

Generating Random Well-Typed Java Programs

Samuel Feitosa, Rodrigo Ribeiro, Andre Du Bois
Technical Report

Abstract—Currently, Java is 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 technical report addresses the problem of generating random well-typed programs in the context of Featherweight Java, a well-known object-oriented calculus, using a type-directed procedure on QuickCheck, a Haskell library for property-based testing.

I. INTRODUCTION

Currently, Java is one of the most popular programming languages [1]. It is a general-purpose, concurrent, strongly typed, class-based object-oriented language. Since its release in 1995 by Sun Microsystems, and acquired by Oracle Corporation, Java has been evolving over time, adding features and programming facilities in its new versions. For example, 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 styles [2].

The adoption of the Java language is growing for large projects, where 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 to guarantee the absence of unexpected behaviors [3]. The use of formal subsets helps in the understanding of the problem, and allows the use of automatic tools, since a certain degree of abstraction is applied, and only properties of interest are used, providing a degree of confidence that cannot be reached using informal approaches.

Creating tests for programming languages or compilers is difficult since several requirements should be respected to produce a valid and useful test case. When a person is responsible for this task, tests could be limited by human imagination, the creator can make assumptions about the implementation, impacting in the quality of the test cases, and the maintenance of such tests is also an issue when the language evolves. Because of this, there is a growing research community studying random test generation, which is not an easy task, since the generated programs should respect the constraints of the programming language compiler, such as the correct syntax, or the type-system requirements in a statically-typed language.

In this context, this work provides the formal specification of a type-directed procedure for generating Java programs, using the typing rules of Featherweight Java (FJ) [4] to generate only well-typed programs. FJ is a small core calculus with a rigorous semantic definition of the main core aspects of Java. The motivations for using the specification of FJ are that it is very compact, so we can specify our generation algorithm 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 is no formal specification of well-typed test generators for an object-oriented calculus like FJ. This work aims to fill this gap, providing the description of a generation procedure for FJ programs by using a syntax directed judgment for generating random type-correct FJ programs, adapting the approach of Palka et al. [5] in terms of QuickCheck [6]. We are aware that using only automated testing is not sufficient to ensure safety or correctness, but it can expose bugs before using more formal approaches, like formalization in a proof assistant.

Specifically, we made the following contributions:

- We provided a type-directed [7] formal specification for constructing random programs. We proved that our specification is sound with respect to FJ type system, i.e. it generates only well-typed programs.
- We implemented an interpreter¹ for FJ and the type-directed algorithm to generate random FJ programs following our formal specification using the Haskell programming language.
- We used ‘javac’ as an oracle to compile the random programs constructed through our type-directed procedure, and we used QuickCheck to check type-soundness proofs of FJ by using the generated programs².

The remainder of this text is organized as follows: Section II summarizes the FJ proposal. Section III presents the process of generating well-typed random programs in the context of FJ. Section IV proves that our generation procedure is sound with respect to FJ typing rules. Section V shows how we implement the generation rules in Haskell. Section VI demonstrates the steps to compile the generated programs with ‘javac’. Section VII shows how the results of testing type-safety properties of FJ with QuickCheck. Section VIII discusses some related works. Finally, we present the final remarks in Section IX.

¹The source-code for our Haskell interpreter and the complete test suite is available at: <https://github.com/fjpub/fj-qc/>.

²Details of implementation and experiments are presented in our technical report, which can be found at: <https://github.com/fjpub/fj-qc/raw/master/tr.pdf>.

II. FEATHERWEIGHT JAVA

Featherweight Java (FJ) [4] 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 *public static void main* method of Java.

FJ has a similar relation with Java, like λ -calculus has with Haskell. It offers similar operations, providing classes, methods, attributes, inheritance and dynamic casts with semantics close to Java's. The Featherweight Java project favors simplicity over expressivity and offers only five ways to create terms: object creation, method invocation, attribute access, casting and variables [4]. 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 of them: `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 [8]. 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 [8]. Next, we show how FJ was formalized by the authors [4].

A. 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, E, and F 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. We let $\varphi : L \rightarrow C$ denote a function that returns a class name (C) from a given class declaration (L). Throughout this paper, we write \overline{C} as shorthand for a possibly empty sequence C_1, \dots, C_n (similarly for \overline{f} , \overline{x} , etc.). An empty sequence is denoted by \bullet , and the length of a sequence \overline{x} is written $\#\overline{x}$. The inclusion of an item x in a sequence \overline{X} is denoted by $x : \overline{X}$, following Haskell notation for lists. We consider that a finite mapping M is just a sequence of key-value pairs. Notation $M(K) = V$ if $K V \in M$. Following common practice, we let the metavariable Γ denote an arbitrary typing environment which consists of a finite mapping between variables and types.

Syntax

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

$$\begin{array}{c}
 C <: C \\
 \\
 \frac{C <: D \quad D <: E}{C <: E} \\
 \\
 \frac{CT(C) = \text{class } C \text{ extends } D \{ \dots \}}{C <: D}
 \end{array}$$

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{Cf} pairing the class of each field with its name, for all fields declared in the given class and all of its superclasses. When

dealing with methods it was used $m \notin \overline{M}$ when the method definition with name m is not included in C . The rules for *method type lookup* show how is obtained the type of method m in class C . The first return a pair, written $\overline{B} \rightarrow B$, 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

$$fields(Object) = \bullet$$

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

Method type lookup

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

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

Method body lookup

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

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

Fig. 3. Auxiliary definitions.

B. 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 the proposed constructors. 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 [4].

We can note the use of an environment Γ , which is a finite mapping from variables to types, written $\overline{x} : \overline{C}$. The typing judgment for expressions has the form $\Gamma \vdash e : C$, meaning that in the environment Γ , 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 Γ . If the variable x is not contained in Γ , 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 e_0 , which results in the type C_0 . Then it obtains the *fields* of class C_0 , matching the position of f_i in the resultant list, to return the respective type C_i . The rule T-Invk also applies the typing judgment on the subexpression e_0 , which results in the type C_0 , then it uses *mtype* to get the formal parameter types \overline{D} and the return type C . The formal parameter types are used to check if the actual parameters \overline{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*. The 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 Γ 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 match the position i of the field f_i in the resulting list, returning the value 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 λ -calculus. The last rule R-Cast refers to *cast processing*, where the same subexpression *new* $C(\overline{e})$ is returned in case the subject class C is subtype of the target class D . There are also five congruence rules (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

Expression typing

$$\begin{array}{c}
\frac{}{\Gamma \vdash x : \Gamma(x)} \text{ [T-Var]} \\
\\
\frac{\Gamma \vdash e_0 : C_0 \quad \text{fields}(C_0) = \bar{C} \bar{f}}{\Gamma \vdash e_0.f_i : C_i} \text{ [T-Field]} \\
\\
\frac{\text{mtype}(m, C_0) = \bar{D} \rightarrow C \quad \Gamma \vdash e_0 : C_0 \quad \Gamma \vdash \bar{e} : \bar{C} \quad \bar{C} <: \bar{D}}{\Gamma \vdash e_0.m(\bar{e}) : C} \text{ [T-Invk]} \\
\\
\frac{\text{fields}(C) = \bar{D} \bar{f} \quad \Gamma \vdash \bar{e} : \bar{C} \quad \bar{C} <: \bar{D}}{\Gamma \vdash \text{new } C(\bar{e}) : C} \text{ [T-New]} \\
\\
\frac{\Gamma \vdash e_0 : D \quad D <: C}{\Gamma \vdash (C) e_0 : C} \text{ [T-UCast]} \\
\\
\frac{\Gamma \vdash e_0 : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C) e_0 : C} \text{ [T-DCast]} \\
\\
\frac{\Gamma \vdash e_0 : D \quad C \not<: D \quad D \not<: C \quad \text{stupid warning}}{\Gamma \vdash (C) e_0 : C} \text{ [T-SCast]}
\end{array}$$

Method typing

$$\frac{\bar{x} : \bar{C}, \text{this} : C \vdash e_0 : E_0 \quad E_0 <: C_0 \quad \text{class } C \text{ extends } D \{ \dots \} \quad \text{if } \text{mtype}(m, D) = \bar{D} \rightarrow D_0, \text{ then } \bar{C} = \bar{D} \text{ and } C_0 = D_0}{C_0 \text{ m}(\bar{C} \bar{x}) \{ \text{return } e_0; \} \text{ OK in } C}$$

Class typing

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

Evaluation

$$\begin{array}{c}
\frac{\text{fields}(C) = \bar{C} \bar{f}}{(\text{new } C(\bar{v})).f_i \longrightarrow v_i} \text{ [R-Field]} \\
\\
\frac{\text{mbody}(m, C) = (\bar{x}, e_0)}{(\text{new } C(\bar{v})).m(\bar{d}) \longrightarrow [\bar{d} \mapsto \bar{x}, \text{new } C(\bar{v}) \mapsto \text{this}] e_0} \text{ [R-Invk]} \\
\\
\frac{C <: D}{(D) (\text{new } C(\bar{v})) \longrightarrow \text{new } C(\bar{v})} \text{ [R-Cast]}
\end{array}$$

Fig. 4. Typing and evaluation rules.

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.

III. 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 [9].

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 [5].

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 [6].

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 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 technical report, we generalized the approach of [5] for generating random programs considering that FJ has a nominal type system instead of a structural one. In this way, we specified a formal generation rule derived from each typing rule, both for expression generation and class table generation. The generation rules are explained as follows.

A. Expression Generation

We assume that a class table CT is a finite mapping between names and its corresponding classes. We let $\text{dom}(CT)$ denote the set of names in the domain of the finite mapping CT . The generation algorithm uses a function $\xi : [a] \rightarrow a$, which returns a random element from an input list. We slightly abuse notation by using set operations on lists (sequences) and its meaning is as usual.

The expression generation is represented by the following judgment:

$$CT ; \Gamma ; C \rightarrow e \quad (1)$$

There CT is a class table, Γ is a typing environment, C is a type name and e is the produced expression.

For generating *variables*, we just need to select a name from the typing environment, which has a type C .

$$\frac{}{CT ; \Gamma ; C \rightarrow \xi(\{x \mid \Gamma(x) = C\})} \text{ [G-Var]}$$

For *fields access*, we first need to generate a list of candidate type names for generating an expression with type C' which has at least one field whose type is C . We name such list $\overline{C_c}$:

$$\overline{C_c} = \{C_1 \mid C_1 \in \text{dom}(CT) \wedge \exists x. C \ x \in \text{fields}(C_1)\}$$

Now, we can build a random expression by using a type randomly chosen from it.

$$\begin{aligned} C' &= \xi(\overline{C_c}) \\ CT ; \Gamma ; C' &\rightarrow e \end{aligned}$$

Since type C' can have more than one field with type C , we need to choose one of them (note that, by construction, such set is not empty).

$$C \ f = \xi(\{C \ x \mid C \ x \in \text{fields}(C')\})$$

The rule G-Field combines these previous steps to generate a field access expression:

$$\frac{\begin{aligned} \overline{C_c} &= \{C_1 \mid C_1 \in \text{dom}(CT) \wedge \\ &\quad \exists x. C \ x \in \text{fields}(C_1)\} \\ C' &= \xi(\overline{C_c}) \\ CT ; \Gamma ; C' &\rightarrow e \\ C \ f &= \xi(\{C \ x \mid C \ x \in \text{fields}(C')\}) \end{aligned}}{CT ; \Gamma ; C \rightarrow e.f} \text{ [G-Field]}$$

For *method invocations*, we first need to find all classes which have methods signatures with return type C . As before, we name such candidate class list as $\overline{C_c}$.

$$\overline{C_c} = \{C_1 \mid C_1 \in \text{dom}(CT) \wedge \exists m \ \bar{D}. \text{mtype}(m, C_1) = \bar{D} \rightarrow C\}$$

Next, we need to generate an expression e_0 from a type chosen from $\overline{C_c}$, we name such type as C' .

$$\begin{aligned} C' &= \xi(\overline{C_c}) \\ CT ; \Gamma ; C' &\rightarrow e_0 \end{aligned}$$

From such type C' , we need to chose which method with return type C will be called. For this, we select a random signature from its list of candidate methods.

$$\begin{aligned} \overline{M_c} &= \{(m, \bar{D} \rightarrow C) \mid \exists m. \text{mtype}(m, C') = \bar{D} \rightarrow C\} \\ (m', \bar{D}' \rightarrow C) &= \xi(\overline{M_c}) \end{aligned}$$

Next, we need to generate arguments for all formal parameters of method m' . For this, since arguments could be of any subtype of the formal parameter type, we need to choose it from the set of all candidate subtypes.

First, we define a function called *subtypes*, which return a list of all subtypes of some type.

$$\begin{aligned} \text{subtypes}(CT, \text{Object}) &= \{\text{Object}\} \\ \text{subtypes}(CT, C) &= \{C\} \cup \text{subtypes}(CT, D), \text{ if class } C \text{ extends } D \in CT \end{aligned}$$

Using this function, we can build the list of arguments for a method call.

$$\bar{a} = \{e \mid D \in \bar{D}' \wedge CT ; \Gamma ; \xi(\text{subtypes}(CT, D)) \rightarrow e\}$$

The rule G-Invk combine all these previous steps to produce a method call.

$$\frac{\begin{aligned} \overline{C_c} &= \{C_1 \mid C_1 \in \text{dom}(CT) \wedge \\ &\quad \exists m \ \bar{D}. \text{mtype}(m, C_1) = \bar{D} \rightarrow C\} \\ C' &= \xi(\overline{C_c}) \\ CT ; \Gamma ; C' &\rightarrow e_0 \\ \overline{M_c} &= \{(m, \bar{D} \rightarrow C) \mid \exists m. \text{mtype}(m, C') = \bar{D} \rightarrow C\} \\ (m', \bar{D}' \rightarrow C) &= \xi(\overline{M_c}) \\ \bar{a} &= \{e \mid D \in \bar{D}' \wedge CT ; \Gamma ; \xi(\text{subtypes}(CT, D)) \rightarrow e\} \end{aligned}}{CT ; \Gamma ; C \rightarrow e_0.m'(\bar{a})} \text{ [G-Invk]}$$

The generation of a random *object creation* expression is straightforward: First, we need to get all field types of the class C and produce arguments for C 's constructor parameters, as demonstrated by rule G-New.

$$\frac{\begin{aligned} \bar{F} &= \{C' \mid C' \ f \in \text{fields}(C)\} \\ \bar{a} &= \{e \mid F \in \bar{F} \wedge CT ; \Gamma ; \xi(\text{subtypes}(CT, F)) \rightarrow e\} \end{aligned}}{CT ; \Gamma ; C \rightarrow \text{new } C(\bar{a})} \text{ [G-New]}$$

We construct *upper casts* expressions for a type C using G-UCast rule.

$$\frac{\begin{aligned} \bar{D} &= \text{subtypes}(CT, C) \\ CT ; \Gamma ; \xi(\bar{D}) &\rightarrow e \end{aligned}}{CT ; \Gamma ; C \rightarrow (C) \ e} \text{ [G-UCast]}$$

Although we do not start a program with *downcasts* or *stupid casts*, because expressions generated by these typing rules can reduce to *cast unsafe* terms [4], we defined the generation process in the rules G-DCast and G-SCast, since they can be used to build inner subexpressions.

For generating downcasts, first we need the following function, which returns the set of super types of a given class name C .

$$\begin{aligned} \text{supertypes}(CT, \text{Object}) &= \bullet \\ \text{supertypes}(CT, C) &= \{D\} \cup \text{supertypes}(CT, D), \\ &\quad \text{if class } C \text{ extends } D \in CT \end{aligned}$$

Then, we can produce the rule G-DCast to generate a downcast expression.

$$\frac{\bar{D} = \text{supertypes}(\text{CT}, \text{C}) \quad \text{CT} ; \Gamma ; \xi(\bar{D}) \rightarrow e}{\text{CT} ; \Gamma ; \text{C} \rightarrow (\text{C}) e} \quad [\text{G-DCast}]$$

The generation of stupid casts has a similar process, except that it generates a list of unrelated classes, as we can see in the first line of the rule G-SCast .

$$\frac{\bar{C} = \text{dom}(\text{CT}) - (\text{subtypes}(\text{CT}, \text{C}) \cup \text{supertypes}(\text{CT}, \text{C})) \quad \text{CT} ; \Gamma ; \xi(\bar{C}) \rightarrow e}{\text{CT} ; \Gamma ; \text{C} \rightarrow (\text{C}) e} \quad [\text{G-SCast}]$$

Considering the presented generation rules, we are able to produce well-typed expressions for each constructor of FJ.

B. Class Table Generation

To generate a class table, we assume the existence of an enumerable set \bar{C}_n of class names and \bar{V}_n of variable names. The generation rules are parameterized by an integer n which determines the number of classes that will populate the resulting table, a limit m for the number of members in each class and a limit p for the number of formal parameters in the generated methods. This procedure is expressed by the following judgment:

$$\text{CT} ; n ; m ; p \rightarrow \text{CT}'$$

It is responsible to generate n classes using as input the information in class table CT (which can be empty), each class will have up to m members. As a result, the judgment will produce a new class table CT' . As expected, this judgment is defined by recursion on n :

$$\frac{}{\text{CT} ; 0 ; m ; p \rightarrow \text{CT}} \quad [\text{CT-Base}]$$

$$\frac{\text{CT} ; m ; p \rightarrow L \quad \varphi(L) L : \text{CT} ; n ; m ; p \rightarrow \text{CT}'}{\text{CT} ; n + 1 ; m ; p \rightarrow \text{CT}'} \quad [\text{CT-Step}]$$

Rule CT-Base specifies when the class table generation procedure stops. Rule CT-Step uses a specific judgment to generate a new class, inserts it in the class table CT , and generate the next n classes using the recursive call $\varphi(L) L : \text{CT} ; n ; m ; p \rightarrow \text{CT}'$. The following judgment presents how classes are generated:

$$\text{CT} ; m ; p \rightarrow \text{C}$$

It generates a new class, with at most m members, with at most p formal parameters in each method, using as a starting point a given class table. First, we create a new name which is not in the domain of the input class table, using:

$$\text{C} = \xi(\bar{C}_n - (\text{dom}(\text{CT}) \cup \{\text{Object}\}))$$

This rule selects a random class name from the set \bar{C}_n excluding the names in the domain of CT and Object . Next,

we need to generate a valid super class name, which can be anyone of the set formed by the domain of current class table CT and Object :

$$\text{D} = \xi(\text{dom}(\text{CT}) \cup \{\text{Object}\})$$

After generating a class name and its super class, we need to generate its members. For this, we generate random values for the number of fields and methods, named f_n and m_n , respectively. Using such parameters we build the fields and methods for a given class.

Field generation is straightforward. It proceeds by recursion on n , as shown below. Note that we maintain a set of already used attribute names \bar{U}_n to avoid duplicates.

$$\frac{}{\text{CT} ; 0 ; \bar{U}_n \rightarrow \bullet} \quad [\text{G-Fields-Base}]$$

$$\frac{\text{C} = \xi(\text{dom}(\text{CT}) \cup \{\text{Object}\}) \quad f = \xi(\bar{V}_n - \bar{U}_n) \quad \text{CT} ; n ; f : \bar{U}_n \rightarrow \bar{C} \bar{f}}{\text{CT} ; n + 1 ; \bar{U}_n \rightarrow \text{C } f : \bar{C} \bar{f}} \quad [\text{G-Fields-Step}]$$

Generation of the method list proceeds by recursion on m , as shown below. We also maintain a set of already used method names \bar{U}_n to avoid method overload, which is not supported by FJ. The rule G-Method-Step uses a specific judgment to generate each method, which is described by rule G-Method .

$$\frac{}{\text{CT} ; \text{C} ; 0 ; p ; \bar{U}_n \rightarrow \bullet} \quad [\text{G-Methods-Base}]$$

$$\frac{x = \xi(\bar{V}_n - \bar{U}_n) \quad \text{CT} ; \text{C} ; p ; x \rightarrow \text{M} \quad \text{CT} ; \text{C} ; m ; p ; x : \bar{U}_n \rightarrow \bar{M}}{\text{CT} ; \text{C} ; m + 1 ; p ; \bar{U}_n \rightarrow \text{M} : \bar{M}} \quad [\text{G-Methods-Step}]$$

The rule G-Method uses an auxiliary judgment for generating formal parameters (note that we can generate an empty parameter list). To produce the expression, which defines the method body, we build a typing environment using the formal parameters and a variable this to denote this special object. Also, such expression is generated using a type that can be any of the possible subtypes of the method return type C_0 .

$$\begin{aligned} n &= \xi([0..(p - 1)]) \\ \text{CT} ; n ; \bullet &\rightarrow \bar{C} \bar{x} \\ \text{C}_0 &= \xi(\text{dom}(\text{CT}) \cup \{\text{Object}\}) \\ \Gamma &= \bar{C} \bar{x}, \text{this} : \text{C} \\ \bar{D} &= \text{subtypes}(\text{CT}, \text{C}_0) \\ \text{E}_0 &= \xi(\bar{D}) \\ \text{CT} ; \Gamma ; \text{E}_0 &\rightarrow e \end{aligned} \quad [\text{G-Method}]$$

$$\text{CT} ; \text{C} ; p ; m \rightarrow (\text{C}_0 \text{ m } (\bar{C} \bar{x}) \{\text{return } e;\})$$

We create the formal parameters for methods using a simple recursive judgment that keeps a set of already used variable names \bar{U}_n to ensure that all variables produced are distinct.

$$\frac{}{CT ; 0 ; \bar{U}_n \rightarrow \bullet} \text{ [G-Param-Base]}$$

$$\frac{\begin{array}{l} C = \xi(\text{dom}(CT) \cup \{\text{Object}\}) \\ x = \xi(\bar{V}_n - \bar{U}_n) \\ CT ; n ; x : \bar{U}_n \rightarrow \bar{C} \bar{x} \end{array}}{CT ; n + 1 ; \bar{U}_n \rightarrow (C \ x : \bar{C} \bar{x})} \text{ [G-Param-Step]}$$

Finally, using the generated class name and its super class, we build its constructor definition using the judgment:

$$CT ; C ; D \rightarrow K$$

Rule G-Const represents the process to generate the constructor.

$$\frac{\begin{array}{l} \bar{D} \bar{g} = \text{fields}(D) \\ \bar{C} \bar{f} = \text{fields}(C) - \bar{D} \bar{g} \end{array}}{CT ; C ; D \rightarrow C(\bar{D} \bar{g}, \bar{C} \bar{f}) \{ \text{super}(\bar{g}); \text{this}.\bar{f} = \bar{f}; \}} \text{ [G-Const]}$$

The process for generating a complete class is summarized by rule G-Class, which is composed by all previously presented rules.

$$\frac{\begin{array}{l} C = \xi(\bar{C}_n - (\text{dom}(CT) \cup \{\text{Object}\})) \\ D = \xi(\text{dom}(CT) \cup \{\text{Object}\}) \\ \text{fn} = \xi([1..m]) \\ \text{mn} = \xi([1..(m - \text{fn})]) \\ CT' = C \text{ (class } C \text{ extends } D \{ \}) : CT \\ CT' ; \text{fn} ; \bullet \rightarrow \bar{C} \bar{f} \\ CT'' = C \text{ (class } C \text{ extends } D \{ \bar{C} \bar{f} \}) : CT \\ CT'' ; C ; \text{mn} ; p ; \bullet \rightarrow \bar{M} \\ CT' ; C ; D \rightarrow K \end{array}}{CT ; m ; p \rightarrow (\text{class } C \text{ extends } D \{ \bar{C} \bar{f}; K \bar{M} \})} \text{ [G-Class]}$$

Considering the presented generation rules, we are able to fill a class table with well-formed classes in respect to FJ typing rules.

IV. SOUNDNESS OF PROGRAM GENERATION

The generation algorithm described in the previous section produces only well-typed FJ programs.

Lemma 1 (Soundness of expression generation). *Let CT be a well-formed class table. For all Γ and $C \in \text{dom}(CT)$, if $CT ; \Gamma ; C \rightarrow e$ then exists D , such that $\Gamma \vdash e : D$ and $D <: C$.*

Proof. The proof proceeds by induction on the derivation of $CT ; \Gamma ; C \rightarrow e$ doing a case analysis on the last rule used to deduce $CT ; \Gamma ; C \rightarrow e$. We show some cases of the proof.

Case (G-Var): Then, $e = x$, for some variable x . By rule G-Var, $x = \xi(\{y \mid \Gamma(y) = C\})$ and from this we can deduce that $\Gamma(x) = C$ and the conclusion follows by rule T-Var.

Case (G-Invk): Then, $e = e_0.m(\bar{e})$ for some e_0 and \bar{e} ; $CT ; \Gamma ; C' \rightarrow e_0$, for some C' ; there exists $(m, \bar{D}' \rightarrow C)$, such that $mtype(m, C') = \bar{D}' \rightarrow C$ and for all $e' \in \bar{e}$, $D \in \bar{D}'$, $CT ; \Gamma ; \xi(\text{subtypes}(CT, D)) \rightarrow e'$. By the induction hypothesis, we

have that: $\Gamma \vdash e_0 : D'$, $D' <: C'$, for all $e' \in \bar{e}$, $D \in \bar{D}'$. $\Gamma \vdash e' : B$, $B <: D$ and the conclusion follows by the rule T-Invk and the definition of subtyping relation. \square

Lemma 2 (Soundness of subtypes). *Let CT be a well-formed class table and $C \in \text{dom}(CT)$. For all D . if $D \in \text{subtypes}(CT, C)$ then $C <: D$.*

Proof. Straightforward induction on the structure of the result of $\text{subtypes}(CT, C)$. \square

Lemma 3 (Soundness of method generation). *Let CT be a well-formed class table and $C \in \text{dom}(CT) \cup \{\text{Object}\}$. For all p and m , if $CT ; C ; p ; m \rightarrow C_0 \ m \ (\bar{C} \bar{x}) \{ \text{return } e; \}$ then $C_0 \ m \ (\bar{C} \bar{x}) \{ \text{return } e; \}$ OK in C .*

Proof. By rule G-Method, we have that:

- $\bar{C} \subseteq \text{dom}(CT)$
- $\Gamma = \{ \bar{C} \bar{x}, \text{this} : C \}$
- $C_0 = \xi(\text{dom}(CT) \cup \{\text{Object}\})$
- $\bar{D} = \text{subtypes}(CT, C_0)$
- $CT ; \Gamma ; E_0 \rightarrow e$
- $E_0 = \xi(\bar{D})$

By Lemma 2, we have that for all $D \in \bar{D}$, $C_0 <: D$.

By Lemma 1, we have that $\Gamma \vdash e : E'$ and $E' <: E_0$.

Since CT is well-formed, then $mtype(m, C) = \bar{C} \rightarrow C_0$ and the conclusion follows by rule *method typing* and the definition of the subtyping relation. \square

Lemma 4 (Soundness of class generation). *Let CT be a well-formed class table. For all m, p , if $CT ; m ; p \rightarrow CD$ then CD OK.*

Proof. By rule G-Class, we have that:

- $CD = \text{class } C \text{ extends } D \{ \bar{C} \bar{f}; K \bar{M} \}$
- $C = \xi(\bar{C}_n - (\text{dom}(CT) \cup \text{Object}))$
- $D = \xi(\text{dom}(CT) \cup \text{Object})$
- $\text{fn} = \xi([1..m])$
- $\text{mn} = \xi([1..(m - \text{fn})])$
- $CT' = C \text{ (class } C \text{ extends } D \{ \}) : CT$
- $CT' ; \text{fn} \rightarrow \bar{C} \bar{f}$
- $CT'' = C \text{ (class } C \text{ extends } D \{ \bar{C} \bar{f}; \}) : CT$
- $CT'' ; C ; \text{mn} ; p ; \bullet \rightarrow \bar{M}$
- $CT' ; C ; D \rightarrow K$

By Lemma 3, we have that for all m . $m \in \bar{M}$, m OK.

By rule (G-Const) we have that $K = C \ (\bar{D} \bar{g}, \bar{C} \bar{f}) \{ \text{super}(\bar{g}); \text{this}.\bar{f} = \bar{f}; \}$, where $\bar{D} \bar{g} = \text{fields}(D)$.

The conclusion follows by rule *class typing*. \square

Lemma 5. *Let CT be a well-formed class table. For all n, m and p , if $CT ; n ; m ; p \rightarrow CT'$ then for all $C, D \in \text{dom}(CT')$, if $C <: D$ and $D <: C$ then $CT(C) = CT(D)$.*

Proof. By induction on n .

Case $n = 0$: We have that $CT' = CT$. Conclusion follows by the fact that CT is a well-formed class table.

Case $n = n' + 1$: Suppose $C, D \in \text{dom}(CT')$, $C <: D$ and $D <: C$. By the induction hypothesis we have that for all CT_1 ,

$C', D' \in CT_1$, if $C' <: D'$ and $D' <: C'$ then $C' = D'$. Let L be a class such $CT ; m ; p \rightarrow L$. By Lemma 4, we have L OK in CT . By the induction hypothesis on $\varphi(L) L : CT ; n ; p \rightarrow CT'$ we have the desired conclusion. \square

Lemma 6 (Soundness of class table generation). *Let CT be a well-formed class table. For all n, m and p , if $CT ; n ; m ; p \rightarrow CT'$ then CT' is a well-formed class table.*

Proof. By induction on n .

Case $n = 0$: We have that $CT' = CT$ and the conclusion follows.

Case $n = n' + 1$: By rule CT-Step we have that:

- $CT ; m ; p \rightarrow L$
- $\varphi(L) L : CT ; n ; m ; p \rightarrow CT'$

By Lemma 4, we have that L OK. By the induction hypothesis we have that CT' is a well-formed class table. By Lemma 5, we have that subtyping in CT' is antisymmetric. Conclusion follows by the definition of a well-formed class table. \square

Theorem 1 (Soundness of program generation). *For all n, m and p , if $\bullet ; n ; m ; p \rightarrow CT$ then:*

- (1) CT is a well-formed class table.
- (2) For all $C \in CT$, we have C OK.

Proof. Corollary of Lemmas 4, 5 and 6. \square

V. IMPLEMENTATION

In this section we briefly discuss the implementation of our test generators, which was conducted by using the Haskell language and the QuickCheck library. The function $\xi : [a] \rightarrow a$ (presented in Section III) is represented by the QuickCheck function `oneof`, responsible to return a random element of a given list, and was used in the implementation of all the previously defined rules.

A. Expression Generation

The implementation of rule G-Var is shown below, where a variable is obtained from a given context Γ , represented by the variable `ctx`.

```
pickVar :: Env -> Type -> Maybe (Gen Expr)
pickVar ctx t = maybeElements [Var x | (x,
    t') <- Data.Map.assocs ctx, t' == t]
```

For the rules G-Field, G-Invk, and for *casts*, we had to prepare a candidate list for each rule, by reading the class table CT . As an example, the following function presents how we obtain a candidate list for rule G-Field. As we can note, for each class in CT , we check if there exist a field of the given type t , and it is generated a *field access* expression. This function returns a list of *field access* expressions.

```
cfields :: Int -> Env -> CT -> String -> [Gen Expr]
cfields size ctx ct t =
    Data.List.concatMap
        (\k -> case (fields k ct) of
```

```
Just lst -> Data.List.concatMap
    (\((TypeClass t'), f) ->
        if (t == t') then
            [genFieldAccess
                size ctx ct
                k f]
        else
            [])
        ) lst
    -> []
) ("Object" : keys ct)

The process is similar to generate a candidate list for the
rule G-Invk, except here, we check if there exist a method
that returns the given type  $t$  in some class, then generating a
method invocation expression. The code can be seen below.

cmeths :: Int -> Env -> CT -> String ->
    [Gen Expr]
cmeths size ctx ct t =
    Data.List.concatMap
        (\k -> case (methods k ct) of
            Just lst ->
                Data.List.concatMap (\(Method
                    (TypeClass t') m pt _) ->
                    if (t == t') then
                        let pt' = Data.List.map
                            (\(TypeClass t'', _) ->
                                t'') pt
                        in [genMethodInvk size
                            ctx ct k m pt']
                    else
                        []
                ) lst
            -> []
        ) ("Object" : keys ct)
```

For *casts*, the process verifies the *subtyping* or *supertyping* relation of the given type t with all classes in the class table. For example, the following code shows how a candidate list for the rule G-UCast is filled. Basically, for each class on CT , it checks if this class is a subtype of the given type t . The process is similar to rule G-DCast, except that there the class on CT should be supertype of the given type t , and in the rule G-SCast, the given type t should be unrelated to the class in CT .

```
cucast :: Int -> Env -> CT -> String -> [Gen Expr]
cucast size ctx ct t =
    Data.List.concatMap
        (\k -> if (subtyping k t ct) then
            [genCast size ctx ct t k]
        else
            [])
        ) ("Object" : keys ct)
```

The process for generating a *field access* expression is done by the following function. It receives the class name to generate a subexpression, and the field name of the desired type.

```
genFieldAccess :: Int -> Env -> CT -> String
    -> String -> Gen Expr
genFieldAccess size ctx ct t f =
    do e <- genExpr (size `div` 2) ct ctx t
    return (FieldAccess e f)
```


To generate a *method invocation* expression, the next function receives a class name to generate the subexpression, a method name, and a list of type names to generate the method parameters.

```
genMethodInvk :: Int -> Env -> CT -> String
-> String -> [String] -> Gen Expr
genMethodInvk size ctx ct t m pt =
  do e <- genExpr (size `div` 2) ct ctx t
  el <- Control.Monad.mapM (genExpr (size
    `div` 4) ct ctx) pt
  return (MethodInvk e m el)
```

For *casts* the process is similar, as we can see below.

```
genCast :: Int -> Env -> CT -> String ->
String -> Gen Expr
genCast size ctx ct t tc = do e <- genExpr
  (size `div` 2) ct ctx tc
return (Cast t e)
```

We can note the use of an integer when calling the `genExpr` function. This `size` variable is used to control the depth of the recursion procedure, avoid the generation of giant subexpressions.

To generate an expression for rule G-New, basically we have to generate the constructor parameters of a given class, similarly to what is done for method parameters.

By using these functions, we have a candidate list for each of the FJ constructors. The function `genExpr` uses each of these lists, and selects one potential candidate of each constructor (using the function `oneof`), adding it in a new list. Then, it selects one element of this list to be generated expression. This guarantee a uniform distribution for each of the FJ constructors.

B. Class Table Generation

We start off by defining three generators, one for class names (`genClassName`), one for variable names (`genVar`), and another for types (`genType`). 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 (rules CT-Base and CT-Step), using a list of non-duplicated class names. For each element of that list, our generator produces a class (by using the rule G-Class) and inserts it in the class table. The code to generate a class is shown below.

```
genClass :: CT -> String -> String -> Gen
Class
genClass ct c b =
  do attrs <- genAttrs ct c b
  meths <- genMethods (Data.Map.insert c
    (cl attrs []) ct) c b
  return (Class c b
    -- Attributes
    (Data.List.map (\(k,p) ->
      (TypeClass k,p)) attrs)
    -- Constructor
```

```
(formatConstr ct c b
  -- Constr Params
  (Data.List.map (\(k,p) ->
    (TypeClass k,p)) attrs))
  -- Methods
  meths)
where cl at mt =
  (Class c b
    -- Attrs
    (Data.List.map (\(k,p) -> (TypeClass
      k, p)) at)
    -- Constructor
    (formatConstr ct c b
      (Data.List.map (\(k,p) ->
        (TypeClass k, p)) at))
    mt) -- Methods
```

The function `genClass` is called recursively to produce all classes to fill the class table `CT`, where each class contains attributes, constructor, and methods. The generation procedure will be explained next.

The process for filling the class components is divided into two parts. The first is demonstrated in the code below, where a list of attributes is randomly generated, following the recursive rules G-Fields-Base and G-Fields-Step. 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 , and a list of non-duplicate variable names with size n is generated. As we can note in the generation rule, the function `fields` is called with the base class `Object`, returning an empty sequence. After that, a constructor is formatted according to rule G-Constr.

```
genAttrs :: CT -> String -> String -> Gen
[(String,String)]
genAttrs ct c b =
  do n <- choose (0,5)
  tl <- vectorOf n (genType ct)
  vl <- (vectorOf n (genVar))
  tl' <- Control.Monad.mapM
    (\n -> if (subtyping n c ct) then
      return "Object"
    else
      return n
    ) tl
  case (fields b ct) of
    Just bfls ->
      return (zip tl' ((nub vl) Data.List.\
        (snd (unzip bfls))))
    _ -> return (zip tl' (nub vl))
```

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`.

The second part to conclude the class generation concerns to method generation following the rule G-Method. The process starts generating a random number n , which represents the number of methods to be generated for a 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). The source-code to generate a method can be seen below.

```
genMethod :: CT -> String -> String -> Gen
  Method
genMethod ct c b =
  do n <- choose (0,3) -- Number of params
  tl <- vectorOf n (genType ct) -- Param types
  vl <- vectorOf n (genVar) -- Name of params
  m <- genVar -- Method name
  r <- genType ct -- Return type
  e <- genExpression ct -- Body expression
    (fromList (("this", TypeClass c) :
      (zip (nub vl) (Data.List.map (\t
        -> TypeClass t) tl))))
  ) r
  return (Method
    (TypeClass r)
    ('m':m ++ c)
    (zip (Data.List.map (\t ->
      TypeClass t) tl) (nub vl))
    e)
```

The presented functions are responsible to generate complete random classes using all the possible constructors of FJ. For space reasons, we omit some minor functions.

VI. COMPILING GENERATED PROGRAMS

The previous sections shown how we implemented our type-directed generation procedure using Haskell. The presented process is responsible to generate the internal structures of a program, i.e., the abstract syntax tree (AST). From that ASTs, we had to produce regular Java code, in order to compile it using the standard Java compiler. Basically, we encode instances of the Haskell standard class `Show`, to generate strings representing the ASTs in the Java syntax. As an example, the following code shows how expressions were converted from the ASTs to Java.

```
instance Show Expr where
  show (Var v) = v
  show (FieldAccess e f) = show e ++ "." ++ f
  show (MethodInvk e m p) = show e ++ "." ++
    m ++ "(" ++ showExprs p ++ ")"
  show (CreateObject c p) = "new " ++ c ++
    "(" ++ showExprs p ++ ")"
  show (Cast c e) = "(" ++ c ++ " " ++ show
    e ++ ")"
```

Since FJ has only a *main* expression, we created a function in Haskell to format the main class of Java, as follows.

```
formatExpr :: Expr -> String
formatExpr e = "class FJMain {\n" ++
  "  public static void\n    main(String args[]) {\n" ++
  "    System.out.println((" ++
  "      show e ++\n    ").toString());\n" ++
  "  }\n" ++
  "}\n"
```

The process is similar to convert the AST of a class to Java code. We created some auxiliary functions (`showAttrs`, `showMethods`, etc.) to facilitate the formatting process.

```
instance Show Class where
  show (Class c b at cn m) = "class " ++ c ++
    " extends " ++ b ++ " {\n" ++
    showAttrs at ++ show cn ++ "\n" ++
    showMethods m ++ "\n}"
```

As a last example, below we show how ASTs of constructors are converted to Java syntax.

```
instance Show Constr where
  show (Constr c fp at ths) = c ++ "(" ++
    showFormalParams fp ++
    ") {\n  super(" ++ showParams at ++
    ");\n" ++
    showAttrAssign ths ++ "\n  }"
```

After generating the regular Java code by the ASTs of FJ, we created a QuickCheck property to export the generated code to a Java file, and to compile it using the ‘javac’ compiler (the closest implementation of Java Specification Language), which was assumed as an oracle for our algorithm. The function `prop_compile` (shown below) first generates a class table, a type, and an expression. Then it writes the code into a Java file (`program.java`), and invokes the ‘javac’ compiler passing this program as parameter, checking for success or failure in the process. In case of failure, QuickCheck presents the generated code as a counter example of the property.

```
prop_compile =
  forAll (genClassTable) $
    \ct -> forAll (genType ct) $
      \t -> forAll (genExpression ct
        Data.Map.empty t) $
        \e -> monadicIO $
          do f <- run (writeFile "program.java"
            (formatJavaProgram ct e))
            (ex,out,err) <- run
              (readProcessWithExitCode "javac"
                ["program.java"] "")
            assert (ex == ExitSuccess)
```

By using this function, we were able to find two problems in our generation, when compiling it with ‘javac’. The first occurred when generating an expression with a *cast*, where we notice lack of parenthesis on the cast expression, which caused an error of precedence. Another problem found was the size limit for a method, since the Java Virtual Machine specification limits the size of generated Java byte code for each method in a class to the maximum of 64Kb. This limitation caused the JVM to throw `java.lang.VerifyError` at runtime when the method size exceeded this limit. Both bugs were easily corrected by using the presented counter examples.

VII. VALIDATION OF SEMANTICS PROPERTIES

After the presentation of FJ language semantics and how random tests are generated, we demonstrate how QuickCheck [6] 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 [10]. 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 result 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 follows:

```
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 \forall , 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 indeed 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 [4]), 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 [4]), 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 [11]. 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.

```
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
          _ -> Nothing
      _ -> Nothing
    else -- RC-Field
      maybe (Nothing)
        (\e' -> Just (FieldAccess e' f))
        (eval' ct e)
```

Fig. 6. Unreachable code on evaluation.

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 [4], a well-typed expression does not get stuck.

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

```
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)
        _ -> throwError (ClassNotFound c)
    e -> e -- Error: Expression type not found
```

Fig. 7. Unreachable code on type-checker.

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

VIII. 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 [12]. 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 [5]. 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 [13] is a generator of programs for language C, 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. [14] developed a methodology that uses differential testing for C compilers. Lindig [15] 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 [16]. 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. [17] generate random Java programs to test refactoring engines in Eclipse and NetBeans. Klein et al. [18] generated random programs to test an object-oriented library. Allwood and Eisenbach 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 [5] 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 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.

IX. 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.

REFERENCES

- [1] tiobe.com, “TIOBE Index,” <https://www.tiobe.com/tiobe-index/>, 04 2018, accessed: 2019-04-09.
- [2] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, “The java language specification, java se 8 edition (java series),” 2014.
- [3] M. Debbabi and M. Fourati, “A formal type system for Java,” *Journal of Object Technology*, vol. 6, no. 8, pp. 117–184, 2007.
- [4] A. Igarashi, B. C. Pierce, and P. Wadler, “Featherweight java: A minimal core calculus for java and gj,” *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, pp. 396–450, May 2001. [Online]. Available: <http://doi.acm.org/10.1145/503502.503505>
- [5] M. H. Palka, K. Claessen, A. Russo, and J. Hughes, “Testing an optimising compiler by generating random lambda terms,” in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST ’11. New York, NY, USA: ACM, 2011, pp. 91–97. [Online]. Available: <http://doi.acm.org/10.1145/1982595.1982615>
- [6] K. Claessen and J. Hughes, “Quickcheck: A lightweight tool for random testing of haskell programs,” in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’00. New York, NY, USA: ACM, 2000, pp. 268–279. [Online]. Available: <http://doi.acm.org/10.1145/351240.351266>
- [7] C. McBride, “Djinn, monotonic,” in *PAR@ ITP*, 2010, pp. 14–17.
- [8] B. C. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002.
- [9] T. O. R. Allwood and S. Eisenbach, “Tickling java with a feather,” *Electron. Notes Theor. Comput. Sci.*, vol. 238, no. 5, pp. 3–16, Oct. 2009. [Online]. Available: <http://dx.doi.org/10.1016/j.entcs.2009.09.037>
- [10] J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson, “Effect-driven quickchecking of compilers,” *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 15:1–15:23, Aug. 2017. [Online]. Available: <http://doi.acm.org/10.1145/3110259>
- [11] A. Gill and C. Runciman, “Haskell program coverage,” in *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, ser. Haskell ’07. New York, NY, USA: ACM, 2007, pp. 1–12. [Online]. Available: <http://doi.acm.org/10.1145/1291201.1291203>
- [12] R. Blanco, D. Miller, and A. Momigliano, “Property-based testing via proof reconstruction work-in-progress,” in *LFMTP 17: Logical Frameworks and Meta-Languages: Theory and Practice*, 2017.
- [13] X. Yang, Y. Chen, E. Eide, and J. Regehr, “Finding and understanding bugs in c compilers,” *SIGPLAN Not.*, vol. 46, no. 6, pp. 283–294, Jun. 2011. [Online]. Available: <http://doi.acm.org/10.1145/1993316.1993532>
- [14] V. Le, M. Afshari, and Z. Su, “Compiler validation via equivalence modulo inputs,” *SIGPLAN Not.*, vol. 49, no. 6, pp. 216–226, Jun. 2014. [Online]. Available: <http://doi.acm.org/10.1145/2666356.2594334>
- [15] C. Lindig, “Random testing of c calling conventions,” in *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, ser. AADEBUG’05. New York, NY, USA: ACM, 2005, pp. 3–12. [Online]. Available: <http://doi.acm.org/10.1145/1085130.1085132>
- [16] D. Drienovszky, D. Horpácsi, and S. Thompson, “Quickchecking refactoring tools,” in *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, ser. Erlang ’10. New York, NY, USA: ACM, 2010, pp. 75–80. [Online]. Available: <http://doi.acm.org/10.1145/1863509.1863521>
- [17] B. Daniel, D. Dig, K. Garcia, and D. 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*, ser. ESEC-FSE ’07. New York, NY, USA: ACM, 2007, pp. 185–194. [Online]. Available: <http://doi.acm.org/10.1145/1287624.1287651>
- [18] C. Klein, M. Flatt, and R. B. Findler, “Random testing for higher-order, stateful programs,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA ’10. New York, NY, USA: ACM, 2010, pp. 555–566. [Online]. Available: <http://doi.acm.org/10.1145/1869459.1869505>