Property-Based Testing for Lambda Expressions Semantics in Featherweight Java

Samuel da Silva Feitosa
Federal University of Pelotas
samuel.feitosa@inf.ufpel.edu.br

Rodrigo Geraldo Ribeiro
Federal University of Ouro Preto
rodrigo@decsi.ufop.br

Andre Rauber Du Bois
Federal University of Pelotas
dubois@inf.ufpel.edu.br

Abstract

The release of Java 8 represents one of the most significant updates to the Java language since its inception. The addition of λ-expressions allows the treatment of code as data in a compact way, improving the language expressivity. This paper addresses the problem of defining rigorous semantics for new features of Java, such as λ-expressions and default methods, using Featherweight Java (FJ), a well-known object-oriented calculus. To accomplish this task, we embed the formalization of these new features in two different semantics, checking them for safety properties using QuickCheck, a property-based testing library for Haskell.

Keywords

Property-based testing, λ-expressions, Featherweight Java

ACM Reference Format:
Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and Andre Rauber Du Bois. 1997. Property-Based Testing for Lambda Expressions Semantics in Featherweight Java. In *Proceedings of Brazilian Symposium on Programming Languages (SBLP'2018).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.475/123_4

1 Introduction

Nowadays, Java is one of the most popular programming languages [19]. It is a general-purpose, concurrent, strongly typed, class-based object-oriented language. Since its release in 1995 by Sun Microsystems, and acquisition by Oracle Corporation, Java has been evolving over time, adding features and programming facilities in its new versions. In a recent major release of Java, new features such as lambda expressions, method references, and functional interfaces, were added to the core language, offering a programming model that fuses the object-oriented and functional paradigms [11].

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 [8]. Normally, this task is hard to be accomplished, however it is possible to model formal subsets of the problem applying a certain degree of abstraction, using only properties of interest, facilitating the understanding and also allowing the use of automatic tools.

Therefore, an important research area concerns the formal semantics of languages and type-system specification, which enables the verification of a problem consistency, allowing formal proofs, and establishing program properties. Besides, solutions can be machine checked providing a degree of confidence that cannot be reached using informal approaches.

In this context, this work provides the description of two different semantics for λ-expressions and default methods, extending Featherweight Java [12], a small core calculus with a rigorous semantic definition of the main core aspects of Java. We work on two different semantics because λ-expressions types are lost during reduction steps [2]. One of those semantics uses type elaboration [18] to annotate all the λ-expression types during the type checking phase, producing new expressions in each rule, releasing the evaluation rules to deal with types. The other annotates types at the moment they are needed, making the evaluation independent of type-checking.

The motivations for using FJ as a starting point is twofold: first, it is very compact, so we can focus on the essential aspects of our extensions. The minimal syntax, typing rules, and operational semantics fit well for studying the new aspects of Java 8. Second, we are interested in the formal definition of FJ, which allows modeling and proving properties of programs. As far as we know, there is no implemented[1] versions of a formal calculus for λ-expressions in the context of Java. This work aims to fill this gap by specifying a standard small-step operational semantics and using property-based testing to check if safety properties are satisfied by using randomly generated programs. We are aware that using automated testing is not sufficient to ensure correctness, but it can expose bugs before using more formal approaches, like formalizing the semantics in a proof assistant.

Specifically, we made the following contributions:

- We define two small-step reduction and typing rules for the Java 8 features: λ-expressions and default methods. In addition, we show informal (non-mechanized) proofs for type soundness, and we implement interpreters for each semantics.
- We define a type-directed heuristic for constructing random programs. We conjecture that our heuristic is sound with respect to FJ type system, i.e. it generates only well-typed programs.

[1] The source-code for our interpreters and the test suite is available at: https://github.com/fjpub/fj-lam.

SBLP'2018, Sep. 2018, São Carlos, SP, Brazil

ACM ISBN 123-4567-24-567/08/06...$15.00
https://doi.org/10.475/123_4

- We use QuickCheck to verify the following properties: our program generators are only producing well-formed class tables and well-typed expressions; the type soundness properties hold for the proposed semantics; and the two semantics are equivalent.

The remainder of this text is organized as follows: Section 2 summarizes the FJ proposal. Section 3 presents the operational semantics for λ-expressions and default methods. Section 4 shows how we modeled the program generators and presents the results of testing our semantics with QuickCheck. Section 5 discusses some related work. Finally, we present the final remarks in Section 6.

2 Featherweight Java

Featherweight Java (FJ) [12] 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 the declaration of a set of classes and an expression to be evaluated, that corresponds to the Java's main method.

FJ is to Java what λ-calculus is to Haskell. It offers similar operations, providing classes, methods, attributes, inheritance and dynamic casts with semantics close to Java's. The Featherweight Java project favors simplicity over expressivity and offers only five ways to create terms: object creation, method invocation, attribute access, casting and variables [12]. 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 {
  Object fst;
  Object snd;
  Pair(X fst, Y snd) {
    super();
    this.fst=fst;
    this.snd=snd;
  }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
}
```

In the following example we can see two different expressions: `new A()`, `new B()`, and `new Pair(...)` are *object constructors*, and `.setfst(...)` refers to a *method invocation.*

```
new Pair(new A(),new B()).setfst(new B());
```

FJ semantics provides a purely functional view without side effects. In other words, attributes in memory are not affected by object operations. 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 [12]. As 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.

This calculus 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. Besides the rules for evaluation and type-checking, the authors present proofs of type soundness for FJ as another important contribution. The rules for FJ augmented with the semantics of λ-expressions, appear in the next section.

3 The Semantics for λ-Expressions and Default Methods

In the release of Java Development Kit version 8 (JDK 8), *λ-expressions* and *default methods* were added in the kernel of the Java language. The λ-expressions enable the programmer to treat functions as a method argument, or code as data. They provide a clear and concise way to instantiate one method interface's using an expression and facilitates programming in a functional style. Particularly, they are anonymous methods (methods without names) used to implement a method defined by a functional interface[2]. Default methods allow new functionality to be added to the interfaces of libraries assuring binary compatibility with older versions.

In order to study rigorously object-oriented languages such as Java, C++ or C#, a common practice is to define lightweight fragments, which are sufficiently small to facilitate formal proofs of key properties. This section addresses the problem of defining a rigorous semantics for λ-expressions and default methods, by extending the FJ calculus adding the syntax and semantics for these features. This technique is well known and has already been used in several projects to study interesting features and properties related to object-oriented languages [1–3, 9, 14, 16]. As we have defined two different approaches for dealing with λ-expressions and default methods, at the first moment we present the syntax and auxiliary definitions, which are common for both approaches. Then, we present the first version, where we use type elaboration [18] to annotate the types for λ-expressions producing, as a result, an annotated program. After that, we show the second approach, where both type system and reduction rules annotate types for λ-expressions whenever it is necessary.

3.1 Syntax and Auxiliary Functions

The abstract syntax of FJ augmented with interfaces, λ-expressions and default methods is given in Figure 1, where `T` represents type declarations, `L` and `P` express classes and interfaces, `K` defines constructors, `S` stands for signatures, `M` for methods, and `e` refers to the possible expressions. The metavariables `A`, `B`, `C`, `D`, and `E` can be used to represent class names, `F`, `G`, `H`, `I`, and `J` range over interface names, `T`, `U`,

[2] A functional interface is an interface that contains one and only one abstract method.

and `V` represent generic names for classes or interfaces, `f` and `g` range over field names, `m` ranges over method names, `x` and `y` range over variables, `d` and `e` range over expressions. Throughout this paper, we write $\overline{T}$ as shorthand for a possibly empty sequence $T_1, ..., T_n$ (similarly for $\overline{C}$, $\overline{f}$, $\overline{x}$, etc.). An empty sequence is denoted by $\bullet$, and the length of a sequence $\bar{\mathtt{x}}$ is written $\#\bar{\mathtt{x}}$. We use Γ to represent an environment, which is a finite mapping from variables to types, written $\overline{x} : \overline{T}$, and we let $\Gamma(x)$ denote the type T such that x: $T \in \Gamma$. We slightly abuse notation by using set operators on sequences.

Syntax

$T ::=$	type definitions
$C \mid I$	
$L ::=$	class declarations
class C extends C implements $\overline{I}$ $\{\overline{T}\ \overline{f}; K\ \overline{M}\}$	
$P ::=$	interface declarations
interface I extends $\overline{I}$ $\{\overline{S};$ default $\overline{M}\}$	
$K ::=$	constructor declarations
$C(\overline{T}\ \overline{f})$ $\{$super$(\overline{f});$ this.$\overline{f} = \overline{f};\}$	
$S ::=$	signature declarations
T m$(\overline{T}\ \overline{x})$	
$M ::=$	method declarations
S $\{$ return $e;\}$	
$e ::=$	expressions
x	variable
$e.f$	field access
$e.$m$(\overline{e})$	method invocation
new $C(\overline{e})$	object creation
$(T)\ e$	cast
$(\overline{T}\ \overline{x}) \rightarrow e$	λ-expression

Figure 1: Syntactic definitions for FJ augmented with λ-expressions.

The differences from original FJ [12] were given firstly by the introduction of *interface declarations*, where `interface` I `extends` $\overline{I}$ `{` $\overline{S}$`;` `default` $\overline{M}$ `}` introduces an interface named I with a list of super-interfaces $\overline{I}$. The new interface defines a list of signatures $\overline{S}$ and a list of default methods `default` $\overline{M}$. For completeness, since Java's semantics allows a class to implement a list of interfaces, we changed the *class declarations* accordingly. Second, the *signature declarations* were added representing prototypes for abstract and concrete methods, where T `m(`$\overline{T}$ $\overline{x}$`)` introduces a method named `m`, a return type T, and parameters $\overline{x}$ of types $\overline{T}$. As a consequence of this, the *method declarations* were also modified. Lastly, we added the constructor for λ*-expressions*, where $(\overline{T}\ \overline{x}) \rightarrow e$ represents an anonymous function, which has a list of arguments with type $\overline{T}$ and names $\overline{x}$, and a body expressions e.

The class table was also modified to accept both class and interface declarations. Therefore, a class table CT is a mapping from class or interface names, to class or interface declarations, `L` or `P` respectively. Considering the addition of interfaces, the *subtyping* relation had to be extended. Following the standard, we write T `<:` U when T is a subtype of U, and T `<:` $\overline{U}$ when T is subtype of all the occurrences of $U_1, ..., U_n$. Formally, the rules for subtyping were defined in Figure 2.

$$T <: T$$

$$\frac{T <: U \qquad U <: V}{T <: V}$$

$$\frac{\mathrm{CT}(C) = \text{class } C \text{ extends } D \text{ implements } \overline{I} \ \{\ ...\ \}}{C <: D \qquad C <: \overline{I}}$$

$$\frac{\mathrm{CT}(I) = \text{interface } I \text{ extends } \overline{I} \ \{\ ...\ \}}{I <: \overline{I}}$$

Figure 2: Subtyping relation between classes and interfaces.

As in FJ, we need some auxiliary definitions for working in the typing and reduction rules. First, we add two rules to obtain abstract and concrete methods, respectively, which are given in Figure 3.

Abstract method lookup

$$\textit{abs-methods}(\text{Object}) = \bullet$$

$$\frac{\begin{array}{c} CT(C) = \text{class } C \text{ extends } D \text{ implements } \overline{I}\ \{\overline{T}\ \overline{f};\ \mathrm{K}\ \overline{M}\} \\ \overline{M} = \overline{S\ \{\text{ return e; }\}} \\ \textit{abs-methods}(D) = \overline{S_1} \\ \textit{abs-methods}(\overline{I}) = \overline{S_2} \end{array}}{\textit{abs-methods}(C) = \overline{S_1} \cup \overline{S_2} - \overline{S}}$$

$$\frac{\begin{array}{c} CT(I) = \text{interface } I \text{ extends } \overline{I}\ \{\overline{S};\ \text{default } \overline{M}\} \\ \textit{abs-methods}(\overline{I}) = \overline{S_1} \end{array}}{\textit{abs-methods}(I) = \overline{S} \cup \overline{S_1}}$$

Concrete method lookup

$$\textit{methods}(\text{Object}) = \bullet$$

$$\frac{\begin{array}{c} CT(C) = \text{class } C \text{ extends } D \text{ implements } \overline{I}\ \{\overline{T}\ \overline{f};\ \mathrm{K}\ \overline{M}\} \\ \textit{methods}(D) = \overline{M_1} \\ \textit{methods}(\overline{I}) = \overline{M_2} \end{array}}{\textit{methods}(C) = \overline{M} \cup \overline{M_1} \cup \overline{M_2}}$$

$$\frac{\begin{array}{c} CT(I) = \text{interface } I \text{ extends } \overline{I}\ \{\overline{S};\ \text{default } \overline{M}\} \\ \textit{methods}(\overline{I}) = \overline{M_1} \end{array}}{\textit{methods}(I) = \overline{M} \cup \overline{M_1}}$$

Figure 3: Abstract and concrete method lookup.

To be considered abstract, a method should have no implementation. Thus, the function `abs-methods` returns the left union of signatures contained in base classes or interfaces, removing those that were implemented by the current class. A concrete method is defined with its body implementation, so the function `methods` returns the left union of all methods contained in the current class, the base class and the default methods contained in the interface list. For both functions, it is important to note the order of the *union* operator, for cases when classes or interfaces have methods with the same names.

Second, we adapted the auxiliary definitions when dealing with field, method type and body lookup[3]. The expected result when calling these functions is similar to those presented in original FJ.

Last, we added a function to annotate λ-expressions with their types. This is important since a λ-expression is written without a type. Instead, the compiler is responsible for inferring the type of each λ-expression, by using the type expected in the context in which the expression appears. This type is called *target type*. According to Java's documentation [11], a λ-expression can appear in a field of a constructor, in an actual parameter on a method call, as a return term of a method body or another λ-expression, and also enclosed by a cast. For example, the target type of a λ-expression that occurs as the actual parameter is the type of the parameter in the method declaration. However, applying small-step reductions, the target type of the λ-expression is not preserved, as we show in the next example. Let's suppose a class `C`, with an attribute `f` which type is a functional interface `I`.

```
(new C((Object x) → x)).f ⟶ (Object x) → x
```

This example shows a reduction step considering the access to attribute `f`, of an object creation `new C((Object x) → x)`, where the constructor receives a λ-expression as parameter, which target type is represented by the functional interface `I`. The orignal rule `R-Field` (p. 407 of [12]) reduces the field access to the value of the actual parameter `f`, here represented by the λ-expression (`(Object x) → x`). As can be noted, after the reduction step, there is no more a target type, and a λ-expression without a type cannot be invoked, justifying the need of our `λmark` function given in Figure 4. The same occurs whenever a λ-expression appears in other kinds of expressions.

$$\frac{\mathrm{e}_0 \text{ is } (\overline{T}\ \overline{x}) \rightarrow \mathrm{e}}{\lambda mark(\mathrm{e}_0,T) = (T)\ \mathrm{e}_0}$$

$$\frac{\mathrm{e}_0 \text{ is not } (\overline{T}\ \overline{x}) \rightarrow \mathrm{e}}{\lambda mark(\mathrm{e}_0,T) = \mathrm{e}_0}$$

Figure 4: Annotating types for λ-expressions.

The function `λmark` is very simple. Its role is to add a *cast* definition if and only if a λ-expression appears in the source-code. We chose to use casts to annotate λ-expressions to avoid extra syntactic definitions. It is important to note that this extra *cast* is a runtime annotation, which will be invisible to the programmer during code development.

3.2 First Approach

In this section, we show the typing and reduction rules considering that the type system is the only responsible for annotating types for λ-expressions by using the type elaboration technique [18]. As result, each typing rule of FJ should produce a new expression, where the λ-expressions are decorated with a *cast* indicating their types.

The typing rules for expressions, method declarations, class declarations, and interface declarations are shown in Figure 5. A difference from original FJ is that the typing judgment for expressions results in addition to the type a new (possibly annotated) expression with the form $\Gamma \vdash e : \langle T, e' \rangle$. This adaptation was made so that the type system produced a new expression with the λ-expressions annotated with a cast. We abbreviate typing judgments and function calls the same way as in previous sections.

The typing rules for expressions `T-Var`, `T-Field`, `T-DCast`, `T-SCast` are very close to FJ. The rule `T-Invk` was changed to call the function `λmark` on the actual parameters $\overline{e}$ when a method invocation occurs before type checking, assuring that if a λ-expression appears, it will be annotated with the respective formal parameter in type list $\overline{U}$, producing a new expression list $\overline{e'}$. Then, the typing judgment is applied to that expression list, resulting in a list of types $\overline{T}$ and a list of terms $\overline{e''}$. This recursive processing is necessary to guarantee that subexpressions are correctly typed. As the expression e_0 can also contain a λ-expression, the typing judgment for it produces another term e_0'. The rules `T-New` and `T-UCast` are similar to `T-Invk` in the sense that `T-New` applies `λmark` and typing judgments on the actual parameters, and `T-UCast` applies `λmark` and typing judgment on the expression e_0. The typing rule `T-Lam` shows how a λ-expression is typed. Firstly, we can note that the *annotated* cast for the λ-expression should be an interface I. Then, a call to the function `abs-methods` checks if the interface has only one abstract method, which indicates that it is a functional interface. Secondly, we apply `λmark` on the λ-expression body e with the return type T of the abstract method, once it can be another λ-expression, producing a new term e'. Lastly, the typing judgment is applied to the λ-expression body with the formal parameters of that λ-expression added to context Γ, resulting in a new body e''. This guarantees subexpressions are correctly typed.

Considering that the body of a method is represented by an expression, consequently, we can have λ-expressions inside methods. This occurs in two ways in the context of FJ: the method returns a λ-expression or a λ-expression occurs as a subexpression inside the method body. Therefore, the *method typing* rule should, besides its role in the original FJ, produce a new method instance containing the possibly annotated body expression. We also adapted the rules for using our previously defined functions `methods` and `abs-methods`. The first step is to call `λmark` on the body expression e with the return type U, which produces the expression e'. Then the context Γ is augmented with the formal parameters and the special variable `this`, where the typing judgment is applied to expression e', producing a new expression e''. Thereafter, the rule checks for subtyping and if the method being processed is present in the class according to the class table. As result, we have a method with the body expression modified. The *class typing* rule checks if the class is well-formed, i.e., if the constructor was defined considering the attributes of the current and the base classes, if all methods are well-typed, and if there are no abstract methods since FJ does

[3] For the lack of space, we omit these rules from the text, however they can be found in p. 402 of the original FJ paper [12].

Expression typing

$$\frac{\Gamma \vdash x\text{: T}}{\Gamma \vdash x \leadsto \langle \text{T},x\rangle}\ \text{[T-Var]}$$

$$\frac{\Gamma \vdash \text{e}_0 : \langle \text{C}_0,\text{e}_0'\rangle \qquad \textit{fields}(\text{C}_0) = \bar{\text{T}}\ \bar{\text{f}}}{\Gamma \vdash \text{e}_0.\text{f}_i \leadsto \langle \text{T}_i,\text{e}_0'.\text{f}_i\rangle}\ \text{[T-Field]}$$

$$\frac{\begin{gathered}\textit{mtype}(\text{m, T}_0) = \bar{\text{U}} \to \text{T}\\ \Gamma \vdash \text{e}_0\text{: } \langle \text{T}_0,\text{e}_0'\rangle \qquad \lambda\text{mark}(\bar{\text{e}},\ \bar{\text{U}}) = \overline{e'}\\ \Gamma \vdash \overline{e'} : \langle \bar{\text{T}},\overline{e''}\rangle \qquad \bar{\text{T}} <: \bar{\text{U}}\end{gathered}}{\Gamma \vdash \text{e}_0.\text{m}(\bar{\text{e}}) \leadsto \langle \text{T},\text{e}_0'.\text{m}(\overline{e''})\rangle}\ \text{[T-Invk]}$$

$$\frac{\begin{gathered}\textit{fields}(\text{C}) = \bar{\text{U}}\ \bar{\text{f}}\\ \lambda\text{mark}(\bar{\text{e}},\ \bar{\text{U}}) = \overline{e'} \qquad \Gamma \vdash \overline{e'} : \langle \bar{\text{T}},\overline{e''}\rangle \qquad \bar{\text{T}} <: \bar{\text{U}}\end{gathered}}{\Gamma \vdash \text{new C}(\bar{\text{e}}) \leadsto \langle \text{C,new C}(\overline{e''})\rangle}\ \text{[T-New]}$$

$$\frac{\begin{gathered}\text{abs-methods(I)} = \{\text{T m}(\bar{\text{T}}\ \bar{\text{y}})\} \quad \lambda\text{mark(e,T)} = \text{e}'\\ \bar{\text{x}}\text{: } \bar{\text{T}},\ \Gamma \vdash \text{e}' : \langle \text{U,e}''\rangle \qquad \text{U} <: \text{T}\end{gathered}}{\Gamma \vdash \text{(I) }((\bar{\text{T}}\ \bar{\text{x}}) \to \text{e}) \leadsto \langle \text{I,(I) }((\bar{\text{T}}\ \bar{\text{x}}) \to \text{e}''))\rangle}\ \text{[T-Lam]}$$

$$\frac{\lambda\text{mark}(\text{e}_0,\ \text{T}) = \text{e}_0' \qquad \Gamma \vdash \text{e}_0' : \langle \text{U,e}_0''\rangle \qquad \text{U} <: \text{T}}{\Gamma \vdash \text{(T) e}_0 \leadsto \langle \text{T,(T) e}_0''\rangle}\ \text{[T-UCast]}$$

$$\frac{\Gamma \vdash \text{e}_0 : \langle \text{U,e}_0'\rangle \qquad \text{T} <: \text{U} \qquad \text{T} \neq \text{U}}{\Gamma \vdash \text{(T) e}_0 \leadsto \langle \text{T,(T) e}_0'\rangle}\ \text{[T-DCast]}$$

$$\frac{\begin{gathered}\Gamma \vdash \text{e}_0 : \langle \text{U,e}_0'\rangle \qquad \text{T} \not<: \text{U} \qquad \text{U} \not<: \text{T}\\ \textit{stupid warning}\end{gathered}}{\Gamma \vdash \text{(T) e}_0 \leadsto \langle \text{T,(T) e}_0'\rangle}\ \text{[T-SCast]}$$

Method typing

$$\frac{\begin{gathered}\lambda\text{mark(e,U)} = \text{e}' \qquad \bar{\text{x}}\text{: }\bar{\text{U}},\text{ this: T} \vdash \text{e}'\text{: } \langle \text{V,e}''\rangle \qquad \text{V} <: \text{U}\\ \text{U m}(\bar{\text{U}}\ \bar{\text{x}})\ \{\text{ return e; }\} \in \text{methods(T)}\end{gathered}}{\text{U m}(\bar{\text{U}}\ \bar{\text{x}})\ \{\text{ return e; }\} \leadsto \langle \text{OK,U m}(\bar{\text{U}}\ \bar{\text{x}})\ \{\text{ return e}''\text{; }\}\rangle}$$

Class typing

$$\frac{\begin{gathered}\text{K} = \text{C}(\bar{\text{U}}\ \bar{\text{g}},\ \bar{\text{T}}\ \bar{\text{f}})\ \{\text{ super}(\bar{\text{g}})\text{; this.}\bar{\text{f}} = \bar{\text{f}}\text{; }\}\\ \textit{fields}(\text{D}) = \bar{\text{U}}\ \bar{\text{g}} \qquad (\bar{\text{M}}\text{ OK in C}) = \langle \text{OK, }\overline{M'}\rangle\\ \textit{abs-methods}(\text{C}) = \bullet\end{gathered}}{\begin{gathered}\text{class C extends D implements }\bar{\text{I}}\ \{\ \bar{\text{T}}\ \bar{\text{f}}\text{; K }\bar{\text{M}}\ \} \leadsto\\ \langle\text{OK,class C extends D implements }\bar{\text{I}}\ \{\ \bar{\text{T}}\ \bar{\text{f}}\text{; K }\overline{M'}\ \}\rangle\end{gathered}}$$

Interface typing

$$\frac{\bar{\text{M}}\text{ OK in I} = \langle \text{OK, }\overline{M'}\rangle \qquad \text{abs-methods(I)} \neq \bullet}{\begin{gathered}\text{interface I extends }\bar{\text{I}}\ \{\ \bar{\text{S}}\text{; }\overline{\texttt{default M}}\ \} \leadsto\\ \langle\text{OK,interface I extends }\bar{\text{I}}\ \{\ \bar{\text{S}}\text{; }\overline{\texttt{default M}'}\ \}\rangle\end{gathered}}$$

Figure 5: First approach: typing rules.

not have abstract classes. When checking the methods, a call for *method typing* results in modified methods $\overline{M'}$, thus, the result of processing *class typing* produces a modified class. The *interface typing* rule is similar to *class typing*, except that interfaces do not have attributes nor constructors, and it should have at least one abstract method.

The added evaluation rules for our first approach are given in Figure 6.

$$\frac{\textit{mbody}(\text{m, I}) = (\bar{\text{y}},\ \text{e}_0)}{((\text{I})\ ((\bar{\text{U}}\ \bar{\text{x}}) \to \text{e})).\text{m}(\bar{\text{u}}) \longrightarrow [\bar{\text{y}} \mapsto \bar{\text{u}}]\ \text{e}_0}\ \text{[R-Default]}$$

$$\frac{}{((\text{I})\ ((\bar{\text{T}}\ \bar{\text{x}}) \to \text{e})).\text{m}(\bar{\text{u}}) \longrightarrow [\bar{\text{x}} \mapsto \bar{\text{u}}]\ \text{e}}\ \text{[R-Lam]}$$

$$\frac{\text{I} <: \text{T}}{(\text{T})\ ((\text{I})\ ((\bar{\text{T}}\ \bar{\text{x}}) \to \text{e})) \longrightarrow (\text{I})\ ((\bar{\text{T}}\ \bar{\text{x}}) \to \text{e})}\ \text{[R-Cast-Lam]}$$

Figure 6: First approach: New reduction rules.

The three original rules of FJ (`R-Field`, `R-Invk`, and `R-Cast`) were kept unchanged and so we omitted them. For dealing with λ-expressions and default methods we added three rules. The reduction rule `R-Default` is applied when a method invocation happens on an annotated λ-expression. Then, it gets the body of a method m defined in an interface I through the function `mbody`. If this body is defined, it performs the substitution of the formal parameters by the actual ones, similarly what happens in the `R-Invk` rule. The rule `R-Lam` is also applied when a method invocation happens on an annotated λ-expression, whereas in this case, the `mbody` function is not defined for method m in the interface I. Therefore, it reduces to the body of the λ-expression e with the formal parameters replaced by the actual ones. The `R-Cast-Lam` is very close to `R-Cast`. When a cast is used on an annotated λ-expression, it checks for subtyping and results in the annotated λ-expression itself. For short, we also omitted the congruence rules, which can be found in [12].

Analogously to FJ, we prove subject reduction and progress to demonstrate type soundness for the Java 8 features of λ-expressions and default methods.

THEOREM 3.1 (SUBJECT REDUCTION). *If* $\Gamma \vdash e : T$ *and* $e \to e'$, *then* $\Gamma \vdash e' : T'$ *for some* $T' <: T$.

PROOF. By induction on the reduction $e \to e'$, with a case analysis on the reduction rule used. It extends the original proof [12] of the corresponding theorem for FJ with the following cases:

Case R-Default. $\texttt{e} = ((\texttt{I})((\bar{\texttt{T}}\ \bar{\texttt{x}}) \to \texttt{e})).\texttt{m}(\bar{\texttt{u}})$, *mbody*`(m, I) = `$(\bar{\texttt{y}},\texttt{e}_0)$, and by rules `T-Invk` and `T-Lam`, we have: $\Gamma \vdash ((\texttt{I})((\bar{\texttt{T}}\ \bar{\texttt{x}}) \to \texttt{e}))\texttt{: I}$, *mtype*`(m,I) = `$\bar{\texttt{U}} \to \texttt{U}$, $\Gamma \vdash \bar{\texttt{u}} : \bar{\texttt{T}}$, and $\bar{\texttt{T}} <: \bar{\texttt{U}}$. Furthermore, $\texttt{e}' = [\bar{\texttt{u}} \mapsto \bar{\texttt{y}}]\texttt{e}_0$ and by the Lemma A.1.2[4], $\Gamma \vdash \texttt{e}' = [\bar{\texttt{u}} \mapsto \bar{\texttt{y}}]\texttt{e}_0 : \texttt{T}$ for some $\texttt{T} <: \texttt{U}$.

Case R-Lam. $\texttt{e} = ((\texttt{I})((\bar{\texttt{T}}\ \bar{\texttt{x}}) \to \texttt{e})).\texttt{m}(\bar{\texttt{u}})$, and by rules `T-Invk` and `T-Lam`, we have: $\Gamma \vdash ((\texttt{I})((\bar{\texttt{T}}\ \bar{\texttt{x}}) \to \texttt{e}))\texttt{: I}$, *mtype*`(m,I) = `$\bar{\texttt{U}} \to \texttt{U}$, $\Gamma \vdash \bar{\texttt{u}} : \bar{\texttt{T}}$, and $\bar{\texttt{T}} <: \bar{\texttt{U}}$. Furthermore, $\texttt{e}' = [\bar{\texttt{u}} \mapsto \bar{\texttt{x}}]\texttt{e}$ and by the Lemma A.1.2, $\Gamma \vdash \texttt{e}' = [\bar{\texttt{u}} \mapsto \bar{\texttt{x}}]\texttt{e} : \texttt{T}$ for some $\texttt{T} <: \texttt{U}$.

[4] The Lemma A.1.2 states that "terms substitution preserves typing" and can be found in p. 426 of FJ paper [12].

Case R-Cast-Lam. e = (T)((I)(($\bar{\texttt{T}}$ $\bar{\texttt{x}}$) $\to$ e)), I <: T, and by rules T-UCast and T-Lam, we have: $\Gamma \vdash$ ((I)(($\bar{\texttt{T}}$ $\bar{\texttt{x}}$) $\to$ e)): I, and (T)((I)(($\bar{\texttt{T}}$ $\bar{\texttt{x}}$) $\to$ e)): T. Furthermore, e′ = (I)(($\bar{\texttt{T}}$ $\bar{\texttt{x}}$) $\to$ e), finishing the case since I <: T. □

Theorem 3.2 (Progress). *Suppose e is a well-typed expression.*

(1) If e includes ((I)(($\bar{\texttt{T}}$ $\bar{\texttt{x}}$) $\to$ e)).m($\bar{\texttt{u}}$) as a subexpression, and mbody(m, I) = ($\bar{\texttt{y}}$,$\texttt{e}_0$), then #$\bar{\texttt{y}}$ = #$\bar{\texttt{u}}$ for some $\bar{\texttt{y}}$ and $\texttt{e}_0$.

(2) If e includes ((I)(($\bar{\texttt{T}}$ $\bar{\texttt{x}}$) $\to$ e)).m($\bar{\texttt{u}}$) as a subexpression, and mbody(m, I) is not defined, then #$\bar{\texttt{x}}$ = #$\bar{\texttt{u}}$ for some $\bar{\texttt{y}}$ and e.

Proof. The proof is based on the analysis of all well-typed expressions, extending previous proofs [12], which can be reduced to the above cases, to conclude that either it is in normal form or it can be further reduced to obtain a normal form. There are two possible normal forms. They are:

- new C($\bar{\texttt{v}}$)	Object as in FJ.
- (I)(($\bar{\texttt{T}}$ $\bar{\texttt{x}}$) $\to$ e)	A well-typed λ-expression. □

Theorem 3.3 (Type Soundness). *If $\emptyset \vdash e : T$ and $e \to^* e'$ with e' being a normal form, then e' is a value w with $\emptyset \vdash w : S$ and $\emptyset \vdash S <: T$.*

Proof. Immediate from above theorems. □

3.3 Second Approach

This section discusses the typing and reduction rules for λ-expressions and default methods in a different way. Here, the type system uses the function λmark when the current expression contains a λ-expression and does not produce a new expression. This annotation is used during the type checking process and is lost after that. Thus, during the evaluation, it is necessary to call λmark again to annotate the types of expressions according to their contexts.

Figure 7 shows on the left the typing rules and on the right the evaluation rules for this approach. There we show only the rules that differ from the original FJ [12], hence, on the expression typing we omit the rules T-Var, T-Field, T-DCast, and T-SCast. In the rule T-Invk we can note a call for λmark using the formal parameters types, obtained by the function mtype, and the actual parameters $\bar{e}$. The purpose of this is to annotate the types when a λ-expression is passed as an argument on a method invocation. The same is performed on the T-New rule, considering the parameters of a constructor, which uses the field types as target types for the λ-expression. The rule T-Lam is similar to the first approach, except here it does not produce a new expression. This rule checks if I is a functional interface, applies λmark in the body expression e with the return type T of the method m. This is done because e can also be a λ-expression. In the end, it verifies if the resulting type of the body is a subtype of the return type of the method m. Finally, in the rule T-UCast the function λmark is used in the case e_0 is a λ-expression.

The rules for *class typing* and *interface typing* were also omitted from the text, once they are mere adaptations of FJ, and very close to the previous section. The difference is that here they do not produce new class or interface instances. The rule for *method typing* uses the function λmark for the method body expression e, to annotate in case it is a λ-expression. After that, it matches the resulting type of the body with the method return type and checks if the current method belongs to a certain class or interface.

For the evaluation rules, we omit the rule R-Cast, since it is the same as FJ. When processing the rule R-Field it is necessary to apply the function λmark in the resulting field, once it can represent a λ-expression. The rule R-Invk applies the λmark twice. First, it is applied to the actual parameters to annotated types in case some parameter is a λ-expression. Second, it is applied to the resulting expression, similarly to the R-Field rule. The rule R-Default is very close to R-Invk, but in this case, the processing occurs with an interface. The rule R-Lam refers to an invocation of a λ-expression. It uses the function mtype to obtain types for the parameters and return of method m from interface I. Then λmark is used on the parameters and in the body expression. The rule R-Cast-Lam is the same as the first approach. In this approach, for short, we also omit the congruence rules, and the soundness proofs, which are similar to those in the previous section.

4 Validation of Semantic Properties

After the presentation of language semantics and the sketches for proofs of type soundness properties, we demonstrate how QuickCheck [5] helps on testing our proposed semantics against these properties using randomly generated high-level programs. For generating random programs in the context of FJ, it is necessary to generate instances of class tables and expressions to be evaluated. Furthermore, to test properties, it is necessary to define Haskell functions in the format accepted by QuickCheck library[5].

We adapted the approach of [17] for generating random programs considering that FJ has nominal instead structural types. In this way, each typing rule is interpreted as a generation rule. We started by producing the algorithm for generating expressions adopting a goal-oriented procedure, which receives as input a class table, an environment and the desired type for the expression being generated. This type should represent a class or interface name present in the class table, with the restriction that it should be instantiable, i.e., or it is a class, or it represents a functional interface to be filled with a λ-expression. Depending on the desired type, a subset of the typing rules may be applicable. For that reason, we prepare a list of candidate expressions for each typing rule, considering what is defined on the class table.

Suppose a class table containing the three classes shown in Section 2, an empty environment Γ, and that we want to generate an expression of type Object. A typing rule can be formatted using ? as a placeholder for that expression, as follows.

$$\Gamma \vdash \texttt{?} : \texttt{Object} \tag{1}$$

[5] Our technical report about random FJ program generation can be found at: https://github.com/fjpub/fj-lam/blob/master/tr.pdf.

Expression typing

$$\frac{\begin{array}{c} \mathit{mtype}(\mathrm{m},\ \mathrm{T}_0) = \bar{\mathrm{U}} \to \mathrm{T} \\ \Gamma \vdash \mathrm{e}_0\!:\ \mathrm{T}_0 \quad \lambda\mathrm{mark}(\bar{\mathrm{e}},\ \bar{\mathrm{U}}) = \overline{e'} \quad \Gamma \vdash \overline{e'} : \bar{\mathrm{T}} \quad \bar{\mathrm{T}} <: \bar{\mathrm{U}} \end{array}}{\Gamma \vdash \mathrm{e}_0.\mathrm{m}(\bar{\mathrm{e}}) : \mathrm{T}} \ [\text{T-Invk}]$$

$$\frac{\begin{array}{c} \mathit{fields}(\mathrm{C}) = \bar{\mathrm{U}}\ \bar{\mathrm{f}} \\ \lambda\mathrm{mark}(\bar{\mathrm{e}},\ \bar{\mathrm{U}}) = \overline{e'} \quad \Gamma \vdash \overline{e'} : \bar{\mathrm{T}} \quad \bar{\mathrm{T}} <: \bar{\mathrm{U}} \end{array}}{\Gamma \vdash \text{new } \mathrm{C}(\bar{\mathrm{e}}) : \mathrm{C}} \ [\text{T-New}]$$

$$\frac{\begin{array}{c} \text{abs-methods}(\mathrm{I}) = \{\mathrm{T}\ \mathrm{m}(\bar{\mathrm{T}}\ \bar{\mathrm{y}})\} \\ \lambda\mathrm{mark}(\mathrm{e},\mathrm{T}) = \mathrm{e}' \quad \bar{\mathrm{x}}\!:\ \bar{\mathrm{T}},\ \Gamma \vdash \mathrm{e}' : \mathrm{U} \quad \mathrm{U} <: \mathrm{T} \end{array}}{\Gamma \vdash (\mathrm{I})\ ((\bar{\mathrm{T}}\ \bar{\mathrm{x}}) \to \mathrm{e}) : \mathrm{I}} \ [\text{T-Lam}]$$

$$\frac{\lambda\mathrm{mark}(\mathrm{e}_0,\ \mathrm{T}) = \mathrm{e}_0' \quad \Gamma \vdash \mathrm{e}_0' : \mathrm{U} \quad \mathrm{U} <: \mathrm{T}}{\Gamma \vdash (\mathrm{T})\ \mathrm{e}_0 : \mathrm{T}} \ [\text{T-UCast}]$$

Method typing

$$\frac{\begin{array}{c} \lambda\mathrm{mark}(\mathrm{e},\mathrm{U}) = \mathrm{e}' \quad \bar{\mathrm{x}}\!:\ \bar{\mathrm{U}},\ \text{this: } \mathrm{T} \vdash \mathrm{e}'\!:\ \mathrm{V} \quad \mathrm{V} <: \mathrm{U} \\ \mathrm{U}\ \mathrm{m}(\bar{\mathrm{U}}\ \bar{\mathrm{x}})\ \{\text{ return e; }\} \in \text{methods}(\mathrm{T}) \end{array}}{\mathrm{U}\ \mathrm{m}(\bar{\mathrm{U}}\ \bar{\mathrm{x}})\ \{\text{ return e; }\}\ \text{OK in T}}$$

Evaluation

$$\frac{\mathit{fields}(\mathrm{C}) = \bar{\mathrm{T}}\ \bar{\mathrm{f}}}{(\text{new } C(\bar{v})).\mathrm{f}_i \longrightarrow \lambda\mathrm{mark}(\mathrm{v}_i,\mathrm{T}_i)} \ [\text{R-Field}]$$

$$\frac{\mathit{mtype}(\mathrm{m},\ \mathrm{I}) = \bar{\mathrm{T}} \to \mathrm{T} \quad \mathit{mbody}(\mathrm{m},\ \mathrm{C}) = (\bar{\mathrm{x}},\ \mathrm{e}_0)}{\begin{array}{c} (\text{new } C(\bar{v})).\mathrm{m}(\bar{\mathrm{u}}) \longrightarrow \\ [\bar{\mathrm{x}} \mapsto \lambda\mathrm{mark}(\bar{\mathrm{u}},\bar{\mathrm{T}}),\ \text{this} \mapsto \text{new } \mathrm{C}(\bar{\mathrm{v}})]\ \lambda\mathrm{mark}(\mathrm{e}_0,\mathrm{T}) \end{array}} \ [\text{R-Invk}]$$

$$\frac{\mathit{mtype}(\mathrm{m},\ \mathrm{I}) = \bar{\mathrm{T}} \to \mathrm{T} \quad \mathit{mbody}(\mathrm{m},\ \mathrm{I}) = (\bar{\mathrm{y}},\ \mathrm{e}_0)}{\begin{array}{c} ((\mathrm{I})\ ((\bar{\mathrm{U}}\ \bar{\mathrm{x}}) \to \mathrm{e})).\mathrm{m}(\bar{\mathrm{u}}) \longrightarrow \\ [\bar{\mathrm{y}} \mapsto \lambda\mathrm{mark}(\bar{\mathrm{u}},\bar{\mathrm{T}})]\ \lambda\mathrm{mark}(\mathrm{e}_0,\mathrm{T}) \end{array}} \ [\text{R-Default}]$$

$$\frac{\mathit{mtype}(\mathrm{m},\ \mathrm{I}) = \bar{\mathrm{T}} \to \mathrm{T}}{\begin{array}{c} ((\mathrm{I})\ ((\bar{\mathrm{T}}\ \bar{\mathrm{x}}) \to \mathrm{e})).\mathrm{m}(\bar{\mathrm{u}}) \longrightarrow \\ [\bar{\mathrm{x}} \mapsto \lambda\mathrm{mark}(\bar{\mathrm{u}},\bar{\mathrm{T}})]\ \lambda\mathrm{mark}(\mathrm{e},\mathrm{T}) \end{array}} \ [\text{R-Lam}]$$

$$\frac{\mathrm{I} <: \mathrm{T}}{(\mathrm{T})\ ((\mathrm{I})\ ((\bar{\mathrm{T}}\ \bar{\mathrm{x}}) \to \mathrm{e})) \longrightarrow (\mathrm{I})\ ((\bar{\mathrm{T}}\ \bar{\mathrm{x}}) \to \mathrm{e})} \ [\text{R-Cast-Lam}]$$

Figure 7: Second approach: typing and reduction rules.

By looking at the class table, we can have candidates using the rules `T-Field`, where both fields `fst` and `snd` of class `Pair` are of type `Object`; `T-New`, which can be used to create a new instance of `Object`; and `T-UCast`[6] which can cast any class on class table, since `Object` is the superclass of them. For example, consider the algorithm to produce the candidate list for the rule `T-Field`. It will lookup for classes containing attributes of type `Object`, which results the fields of class `Pair`. Below we show another step during the construction of a candidate expression.

$$\frac{\Gamma \vdash ?_1 : \texttt{Pair}}{\Gamma \vdash (?_1).\texttt{fst} : \texttt{Object}} \qquad (2)$$

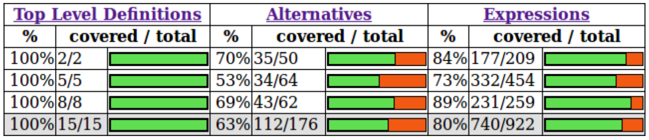
There, we can note the access to a field called `fst` of an expression which should have the type `Pair`. The generator is performed recursively for each placeholder, using the QuickCheck `sized` function, until the expression is complete or depth bound exceeded. The same process is applied to the others typing rules. Our algorithm produces a list of candidate expressions for each typing rule. We use the QuickCheck function `oneof` to select one random expression for each candidate list and we use `oneof` again to select one of the remaining expressions. This way, we generate well-typed expressions with the same probability distribution for each typing rule.

A similar approach is defined to fill the class table with classes and interfaces. For example, through the *class typing* rule (Figure 5) we have:

$$\text{class } ?_1 \text{ extends } ?_2 \text{ implements } ?_3\ \{\ ?_4;\ ?_5\ ?_6\ \} \qquad (3)$$

Where $?_1$ refers to a new class name, not yet present in the class table, $?_2$ is the base class, which is `Object` or a class contained in the class table, $?_3$ is a possibly empty list of interfaces obtained from the class table, $?_4$ is a list of already defined types and names for attributes, $?_5$ is formatted according to the base class attributes and the new ones defined in $?_4$, and $?_6$ is a list of randomly generated methods in addition to the implementation of the abstract methods present in the interface list $?_3$, following the *method typing* rule. For the body of a method, we use our previously defined expression generator with the context augmented with the formal parameters and the special keyword `this`. A similar approach is adopted when generating interfaces.

We use QuickCheck to test if all generated class tables are well-formed, if all generated expressions are well-typed and cast-safe, if the properties of Progress and Preservation hold for both semantic approaches, and if both versions are equivalent. After running many thousands of well-succeeded tests, we gain a high degree of confidence in the safety of our semantics, however, it is important to measure how much of code base is covered by the test suite. Such statistics are provided by Haskell Program Coverage tool [10]. Results of code coverage are presented in Figure 8.



Top Level Definitions			Alternatives			Expressions		
%	covered / total		%	covered / total		%	covered / total	
100%	2/2	🟩	70%	35/50	🟩🟥	84%	177/209	🟩🟥
100%	5/5	🟩	53%	34/64	🟩🟥	73%	332/454	🟩🟥
100%	8/8	🟩	69%	43/62	🟩🟥	89%	231/259	🟩🟥
100%	15/15	🟩	63%	112/176	🟩🟥	80%	740/922	🟩🟥

Figure 8: Test coverage results.

Although not having 100% of code coverage, our test suite provides a strong evidence that proposed semantics enjoys safety properties by exercising on randomly generated programs of increasing size. By analyzing test coverage results, we can observe that code not reached by test cases consists of stuck states on program semantics or error control for not well-typed expressions.

[6] The rules `T-DCast` and `T-SCast` are not used in expression generation since they can produce cast unsafe expressions.

5 Related Work

The use of FJ as the basis for investigations of novel structures for object-oriented languages has been explored in several projects [1–3, 9, 14, 16], since it is the current most popular Java formalism. The study of extensions focusing on higher-order mechanisms to enhance expressivity, conciseness, good structuring, reusability, and factoring of code has been performed in the last years. Discussions for adding λ-expressions in Java appear since 2006 [4, 6, 15], however without a formal description. Specifically, Bellia and Occhiuto [1] modeled a proposal for closures in Featherweight Java, before Java 8 release, where *closure types* were added to the type system as first-class values, which can be bound to parameters, hence applied to methods or other closures. Their proposal differs from the current Java implementation. Bettini et al. [2] formalized λ-expressions in FJ, after Java 8 release, adding interfaces and *intersection types*, aiming to study the relation between λ-expressions and intersection types in the Java language. The last is very close to our approach (since we adapted their function to annotate λ-types), except that we model our semantics without the need for intersection types, and two different approaches for λ-types annotation.

Random testing for finding bugs in compilers and programming language tools received some attention in recent years. Daniel et al. [7] generate random Java programs to test refactoring engines in Eclipse and NetBeans. Klein et al. [13] generated random programs to test an object-oriented library. These projects are close related to ours since they are generating code in the object-oriented context. The difference of our approach is that we generate randomly class tables and expressions, both well-formed and well-typed. The work of Palka, Claessen and Hughes [17] used QuickCheck library to generate λ-terms to test the GHC compiler. Their approach for generating terms was adopted in our project, in the sense we also used QuickCheck and the typing rules for generating well-typed programs. Unlike their approach, by reading the generated class table, we generate a list of candidate expressions, which eliminates the need for backtracking. Furthermore, the use of QuickCheck helped us on refining our semantics, our implementation, and allowed testing for type-safety properties.

6 Conclusion

In this work, we presented two different semantic versions for λ-expressions and default methods in the context of Featherweight Java and used property-based testing to verify it. The lightweight approach provided by QuickCheck allows to experiment with different semantic designs and implementations and to quickly check any changes. During the development of this work, we have changed our basic definitions many times, both as a result of correcting errors and streamlining the presentation. Ensuring that our changes were consistent was simply a matter of re-running the test suite. Encoding the type soundness properties as Haskell functions provides a clean and concise implementation that helps not only to fix semantics but also to improve understanding the meaning of our extensions.

As future work, we intend to use Coq to provide formally certified proofs that the presented semantic does enjoy safety properties and also to explore the approach used in our test suite for other extensions of FJ, besides using other tools like QuickChick with the same purpose.

Acknowledgement

Thanks to Prof. Wouter Swierstra for his valuable suggestions and comments on early drafts of this paper.

References

[1] Marco Bellia and M. Eugenia Occhiuto. 2011. Properties of Java Simple Closures. *Fundam. Inf.* 109, 3 (Aug. 2011), 237–253. http://dl.acm.org/citation.cfm?id=2361335.2361338

[2] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 2018. Java & Lambda: a Featherweight Story. *CoRR* abs/1801.05052 (2018).

[3] Gavin M Bierman, MJ Parkinson, and AM Pitts. 2003. *MJ: An imperative core calculus for Java and Java with effects*. Technical Report. University of Cambridge, Computer Laboratory.

[4] G Bracha, N Gafter, J Gosling, and P von der Ahe. 2008. Closures for the Java prog. lang. (aka BGGA). http://javac.info/. (2008).

[5] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ACM SIGPLAN Int. Conf. on Func. Prog. (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[6] Stephen Colebourne and S Shulz. 2006. First-class methods: Java-style closures. (2006).

[7] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *European Software Eng.Conference and the ACM SIGSOFT Symposium on The Foundations of Software Eng. (ESEC-FSE '07)*. ACM, New York, NY, USA, 185–194. https://doi.org/10.1145/1287624.1287651

[8] Mourad Debbabi and Myriam Fourati. 2007. A Formal Type System for Java. *J. of Object Technology* 6, 8 (2007), 117–184.

[9] Benjamin Delaware, William Cook, and Don Batory. 2011. Product lines of theorems. In *ACM SIGPLAN Notices*. ACM, 595–608.

[10] Andy Gill and Colin Runciman. 2007. Haskell Program Coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1291201.1291203

[11] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2014. The Java Lang. Specification, Java SE 8. (2014).

[12] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

[13] Casey Klein, Matthew Flatt, and Robert Bruce Findler. 2010. Random Testing for Higher-order, Stateful Programs. In *Proc. of the ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 555–566. https://doi.org/10.1145/1869459.1869505

[14] Edlira Kuci, Sebastian Erdweg, Oliver Bračevac, Andi Bejleri, and Mira Mezini. 2017. A Co-contextual Type Checker for Featherweight Java. *arXiv preprint arXiv:1705.05828* (2017).

[15] Bob Lee, Doug Lea, and Josh Bloch. 2006. Concise Instance Creation Expressions: Closure without Complexity. (2006).

[16] Johan Östlund and Tobias Wrigstad. 2010. Welterweight Java. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS'10)*. Springer-Verlag, Berlin, Heidelberg, 97–116. http://dl.acm.org/citation.cfm?id=1894386.1894392

[17] Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

[18] François Pottier. 2014. Hindley-milner Elaboration in Applicative Style: Functional Pearl. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 203–212. https://doi.org/10.1145/2628136.2628145

[19] tiobe.com. 2018. TIOBE Index. https://www.tiobe.com/tiobe-index/. (04 2018). Accessed: 2019-04-09.