Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 342 (2019) 3–20

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Generating Random Well-Typed Featherweight Java Programs Using QuickCheck

Samuel da Silva Feitosa[a,1] Rodrigo Geraldo Ribeiro[b,2]
Andre Rauber Du Bois[a,3]

[a] *Programa de Pós Graduação em Computação / PPGC*
*Universidade Federal de Pelotas*
*Pelotas - RS, Brazil*

[b] *Programa de Pós Graduação em Ciência da Computação / PPGCC*
*Universidade Federal de Ouro Preto*
*Ouro Preto - MG, Brazil*

**Abstract**

Currently, Java is one of the most used programming language, being adopted in many large projects, where applications reach a level of complexity for which manual testing and human inspection are not enough to guarantee quality in software development. Even when using automated unit tests, such tests rarely cover all interesting cases of code, which means that a bug could never be discovered, once the code is tested against the same set of rules over and over again. This paper addresses the problem of generating random well-typed programs in the context of Featherweight Java, a well-known object-oriented calculus, using QuickCheck, a Haskell library for property-based testing.

*Keywords:* Random Program Generation, Property-Based Testing, Featherweight Java.

## 1 Introduction

Nowadays, Java is one of the most popular programming languages [24]. 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. In a recent major release of Java, new features such as lambda expressions, method references, and functional interfaces, were added to

[1] Email: samuel.feitosa@inf.ufpel.edu.br
[2] Email: rodrigo@decsi.ufop.br
[3] Email: dubois@inf.ufpel.edu.br

https://doi.org/10.1016/j.entcs.2019.04.002

the core language, offering a programming model that fuses the object-oriented and functional styles [13].

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 [6]. 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 a certain degree of abstraction, using only properties of interest, facilitating the understanding of the problem and also allowing the use of automatic tools [11].

Therefore, an important research area concerns the formal semantics of languages and type-system specification, which enables formal proofs and establishing program properties. Besides, solutions can be machine checked 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 [3].

In this context, this work provides the specification of a test generator for Java programs, using the typing rules of Featherweight Java [14] (FJ) to generate only well-typed programs. FJ is a small core calculus of Java with a rigorous semantic definition of its main core aspects. The motivations for using FJ as a starting point are that it is compact, so we can model our test generators in a way that it can be extended with new features, and its minimal syntax, typing rules, and operational semantics fit well for modeling and proving properties for the compiler and programs. As far as we know, there are no well-typed test generators for FJ. This work aims to fill this gap by specifying a generator for FJ programs using QuickCheck, a property-based testing library for Haskell. 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 implement an interpreter [4] for FJ in Haskell, which can be used as the basis to study new features on the object-oriented context.
- We provide a type-directed heuristic [20] for constructing random programs. We conjecture that our specification is sound with respect to FJ type system, i.e. it generates only well-typed programs.
- We use QuickCheck as a lightweight manner to check if all generated programs are well-typed and to test our interpreter against type soundness proofs in order to validate the proposed approach.

The remainder of this text is organized as follows: Section 2 summarizes FJ.

[4] The source-code for our interpreter and the test suite is available at https://github.com/sfeitosa/fj-qc.

Section 3 presents the process of generating well-typed random programs in the context of FJ. Section 4 shows the results of testing type-safety properties of FJ with QuickCheck. Section 5 discusses related works. Finally, we present the final remarks in Section 6.

## 2 Featherweight Java

Featherweight Java (FJ) [14] 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 $\lambda$-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 [14]. 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 different kinds of terms: `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 [23]. 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. 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 [23]. Next, we present the original description of FJ [14].

### 2.1 Syntax and Auxiliary Functions

The abstract syntax of FJ is given in Figure 1, where `L` represents classes, `K` defines constructors, `M` stands 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` 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 $\overline{C}$ as shorthand for a possibly empty sequence $C_1, ..., C_n$ (similarly for $\overline{f}$, $\overline{x}$, etc.). An empty sequence is denoted by $\bullet$, and the length of a sequence $\bar{\texttt{x}}$ is written $\#\bar{\texttt{x}}$. We use $\Gamma$ 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 C such that x: C $\in \Gamma$. We slightly abuse notation by using set operators on sequences. Their meaning is as usual.

**Syntax**

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

Fig. 1. Syntactic definitions for FJ.

A class table $CT$ is a mapping from class names, to class declarations $L$, and it should satisfy some conditions, such as each class `C` should be in $CT$, except `Object`, which is a special class; and there are no cycles in the subtyping relation. Thereby, a program is a pair $(CT, e)$ of a class table and an expression.

Figure 2 shows the rules for subtyping, where we write $C$ `<:` $D$ when $C$ is a subtype of $D$.

$$\frac{}{C <: C}$$

$$\frac{C <: D \qquad D <: E}{C <: E}$$

$$\frac{\text{CT}(C) = \text{class } C \text{ extends } D \ \{\ ...\ \}}{C <: D}$$

Fig. 2. Subtyping relation between classes.

The authors also proposed some auxiliary definitions for working in the typing and reduction rules. These definition are given in Figure 3. The rules for *field lookup*

demonstrate how to obtain the fields of a given class. If the class is `Object`, an empty list is returned. Otherwise, it returns a sequence $\overline{C}\ \overline{f}$ pairing the type of each field with its name, for all fields declared in the given class and all of its superclasses. The rules for *method type lookup* (*mtype*) show how the type of method $m$ in class $C$ can be obtained. The first rule of *mtype* returns a pair, written $\overline{B} \to B$, of a sequence of argument types $\overline{B}$ and a result type $B$, when the method $m$ is contained in $C$. Otherwise, it returns the result of a call to *mtype* with the superclass. A similar approach is used in the rules for *method body lookup*, where *mbody*(m, $C$) returns a pair $(\overline{x}, e)$, of a sequence of parameters $\overline{x}$ and an expression $e$. Both *mtype* and *mbody* are partial functions.

**Field lookup**

$$\mathit{fields}(\text{Object}) = \bullet$$

$$\frac{CT(C) = \text{class } C \text{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \qquad \mathit{fields}(D) = \overline{D}\ \overline{g}}{\mathit{fields}(C) = \overline{D}\ \overline{g}, \overline{C}\ \overline{f}}$$

**Method type lookup**

$$\frac{CT(C) = \text{class } C \text{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \qquad B\ \text{m}(\overline{B}\ \overline{x})\ \{\ \text{return } e;\ \} \in \overline{M}}{\mathit{mtype}(\text{m}, C) = \overline{B} \to B}$$

$$\frac{CT(C) = \text{class } C \text{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \qquad \text{m} \notin \overline{M}}{\mathit{mtype}(\text{m}, C) = \mathit{mtype}(\text{m}, D)}$$

**Method body lookup**

$$\frac{CT(C) = \text{class } C \text{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \qquad B\ \text{m}(\overline{B}\ \overline{x})\ \{\ \text{return } e;\ \} \in \overline{M}}{\mathit{mbody}(\text{m}, C) = (\overline{x}, e)}$$

$$\frac{CT(C) = \text{class } C \text{ extends } D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\} \qquad \text{m} \notin \overline{M}}{\mathit{mbody}(\text{m}, C) = \mathit{mbody}(\text{m}, D)}$$

Fig. 3. Auxiliary definitions.

### *2.2 Typing and Reduction Rules*

This section presents how the typing rules of FJ are used to guarantee type soundness, i.e., well-typed terms do not get stuck, and the reduction rules showing how each step of evaluation should be processed for FJ syntax. Figure 4 shows in the left side, the typing rules for expressions, and in the right side, it shows first the

rules to check if methods and classes are well-formed, then the reduction rules for this calculus. We omit here the congruence rules, which can be found in the original paper [14].

The typing judgment for expressions has the form $\Gamma \vdash \texttt{e}: \ \texttt{C}$, meaning that in the environment $\Gamma$, expression `e` has type `C`. The abbreviations when dealing with sequences is similar to the previous section. The typing rules are syntax directed, with one rule for each form of expression, save that there are three rules for casts.

The rule `T-Var` results in the type of a variable $x$ according to the context $\Gamma$. If the variable $x$ is not contained in $\Gamma$, the result is undefined. Similarly, the result is undefined when calling the functions `fields`, `mtype`, and `mbody` in cases when the target class or the methods do not exist in the given class. The rule `T-Field` applies the typing judgment on the subexpression $\texttt{e}_0$, which results in the type $\texttt{C}_0$. Then it obtains the *fields* of class $\texttt{C}_0$, matching the position of $\texttt{f}_i$ in the resultant list, to return the respective type $\texttt{C}_i$. The rule `T-Invk` also applies the typing judgment on the subexpression $\texttt{e}_0$, which results in the type $\texttt{C}_0$, then it uses *mtype* to get the formal parameter types $\bar{\texttt{D}}$ and the return type `C`. The formal parameter types are used to check if the actual parameters $\bar{\texttt{e}}$ are subtypes of them, and in this case, resulting in the return type `C`. The rule `T-New` checks if the actual parameters are a subtype of the constructor formal parameters, which are obtained by using the function *fields*. There are three rules for casts: one for *upcasts*, where the subject is a subclass of the target; one for *downcasts*, where the target is a subclass of the subject; and another for *stupid casts*, where the target is unrelated to the subject. Even considering that Java's compiler rejects as ill-typed an expression containing a stupid cast, the authors found that a rule of this kind is necessary to formulate type soundness proofs.

The rule for *method typing* checks if a method declaration `M` is well-formed when it occurs in a class `C`. It uses the expression typing judgment on the body of the method, with the context $\Gamma$ augmented with variables from the actual parameters with their declared types, and the special variable `this`, with type `C`. The rule for *class typing* checks if a class is well-formed, by checking if the constructor applies super to the fields of the superclass and initializes the fields declared in this class, and that each method declaration in the class is well-formed.

There are only three computation rules, indicating which expressions can be used in the main program. The first rule `R-Field` formalizes how to evaluate an *attribute access*. Similarly to the typing rule `T-Field`, it uses the function *fields*, and matches the position $i$ of the field $\texttt{f}_i$ in the resulting list, returning the value $\texttt{v}_i$, which refers to the value in the position $i$ of the actual parameter list. The second rule `R-Invk` shows the evaluation procedure for a *method invocation*, where firstly it obtains the method body expression `m` of class `C` through the function *mbody*, and then performs substitution of the actual parameters and the special variable `this` in the body expression, similar to a beta reduction on $\lambda$-calculus. The last rule `R-Cast` refers to *cast processing*, where the same subexpression `new C(`$\bar{\texttt{e}}$`)` is returned in case the subject class `C` is subtype of the target class `D`. There are

**Expression typing**

$$\frac{}{\Gamma \vdash \mathrm{x}\colon \Gamma(\mathrm{x})}\ [\text{T-Var}]$$

$$\frac{\Gamma \vdash \mathrm{e}_0\colon \mathrm{C}_0 \qquad \mathit{fields}(\mathrm{C}_0) = \bar{\mathrm{C}}\ \bar{\mathrm{f}}}{\Gamma \vdash \mathrm{e}_0.\mathrm{f}_i\colon \mathrm{C}_i}\ [\text{T-Field}]$$

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

$$\frac{\mathit{fields}(\mathrm{C}) = \bar{\mathrm{D}}\ \bar{\mathrm{f}} \qquad \Gamma \vdash \bar{\mathrm{e}} : \bar{\mathrm{C}} \qquad \bar{\mathrm{C}} <: \bar{\mathrm{D}}}{\Gamma \vdash \text{new } \mathrm{C}(\bar{\mathrm{e}}) : \mathrm{C}}\ [\text{T-New}]$$

$$\frac{\Gamma \vdash \mathrm{e}_0 : \mathrm{D} \qquad \mathrm{D} <: \mathrm{C}}{\Gamma \vdash (\mathrm{C})\ \mathrm{e}_0 : \mathrm{C}}\ [\text{T-UCast}]$$

$$\frac{\Gamma \vdash \mathrm{e}_0 : \mathrm{D} \qquad \mathrm{C} <: \mathrm{D} \qquad \mathrm{C} \neq \mathrm{D}}{\Gamma \vdash (\mathrm{C})\ \mathrm{e}_0 : \mathrm{C}}\ [\text{T-DCast}]$$

$$\frac{\Gamma \vdash \mathrm{e}_0 : \mathrm{D} \qquad \mathrm{C} \not<: \mathrm{D} \qquad \mathrm{D} \not<: \mathrm{C} \qquad \textit{stupid warning}}{\Gamma \vdash (\mathrm{C})\ \mathrm{e}_0 : \mathrm{C}}\ [\text{T-SCast}]$$

**Method typing**

$$\frac{\bar{\mathrm{x}}\colon \bar{\mathrm{C}}, \mathrm{this}\colon \mathrm{C} \vdash \mathrm{e}_0\colon \mathrm{E}_0 \qquad \mathrm{E}_0 <:\ \mathrm{C}_0 \qquad \text{class C extends D } \{...\} \qquad \text{if } \mathit{mtype}(\mathrm{m}, \mathrm{D}) = \bar{\mathrm{D}} \to \mathrm{D}_0, \text{ then } \bar{\mathrm{C}} = \bar{\mathrm{D}} \text{ and } \mathrm{C}_0 = \mathrm{D}_0}{\mathrm{C}_0\ \mathrm{m}(\bar{\mathrm{C}}\ \bar{\mathrm{x}})\ \{\ \text{return } \mathrm{e}_0;\ \}\ \text{OK in C}}$$

**Class typing**

$$\frac{\mathrm{K} = \mathrm{C}(\bar{\mathrm{D}}\ \bar{\mathrm{g}}, \bar{\mathrm{C}}\ \bar{\mathrm{f}})\ \{\ \mathrm{super}(\bar{\mathrm{g}}); \mathrm{this}.\bar{\mathrm{f}} = \bar{\mathrm{f}};\ \} \qquad \mathit{fields}(\mathrm{D}) = \bar{\mathrm{D}}\ \bar{\mathrm{g}} \qquad \bar{\mathrm{M}}\ \text{OK in C}}{\text{class C extends D } \{\ \bar{\mathrm{C}}\ \bar{\mathrm{f}};\ \mathrm{K}\ \bar{\mathrm{M}}\ \}\ \text{OK}}$$

**Evaluation**

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

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

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

Fig. 4. Typing and evaluation rules.

also five congruence rules[5] (omitted from Figure 4), which are responsible for the intermediary evaluation steps for the proposed small-step semantics.

The FJ 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

[5] The congruence rules omitted from the text can be found in p. 407 of [14].

and type-checking rules, the authors present proofs of type soundness for FJ as another important contribution, which will be explored by our test suite in the next sections.

# 3 Program Generation

The task of creating tests for a programming language is time-consuming. First, because it should respect the programming language requirements, in order to produce a valid test case. Second, if the test cases are created by a person, it stays limited by human imagination, where obscure corner cases could be overlooked. If the compiler writers are producing the test cases, they can be biased, since they can make assumptions about their implementation or about what the language should do. Furthermore, when the language evolves, previous test cases could be an issue, considering the validity of some old tests may change if the language semantics is altered [1].

Considering the presented problem, there is a growing research field exploring random test generation. However, generating good test programs is not an easy task, since these programs should have a structure that is accepted by the compiler, respecting some constraints, which can be as simple as a program having the correct syntax, or more complex such as a program being type-correct in a statically-typed programming language [22].

For generating random programs in the context of FJ, the generation step has two distinct phases. First, it is necessary to randomly generate classes to compose the class table. Second, an expression should be generated by using the class table. Hence, this section describes the proposed type-directed procedure for generating well-typed terms, and well-formed classes by using the QuickCheck library [4].

QuickCheck is an automated testing tool for Haskell. It defines a formal specification language allowing its use to specify code under test, and to check if certain properties hold in a large number of randomly generated test cases. This library provides several test case generators for constructors of the Haskell language, but it leaves for its users the definition of generators for user-defined types. The library provides combinators which help the programmer in this process.

In this paper, we generalized the approach of [22] for generating random programs considering that FJ has a nominal type system instead of a structural one. In this way, each typing rule is interpreted as a generation rule, both for expression generation and class table generation. The generation process for each of them is explained as follows.

## *3.1 Expression Generation*

We started by defining the process for generating FJ expressions adopting a goal-oriented procedure, which receives as input an arbitrary class table, an environment and the desired type for the expression being generated. This desired type should represent a class name contained in the class table. The aim of the expression generator is to produce a well-typed term of the desired type, which can contain

free variables from a given environment. For generating an expression of a given type, only a subset of typing rules can be used. For example, the rule `T-Var` can only be used when generating a method body expression, because the formal parameters of a method represent the free variables in the environment, the rule `T-Field` can only be used if some class in the class table has attributes of the desired type, and so on.

The adopted generation method for expressions is obtained by reading the expression typing rules in Figure 4 backwards, i.e. to generate an expression that is in the consequence of a rule it is first necessary to generate expressions that are in its premises, and then combine them. This way, the goal of generating a term might involve generating the subgoals recursively. By using the typing rules we ensure that the resulting terms are well-typed.

Suppose a class table containing the three classes `A`, `B`, and `Pair` shown in Section 2, an empty environment $\Gamma$, and that we want to generate an expression of type `Object`. A typing rule can be formatted using the question mark `?` as a placeholder for that expression, representing the first generation step, as follows:

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

By looking at the class table, by the rule `T-Field`, we can generate an expression of type `Object` accessing the attributes `fst` or `snd` of class `Pair`, `T-New`[6], which can be used to create a new instance of `Object`, and rule `T-UCast`[7] which can cast any class on class table, since `Object` is the superclass of all classes. Lets look at the first one, with the typing rule `T-Field` showing another step to generate the term, as follows.

$$\frac{\Gamma \vdash \texttt{?}_1\text{: Pair} \qquad \mathit{fields}(\text{Pair}) = \{\text{Object fst, Object snd}\}}{\Gamma \vdash (\texttt{?}_1).\text{fst: Object}}\ \text{[T-Field]} \tag{2}$$

The question mark $\texttt{?}_1$ represents the subexpression that will be generated as a subgoal. In this sense, we can note the access to field `fst` of a subexpression $\texttt{?}_1$ which should have the type `Pair`, as stated by its premise.

To generate an expression for the subgoal, we have to look at the class table again to generate a term of type `Pair`. There are two ways for doing this, by instantiating the class `Pair` or invoking the method `.setfst(...)`. Let's consider the first, which uses the typing rule `T-New`, demonstrating another step in the generation process,

[6] The class `Object` is considered a distinguished class name whose definition does not appear in the class table.

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

as follows:

$$\dfrac{\dfrac{\textit{fields}(\text{Pair}) = \bar{\text{D}}\ \bar{\text{f}} \quad \Gamma \vdash \overline{?_2}\text{: } \bar{\text{C}} \quad \bar{\text{C}} <\text{: } \bar{\text{D}}}{\Gamma \vdash \text{new Pair}(\overline{?_2})\text{: Pair}}\ \text{[T-New]} \quad \textit{fields}(\text{Pair}) = \{\text{Object fst, Object snd}\}}{\Gamma \vdash (\text{new Pair}(\overline{?_2})).\text{fst: Object}}\ \text{[T-Field]} \tag{3}$$

As can be noted, the generator is applied recursively for each placeholder. We denote $\overline{?_2}$ as a sequence of placeholders, similarly to previous sections when dealing with sequences of types or variables. The placeholder $\overline{?_2}$ represents the actual parameters passed for the class `Pair` constructor, where each subexpression should be generated according to the types returned by the function *fields*.

The process for generating an expression using the rule `T-Invk` is similar, in the sense that it should generate a subexpression representing the instantiation of an object, which contains the given method, and should generate the actual parameters, where each one should have the expected type according to *mtype*. The discussed generation rules are capable of generating every well-typed expression in FJ since an expression is well-typed if there exists a typing derivation for it.

We define an algorithm that generates expressions recursively by applying the generation rules using the QuickCheck library. To prevent non-terminating generation, each recursive invocation of the algorithm uses a *size* parameter, which is decreased in subsequent invocations. When size becomes zero, only the rules `T-Var` and `T-New` can be used, which avoid excessive recursion.

The algorithm for generating an expression first creates a list of candidate expressions for each typing rule. A candidate list for a typing rule can be empty, indicating that it is impossible to generate an expression of the desired type for that rule, and as consequence, it is ignored. After producing all the candidate lists, the algorithm randomly chooses one candidate for each of those lists, using the QuickCheck function *oneof*. Over the selected candidates, we apply the function *oneof* one more time, resulting in one candidate expression, which will be used by our recursive generation rules. We chose this approach to guarantee an equal distribution for each non-empty typing rule.

### *3.2 Class Table Generation*

The process for generating the class table is more elaborated, since at first we do not have any information to start with. What we have is just a set of conditions that a class table should satisfy, such as: (1) $CT(\text{C}) =$ `class C...` for every `C` $\in$ *dom(CT)*; (2) `Object` $\notin$ *dom(CT)*; (3) for every class name `C` (except `Object`) appearing anywhere in *CT*, we have `C` $\in$ *dom(CT)*; and (4) there are no cycles in the subtype relation induced by *CT*, i.e. the relation `<:` is antisymmetric.

We start off by defining three generators, one for class names, one for variable names, and another for types. Our generator for class names just chooses randomly an uppercase letter, which can be easily adapted to generate names with different

lengths. The process is similar to variable names, except that it is generated a lowercase letter. The generator for types chooses randomly one class name present in the class table or `Object`. After that, we proceed for the class table generation, using a list of non-duplicated class names. For each element of that list, our generator produces a class and inserts it in the class table.

The first step to generate a class, after we have its name, is to define its base class. Here, the type generator is used. If the class table is empty, the type generator returns `Object`, otherwise, it returns an already produced class. This simple procedure assures that there are no cycles on the class hierarchy. Suppose that a class name `Z`, and a base class `Object` were generated in the first step. Then, the class typing rule is used for the next steps, as follows:

$$\frac{\begin{array}{c}\mathrm{K} = \mathrm{C}(\overline{\texttt{?}_1}\ \overline{\texttt{?}_2})\ \{\ \mathrm{super}();\ \mathrm{this}.\overline{\texttt{?}_2} = \overline{\texttt{?}_2};\ \}\\ \mathit{fields}(\mathrm{Object}) = \bullet \qquad \bar{\mathrm{M}}\ \mathrm{OK\ in\ C}\end{array}}{\mathrm{class\ Z\ extends\ Object}\ \{\ \overline{\texttt{?}_1}\ \overline{\texttt{?}_2};\ \mathrm{K}\ \bar{\mathrm{M}}\ \}}\ \text{[C OK]} \tag{4}$$

The process for filling the class components is divided into two parts. The first is demonstrated above, where a list of attributes is randomly generated. Initially, the generator chooses a random number $n$ to define how many fields the class should have. Then a list of types with size $n$ is generated for the placeholder $\overline{\texttt{?}_1}$, and a list of non-duplicate variable names with size $n$ is generated for the placeholder $\overline{\texttt{?}_2}$. As we can note in the generation rule, the function *fields* is called with the base class `Object`, returning an empty sequence. Then, the constructor is formatted accordingly.

The described process has already generated a minimalist class, considering that it has a class name, a base class, its attributes, and a constructor. Before proceeding to the second part of a class generation, the generator algorithm adds this minimalist class in the class table. This is necessary to allow method body expressions to use the class attributes through the special variable `this`. Then we move to method generation, which appears in the generation rule as $\bar{\texttt{M}}$ `OK in C`. As we saw in previous sections, the use of $\bar{\texttt{M}}$ indicates a possibly empty list of methods.

The second part to conclude the class generation concerns to method generation. The process starts generating a random number $n$, which represents that it will be generated $n$ methods for the given class `C`. A method is represented by its signature and by its body. For generating the signature, it is necessary to produce the method name, the return type, and the formal parameters (types and names). This step is performed according to the *method typing* rule, as follows:

$$\frac{\begin{array}{c}\overline{\texttt{?}_4}:\ \overline{\texttt{?}_3},\ \mathrm{this}:\ \mathrm{C} \vdash \mathrm{e}_0:\ \mathrm{E}_0 \qquad \mathrm{E}_0\ \texttt{<:}\ \texttt{?}_1\\ \mathrm{class\ C\ extends\ D}\ \{\ldots\}\\ \mathrm{if}\ \mathit{mtype}(\texttt{?}_2, \mathrm{D}) = \bar{\mathrm{D}} \to \mathrm{D}_0,\ \mathrm{then}\ \overline{\texttt{?}_3} = \bar{\mathrm{D}}\ \mathrm{and}\ \texttt{?}_1 = \mathrm{D}_0\end{array}}{\texttt{?}_1\ \texttt{?}_2(\overline{\texttt{?}_3}\ \overline{\texttt{?}_4})\ \{\ \mathrm{return\ e}_0;\ \}} \tag{5}$$

In this step, it is randomly produced a return type to be allocated in the placeholder $\texttt{?}_1$, already defined in the class table, and a lowercase letter to compose the method name $\texttt{?}_2$. As FJ does not allow method overloading, to avoid methods with the same name in the class hierarchy, the generator appends the class name at the

end of the method name. After that, it is generated a list of random size for the formal parameters. The list of types is placed in $\overline{?_3}$, and the non-duplicate list of names is placed in $\overline{?_4}$. As we are generating exclusive method names for each class, the function *mtype*(?$_2$, D) is always undefined. It means that the generated method is not overriding a method from the base class.

The last step in our method generation is to produce the method body expression. For example, suppose we are generating the methods for class `Z`, which has `Object` as a base class, and just the minimalist class `Z` is defined in the class table. A possible signature produced in the process described above could be, as follows:

$$\frac{\begin{array}{c}\text{a: Object, b: Object, this: Z} \vdash \text{?: } E_0 \qquad E_0 \texttt{ <: } \text{Z} \\ \text{class Z extends Object } \{...\} \\ \textit{mtype}(\text{mZ, Object}) = \textit{undefined}\end{array}}{\text{Z mZ(Object a, Object b) } \{ \text{ return ?; } \}} \tag{6}$$

Then, the placeholder `?` should be filled with a randomly generated expression, which process was explained in the last subsection. The important difference here is that the context is augmented with the variables and types of the formal parameters and with the special variable `this`, whose type is the class being generated. The resulting expression should be a subtype of the generated return type.

The current design shows a process for generating both a well-formed class table and a well-typed expression, which represents the *main* method of a Java program, following the formal typing rules in the specification of FJ. Next section shows how we use the generated programs for testing against some type-soundness properties. As FJ code represents a valid Java program, the randomly generated source-code can be used for testing purpose on the original language.

## 4 Validation of Semantics Properties

After the presentation of FJ language semantics and how random tests are generated, we demonstrate how QuickCheck [4] helps on testing the semantics against some properties, including those for type-soundness presented in the FJ original paper, using randomly generated programs.

Considering that testing requires additional programming, there is a natural risk that the testing code itself contain bugs [21]. In order to reduce the risk of bugs in our implementation, we have tested it with QuickCheck, by using our interpreter and the test generators. We check the following:

- That our custom generator produces only well-formed class tables.
- That our custom generator produces only well-typed expressions, according to a randomly generated class table.
- And if all generated expressions are cast-safe.

The QuickCheck library provides a way to define a property as a Haskell function. Thus, testing this property involves running the function on a finite number of inputs when the number of all inputs is infinite. This way, testing can only re-

sult in disproving the property, by finding a counter-example or leaving its validity undecided. If a counter-example is found, it can be used in order to help to fix the bug. Considering that, we started defining a function to check if generated class tables are well-formed, as the following code.

```
prop_genwellformedct :: Bool
prop_genwellformedct =
  forAll (genClassTable) $
    \ct -> Data.List.all
             (\(c,cl) -> classTyping cl Data.Map.empty ct) (Data.Map.toList
                 ct)
```

The above code uses the QuickCheck function `forAll`, which mimics the universal quantifier ∀, generating a user-defined number of instances of class tables, and testing if all produced classes inside a given class table are well-formed, by running the function `classTyping`.

We also define a function to test if the generated expressions are well-typed, as in the following piece of code. This function starts by generating an instance of a class table `ct`. After that, it randomly chooses a type `t` present in the class table. Then, it uses the produced `ct` and an empty environment, to generate an expression of type $t$. In the end, by using the function `typeof`, it checks if the expression has the type $t$.

```
prop_genwelltypedexpr :: Bool
prop_genwelltypedexpr =
  forAll (genClassTable) $
    \ct -> forAll (genType ct) $
    \t -> forAll (genExpression ct Data.Map.empty t) $
    \e -> either (const False)
                 (\(TypeClass t') -> t == t')
                 (typeof Data.Map.empty ct e)
```

As a last check for our generators, the following function tests if a produced expression is *cast-safe*, i.e., the subject expression is a subtype of the target type.

```
prop_gencastsafeexpr :: Bool
prop_gencastsafeexpr =
 forAll (genClassTable) $
  \ct -> forAll (genType ct) $
  \t -> forAll (genExpression ct Data.Map.empty t) $
  \e -> case e of
      (Cast c e) -> case (typeof Data.Map.empty ct e) of
                      Right (TypeClass t') -> subtyping t' c ct
                      _ -> False
       _ -> True
```

Thanks to these checks we found and fixed a number of programming errors in our generator, and in our interpreter implementation. Although testing can't state correctness, we gain a high-degree of confidence in using the generated programs.

We have used our test suite as a lightweight manner to check the properties of *preservation* and *progress* presented in the FJ paper. The informal (non-mechanized) proofs were also modeled as Haskell functions to be used with QuickCheck.

The preservation (subject reduction) is presented by Theorem 2.4.1 (p. 406 of [14]), stating that "*If* $\Gamma \vdash$ e: C *and* e $\rightarrow$ e′, *then* $\Gamma \vdash$ e′: C′ *for some* C′ $<:$ C.". Our function was modeled as follows:

```
prop_preservation :: Bool
prop_preservation =
 forAll (genClassTable) $
  \ct -> forAll (genType ct) $
  \t -> forAll (genExpression ct Data.Map.empty t) $
  \e -> either (const False)
         (\(TypeClass t') ->
             subtyping t' t ct)
         (case (eval' ct e) of
            Just e' ->
              typeof Data.Map.empty ct e'
            _ -> throwError (UnknownError e))
```

As we can see in the code, after generating an instance for `ct`, a type `t`, and an expression `e` of type `t`, a reduction step is performed by function `eval'` over expression `e` producing an `e'`. Then, the function `typeof` is used to obtain the type of `e'`. Last, the `subtyping` function is used to check if the expression keeps the typing relation after a reduction step.

Similarly, we modeled (as follows) a function for the progress property (Theorem 2.4.2, p. 407 [14]), which states that a well-typed expression does not get stuck.

```
prop_progress :: Bool
prop_progress =
  forAll (genClassTable) $
    \ct -> forAll (genType ct) $
    \t -> forAll (genExpression ct Data.Map.empty t) $
    \e -> isValue ct e || maybe (False) (const True) (eval' ct e)
```

This function also generates a class table, a type, and an expression of that type. Then it checks that or the expression is a value, or it can take a reduction step through the function `eval'`.

We ran many thousands of well-succeeded tests for the presented functions. As a way to measure the quality of our tests, we check how much of the code base was covered by our test suite. Such statistics are provided by the Haskell Program Coverage (HPC) tool [12]. Results of code coverage for each module (evaluator, type-checker, auxiliary functions, and total, respectively) are presented in Figure 5.

| Top Level Definitions | | Alternatives | | Expressions | |
|---|---|---|---|---|---|
| % | covered / total | % | covered / total | % | covered / total |
| 100% | 2/2 | 85% | 18/21 | 92% | 165/179 |
| 100% | 3/3 | 52% | 22/42 | 68% | 163/237 |
| 100% | 6/6 | 77% | 27/35 | 91% | 98/107 |
| 100% | 11/11 | 68% | 67/98 | 81% | 426/523 |

Fig. 5. Test coverage results.

Figure 6 presents another result of HPC, showing a piece of code of our evaluator with unreachable code highlighted.

There we can note that to reach the highlighted code it is necessary: (1) the field `f` was not found in the fields of class `c`; (2) an error processing function `eval'` for the subexpression `e`. Both cases represent stuck states, which can be only executed if we have a not well-typed expression. As stated on type soundness proofs [14], a well-typed expression does not get stuck.

Similarly, Figure 7 shows a piece of code of our type-checker with unreachable code highlighted.

We notice that the highlighted code would be executed only if: (1) we have an

```
eval' ct (FieldAccess e f) =
  if (isValue ct e) then -- R-Field
    case e of
      (CreateObject c p) ->
        case (fields c ct) of
          Just flds ->
            maybe (Nothing)
                  (\idx -> Just (p !! idx))
                  (Data.List.findIndex (\(tp,nm) -> f == nm) flds)
           _ -> Nothing
  else -- RC-Field
    maybe (Nothing)
          (\e' -> Just (FieldAccess e' f))
          (eval' ct e)
```

Fig. 6. Unreachable code on evaluation.

```
typeof :: Env -> CT -> Expr -> Either TypeError Type
typeof ctx ct (Var v) = -- T-Var
  maybe (throwError (VariableNotFound v))
        (\t -> return t)
        (Data.Map.lookup v ctx)
typeof ctx ct (FieldAccess e f) = -- T-Field
  case (typeof ctx ct e) of
    Right (TypeClass c) ->
      case (fields c ct) of
        Just flds ->
          case (Data.List.find (\(tp,nm) -> f == nm) flds) of
            Just (tp,nm) -> return tp
            _ -> throwError (FieldNotFound f)
        _ -> throwError (ClassNotFound c)
    e -> e -- Error: Expression type not found
```

Fig. 7. Unreachable code on type-checker.

undefined variable in the typing context Γ; (2) the code is using a field that is not present in the class of current expression; (3) the type of subexpression `e` could not be obtained. In all situations, we have a not well-typed program.

Finally, Figure 8 shows a piece of code of our auxiliary functions, where the highlighted code could be reached in two cases: (1) the class `c` is not present on the class table; (2) performing `fields` on a base class results in an error. This would only happen if we had a not well-typed program.

```
fields :: String -> CT -> Maybe [(Type,String)]
fields "Object" _ = Just []
fields c ct = case (Data.Map.lookup c ct) of
                Just (Class _ c'' attrs _ _) ->
                  case (fields c'' ct) of
                    Just base -> Just (base ++ attrs)
                    _ -> Nothing
                _ -> Nothing
```

Fig. 8. Unreachable code on auxiliary functions.

Although not having 100% of code coverage, our test suite was capable to verify the main safety properties present in FJ paper, by exercising on randomly generated programs of increasing size. By analyzing test coverage results, we could observe that code not reached by test cases consists of stuck states on program semantics or error control for expressions that are not well-typed.

## 5 Related Work

Property-based testing is a technique for validating code against an executable specification by automatically generating test-data, typically in a random and/or

exhaustive fashion [2]. However, the generation of random test-data for testing compilers represents a challenge by itself, since it is hard to come up with a generator of valid test data for compilers, and it is difficult to provide a specification that decides what should be the correct behavior of a compiler [22]. As a consequence of this, random testing for finding bugs in compilers and programming language tools received some attention in recent years.

The testing tool Csmith [25] is a generator of programs for the C language, supporting a large number of language features, which was used to find a number of bugs in compilers such as GCC, LLVM, etc. Le et al. [18] developed a methodology that uses differential testing for C compilers. Lindig [19] created a tool for testing the C function calling convention of the GCC compiler, which randomly generates types of functions. There are also efforts on randomly generate case tests for other languages [8]. The main difference between these projects to ours is that our generators were created by using a formal specification of typing rules. Furthermore, we used property-based testing for checking type-soundness proofs.

More specifically, Daniel et al. [5] generate random Java programs to test refactoring engines in Eclipse and NetBeans. Klein et al. [15] generated random programs to test an object-oriented library. Allwood and Eisenbach [1] also used FJ as a basis to define a test suite for the mainstream programming language in question, testing how much of coverage their approach was capable to obtain. These projects are closed related to ours since they are generating code in the object-oriented context. The difference of our approach is that we generate randomly complete classes and expressions, both well-formed and well-typed by using the formal specification of typing rules in the process of generation. Another difference is that none of them used property-based testing in their approaches.

The work of Palka, Claessen and Hughes [22] used QuickCheck library to generate $\lambda$-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 terms. 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.

There is also an effort on automatic random test generation from the definition of a type-system. The work of Fetscher et al. [10] presents a generic method for randomly generating well-typed expressions in the context of PLT Redex [9]. The works of Lampropoulos et al. [16,17] present different ways to automatically generate random expressions by using QuickChick [7], an existing tool for property-based testing in Coq. These approaches differ to ours in the sense that the authors provide tools to generate terms automatically according to a formal specification, usually by annotating the typing rules, while in our work we focus on a specific type-system of a high-level programming language.

# 6 Conclusion

In this work, we presented a type-directed heuristic for constructing random programs 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 implementations 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 bugs but also to improve understanding the meaning of the presented semantics properties.

As future work, we intend to use Coq to provide formally certified proofs that the FJ semantics 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

This work was partially supported by CAPES/Brazil. Process number: 88882.151433/2017-01.

# References

[1] Tristan O. R. Allwood and Susan Eisenbach. Tickling Java with a feather. *Electron. Notes Theor. Comput. Sci.*, 238(5):3–16, October 2009.

[2] Roberto Blanco, Dale Miller, and Alberto Momigliano. Property-based testing via proof reconstruction work-in-progress. In *LFMTP 17: Logical Frameworks and Meta-Languages: Theory and Practice*, 2017.

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

[4] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[5] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM.

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

[7] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for Coq. In *The Coq Workshop*, 2014.

[8] Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. QuickChecking refactoring tools. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, Erlang '10, pages 75–80, New York, NY, USA, 2010. ACM.

[9] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. 2009.

[10] Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In Jan Vitek, editor, *Programming Languages and Systems*, pages 383–405, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[11] Daniele Filaretti and Sergio Maffeis. 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, 2014. Springer-Verlag New York, Inc.

[12] Andy Gill and Colin Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 1–12, New York, NY, USA, 2007. ACM.

[13] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 8 edition (Java series), 2014.

[14] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[15] Casey Klein, Matthew Flatt, and Robert Bruce Findler. Random testing for higher-order, stateful programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 555–566, New York, NY, USA, 2010. ACM.

[16] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. Beginner's luck: A language for property-based generators. *CoRR*, abs/1607.05443, 2016.

[17] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2(POPL):45, 2017.

[18] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *SIGPLAN Not.*, 49(6):216–226, June 2014.

[19] Christian Lindig. Random testing of C calling conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 3–12, New York, NY, USA, 2005. ACM.

[20] Conor McBride. Djinn, monotonic. In *PAR@ ITP*, pages 14–17, 2010.

[21] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. Effect-driven QuickChecking of compilers. *Proc. ACM Program. Lang.*, 1(ICFP):15:1–15:23, August 2017.

[22] Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM.

[23] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[24] tiobe.com. TIOBE Index. https://www.tiobe.com/tiobe-index/, 04 2018. Accessed: 2018-04-09.

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.