A Type-Directed Algorithm to Generate Well-Typed Featherweight Java Programs

Samuel S. Feitosa[1], Rodrigo G. Ribeiro[2], and Andre R. Du Bois[1]

[1] PPGC - Universidade Federal de Pelotas, Pelotas - RS, Brazil
{samuel.feitosa,dubois} [at] inf.ufpel.edu.br
[2] PPGCC - Universidade Federal de Ouro Preto, Ouro Preto - MG, Brazil
rodrigo [at] decsi.ufop.br

Abstract. Property-based testing of compilers or programming languages semantics is difficult to accomplish because it is hard to design a random generator for valid programs. Most compiler test tools do not have a well-specified way of generating type-correct programs, which is a requirement for such testing activities. In this work, we formalize a type-directed procedure to generate random well-typed programs in the context of Featherweight Java, a well-known object-oriented calculus for the Java programming language. We implement the approach using the Haskell programming language and verify it against relevant properties using QuickCheck, a library for property-based testing.

Keywords: Featherweight Java · QuickCheck · Property-Based Testing.

1 Introduction

Currently, Java is one of the most popular programming languages [16]. 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 [8].

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 [5]. 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) [9] 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. [14] in terms of QuickCheck [3]. 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 [13] 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 interpreter[3] 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. We also used QuickCheck as a proof of concept to check type-soundness proofs using the interpreter and the generated programs[4].

The remainder of this text is organized as follows: Section 2 summarizes the FJ proposal. Section 3 presents the process of generating well-typed random programs in the context of FJ. Section 4 proves that our generation procedure is sound with respect to FJ typing rules. Section 5 shows how the results of testing type-safety properties of FJ with QuickCheck. Section 6 discusses some related works. Finally, we present the final remarks in Section 7.

[3] The source-code for our Haskell interpreter and the complete test suite is available at: `https://github.com/fjpub/fj-qc/`.

[4] Details 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`.

2 Featherweight Java

Featherweight Java [9] is a minimal core calculus for Java, in the sense that as many features of Java as possible are omitted, while maintaining the essential flavor of the language and its type system. However, this fragment is large enough to include many useful programs. A program in FJ consists of the declaration of a set of classes and an expression to be evaluated, that corresponds to the Java's main method.

FJ is to Java what λ-calculus is to Haskell. It offers similar operations, providing classes, methods, attributes, inheritance and dynamic casts with semantics close to Java's. The Featherweight Java project favors simplicity over expressivity and offers only five ways to create terms: object creation, method invocation, attribute access, casting and variables [9].

FJ semantics provides a purely functional view without side effects. In other words, attributes in memory are not affected by object operations [15]. 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 [15].

The abstract syntax of FJ is given in Figure 1.

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
$e.\text{m}(\overline{e})$	method invocation
new $C(\overline{e})$	object creation
$(C)\ e$	cast

Fig. 1. Syntactic definitions for FJ.

In the syntactic definitions `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 $\varphi : L \to C$ 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 $C_1, ..., C_n$ (similarly for $\overline{f}$, $\overline{x}$, etc.). An empty sequence is denoted by $\bullet$, and the length of a sequence $\bar{\mathtt{x}}$ is written $\#\bar{\mathtt{x}}$. The inclusion of an item x in a sequence $\overline{X}$ is denoted by $x : \overline{X}$, following Haskell's 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.

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. The FJ authors presented rules for subtyping and auxiliary definitions (functions *fields*, *mtype*, and *mbody*), which are omitted from this text for space reasons.

Figure 2 shows the typing rules for FJ expressions.

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

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

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

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

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

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

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

Fig. 2. Expression typing.

The typing judgment for expressions has the form $\Gamma \vdash$ `e: C`, meaning that in the environment Γ, expression `e` has type `C`. 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 $\mathtt{e}_0$, which results in the type $\mathtt{C}_0$. Then it obtains the *fields* of class $\mathtt{C}_0$, matching the position of $\mathtt{f}_i$ in the resultant list, to return the respective type $\mathtt{C}_i$. 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[5].

Figure 3 shows the rules to check if methods and classes are well-formed.

Method typing

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

Class typing

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

Fig. 3. Method and class typing.

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 formal 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.

The authors also presented the semantic rules for FJ, which are omitted here, but can be found in the original paper [9]. 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 and type-checking rules, the authors presented proofs of type soundness for FJ as another important contribution, which will be explored by our test suite in the next sections.

[5] A detailed explanation about *stupid casts* can be found in p. 260 of [15].

3 Program Generation

The creation of tests for a programming language semantics or compiler 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 [14].

For generating random programs in the context of FJ, we follow two distinct phases, expression and class generation, generalizing the approach of [14] considering that FJ has a nominal type system instead of a structural one. In this way, we have specified a generation rule inspired by each typing rule, both for expression generation and class table generation.

3.1 Expression Generation

We assume that a class table *CT* is a finite mapping between names and its corresponding classes. We let *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:

$$\mathrm{CT} \;;\; \Gamma \;;\; \mathrm{C} \rightarrow \mathrm{e} \tag{1}$$

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`.

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

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 $\overline{\mathtt{C}_c}$:

$$\overline{C_c} = \{\ \mathrm{C}_1 \mid \mathrm{C}_1 \in \mathit{dom}(\mathrm{CT}) \wedge \exists\ \mathrm{x}.\ \mathrm{C}\ \mathrm{x} \in \mathit{fields}(\mathrm{C}_1)\ \}$$

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

$$\begin{gathered} \mathrm{C}' = \xi(\overline{C_c}) \\ \mathrm{CT}\ ;\ \mathit{\Gamma}\ ;\ \mathrm{C}' \rightarrow \mathrm{e} \end{gathered}$$

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).

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

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

$$\frac{\begin{gathered} \overline{C_c} = \{\mathrm{C}_1 \mid \mathrm{C}_1 \in \mathit{dom}(\mathrm{CT}) \wedge \exists\ \mathrm{x.\ C\ x} \in \mathit{fields}(\mathrm{C}_1)\} \\ \mathrm{C}' = \xi(\overline{C_c}) \\ \mathrm{CT}\ ;\ \mathit{\Gamma}\ ;\ \mathrm{C}' \rightarrow \mathrm{e} \\ \mathrm{C\ f} = \xi(\{\mathrm{C\ x} \mid \mathrm{C\ x} \in \mathit{fields}(\mathrm{C}')\} \end{gathered}}{\mathrm{CT}\ ;\ \mathit{\Gamma}\ ;\ \mathrm{C} \rightarrow \mathrm{e.f}}\ \text{[G-Field]}$$

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

$$\overline{C_c} = \{\mathrm{C}_1 \mid \mathrm{C}_1 \in \mathit{dom}(\mathrm{CT}) \wedge \exists\ \mathrm{m}\ \bar{\mathrm{D}}.\ \mathit{mtype}(\mathrm{m},\ \mathrm{C}_1) = \bar{\mathrm{D}} \rightarrow \mathrm{C}\}$$

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

$$\begin{gathered} \mathrm{C}' = \xi(\overline{C_c}) \\ \mathrm{CT}\ ;\ \mathit{\Gamma}\ ;\ \mathrm{C}' \rightarrow \mathrm{e}_0 \end{gathered}$$

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.

$$\begin{gathered} \overline{M_c} = \{(\mathrm{m},\ \bar{\mathrm{D}} \rightarrow \mathrm{C}) \mid \exists\ \mathrm{m}.\ \mathit{mtype}(\mathrm{m}\ ,\ \mathrm{C}') = \bar{\mathrm{D}} \rightarrow \mathrm{C}\} \\ (\mathrm{m}',\ \bar{\mathrm{D}}' \rightarrow \mathrm{C}) = \xi(\overline{M_c}) \end{gathered}$$

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.

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

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

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

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

$$\frac{\begin{array}{c}\overline{C_c} = \{\mathrm{C}_1 \mid \mathrm{C}_1 \in \mathit{dom}(\mathrm{CT}) \land \exists\ \mathrm{m}\ \bar{\mathrm{D}}.\ \mathit{mtype}(\mathrm{m}, \mathrm{C}_1) = \bar{\mathrm{D}} \to \mathrm{C}\} \\ \mathrm{C}' = \xi(\overline{C_c}) \\ \mathrm{CT}\ ;\ \Gamma\ ;\ \mathrm{C}' \to \mathrm{e}_0 \\ \overline{M_c} = \{(\mathrm{m}, \bar{\mathrm{D}} \to \mathrm{C}) \mid \exists\ \mathrm{m}.\ \mathit{mtype}(\mathrm{m}\ , \mathrm{C}') = \bar{\mathrm{D}} \to \mathrm{C}\} \\ (\mathrm{m}', \bar{\mathrm{D}}' \to \mathrm{C}) = \xi(\overline{M_c}) \\ \bar{\mathrm{a}} = \{\mathrm{e} \mid \mathrm{D} \in \bar{\mathrm{D}}' \land \mathrm{CT}\ ;\ \Gamma\ ;\ \xi(\mathrm{subtypes}(\mathrm{CT}, \mathrm{D})) \to \mathrm{e}\}\end{array}}{\mathrm{CT}\ ;\ \Gamma\ ;\ \mathrm{C} \to \mathrm{e}_0.\mathrm{m}'(\bar{\mathrm{a}})}\ \text{[G-Invk]}$$

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`.

$$\frac{\begin{array}{c}\bar{\mathrm{F}} = \{\mathrm{C}' \mid \mathrm{C}'\ \mathrm{f} \in \mathit{fields}(\mathrm{C})\} \\ \bar{\mathrm{a}} = \{\mathrm{e} \mid \mathrm{F} \in \bar{\mathrm{F}} \land \mathrm{CT}\ ;\ \Gamma\ ;\ \xi(\mathit{subtypes}(\mathrm{CT}, \mathrm{F})) \to \mathrm{e}\}\end{array}}{\mathrm{CT}\ ;\ \Gamma\ ;\ \mathrm{C} \to \mathrm{new}\ \mathrm{C}(\bar{\mathrm{a}})}\ \text{[G-New]}$$

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

$$\frac{\begin{array}{c}\bar{\mathrm{D}} = \mathit{subtypes}(\mathrm{CT}, \mathrm{C}) \\ \mathrm{CT}\ ;\ \Gamma\ ;\ \xi(\bar{\mathrm{D}}) \to \mathrm{e}\end{array}}{\mathrm{CT}\ ;\ \Gamma\ ;\ \mathrm{C} \to (\mathrm{C})\ \mathrm{e}}\ \text{[G-UCast]}$$

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 [9], 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`.

$$\mathit{supertypes}(\mathrm{CT}, \mathrm{Object}) = \bullet$$
$$\mathit{supertypes}(\mathrm{CT}, \mathrm{C}) = \{\mathrm{D}\} \cup \mathit{supertypes}(\mathrm{CT}, \mathrm{D}),\ \text{if class C extends D} \in \mathrm{CT}$$

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

$$\frac{\begin{array}{c}\bar{\mathrm{D}} = \mathit{supertypes}(\mathrm{CT},\mathrm{C}) \\ \mathrm{CT}\ ;\ \Gamma\ ;\ \xi(\bar{\mathrm{D}}) \to \mathrm{e}\end{array}}{\mathrm{CT}\ ;\ \Gamma\ ;\ \mathrm{C} \to (\mathrm{C})\ \mathrm{e}}\ \text{[G-DCast]}$$

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`.

$$\frac{\begin{array}{c}\bar{\mathrm{C}} = \mathit{dom}(\mathrm{CT})\ \text{-}\ (\mathit{subtypes}(\mathrm{CT},\mathrm{C}) \cup \mathit{supertypes}(\mathrm{CT},\mathrm{C})) \\ \mathrm{CT}\ ;\ \Gamma\ ;\ \xi(\bar{\mathrm{C}}) \to \mathrm{e}\end{array}}{\mathrm{CT}\ ;\ \Gamma\ ;\ \mathrm{C} \to (\mathrm{C})\ \mathrm{e}}\ \text{[G-SCast]}$$

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

3.2 Class Table Generation

To generate a class table, we assume the existence of an enumerable set $\overline{C_n}$ of class names and $\overline{V_n}$ 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:

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

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:

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

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

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 $\varphi(L)\ L : CT$; n ; m ; p $\rightarrow$ CT'. The following judgment presents how classes are generated:

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

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:

$$\mathrm{C} = \xi(\overline{C_n}\ \text{-}\ (dom(\mathrm{CT}) \cup \{\mathrm{Object}\}))$$

This rule selects a random class name from the set $\overline{C_n}$ 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`:

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

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 `fn` and `mn`, 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 $\overline{U_n}$ to avoid duplicates.

$$\frac{}{\text{CT} \;;\; 0 \;;\; \overline{U_n} \rightarrow \bullet} \;\text{[G-Fields-Base]}$$

$$\frac{\begin{array}{c}\text{C} = \xi(\mathit{dom}(\text{CT}) \cup \{\text{Object}\}) \\ \text{f} = \xi(\overline{V_n} \text{ - } \overline{U_n}) \\ \text{CT} \;;\; \text{n} \;;\; \text{f} : \overline{U_n} \rightarrow \bar{\text{C}}\ \bar{\text{f}}\end{array}}{\text{CT} \;;\; \text{n} + 1 \;;\; \overline{U_n} \rightarrow \text{C f} : \bar{\text{C}}\ \bar{\text{f}}} \;\text{[G-Fields-Step]}$$

Generation of the method list proceeds by recursion on m, as shown below. We also maintain a set of already used method names $\overline{U_n}$ 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`.

$$\frac{}{\text{CT} \;;\; \text{C} \;;\; 0 \;;\; \text{p} \;;\; \overline{U_n} \rightarrow \bullet} \;\text{[G-Methods-Base]}$$

$$\frac{\begin{array}{c}\text{x} = \xi(\overline{V_n} \text{ - } \overline{U_n}) \\ \text{CT} \;;\; \text{C} \;;\; \text{p} \;;\; \text{x} \rightarrow \text{M} \\ \text{CT} \;;\; \text{C} \;;\; \text{m} \;;\; \text{p} \;;\; \text{x} : \overline{U_n} \rightarrow \bar{\text{M}}\end{array}}{\text{CT} \;;\; \text{C} \;;\; \text{m} + 1 \;;\; \text{p} \;;\; \overline{U_n} \rightarrow \text{M} : \bar{\text{M}}} \;\text{[G-Methods-Step]}$$

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 $\texttt{C}_0$.

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

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

$$\frac{}{\text{CT} \;;\; 0 \;;\; \overline{U_n} \rightarrow \bullet} \;\text{[G-Param-Base]}$$

$$\frac{\begin{array}{c}\text{C} = \xi(\mathit{dom}(\text{CT}) \cup \{\text{Object}\}) \\ \text{x} = \xi(\overline{V_n} \text{ - } \overline{U_n}) \\ \text{CT} \;;\; \text{n} \;;\; \text{x} : \overline{U_n} \rightarrow \bar{\text{C}}\ \bar{\text{x}}\end{array}}{\text{CT} \;;\; \text{n} + 1 \;;\; \overline{U_n} \rightarrow (\text{C x} : \bar{\text{C}}\ \bar{\text{x}})} \;\text{[G-Param-Step]}$$

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

$$CT \;;\; \mathrm{C} \;;\; \mathrm{D} \rightarrow \mathrm{K}$$

Rule `G-Constr` represents the process to generate the constructor.

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

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

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

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

4 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* $C \in \mathit{dom}(\mathrm{CT})$*, if* CT *;* Γ *;* $C \rightarrow e$ *then exists* D*, such that* $\Gamma \vdash e : D$ *and* $D <: C$*.*

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

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

Case (G-Invk): Then, $\mathrm{e} = \mathrm{e}_0.\mathrm{m}(\bar{\mathrm{e}})$ for some e_0 and $\bar{\mathrm{e}}$; $CT \;;\; \Gamma \;;\; \mathrm{C}' \rightarrow \mathrm{e}_0$, for some C'; there exists $(\mathrm{m}, \bar{\mathrm{D}}' \rightarrow \mathrm{C})$, such that $\mathit{mtype}(\mathrm{m}, \mathrm{C}') = \bar{\mathrm{D}} \rightarrow \mathrm{C}$ and for all $\mathrm{e}' \in \bar{\mathrm{e}}$, $\mathrm{D} \in \bar{\mathrm{D}}'$, $CT \;;\; \Gamma \;;\; \xi(\mathit{subtypes}(CT,\mathrm{D})) \rightarrow \mathrm{e}'$. By the induction hypothesis, we have that: $\Gamma \vdash \mathrm{e}_0 : \mathrm{D}'$, $\mathrm{D}' <: \mathrm{C}'$, for all $\mathrm{e}' \in \bar{\mathrm{e}}$, $\mathrm{D} \in \bar{\mathrm{D}}'$. $\Gamma \vdash \mathrm{e}' : \mathrm{B}$, $\mathrm{B} <: \mathrm{D}$ and the conclusion follows by the rule `T-Invk` and the definition of subtyping relation.

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

Proof. Straightforward induction on the structure of the result of *subtypes*(CT, C).

Lemma 3 (Soundness of method generation). *Let* CT *be a well-formed class table and* $C \in \mathit{dom}(\mathrm{CT}) \cup \{\texttt{Object}\}$*. For all p and m, if* CT *; C ; p ; m* $\rightarrow \mathrm{C}_0$ m ($\bar{\mathrm{C}}$ $\bar{\mathrm{x}}$) { return e; } *then* C_0 m ($\bar{\mathrm{C}}$ $\bar{\mathrm{x}}$) { return e; } *OK in C.*

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

- $\bar{\mathrm{C}} \subseteq \mathit{dom}(\mathrm{CT})$
- $\Gamma = \{\bar{\mathrm{C}}\ \bar{\mathrm{x}}, \texttt{this} : \mathrm{C}\}$
- $\mathrm{C}_0 = \xi(\mathit{dom}(\mathrm{CT}) \cup \{\texttt{Object}\})$
- $\bar{\mathrm{D}} = \mathit{subtypes}(\mathrm{CT}, \mathrm{C}_0)$
- $CT\ ;\ \Gamma\ ;\ \mathrm{E}_0 \rightarrow \mathrm{e}$
- $\mathrm{E}_0 = \xi(\bar{\mathrm{D}})$

By Lemma 2, we have that for all $\mathrm{D} \in \bar{\mathrm{D}}$, $\mathrm{C}_0 <: \mathrm{D}$.

By Lemma 1, we have that $\Gamma \vdash \mathrm{e} : \mathrm{E}'$ and $\mathrm{E}' <: \mathrm{E}_0$.

Since CT is well-formed, then $\mathit{mtype}(\mathrm{m}, \mathrm{C}) = \bar{\mathrm{C}} \rightarrow \mathrm{C}_0$ and the conclusion follows by rule *method typing* and the definition of the subtyping relation.

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 = class C extends D { $\bar{\mathrm{C}}$ $\bar{\mathrm{f}}$; K $\bar{\mathrm{M}}$ }
- $\mathrm{C} = \xi(\overline{C_n}$ - $(\mathit{dom}(\mathrm{CT}) \cup \texttt{Object}))$
- $\mathrm{D} = \xi(\mathit{dom}(\mathrm{CT}) \cup \texttt{Object})$
- fn = ξ([1..m])
- mn = ξ([1..(m - fn)])
- CT' = C (class C extends D {}) : CT
- CT' ; fn $\rightarrow \bar{\mathrm{C}}\ \bar{\mathrm{f}}$
- CT'' = C (class C extends D { $\bar{\mathrm{C}}$ $\bar{\mathrm{f}}$; }) : CT
- CT'' ; C ; mn ; p ; $\bullet \rightarrow \bar{\mathrm{M}}$
- CT' ; C ; D $\rightarrow$ K

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

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

The conclusion follows by rule *class typing*.

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* $\in$ *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 $\in$ *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.

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.

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.

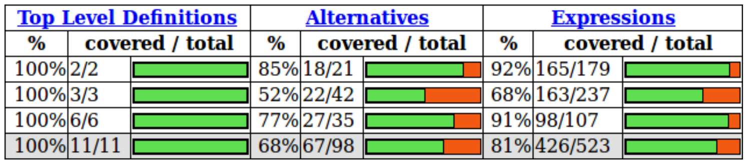
5 Quick-Checking Semantic Properties

As a proof of concept we have implemented an interpreter following the semantics of FJ and used random generated programs to test this interpreter against some properties[6], including those for type-soundness presented in the FJ original paper. The properties were specified and tested using QuickCheck [3]. Besides progress and preservation of the interpreter, we also used QuickCheck to verify

[6] More details about using QuickCheck for testing the semantic properties of FJ are in our technical report at: `https://github.com/fjpub/fj-qc/raw/master/tr.pdf`.

if all generated class tables are well-formed, and also if all generated expressions are well-typed and cast-safe. Furthermore, our tests cases were generated into Java files, and compiled using the Oracle's standard 'javac' compiler (the closest implementation of Java Language Specification) to validate our generator algorithm. After compiling and running many thousands of well-succeeded tests, we gain a high-degree of confidence in our type-directed procedure for generating programs.

As a way to measure the quality of the generated test cases, we used the Haskell Program Coverage tool [7] to check how much of the interpreter code base was covered by our test suite. Results of code coverage for each module (evaluator, type-checker, auxiliary functions, and total, respectively) are presented in Figure 4.



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. 4. Test coverage results.

Although not having 100% of code coverage, the proposed generation algorithm was capable to verify the main safety properties present in FJ paper. After analyzing test coverage results, we could observe that code not reached by test cases consisted of error control when evaluating the semantics or when dealing with expressions that are not well-typed.

6 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 [14]. 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 [17] 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. [11] developed a methodology that uses differential testing for C compilers. Lindig [12] 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 [6]. All of these projects rely on informal approaches, while ours is described formally and applied to property-based testing.

More specifically, Daniel et al. [4] generate random Java programs to test refactoring engines in Eclipse and NetBeans. Klein et al. [10] 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 generated randomly complete classes and expressions, and proved that both are well-formed and well-typed. Another difference is that we also used property-based testing to check that the properties of the FJ semantics hold by using the generated programs.

The work of Palka, Claessen and Hughes [14] also used the QuickCheck library in their work aiming to generate λ-terms to test the GHC compiler. Our approach was somewhat inspired by theirs, in the sense we also used QuickCheck and the typing rules for generating well-typed terms. Unlike their approach, we provided a standard small-step operational semantics to describe our generation algorithm.

7 Conclusion

In this work, we presented a syntax directed judgment for generating random type correct FJ programs, proving soundness with respect to FJ typing rules, and using 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 definitions many times, both as a result of correcting errors and streamlining the presentation. Ensuring that our changes were consistent was simply a matter of re-running the test suite. Encoding the type soundness properties as Haskell functions provides a clean and concise implementation that helps not only to fix bugs but also to improve understanding the meaning of the presented semantic properties.

As future work, we intend to use Coq to provide formally certified proofs for our generation procedure, as well as for the FJ semantics, showing that they do enjoy safety properties. We can also to explore the approach used in our test suite for other FJ extensions, besides using other tools like QuickChick with the same purpose.

References

1. Allwood, T.O.R., Eisenbach, S.: Tickling java with a feather. Electron. Notes Theor. Comput. Sci. **238**(5), 3–16 (Oct 2009), `http://dx.doi.org/10.1016/j.entcs.2009.09.037`

2. Blanco, R., Miller, D., Momigliano, A.: Property-based testing via proof reconstruction work-in-progress. In: LFMTP 17: Logical Frameworks and Meta-Languages: Theory and Practice (2017)
3. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 268–279. ICFP '00 (2000), `http://doi.acm.org/10.1145/351240.351266`
4. Daniel, B., Dig, D., Garcia, K., Marinov, D.: 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. pp. 185–194. ESEC-FSE '07 (2007), `http://doi.acm.org/10.1145/1287624.1287651`
5. Debbabi, M., Fourati, M.: A formal type system for Java. Journal of Object Technology **6**(8), 117–184 (2007)
6. Drienyovszky, D., Horpácsi, D., Thompson, S.: Quickchecking refactoring tools. In: Proceedings of the 9th ACM SIGPLAN Workshop on Erlang. pp. 75–80. Erlang '10 (2010), `http://doi.acm.org/10.1145/1863509.1863521`
7. Gill, A., Runciman, C.: Haskell program coverage. In: Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop. pp. 1–12. Haskell '07 (2007), `http://doi.acm.org/10.1145/1291201.1291203`
8. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The java language specification, java se 8 edition (java series) (2014)
9. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: A minimal core calculus for java and gj. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (May 2001), `http://doi.acm.org/10.1145/503502.503505`
10. Klein, C., Flatt, M., Findler, R.B.: Random testing for higher-order, stateful programs. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. pp. 555–566. OOPSLA '10 (2010), `http://doi.acm.org/10.1145/1869459.1869505`
11. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs. SIGPLAN Not. **49**(6), 216–226 (Jun 2014), `http://doi.acm.org/10.1145/2666356.2594334`
12. Lindig, C.: Random testing of c calling conventions. In: Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging. pp. 3–12. AADEBUG'05 (2005), `http://doi.acm.org/10.1145/1085130.1085132`
13. McBride, C.: Djinn, monotonic. In: PAR@ ITP. pp. 14–17 (2010)
14. Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: Proceedings of the 6th International Workshop on Automation of Software Test. pp. 91–97. AST '11 (2011), `http://doi.acm.org/10.1145/1982595.1982615`
15. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
16. tiobe.com: TIOBE Index. `https://www.tiobe.com/tiobe-index/` (04 2018), accessed: 2019-04-09
17. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. SIGPLAN Not. **46**(6), 283–294 (Jun 2011), `http://doi.acm.org/10.1145/1993316.1993532`