the environment Γ, expression e has type C". The typing rules are *syntax directed*[1], with one rule for each form of expression, except for casts. Most of the typing rules are straightforward adaptations of the rules of original Java: the rule (T-Var) checks if there is a type *C* such that *x:* $C \in \Gamma$ and returns *C* as *x*'s type; rule (T-Field) uses the function $fields$ to obtain all fields in class C_0 which is *e*'s 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, one for $upcasts$, $downcasts$ 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 and classes were omitted here, but can be found in the original FJ paper [Igarashi et al. 2001].

An important contribution of FJ is the proof of type soundness, stating that "well-typed programs do not get stuck", made through the next Theorem:

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 [Igarashi et al. 2001].

[1]A syntax-directed system is one in which each typing judgment has a unique derivation, determined entirely by the syntactic structure of the term in question.

Expression Typing

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

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

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

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

Figure 3. Typing rules for FJ.

2.1.1. 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. In the FJ paper itself, it was presented an extension that models *generics* [Igarashi et al. 2001]. Middleweight Java [Bierman et al. 2003] and Welterweight Java [Östlund and Wrigstad 2010] 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.

For being a small subset of Java, novel constructions for object-oriented languages can be embedded in this language, for formalizing type-safety. There exists several examples, and here we cite just a few. A proposal for closures [Bellia and Occhiuto 2011] was presented years before the release of Java 8, quantum investigations in the OO context [Feitosa et al. 2016], a Coq formalization of FJ for studying product lines of theorems [Delaware et al. 2011], and more recently an study addressing compositional and incremental type checking for object-oriented programming languages through co-contextual typing rules [Kuci et al. 2017].

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. It is important to note that this formalism is very concise and easy to fit into a regular paper, and seems to be easier to understand than the others calculus that follow this section.

2.2. ClassicJava

ClassicJava [Flatt et al. 1998, Flatt et al. 1999] is a small subset of sequential Java proposed by Flatt, Krishnamurthi and Felleisen. To model its type structure, the authors use type elaborations, where 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 method signatures. 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 represent casts (*view* operator) and assignments (*let* operator). Figure 4 shows the formal syntax of ClassicJava.

Syntax

$P ::=$	program specification
$\quad defn^* \ e$	
$defn ::=$	class and interface declarations
$\quad class\ c\ extends\ c\ implements\ i^*$	$\{field^*\ meth^*\}$
$field ::=$	field statement
$\quad t\ fd$	
$meth ::=$	method declarations
$\quad t\ md(arg^*)\ \{body\}$	
$arg ::=$	argument list
$\quad tvar$	
$body ::=$	method body declarations
$\quad e \mid abstract$	
$e ::=$	expressions
$\quad new\ c$	instancing a class with name c or $Object$
$\quad var$	a variable name or $this$
$\quad null$	$null$ value
$\quad e :\ c.fd$	field access
$\quad e :\ c.fd = e$	field assignment
$\quad e.md(e^*)$	method invocation
$\quad super \equiv this :\ c.md(e^*)$	method invocation inside a class
$\quad view\ t\ e$	cast
$\quad let\ var = e\ in\ e$	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 uniqueness 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 $\Subset_P^c$ (overloaded) captures the field and method declarations of P, and so on. The complete list of auxiliary definitions can be found in the original paper [Flatt et al. 1998].

The operational semantics for ClassicJava is defined as a contextual rewriting system on pairs of expressions and stores. A store $\mathcal{S}$ is a mapping from *objects* to class-tagged field records. A field record $\mathcal{F}$ is a mapping from elaborated field names to values. The evaluation rules for this language are presented in Figure 5.

Evaluation Rules

$$e = ... \mid object \qquad\qquad E = [] \mid E : c.fd \mid E : c.fd = e \mid v : c.fd = E$$

$$v = object \mid null \qquad\qquad \mid E.md(e...) \mid v.md(v...E\ e...) \mid super \equiv v :\ c.md(v...E\ e...) \mid view\ t\ E \mid let\ var = E\ in\ e$$

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

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

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

$$P \vdash \langle E[object.md(v_1, ..., v_n)], \mathcal{S}\rangle \hookrightarrow \langle E[e[object/this, v_1/var_1, ..., v_n/var_n]], \mathcal{S}\rangle\ where\ \mathcal{S}(object) = \langle c, \mathcal{F}\rangle\ and\ \langle md, (t_1...t_n) \longrightarrow t), (var_1...var_n), e\rangle \in_P^c\ c \qquad [call]$$

$$P \vdash \langle E[super \equiv object : c'.md(v_1, ..., v_n)], \mathcal{S}\rangle \hookrightarrow \langle E[e[object/this, v_1/var_1, ..., v_n/var_n]], \mathcal{S}\rangle\ where\ \langle md, (t_1...t_n) \longrightarrow t), (var_1...var_n), e\rangle \in_P^c\ c' \qquad [super]$$

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

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

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

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

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

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

Figure 5. Evaluation rules for ClassicJava.

By looking, for example, in the *call* rule in Figure 5 one can note that it invokes a method by rewriting the method call expression by the body of the invoked method, syntactically replacing argument variables in this expression with the supplied argument values. The other rules can be understood in a similar way.

The type elaborations 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 [Flatt et al. 1998]. 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 the type t. A complete proof is available in an extended version of the original ClassicJava paper [Flatt et al. 1999].

2.2.1. 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 analysis for multi-threaded Java programs, through a small multi-threaded subset of Java, which extends the mentioned formalization [Flanagan and Freund 2000]. 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 [Boyapati et al. 2002]. Also in this research area, among others, a core language to examine notions of safety with respect to transactions and mutual exclusion was proposed [Welc et al. 2006].

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 an object-oriented world from a foundational perspective is Contract Java [Findler and Felleisen 2001]. In another project, it was proposed a type system to work with ownership types, which provide a statically enforceable way of specifying object encapsulation and enable local reasoning about correctness in object-oriented languages [Boyapati et al. 2003]. And recently, some works for aspect oriented languages were done [Hamlen et al. 2012, Molderez and Janssens 2015].

2.3. 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 an outline of the type soundness proof [Drossopoulou and Eisenbach 1997, Drossopoulou et al. 1999, Drossopoulou and Eisenbach 1999]. 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 type inference to describe compile-time type checking. They also prove that program execution preserves the type up to the subclass/subinterface relationship [Drossopoulou and Eisenbach 1999]. 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$_S$.

Syntax

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

Figure 6. Syntactic definitions for ClassicJava.

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 consider 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 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 other Java calculus, this proposal also models the class hierarchy, implementing the sub-typing relationship through the $\sqsubseteq$ operator. Moreover, it also describes 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 the lack of space, this grammar 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. For example, the class *Phil* extends *Object*, and has two (overoaded) methods called *think*. The first one receives a parameter y of type *Phil*, and the second also receives a parameter y with type *FrPhil*.

```
p_s = 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 [Drossopoulou and Eisenbach 1999]. 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 typechecking) and the program p being executed [Syme 1999].

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. Below is a sample of these rules, showing a sequence of statements:

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

There are 11 rules for dealing with 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 of assignment that generates an exception is presented in the folowing rule:

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

In this calculus, a term is *ground* if it is in normal form, i.e. no further reduction can be made. Several rules were ommited from this text due to lack of space, such as 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 [Drossopoulou and Eisenbach 1997]. For short, 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.

2.3.1. 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 [Syme 1999].

The language Java$_{light}$ is directly related to Java$_S$, although it uses a different approach in the representation of programs and use of 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 [Nipkow and von Oheimb 1998].

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 [Myers 1999]. There are also some mechanized specifications for other languages, such as JavaScript and PHP, that were inspired by this work [Bodin et al. 2014, Bodin 2016, Filaretti 2015].

2.4. Java$_{light}$ and Jinja

Jinja [Nipkow 2003, Klein and Nipkow 2006] 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$_{light}$ [Nipkow and von Oheimb 1998], enhancing the treatment of exceptions.

In contrast to others formalizations, they present a big and a small step semantics, which are independent of the type system, showing their equivalence. They also present the type system, a definite initialization analysis and type safety of the small step semantics. Additionally, the whole development has been carried out in the theorem prover Isabelle/HOL [Klein and Nipkow 2017].

The abstract syntax of programs is given 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 [Klein and Nipkow 2006].

Syntax

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

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: 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 possible syntax for 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 to 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 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.

Evaluation Rules

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

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

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

Figure 8. Partial big-step semantics for Jinja.

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 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 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 of the parameters. The complete set of rules can be found at the original papers [Nipkow 2003, Klein and Nipkow 2006].

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 about the two proposed semantics, is that in the small-step they present *subexpression reduction*, which essentially describes 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 [Klein and Nipkow 2006]. These rules were omitted from this text due to lack of space.

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 [Klein and Nipkow 2017].

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

Jinja has become the basis for further investigations concerning Java-like languages. For example, it was possible to work on a framework to transform a single-threaded operational semantics into a semantics with interleaved execution of threads, as an extension of Jinja [Lochbihler 2008]. Jinja was also used as starting point for a formalization of multiple inheritance in C++ [Wasserrab et al. 2006]. Proof-synthesis algorithms for flow chart languages were also derived from this formalization [Chaieb 2006].

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 [Pirker and Schaper 2012]. In this same research area, it was revisited some know transformations from Jinja bytecode to rewrite systems from the viewpoint of runtime complexity, where the authors proposed an alternative representation of Jinja bytecode executions as computational graphs obtaining a novel representation [Moser and Schaper 2012]. 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 [Küsters et al. 2015]. Another interesting project, concerns about a Coq formal specification of the Siaam model, built over the Jinja specification [Claudel et al. 2015].

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 researches.

2.5. A Comparison Between the Most Used Semantics

As already mentioned, there exist several works on the formalization of the Java language. It was chosen to work with the most cited projects that models Java semantics according to Google Scholar. Using this criteria, Featherweight Java is the most cited, with almost 900 citations. Classic Java has approximately 500 quotes. Java$_S$ and Jinja currently present between 300 and 400 quotes each one. Table 1 shows which features are modeled in each of the discussed formal languages according to their completeness and compliance with the official specification of Java. Features of original Java that are not modeled in any of the languages were suppressed from this table.

Feature	**FJ**	**CJ**	**JS**	**JJ**
Primitive Types	○	○	●	◐
Integer	○	○	●	●
Boolean	○	○	●	●
String literal	○	○	●	○
Basic Statements	○	◐	●	●
Assignment	○	●	●	●
Conditionals	○	○	●	●
Loops	○	○	●	●
Try-catch-finally	○	○	●	●
Basic OOP	◐	◐	◐	◐
Classes and inheritance	●	●	●	●
Interfaces	○	●	○	○
Casts	●	●	○	●
Constructors, super and this	●	◐	○	◐
Polymorphism and overriding	●	●	○	●
Overloading and static methods	○	○	○	○
Generics	●	○	○	○
Arrays	○	○	●	○
Semantics, Type System and Safety Proofs	◐	◐	◐	●
Big-step semantics	○	○	○	●
Small-step semantics	●	●	●	●
Progress and preservation	●	●	●	●
Use of proof assistant	○	○	◐	●

Support level: ● = Full ◐ = Partial ○ = None
FJ is Featherweight Java, CJ is ClassicJava, JS is Java$_S$, JJ is Jinja.

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

By this table, we can note that some of these formalisms (JS, JJ) are concerning about basic features of a programming language, like primitive types and basic statements, where others works (FJ, CJ) are focusing in features of the object-oriented paradigm, as classes, inheritance, polymorphism, etc. Furthermore, there is only one project (JJ) that presents machine-checked proofs originally. The other semantics (JS) was cited with partial support, because the proofs were made as an extension of the original paper by other author [Syme 1999]. This table can be helpful to make a choice between the presented formal languages according to the need of the project.

3. Research Opportunities

Traditional programming languages (imperative) are incorporating functional features among the years, making the code writing more and more multi-paradigm. As mentioned previously, the concept of lambda expressions was embedded in version 8 of Java language [oracle.com 2015], which allows treating functions as arguments, making the language more expressive. Other languages, as C++ [Jarvi and Freeman 2010] and C# [Deitel 2008] also permit similar usage. 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, etc. As far as the author knows, there is no formalization of these new aspects exactly in the way they work in Java, such as lambda expressions, method references, and functional interfaces, and it can also be explored to work on.

Having lambda expressions in the language it is possible to model and explore the concept of monad, also coming from functional languages, where it is possible to represent computation in terms of values and sequence of computations. This concept, that was proposed by Moggi [Moggi 1989] and comes from category theory, is very accepted in the functional languages community and is used for modeling lists, parsers, exceptions, transactions, etc. Although it is possible to use lambda expressions and consequently monads in Java (and similar languages), there are no formal description of these aspects. When considering formal aspects, the use of proof assistants become powerful tools for specification and verification, although proof development is time consuming and complicated in general.

Another promising research area consists on cost estimation for programs using functional concepts, like higher-order functions. Usually, asymptotic analysis is used for estimating costs in imperative languages. Therefore, when dealing with functional constructors, the cost analysis is more complicated. With this purpose in mind, it was proposed the *cost semantics* [Blelloch and Harper 2013], that allows the specification of an abstract cost for a program that is validated by a provable implementation that transfers the abstract cost to a precise concrete cost on a particular platform [Harper 2012]. There are several works dealing with functional languages. It is an opportunity to research in this subject for object-oriented languages for both sequential and parallel executions.

The study of dependent types has been applied on the object-oriented context. For example, there is some research about index refinements, or dependent types over a restricted domain, which are combined with the notion of pre- and post-type, giving programmers the ability to reason about effectful computations [Campos and Vasconcelos 2015]. 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 [Rondon et al. 2008].

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

4. Final Remarks

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

Through this study, it is possible to make a choice between the most used Java-like semantics, considering the needs for different projects. As an indirect consequence of this text, is a better understanding about distinct ways to formalize the semantics and the type system of a programming language. In addition, the section that discusses about research opportunities has pointed out several paths that can be followed by this project, and for future projects, with help of the provided bibliographic references.

References

[Bellia and Occhiuto 2011] Bellia, M. and Occhiuto, M. E. (2011). Properties of java simple closures. *Fundam. Inf.*, 109(3):237–253.

[Bierman et al. 2003] Bierman, G. M., Parkinson, M., and Pitts, A. (2003). Mj: An imperative core calculus for java and java with effects. Technical report, University of Cambridge, Computer Laboratory.

[Blelloch and Harper 2013] Blelloch, G. E. and Harper, R. (2013). Cache and i/o efficent functional algorithms. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 39–50, New York, NY, USA. ACM.

[Bodin 2016] Bodin, M. (2016). *Certified semantics and analysis of JavaScript*. PhD thesis, Université Rennes 1.

[Bodin et al. 2014] Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., and Smith, G. (2014). A trusted mechanised javascript specification. In *ACM SIGPLAN Notices*, volume 49, pages 87–100. ACM.

[Bogdanas and Roşu 2015] Bogdanas, D. and Roşu, G. (2015). K-Java: A complete semantics of Java. *SIGPLAN Not.*, 50(1):445–456.

[Boyapati et al. 2002] Boyapati, C., Lee, R., and Rinard, M. (2002). 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*, pages 211–230.

[Boyapati et al. 2003] Boyapati, C., Liskov, B., and Shrira, L. (2003). Ownership types for object encapsulation. In *ACM SIGPLAN Notices*, volume 38, pages 213–223. ACM.

[Campos and Vasconcelos 2015] Campos, J. and Vasconcelos, V. T. (2015). Imperative objects with dependent types. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*, FTfJP '15, pages 2:1–2:6, New York, NY, USA. ACM.

[Chaieb 2006] Chaieb, A. (2006). Proof-producing program analysis. In *Proceedings of the Third International Conference on Theoretical Aspects of Computing*, ICTAC'06, pages 287–301, Berlin, Heidelberg. Springer-Verlag.

[Claudel et al. 2015] Claudel, B., Sabah, Q., and Stefani, J.-B. (2015). Simple isolation for an actor abstract machine. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 213–227. Springer.

[Debbabi and Fourati 2007] Debbabi, M. and Fourati, M. (2007). A formal type system for Java. *Journal of Object Technology*, 6(8):117–184.

[Deitel 2008] Deitel, H. (2008). *Visual C# 2008 How to Program.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3 edition.

[Delaware et al. 2011] Delaware, B., Cook, W., and Batory, D. (2011). Product lines of theorems. In *ACM SIGPLAN Notices*, volume 46, pages 595–608. ACM.

[Drossopoulou and Eisenbach 1997] Drossopoulou, S. and Eisenbach, S. (1997). Java is type safe - probably. In *In European Conference On Object Oriented Programming*, pages 389–418. Springer-Verlag.

[Drossopoulou and Eisenbach 1999] Drossopoulou, S. and Eisenbach, S. (1999). Describing the semantics of java and proving type soundness. In *Formal Syntax and Semantics of Java*, pages 41–82, London, UK, UK. Springer-Verlag.

[Drossopoulou et al. 1999] Drossopoulou, S., Eisenbach, S., and Khurshid, S. (1999). Is the java type system sound? *Theor. Pract. Object Syst.*, 5(1):3–24.

[Feitosa et al. 2016] Feitosa, S. d. S., Vizzotto, J. K., Piveta, E. K., and Du Bois, A. R. (2016). *A Monadic Semantics for Quantum Computing in Featherweight Java*, pages 31–45. Springer International Publishing, Cham.

[Filaretti 2015] Filaretti, D. (2015). An executable formal semantics of php with applications to program analysis.

[Filaretti and Maffeis 2014] Filaretti, D. and Maffeis, S. (2014). An executable formal semantics of PHP. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pages 567–592, New York, NY, USA. Springer-Verlag New York, Inc.

[Findler and Felleisen 2001] Findler, R. B. and Felleisen, M. (2001). Contract soundness for object-oriented languages. *SIGPLAN Not.*, 36(11):1–15.

[Flanagan and Freund 2000] Flanagan, C. and Freund, S. N. (2000). Type-based race detection for java. *SIGPLAN Not.*, 35(5):219–232.

[Flatt et al. 1998] Flatt, M., Krishnamurthi, S., and Felleisen, M. (1998). Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 171–183, New York, NY, USA. ACM.

[Flatt et al. 1999] Flatt, M., Krishnamurthi, S., and Felleisen, M. (1999). A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, UK. Springer-Verlag.

[Hamlen et al. 2012] Hamlen, K. W., Jones, M. M., and Sridhar, M. (2012). Aspect-oriented runtime monitor certification. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 126–140, Berlin, Heidelberg. Springer-Verlag.

[Harper 2012] Harper, R. (2012). Yet another reason not to be lazy or imperative. *Available at `https://existentialtype.wordpress.com/2012/08/26/yet-another-reason-not-to-be-lazy-or-imperative/`.*

[Igarashi et al. 2001] Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450.

[Jarvi and Freeman 2010] Jarvi, J. and Freeman, J. (2010). C++ lambda expressions and closures. *Science of Computer Programming*, 75(9):762 – 772. Special Issue on Object-Oriented Programming Languages and Systems (OOPS 2008), A Special Track at the 23rd {ACM} Symposium on Applied Computing.

[Klein and Nipkow 2006] Klein, G. and Nipkow, T. (2006). A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695.

[Klein and Nipkow 2017] Klein, G. and Nipkow, T. (2017). Jinja is not Java - archive of formal proofs. `https://www.isa-afp.org/entries/Jinja.html`. Accessed: 2017-09-06.

[Kuci et al. 2017] Kuci, E., Erdweg, S., Bračevac, O., Bejleri, A., and Mezini, M. (2017). A co-contextual type checker for featherweight java (incl. proofs). *arXiv preprint arXiv:1705.05828.*

[Küsters et al. 2015] Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., and Mohr, M. (2015). A hybrid approach for proving noninterference of java programs. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pages 305–319. IEEE.

[langpop.com 2013] langpop.com (2013). Programming language popularity index. `http://langpop.corger.nl/`. Accessed: 2017-08-28.

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

[Milner 1997] Milner, R. (1997). *The definition of standard ML: revised.* MIT press.

[Moggi 1989] Moggi, E. (1989). Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23. IEEE Press.

[Molderez and Janssens 2015] Molderez, T. and Janssens, D. (2015). Modular reasoning in aspect-oriented languages from a substitution perspective. In *Transactions on Aspect-Oriented Software Development XII*, pages 3–59. Springer.

[Moser and Schaper 2012] Moser, G. and Schaper, M. (2012). A complexity preserving transformation from jinja bytecode to rewrite systems. *arXiv preprint arXiv:1204.1568.*

[Myers 1999] Myers, A. C. (1999). *Mostly-static decentralized information flow control.* PhD thesis, Massachusetts Institute of Technology.

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

[Nipkow and von Oheimb 1998] Nipkow, T. and von Oheimb, D. (1998). Javalight is type-safe&mdash;definitely. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 161–170, New York, NY, USA. ACM.

[oracle.com 2015] oracle.com (2015). The Java language specification. `http://docs.oracle.com/javase/specs/jls/se8/html/`. Accessed: 2017-08-28.

[Östlund and Wrigstad 2010] Östlund, J. and Wrigstad, T. (2010). Welterweight java. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 97–116, Berlin, Heidelberg. Springer-Verlag.

[Pierce 2002] Pierce, B. C. (2002). *Types and Programming Languages*. The MIT Press, 1st edition.

[Pirker and Schaper 2012] Pirker, M. and Schaper, B. M. (2012). From jinja bytecode to computation graphs.

[Rondon et al. 2008] Rondon, P. M., Kawaguci, M., and Jhala, R. (2008). Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA. ACM.

[Syme 1999] Syme, D. (1999). Proving java type soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, London, UK, UK. Springer-Verlag.

[tiobe.com 2017] tiobe.com (2017). TIOBE Index. `https://www.tiobe.com/tiobe-index/`. Accessed: 2017-08-28.

[Wasserrab et al. 2006] Wasserrab, D., Nipkow, T., Snelting, G., and Tip, F. (2006). An operational semantics and type safety prooffor multiple inheritance in c++. In *ACM SIGPLAN Notices*, volume 41, pages 345–362. ACM.

[Welc et al. 2006] Welc, A., Hosking, A. L., and Jagannathan, S. (2006). Transparently reconciling transactions with locking for java synchronization. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 148–173, Berlin, Heidelberg. Springer-Verlag.