UNIVERSIDADE FEDERAL DE PELOTAS
Centro de Desenvolvimento Tecnológico
Programa de Pós-Graduação em Computação

Tese

Strategies for Testing and Formalizing Properties of Modern Programming Languages

Samuel da Silva Feitosa

Pelotas, 2019

Samuel da Silva Feitosa

Strategies for Testing and Formalizing Properties of Modern Programming Languages

Tese apresentada ao Programa de Pós-Graduação em Computação do Centro de Desenvolvimento Tecnológico da Universidade Federal de Pelotas, como requisito parcial à obtenção do título de Doutor em Ciência da Computação.

Advisor: Prof. Dr. Andre Rauber Du Bois
Coadvisor: Prof. Dr. Rodrigo Geraldo Ribeiro

Pelotas, 2019

Insira AQUI a ficha catalográfica
(solicite em http://sisbi.ufpel.edu.br/?p=reqFicha)

Samuel da Silva Feitosa

Strategies for Testing and Formalizing Properties of Modern Programming Languages

Tese aprovada, como requisito parcial, para obtenção do grau de Doutor em Ciência da Computação, Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas.

Data da Defesa: 20 de dezembro de 2019

Banca Examinadora:

Prof. Dr. Andre Rauber Du Bois (orientador)
Doutorado em Ciência da Computação pela Heriot-Watt University, Escócia.

Prof. Dr. Alvaro Freitas Moreira.
Doutorado em Ciência da Computação pela University of Edinburgh, Escócia.

Prof. Dr. Juliana Kaizer Vizzotto
Doutorado em Computação pela Universidade Federal do Rio Grande do Sul.

Prof. Dr. Luciana Foss
Doutorado em Computação pela Universidade Federal do Rio Grande do Sul.

To my wife Cheila, for her kindness and devotion, and her endless support.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my parents for their love and support throughout my life. Thank you both for encourage me to pursue my dreams.

I am profoundly grateful to my thesis advisers, Prof. André Rauber Du Bois, and Prof. Rodrigo Geraldo Ribeiro, for their guidance and support throughout this study and especially for their confidence in me. I also would like to thank Prof. Wouter Swierstra and Prof. Alejandro Serrano Mena for their kindness support during my studies abroad.

Last but not least, I thank for having my wife Cheila during all this time, sharing the good and bad moments, always understanding and encouraging me to keep working hard toward my goals. Your partnership is what makes me grow.

This material was based upon work supported by CAPES/Brazil.

This thesis is only the beginning of my journey.

If debugging is the process of removing software bugs, programming must be the process of putting them in.
— EDSGER DIJKSTRA

ABSTRACT

FEITOSA, Samuel da Silva. **Strategies for Testing and Formalizing Properties of Modern Programming Languages**. Advisor: Andre Rauber Du Bois. 2019. 132 f. Thesis (Doctorate in Computer Science) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2019.

Today's world is full of devices and machines controlled by software, which depend upon programming languages and compilers to be produced and executed. The importance of correct software development goes beyond personal computers and smartphone apps. An error in a critical system, such as on a nuclear power plant or on an airplane controller, can cause catastrophic damage in our society. Nowadays, essentially two software validation techniques are used to avoid such problems: software testing and software verification.

In this thesis we combine both validation techniques in the programming languages research area, applying property-based testing first to improve specifications and debugging programs, before an attempt of formal verification. By using such testing approach we can quickly eliminate false conjectures, by using the generated counterexamples, which help to correct them. Then, having confidence that the specification is correct, one can give a step forward and formalize the specification and prove its properties in an interactive theorem prover, which uses a mathematical framework to guarantee that these properties hold for a given specification.

We apply different strategies to test and formalize two major programming languages, the functional Lambda Calculus, and the modern object-oriented calculus Featherweight Java. The first branch of this thesis defines a type-directed procedure to generate random programs for each calculus in order to apply property-based testing to check soundness properties on them, using the Haskell library QuickCheck. And in the second branch, we apply the two most used approaches, extrinsic and intrinsic, to formalize and prove type safety for both studied programming languages using the dependently-typed programming language Agda, comparing the subtleties of each technique. Furthermore, we shown that our formalizations can be extended to new language constructions, by specifying and proving the same properties for the Java 8 constructions. We believe that this combination of property-based testing with formal verification can improve the quality of software in general and increase the productivity during formal proof development.

Keywords: Palavrachave-um. Palavrachave-dois. Palavrachave-tres. Palavrachave-quatro.

RESUMO

FEITOSA, Samuel da Silva. **Estratégias para Teste e Formalização de Propriedades de Linguagens de Programação**. Orientador: Andre Rauber Du Bois. 2019. 132 f. Tese (Doutorado em Ciência da Computação) – Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, Pelotas, 2019.

O mundo atual está repleto de dispositivos e máquinas controladas por software, os quais dependem de linguagens de programação e compiladores para serem produzidos e executados. A importância do desenvolvimento de software correto vai além de computadores pessoais e aplicativos de smartphones. Um erro em um sistema crítico, como em uma usina nuclear ou em um controlador de aviação, pode causar danos catastróficos em nossa sociedade. Hoje em dia, essencialmente duas técnicas de validação de software são utilizadas para evitar tais problemas: teste e verificação de software.

Nesta tese, são combinadas ambas as técnicas de validação na área pesquisa de linguagens de programação, aplicando testes baseados em propriedades inicialmente para melhorar especificações e depurar programas, antes de uma tentativa de verificação formal. Por usar esta abordagem de testes, é possível eliminar falsas conjecturas rapidamente e usar os contra-exemplos gerados para corrigí-las. Então, tendo confiança de que a especificação está correta, é possível ir além, formalizando a especificação e provando propriedades em um provador de teoremas interativo, o qual usa um aparato matemático para garantir que estas propriedades são válidas para uma dada especificação.

Aplicou-se diferentes estratégias para testar e formalizar duas linguages de programação, o Cálculo Lambda, e o cálculo orientado a objetos Featherweight Java. A primeira parte desta tese define um procedimento direcionado por tipos para gerar programas aleatórios para cada linguagem de modo a aplicar testes baseados em propriedades para verificar propriedades de segurança, usando Haskell e a biblioteca QuickCheck. E, na segunda parte, foram aplicadas duas abordagens, extrínseca e intrínseca, para formalizar e provar segurança de tipos para ambas as linguagens de programação estudadas, usando a linguagem de tipos dependentes Agda, comparando as sutilezas de cada técnica. Além disso, foi demonstrado que as formalizações apresentadas podem ser estendidas para novas construções de linguagens, a partir da especificação e provas das mesmas propriedades para as construções da linguagem Java 8. Acredita-se que esta combinação de testes baseados em propriedades com verificação formal pode melhorar a qualidade de software em geral e aumentar a produtividade durante o desenvolvimento de provas formais.

Palavras-chave: Keyword-one. Keyword-two. Keyword-three. Keyword-four.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AST	Abstract Syntax Tree
BNF	Backus-Naur Form
FJ	Featherweight Java
JDK	Java Development Kit
JVM	Java Virtual Machine
LC	Lambda Calculus
LOC	Lines of Code
STLC	Simply Typed Lambda Calculus

CONTENTS

1 INTRODUCTION

Programming languages are applied in every computational problem, from simple applications, such as smartphone apps, to complex critical systems, like those for flight control. Therefore, it is important to have mechanisms to verify that a programming language indeed implements certain behavior and enjoys some desired properties. Typically, the development of compilers (or interpreters) of major programming languages in the industry relies on dynamic analysis (testing) for detecting and eliminating bugs. On the other hand, most of the research about programming languages is done using static analysis (proving properties) for producing correct implementations. Both techniques have pros and cons. Testing can be done easily, using several well-studied approaches, producing counter-examples of the specification which can be used to fix the implementation faster. However, only by testing, it is impossible to prove the absence of errors. Theorem proving can provide such guarantees, however, in spite of continuous progress in the area, the process of mechanizing properties is still very time consuming and requires good skills from the user. Our methodology applies one branch of the testing area, called property-based testing, to improve specifications and to debug interpreters, before proving properties on a proof assistant.

Property-based testing is an automated approach to testing in which a program is validated against a specification, using randomly generated input data, in order to find counterexamples for the property being checked. Applying property-based testing for compilers or programming language semantics is not an easy task, because it is difficult to design a random generator for valid programs. Generating good test cases can be tricky, since these programs should have a structure that is accepted by the compiler, respecting several 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 (CELENTANO et al., 1980; BAZZICHI; SPADAFORA, 1982). Most compiler test tools do not have a well-specified way for generating type-correct programs, which is a requirement for such testing activities. However, despite the initial complication, property-based testing can be a handy tool to achieve confidence that a compiler or interpreter is correct, or that a specification is accurate. If one

needs to change the definition or the implementation, ensuring that they are consistent is just a matter of re-running the test suite.

Regarding formal verification of programming languages, the most used method for checking correctness or soundness is the syntactic approach (sometimes called extrinsic) proposed by Wright and Felleisen (WRIGHT; FELLEISEN, 1994). Using this technique, the syntax is defined first, and then relations are defined to represent both the typing judgments (static semantics), and the evaluation through reduction steps (dynamic semantics). The common theorems of *progress* and *preservation* link the static and the dynamic semantics, guaranteeing that a well-typed term does not get stuck, i.e., it should be a value or be able to take another reduction step, preserving the intended type. However, another technique, proposed by Altenkirch and Reus (ALTENKIRCH; REUS, 1999), is becoming increasingly popular in recent years, which uses a functional approach (sometimes called intrinsic) to achieve a similar result. The idea is to first encode the syntax and the typing judgments in a single definition which captures only well-typed expressions, usually within a total dependently-typed programming language. After that, one writes a definitional interpreter (REYNOLDS, 1972) which evaluates the well-typed expressions. By using this approach, type-soundness is guaranteed by construction, and the necessary lemmas and proofs of the syntactic approach can be obtained (almost) for free.

Ultimately, this thesis builds on research on techniques to verify properties of programming languages. Our project explores two different branches: (1) a lightweight approach, where we generate well-typed programs in order to test type safety properties; (2) a completely formal approach, where we prove type safety for languages using different formalization styles (extrinsic and intrinsic). We study these techniques by working on two major programming languages, which implement different paradigms. The first, λ-calculus (CHURCH, 1932), is a well-studied language within the functional programming community, used as basis to introduce the concepts regarding to both property-based testing, and mechanized formalization. The second, Featherweight Java (IGARASHI; PIERCE; WADLER, 2001), is a core calculus of a modern object-oriented language with a rigorous semantic definition of the main core aspects of Java. We argue that we can explore and apply the techniques simultaneously, since the lightweight approach provided by property-based testing allows to experiment with different design and implementations, before working on the proof assistant, thus avoiding trying to prove something impossible. Furthermore, by providing the reader with several case studies, we fulfill also an educational purpose, allowing the understanding of the area by running, reusing, or extending our examples.

1.1 Contributions

More precisely, this thesis makes the following contributions:

- We provide a catalog containing the description of the four most popular Java-like formalisms, presenting and comparing their main characteristics, aiming to aid in the process of choosing between them.
- We provide type-directed algorithms to construct random programs for λ-calculus, FJ, and an extension of FJ with the new Java 8 features (functional interfaces, λ-expressions, and default methods), proving that our specifications are sound with respect of the languages type-system, i.e., only well-typed programs are generated.
- We use QuickCheck as a lightweight manner to check for type soundness against hand-written interpreters using the programs produced by our type-directed procedure. We also use 'javac' as an oracle to compile the generated Java programs.
- We formalize the static and dynamic semantics of λ-calculus in both extrinsically and intrinsically-typed styles. We prove type soundness for the first using the common theorems of *progress* and *preservation*, and we implement a definitional interpreter for the second, which together with the intrinsic representation embeds the soundness proofs.
- We formalize the semantics of Featherweight Java, also considering both approaches. Similarly, we prove soundness for the extrinsic version by using the theorems of *progress* and *preservation*, and by using a definitional interpreter for the intrinsic version.
- We show that the studied approaches are useful to reason about different programming language concepts, by extending the formalization of Featherweight Java with Java 8 constructions.
- We discuss the formalization approaches and their subtleties, using some metrics (such as lines of code, and number of lemmas and theorems) to provide a comparison between the formalization styles.

1.1.1 Publications

Parts of this thesis are based on published papers, with the author of the thesis as the lead author for each paper.

- Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and Andre Rauber Du Bois. "Formal Semantics for Java-like Languages and Research Opportunities", in *Revista de Informática Teórica e Aplicada*, 2018. (FEITOSA; RIBEIRO; DU BOIS, 2018a)

- Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and Andre Rauber Du Bois. “Generating Random Well-Typed Featherweight Java Programs Using QuickCheck”, in *Proceedings of the XLIV Latin American Computing Conference*, 2018. (FEITOSA; RIBEIRO; DU BOIS, 2019)

- Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and Andre Rauber Du Bois. “Property-based Testing for Lambda Expressions Semantics in Featherweight Java”, in *Proceedings of the XXII Brazilian Symposium on Programming Languages*, 2018. (FEITOSA; RIBEIRO; DU BOIS, 2018b)

- Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and Andre Rauber Du Bois. “A Type-Directed Algorithm to Generate Well-Typed Featherweight Java Programs”, in *Proceedings of the XXI Brazilian Symposium on Formal Methods*, 2018. (FEITOSA; RIBEIRO; DU BOIS, 2018c)

- Samuel da Silva Feitosa, Alejandro Serrano Mena, Rodrigo Geraldo Ribeiro, and Andre Rauber Du Bois. “An Inherently-Typed Formalization for Featherweight Java”, in *Proceedings of the XXIII Brazilian Symposium on Programming Languages*, 2019. (FEITOSA et al., 2019)

1.2 Structure

This thesis is structured as follows:

Chapter 2 provides the necessary background to the theory used in this thesis. We introduce the concepts of operational semantics, property-based testing, and formal verification.

Chapter 3 presents the target languages studied during the development of this project. We briefly introduce subsets of functional and object-oriented languages, accompanied with their operational semantics. We also discuss the criteria for choosing the languages used in the next chapters.

Chapter 4 shows the generation of well-typed random programs to test properties of well-known programming languages, and applies it with *QuickCheck*.

Chapter 5 discusses the implementation of the same programming languages in Agda, applying two formalization techniques to prove type soundness, comparing the approaches with each other.

Chapter 6 presents future work, and concludes the present thesis.

All source code presented in this thesis has been written in Haskell version 8.6 or formalized in Agda version 2.6.0. We present here parts of the code used in our definitions, not necessarily in a strict lexically-scoped order. Some functions or formal proofs were omitted from the text to not distract the reader from understanding the high-level structure of the source code. In such situations we give a brief explanation and point out where all details can be found. All source code produced, including the LaTeX source of this thesis, is available online (FEITOSA, 2019).

This thesis is aimed at a reader familiar with the basics of functional programming in a language such as Haskell. Introductions to the Haskell programming language and functional programming can be found elsewhere (SERRANO MENA, 2014; LIPOVACA, 2011). However we do not assume any specialized knowledge in areas such as program semantics, property-based testing, or formal verification.

2 BACKGROUND

The purpose of this chapter is to cover some important background on a range of different topics which are central for this thesis, namely operational semantics (Section 2.1), property-based testing and random program generation (Section 2.2), and finally, formal verification and interactive theorem proving (Section 2.3). As stated in the introductory Chapter, the reader is assumed to be familiar with the basics of functional programming.

2.1 Operational Semantics

In this introductory section we explain the main concepts of formal semantics for a programming language using a very simple language for *boolean* and *arithmetic* expressions with two operations, *addition* and the logic *conjunction*. This example language is quite simple, sometimes known as Hutton's Razor, but useful to deal formally with some aspects of programming languages, allowing us explain the mathematical tools to express and reason clearly and precisely about the syntax and semantics of programs. With this, we present fundamental concepts, such as abstract syntax, evaluation, and type system.

Figure 1 presents the abstract syntax of the target language we shall work on the Backus-Naur Form (BNF), adapted from (PIERCE, 2002). It uses an auxiliary set of numerals n, which is a syntactic representation of the more abstract set of natural numbers $\mathbb{N}$.

$n ::=$	$0, 1, 2, ...$	numeric expressions
$e ::=$		expressions:
	$true$	constant true
	$false$	constant false
	n	numeric constant
	$e + e$	math operator
	$e \wedge e$	logic operator

Figure 1 – Abstract syntax for the expression language.

Besides numerals, the BNF schema in Figure 1 also uses two extra symbols, $+$ and $\wedge$. Their meanings are well-know in the programming language community: the first represents mathematical *addition*, and the second represents the logical *and* operator. The presented abstract syntax definition says that there are five ways to construct an arbitrary expression e.

- The first three lines of our definition of expressions e state that $true$, $false$, and n are possible constants in the presented language. The numeric constants allow the encoding of an infinite number of expressions, namely $0, 1, 2, ...$
- The fourth line shows that if we have already constructed two expressions e_1 and e_2, then $e_1 + e_2$ is also a possible expression in this language.
- The last line is similar, if e_1 and e_2 are expressions, then $e_1 \wedge e_2$ is also an expression.

We also define, in Figure 2 a subset of expressions, called *values*, which represent possible final results of evaluation. In the presented language, values are just the boolean constants $true$ and $false$, and the infinite list of natural numbers $(0, 1, 2, ...)$. Throughout our text, we use the meta-variable v to stand for values, bv for boolean values, and nv for numeric values.

$v ::=$		values:
	$true$	true value
	$false$	false value
	nv	numeric values
$nv ::=$	$0, 1, 2, ...$	numeric definitions

Figure 2 – Values for the expressions language.

Throughout this thesis, we will use BNF grammars to present syntactic definitions of programming languages. We could have used different schemes, such as inductive definitions, inference rules, or the concrete syntax with equivalent meaning (PIERCE, 2002). When using BNF grammars we are concerned in describing the expressions purely in terms of their structure, rather than a precise linear sequence of symbols which are valid expressions of the target language. We say that an expression is represented by its abstract syntax tree (AST).

The semantics of a programming language describes its behavior, giving to each program, defined as an abstract syntax tree, a unambiguous meaning. There are three basic approaches to formalizing semantics (NIELSON; NIELSON, 2007):

1. *Operational semantics:* The meaning of a construct is specified by the computation it induces when executed on an abstract machine.

2. *Denotational semantics:* Meanings are modeled by mathematical objects representing the effect of executing the constructs.

3. *Axiomatic semantics:* Expresses specific properties of the effect of executing the constructs. The meaning of a construct is just what can be proved about it.

In this thesis we explore the *operational semantics* approach, where the behavior of a program is described as a series of computational steps. Operational semantics can be classified into two styles, *small-step*, and *big-step*, both of which are used in different parts of this thesis, as we will see next.

2.1.1 Small-Step Semantics

The origins of the *small-step* semantics, also known as structural operational semantics, go back to a technical report by Plotkin (PLOTKIN, 1981). With small-step semantics, we represent computation by means of deductive systems that turn an abstract machine into a system of logical inference. The purpose is to describe how individual steps of computations take place, which are represented as a *transition* system.

To describe the behavior of a program, definitions are (usually) given by *inference rules* consisting of a conclusion that follows from a set of premises. The general form of an inference rule has the premises listed above a horizontal line, and the conclusion below, as follows:

$$\frac{premise_1 \quad premise_2 \ \ldots \ premise_3}{conclusion}$$

If the number of premises is zero, the horizontal line can be omitted, and we refer to the rule as an *axiom*. This inference rule is used to define an *evaluation relation* or *reduction* of an expression.

Figure 3 presents the evaluation relation of all possible expressions defined for the language of *boolean* and *arithmetic* expressions. Besides the meta-variables for values (v, bv, and nv), we let e denote an expression. Following common practice, all meta-variables can appear primed or sub-scripted. In all rules, we can see the transition relation $e \longrightarrow e'$, which expresses that "e evaluates to e' in one step of reduction". This *one-step* relation $\longrightarrow$ is the smallest binary relation on expressions. A *multi-step* semantics representing the reflexive and transitive closure of the one-step relation should be able to produce a value, or get *stuck*. We say that a computation is stuck when there is no rule to reduce an expression further, or to produce a value.

The presented small-step relation has three rules for each operator in this language. The first rule S-Add$_1$, says that if we have an expression $e_1 + e_2$, we should first evaluate the expression on the left (e_1) producing an expression e_1' (as described by the premise

$$\frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2}\ \text{[S-Add}_1\text{]} \qquad \frac{e_2 \longrightarrow e_2'}{nv + e_2 \longrightarrow nv + e_2'}\ \text{[S-Add}_2\text{]}$$

$$nv_1 + nv_2 \longrightarrow nv_1 \oplus nv_2\ \text{[S-Add]} \qquad \frac{e_1 \longrightarrow e_1'}{e_1 \wedge e_2 \longrightarrow e_1' \wedge e_2}\ \text{[S-And}_1\text{]}$$

$$true \wedge e_2 \longrightarrow e_2\ \text{[S-And}_2\text{]} \qquad false \wedge e_2 \longrightarrow false\ \text{[S-And}_3\text{]}$$

Figure 3 – Small-step semantics for the expression language.

$e_1 \longrightarrow e_1'$), resulting in a new expression $e_1' + e_2$. Similarly, the rule S-Add$_2$ says that if we have an expression $nv + e_2$, i.e., we have a value in the left-hand side expression (hopefully[1] a numeric value), we should evaluate the right-hand side expression (e_2) producing an expression e_2', resulting in a new expression $nv + e_2'$. The last rule regarding mathematical addition S-Add, performs the actual addition on numbers. It says that, if we have both sides as values ($nv_1 + nv_2$), we should produce a new value containing the result of nv_1 plus nv_2. We use the operator $\oplus$ to represent the actual sum of two numbers.

The rule S-And$_1$ has the same purpose of S-Add$_1$, evaluating the left-hand side expression. Rule S-And$_2$ says that if the left-hand side expression is literally the constant $true$, then the reduction of $true \wedge e_2$ should leave only the right-hand side expression e_2. Rule S-And$_3$ is described similarly, however, if the left-hand side expression is $false$, evaluating $false \wedge e_2$ will always produce $false$. The reader can note that, by using a small-step relation, one can define the order of evaluation of expressions, in which case, for this simple language, evaluation is done from left to right.

Apart from having a clear and concise way of expressing evaluation, the small-step semantics provides one of the main justifications of formal descriptions of languages: proving properties of programs and constructs of programming languages. Since the semantic descriptions are based on deductive logic, proofs of program properties are derived directly from the definitions of language constructs.

2.1.2 Big-Step Semantics

The *big-step* semantics (KAHN, 1987), also known as *natural semantics*, is an alternative style which directly formulates the notion of "expression e evaluates to the final value v", written as $e \Downarrow v$, i.e., it describes the complete reduction of an expression to its final result in one big-step.

Figure 4 presents the semantic rules considering the big-step style. Similarly to the small-step semantics, the computation of a given expression is also defined using *inference rules*.

[1] Only the dynamic semantics cannot guarantee that expressions will have the correct type. It is responsibility of the type system to prevent ill-typed expressions.

$$v \Downarrow v \quad \text{[B-Value]} \qquad \frac{e_1 \Downarrow nv_1 \qquad e_2 \Downarrow nv_2}{e_1 + e_2 \Downarrow nv_1 \oplus nv_2} \quad \text{[B-Add]}$$

$$\frac{e_1 \Downarrow true \qquad e_2 \Downarrow bv}{e_1 \wedge e_2 \Downarrow bv} \quad \text{[B-And}_1\text{]} \qquad \frac{e_1 \Downarrow false}{e_1 \wedge e_2 \Downarrow false} \quad \text{[B-And}_2\text{]}$$

Figure 4 – Big-step semantics for the expression language.

We have four rules in this style of formalization, against the six of the previous one. The first rule, B-Value, expresses that if the initial expression is already a value, then this value is the result of the evaluation. The rule B-Add deals with the arithmetic addition operator. If we have an expression $e_1 + e_2$, then we evaluate both e_1 and e_2, using our evaluation judgment $\Downarrow$. For example, $e_1 \Downarrow nv_1$ evaluates expression e_1 producing (hopefully) a numeric value nv_1. Having both nv_1 and nv_2, again we use the operator $\oplus$ to perform the actual sum of two numbers, which is the result for this rule. To deal with the logic *and* operator, we have two rules: (1) rule B-And$_1$ deals with the case when e_1 evaluates to $true$, evaluating e_2 to a boolean value bv, which is the result for this rule; (2) rule B-And$_2$ deals with the case when e_2 is false, which means that the *and* operator will produce always $false$, which is the presented result for this rule. The reader can note that for the rule B-And$_2$ there is no need for evaluating the second expression e_2.

2.1.3 Type Checking

In the last sections we presented two ways to describe precisely the semantics of a small expression language. We briefly discussed that evaluating an expression could produce a value or get stuck at some point, when there is no applicable reduction rule to reduce it further. For example, if we reach an expression such as $true + 2$, it is impossible to reduce it, since $true$ is not a numeric value. Usually, such expressions correspond to meaningless or erroneous programs (PIERCE, 2002). Using just what we saw so far, we cannot guarantee that only values with correct types are assigned for sub-expressions. In this section we introduce the concept of *static semantics*, allowing us to check, without evaluating an expression, if it is correctly defined. This static analysis of terms is called *type checking*, which is able to differentiate well-typed from ill-typed expressions.

Figure 5 introduces two types in our expression language, $Bool$ to represent boolean types, and Num to represent numeric types. We use the meta-variable T in our rules to range over types.

To define the static semantics of a language we use an inductive definition of generic hypothetical judgments of the form $\vdash e : T$, which says that "an expression e has type T", meaning that we can compute the type of a given expression *statically*, i.e., without evaluating it (PIERCE, 2002).

$T ::=$		types:
	$Bool$	type of booleans
	Num	type of numbers

Figure 5 – Types for the expression language.

Figure 6 presents the type system rules for the calculus of expressions. The rule T-Num assigns the type Num for numeric constants. Rules T-True and T-False assign the type $Bool$ to the boolean constants $true$ and $false$. These three rules are axioms in our type system. Rule T-Add assigns a type Num for mathematical addition, considering that both e_1 and e_2 should be of type Num, as stated by the premises of this rule. Similarly, rule T-And assigns a type $Bool$ for the logic operation, as long as both e_1 and e_2 have type $Bool$.

$$\vdash n : Num \text{ [T-Num]} \qquad \vdash true : Bool \text{ [T-True]} \qquad \vdash false : Bool \text{ [T-False]}$$

$$\frac{\vdash e_1 : Num \quad \vdash e_2 : Num}{\vdash e_1 + e_2 : Num} \text{ [T-Add]} \qquad \frac{\vdash e_1 : Bool \quad \vdash e_2 : Bool}{\vdash e_1 \wedge e_2 : Bool} \text{ [T-And]}$$

Figure 6 – Typing rules for the expression language.

Formally, the typing relation for the presented language is the smallest binary relation between expressions and types satisfying all instances of the rules presented in Figure 6. An expression e is *typable*, or *well-typed*, if there is some T such that $\vdash e : T$ (PIERCE, 2002).

2.1.4 Properties

Having defined the static and dynamic semantics, we can state the most basic properties for this expression language: *safety* or *soundness*. We mentioned before that an expression is stuck when it is not a final value, and there is no evaluation rule to perform another reduction step. We show *safety* in two steps, commonly known as the *progress* and *preservation* theorems. Together, these properties tell us that a well-typed expression can never reach a stuck state during evaluation (PIERCE, 2002).

When working with *structural type systems*[2], it is common to define a lemma about the *canonical forms* of well-typed closed values. This lemma relates the possible values with their types. A numeric expression should be related to the Num type, as well as $true$ and $false$ should be related to the $Bool$ type. The following lemma presents this idea (HARPER, 2016).

[2] In a *structural type system*, the equivalence of types is determined by the type's actual structure, rather than by other characteristics such as its name or place of declaration.

Lemma 1 (Canonical Forms)**.** *Let v be a well-typed value such that $\vdash v : T$. Then:*

1. *if T is of type Num, then v is a numeric value ($1, 2, ...$).*
2. *if T is of type $Bool$, then v is either $true$ or $false$.*

Proof. Immediate from the definition of values (Figure 2) and typing rules (Figure 6). □

Intuitively, the *progress* theorem means that if a program has not terminated then it can continue to be evaluated.

Theorem 1 (Progress)**.** *Let e be a well-typed expression such that $\vdash e : T$. Then either e is a value or there is some expression e' such that $e \longrightarrow e'$.*

Proof. By induction on the derivation of $\vdash e : T$. The T-Num, T-True, and T-False cases are immediate, since e in these cases is a value. For the other cases, we argue as follows.

- Case T-Add: $e = e_1 + e_2 \quad e_1 : Num \quad e_2 : Num$

  By the induction hypothesis, we have that either e_1 is a value or else there is some e_1' such that $e_1 \longrightarrow e_1'$. If $e_1 \longrightarrow e_1'$ conclusion follows by rule S-Add$_1$. If e_1 is a value, by the induction hypothesis, we have that either e_2 is a value or else there is some e_2' such that $e_2 \longrightarrow e_2'$. If $e_2 \longrightarrow e_2'$ conclusion follows by rule S-Add$_2$. If e_2 is also a value, then the canonical forms (Lemma 2) assures that $e_1 : Num$ and $e_2 : Num$, and conclusion follows by rule S-Add.

- Case T-And: $e = e_1 \wedge e_2 \quad e_1 : Bool \quad e_2 : Bool$

  By the induction hypothesis, we have that either e_1 is a value or else there is some e_1' such that $e_1 \longrightarrow e_1'$. If e_1 is a value, then the canonical forms (Lemma 2) assures that it must be either $true$ or $false$, in which case S-And$_2$ or S-And$_3$ applies to e. On the other hand, if $e_1 \longrightarrow e_1'$ conclusion follows by rule S-And$_1$.

□

The *preservation* property means that whenever a program can be assigned to a type, if it takes a computation step, then the resulting expression can also be assigned to the same type[3].

Theorem 2 (Preservation)**.** *Let e be a well-typed expression such that $\vdash e : T$. Then $e \longrightarrow e'$ implies $\vdash e' : T$.*

Proof. By induction on the derivation of $\vdash e : T$. The rules T-Num, T-True, and T-False represent impossible cases, since e is a value, and it cannot take a reduction step. For the other cases, we argue as follows.

[3]There is some systems where types can change during evaluation. For example, in systems with subtyping, types can become smaller during evaluation.

- Case T-Add: $e = e_1 + e_2 \quad e_1 : Num \quad e_2 : Num$

  If the last rule in the derivation is T-Add, then we know from the form of this rule that e must have the form $e_1 + e_2$, for some e_1 and e_2. We must also have sub-derivations with conclusions $e_1 : Num$ and $e_2 : Num$. Now, looking at the (small-step) evaluation rules for the addition operator, we find that there are three rules by which $e \longrightarrow e'$ can be derived. We consider each case separately.

  - Sub-case S-Add$_1$: $e_1 \longrightarrow e_1' \quad e' = e_1' + e_2$

    From the assumptions of the T-Add case, we have a sub-derivation of the original typing derivation whose conclusion is $e_1 : Num$. We can apply the induction hypothesis to this sub-derivation, obtaining $e_1' : Num$. Combining this with the facts (from the assumptions of the T-Add case) that $e_2 : Num$, we can apply rule T-Add to conclude that $e_1' + e_2 : Num$, that is $e' : Num$.

  - Sub-case S-Add$_2$: $e_2 \longrightarrow e_2' \quad e' = nv + e_2'$

    From the assumptions of the T-Add case, we have a sub-derivation of the original typing derivation whose conclusion is $e_2 : Num$. We can apply the induction hypothesis to this sub-derivation, obtaining $e_2' : Num$. Combining this with the facts (from the assumptions of the T-Add case) that $nv : Num$, we can apply rule T-Add to conclude that $nv + e_2' : Num$, that is $e' : Num$.

  - Sub-case S-Add: $e_1 = nv_1 \quad e_2 = nv_2 \quad e' = nv_1 \oplus nv_2$

    If $e \longrightarrow e'$ is derived using S-And, then from the form of this rule we see that e_1 must be a numeric value nv_1, and e_2 must also be a numeric value nv_2, and the resulting expression e' is represented by the arithmetic sum $nv_1 \oplus nv_2$. This means we are finished, since we know (by the assumptions of the T-Add case) that $nv_1 \oplus nv_2 : Num$, which is what we need.

- Case T-And: $e = e_1 \wedge e_2 \quad e_1 : Bool \quad e_2 : Bool$

  If the last rule in the derivation is T-And, then we know from the form of this rule that e must have the form $e_1 \wedge e_2$, for some e_1 and e_2. We must also have sub-derivations with conclusions $e_1 : Bool$ and $e_2 : Bool$. Now, looking at the (small-step) evaluation rules for the boolean *and* operator, we find that there are three rules by which $e \longrightarrow e'$ can be derived. We consider each case separately.

  - Sub-case S-And$_1$: $e_1 \longrightarrow e_1' \quad e' = e_1' \wedge e_2$

    From the assumptions of the T-And case, we have a sub-derivation of the original typing derivation whose conclusion is $e_1 : Bool$. We can apply the induction hypothesis to this sub-derivation, obtaining $e_1' : Bool$. Combining this with the facts (from the assumptions of the T-And case) that $e_2 : Bool$, we can apply rule T-And to conclude that $e_1' \wedge e_2 : Bool$, that is $e' : Bool$.

– Sub-case S-And$_2$: $e_1 = true \quad e' = e_2$

If $e \longrightarrow e'$ is derived using S-And$_2$, then from the form of this rule we see that e_1 must be $true$ and the resulting expression e' is the second sub-expression e_2. This means we are finished, since we know (by the assumptions of the T-And case) that $e_2 : Bool$, which is what we need.

– Sub-case S-And$_3$: $e_1 = false \quad e' = false$

If $e \longrightarrow e'$ is derived using S-And$_3$, then from the form of this rule we see that e_1 must be $false$ and the resulting expression e' is also $false$. This means we are finished, since we know (from the canonical forms lemma) that $false : Bool$, which is what we need.

□

These two generic results, *progress* and *preservation*, when applied to this language of expressions guarantee that every well-typed program will always evaluate completely to a value.

2.2 Property-Based Testing

Property-based random testing is a popular technique for quickly discovering software errors. It is similar to formal verification, in that the user specifies desired properties of a unit under test, checking that a function or program obeys a given property. Usually, there is no need to specify example inputs and outputs as with unit tests. Instead, one uses a generative-testing engine to create randomized inputs to find out the defined properties are correct, or to present counter-examples of the given specification (LAMPROPOULOS; PARASKEVOPOULOU; PIERCE, 2017). In this way, code can be performed with thousands of tests that would be infeasible to write by hand, often uncovering subtle corner cases that would not be found otherwise. However, it is important to remember that, by running a finite number of tests when the number of all inputs is infinite, can only disprove a property, or leaving its validity undecided.

Claessen and Hughes (CLAESSEN; HUGHES, 2000) introduced property-based testing with QuickCheck, a combinator library for the functional language Haskell. By now, frameworks which implement at least some parts of the QuickCheck functionality are available for most general-purpose programming languages. In this section we give an overview on the basics of QuickCheck through examples, discussing how its combinators can be used to generate user-defined random test-data, and presenting a tool to measure test coverage.

2.2.1 Overview of QuickCheck

QuickCheck is a Haskell library that provides comprehensive support for property-based testing. It contains a combinator library for building composable properties (or test oracles), as well as random generators for basic Haskell data types. In property-based testing, we need to define a *specification*, i.e., a set of properties the program should satisfy. For example, let's consider the commutativity law for integers.

$$\forall n\ m.\ n + m = m + n$$

We can write a testable property for it by writing the Haskell function prop_PlusComm that checks this equality for two numbers, as follows.

```
prop_PlusComm :: Int → Int → Bool
prop_PlusComm n m = n + m ≡ m + n
```

This function is an executable version of the logical property and may be used as an oracle in random testing. A single test of this property is performed by generating two numbers, running the function, and checking if the produced result is True.

To discuss this process further, we take the examples about the standard function reverse presented in the original QuickCheck paper (CLAESSEN; HUGHES, 2000). This function satisfies the following laws.

$$\begin{aligned} reverse[x] &= [x] \\ reverse(xs +\!\!+ ys) &= reverse\ ys +\!\!+ reverse\ xs \\ reverse(reverse\ xs) &= xs \end{aligned}$$

We can convert each law into a function which determines whether the property fails or succeeds on a given input. The specifications of the properties above can be written in Haskell as follows. Note that each function starts with *prop* as a convention in QuickCheck.

```
prop_RevUnit x      = reverse [x] ≡ [x]
prop_RevApp xs ys   = reverse (xs ++ ys) ≡ reverse ys ++ reverse xs
prop_RevRev xs      = reverse (reverse xs) ≡ xs
```

Having specified the property as a function, we can use QuickCheck to generate inputs and check the results.

```
Main> quickCheck prop_RevRev
OK: passed 100 tests.
```

In this sample execution, QuickCheck generated one hundred[4] random lists and found that each of them satisfied the equality of prop_RevRev function.

[4]One hundred is the standard number of test cases. The library allows the specification of different numbers according to the user needs.

2.2.1.1 Conditional Laws

Sometimes, when checking some property, we need to constrain the input under certain conditions. QuickCheck provides an implication combinator to represent such conditional laws (CLAESSEN; HUGHES, 2000). For example, consider the following law.

$$x \leq y \Longrightarrow max\ x\ y = y$$

It says that $max\ x\ y = y$ if and only if $x \leq y$. Function prop_MaxLe represents such definition.

```
prop_MaxLe :: Int → Int → Property
prop_MaxLe x y = x ⩽ y ⟹ max x y ≡ y
```

Similarly, to test a function that inserts an element into an ordered list, we need the input xs to be ordered.

```
prop_Insert :: Int → [Int] → Property
prop_Insert x xs = ordered xs ⟹ ordered (insert x xs)
```

The reader can note that, instead of returning a Bool value, the functions over conditional laws return a Property value. This type is used to indicate QuickCheck to generate one hundred test cases *satisfying* the condition, i.e., for each generated test case, the left hand-side condition is checked, and only those satisfying the condition are used to verify the given property. Sometimes, checking a conditional law can produce the following output.

```
Arguments exhausted after 64 tests.
```

The process of generating tests under conditional laws can be difficult for a random generator. For example, generating small ordered lists can be easy, but when the number of elements grows, it becomes a problem. To avoid non-termination QuickCheck tries (by default) generating one thousand test cases to find one hundred respecting the expected law. If it cannot generate these tests, a message like the above is presented. It says that QuickCheck was able to generate only 64 tests under the specified condition. We will see next, that for several purposes, the standard generators are not able to produce good test cases, and the programmer should guide QuickCheck during the generation process.

2.2.2 User-Defined Test Cases

In this section, we will look into the problem of generating test cases through an example. QuickCheck provides support for programmers to define their own test data generators, being able to control the whole generation process and the distribution of test data. Here, we consider the problem of generating random programs according to the expression language presented previously. For this kind of problem, in practive, this means that to obtain a system that is suitable for property-based testing, special considerations are needed in the design.

2.2.2.1 An Interpreter for the Expression Language

Before presenting the generation process, we implement the expression language we are studying in Haskell. First we define its syntax.

```
data Expr = BTrue | BFalse | Num Int | Add Expr Expr | And Expr Expr
```

And then, following the small-step semantic rules, we define the function `step`. In this simple example, we are not dealing with error cases, since we want to test the semantics only with well-typed expressions[5].

```
step :: Expr → Maybe Expr
step (Add (Num n1) (Num n2)) = Just (Num (n1 + n2))
step (Add (Num n1) e2)       = do e2' ← step e2
                                  return (Add (Num n1) e2')
step (Add e1 e2)             = do e1' ← step e1
                                  return (Add e1 e2)
step (And BTrue e2)          = Just e2
step (And BFalse e2)         = Just BFalse
step (And e1 e2)             = do e1' ← step e1
                                  return (And e1' e2)
```

This function is responsible to implement only one reduction step, where the first three defining equations (i.e., the first three pattern matching structures) deal with the reduction for the arithmetic operator, and the last three defining equations perform reduction for the boolean operator.

The language we are working on has two possible types (`TBool` and `TNum`), as defined below.

```
data Ty = TBool | TNum
```

Ultimately, we want to check well-typed programs, in order to test safety properties. Thus, we define the function `typeof` following the static semantic rules for the studied calculus.

[5] A complete interpreter would check and present error messages for the user.

```
typeof :: Expr → Maybe Ty
typeof BTrue = Just TBool
typeof BFalse = Just TBool
typeof (Num _) = Just TNum
typeof (Add e1 e2) = do TNum ← typeof e1
                        TNum ← typeof e2
                        return TNum
typeof (And e1 e2) = do TBool ← typeof e1
                        TBool ← typeof e2
                        return TBool
```

The first three cases are straightforward. We return the types associated with each value. The last two cases are similar to each other. First they obtain recursively the types for `e1` and `e2`, forcing them to have the correct type `TNum` or `TBool` according to the case. If both `e1` and `e2` are with correct types, then we return type `TNum` for the arithmetic operation, and `TBool` for the boolean operator. It is important to note that the return type of `typeof` is a `Maybe` value. Case `e1` or `e2` have types different of the expected, the `typeof` function returns `Nothing`[6].

2.2.2.2 Generating Test Cases

In order to successfully test a compiler or interpreter, programs not only need to be grammatically correct, they may also need to satisfy other properties such as all variables being bound, all expressions well-typed, certain combinations of constructs not occurring in the programs, or a combination of such properties. We saw already some cases when an ill-typed expression can be stuck during interpretation, and these cases should be avoided by a program generator. We could try to use conditional laws to generate only well-typed expressions, using an approach as follows.

$$well\text{-}typed\ e \Longrightarrow some\text{-}property\ e$$

However, it is not hard to see that a random approach would hardly generate any valid program at all. Considering this problem, PAŁKA et al. (2011) proposed a type-directed procedure to generate only well-typed programs in the context of the λ-calculus. We will give more details about their work in Chapter 4. By now, we adapted their technique to generate well-typed programs for our expression language.

QuickCheck introduced the type class `Arbitrary`, of which a type is an instance if we can generate arbitrary elements of it. We define an instance of this type class for our `Expr`, to be able to randomly generate expressions. The next code is very simple. First we generate a type using the function `genType`, which will bring us (randomly) one of

[6] This is the default behavior when using the “do notation” when a fail happens.

our two possible types (TBool and TNum). Then, we use the function sized together with the function genExpr to generate expressions. We use the QuickCheck function sized to limit the depth of the recursive generation procedure, in order to avoid non-termination.

```
instance Arbitrary Expr where
  arbitrary = do
    t ← genType
    sized (λ n → genExpr n t)
```

The function genExpr is responsible for generating well-typed expressions for our language. It has two similar branches, one to generate expression with type TBool, and another for type TNum. Each of them have two cases: (1) when the size (coming from the sized function) is bigger than zero, then we can generate both composed expressions, and values; (2) otherwise, we generate only values to stop the recursive process. To generate expressions for constructors And and Add, we use the function liftM2 from Control.Monad library[7], generating the two expected expressions for those constructors. The use of liftM[8] is similar. We use the QuickCheck function oneof to randomly select one of the alternatives in the given list. Furthermore, one can note that the recursive call to genExpr is decreasing the size parameter for each call.

```
genExpr :: Int → Ty → Gen Expr
genExpr size TBool
  | size > 0 = oneof ([ return BTrue , return BFalse , liftM2 And gt gt ])
  | otherwise = oneof ([ return BTrue , return BFalse ])
  where gt = genExpr (size `div` 2) TBool
genExpr size TNum
  | size > 0 = oneof ([ liftM Num n , liftM2 Add gt gt ])
  | otherwise = oneof ([ liftM Num n ])
  where n = genNat
    gt = genExpr (size `div` 2) TNum
```

After defining the genExpr function, it is possible to generate well-typed programs for the expression language to be used as input for testing properties.

2.2.2.3 QuickChecking Properties

QuickCheck can be used as a lightweight manner to check formal properties of programming languages. To accomplish this task, we have to write such properties as Haskell functions, similarly to what we have done before. The *progress* property

[7]The function liftM2 lifts a function (or constructor) with two arguments to a monadic counterpart, i.e., it receives two monadic values, applies the given function, and wraps the result in a new monadic value.

[8]The function liftM has a similar meaning, however it lifts a function with a single argument to a monadic counterpart.

says that either an expression is a value, or it can take a reduction step. This property is represented by the function prop_Progress, which should return a boolean value to be used with QuickCheck. Since the function step returns a monadic value, we use the function maybe[9] returning False when step fails, and True when step is performed successfully.

```
prop_Progress :: Expr → Bool
prop_Progress e = isVal e ∨ maybe (False) (const True) (step e)
```

Similarly, we can check the *preservation* property, which states that if an expression can take a reduction step, the reduced expression should remain with the same type. If the generated expression is not a value, we first infer its type by using the function typeof. Then we use function step to reduce the expression, inferring its type. After that, we check whether the type of the original expression is equal to the type of the evaluated expression.

```
prop_Preservation :: Expr → Bool
prop_Preservation e =
  isVal e ∨
  case (typeof e) of
    Just t → case (step e) of
      Just e′ → case (typeof e′) of
        Just t′ → t ≡ t′
        _ → False
      _ → False
    _ → False
```

Both properties are considered valid if all test cases are performed successfully, i.e., all of them returning True. In case of a failing property, QuickCheck presents the counter-example which caused the problem. This result is very useful to fix bugs in the semantics and implementations. We highlight here that only by testing a property it is impossible to assure that it is indeed valid. However, by using good tests we gain confidence that the semantics is working for the majority of the cases. Next section will bring us an alternative to definitely prove these properties.

[9]The maybe function takes three arguments: a default value, a function, and a Maybe value. If the Maybe value is Nothing, the function returns the default value (first argument). Otherwise it applies and returns the result of the function (second argument).

2.2.3 Evaluating Test Coverage

As a way to measure the quality of the generated tests, we can check how much of the code base of our interpreter was covered by the test suite. Such statistics are provided by the Haskell Program Coverage (HPC) tool (GILL; RUNCIMAN, 2007). Results of code coverage are presented in Figure 7.

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Exp	80%	4/5	100%	15/15	92%	51/55
Program Coverage	80%	4/5	100%	15/15	92%	51/55

Figure 7 – Test coverage results for the expression language.

HPC displays information in two different ways: reports with summary statistics (as shown above) and sources with color mark-up (which we will explore in Chapter 4). The information provided in Figure 7 is measured at three different levels: declarations (both top-level and local), alternatives (among several equations or case branches) and expressions (at every level) (GILL; RUNCIMAN, 2007), showing important information to improve the quality of test cases. In this case, the figure says that 4 of 5 functions (80%) were reached, all alternatives were covered, and almost all expressions (92%) were performed during the execution of our tests.

2.3 Formal Verification

Formal verification uses logical methods to establish claims expressed in precise mathematical terms. When combined with an interactive theorem prover, i.e. the use of proof assistants (like Agda, Coq, or Isabelle), the user can check whether these mathematical properties are satisfied, or verify that certain software (or hardware) meets its formal specifications. Having the properties established, proving correctness becomes a form of theorem proving, and its validity is checked by the system. Every claim (lemma or theorem) about the specification should be supported by a proof in a suitable axiomatic foundation. It means that every inference rule and every step of a calculation has to be justified by prior definitions and theorems (MOURA et al., 2015).

In this section, we present the basics of the dependently-typed programming language Agda, which is a system combining a powerful programming language with mechanisms to verify logic properties. After that, we mechanically check the expression language previously presented using two different (and equivalent) styles of formalization, extrinsic and intrinsic, which will be explored deeply in Chapter 5. The most important benefit Agda brings us is that the results implemented in the language are correct according to their specifications, and are checked every time by its underlying system.

2.3.1 Overview of Agda

Agda is a dependently-typed functional programming language based on Martin-Löf intuitionistic type theory (MARTIN-LÖF, 1998). Function types and an infinite hierarchy of types of types, Set ℓ, where ℓ is a natural number, are built-in. Everything else is a user-defined type. The type Set, also known as Set_0, is the type of all "small" types, such as Bool, String and List Bool. The type Set_1 is the type of Set and "others like it", such as Set $\to$ Bool, String $\to$ Set, and Set $\to$ Set. We have that Set ℓ is an element of the type Set $(\ell + 1)$, for every $\ell \geqslant 0$. This stratification of types is used to keep Agda consistent as a logical theory (SØRENSEN; URZYCZYN, 2006).

An ordinary (non-dependent) function type is written A $\to$ B and a dependent one is written (x : A) $\to$ B, where type B depends on x, or $\forall$ (x : A) $\to$ B. Agda allows the definition of *implicit parameters*, i.e. parameters whose values can be infered from the context, by surrounding them in curly braces: $\forall$ {x : A} $\to$ B. To avoid clutter, sometimes we omit implicit arguments from the source code presentation. The reader can safely assume that every free variable in a type is an implicit parameter.

As an example of Agda code, consider the following datatype of natural numbers and length-indexed lists, also known as vectors.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
data Vec (A : Set) : ℕ → Set where
  []    : Vec A zero
  _::_  : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Constructor [] builds empty vectors. The cons-operator (_ :: _)[10] inserts a new element in front of a vector of n elements (of type Vec A n) and returns a value of type Vec A (suc n). The Vec datatype is an example of a dependent-type, i.e., a type that uses a value (that denotes its length). The usefulness of dependent types can be illustrated with the definition of a safe list head function: head can be defined to accept only non-empty vectors, i.e., values of type Vec A (suc n), which have at least one element.

```
head : Vec A (suc n) → A
head (x :: xs) = x
```

In head's definition, the constructor [] is not used. The Agda type-checker can figure out, from head's parameter type, that argument [] to head is not type-correct, hence we do not have to give a definition for that case. In Haskell, head [] throws an exception instead.

[10] Agda supports the definition of mixfix operators. We can use underscores to mark arguments positions.

Another useful datatype is the finite type Fin, which is defined in Agda's standard library as:

```
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (suc n)
  suc  : ∀ {n} → Fin n → Fin (suc n)
```

Note that Agda supports the overloading of data type constructor names. Constructor zero can refer to type ℕ or Fin, depending on the context where the name is used. Type Fin n has exactly n inhabitants (elements), i.e., it is isomorphic to the set $\{0, ..., n-1\}$. An application of such type is to define a safe vector lookup function, which avoids the access of invalid positions.

```
lookup : ∀ {A n} → Fin n → Vec A n → A
lookup zero (x :: _) = x
lookup (suc idx) (_ :: xs) = lookup idx xs
```

Thanks to the propositions-as-types principle[11] we can interpret types as logical formulas and terms as proofs. An example is the representation of equality as the following Agda type:

```
data _≡_ ∀ {ℓ} {A : Set ℓ} (x : A) : A → Set where
  refl : x ≡ x
```

This type is called propositional equality. It defines that there is a unique evidence for equality, constructor refl (for reflexivity), asserting that the only value equal to x is itself. Given a predicate P : A → Set and a vector xs, the type All P xs is used to build proofs that P holds for all elements in xs and it is defined as:

```
data All (P : A → Set) : Vec A n → Set where
  []  : All P []
  _::_ : ∀ {x xs} → P x → All P xs → All P (x :: xs)
```

The first constructor specifies that All P holds for the empty vector and constructor _ :: _ builds a proof of All P (x :: xs) from proofs of P x and All P xs. Since this type shares the structure with vectors, some functions on Vec have similar definitions for type All. As an example, consider the function lookup, which extracts a proof of P for the element at position v :: Fin n in a Vec:

```
lookup : {xs : Vec A n} → Fin n → All P xs → P x
lookup zero (px :: _) = px
lookup (suc idx) (_ :: pxs) = lookup idx pxs
```

[11] Also known as Curry-Howard "isomorphism" (SØRENSEN; URZYCZYN, 2006).

An important application of Agda (and dependent-types) is to encode and prove properties about the specification of programming languages. We will discuss next two different forms to formally describe programming languages, achieving an equivalent soundness result.

2.3.2 Formalization Styles

Nowadays, there are two main approaches to formalize and prove type safety for a programming language. In the first one, called *extrinsic*, usually, the syntax, typing, and evaluation rules are described separately, and the common theorems of progress and preservation link the rules to prove type safety. In the second one, called *intrinsic*, the syntax and the typing judgments are expressed as a single definition, thus allowing only the representation of well-typed terms. Using such definition together with a terminating definitional interpreter, which implements the evaluation rules, type-safety is guaranteed by construction.

Next we present the formalization of the expression language presented in the beginning of this chapter. First we give its definition using the most traditional *extrinsic* approach, and then we present an *intrinsic* variant of the same language.

2.3.2.1 Extrinsic Formalization

To formalize a programming language in the extrinsic format, we follow the usual script: first we give the syntax, the semantics and typing rules, and then we prove the properties of progress and preservation to guarantee type-safety.

Syntax definition. Defining the syntax is similar to what was done in Haskell. We have a datatype Expr, and one constructor for each expressions[12].

```
data Expr : Set where
  BTrue BFalse : Expr
  Num          : ℕ → Expr
  _∧_ _+_      : Expr → Expr → Expr
```

The reader can note that, similarly to our Haskell definition (presented earlier), using this datatype we can construct expressions denoting terms that should not be considered well-typed such as $(\mathsf{Num}\ 1) + \mathsf{True}$, but our typing relation will forbid this.

Values. The presented language has three constructors to define values. Constructors VTrue and VFalse say that the boolean constants are values. Constructor VNum also represents a value giving that n is of type $\mathbb{N}$.

[12] Agda allows inline definitions of constructors resulting on the same type. We can see that for constructor True and False with type Expr, and for $\_\wedge\_$ and $\_+\_$ with type Expr → Expr → Expr.

```
data Val : Expr → Set where
  VTrue  : Val BTrue
  VFalse : Val BFalse
  VNum   : ∀ {n} → Val (Num n)
```

The inductive definition Val is indexed by an Expr, showing which syntactical constructor is associated with each value. We need this definition to define the reduction steps.

Dynamic semantics. The formalization of the evaluation for the expression language using the extrinsic approach follows the small-step semantics presented in Figure 3. The Agda code is basically a translation of the presented inference rules. The first three constructors deal with the arithmetic operator, and the last three constructors deal with the boolean operator. We use the Val datatype to force some of the expressions to be values in some rules. Following the same idea of the small-step rules, we rename Agda's addition operator ($+$ to $\oplus$) to avoid name conflicts.

```
data _⟶_ : Expr → Expr → Set where
  S-Add₁ : ∀ {e₁ e₁′ e₂}
         → e₁ ⟶ e₁′
         → e₁ + e₂ ⟶ e₁′ + e₂
  S-Add₂ : ∀ {v₁ e₂ e₂′}
         → Val v₁
         → e₂ ⟶ e₂′
         → v₁ + e₂ ⟶ v₁ + e₂′
  S-Add  : ∀ {n₁ n₂}
         → Val (Num n₁)
         → Val (Num n₂)
         → (Num n₁) + (Num n₂) ⟶ Num (n₁ ⊕ n₂)
  S-And₁ : ∀ {e₁ e₁′ e₂}
         → e₁ ⟶ e₁′
         → e₁ ∧ e₂ ⟶ e₁′ ∧ e₂
  S-And₂ : ∀ {e₂}
         → Val BTrue
         → BTrue ∧ e₂ ⟶ e₂
  S-And₃ : ∀ {e₂}
         → Val BFalse
         → BFalse ∧ e₂ ⟶ BFalse
```

Syntax of types. The present formalization presumes the existence of only two types: TBool representing booleans, and TNum representing numbers.

```
data Ty : Set where
  TBool TNum : Ty
```

Expression typing. The $\vdash \_ : \_$ relation encodes the typing rules for the expression language, indicating that an expression Expr has type Ty. Again, the Agda encoding is very similar to the rules presented in Figure 6. First three constructors give types for values BTrue, BFalse, and Num. Constructor T-Add will have type TNum giving that both e_1 and e_2 have type TNum. Similarly, T-And has type TBool if both e_1 and e_2 have type TBool.

```
data ⊢ _ : _ : Expr → Ty → Set where
  T-True  : ⊢ BTrue : TBool
  T-False : ⊢ BFalse : TBool
  T-Num   : ∀ {n} → ⊢ Num n : TNum
  T-Add   : ∀ {e₁ e₂}
          → ⊢ e₁ : TNum
          → ⊢ e₂ : TNum
          → ⊢ e₁ + e₂ : TNum
  T-And   : ∀ {e₁ e₂}
          → ⊢ e₁ : TBool
          → ⊢ e₂ : TBool
          → ⊢ e₁ ∧ e₂ : TBool
```

Expressions obeying the rules expressed in this relation are considered well-typed, otherwise they are said to be ill-typed.

Soundness proofs. We now prove type soundness for the language tested before with QuickCheck using the extrinsic approach.

First we need to define formally the basic property of reduction and types, identifying the possible *canonical forms* (i.e., well-typed closed values) belonging to each type (PIERCE et al., 2018). The definition has one constructor for each value (C-True, C-False, and C-Num).

```
data Canonical : Expr → Ty → Set where
  C-True  : Canonical BTrue TBool
  C-False : Canonical BFalse TBool
  C-Num   : ∀ {n} → Canonical (Num n) TNum
```

And a proof linking each value with its respective type.

```
canonical : ∀ {v τ} → ⊢ v : τ → Val v → Canonical v τ
canonical T-True _ = C-True
canonical T-False _ = C-False
canonical T-Num _ = C-Num
```

The progress function represents the theorem with the same name presented earlier, stating that if a well-typed expression e has type τ, then it can make *Progress*, i.e., or e is a value, or it can take another reduction step. Before giving the proof, we define an inductive datatype to hold the result of our proof, with two constructors: Done when e is a value, and Step when e reduces to an e′[13].

```
data Progress (e : Expr) : Set where
  Step : ∀ {e′}
    → e ⟶ e′
    → Progress e
  Done : Val e
    → Progress e
```

The proof of *progress* for this language in Agda is straightforward: cases with values are finished with Done, and the respective Val constructor; the case for T-Add makes progress (applying the induction hypothesis) for e_1 and e_2 and finishes with the canonical values and the rule S-Add; and the case for T-And makes progress for e_1, finishing the proof with the rules $S\text{-}And_2$ or $S\text{-}And_3$ according to the respective canonical value[14].

```
progress : ∀ {e τ} → ⊢ e : τ → Progress e
progress T-True  =  Done VTrue
progress T-False  =  Done VFalse
progress T-Num  =  Done VNum
progress (T-Add e₁ e₂) with progress e₁
... | Step stp₁  =  Step (S-Add₁ stp₁)
... | Done v₁ with progress e₂
... | Step stp₂  =  Step (S-Add₂ v₁ stp₂)
... | Done v₂ with canonical e₁ v₁ | canonical e₂ v₂
... | C-Num | C-Num  =  Step (S-Add v₁ v₂)
progress (T-And e₁ e₂) with progress e₁
... | Step stp₁  =  Step (S-And₁ stp₁)
... | Done v₁ with canonical e₁ v₁
... | C-True  =  Step (S-And₂ v₁)
... | C-False  =  Step (S-And₃ v₁)
```

The preservation function also represents the theorem with the same name presented earlier, stating that if a well-typed expression e has type τ, and it takes a reduction step e ⟶ e′, then e′ remains with type τ. The proof proceeds by induction on the

[13]For clarity, we use the definition of a Progress datatype, following the ideas presented in Wadler's book (WADLER, 2018). We could achieve similar result using a disjunction operator.

[14]The '...' notation is used when the original arguments for the with constructor are the same in the new clauses.

typing relation, case splitting each reduction case. First three cases deal with the arithmetic operator, and the last three cases deal with the boolean operator. The reader can note that the Agda system is smart enough to figure out that it is not necessary to deal with values, since they cannot take any reduction step.

```
preservation : ∀ {e e′ τ} → ⊢ e : τ → e ⟶ e′ → ⊢ e′ : τ
preservation (T-Add r₁ r₂) (S-Add₁ stp) = T-Add (preservation r₁ stp) r₂
preservation (T-Add r₁ r₂) (S-Add₂ v₁ stp) = T-Add r₁ (preservation r₂ stp)
preservation (T-Add r₁ r₂) (S-Add v₁ v₂) = T-Num
preservation (T-And r₁ r₂) (S-And₁ stp) = T-And (preservation r₁ stp) r₂
preservation (T-And r₁ r₂) (S-And₂ v₁) = r₂
preservation (T-And r₁ r₂) (S-And₃ v₁) = r₁
```

2.3.2.2 Intrinsic Formalization

Another possibility to formalize a programming language is to define an intrinsically-typed syntax, combining both the syntax and the typing rules in a single relation, and to define a definitional interpreter, also known as functional big-step semantics, to define its evaluation. Using this approach, we can use dependent-types and the well-behaved features of the host language (Agda) to state soundness of the target language we are working on.

Intrinsically-typed syntax. Representing the typing rules combined with the language syntax is a well-known approach (ALTENKIRCH; REUS, 1999). Using such approach, only well-typed expressions are accepted by the host language, and ill-typed expressions are rejected by the compiler accusing type error. Considering this, the abstract syntax trees capture not only the syntactic properties of the language, but semantic properties as well, allowing programmers to reason about their programs as they write them rather than separately at the meta-logical level.

Now, the definition of expressions Expr is parameterized by a type Ty (the same presented for the extrinsic approach).

```
data Expr : Ty → Set where
  True False : Expr Bool
  Num : Nat → Expr Nat
  _+_ : Expr Nat → Expr Nat → Expr Nat
  _∧_ : Expr Bool → Expr Bool → Expr Bool
```

In this definition, the Expr datatype carries out information about the type of each expression being built. For example, the first two constructor lines define the value expressions with their types. And the last two lines define the operators present in our language. We can note that for these operators, the intrinsically-typed syntax defines

the expected type for each (left and right-hand side) expression. This way, Agda's type system can enforce that only well-typed terms could be written. A definition which uses the expression $(\mathsf{Num}\ 1) + \mathsf{True}$ will be rejected by Agda's type checker automatically.

Definition of values. In order to define our interpreter we need a way to define an intrinsically-typed value. We define the datatype Val indexed by a Ty. By using this definition, when interpreting the code, the produced results are converted to an Agda type (host language semantics). Thus, if the value represents a TBool, it results in the Agda's Bool type. Similarly, TNum results in a natural number.

```
Val : Ty → Set
Val TBool = Bool
Val TNum = ℕ
```

Definitional interpreter. Next we present a fairly standard definition of an interpreter for the studied language in Agda. Basically, what we do is to use the host language structure to evaluate the target language, implementing the big-step semantics presented in Figure 4. The three first defining equations are considering the evaluation of values in the target language, resulting in values in the Agda language. The case for Add, both side expressions (e_1 and e_2) are evaluated at once, then summing up their results with $\oplus$ operator. The process is similar to deal with booleans. The considerate reader can notice that the treatment of And differs from the big-step semantics. In our implementation we chose to let Agda evaluate booleans using its standard library definition. Again here, we rename the operators ($+$ to $\oplus$, and $\wedge$ to $\&\&$) to avoid name conflicts.

```
eval : ∀ {τ} → Expr τ → Val τ
eval BTrue = true
eval BFalse = false
eval (Num x) = x
eval (Add e₁ e₂) = eval e₁ ⊕ eval e₂
eval (And e₁ e₂) = eval e₁ && eval e₂
```

There are two points to highlight on the intrinsic approach. First, we can note that we do not have any error treatment in our interpreter. This is happening because we are working only with a (intrinsically) well-typed expression, so in this case, the subexpression types are guaranteed by construction by Agda's type checker. Second, by allowing only the representation of well-typed expressions, the *preservation* property is also assured by construction (the compiler checks that the two τ's in the function definition match), and by writing such evaluator in a total language like Agda, the *progress* property is consequently guaranteed.

For further information about Agda, see (NORELL, 2009; STUMP, 2016; WADLER, 2018).

2.4 Chapter's Final Remarks

In this chapter we have set the basic script for the rest of this thesis, by reviewing the basic ideas of operational semantics, property-based testing, and formal verification. We equipped this chapter with a basic language serving as example for setting the ground for all aspects we will work in the next chapters. We shall refer later to the examples presented here, to explain most advanced concepts involving property-based testing or formal verification of programming languages.

3 TARGET LANGUAGES

In this chapter we bring an overview of the languages we considered during our research about programming language subsets. We start with a classic programming language (λ-calculus), which is widely used to study concepts of the functional paradigm, to develop the techniques presented in this thesis. After that, we conduct a research to find a subset of a modern object-oriented calculus, considering the Java language, which is one of the most used programming language nowadays. We present a comparison among the formalizations and the criteria to choose the one to explore in depth the techniques discussed in this work.

3.1 Functional Subsets

In this section we study one of the most famous example of programming languages (at least in research circles), the λ-calculus. It is a well-known purely functional core calculus proposed by Church in 1932 (CHURCH, 1932) capable of expressing computation with only three syntactic constructors: variables, abstractions, and application. The λ-calculus is a universal model of computation equivalent to Turing machines, and serves as basis for most of the current functional programming languages. Roughly, this calculus consists of constructing lambda expressions and performing reductions operations on them, through function abstraction and application using variable binding and substitution (WADLER, 2018).

Here we present a variant of λ-calculus, called simply-typed lambda calculus (STLC), which consists of the same base language augmented with types. We discuss the introduction of base types, presenting the syntax, semantics, and type system for this small calculus. For a thorough introduction of the λ-calculus, the reader is directed to (HINDLEY; SELDIN, 2008; BARENDREGT, 1992). The goal of this section is to provide a minimal understanding of STLC, which will be referred in other parts of this thesis when implementing the studied techniques.

3.1.1 Simply-Typed Lambda Calculus

Pure λ-calculus has only three syntactic constructors. For simplicity, we augment the pure calculus with the boolean constants $true$ and $false$. The abstract syntax of STLC is defined in Figure 8. Besides constants, we have *variables*, function definition through *abstraction*, and function invocation with *application*. In the same figure, we also define an *abstraction* as a value for the STLC language, plus the two boolean constants. Similarly to the previous chapter, we use meta-variables e and v to range over expressions and values, following the same conventions. Here, we also let x, y, and z denote variables.

$e ::=$		expressions:
	$true$	constant true
	$false$	constant false
	x	variable
	$\lambda x.e$	abstraction
	$e\ e$	application
$v ::=$		values:
	$true$	true value
	$false$	false value
	$\lambda x.e$	abstraction value

Figure 8 – Abstract syntax for STLC.

Suppose we have a λ-expression $\lambda y.(\lambda z.x(y\ z))$. We say that y and z are bound variables and x is a free variable (as there is no visible abstraction binding it). This notion is important to clarify the *scope* of variables. For this calculus, we assume Barendregt's variable convention, which says that expressions do not have any name clashing.

The most important notion of λ-calculus (and consequently STLC) is *substitution* to define how to compute an expression. The computation is given by the *application* of functions to arguments (which themselves can be functions). Each step in the computation consists of rewriting an application whose left-hand side is an *abstraction*, substituting the right-hand side for the bound variable in the abstraction's body (PIERCE, 2002). We will write $(\lambda x.e_1)e_2 \longrightarrow [x \mapsto e_2]e_1$, where $[x \mapsto e_2]e_1$ means "the expression obtained by replacing all free occurrences of x in e_1 by e_2". The substitution operation is defined inductively in Figure 9.

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } x \neq y \\
[x \mapsto s[(\lambda y.e_1) & = & \lambda y.[x \mapsto s]e_1 & \\
[x \mapsto s](e_1\ e_2) & = & ([x \mapsto s]e_1)\ ([x \mapsto s]e_2) &
\end{array}
$$

Figure 9 – Variable substitution operation.

An expression of the form $(\lambda x.e_1)e_2$ is called a *redex*, i.e., a reducible expression, and the operation of rewriting a redex is called *beta-reduction*. Several different reduction strategies can be applied to λ-calculus (such as *call-by-value*, *call-by-name*, etc.), defining which redex (or redexes) should be reduced on the next step of evaluation.

We present the small-step semantics for STLC in Figure 10. Rule S-App$_1$ and S-App$_2$ are *congruence* rules (call-by-value), where the first evaluates the expression on the left of a function application, while the second evaluates the expression on the right, once the first is a value. Rule S-AppAbs uses the substitution operation to perform the actual computation, substituting all free occurrences of x on the λ-expression body e_1. The reader can note that we are presenting a *call-by-value* evaluation strategy, since we are only applying the substitution if the right expression is a value (v_2).

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{[S-App}_1\text{]} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'}\ \text{[S-App}_2\text{]}$$

$$(\lambda x.e_1)v_2 \longrightarrow [x \mapsto v_2]e_1\ \text{[S-AppAbs]}$$

Figure 10 – Small-step semantics for STLC.

In order to define the type system for STLC, first we define the set of types in Figure 11.

$T ::=$		types:
	$Bool$	type of booleans
	$T \to T$	type of functions

Figure 11 – Simple types for STLC.

Type-checking in STLC can be carried out structurally. Expressions (on the language we are studying) can have boolean or function types. However, when analyzing λ-expressions of the form $(\lambda x.e_1)\ e_2$, the type which should be assigned to the whole program depends on that assigned to the expression e_2 according to the form of expression body e_1. This dependency can be expressed in terms of assumptions about the free variable x occurring in e_1, according to the type assigned to e_2. The standard procedure is to define a context Γ as a type environment, to hold the association of variables and types. Having this setting, a typing rule is expressed as $\Gamma \vdash e : T$, stating that e has type T according to variables in the Γ context. Figure 12 presents the definition of the type environment and the lookup operations (WADLER, 2018).

In this figure, $\varnothing$ represents an empty context (we can see the context as a list), and $\Gamma, x : T$ represents the operation of extending a typing environment with variable x having type T. We also use next $dom(\Gamma)$ to denote the set of variables in the domain of context Γ. Rule E-Lkp$_1$ is the base case, when the variable appears in the current

$$
\begin{array}{rl}
\Gamma ::= & \text{contexts:} \\
\varnothing & \text{empty context} \\
\Gamma, x : T & \text{variable binding}
\end{array}
$$

$$\Gamma, x : T \vdash x : T \quad \text{[E-Lkp}_1\text{]} \qquad \frac{\Gamma \vdash x_2 : T_2 \quad x_1 \neq x_2}{\Gamma, x_1 : T_1 \vdash x_2 : T_2} \quad \text{[E-Lkp}_2\text{]}$$

Figure 12 – Environment definition and operations.

scanning position, and rule E-Lkp$_2$ is the recursive case, looking up further on the list, when the variable is different from the one being sought.

Figure 13 summarizes the typing judgment rules for STLC. Rules T-True and T-False define the type for the boolean constants. Rule T-Var looks up the Γ environment (using the rules presented earlier) and returns the type T associated with the variable represented by x. Rule T-Abs assigns a function type ($T_1 \to T_2$) to a λ-expression provided under the assumption that x will have type T_1, and augmenting the Γ context with this information to obtain the type T_2 for the body expression e_2. Lastly, rule T-App assigns the type T_2 to a function application, where e_1 (using the assumptions in Γ) should have a function type ($T_1 \to T_2$), and e_2 should be of type T_1.

$$\Gamma \vdash true : Bool \quad \text{[T-True]} \qquad \Gamma \vdash false : Bool \quad \text{[T-False]}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{[T-Var]} \qquad \frac{\Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \lambda x.e_2 : T_1 \to T_2} \quad \text{[T-Abs]}$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2} \quad \text{[T-App]}$$

Figure 13 – Type system for STLC.

With the dynamic and static semantics for STLC in hand, we can prove soundness similarly to the previous chapter. Here we focus only on the important aspects of the proof that differ from the previous presentation. We start off by defining the canonical forms of values.

Lemma 2 (Canonical Forms)**.** *Let v be a well-typed value such that $\varnothing \vdash v : T$. Then:*

1. *if T is of type $Bool$, then v is either $true$ or $false$.*
2. *if T is of type $T_1 \to T_2$, then v has form $\lambda x.e$ and $\varnothing, x : T_1 \vdash e : T_2$.*

Proof. Immediate from the definition of values (Figure 8) and typing rules (Figure 11). □

Then we define the progress theorem. The statement of this theorem needs only one small change. We are interested only in *closed* expressions, without free variables, i.e., the type environment is empty (PIERCE, 2002).

Theorem 3 (Progress)**.** *Let e be a well-typed closed expression such that $\varnothing \vdash e : T$. Then either e is a value or there is some expression e' such that $e \longrightarrow e'$.*

Proof. Straightforward induction on typing derivations. The case for booleans are the same presented earlier. The variable case cannot happen, because e is a close expression. The abstraction case is immediate, since abstractions are values.

Case e represents an application, we have that $e = e_1\ e_2$, with $\varnothing \vdash e_1 : T_1 \to T_2$ and $\varnothing \vdash e_2 : T_1$. By the induction hypothesis both e_1 and e_2 are values, or can take a step. If they can take a step either S-App$_1$ or S-App$_2$ can be applied (according to the case). If both are values, then by the canonical lemma, we know that e_1 has the form $\lambda x.e_b$, and so rule S-AppAbs applies to e. □

To prove that evaluation preserves typing (preservation theorem), we need first to define some extra lemmas. The main difference from the proof of preservation presented in Chapter 2 is that now we have a context Γ, which stores information about variables to be used during evaluation, and we need to state some properties about it. We recall that we assume the Barendregt's variable convention.

First lemma says that we can permute elements of a context, without changing the typing statements that can be derived under it.

Lemma 3 (Permutation)**.** *If $\Gamma \vdash e : T$ and Γ_p is a permutation of Γ then $\Gamma_p \vdash e : T$.*

Proof. Straightforward induction on typing derivations. □

The second lemma says that if we can derive a type from an expression e, we should be able to derive the same type after adding a new (different) variable in the context.

Lemma 4 (Weakening)**.** *If $\Gamma \vdash e : T$ and $x \notin dom(\Gamma)$, then $\Gamma, x : T_1 \vdash e : T$.*

Proof. Straightforward induction on typing derivations. □

Using these two lemmas, we can prove another important property of the typing relation, stating that types are preserved after variable substitution.

Lemma 5 (Substitution)**.** *Suppose $\Gamma, x : T_x \vdash e : T$, where e is any expression in STLC. If $\varnothing \vdash v : T_x$, then $\Gamma \vdash [x \mapsto v]e : T$.*

Proof. By induction on a derivation of the statement $\Gamma, x : T_x \vdash e : T$ with case analysis on the shape of e. The cases for boolean rules are immediate.

Case e represents a variable, from the assumptions of the T-Var rule, we have that $e = y$ with $y : T \in (\Gamma, x : T_x)$. If $y = x$, then $[x \mapsto v]y = v$, and the result is $\Gamma \vdash v : T_x$, which is an assumption of this lemma. Otherwise, $[x \mapsto v]y = y$, and the result is immediate.

Case e represents an abstraction, from the assumptions of the T-Abs rule, we have that $e = \lambda y.e_1$, $T = T_1 \to T_2$, and $\Gamma, x : T_x, y : T_1 \vdash e_1 : T_2$. By the Barendregt's convention we may assume $x \neq y$. Using the permutation lemma, we obtain $\Gamma, y : T_1, x : T_x \vdash e_1 : T_2$. Using the weakening lemma, we obtain $\Gamma, y : T_1 \vdash v : T_x$. By the induction hypothesis, $\Gamma, y : T_1 \vdash [x \mapsto v]e_1 : T_2$. By T-Abs, $\Gamma \vdash \lambda y.[x \mapsto v]e_1 : T_1 \to T_2$, which is what we need, since by the definition of substitution, $[x \mapsto v]e = \lambda y.[x \mapsto v]e_1$.

Case e represents an application, from the assumptions of the T-Add rule, we have that $e = e_1\ e_2$, with $\Gamma, x : T_x \vdash e_1 : T_1 \to T_2$, $\Gamma, x : T_x \vdash e_2 : T_1$, and $T = T_2$. By the induction hypothesis, $\Gamma \vdash [x \mapsto v]e_1 : T_1 \to T_2$ and $\Gamma \vdash [x \mapsto v]e_2 : T_1$. By T-Add, $\Gamma \vdash [x \mapsto v]e_1\ [x \mapsto v]e_2 : T$, i.e., $\Gamma \vdash [x \mapsto v](e_1\ e_2) : T$. □

Having the substitution lemma, we can easily show the preservation property.

Theorem 4 (Preservation)**.** *Let e be a well-typed expression such that $\varnothing \vdash e : T$. Then $e \longrightarrow e'$ implies $\varnothing \vdash e' : T$.*

Proof. Induction on typing derivations and case analysis on the evaluation rules. The case for booleans and λ-expressions are impossible, since both represent values, which cannot take a step. Similarly, the case for variable is impossible, since we are working with closed values.

Case e represents an application, from the assumptions of the T-Add rule, we have that $e = e_1\ e_2$, with $\varnothing \vdash e_1 : T_1 \to T_2$ and $\varnothing \vdash e_2 : T_1$. By the evaluation rules, we have three rules by which $e \longrightarrow e'$. From rule S-App$_1$, we have that $e' = e'_1\ e_2$, where $e_1 \longrightarrow e'_1$. We can apply the induction hypothesis to ensure that $\varnothing \vdash e'_1 : T_1 \to T_2$. Combining this with the facts that $\varnothing \vdash e_2 : T_1$, we can apply the rule T-Add to conclude that $\varnothing \vdash e'_1\ e_2 : T_2$, that is $\varnothing \vdash e' : T_2$. The treatment for rule S-App$_2$ is similar. From rule S-AppAbs we have that $e' = [x \mapsto v_2]e_1$, and the result follows by the substitution lemma. □

In this section we chose to show the complete lemmas, theorems, and proofs to demonstrate soundness of the presented STLC setting, because we gather information from different sources, such as (PIERCE, 2002; WADLER, 2018; PIERCE et al., 2018). In later sections, we will prefer to point out the references in which the proofs are written, and omit them from this text.

3.2 Object-Oriented Subsets

This section presents the results of our research on finding a formal subset for a modern object-oriented language to be applied in the next chapters of this thesis. The text presented here is a summary of our paper “Formal Semantics for Java-like Languages and Research Opportunities” (FEITOSA; RIBEIRO; DU BOIS, 2018a). We chose to look into subsets of Java, since it is the most used programming language nowadays (TIOBE.COM, 2019), and also because this language is being adopted in many large projects, where applications reach a level of complexity for which only manual testing and human inspection are not enough to guarantee quality in software development.

Java is a statically, strongly typed, object-oriented, multi-threaded language. Except for threads, it is completely deterministic. The official specification of the Java language is the JLS (ORACLE.COM, 2018). JLS has 755 pages and 19 chapters; more than 650 pages were used to describe the language and its behavior. Java is distributed as part of the Java Development Kit (JDK) and currently is in version 10. At the imperative level, this language has 38 operators (JLS §3.12), 18 statements (§14), and some dozens of expressions (§15), among other features, and is evolving over time (BOGDANAS; ROSU, 2015). A Java program can be represented by a combination of several of its features. Considering that, the formalization (and update) of the whole language becomes an almost impossible task, justifying the need for definitions of formal subsets for Java.

Indeed, there exist several studies on the formalization of parts of the Java language (FLATT; KRISHNAMURTHI; FELLEISEN, 1998; DROSSOPOULOU; EISENBACH, 1999; IGARASHI; PIERCE; WADLER, 2001; KLEIN; NIPKOW, 2006; BOGDANAS; ROSU, 2015; FARZAN; CHEN; MESEGUER, 2004; STARK; BORGER; SCHMID, 2001), and we have defined some criteria to select some of them to be presented in this text. Initially, we looked up for projects that describe the semantics of Java, particularly by structural operational semantics, filtering those that presented proofs of type-safety, both in formal or informal (non-mechanized) ways. From these, we selected the four most popular formalisms, i.e., those with the higher number of citations according to Google Scholar (GOOGLE, 2018) database. Using this criterion, Featherweight Java could be considered the most popular, with almost 900 citations, followed by Classic Java, with approximately 500 quotes. Java$_S$ and Jinja currently present between 300 and 400 citations. The remainder of this section summarizes the selected formalizations, discussing their completeness and conformance with the official specification of Java, and comparing them with each other.

3.2.1 Featherweight Java

Featherweight Java (FJ) (IGARASHI; PIERCE; WADLER, 2001), is a minimal core calculus for Java, in the sense that as many features of Java as possible are omitted while maintaining the essential flavor of the language and its type system. However, this fragment is large enough to include many useful programs. A program in FJ consists of a declaration of a set of classes and an expression to be evaluated, that corresponds to the *public static void main* method of Java.

FJ is related to Java, as 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. The following example shows how classes can be modeled in FJ. There are three classes, A, B, and Pair, with constructor and method declarations.

```
class A extends Object {
  A () {super (); }
}
class B extends Object {
  B () {super (); }
}
class Pair extends Object {
  A fst; B snd;
  Pair (A fst, B snd) {
    super ();
    this.fst = fst;
    this.snd = snd;
  }
  Pair setfst (A newfst) {
    return new Pair (newfst, this.snd);
  }
}
```

FJ semantics provides a purely functional view without side effects. In other words, attributes in memory are not affected by object operations (PIERCE, 2002). Furthermore, interfaces, overloading, call to base class methods, null pointers, base types, abstract methods, statements, access control, and exceptions are not present in the language (IGARASHI; PIERCE; WADLER, 2001).

Because 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.

Figure 14 presents the syntactic definitions originally proposed for FJ, where L refers to class declarations, K and M to constructors and methods respectively, and finally, e represents the expressions of that language. We assume that the set of variables includes the special variable $this$ and $super$ is a reserved keyword. Throughout this document, we write $\bar{f}$ as shorthand for a possibly empty sequence f_1,...,f_n (similarly for $\bar{C}$, $\bar{x}$, $\bar{e}$, etc.).

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

Figure 14 – Syntactic definitions for FJ.

Figure 15 presents the evaluation rules originally proposed for FJ, formalizing how to evaluate *attribute access* (R-Field), *method invocation* (R-Invk), and *casts* (R-Cast) (IGARASHI; PIERCE; WADLER, 2001), the only three possible terms to be used in the *main program*. The presented functions, $fields$ and $mbody$, are also formalized in the original paper, representing respectively a way to obtain a list of attributes of some class C, and the body expression inside a method m which belongs to a given class C. In the *method invocation* rule, we write $[\bar{x} \mapsto \bar{u}, this \mapsto new\ C(\bar{v})]e_0$ for the result of replacing x_1 by u_1,...,x_n by u_n, and $this$ by "$new\ C(\bar{v})$" in expression e_0. In the *cast* rule, the symbol $<:$ is used to express the sub-typing relation between C and D, stating that C is a subtype of D. These symbols are also used throughout the document.

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

$$\frac{mbody(m, C) = (\bar{x}, e_0)}{new\ C(\bar{v}).m(\bar{u}) \longrightarrow [\bar{x} \mapsto \bar{u}, this \mapsto new\ C(\bar{v})]e_0}\ \text{[R-Invk]}$$

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

Figure 15 – Evaluation rules for FJ.

The typing rules for expressions are in Figure 16. There we can note the use of an environment Γ, which represents a finite mapping from variables to types, written $\bar{x} : \bar{C}$. We let $\Gamma(x)$ denote the type C such that $x : C \in \Gamma$. The typing judgment for expressions has the form $\Gamma \vdash e : C$, read as "in the environment Γ, the expression e has type C". The typing rules are syntax directed, with one rule for each form of expression, except for casts. Most of the typing rules are straightforward adaptations of the rules in Java: the rule (T-Var) checks if the variable x is in the Γ context and gets its type; rule (T-Field) uses the function $fields$ to obtain the field type; the rules for method invocations (T-Invk) and for constructors (T-New) check that each actual parameter has a type that is subtype of the corresponding formal parameter type; the last three rules are related to casts, considering *upcasts*, *downcasts*, and unrelated objects. The latter was added to allow proofs of type soundness.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \text{ [T-Var]} \quad \frac{\Gamma \vdash e_0 : C_0 \quad fields(C_0) = \bar{C}\,\bar{f}}{\Gamma \vdash e_0.f_i : C_i} \text{ [T-Field]}$$

$$\frac{\Gamma \vdash e_0 : C_0 \quad mtype(m, C_0) = \bar{D} \to C \quad \Gamma \vdash \bar{e} : \bar{C} \quad \bar{C} <: \bar{D}}{\Gamma \vdash e_0.m(\bar{e}) : C} \text{ [T-Invk]}$$

$$\frac{fields(C) = \bar{D}\,\bar{f} \quad \Gamma \vdash \bar{e} : \bar{C} \quad \bar{C} <: \bar{D}}{\Gamma \vdash new\ C(\bar{e}) : C} \text{ [T-New]} \quad \frac{\Gamma \vdash e_0 : D \quad D <: C}{\Gamma \vdash (C)\ e_0 : C} \text{ [T-UCast]}$$

$$\frac{\Gamma \vdash e_0 : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)\ e_0 : C} \text{ [T-DCast]} \quad \frac{\Gamma \vdash e_0 : D \quad C \not<: D \quad D \not<: C}{\Gamma \vdash (C)\ e_0 : C} \text{ [T-SCast]}$$

Figure 16 – Typing rules for FJ.

For brevity, the formalization of sub-typing relation, auxiliary definitions, congruence and sanity checks for methods and classes were omitted here, but can be found in the original FJ paper (IGARASHI; PIERCE; WADLER, 2001).

An important contribution of FJ is the soundness proofs for the proposed type system. We present the *Type Soundness* theorem as an example, to show the way proofs were modeled by the authors.

Theorem 5 (Type Soundness)**.** *If $\varnothing \vdash e : C$ and $e \to^* e'$ with e' a normal form, then e' is either a value v with $\varnothing \vdash v : D$ and $D <: C$, or an expression containing $(D)\ new\ C(\bar{e})$ where $C <: D$.*

Proof. Immediate from Subject Reduction (Theorem 2.4.1) and Progress (Theorem 2.4.2) theorems found in the original paper (IGARASHI; PIERCE; WADLER, 2001). □

3.2.2 ClassicJava

ClassicJava (FLATT; KRISHNAMURTHI; FELLEISEN, 1998, 1999) is a also small subset of sequential Java. To model its type structure, the authors use type elaborations (POTTIER, 2014), where it is verified that a program defines a static tree of

classes and a directed acyclic graph (DAG) of interfaces. For the semantics, rewriting techniques were used, where evaluation is modeled as a reduction on expression-store pairs in the context of a static type graph. The class model relies on as few implementation details as possible.

In ClassicJava, a program P is represented by a sequence of classes and interfaces followed by an expression. Each class definition consists of a sequence of field declarations and a sequence of method declarations. In the interfaces, the difference is that there are only methods. A method body in a class can be *abstract* when the method should be overridden in a subclass or can be an expression. In the case of interfaces, the method body must be always *abstract*. Similarly to Java, objects are created with the new operator, but the constructors are omitted in the proposed specification. Thus, instance variables are initialized to $null$. There are also constructors that represents casts ($view$ operator) and assignments (let operator). Figure 17 shows the formal syntax of ClassicJava.

$P ::=$		program specification
	$defn^*\ e$	
$defn ::=$		class and interface declarations
	$class\ c\ extends\ c\ implements\ i^*\ \{field^*\ meth^*\}$	
$field ::=$		field statement
	$t\ fd$	
$meth ::=$		method declarations
	$t\ md(arg^*)\ \{body\}$	
$arg ::=$		argument list
	$tvar$	
$body ::=$		method body declarations
	$e \mid abstract$	
$e ::=$		expressions:
	$new\ c$	instancing a class
	var	a variable name or $this$
	$null$	$null$ value
	$e:\ c.fd$	field access
	$e:\ c.fd = e$	field assignment
	$e.md(e^*)$	method invocation
	$super \equiv this:\ c.md(e^*)$	method invocation
	$view\ t\ e$	cast
	$let\ var = e\ in\ e$	assignment

Figure 17 – Syntactic definitions for ClassicJava.

To be considered valid, a program should satisfy a number of simple predicates and relations, for example: *ClassOnce* indicates that a class name is declared only once, *FieldOncePerClass* checks if field names in each class are unique, *MethodOncePerClass* checks oneness for method names, *InterfacesAbstract* verifies that methods in interfaces are *abstract*, relation $\prec^c_P$ associates each class name in P to the class it ex-

$$
\begin{array}{ll}
 & E = [] \mid E : c.fd \mid E : c.fd = e \mid v : c.fd = E \\
e = ... \mid object & \quad \mid E.md(e...) \mid v.md(v...E\ e...) \\
v = object \mid null & \quad \mid super \equiv v : c.md(v...E\ e...) \\
 & \quad \mid view\ t\ E \mid let\ var = E\ in\ e
\end{array}
$$

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

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

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

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

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

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

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

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

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

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

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

Figure 18 – Evaluation rules for ClassicJava.

tends, relation $\Subset^c_P$ (overloaded) capture the field and method declarations of P, and so on. The complete list of auxiliary definitions can be found in the original paper (FLATT; KRISHNAMURTHI; FELLEISEN, 1998).

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

Figure 18 shows the operational semantics for ClassicJava. By looking at the *get* rule, for example, it is possible to note that a search for an attribute *fd* in a class *c'* is performed by using the *field record* $\mathcal{F}$, resulting in a value *v*. For the case of the *call* rule one can note that it invokes a method by rewriting the method call expression

with the body of the invoked method, syntactically replacing argument variables in this expression with the supplied argument values and the special variable *this*. The other rules can be understood in a similar way.

The type elaboration rules translate expressions for field access or method call into annotated expressions. For instance, when a field is used, the annotation contains the compile-time type of the instance expression, which determines the class containing the declaration of the accessed field. The complete typing rules can be found in the original paper (FLATT; KRISHNAMURTHI; FELLEISEN, 1998). There the authors show that a program is well-typed if its class definitions and final expressions are well-typed. A definition, in turn, is well-typed when its field and method declarations use legal types and the method body expressions are well-typed. Finally, expressions are typed and elaborated in the context of an environment that binds free variables to types.

The authors also have presented formal proofs, which aim to guarantee the safety of this calculus, i.e., an evaluation cannot get stuck. This property was formulated through the *type soundness* theorem, where an evaluation step yields one of two possible configurations: either a well-defined error state or a new expression-store pair. In the latter case, there exists a new type environment that is consistent with the new store, and it establishes that the new expression has a type below t. The complete proof is available in an extended version of the original ClassicJava paper (FLATT; KRISHNAMURTHI; FELLEISEN, 1999).

3.2.3 Java$_S$, Java$_{SE}$ and Java$_R$

Another formal semantics for a subset of Java was developed by Drossopoulou and Eisenbach, where they have presented an operational semantics, a formal type system, and sketched[1] an outline of the type soundness proof (DROSSOPOULOU; EISENBACH, 1997; DROSSOPOULOU; EISENBACH; KHURSHID, 1999; DROSSOPOULOU; EISENBACH, 1999). This subset includes primitive types, classes with inheritance, instance variables, and instance methods, interfaces, shadowing of instance variables, dynamic method binding, statically resolvable overloading of methods, object creation, null pointers, arrays and a minimal treatment of exceptions.

The author's approach was to define Java$_S$, which is a provably *safe* subset of Java containing the features listed above, a term rewrite system to describe the operational semantics and a type inference algorithm to describe compile-time type checking. They also prove that program execution preserves the type up to the subclass/subinterface relationship (DROSSOPOULOU; EISENBACH, 1999). Furthermore, the type system

[1] The authors provided informal (and incomplete) proofs to argue that the type system of Java is sound. The work of Syme (SYME, 1999) complemented these proofs and provided a machine-checked version of them in the *Declare* proof assistant.

was described in terms of an inference system.

This formal calculus was designed as a series of components, where Java$_S$ is a formal representation of the subset of Java semantics, Java$_{SE}$ is an enriched version of Java$_S$ containing compile-time type information, and Java$_R$, which extends Java$_{SE}$ and describes the run-time terms. Figure 19 shows the syntax of Java$_S$.

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

Figure 19 – Syntactic definitions for Java$_S$.

In Java$_S$ a program consists of a sequence of class bodies. Class bodies consist of a sequence of method bodies. Method bodies consist of the method identifier, the names and the types of the arguments, and a statement sequence. It is required exactly one $return$ statement in each method body, which should be the last statement. Also, the authors considered only conditional statements, assignments, method calls, try and $throw$ statements. This was done because iteration and other constructors can be achieved in terms of conditionals and recursion.

The calculus considers values, method calls, and instance variable accesses. The values are primitives (such as $true$, 4, 'c', etc.), references or arrays. References are $null$, or pointers to objects. The expression $new\ C$ creates a new object of class C, whereas the expression $new\ T[e_n]^+$ creates a n dimensional array. Also, pointers to objects are implicit.

As the other Java calculi, this proposal also models the class hierarchy, proposing the $\sqsubseteq$ relationship. Moreover, they also describe the environment, usually denoted by a Γ, using the *BNF* notation and containing both the subclass and interface hierarchies and variable type declarations. The environment also holds the type definitions of all variables and methods of a class and its interface. For the sake of brevity, this grammar (DROSSOPOULOU; EISENBACH, 1999) was omitted from here.

The following piece of code serves to demonstrate the Java$_S$ syntax and some of the features tacked by the authors.

```
p_s = Phil ext Object {
    think is λ y : Phil. {...}
    think is λ y : FrPhil. {...}
}
FrPhil ext Phil {
    think is λ y : Phil. {this.like := oyster; ...}
}
```

Considering the presented program, the environment Γ is:

```
Γ = Phil ext Object {
        like : Truth ,
        think : Phil → Phil
        think : FrPhil → Book} ,
    FrPhil ext Phil {like : Food ,
        think : Phil → Phil}
```

The operational semantics for this language was defined as a ternary rewrite relationship between configurations, programs, and configurations. Configurations are tuples of Java$_R$ terms and states. The terms represent the part of the original program remaining to be executed. The method calls evaluation were described as textual substitutions (DROSSOPOULOU; EISENBACH, 1999). There are three relations for specifying the reduction of terms, one for each syntax category: $\overset{exp}{\rightsquigarrow}_{(\Gamma,p)}$, $\overset{var}{\rightsquigarrow}_{(\Gamma,p)}$, $\overset{stmt}{\rightsquigarrow}_{(\Gamma,p)}$. Global parameters are an environment Γ (containing the class and interface hierarchies, needed for runtime type checking) and the program p being executed (SYME, 1999).

The proposed rewrite system has 36 rules in total, where 15 of them are “redex” rules that specify the reduction of expressions in the cases where sub-expressions have reductions. A sample of this rules is:

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

There are 11 rules for dealing with the generation of exceptions: 5 for *null* pointers dereferences, 4 for bad array index bounds, one for bad size when creating a new array and one for runtime type checking when assigning to arrays. A simple example is:

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

In this calculus, a term is *ground* if it is in normal form, i.e. no further reduction can be made. The presented rule results a *null pointer exception* when one tries to assign a value to a *null* pointer. The complete set of rules, such as for field dereferencing, variable lookup, class creation, field assignment, local variable assignment, conditional statements, method call and for dealing with arrays, are covered in the original paper (DROSSOPOULOU; EISENBACH, 1997). This presentation also omits the type system rules and auxiliary definitions.

By proving subject reduction and soundness, the authors argue that the type system of Java$_S$ is sound, in the sense that unless an exception is raised, the evaluation of any expression will produce a value of a type "compatible" with the type assigned to it by the type system.

3.2.4 Java$_{light}$ and Jinja

Jinja (NIPKOW, 2003; KLEIN; NIPKOW, 2006) is a Java-like programming language with a formal semantics designed to exhibit core features of Java, proposed by Nipkow and improved in conjunction with Klein. According to the authors, the language is a compromise between the realism of the language and tractability and clarity of the formal semantics. It is also an improvement of Java$_{light}$ (NIPKOW; OHEIMB, 1998), enhancing the treatment of exceptions.

In contrast to others formalizations, they presented a big and a small-step semantics, which are independent of the type system, showing their equivalence. They also presented the type system rules, a definite initialization analysis, and the type safety proofs of the small-step semantics. Additionally, the whole development has been carried out in the theorem prover Isabelle/HOL (KLEIN; NIPKOW, 2017).

The abstract syntax of programs is given by the type definitions in Figure 20. A program is a list of *class declarations*. A class declaration consists of the name of the class and the class itself. A *class* consists of the name of its direct superclass, a list of field declarations, and a list of method declarations. A *field declaration* is a pair consisting of a field name and its type. A *method declaration* consists of the method name, the parameter types, the result type, and the method body. A *method body* is a pair of formal parameter names and an expression (KLEIN; NIPKOW, 2006).

Jinja is an imperative language, where all the expressions evaluate to certain values. Values in this language can be primitive, references, null values or the dummy

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

Figure 20 – Syntactic definitions for Jinja.

value *Unit*. As an expression-based language, the statements are expressions that evaluate to *Unit*. The following expressions are supported by Jinja: the creation of new objects, casting, values, variable access, binary operations, variable assignment, field access, field assignment, method call, block with locally declared variables, sequential composition, conditionals, loops, and exception throwing and catching. The following example shows a program source-code using this language.

```
class B extends A {field F : TB
method M : TBs → T1 = (pB, bB)}
class C extends B {field F : TC
method M : TCs → T2 = (pC, bC)}
```

In this example, the field F in class C hides the one in class B. The same occurs with the method M. This differs from Java, where methods can also be *overloaded*, which means that multiple declarations of M can be visible simultaneously since they are distinguished by their argument types.

In this language, everything (expression evaluation, type checking, etc.) is performed in the context of a program P. Thus, there are some auxiliary definitions, omitted from here, like *is-class*, *subclass*, *sees-method*, *sees-field*, *has-field*, etc., that can be used to obtain information that are inside the abstract syntax tree of a program to assist on the evaluation.

The evaluation rules were presented in two parts: first, the authors introduce a big-step or evaluation semantics, and then a small-step or reduction semantics. The big-step semantics was used in the compiler proof, and the small-step semantics in the type safety proof. As this language deals with effects, it was necessary to define a *state*, represented by a pair, which models a *heap* and a *store*. A store is a map from variable names to values and a heap is a map from address to objects.

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

$$\frac{\textit{P} \vdash \langle \textit{e, s}_0 \rangle \Rightarrow \langle \textit{addr a,(h,l)} \rangle \quad \textit{h a} = \lfloor \textit{(C, fs)} \rfloor \quad \textit{fs(F, D)} = \lfloor v \rfloor}{\textit{P} \vdash \langle \textit{e.F\{D\},s}_0 \rangle \Rightarrow \langle \text{Val } v, \text{(h,l)} \rangle} \text{ (R-Field)}$$

$$\frac{\begin{array}{c}\textit{P} \vdash \langle \textit{e,s}_0 \rangle \Rightarrow \langle \textit{addr a, s}_1 \rangle \quad \textit{P} \vdash \langle \textit{ps,s}_1 \rangle [\Rightarrow] \langle \textit{map Val } vs, \textit{(h}_2\textit{,l}_2\textit{)} \rangle \\ \textit{h}_2\ \textit{a} = \lfloor \textit{(C,fs)} \rfloor \quad \textit{P} \vdash \textit{C sees M: Ts} \rightarrow \textit{T = (pns, body) in D} \quad \mid vs \mid = \mid pns \mid \\ \textit{l'}_2 = [\textit{this} \mapsto \textit{Addr a, pns} [\mapsto] vs] \quad \textit{P} \vdash \langle \textit{body,(h}_2\textit{,l'}_2\textit{)} \rangle \Rightarrow \langle \textit{e'},\textit{(h}_3\textit{,l}_3\textit{)} \rangle\end{array}}{\textit{P} \vdash \langle \textit{e.M(ps),s}_0 \rangle \Rightarrow \langle \textit{e'},\textit{(h}_3\textit{,l}_2\textit{)} \rangle} \text{ (R-Method)}$$

Figure 21 – Partial big-step semantics for Jinja.

For the big-step semantics, the evaluation judgment is of the form $P \vdash \langle e, s \rangle \Rightarrow \langle e', s' \rangle$, where e and s are the initial expression and state, and e' and s' the final expression and state. Figure 21 shows some of the rules for Jinja's big-step semantics.

The first rule (R-New) first allocates a new address: function *new-Addr* returns a "new" address, that is, *new-Addr h = ⌊a⌋* implies *h a = None*. Then predicate *has-fields* computes the list of all field declarations in and above class C, and *init-fields* creates the default field table. The second (R-Field) evaluates e to an address, looks up the object at the address, indexes its field table with (*F, D*), and evaluates to the value found in the field table. The lengthiest rule presented here (R-Method) is the one for a method call. It evaluates e to an address a and the parameter list ps to a list of values vs, looks up the class C of the object in the heap at a, looks up the parameter names pns and the $body$ of the method M visible from C, and evaluates the body in a store that maps $this$ to *Addr a* and the formal parameter names to the actual parameter values. The final store is the one obtained from the evaluation of the parameters. The complete set of rules can be found at the original papers (NIPKOW, 2003; KLEIN; NIPKOW, 2006).

According to the authors, the small-step semantics was provided because the big-step semantics has several drawbacks, for example, it cannot accommodate parallelism, and the type safety proof needs a fine-grained semantics. The main difference between the two proposed semantics is that in the small-step they present *subexpression reduction*, which essentially describes the order that subexpressions are evaluated. Having the subexpressions sufficiently reduced, they describe the *expression reduction*. Most of that rules are fairly intuitive and many resemble their big-step counterparts (KLEIN; NIPKOW, 2006). These rules were omitted from this text.

In their papers, the authors relate the big-step with the small-step semantics, show the type-system rules and then prove its type-safety by showing the *progress* and *preservation* theorems for the proposed language. Additionally, the whole development of this project runs to 20000 lines of Isabelle/HOL text, which can be found online (KLEIN; NIPKOW, 2017).

3.2.5 Comparing the Most Used Semantics

The use of formal modeling can offer significant advances to the design of a complex system. The introduction of lightweight versions of a programming language, where complex features are dropped to enable rigorous arguments about key properties, allow a better understanding of the language characteristics, facilitate the investigation of novel constructions, and can be a useful tool for studying the consequences of extensions and variations.

However, choosing a formal model for a programming language when starting a research can be a difficult task, since there is a large number of projects in this area and several factors to be considered. For example, one project may need a more complete semantics, where a big number of features are included, while another project could render itself better to a more compact language, in order to study some specific extension that does not depend on a complete approach. A look at the related work can be a good starting point to choose among the variety of formalisms, because one can observe the pattern applied to these projects, and the similarities with the work to be developed. This section provides a comparison between the models presented in previous sections, helping on this choice process.

Table 1 shows which features are modeled on each of the discussed formal languages. Features of original Java that are not modeled in any of the presented languages (for example packages, access modifiers, λ-expressions, concurrency, reflection, among others) were suppressed from this table. For clarity, we split the table into categories, and we group some similar features for space optimization. We use three different kinds of support level, where ● means that the category or feature is fully supported by the presented formalism, ◐ stands for a partial support, and ○ when the feature is not supported at all. For presentation purposes we abbreviate Featherweight Java as FJ, ClassicJava as CJ, Java$_S$ as JS, and Jinja as JJ in the first line of the table. Our intention is to present the main functionalities for each studied language, thus some minor features may not appear in this table.

When looking at the table, we can identify two different patterns. The first, represented by Featherweight Java and ClassicJava, is compactness. The second, represented by Java$_S$ and Jinja, is completeness. While the first ones are concentrated on a minimal object-oriented formalism, the others formalized a larger subset of the Java language. These patterns can be useful to choose between one or another approach, so next, we discuss each pattern separately.

Featherweight Java and ClassicJava offer similar functionalities, where the goal of these projects was to define a core calculus that is as small as possible, capturing just the features of Java that are relevant for some particular task. In the case of FJ, the task was to analyze extensions of the core type system. The task of CJ was to analyze an extension of Java with mixins (FLATT; KRISHNAMURTHI; FELLEISEN, 1998), a

Feature	FJ	CJ	JS	JJ
Primitive Types and Values	○	○	●	◐
Integer	○	○	●	●
Boolean	○	○	●	●
String literal	○	○	●	○
Basic Statements	○	◐	●	●
Null values and Assignment	○	●	●	●
Conditionals	○	○	●	●
Loops and Sequences	○	○	●	●
Try-catch-finally	○	○	●	●
Math Operators	○	○	○	●
Basic OOP	◐	◐	◐	◐
Classes and inheritance	●	●	●	●
Interfaces	○	●	●	○
Casts	●	●	○	●
Constructors, super and this	●	◐	○	◐
Polymorphism and overriding	●	●	○	●
Overloading and static methods	○	○	○	○
Generics	●	○	○	○
Arrays	○	○	●	○
Formal Specifications	◐	◐	◐	●
Big-step semantics	○	○	○	●
Small-step semantics	●	●	●	●
Progress and preservation	●	●	●	●
Use of proof assistant	○	○	◐	●

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

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

feature of the Common Lisp language. The approach demonstrated by the authors of FJ is somewhat smaller than CJ, where the syntax, typing rules, and operational semantics of FJ take approximately three times less space than the other. Consequently, the soundness proofs are also correspondingly smaller.

The functionalities offered by Java$_S$ and Jinja are also similar. The goal of the first was to show that Java's type system is sound, while the goal of the second was to provide a formal semantics of the core features of Java, emphasizing on a unified model of the source language, the virtual machine, and the compiler. JS was one of the first steps toward a formal semantics for Java, and hence, it was an inspiration for later projects. However, because it explored a larger subset of an older version of Java, it is not usually taken as the basis for new projects, being useful for study purposes. In contrast, JJ was widely used as the basis for investigations of object-oriented characteristics. As JJ is not properly a subset of Java, it was also used on formalizations of other object-oriented languages. Because JJ also offers proofs, a

virtual machine, and a compiler verified in the theorem prover Isabelle/HOL, it seems to be a good formalism when a more complete formalism of Java is needed.

An indirect consequence of this section is that, although not covering all aspects of the presented formalisms, it can provide insights on useful criteria for others when choosing a formalism to be applied on their projects. Indeed, the comparison presented here helped us on choosing the formalism to apply the techniques presented in this thesis. We chose Featherweight Java as basis for our investigations, mainly because a smaller calculus fit better for our purposes. When comparing specifically FJ with CJ, and looking at the derived projects of both (FEITOSA; RIBEIRO; DU BOIS, 2018a), we could see that FJ was applied mostly in Java extensions or novel constructions in the object-oriented context, while CJ was applied in several projects to work with threads and concurrency. We also considered that as a reason to choose FJ, as well as because its formalization rules and proofs are more compact (and understandable) than the other.

3.3 Chapter's Final Remarks

This chapter summarized the programming languages we will use for the rest of this thesis. First, we introduced a variant of λ-calculus, called STLC, augmented with boolean constants and types, and we shown the soundness proofs of this calculus. Second, we presented the four most used subset of object-oriented programming languages, using a comparison among them to choose Featherweight Java as the formalism to apply the techniques we explored in this project.

4 TESTING PROPERTIES OF LANGUAGES

In the previous chapter we presented the semantics of two major languages to be explored in this thesis. In this chapter, we start working with the first branch of this thesis, where the idea is to explore property-based testing in order to check soundness properties of programming languages. We begin by developing a formal definition of a type-directed algorithm to generate random programs, first for STLC, then for the more complex language FJ. We then show how this algorithm can be implemented in Haskell, together with QuickCheck, and how the resulting system can be used for property-based testing. And finally, we evaluate the quality of our type-directed procedure by measuring the code coverage of hand-written interpreters for both presented languages. To show that our process can scale up for bigger languages, we extend FJ and our program generator with some Java 8 features, checking the soundness properties for the resulting calculus.

The first problem one faces when working with property-based testing is the creation of good test cases. 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 (ALLWOOD; EISENBACH, 2009).

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 (PAŁKA et al., 2011). We present next our efforts on this research field.

4.1 Simply-Typed Lambda Calculus

We propose in this section a generation algorithm to generate random programs for STLC, inspired by the approach of PAŁKA et al. (2011). Different form the previous works, we formalize a type-directed procedure having one generation rule for each typing rule considering the structural type system of STLC, and we also prove that our generation algorithm is sound with respect to the STLC typing rules.

4.1.1 Expression Generation

For our generation procedure we assume the existence of a function $\xi : [a] \to a$, which returns a random element from an input list. We also assume that $\mathbb{T}$ is a list of valid types for STLC, and that there exist a list $\overline{V_n}$ of possible variable names. We slightly abuse notation by using set operations on lists (sequences) and its meaning is as usual.

The expression generation for STLC is represented by the following judgment, where Γ is a typing environment, T is a STLC type (which can be $Bool$, $Bool \to Bool$, and so on), and e is the produced expression.

$$\Gamma; T \Rightarrow_e e$$

Having defined this general judgment, we present different generation rules for the syntactical elements of the STLC language presented in the last chapter.

Boolean value generation. The generation of boolean values is straightforward. We have one rule for each possible value.

$$\Gamma; Bool \Rightarrow_e true \quad \text{[G-True]}$$
$$\Gamma; Bool \Rightarrow_e false \quad \text{[G-False]}$$

Variable generation. The process for generating a variable constructor needs to select one element in the Γ context whose type is T. First we select all x such that $\Gamma(x) = T$, producing a list of candidates (which can be empty). After that, we select one element of the produced list. If the list of candidates is empty, the rule [G-Var] cannot be used.

$$\Gamma; T \Rightarrow_e \xi(\{x \mid \Gamma(x) = T\}) \quad \text{[G-Var]}$$

Abstraction generation. To generate an abstraction, we select one variable name x from the list of variable names $\overline{V_n}$, and then we add this variable in the Γ context with type T_1 to generate the body expression e with type T_2. The result of rule [G-Var] is a λ-expression with v as its parameter and e as its body.

$$\frac{v = \xi(\overline{V_n}) \quad \Gamma, v : T_1; T_2 \Rightarrow_e e}{\Gamma; T_1 \to T_2 \Rightarrow_e \lambda v.e} \text{ [G-Abs]}$$

Application generation. The last generation rule [G-App] produces a function application. First it generates a type T_1 from the list of valid types $\mathbb{T}$. Having this type, it generates an abstraction expression e_1 with type $T_1 \to T_2$, and an expression e_2 with type T_1. The function application is represented by the application of the expression e_2 with the function expression e_1.

$$\frac{T_1 = \xi(\mathbb{T}) \quad \Gamma; T_1 \to T_2 \Rightarrow_e e_1 \quad \Gamma; T_1 \Rightarrow_e e_2}{\Gamma; T_2 \Rightarrow_e e_1\ e_2} \text{ [G-App]}$$

By using these four generation rules, one can generate well-typed λ-terms with respect to the presented STLC typing rules.

4.1.2 Soundness of Expressions Generation

We prove next that the presented type-directed algorithm is sound, i.e., it produces only well-typed expressions, according to STLC rules.

Theorem 6 (Soundness of expression generation)**.** *For all Γ and T, if $\Gamma; T \Rightarrow_e e$, then e is well-typed.*

Proof. The proof proceeds by induction on the derivation of $\Gamma; T \Rightarrow_e e$ with case analysis on the last rule used to deduce $\Gamma; T \Rightarrow_e e$.

Case (G-True): By rule G-True, $e = true$ and conclusion follows directly by rule T-True.

Case (G-False): Similar.

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

Case (G-Abs): Then, $e = \lambda v.e_0$ for some v and e_0; By rule G-Abs, we know that $\Gamma, v : T_1; T_2 \Rightarrow_e e_0$, for some T_1 and T_2; By the induction hypothesis we have that $\Gamma, v : T_1 \vdash e_0 : T_2$, and the conclusion follows by the rule T-Abs.

Case (G-App): Then, $e = e_1\ e_2$ for some e_1 and e_2; By rule G-App, we know that $T_1 = \xi(\mathbb{T})$, $\Gamma; T_1 \to T_2 \Rightarrow_e e_1$, and $\Gamma; T_2 \Rightarrow_e e_2$ for some T_1 and T_2; By the induction hypothesis we have that $\Gamma \vdash e_1 : T_1 \to T_2$, and $\Gamma \vdash e_2 : T_1$, thus the conclusion follows by the rule T-App. □

4.1.3 QuickChecking Semantic Properties

This thesis aims to explore techniques to check properties of programming languages. In order to do that for STLC, we implemented an interpreter following the se-

mantic rules presented earlier, and a test suite[1] which implements the type-directed algorithm to generate random programs using the QuickCheck tool (CLAESSEN; HUGHES, 2000). We combine both to test if this hand-written interpreter is sound with respect to STLC typing rules by testing the usual theorems of progress and preservation, similarly to what we have done for the expression language in Chapter 2.

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. We are aware that testing can only result in disproving the property, by finding a counter-example or leaving its validity undecided. However, if a counter-example is found, it can be used in order to help to fix implementation bugs. Considering that, we start modeling the desired properties. The first is the *progress* property.

```
prop_progress :: Expr → Bool
prop_progress e = isVal e ∨ maybe (False) (const True) (step e)
```

This property is encoded exactly the same way presented before. Function prop_progress receives an expression Expr as parameter (which is automatically generated by QuickCheck), and performs a reduction using the function step when the generated expression is not a value. We use the function isVal to check this fact. If the tested property holds, the result is True, and False otherwise.

Then, we encode the *preservation* property.

```
prop_preservation :: Expr → Bool
prop_preservation e =
  isVal e ∨
  case (typeof Data.Map.empty e) of
    Just t → case (step e) of
      Just e′ → case (typeof Data.Map.empty e′) of
        Just t′ → t ≡ t′
        _ → False
      _ → False
    _ → False
```

Similarly, the function prop_preservation receives an expression Expr as parameter. It also checks if expression e is a value, because a value cannot take a reduction step. Considering that e is not a value, we use the function typeof to obtain the type of expression e before evaluation. The reader can note the use of an extra parameter Data.Map.empty, which represents an empty Γ context. Afterwards, the function performs a reduction step on e, having as result an e′. Then, we use again the function

[1] The source-code of our interpreter and the test suite is available online (FEITOSA, 2019).

typeof on expression e′, to check if the types are preserved after a reduction step, i.e., the type t of e is the same of type t′ of e′.

4.1.4 Measuring the Quality of Tests

After running thousands of well-succeeded tests for both presented properties, we gain confidence that our interpreter and test suite is working properly. Indeed, during our tests, we found some small bugs in our implementation, which were fixed and quickly re-checked by running the test suite again. Then, as seen in the background chapter, we apply the HPC tool to measure code coverage of our interpreter. The summary of the results produced by HPC is presented in Figure 22.

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module STLC	71%	5/7	95%	19/20	98%	95/96
Program Coverage Total	71%	5/7	95%	19/20	98%	95/96

Figure 22 – Test coverage results for STLC.

As the interpreter of STLC is encoded in a unique module, we have just one line showing the coverage results for “Top level definitions” (71%), “Alternatives” (95%), and “Expressions” (98%). Then, we used the detailed coverage description to see which part of code was not reached by the performed tests. Figure 23 shows a result of HPC with highlighted code (using the yellow color) for unreachable code.

```
data Ty = TBool
        | TLam Ty Ty
        deriving (Show, Eq)
```

Figure 23 – Unreachable code on STLC definition of types.

This result is obviously correct, since we are not concerned in showing any information to the user (i.e., there is no instance to Show) when running our tests of properties. Figure 24 shows a piece of code highlighted using the green color, which means that whenever the interpreter reaches the highlighted code, the result is always True. Here again, the result is obvious, because the otherwise keyword will always produce such result.

```
step :: Expr -> Maybe Expr
step (App (Lam x t e1) e2)
    | isVal e2   = subs x e2 e1
    | otherwise  = do e2' <- step e2
                      return (App (Lam x t e1) e2')
step (App e1 e2) = do e1' <- step e1
                      return (App e1' e2)
```

Figure 24 – Unreachable code on STLC evaluation.

The last result we show for STLC is related to its type-checker. Figure 25 shows the complete implementation of function typeof which encodes the static semantics of this

calculus. By observing this results, we have two distinct parts highlighted with different colors. The first part appears in green, stating that the test $\texttt{t1} \equiv \texttt{t1}'$ is always true, and as result the line in yellow is never reached. Indeed, this is the expected result for this code, since we are generating only well-typed expressions, which means that we should never return an error during type-checking.

```
typeof :: Env -> Expr -> Maybe Ty
typeof ctx BTrue = Just TBool
typeof ctx BFalse = Just TBool
typeof ctx (Var x) = Data.Map.lookup x ctx
typeof ctx (Lam x t e) = do t1 <- typeof (Data.Map.insert x t ctx) e
                            return (TLam t t1)
typeof ctx (App e1 e2) = do (TLam t1 t2) <- typeof ctx e1
                            t1' <- typeof ctx e2
                            if (t1 == t1') then
                              return t2
                            else
                              Nothing
```

Figure 25 – Unreachable code on STLC type-checker.

The presented statistics provided by HPC are very useful to improve the quality of tests, since we have information about each individual line of code performed when running the test suite. Specifically for the presented interpreter, we can say that although not covering all the source-code, we have good test cases, because code not reached by the interpreter would only be reached by ill-typed programs, i.e., error handling code.

4.2 Featherweight Java

To generate random programs in the context of FJ, we follow two distinct phases, expression and class generation, generalizing the approach presented in the last section, considering that FJ has a nominal type system instead of a structural one. Similarly to STLC, we specified a generation rule for each typing rule, both for expression generation and class table generation. This section summarizes the results of the papers “Generating Random Well-Typed Featherweight Java Programs Using QuickCheck” (FEITOSA; RIBEIRO; DU BOIS, 2019), and “A Type-Directed Algorithm to Generate Well-Typed Featherweight Java Programs” (FEITOSA; RIBEIRO; DU BOIS, 2018c).

4.2.1 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. An empty sequence is denoted by $\bullet$, and the length of a sequence $\bar{\texttt{x}}$ is written $\#\bar{\texttt{x}}$. The generation algorithm also uses the function $\xi : [a] \rightarrow a$, which

returns a random element from an input list. Similarly the previous section, we use set operations on lists (sequences) and its meaning is as usual.

The expression generation is represented by the following judgment:

$$\mathsf{CT} \; ; \; \Gamma \; ; \; \mathsf{C} \Rightarrow_e \mathsf{e}$$

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.

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

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 $\overline{C_c}$:

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

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

$$\begin{gathered} \mathsf{C}' = \xi(\overline{C_c}) \\ \mathsf{CT} \; ; \; \Gamma \; ; \; \mathsf{C}' \Rightarrow_e \mathsf{e} \end{gathered}$$

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).

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

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

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

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 $\overline{C_c}$.

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

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

$$\text{C}' = \xi(\overline{C_c})$$
$$\text{CT} \; ; \; \Gamma \; ; \; \text{C}' \Rightarrow_e \text{e}_0$$

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.

$$\overline{M_c} = \{(\text{m}, \bar{\text{D}} \rightarrow \text{C}) \mid \exists \; \text{m}.\; \textit{mtype}(\text{m} \; , \; \text{C}') = \bar{\text{D}} \rightarrow \text{C}\}$$
$$(\text{m}', \bar{\text{D}}' \rightarrow \text{C}) = \xi(\overline{M_c})$$

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.

$$\textit{subtypes}(\text{CT}, \text{Object}) = \{\text{Object}\}$$
$$\textit{subtypes}(\text{CT}, \text{C}) = \{\text{C}\} \cup \textit{subtypes}(\text{CT}, \text{D}), \text{ if class C extends D} \in \text{CT}$$

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

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

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

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

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.

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

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

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

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 (IGARASHI;

PIERCE; WADLER, 2001), we defined the generation process in the rules G-DCast and G-SCast, since they can be used to build inner sub-expressions.

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

supertypes(CT, Object) = •
supertypes(CT, C) = {D} ∪ *supertypes*(CT, D), if class C extends D ∈ CT

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

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

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.

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

Considering the presented generation rules, we are able to produce well-typed expressions for each FJ's definitions.

4.2.2 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. We let $\varphi : L \to C$ denote a function that returns a class name C from a given class declaration L. The inclusion of an item x in a sequence $\overline{X}$ is denoted by $x : \overline{X}$, following Haskell's notation for lists. 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 (attributes or methods) 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:

$$\mathsf{CT}\ ;\ \mathsf{n}\ ;\ \mathsf{m}\ ;\ \mathsf{p} \Rightarrow_{ct} \mathsf{CT}'$$

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:

$$\frac{}{\mathsf{CT}\ ;\ 0\ ;\ \mathsf{m}\ ;\ \mathsf{p} \Rightarrow_{ct} \mathsf{CT}}\ \text{[CT-Base]}$$

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

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 φ(L) L : CT; n ; m ; p $\Rightarrow_{ct}$ CT′. The following judgment presents how classes are generated:

$$\text{CT ; m ; p} \Rightarrow_c \text{L}$$

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:

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

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 any of the set formed by the domain of current class table CT and Object:

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

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 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.

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

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

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.

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

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

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 C_0.

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

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 produced variables are distinct.

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

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

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

$$\mathit{CT} \text{ ; C ; D} \Rightarrow_k \text{K}$$

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

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

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

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

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

4.2.3 Soundness of Program Generation

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

Lemma 6 (Soundness of expression generation)**.** *Let* CT *be a well-formed class table. For all* Γ *and C* $\in$ *dom*(CT)*, if* CT *;* Γ *; C* $\Rightarrow_e$ *e then exists D, such that* $\Gamma \vdash$ *e : D and D* $<:$ *C.*

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

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

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

Lemma 7 (Soundness of subtypes)**.** *Let* CT *be a well-formed class table and C* $\in$ *dom*(CT)*. For all D. if D* $\in$ *subtypes*(CT*,C*) *then C* $<:$ *D.*

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

Lemma 8 (Soundness of method generation)**.** *Let* CT *be a well-formed class table and C* $\in$ *dom*(CT) $\cup$ {Object}*. For all p and m, if* CT *; C ; p ; m* $\Rightarrow_m$ C_0 m ($\bar{\mathsf{C}}$ $\bar{\mathsf{x}}$) { return e; } *then* C_0 m ($\bar{\mathsf{C}}$ $\bar{\mathsf{x}}$) { return e; } *OK in C.*

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

- $\bar{C} \subseteq$ *dom*(CT)
- Γ = {$\bar{C}$ $\bar{x}$, `this` : C}
- $C_0 = \xi$(*dom*(CT) $\cup$ {Object})
- $\bar{D}$ = *subtypes*(CT,C_0)
- *CT* ; Γ ; $E_0 \Rightarrow_e$ e
- $E_0 = \xi(\bar{D})$

By Lemma 7, we have that for all D $\in \bar{D}$, $C_0 <:$ D.
By Lemma 6, 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. □

Lemma 9 (Soundness of class generation)**.** *Let* CT *be a well-formed class table. For all m, p, if* CT *; m ; p* $\Rightarrow_c$ *CD then CD OK.*

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

- CD = class C extends D { $\bar{C}$ $\bar{f}$; K $\bar{M}$ }
- C = $\xi(\overline{C_n}$ - (*dom*(CT) $\cup$ Object))
- D = ξ(*dom*(CT) $\cup$ Object)
- fn = $\xi([1..m])$
- mn = $\xi([1..(m - fn)])$
- *CT′* = C (class C extends D {}) : *CT*
- *CT′* ; fn $\Rightarrow_{fs}$ $\bar{C}$ $\bar{f}$
- *CT″* = C (class C extends D { $\bar{C}$ $\bar{f}$; }) : *CT*
- *CT″* ; C ; mn ; p ; • $\Rightarrow_{ms}$ $\bar{M}$
- *CT′* ; C ; D $\Rightarrow_k$ K

By Lemma 8, we have that for all m. m $\in \bar{M}$, m OK.
By rule G-Constr we have that K = C ($\bar{D}$ $\bar{g}$, $\bar{C}$ $\bar{f}$) {super($\bar{g}$); this.$\bar{f}$ = $\bar{f}$;}, where $\bar{D}$ $\bar{g}$ = fields(D).
The conclusion follows by rule *class typing*. □

Lemma 10. *Let* CT *be a well-formed class table. For all n, m and p, if* CT *; n ; m ; p* $\Rightarrow_{ct}$ 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′, where CT ; n ; m ; p $\Rightarrow_{ct}$ CT_1. Let L be a class such that CT ; m ; p $\Rightarrow_c$ L. By Lemma 9, we have L OK in CT. By the induction hypothesis on φ(L) L : CT ; n ; m ; p $\Rightarrow_{ct}$ CT′ and rule CT-Step we have the desired conclusion. □

Lemma 11 (Soundness of class table generation)**.** *Let* CT *be a well-formed class table. For all n, m and p, if* CT *; n ; m ; p* $\Rightarrow_{ct}$ 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 $\Rightarrow_c$ L
- φ(L) L : CT ; n ; m ; p $\Rightarrow_{ct}$ CT′

By Lemma 9, we have that L *OK*. By the induction hypothesis we have that CT′ is a well-formed class table. By Lemma 10, we have that subtyping in CT′ is antisymmetric. Conclusion follows by the definition of a well-formed class table. □

Theorem 7 (Soundness of program generation)**.** *For all n, m and p, if* • *; n ; m ; p* $\Rightarrow_{ct}$ CT *then:*

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

Proof. Corollary of Lemmas 9, 10 and 11. □

4.2.4 QuickChecking Semantic Properties

As a proof of concept we have implemented an interpreter following the semantics of FJ, and we used random generated programs to check this interpreter against some properties, including those for type-soundness. The properties were specified and tested using QuickCheck (CLAESSEN; HUGHES, 2000).

Considering that testing requires additional programming, there is a natural risk that the testing code itself contain bugs (MIDTGAARD et al., 2017). 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 checked 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.

We started defining a function to check if generated class tables are well-formed, as follows:

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

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

We also defined 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 (class or interface). 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) $
    λ ct → forAll (genType ct) $
    λ t → forAll (genExpression ct Data.Map.empty t) $
    λ e → either (const False)
      (λ (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 (IGARASHI; PIERCE; WADLER, 2001).

```
prop_gencastsafeexpr :: Bool
prop_gencastsafeexpr =
  forAll (genClassTable) $
    λ ct → forAll (genType ct) $
    λ t → forAll (genExpression ct Data.Map.empty t) $
    λ 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) property is presented by Theorem 2.4.1 (p. 406 of (IGARASHI; PIERCE; WADLER, 2001)), stating that "*If* $\Gamma \vdash$ e: T *and* e $\rightarrow$ e′*, then* $\Gamma \vdash$ e′: T′ f*or some* T′ $<:$ T.". Our function was modeled as follows:

```
prop_preservation :: Bool
prop_preservation =
forAll (genClassTable) $
  λ ct → forAll (genType ct) $
  λ t → forAll (genExpression ct Data.Map.empty t) $
  λ e → either (const False)
    (λ (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 (IGARASHI; PIERCE; WADLER, 2001), which states that a well-typed expression does not get stuck.

```
prop_progress :: Bool
prop_progress =
  forAll (genClassTable) $
    λ ct → forAll (genType ct) $
    λ t → forAll (genExpression ct Data.Map.empty t) $
    λ 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′.

4.2.5 Measuring the Quality of Tests

We also ran many thousands of tests for the presented functions, all of them with success. Then, similarly to the previous section, we measured the quality of our tests with HPC by checking how much of the 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 26.

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module FJInterpreter	100%	2/2	82%	19/23	91%	166/182
module FJTypeChecker	100%	3/3	52%	22/42	68%	163/237
module FJUtils	100%	6/6	77%	27/35	91%	98/107
Program Coverage Total	100%	11/11	68%	68/100	81%	427/526

Figure 26 – Test coverage results for FJ.

In Figure 27 we present 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
  else -- RC-Field
    maybe (Nothing)
          (\e' -> Just (FieldAccess e' f))
          (eval' ct e)
```

Figure 27 – Unreachable code on FJ evaluation.

There we can note that to reach the highlighted code it is necessary: (1) the field f is 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 an ill-typed expression. As stated on type soundness proofs (IGARASHI; PIERCE; WADLER, 2001), a well-typed expression does not get stuck.

Similarly, Figure 28 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)
    e -> e -- Error: Expression type not found
```

Figure 28 – Unreachable code on FJ 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 sub-expression `e` could not be obtained. In all situations, we have an ill-typed program.

Finally, Figure 29 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 an ill-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
```

Figure 29 – Unreachable code on FJ auxiliary functions.

Although not having 100% of code coverage, our test suite was capable to verify the main safety properties presented 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 ill-typed.

After testing our generation procedure against our hand-written interpreter, we compile the generated ASTs of FJ to regular Java code, and 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) $
  λ ct → forAll (genType ct) $
  λ t → forAll (genExpression ct Data.Map.empty t) $
  λ e → monadicIO $
    do f ← run (writeFile "program.java" (formatJavaProgram ct e))
      (ex, out, err) ← run (readProcessWithExitCode "javac" ["programa.java"] "")
      assert (ex ≡ ExitSuccess)
```

By using this function, while developing our code generator we were able to find two problems in our implementation 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 we 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, and after that 100% of the test cases were successfully performed.

4.3 Semantics of Java 8 features in Featherweight Java

In order to explore in depth the property-based testing approach regarding programming language properties, we extended Featherweight Java with new features. We defined the formal semantics of λ-*expressions* and *default methods* which were added in the kernel of Java in the release of Java Development Kit version 8 (JDK 8). By using λ-expressions a programmer can treat functions as a method argument, or code as data in Java in a similar way functional languages work, offering a programming model that fuses the object-oriented and functional paradigms. The text presented in this section is based upon our paper "Property-based Testing the Semantics of Java 8 Features in Featherweight Java" (FEITOSA; RIBEIRO; DU BOIS, 2018b).

4.3.1 Extended Definitions

The abstract syntax of FJ augmented with interfaces, λ-expressions and default methods is given in Figure 30, where T represents type declarations, L and P express classes and interfaces, K represents constructors (presented earlier), S stands for signatures, M for methods, and e refers to the possible expressions. We use the meta-variables C, D, and E to represent class names, I, and J to represent interface names, and T, U, and V to represent generic names for classes or interfaces. Other definitions are similar to those presented earlier in this chapter.

$T ::=$		type definitions
	$C \mid I$	
$L ::=$		class declarations
	$class\ C\ extends\ C\ implements\ \overline{I}\ \{\overline{T}\ \overline{f}; K\ \overline{M}\}$	
$P ::=$		interface declarations
	$interface\ I\ extends\ \overline{I}\ \{\overline{S};\ default\ \overline{M}\}$	
$S ::=$		signature declarations
	$T\ m(\overline{T}\ \overline{x})$	
$M ::=$		method declarations
	$S\ \{\ return\ e;\}$	
$e ::=$		expressions:
	$\ldots$	original FJ constructors
	$(\overline{T}\ \overline{x}) \rightarrow e$	λ-expression

Figure 30 – Syntactic definitions for the extended FJ.

The differences from original FJ (IGARASHI; PIERCE; WADLER, 2001) were given firstly by the introduction of *interface declarations*, where $interface\ I\ extends\ \overline{I}\ \{\ \overline{S};\ default\ \overline{M}\ \}$ introduces an interface named I with a list of super-interfaces $\overline{I}$. The new interface defines a list of signatures $\overline{S}$ and a list of default methods $default\ \overline{M}$. For completeness, since Java's semantics allows a class to implement a list of interfaces, we changed the *class declarations* accordingly. Second, the *signature declarations* were added representing prototypes for abstract and concrete methods, where $T\ m(\overline{T}\ \overline{x})$ introduces a method named m, a return type T, and parameters $\overline{x}$ of types $\overline{T}$. As a consequence, the *method declarations* were also modified. Lastly, we added the constructor for *λ-expressions*, where $(\overline{T}\ \overline{x}) \rightarrow e$ represents an anonymous function, which has a list of arguments with type $\overline{T}$ and names $\overline{x}$, and a body expressions e.

The class table was also modified to accept both class and interface declarations. Therefore, a class table *CT* is a mapping from class or interface names, to class or interface declarations, L or P respectively. It still should satisfy some conditions, such as each type T should be in *CT*, except $Object$, which is a special class; and there

are no cycles in the sub-typing relation. Thereby, a program is a pair (CT, e) of a class table and an expression. For brevity, we omit from this text the details about auxiliary definitions (*fields*, *mtype*, *mbody*, λ*mark* and *abs-methods*), sub-typing, and the dynamic semantics, but these definitions can be found in (FEITOSA; RIBEIRO; DU BOIS, 2018b). We explain the meaning of these definitions as they appear in the sequence of this text.

Since our generation procedure relies on the typing rules, we show in Figure 31 the new typing rule T-Lam, which was added to type the λ-expression constructor. This rule first checks if I is a functional interface[2], using the function *abs-methods* which should bring information about its only method signature. Next, we use our λmark[3] definition to annotate the source-code with types for λ-expressions. Lastly, this rule verifies if the resulting type of the body is a sub-type of the return type of the method m.

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

Figure 31 – Typing λ-expressions in FJ.

We also had to add a rule to check if interfaces are well-formed, as shown in Figure 32. Roughly, this rule checks if all *default methods* are well-formed (using the rule presented in the original FJ paper (IGARASHI; PIERCE; WADLER, 2001)), and that there is at least one abstract method in such interface definition.

$$\frac{\bar{\text{M}}\ \text{OK in I} \quad \text{abs-methods(I)} \neq \bullet}{\text{interface I extends }\bar{\text{I}}\ \{\ \bar{\text{S}}\text{; default }\bar{\text{M}}\ \}\ \text{OK}}$$

Figure 32 – Interface typing in FJ.

Having defined the new rules, we proved in (FEITOSA; RIBEIRO; DU BOIS, 2018b) the subject reduction and progress to demonstrate type soundness for the Java 8 features of λ-expressions and default methods analogously to FJ. We kept the theorem names of the original FJ paper (IGARASHI; PIERCE; WADLER, 2001), presenting only the new cases for each of them.

Lemma 12 (Term substitution preserves typing)**.** *If* $\Gamma, \overline{x} : \overline{U} \vdash e : U$ *and* $\Gamma \vdash \overline{d} : \overline{U_0}$ *where* $\overline{U_0} <: \overline{U}$*, then* $\Gamma \vdash [\overline{x} \mapsto \overline{d}]e : T$ *for some* $T <: U$*.*

Proof. By induction on the derivation of $\Gamma, \overline{x} : \overline{U} \vdash e : U$ with case analysis on the typing rule used. It extends the original lemma for FJ (Lemma A.1.2 (IGARASHI; PIERCE; WADLER, 2001)) with the following case:

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

[3] A detailed explanation about this definition and its use can be found in our paper (FEITOSA; RIBEIRO; DU BOIS, 2018b)

Case (T-Lam). $e = (I)((\overline{T}\ \overline{x_0}) \to e_0)$, *abs-methods*$(I)$ = $\{T_0\ m(\overline{T}\ \overline{y})\}$, and $\Gamma, \overline{x} : \overline{U}, \overline{x_0} : \overline{T} \vdash e_0 : V$ where $V <: T_0$. By the induction hypothesis $\Gamma, \overline{x_0} : \overline{T} \vdash [\overline{x} \mapsto \overline{d}]e_0 : V$, finishing the case by applying the rule T-Lam. □

Theorem 8 (Subject reduction)**.** *If* $\Gamma \vdash e : T$ *and* $e \to e'$*, then* $\Gamma \vdash e' : T'$ *for some* $T' <: T$*.*

Proof. By induction on the reduction $e \to e'$, with a case analysis on the reduction rule used. It extends the original proof (IGARASHI; PIERCE; WADLER, 2001) of the corresponding theorem for FJ with the following cases:

Case (R-Default). $e = ((I)((\overline{T}\ \overline{x}) \to e_0)).m(\overline{u})$, *mbody*$(m, I) = (\overline{y}, e_1)$, and by rules T-Invk and T-Lam, we have: $\Gamma \vdash ((I)((\overline{T}\ \overline{x}) \to e_0)) : I$, *mtype*$(m, I) = \overline{U} \to U$, $\Gamma \vdash \overline{u} : \overline{T}$, and $\overline{T} <: \overline{U}$. Furthermore, $e' = [\overline{u} \mapsto \overline{y}]e_1$ and by the Lemma 12, $\Gamma \vdash e' = [\overline{u} \mapsto \overline{y}]e_1 : T$ for some $T <: U$.

Case (R-Lam). $e = ((I)((\overline{T}\ \overline{x}) \to e_0)).m(\overline{u})$, and by rules T-Invk and T-Lam, we have: $\Gamma \vdash ((I)((\overline{T}\ \overline{x}) \to e_0)) : I$, *mtype*$(m, I) = \overline{U} \to U$, $\Gamma \vdash \overline{u} : \overline{T}$, and $\overline{T} <: \overline{U}$. Furthermore, $e' = [\overline{u} \mapsto \overline{x}]e_0$ and by the Lemma 12, $\Gamma \vdash e' = [\overline{u} \mapsto \overline{x}]e_0 : T$ for some $T <: U$.

Case (R-Cast-Lam). $e = (T)((I)((\overline{T}\ \overline{x}) \to e))$, $I <: T$, and by rules T-UCast and T-Lam, we have: $\Gamma \vdash ((I)((\overline{T}\ \overline{x}) \to e)) : I$, and $(T)((I)((\overline{T}\ \overline{x}) \to e)) : T$. Furthermore, $e' = (I)((\overline{T}\ \overline{x}) \to e)$, finishing the case since $I <: T$. □

Theorem 9 (Progress)**.** *Suppose* e *is a well-typed expression.*

(1) If e *includes* $((I)((\overline{T}\ \overline{x}) \to e_0)).m(\overline{u})$ *as a sub-expression, and mbody*$(m, I) = (\overline{y}, e_1)$*, then* $\#\overline{y} = \#\overline{u}$ *for some* $\overline{y}$ *and* e_1*.*

(2) If e *includes* $((I)((\overline{T}\ \overline{x}) \to e_0)).m(\overline{u})$ *as a sub-expression, and mbody*(m, I) *is not defined, then* $\#\overline{x} = \#\overline{u}$*.*

Proof. The proof is based on the analysis of all well-typed expressions, extending previous proofs (IGARASHI; PIERCE; WADLER, 2001), which can be reduced to the above cases, to conclude that either it is in normal form or it can be further reduced to obtain a normal form. There are two possible normal forms. They are:

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

Theorem 10 (Type Soundness)**.** *If* $\varnothing \vdash e : T$ *and* $e \to^* e'$ *with* e' *being a normal form, then* e' *is a value* w *with* $\varnothing \vdash w : S$ *and* $\varnothing \vdash S <: T$*.*

Proof. Immediate from above theorems. □

4.3.2 Generation of Random Programs

Using the new typing rules presented above, we also extended our generation algorithm to generate programs including the constructors of Java 8. Some of the existent constructors of FJ were slightly changed to include these new features, and to avoid clutter we omit these small changes from this text. In this formalization, we let *classes*(CT) and *interfaces*(CT) denote the set of names in the domain of the finite mapping CT for classes and interfaces, respectively. Then, *dom*(CT) denotes *classes*(CT) $\cup$ *interfaces*(CT). Next we focus only in the brand new generation rules, including generation for λ-expressions and interfaces.

λ-expression generation. The type for a λ-expression is given by a functional interface I, which should have only one abstract method, and the generation rule is similar to those presented before. First, we need to get information about this abstract method. To accomplish this task, we use the auxiliary definition *abs-methods*[4], which returns the formal parameters and the return type of the expected λ-expression. After that, we generate the body expression by selecting a type from the sub-types of the return type T, and then we assemble the λ-expression with the formal parameters $(\overline{T}\ \overline{x})$ and the generated body expression e. This process is formalized by the G-Lam rule.

$$\frac{\text{abs-methods(I)} = \{\text{T m}(\bar{\text{T}}\ \bar{\text{x}})\} \qquad \text{CT ; } \bar{\text{x}}\text{: } \bar{\text{T}}, \Gamma \text{ ; } \xi(\text{subtypes(CT,T)}) \Rightarrow_e \text{e}}{\text{CT ; } \Gamma \text{ ; I} \rightarrow (\bar{\text{T}}\ \bar{\text{x}}) \Rightarrow_e \text{e}}\ \text{[G-Lam]}$$

Interface generation. Similarly to the rule G-Class presented earlier, in this extended calculus we have a specific judgment to generate interfaces:

$$\text{CT ; m ; p} \Rightarrow_i \text{I}$$

It generates a new interface, with at most *m* members (signatures or default methods), with at most *p* formal parameters for each signature or method, using as starting point a given class table. First, we create a new name which is not in the domain of the input class table, using:

$$\text{I} = \xi(\overline{T_n} \text{ - } (\textit{dom}(\text{CT}) \cup \{\text{Object}\}))$$

Since in our extended FJ, an interface (and also a class) can implement a list of interfaces, we define a recursive procedure to produce a list of valid interfaces on the class table (which can be empty). This judgment is also defined by recursion on n. Note that we maintain a set of already used names $\overline{U_n}$ to avoid duplicates.

[4]The description of all auxiliary definitions is given in our paper (FEITOSA; RIBEIRO; DU BOIS, 2018b)

$$\frac{}{\text{CT ; 0 ; } \overline{U_n} \Rightarrow_{is} \bullet} \text{ [G-Impl-Base]}$$

$$\frac{\begin{array}{c}\text{I} = \xi(\textit{interfaces}(\text{CT}) \text{ - } \overline{U_n}) \\ \text{CT ; n ; I : } \overline{U_n} \Rightarrow_{is} \bar{\text{I}}\end{array}}{\text{CT ; n + 1 ; } \overline{U_n} \Rightarrow_{is} \text{I : } \bar{\text{I}}} \text{ [G-Impl-Step]}$$

Signature generation. The judgment to generate signatures is similar to the one that generates methods presented in the last section. The only difference is that signatures do not have method bodies. Thus, the generation of a signature list also proceeds by recursion on m, using an auxiliary list to keep used names, as shown below. The rule G-Signs-Step uses a specific judgment to generate each signature, which is described by the rule G-Sign.

$$\frac{}{\text{CT ; I ; 0 ; p ; } \overline{U_n} \Rightarrow_{ss} \bullet} \text{ [G-Signs-Base]}$$

$$\frac{\begin{array}{c}\text{x} = \xi(\overline{V_n} \text{ - } \overline{U_n}) \\ \text{CT ; I ; p ; x} \Rightarrow_{s} \text{S} \\ \text{CT ; I ; m ; p ; x : } \overline{U_n} \Rightarrow_{ss} \bar{\text{S}}\end{array}}{\text{CT ; I ; m + 1 ; p ; } \overline{U_n} \Rightarrow_{ss} \text{S : } \bar{\text{S}}} \text{ [G-Signs-Step]}$$

Rule G-Sign reuse the judgment to generate at most p formal parameters (presented earlier). After that, it generates a type (class or interface) to represent the return of such method signature.

$$\frac{\begin{array}{c}\text{n} = \xi([0..(\text{p - 1})]) \\ \text{CT ; n ; } \bullet \Rightarrow_{ps} \bar{\text{T}}\ \bar{\text{x}} \\ \text{T}_0 = \xi(\textit{dom}(\text{CT}) \cup \{\text{Object}\})\end{array}}{\text{CT ; I ; p ; m} \Rightarrow_{s} \text{T}_0\ \text{m}\ (\bar{\text{T}}\ \bar{\text{x}})} \text{ [G-Sign]}$$

Rule G-Interface summarizes the process for generating an interface. The reader can note the we reuse the recursive judgment to generate methods for FJ in the last premise of G-Interface, which is responsible for generating the *default methods* for interfaces.

$$\frac{\begin{array}{c}\text{I} = \xi(\overline{T_n} \text{ - } (\textit{dom}(\text{CT}) \cup \{\text{Object}\})) \\ \text{in} = \xi([0..\#(\textit{interfaces}(\text{CT}))]) \\ \text{sn} = \xi([1..\text{m}]) \\ \text{mn} = \xi([1..(\text{m - sn})]) \\ \text{CT ; in ; } \bullet \Rightarrow_{is} \bar{\text{I}} \\ \text{CT ; I ; sn ; p ; } \bullet \Rightarrow_{ss} \bar{\text{S}} \\ \text{CT ; I ; mn ; p ; } \bullet \Rightarrow_{ms} \bar{\text{M}}\end{array}}{\text{CT ; m ; p} \Rightarrow_{i} (\text{interface I extends } \bar{\text{I}}\ \{\ \bar{\text{S}}\text{; default } \bar{\text{M}}\ \})} \text{ [G-Interface]}$$

Here we also extended the proofs presented in the last section showing that our generation algorithm keeps producing only well-formed class tables and well-typed expressions with Java 8 constructors. We show here only the new cases.

Lemma 13 (Soundness of expression generation)**.** *Let* CT *be a well-formed class table. For all* Γ *and T* $\in$ (*dom*(CT) $\cup$ {Object})*, if* CT *;* Γ *; T* $\Rightarrow_e$ *e then exists U, such that* $\Gamma \vdash$ *e : U and U* $<:$ *T.*

Proof. The proof proceeds by induction on the derivation of *CT* ; Γ ; T $\Rightarrow_e$ e doing a case analysis on the last rule used to deduce *CT* ; Γ ; T $\Rightarrow_e$ e. We show the new case of the proof.

Case (G-Lam): Then, e = ($\bar{\text{T}}$ $\bar{\text{x}}$) $\rightarrow$ e_0 for some e_0, $\bar{\text{x}}$, and $\bar{\text{T}}$; *CT* ; $\bar{\text{x}}$: $\bar{\text{T}}$, Γ ; ξ(subtypes(CT, T)) $\rightarrow_e$ e_0, for some T; there exists (T m ($\bar{\text{T}}$ $\bar{\text{x}}$)), such that *abs-methods*(I) = {T m ($\bar{\text{T}}$ $\bar{\text{x}}$)}. By the induction hypothesis, we have that: $\Gamma \vdash \text{e}_0$: U, U $<:$ T, and the conclusion follows by the rule T-Lam and the definition of the subtyping relation. □

Lemma 14 (Soundness of interface generation)**.** *Let* CT *be a well-formed class table. For all m, p, if* CT *; m ; p* $\Rightarrow_i$ *ID then ID OK.*

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

- ID = interface I extends $\bar{\text{I}}$ { $\bar{\text{S}}$; default $\bar{\text{M}}$ }
- I = $\xi(\overline{T_n}$ - (*dom*(CT) $\cup$ {Object}))
- in = ξ([0..#(*interfaces*(CT))])
- sn = $\xi([1..\text{m}])$
- mn = ξ([1..(m - sn)])
- *CT* ; in ; $\bullet \Rightarrow_{is} \bar{\text{I}}$
- *CT* ; I ; sn ; p ; $\bullet \Rightarrow_{ss} \bar{\text{S}}$
- *CT* ; I ; mn ; p ; $\bullet \Rightarrow_{ms} \bar{\text{M}}$

By Lemma 8, we have that for all m. m $\in \bar{\text{M}}$, m OK.
The conclusion follows by rule *interface typing*. □

Combining these lemmas with those defined for FJ, we prove that our generation procedure is sound with respect to the typing rules of FJ augmented with Java 8 features.

4.3.3 Checking Properties

We adapted the interpreter and test suite used in Section 4.2 to include the Java 8 features, and we use our test suite also as a lightweight manner to check the properties of *preservation* and *progress* for this calculus. Similarly, the informal proofs were also modeled as Haskell functions to be used with QuickCheck.

The preservation (subject reduction) property of FJ extended here, states that "*If* $\Gamma \vdash$ e: T *and* e $\rightarrow$ e′*, then* $\Gamma \vdash$ e′: T′ f*or some* T′ $<:$ T.". Our function was modeled as follows:

```
prop_preservation :: Bool
prop_preservation =
  forAll (genClassTable) $
    λ ct → let ct′ = ctTyping ct Data.Map.empty
               in forAll (genInstantiableType ct′) $
    λ t → forAll (genExpression True ct′ Data.Map.empty t) $
    λ e → case (typeof ct′ Data.Map.empty e) of
      Right (tp, e′) →
        either (const False)
          (λ (Type t′, _) → subtyping ct′ t′ t)
            (case (eval′ ct′ e′) of
              Just e″ →
                typeof ct′ Data.Map.empty e″
              _ → throwError (UnknownError e′)
            )
      _ → False
```

As can be seen in the code, after generating an instance for ct, an instantiable type t (which can be a class or a functional interface), 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, which states that a well-typed expression does not get stuck.

```
prop_progress :: Bool
prop_progress =
  forAll (genClassTable) $
    λ ct → forAll (genInstantiableType ct) $
    λ t → forAll (genExpression True ct Data.Map.empty t) $
    λ 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 performed for this extended calculus the same verifications of the previous section. First we ran many thousands of tests, checking and fixing bugs in our implementations (interpreter and test suite). Then, we verified the code coverage of our interpreter, as a way to improve our test cases. Lastly, we generated actual Java code to execute using the Java compiler. We believe that the material presented in this section can be useful to understand the basics of the new features of Java, and that our test suite can be useful to test different Java compilers, besides quick-checking of programming language properties.

4.4 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 (BLANCO; MILLER; MOMIGLIANO, 2017). 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 (PAŁKA et al., 2011). 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 (YANG et al., 2011) is a generator of programs for the C language, supporting a large number of language features, which was used to find a number of bugs in compilers such as GCC, LLVM, etc. Le et al. (LE; AFSHARI; SU, 2014) developed a methodology that uses differential testing for C compilers. Lindig (LINDIG, 2005) created a tool for testing the C function calling convention of the GCC compiler, which randomly generates types of functions. There are also efforts to generate test cases for other languages (DRIENYOVSZKY; HORPáCSI; THOMPSON, 2010). All of these projects rely on informal approaches, while ours is described formally and applied to property-based testing.

More specifically, Daniel et al. (DANIEL et al., 2007), Soares et al. (SOARES; GHEYI; MASSONI, 2013) and Mongiovi et al. (MONGIOVI et al., 2014) generate Java programs to test refactoring engines, some of them applied in Eclipse and NetBeans IDEs. Gligoric et al. (GLIGORIC et al., 2010) presented an approach for describing tests using non-deterministic test generation programs applying in the Java context. Klein et al. (KLEIN; FLATT; FINDLER, 2010) generated random programs to test an object-oriented library. Silva, Sampaio and Mota (SILVA; SAMPAIO; MOTA, 2015) used program generation to verify transformations in Java programs. Allwood and Eisen-

bach (ALLWOOD; EISENBACH, 2009) 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 closely related to ours since they are also generating code in the object-oriented context. The difference of our approach is that we formalize the generation procedure to randomly generate complete classes and expressions based on the type system of FJ, proving 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 Java semantics hold by using the generated programs.

The work of Palka, Claessen and Hughes (PAŁKA et al., 2011) 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, and we worked with different language settings, starting with STLC, and evolving the approach to a much bigger (and complex) language subset, as FJ extended with Java 8 features. We used as inspiration for our formalization of Java 8 features the works of Bellia and Occhiuto (BELLIA; OCCHIUTO, 2011) and Bettini et al. (BETTINI et al., 2018), where both used FJ as a basis to describe the semantics and type system of λ-expression in Java. The approach used in this thesis can be seen as a simplification of the model presented in (BETTINI et al., 2018). While they provided a semantics for typed and untyped λ-expressions, with an extra syntactic constructor to annotate types, we focused on typed λ-expressions using *casts* to do such type annotation. In (FEITOSA; RIBEIRO; DU BOIS, 2018b) we also describe a different approach for the semantics of λ-expressions: instead of annotating types at run-time, we use type-elaboration to type λ-expressions at compile time. Unlike the work of Bettini et al., besides the semantic specification, we performed property-based testing against hand-written interpreters (following the semantics) to check soundness properties for the proposed extension. While their work provides a proof of type soundness informally, in this thesis we also describe an intrinsically-typed formalization in Agda (see next Chapter).

4.5 Chapter's Final Remarks

In this chapter we presented the contributions of the first branch of this thesis regarding random program generation to be applied to check properties of programming languages. We discussed the subtleties of generating code from the typing rules for both STLC and FJ (and an extended version of FJ with Java 8 features) to produce well-typed programs, the encoding of properties as Haskell functions, as well as the use of a tool to improve the quality of tests. We used this approach as a lightweight form of verification, to gain confidence about the correctness of the tested properties before formally proving them in a proof assistant, as we shall see in the next chapter. We believe that this approach can also be used when working with different programming languages.

5 PROVING PROPERTIES OF LANGUAGES

In this chapter we develop the second main topic of this thesis, where we apply formal verification on the studied programming languages in a proof assistant. Formal verification is a heavyweight approach, which uses a mathematical framework to *prove* that properties are indeed valid with respect to the subsets we are working. We know that by using only tests, it is impossible to offer full guarantees, since tests can reach only a limited amount of cases. Mechanized proof assistants are powerful tools, but proof development can be difficult and time consuming (DELAWARE; COOK; BATORY, 2011). This fact creates a trade-off between testing and formal verification, and the project managers have to carefully select the approach according to their needs.

Nowadays, there are two main approaches to formalize and prove type safety for a programming language, which were briefly presented in Chapter 2. The most used method is the syntactic approach (sometimes called extrinsic) proposed by Wright and Felleisen (WRIGHT; FELLEISEN, 1994). Using this technique, the syntax is defined first, and then relations are defined to represent both the typing judgments (static semantics), and the evaluation through reduction steps (dynamic semantics). The common theorems of *progress* and *preservation* link the static and the dynamic semantics, guaranteeing that a well-typed term do not get stuck, i.e., it should be a value or be able to take another reduction step, preserving the intended type.

Another technique that is becoming increasingly popular in recent years, which uses a functional approach (sometimes called intrinsic) to achieve a similar result, was proposed by Altenkirch and Reus (ALTENKIRCH; REUS, 1999). The idea is to first encode the syntax and the typing judgments in a single definition which captures only well-typed expressions, usually within a total dependently-typed programming language. After that, one writes a definitional interpreter (REYNOLDS, 1972) which evaluates the well-typed expressions. By using this approach, type-soundness is guaranteed by construction, and the necessary lemmas and proofs of the syntactic approach can be obtained (almost) for free.

We apply these two techniques for each language we are studying, exploring different techniques to prove type soundness of programming languages, and showing that

it is possible to represent modern languages (such as FJ) with complex structure and binding mechanisms using both styles of formalization. We also provide a comparison between both formalization approaches with some criteria to help a programmer to choose between them if necessary. Using such criteria, we chose the intrinsically-typed approach to formalize FJ with λ-expressions, following the semantics presented in Section 4.3, showing that it is indeed possible to extend the proposed formalization.

5.1 Simply-Typed Lambda Calculus

This section is divided in two parts, where the first shows how to formalize STLC using the syntactic (extrinsic) approach with explicit proofs for the common theorems of *progress* and *preservation*. The second part uses the functional (intrinsic) approach, which uses dependent types and a definitional interpreter to achieve a similar soundness result. Although STLC is a simple language, it is useful to introduce an important concept, *de Bruijn* indices, which is applied broadly in the FJ formalization.

5.1.1 Extrinsic Formalization

This subsection follows the usual script for when extrinsically formalizing a programming language: first the syntax, semantics and typing rules, and then the properties of progress and preservation are proven. This text is highly based on what is presented on Wadler's (WADLER, 2018) and Pierce's (PIERCE et al., 2018) books for the formalization of STLC in Agda and Coq, respectively.

Syntax definition. As mentioned in Chapter 3, we extended the STLC with Boolean types. Thus, it has the three basic constructors of λ-calculus (Var, Lam, and App), and two constants representing the Boolean values true and false. In this definition, for simplicity, a variable name is represented as a natural number $\mathbb{N}$. The next code shows how expressions are defined in Agda.

```
data Expr : Set where
  true  : Expr
  false : Expr
  Var   : ℕ → Expr
  Lam   : ℕ → Expr → Expr
  App   : Expr → Expr → Expr
```

For example, the identity function is represented as Lam 1 (Var 1). With this syntax one can create ill-scoped terms like Lam 1 (Var 2), but the typing relation will forbid this.

Values. In our representation, besides the constants true and false being values, we also consider an *abstraction* (λ-expression) a value, no matter whether the body expression is a value or not, i.e., reduction stops at abstractions. The reduction of a

λ-expression only begins when a function is actually applied to an argument (PIERCE et al., 2018).

```
data Val : Expr → Set where
  V-True : Val true
  V-False : Val false
  V-Lam : ∀ {x e} → Val (Lam x e)
```

The inductive definition Val is indexed by an Expr, showing which syntactical constructor the value represents. The definition of values will be used next, during the formalization of the reduction steps.

Dynamic semantics. The presented formalization considers the call-by-value evaluation strategy. The only reducible expression is the application App of a λ-expression (represented by the constructor Lam) to a value. In the reduction relation _ ⟶ _ defined below, the rule R-App$_1$ represents one step of evaluation to reduce the left-hand side of an application. This rule should be applied until the expression e_1 becomes a value. Rule R-App$_2$ is used when the left-hand side is already a value, and reduces the right-hand side of an application. Again, it should be applied until e_2 becomes a value. After that, the rule R-App should be applied when both (left and right) expressions are values, and the left one is a Lam. With this setting, it applies a substitution (using function subs[1]) of the actual parameter v_1 where the formal parameter x appears in the body expression e.

```
subs : Expr → ℕ → Expr → Expr
-- Implementation omitted
data _ ⟶ _ : Expr → Expr → Set where
  R-App₁ : ∀ {e₁ e₂ e₁′}
    → e₁ ⟶ e₁′
    → App e₁ e₂ ⟶ App e₁′ e₂
  R-App₂ : ∀ {v₁ e₂ e₂′}
    → Val v₁
    → e₂ ⟶ e₂′
    → App v₁ e₂ ⟶ App v₁ e₂′
  R-App : ∀ {x e v₁}
    → Val v₁
    → App (Lam x e) v₁ ⟶ (subs e x v₁)
```

Syntax of types. In the presented formalization, there are only two types: bool represents Booleans, and _ ⟹ _ represents a *function type*. The function type represents

[1] We omit the implementation of function subs, however it is available online (FEITOSA, 2019).

the type for a λ-expression using two arguments. The first represents the expected parameter type, and the second represents the return type of a given λ-expression.

```
data Ty : Set where
  bool : Ty
  _⟹_ : Ty → Ty → Ty
```

Expression typing. The `_⊢_:_` relation encodes the typing rules of STLC: considering a context `Ctx`, an expression `Expr` has type `Ty`. A context `Ctx` is defined as a list of pairs `List (ℕ × Ty)`, linking each variable with its given type. The first two rules are simple: constants `true` and `false` have always type `bool` no matter what is contained in a context `Γ`. Rule `T-Var` uses an auxiliary definition for context lookup `_∋_:_`[2] which binds a type τ for a variable `x` according to a context `Γ`. Rule `T-Lam` uses our typing judgment to type the right-hand side of a λ-expression with a context `Γ` extended with the formal parameter (left-hand side) of that λ-expression. We use the list concatenation `(x , τ₁) :: Γ` to extend the context, and the result is a function type `τ₁ ⟹ τ₂`. The last rule `T-App` guarantees the correct type for expressions e_1 and e_2 to perform an application. The first (e_1) should have a function type `(τ₁ ⟹ τ₂)`, and the second (e_2) should be of type τ_1, ensuring that the formal and actual parameters are of the same type. If both premises hold, the typing judgment results in τ_2, which is the return type of the λ-expression.

```
data _⊢_:_ : Ctx → Expr → Ty → Set where
  T-True : ∀ {Γ}
    → Γ ⊢ true : bool
  T-False : ∀ {Γ}
    → Γ ⊢ false : bool
  T-Var : ∀ {Γ x τ}
    → Γ ∋ x : τ
    → Γ ⊢ (Var x) : τ
  T-Lam : ∀ {Γ x e τ₁ τ₂}
    → ((x , τ₁) :: Γ) ⊢ e : τ₂
    → Γ ⊢ (Lam x e) : (τ₁ ⟹ τ₂)
  T-App : ∀ {Γ e₁ e₂ τ₁ τ₂}
    → Γ ⊢ e₁ : (τ₁ ⟹ τ₂)
    → Γ ⊢ e₂ : τ₁
    → Γ ⊢ (App e₁ e₂) : τ₂
```

Soundness proofs. We formalized the common theorems of *progress* and *preservation*

[2] We also omit the definition of *context lookup*, but it can be found in our repository (FEITOSA, 2019).

for the presented calculus in Agda. Since STLC is a small calculus, the proofs are carried simply by induction in the structure of the typing judgment.

The preservation function represents the theorem with the same name, stating that if a well-typed expression e has type τ in an empty context $[\,]$, and it takes a reduction step $e \longrightarrow e'$, then e' remains with type τ. The only typing rule that matters is the one that applies one expression to another, with three cases, one for each reduction rule: the first applies the induction hypothesis to the left-side expression, the second is similar, but applies the induction hypothesis to the right-side, and the last one uses an auxiliary function subst representing the lemma which states that *Substitution Preserves Typing*[3]. All the other cases represent values (impossible cases), which cannot take any reduction step.

```
subst : [ ] ⊢ v : τ₁ → (x , τ₁) :: Γ ⊢ e : τ₂ → Γ ⊢ (subs e x v) : τ₂
-- Proof code omitted
preservation : ∀ {e e′ τ} → [ ] ⊢ e : τ → e ⟶ e′ → [ ] ⊢ e′ : τ
preservation T-True ()
preservation T-False ()
preservation (T-Var ())
preservation (T-Lam _) ()
preservation (T-App r₁ r₂) (R-App₁ s) = T-App (preservation r₁ s) r₂
preservation (T-App r₁ r₂) (R-App₂ v₁ s) = T-App r₁ (preservation r₂ s)
preservation (T-App (T-Lam r₁) r₂) (R-App v₁) = subst r₂ r₁
```

Similarly to the previous theorem, the progress function represents the theorem with the same name, stating that if a well-typed expression e has type τ in an empty context $[\,]$, then it can make *Progress*, i.e., or e is a value, or it can take another reduction step. First we define an inductive datatype to hold the result of our proof, with two constructors: Done when e is a value, and Step when e reduces to an e'.

```
data Progress (e : Expr) : Set where
  Step : ∀ {e′}
    → e ⟶ e′
    → Progress e
  Done : Val e
    → Progress e
```

We also need to establish a basic property of reduction and types, identifying the possible *canonical forms* (i.e., well-typed closed values) belonging to each

[3]For brevity, we present only the type of subst here, but the complete proof (and its related lemmas) can be found in our source-code repository (FEITOSA, 2019)

type (PIERCE et al., 2018). The definition has one constructor for each value (C-True, C-False, and C-Lam), and its proof linking each value with its respective type[4].

```
data Canonical : Expr → Ty → Set where
 -- Inductive definition code omitted
canonical : ∀ {v τ} → [ ] ⊢ v : τ → Val v → Canonical v τ
 -- Proof code omitted
```

The proof for *progress* is straightforward: cases with values are finished with Done and the respective Val constructor; and the case dealing with application is finished with Step, using the induction hypothesis for the left and right side, and the canonical form C-Lam, which relates the values with their types.

```
progress : ∀ {e τ} → [ ] ⊢ e : τ → Progress e
progress T-True = Done V-True
progress T-False = Done V-False
progress (T-Var ())
progress (T-Lam _) = Done V-Lam
progress (T-App e₁ e₂) with progress e₁
... | Step stp₁ = Step (R-App₁ stp₁)
... | Done v₁ with progress e₂
... | Step stp₂ = Step (R-App₂ v₁ stp₂)
... | Done v₂ with canonical e₁ v₁
... | C-Lam stp = Step (R-App v₂)
```

5.1.2 Intrinsic Formalization

This subsection introduces the intrinsically-typed formalization, and the implementation of a definitional interpreter for the STLC. This text should serve as basis to understand the basics of this approach, considering how we use dependent-types to encode the syntax and typing rules, and how we use *de Bruijn* indices (BRUIJN, 1972) to deal with name binding. Again, the concepts presented here will be used in a more complex environment when formalizing Featherweight Java.

Intrinsically-typed syntax. Representing the typing rules combined with the language syntax is a well-known approach (ALTENKIRCH; REUS, 1999; REYNOLDS, 2001). Using such approach, only well-typed expressions are accepted by the host language (Agda in our case), and ill-typed expressions are rejected by the compiler accusing a type error. Using this approach the defined abstract syntax trees (ASTs) not only

[4]We show only the type definition for Canonical and canonical. The complete source code is available online (FEITOSA, 2019).

capture the syntactic properties of a language but semantic properties as well. We highlight the importance of the approach here because it allows programmers to reason about their programs as they write them rather than separately at the meta-logical level.

Since we are embedding types together with the syntax, we need a Ty, which is the same presented in the previous subsection, and a type context Ctx, which is represented by a List Ty. Then, the expression definition is parameterized by a Γ of type Ctx, which binds a type for each free variable, and indexed by a Ty, which represents the type of a given expression. The following code show our Expr definition.

```
data Expr (Γ : Ctx) : Ty → Set where
  true  : Expr Γ bool
  false : Expr Γ bool
  Var   : ∀ {x} → x ∈ Γ → Expr Γ x
  Lam   : ∀ σ {τ} → Expr (σ :: Γ) τ → Expr Γ (σ ⟹ τ)
  App   : ∀ {σ τ} → Expr Γ (σ ⟹ τ) → Expr Γ σ → Expr Γ τ
```

The first two constructors represent the constants true and false, both with type bool. The constructor Var shows us a different way to represent variables. All name bindings are done with statically-checked *de Bruijn* indices (BRUIJN, 1972), a technique for handling binding by using a nameless, position-dependent name scheme. Thus, with this constructor, we do not use a name to identity a variable, but a well-typed *de Bruijn* index $x \in \Gamma$ which witnesses the existence of an element x in Γ, as defined by the standard library $\_ \in \_$ operator. The result type of this expression should be the one represented by the x variable type. This technique is well known for avoiding out-of-scope errors. The Lam constructor expects a σ of type Ty, representing the formal parameter of a λ-expression, and an expression Expr $(\sigma :: \Gamma)\ \tau$, representing its body. Here we note that the expected expression has type τ considering an extended context $\sigma :: \Gamma$. Then, the resulting type for this expression is of a function type $\sigma \Longrightarrow \tau$. The last constructor App expects two expressions, the first with a function type $\sigma \Longrightarrow \tau$, and the second with a type σ, which has the same type of the formal parameter of the first expression. If both expressions respect the premises, an expression of type τ is constructed.

Values and Environments. To define the STLC interpreter, we need intrinsically-typed values and environments. First we define a Value indexed by a type Ty. By using this definition, when interpreting the code, we are able to convert the result to an Agda type (host language semantics). So, if the value represents a bool, it results in the Agda's Bool type. In the case of a function type, it results in an Agda's function type.

```
Value : Ty → Set
Value bool  =  Bool
Value (σ ⟹ τ)  =  Value (σ) → Value (τ)
```

When working with such intrinsically-typed definition, an environment holds the information for which Value is associated with each variable in the context Γ. The representation of this environment is not totally obvious, since variables can have different types (AUGUSTSSON; CARLSSON, 1999). We encoded it by using the datatype All, where each type in the context is linked with a typed value, as we can see next.

```
Env : Ctx → Set
Env Γ = All Value Γ
```

Definitional interpreter. We present next a fairly standard definition of an interpreter for STLC in Agda. The interpreter has three main points: processing of primitive values, variable lookup, and performing the actual evaluation of a λ-expression. The function eval pattern matches on the given Expr, dealing with each case, as follows.

```
eval : ∀ {Γ τ} → Env Γ → Expr Γ τ → Value τ
eval env true = true
eval env false = false
eval env (Var x) = lookup env x
eval env (Lam σ e) = λ x → (eval (x :: env) e)
eval env (App e e₁) = eval env e (eval env e₁)
```

The cases for true and false are simple. The case for Var uses the standard library's function lookup to project the appropriate value from the run-time environment env. The typed *de Bruijn* index guarantees that the value projected from the environment has the type demanded since the environment is typed by the context Γ, which allows us to look up values of a particular type x in an environment Env Γ using the witness x ∈ Γ.

The case for Lam is tricky. It converts our definition to a λ-expression in Agda. The case for App evaluates the first expression e, which results in a λ-expression, and applies to it the result of evaluating the second expression e_1. One can note that the actual evaluation of expressions is delegated to Agda, so it is not necessary to define substitution as in the extrinsic formalization.

There are two points to highlight with the presented approach. First, we can note that we do not have any error treatment in our interpreter. This is happening because we are working only with a (intrinsically) well-typed term, so in this case, every time a variable is requested it is guaranteed by construction that it exists with the correct type. The same happens when applying one expression to another. The well-typed syntax guarantees that the first expression is indeed a λ-expression (produced by our Lam case) and the second expression has the correct type.

Second, by allowing only the representation of well-typed expressions, the *preservation* property is also assured by construction, and by writing such evaluator in a total language like Agda, the *progress* property is guaranteed by consequence.

5.1.3 Elaborating STLC Extrinsic to Intrinsic

We show, by using the elab function, that we can elaborate an extrinsic well-typed expression (using our extrinsic typing predicate $\Gamma \vdash e : \tau$) to an intrinsic expression, which is well-typed by construction. The first two cases are straightforward, returning the Boolean expression associated with each constructor. The case T-Var results in a intrinsic Var, and the function elab-var (omitted here) produces a *de Bruijn* index according to its position on the environment. The case for T-Lam results a Lam expression, elaborating its body recursively. Similarly, T-App results in an App expression, elaborating both (left and right) expressions recursively.

```
elab : ∀ {Γ e τ} → Γ ⊢ e : τ → Expr Γ τ
elab T-True = true
elab T-False = false
elab (T-Var x) = Var (elab-var x)
elab (T-Lam {x = x} p) = Lam x (elab p)
elab (T-App p p₁) = App (elab p) (elab p₁)
```

One can use this elaboration function to check that both formalizations (extrinsic and intrinsic) produce the same result, by running the interpreters for both approaches, or by proving it as theorem.

5.2 Featherweight Java

Similarly to STLC, we also split the formalization of FJ in two subsections, using the extrinsic and intrinsic approaches to achieve an equivalent result.

5.2.1 Extrinsic Formalization

This subsection presents our formalization of a large subset of FJ in Agda using the usual syntactic approach. As for STLC, this encoding was also divided in two major parts. First a set of definitions corresponding to the syntax, auxiliary functions, reduction and typing rules were created, followed by the main proofs for type soundness of the encoded language.

Syntax. The definition of FJ is more intricate than STLC. An expression can refer to information of two sources: (1) a context to deal with variables, similar to STLC; (2) a class table, which stores information about all classes in the source-code program. Besides, there is a mutual relation between classes and expressions: an expression can refer to information about classes, and a class can contain expressions (which represent method bodies).

Considering all this, we start our formalization in Agda by defining the syntactic elements regarding FJ. A Class is represented by a record with three fields. The class

name is stored in cname, the attributes are in flds, and the methods are in meths. For simplicity, we represent all names as natural numbers (Name = ℕ).

```
record Class : Set where
  field
    cname : Name
    flds  : List (Name × Name)
    meths : List (Name × Meth)
```

As one can see, attributes are represented by a List of tuples (Name × Name), encoding the name and the type for each field. For methods, we have a similar setting, however, we use a List of tuples (Name × Meth), where the first element is the method name, and the second encodes the method information, containing the return type ret, the method parameters params, and the method body body, as shown below:

```
record Meth : Set where
  field
    ret    : Name
    params : List (Name × Name)
    body   : Expr
```

An expression can be seen in two parts of a FJ program. It can appear in a method body, or it can represent the Java's main method, acting as a starting point for the program. It is represented using an inductive definition, considering the following constructors:

```
data Expr : Set where
  Var   : Name → Expr
  Field : Expr → Name → Expr
  Invk  : Expr → Name → List Expr → Expr
  New   : Name → List Expr → Expr
```

A variable is represented by the constructor Var, a field access is encoded by Field, a method invocation by Invk, and object instantiation is defined by New (IGARASHI; PIERCE; WADLER, 2001).

The only possible value in FJ is encoded in the Val definition. Since Java adopts a call-by-value evaluation strategy, to be a value, we need an object instantiation with all parameters being values themselves. This was encoded using the standard library's datatype All.

```
data Val : Expr → Set where
  V-New : ∀ {c cp} → All Val cp → Val (New c cp)
```

Auxiliary definitions. A FJ expression can refer to information present on the class table, where all classes of a given program are stored. To reason about information of a given class, two auxiliary definitions were defined. Using the definition fields one can refer to information about the attributes of a class.

```
data fields : Name → List (Name × Name) → Set where
  obj   : fields Obj []
  other : ∀ {c cd}
        → Δ ∋ c : cd
        → fields c (Class.flds cd)
```

Similarly to STLC, we also use a predicate to lookup in a given list of pairs $(\_ \ni \_ : \_)$[5]. In FJ we use this definition to lookup for classes, fields, methods, and variables.

By using the predicate method it is possible to refer to information about a specific method in a certain class. Both auxiliary definitions refer to information on a class table Δ, which is defined globally in the working module.

```
data method : Name → Name → Meth → Set where
  this : ∀ {c cd m mdecl}
    → Δ ∋ c : cd
    → (Class.meths cd) ∋ m : mdecl
    → method c m mdecl
```

Reduction rules. The reduction predicate takes two expressions as arguments. The predicate holds when expression e reduces to some expression e′. The evaluation relation is defined with the following type.

```
data _ ⟶ _ : Expr → Expr → Set
```

When encoding the reduction relation, two important definitions[6] were used: interl, which is an inductive definition to interleave the information of a list of pairs (List (Name × A)) with a List B, providing a new list List (Name × B); and subs, which is responsible to apply the substitution of a parameter list into a method body. We present only their types next.

```
data interl : List (Name × A) → List B → List (Name × B) → Set
 -- Inductive definition code omitted.
subs : Expr → List (Name × Expr) → Expr
 -- Function code omitted.
```

[5]We omit the code of $\_ \ni \_ : \_$ predicate, but it can be found in our repository (FEITOSA, 2019).
[6]Both definitions can be found online (FEITOSA, 2019).

From now on we explain each constructor of the evaluation relation separately to make it easier for the reader.

Constructor R-Field encodes the behavior when accessing a field of a given class. All fields of a class are obtained using fields C flds. We interleave the definition of fields flds with the list of expressions cp received as parameters for the object constructor by using interl flds cp fes. With this information, we use fes ∋ f : fi to bind the expression fi related to field f.

```
R-Field : ∀ {C cp flds f fi fes}
  → fields C flds
  → interl flds cp fes
  → fes ∋ f : fi
  → Field (New C cp) f ⟶ fi
```

Constructor R-Invk represents the encoding to reduce a method invocation. We use method C m MD to obtain the information about method m on class C. As in R-Field we interleave the information about the method parameters Meth.params MD with a list of expressions ap received as the actual parameters on the current method invocation. Then, we use the function subs to apply the substitution of parameters in the method body.

```
R-Invk : ∀ {C cp m MD ap ep}
  → method C m MD
  → interl (Meth.params MD) ap ep
  → Invk (New C cp) m ap ⟶ subs (Meth.body MD) ep
```

All the next constructors represent the congruence rules (call-by-value) of the FJ calculus. Reduction of the first expression e is done by RC-Field and RC-InvkRecv, producing an e′.

```
RC-Field : ∀ {e e′ f}
  → e ⟶ e′
  → Field e f ⟶ Field e′ f
RC-InvkRecv : ∀ {e e′ m mp}
  → e ⟶ e′
  → Invk e m mp ⟶ Invk e′ m mp
```

Reduction of arguments when invoking a method or instantiating an object is done by RC-InvkArg and RC-NewArg. An extra predicate (_ ⟼ _) is used to evaluate a list of expressions recursively.

```
RC-InvkArg : ∀ {e m mp mp′}
  → mp ⟼ mp′
  → Invk e m mp ⟶ Invk e m mp′
RC-NewArg : ∀ {C cp cp′}
  → cp ⟼ cp′
  → New C cp ⟶ New C cp′
```

Typing rules. The typing rules for FJ are divided in two main parts: there are two predicates to type an expression, and two predicates to check if classes and methods are well-formed. A FJ program is well-typed if all typing predicates hold for a given program.

To type an expression, the typing judgment predicate `_ ⊢ _ : _` which encodes the typing rules of FJ is used, and the predicate `_ ⊨ _ : _` responsible to apply the typing judgment `_ ⊢ _ : _` to a list of expressions recursively. Their type definitions are shown below:

```
data _ ⊢ _ : _  : Ctx → Expr → Name → Set
data _ ⊨ _ : _ : Ctx → List Expr → List Name → Set
```

Both definitions are similar, receiving three parameters each. The first parameter is a type context Ctx, similar to the one for λ-calculus, aiming to store the types for variables. The second is an Expr for the typing judgment, and a List Expr for the recursive case, both representing the expressions being typed. The last argument is a Name (or List Name) representing the types for the given expressions. Next we present each constructor for the `_ ⊢ _ : _` predicate.

The constructor T-Var is similar to the one presented for λ-calculus. A variable x has type C if this variable is present in a context Γ with its type.

```
T-Var : ∀ {Γ x C}
  → Γ ∋ x : C
  → Γ ⊢ (Var x) : C
```

Constructor T-Field is more elaborated. First, we use the typing judgment to obtain the type of the sub-expression e. Then, we use the auxiliary definition fields which gives us the attributes flds of a class C. Like variables, the type of f is obtained by the information stored in flds.

```
T-Field : ∀ {Γ C Ci e f flds}
  → Γ ⊢ e : C
  → fields C flds
  → flds ∋ f : Ci
  → Γ ⊢ (Field e f) : Ci
```

Constructor T-Invk also uses the typing judgment to obtain the type for the sub-expression e. After that, we use the auxiliary predicate method to obtain the definition of method m in class C. It is used to type-check the method parameters mp[7]. Considering that all the premises hold, the type of a method invocation is given by Meth.ret MD.

```
T-Invk : ∀ {Γ C e m MD mp}
  → Γ ⊢ e : C
  → method C m MD
  → Γ ⊨ mp : proj₂ (unzip (Meth.params MD))
  → Γ ⊢ (Invk e m mp) : (Meth.ret MD)
```

Similarly to T-Invk, in the definition T-New we also use the predicate to type a list of expressions. In this case, the premises will hold if the actual parameters cp of the class constructor are respecting the expected types for the fields of a given class C.

```
T-New : ∀ {Γ C cp flds}
  → fields C flds
  → Γ ⊨ cp : proj₂ (unzip flds)
  → Γ ⊢ (New C cp) : C
```

A class is well-formed if it respects the ClassOk predicate. In the presented definition, the All datatype is used to check if all methods are correctly typed.

```
data ClassOk : Class → Set where
  T-Class : ∀ {CD}
    → All (MethodOk CD) (proj₂ (unzip (Class.meths CD)))
    → ClassOk CD
```

Similarly, a method is well-formed in a class if it respects the MethodOk predicate. The expression typing judgment is used as a premise to type-check the expression body using the formal parameters as the environment Γ, expecting the type defined as the return type of the given method.

```
data MethodOk : Class → Method → Set where
  T-Method : ∀ {CD MD}
    → Meth.params MD ⊢ Meth.body MD : Meth.ret MD
    → MethodOk CD MD
```

[7] We use $proj_2$ to get the second argument of a tuple, and unzip to split a list of tuples.

Properties. We proved type soundness through the standard theorems of *preservation* and *progress* for the presented formalization of FJ.

The function preservation is the Agda encoding for the theorem with the same name, stating that if we have a well-typed expression, it preserves type after taking a reduction step. The proof proceeds by induction on the typing derivation of the first expression.

```
preservation : ∀ {e e′ τ} → [] ⊢ e : τ → e ⟶ e′ → [] ⊢ e′ : τ
preservation (T-Var x) ()
preservation (T-Field tp fls bnd) (RC-Field ev) =
  T-Field (preservation tp ev) fls bnd
preservation (T-Field (T-New fs₁ tps) fs₂ bnd) (R-Field fs₃ zp bnde)
  rewrite ≡-fields fs₁ fs₂ | ≡-fields fs₂ fs₃ = ⊢-interl zp tps bnd bnde
preservation (T-Invk tp tmt tpl) (RC-InvkRecv ev) =
  T-Invk (preservation tp ev) tmt tpl
preservation (T-Invk tp tmt tpl) (RC-InvkArg evl) =
  T-Invk tp tmt (preservation-list tpl evl)
preservation (T-Invk (T-New fls cp) tmt tpl) (R-Invk rmt zp)
  rewrite ≡-method rmt tmt = subst (⊢-method tmt) tpl zp
preservation (T-New fls tpl) (RC-NewArg evl) =
  T-New fls (preservation-list tpl evl)
```

The case for constructor T-Var is impossible, because a variable term cannot take a step, finishing this case using the Agda's absurd () pattern. Constructor T-Field has two cases: (1) the congruence rule, applying the induction hypothesis in the first expression; (2) the reduction step, where using the auxiliary lemmas ≡-fields and ⊢-interl we show that the expression e′ preserves the initial type of expression e. The T-Invk constructor is the most intricate, with three cases: (1) the congruence rule for the first expression, where the induction hypothesis is applied; (2) the congruence for the list of arguments, where we use an auxiliary proof preservation-list which call the induction hypothesis for each argument; (3) the reduction step, where we show that after a reduction step the type is preserved by using the auxiliary lemmas ≡-method, ⊢-method, and subst[8]. This proof is similar to λ-calculus, using the lemma stating that *Expression substitution preserves typing* (IGARASHI; PIERCE; WADLER, 2001). Lastly, T-New has only the congruence case for which we apply the induction hypothesis for each argument of the class constructor.

[8]These lemmas are omitted from this text, but can be found in our source code repository (FEITOSA, 2019).

The proof of progress for FJ follows the same script as usual: induction on the derivation of the typing rule.

```
progress : ∀ {e τ} → [ ] ⊢ e : τ → Progress e
progress (T-Var ())
progress (T-Field tp fls bnd) with progress tp
progress (T-Field tp fls bnd) | Step ev = Step (RC-Field ev)
progress (T-Field (T-New flds fts) fls bnd) | Done ev
  rewrite ≡-fields flds fls = Step (R-Field fls (proj₂ (⊨-interl fts))
    (proj₂ (∋-interl fts (proj₂ (⊨-interl fts)) bnd)))
progress (T-Invk tp mt tpl) with progress tp
progress (T-Invk tp mt tpl) | Step ev = Step (RC-InvkRecv ev)
progress (T-Invk tp mt tpl) | Done ev with progress-list tpl
progress (T-Invk tp mt tpl) | Done ev | Step evl =
  Step (RC-InvkArg evl)
progress (T-Invk (T-New flds fts) mt tpl) | Done ev | Done evl =
  Step (R-Invk mt (proj₂ (⊨-interl tpl)))
progress (T-New fls tpl) with progress-list tpl
progress (T-New fls tpl) | Step evl = Step (RC-NewArg evl)
progress (T-New fls tpl) | Done evl = Done (V-New evl)
```

Most cases are simple, as for λ-calculus. We use the same approach as before: a datatype definition called Progress to hold the cases for when the expression is a value or it can take a step. The complicated cases are those for T-Field and T-Invk, when processing the actual reduction step. When proving *progress* for T-Field, to be able to produce a R-Field we needed to write two extra lemmas ⊨-interl and ∋-interl, which were omitted here for brevity. The case for T-Invk also used the lemma ⊨-interl to produce a R-Invk.

5.2.2 Intrinsic Formalization

In this subsection we present an intrinsically-typed formalization and a definitional interpreter for the same subset of FJ presented in the previous subsection. Here, we show how *de Bruijn* indices can be used to deal with the subtleties of more elaborated binding mechanisms. This calculus uses a nominal type system, which differs from the structural type system of STLC. The text presented here is an adaptation of our paper "An Inherently-Typed Formalization for Featherweight Java" (FEITOSA et al., 2019). In our approach, we chose to drop all names, using *de Bruijn* indices to represent *name bindings* for classes, attributes, methods, and variables. First we define a type Ty where each class is represented by an index Fin n on the class table. Variable n represents the amount of classes in the source program, and it is bound as a module parameter in our Agda implementation.

As seen earlier, the syntax of FJ presents a mutual relation between expressions and class tables, i.e., a class can contain expressions, and an expression can relate to information in the class table. It gives rise to a cyclic dependency between the two elements, which makes the encoding of an intrinsically-typed syntax for FJ tricky. As a solution, we split the definition of a class in two parts: the *signature* of a class defines only the types of the fields and methods, whereas the *implementation* contains the actual code (expression) to be executed. This definition allows us to type-check expressions using information in the class table.

Intrinsically-typed syntax. We define a class signature CSig as a record with two fields. The flds definition represents the types for each attribute in a class. The signs field represents the method signatures, which is defined as a list of MSig. It is worth to note that we omit names for attributes and methods, since we are representing them as *de Bruijn* indices.

```
record CSig : Set where
  field
    flds  : List Ty
    signs : List MSig
```

The method signature is also defined as a record with two fields. The first ret represents the method return type, and the second params a list of types for each method parameter.

```
record MSig : Set where
  field
    ret    : Ty
    params : List Ty
```

We represent the table of *class signatures* as a vector Vec (coming from the standard library) and indexed by n, representing the number of defined classes in the programmer source-code.

```
CTSig : Set
CTSig = Vec CSig n
```

Similarly, we define CImpl, MImpl, and CTImpl[9] to represent the implementation of classes, methods, and class tables, respectively. Using this approach, each instance of CImpl is associated with its respective index of a CSig, and the field impls associates a body expression for each method.

[9]The details of implementation were omitted from this text, but can be found in our source code repository (FEITOSA, 2019).

Auxiliary functions. As for the extrinsically defined version of FJ, we have auxiliary definitions to obtain information from the class table. Function fields provides a list of all available attributes of a given class. The class table Δ is bound as a module parameter.

```
fields : Ty → List Ty
fields τ = CSig.flds (lookup Δ τ)
```

Similarly to fields, the function signatures provides a list of all available method signatures of a given class.

```
signatures : Ty → List MSig
signatures τ = CSig.signs (lookup Δ τ)
```

We also have a function to retrieve the implementations of a given class, which returns a list of all method implementations. We use the functions sym and lookup-allFin from Agda's standard library to associate the signatures with its implementations.

```
implementations : (τ : Ty) → CTImpl → All MImpl (signatures τ)
implementations τ δ rewrite sym (lookup-allFin τ) =
  CImpl.impls (lookup τ δ)
```

Once we have the definition of a class table, we can define the inherently-typed syntax for expressions. We start by defining an object-level type context Ctx encoded as a list of types (similar to STLC), which is used to store variable types. We represent each judgment of the FJ's static semantics as a constructor. As we mentioned before, FJ expressions can refer to information from two different sources: (1) on a well-formed class table Δ, which is defined globally after importing the class table module; (2) on variables in the type context Γ, which is expected as a parameter for the inductive datatype definition. Similarly to the intrinsically-typed definition of STLC, the Expr datatype is *indexed* by Ty, which represents the expected result type for the expression. The representation of expressions is defined in the code as follows. The reader can note that, to define the intrinsically-typed syntax of expressions, we use only *signatures*, never *implementations*.

```
data Expr (Γ : Ctx) : Ty → Set where
  Var   : ∀ {x} → x ∈ Γ → Expr Γ x
  Field : ∀ {c f} → Expr Γ c → f ∈ (fields c) → Expr Γ f
  Invk  : ∀ {c m} → Expr Γ c → m ∈ (signatures c)
    → All (Expr Γ) (MSig.params m) → Expr Γ (MSig.ret m)
  New   : ∀ c → All (Expr Γ) (fields c) → Expr Γ c
```

The constructor for Var is encoded the same way as for STLC, where we use a well-typed *de Bruijn* index x ∈ Γ which binds the type for variable x. The result type of this expression should be the one represented by the x variable.

The constructor Field expects an expression of type c and a *de Bruijn* index f ∈ (fields c). Again, we represent the expected field using a similar scheme to the one used for variables. Here f is the type of the expected field, c is the index of the given class, and fields c returns a list containing all the fields of the class represented by c. The result type Expr Γ f states that the expression has the type defined in f.

The constructor Invk receives three parameters, where the first is an expression having type c, the second a *de Bruijn* index m ∈ (signatures c), and the third uses the predicate All to associate each parameter with its expected type (MSig.params m). The result type for this expression Expr Γ (MSig.ret m) should be the one coming from the method's return type MSig.ret m. The constructor for New receives first an index representing a class c, and then similarly to the Invk constructor, it uses the predicate All to enforce each parameter to have the expected type using the information coming from a call of function fields c. The result type of this constructor is the type of the class c, which is being instantiated.

Values and environments. This procedure is the same adopted when defining the intrinsic version of STLC. First we define a Val, which in FJ has only one constructor representing an object creation with all parameters also being values. Here we also use the predicate All to guarantee this restriction.

```
data Val : Ty → Set where
  V-New : ∀ c → All Val (fields c) → Val c
```

And then, we define an Env, which links each type on the context Γ with a value Val.

```
Env : Ctx → Set
Env Γ = All Val Γ
```

Definitional Interpreter. Having all the building blocks to make the FJ interpreter, we can define the eval function. It receives four arguments, and returns a Maybe value. The first argument is the fuel, encoded as a natural number (Fuel = ℕ), used to ensure that the evaluator always terminates (AMIN; ROMPF, 2017; OWENS et al., 2016). The second parameter is the *implementation* of a class table. The third parameter is the run-time variable environment. And the last one is the expression Expr to be evaluated. The return of this function will provide a Val c in case of success, or nothing when the fuel runs out.

```
eval : Fuel → CTImpl → Env Γ → Expr Γ c → Maybe (Val c)
eval zero δ γ e = nothing
eval (suc f) δ γ (Var x) = just (lookup γ x)
eval (suc f) δ γ (Field e x) with eval f δ γ e
... | nothing = nothing
... | just (V-New c cp) = just (lookup cp x)
eval (suc f) δ γ (Invk e m mp) with eval f δ γ e
... | nothing = nothing
... | just (V-New c cp) with eval-list f δ γ mp
... | nothing = nothing
... | just mp′ = let mi = lookup (implementations c δ) m
      in eval f δ mp′ (MImpl.body mi)
eval (suc f) δ γ (New c cp) with eval-list f δ γ cp
... | nothing = nothing
... | just cp′ = just (V-New c cp′)
```

As we are using *fuel based evaluation* (AMIN; ROMPF, 2017; OWENS et al., 2016), we pattern match first on the fuel argument. It has two cases: zero when the fuel counter reaches zero, and our evaluation function returns nothing, or suc fuel when there is still fuel to proceed with the evaluation. Then we pattern match with the expression being evaluated, with one case for each FJ syntactic constructor. The case for Var is the same of STLC, except here the result is embedded in a Maybe monad. For Field first we have to evaluate the expression e, and then it is necessary to lookup the *de Bruijn* index f on the argument list cp. For the Invk constructor, we have to evaluate the expression e and the list of arguments mp, and then, using the implementations function, we select the method m, to evaluate its body, using the evaluated method parameters mp′ as the γ environment. Lastly, to evaluate New, we have to evaluate the parameters cp. For all recursive cases, we decrement the fuel parameter to guarantee termination, since FJ can implement recursion, and thus run indefinitely. It is worth noticing that the only way the eval function can result in nothing is when the fuel reaches zero. Again, we highlight here that there is no need for error control regarding to indices, since we have ensured everything is well-scoped using *de Bruijn* indices.

5.2.3 Elaborating FJ Extrinsic to Intrinsic

Similarly to STLC, we also elaborate the extrinsic syntax of FJ to the intrinsic version using the elabExpr function. Since in the intrinsic formalization of FJ we use *de Bruijn* indices instead of names (to represent variables, attributes, methods, and classes), we had to define one function to elaborate each name to its correspondent intrinsic index. Thus, the function elabVar is used to elaborate a variable name to an index, elabField and elabMeth work the same way for fields and methods. We also encoded the elabExprs

function to recursively apply elabExpr for each parameter when elaborating the method and constructor parameters, and function Name⇒Ty to produce an index for a given class name.

```
elabExpr : ∀ {Γ e τ} → Γ ⊢ e : τ → Expr (elabCtx Γ) (Name⇒Ty τ)
elabExpr (T-Var x) = Var (elabVar x)
elabExpr (T-Field wte fls wtf) = Field (elabExpr wte) (elabField fls wtf)
elabExpr (T-Invk {Γ = Γ} wte wtm wtmp) =
  Invk (elabExpr wte) (elabMeth wtm)
    (subst (All (Expr (elabCtx Γ))) (eq-mparams wtm) (elabExprs wtmp))
elabExpr (T-New {Γ = Γ} {C = C} flds wtcp) =
  New (Name⇒Ty C)
    (subst (All (Expr (elabCtx Γ))) (eq-fields flds) (elabExprs wtcp))
```

5.3 Comparing Styles of Formalization

We have described in this thesis the formalization of two programming languages in Agda using the syntactical (extrinsic) and functional (intrinsic) approaches. We presented the process of writing the relational semantics rules together with soundness proofs and the intrinsically-typed syntax together with a definitional interpreter to achieve a similar result. In this section we compare the resulting specifications using two metrics: lines of code, and number of high-level structures.

Table 2 compares our two language definitions considering the *extrinsic* and *intrinsic* formalization approaches regarding to the approximate number of lines-of-code[10] (LOC) to provide a type soundness result. Roughly, the table shows us that, if we consider the sum of all modules (*Total*), we had to write a considerable amount of lines when applying the extrinsic compared to its intrinsic version. However, to express the syntax and evaluation rules, we used a similar amount of lines of code. The main difference between both approaches is that we do not need to write any line to define the type-checker nor the soundness proofs using the intrinsic approach (since it guarantees safety-by-construction).

Obviously, lines of code is only one metric which can be used to compare the expressivity of each approach, and not its complexity. The main advantage of using an extrinsic approach is to have a depth control over the structures and the semantics, being able to follow step-by-step during the proof construction. Also, by using proof assistants with proof automation mechanisms (such as Coq), we could considerably decrease the number of lines to prove the same theorems. For the intrinsic version, an

[10]We call here “approximate” because we are not considering lines to create/import modules, and some break lines to improve readability.

	Ext. STLC	Int. STLC	Ext. FJ	Int. FJ
Syntax	10	10	19	26
Typing Rules	16	0	39	0
Evaluation	22	8	40	25
Proofs	75	0	99	0
Auxiliary	8	12	23	13
Total	131	30	220	64

Table 2 – Approximate sizes (in LOC) for our STLC and FJ developments.

advantage is that type safety becomes a guiding contract when writing a program, and this guarantee comes from the host language used in the development. Furthermore, we can reuse more code from Agda's standard library.

In Table 3 we present a comparison of the number of high-level structures developed in our source-code formalizations. Again, we can see that using the intrinsic version, we can express the syntax, typing rules, evaluation and proofs using a small number of high level definitions (functions or lemmas, and inductive definitions). We can see that for the case of STLC, it naturally fits into Agda's definition, since both languages are functional. For the case of FJ, the intricate relation between its object-oriented features forces us to express its internal features using some extra high-level structures than for STLC. However, we could express both languages using less high-level structures in the intrinsic version when compared with its respective extrinsic one.

	Ext. STLC	Int. STLC	Ext. FJ	Int. FJ
Nr. Functions/Lemmas	10	2	21	5
Nr. Inductive Definitions	9	3	19	10

Table 3 – Number of high-level definitions for our STLC and FJ developments.

Although there is a considerable difference between the LOC and high-level structures considering the different approaches to formalize a programming language in Agda, we could note during our development that the intrinsic version requires creativity and insight to find the correct representation of the semantics, which is relatively simple for the extrinsic one. Usually, in the extrinsic approach we have to basically translate the relation on the structural operational semantics of the language into the proof assistant, and follow the script to prove type soundness. The most difficult part is to reason on how to break the proofs in small lemmas, so they are accepted by the proof assistant. For the intrinsic case, sometimes we have to tweak some definitions, to model the necessary invariants to obtain a type soundness result. This effort could be noted in our intrinsic formalization of FJ, where we had to split the definition of a class table in signatures and implementations to allow the intrinsically-typed syntax to be sound-by-construction.

5.4 Formalizing Java 8 Features in Featherweight Java

After comparing both formalization approaches, we chose to formalize the Java 8 features using the intrinsically-typed method, since it is more concise and elegant than the usual extrinsic approach. Although the formalization of FJ is more complicated than the usual λ-calculus, because of the complex binding mechanisms and the mutual relation between the syntactic constructions, the good news is that we only need to do this modeling of the language once. After that, we can add any number of extensions we want to the Java language, as we will show next.

5.4.1 Extended Definitions

As we saw in the previous chapter, the new Java 8 constructions use *functional interfaces* to assign types to λ-expressions in Java. Since FJ does not provide interfaces in its definitions, we have to add it in our calculus. The first step we take is to change our definition of types, now allowing the representation of classes and interfaces. Both CI and Fi are defined as a finite datatype Fin, representing classes and functional interfaces, respectively. Thus, a Ty can be constructed with class, which receives as argument a class index CI, or with interface which receives as argument an interface index Fi.

```
data Ty : Set where
  class     : CI → Ty
  interface : Fi → Ty
```

We represent a functional interface with the record ISig, which models a single method signature. The field sign is defined by a tuple List Ty × Ty, representing the parameter types and the return type of the method being specified.

```
record ISig : Set where
  field
    sign : List Ty × Ty
```

Similar to the CTSig presented in the last section, here we define an interface table ITSig, which is also represented by a Vec with at most i elements, where i is the number of interfaces in the source code program.

```
ITSig : Set
ITSig = Vec ISig i
```

We also defined the auxiliary function isignature to obtain information about the method signature in a functional interface.

```
isignature : ITSig → Fi → List Ty × Ty
isignature ξ τ = ISig.sign (lookup ξ τ)
```

After having implemented the main changes on the well-formed class and interface table, we added two constructors in the intrinsically-typed syntax of FJ to deal with λ-expressions. The reader can note that we added an extra argument (with type Maybe Cl) in the Expr constructor. This argument represents the Java *this* pointer, which is used to refer to the class itself and can be used only inside classes.

```
data Expr (Γ : Ctx) : Maybe Cl → Ty → Set where
  -- Other FJ definitions
  Lam  : ∀ {i} → Expr (proj₁ (isignature (ζ Δ) i)) nothing (proj₂ (isignature (ζ Δ) i))
               → Expr Γ nothing (interface i)
  InvkL : ∀ {i} → Expr Γ nothing (interface i)
               → All (Expr Γ nothing) (proj₁ (isignature (ζ Δ) i))
               → Expr Γ nothing (proj₂ (isignature (ζ Δ) i))
```

The constructor Lam represents a λ-expression. It can be used to implement higher-order functions, and can be assigned to variables, passed as method arguments, etc. This constructor has only one argument, which is an expression Expr with the Γ context set to $(\text{proj}_1\ (\text{isignature}\ (\zeta\ \Delta)\ i))$ (the parameter types of a λ-expressions defined in the functional interface), the *this* pointer set to nothing (because *this* can appear only in classes), and the expected type set to $(\text{proj}_2\ (\text{isignature}\ (\zeta\ \Delta)\ i))$ (the return type of a λ-expression). It constructs an expression with type $(\text{interface}\ i)$. The constructor InvkL is responsible to execute the λ-expression, and receives two arguments. The first is the actual λ-expression (represented by the constructor Lam), and the second is the actual parameters to be substituted on the λ-expression body, using information about types from the functional interface. It constructs an expression which type is the same of the return type of the λ-expression.

A λ-expression is also represented as a value in our calculus, and its definition is presented next. The meaning is similar to the Lam constructor, however here it constructs a value Val.

```
data Val : Ty → Set where
  -- Other FJ definitions
  VLam : ∀ {i} → Expr (proj₁ (isignature (ζ Δ) i)) nothing (proj₂ (isignature (ζ Δ) i))
               → Val (interface i)
```

All the presented definitions were easily incorporated in our previous formalization, showing that indeed we can use the proposed approach to study new constructions in a complex object-oriented setting.

5.4.2 Definitional Interpreter

We adapted our eval function to deal with the new Java 8 constructors. First we added an extra argument to deal with the *this* pointer regarding classes and λ-expressions. It is expected as a monadic Maybe value, since it can only appear in classes. The reader can note the direct relation between this monadic value and the expression to be evaluated.

```
eval : ∀ {Γ τ c} → Fuel → (m : Maybe (Val (class τ))) → CTImpl → Env Γ
                  → Expr Γ (maybe (λ x → just τ) nothing m) c → Maybe (Val c)
```

We added two defining equations in our definitional interpreter presented earlier. The first is very simple, it evaluates a Lam constructor to a VLam value directly.

```
eval (suc fuel) nothing δ γ (Lam e) = just (VLam e)
```

The second defining equation is similar to the evaluation of method invocations. First we use the function eval-list to evaluate the actual parameter list lp, producing a list of values lp′. Then we use the function eval to evaluate the left-hand expression e, producing a λ-expression value. Both calls can ran out of fuel, and abort the evaluation with nothing. Having the λ-expression value we call eval one more time to evaluate the its body b using lp′ as the runtime environment. It is important to highlight again that we do not need to check if the produced value is a VNew, since it is guaranteed by construction that only λ-expression values (VLam) will be allowed when processing the presented constructor.

```
eval {Γ} {τ} (suc fuel) nothing δ γ (InvkL e lp) with eval-list {Γ} {τ} fuel nothing δ γ lp
... | nothing = nothing
... | just lp′ with eval {Γ} {τ} fuel nothing δ γ e
... | nothing = nothing
... | just (VLam b) = eval {τ = τ} fuel nothing δ lp′ b
```

Using such intrinsically-typed approach we achieve a similar soundness result if compared to the extrinsic method. Programs constructed using the inherently-typed ASTs are type-sound by construction, where only well-typed programs can be expressed, so the *preservation* property is enforced by the host-language type checker (AMIN; ROMPF, 2017). And by implementing the definitional interpreter in a total language like Agda, i.e., specifying the dynamic semantics in a functional way, instead of relational we also show the *progress* property, without the need for an extra explicit proof (OWENS et al., 2016). Considering all this, we believe that this approach is promising to be explored on even more complex programming languages.

5.5 Related Work

There is a vast body of literature on soundness and proof techniques regarding programming languages. The most relevant styles are from Wright and Felleisen's syntactic approach (WRIGHT; FELLEISEN, 1994), Plotkin's structural operational semantics (PLOTKIN, 2004), Kahn's natural semantics (KAHN, 1987), and Reynold's definitional interpreters (REYNOLDS, 1972). Although the common ground is to find mechanized formalization using more syntactic (extrinsic) approaches, we could see the number of functional (intrinsic) encodings of semantics increasing in recent years.

Considering the extrinsic approach, there are several papers describing the mechanization of both, λ-calculus and Featherweight Java. For example, in their book, Pierce et al. (PIERCE et al., 2018) describe the formalization of STLC in Coq, and Wadler (WADLER, 2018) present the formalization of STLC in Agda. We applied a simplified version of the ideas presented in these books in our formalization of STLC. Besides these books, there are several other papers mechanizing different versions of λ-calculus (RIBEIRO; FIGUEIREDO; CAMARÃO, 2013; DONNELLY; XI, 2007). Regarding Featherweight Java, there are some projects describing its formalization. Mackay et al. (MACKAY et al., 2012) developed a mechanized formalization of FJ with assignment and immutability in Coq, proving type-soundness for their results. Delaware et al. (DELAWARE; COOK; BATORY, 2011) used FJ as basis to describe how to engineer product lines with theorems and proofs built from feature modules, also carrying the formalization Coq. Both papers inspired us in our modeling of FJ. As far as we know, our work is the first to formalize FJ in Agda using the extrinsic approach.

The formalization of programming languages combining an inherently-typed syntax (showed in (ALTENKIRCH; REUS, 1999; AUGUSTSSON; CARLSSON, 1999; REYNOLDS, 2001)) and definitional interpreters has also been made more often. Danielsson (DANIELSSON, 2012) used the co-inductive partiality monad to formalize λ-calculus using total definitional interpreters, demonstrating that the resulting semantics are useful type-soundness results. Benton et al. (BENTON et al., 2012) used Coq to formalize an intrinsic version of STLC using *de Bruijn* indices to deal with name binding. In his book, Wadler (WADLER, 2018) also discuss the definition of STLC using an intrinsically-typed approach. We can find some other results applying these techniques (MCBRIDE, 2010; ALTENKIRCH; KAPOSI, 2016). On the formalization of modern programming languages (such as Java), Affeldt and Sakaguchi (AFFELDT; SAKAGUCHI, 2014) present an intrinsic encoding in Coq of a subset of C applying it to TLS network processing, Owens et al. (OWENS et al., 2016) advocate for the use of a definitional interpreter written in a purely functional style in a total language. Amin and Rompf (AMIN; ROMPF, 2017) show how type soundness proofs for advanced, polymorphic, and object-oriented type systems can be carried out with an operational

semantics based on definitional interpreters implemented in Coq. More closely to our intrinsic formalization of FJ, Bach Poulsen et al. (BACH POULSEN et al., 2017) present a formalization of Middleweight Java (a variant of FJ) defined in Agda and some techniques to deal with name binding. In our formalization, we used only standard techniques (like *de Bruijn* indices) to formalize FJ, trying to keep the code as simple as possible to facilitate readability and maintainability. Furthermore, following the specifications presented in (FEITOSA; RIBEIRO; DU BOIS, 2018b), where the properties of progress and preservation were proved (on paper) and tested with QuickCheck, we were able to achieve a similar soundness result in our intrinsically-typed formalization in Agda. To the best of our knowledge, we were the first to define an intrinsically-typed (mechanically checked) definition of FJ, as well as an extension of it with functional interfaces and λ-expressions.

5.6 Chapter's Final Remarks

This chapter summarized our results using the main approaches on the formalization of programming languages, extrinsic and intrinsic, applied to two different languages (and an extension), which implement different paradigms. We developed our formalizations in a way that they can be extended in future projects (as shown in Section 5.4), providing insights for how to formalize some concepts, and also to save time by reusing a working source-code. Besides, we provided a comparison between the formalization styles, which can be useful for a programmer when starting a project involving programming language semantics.

6 CONCLUSION

In this final chapter, we present the final remarks about this thesis, discussing the limitations of the achieved results, and presenting some possible paths to continue the research in this area.

6.1 Thesis' Final Remarks

We have studied the formalization of various programming languages subsets with the ultimate goal to be able to apply property-based testing and formal verification on them. Since our ideas needed formal specifications, we restricted our research on calculus described using (at least) small-step semantics and proofs for type soundness. Based in these studies we were able to make a choice between programming language subsets in the functional and object-oriented paradigm. Naturally, we chose λ-calculus to represent the functional paradigm, and we chose FJ after comparing it with different object-oriented subsets. As an indirect consequence, we had the opportunity to understand better distinct ways to formalize the semantics and the type system of programming languages.

After defining the programming languages to explore further, we discussed the results of the first branch of this thesis. We presented a syntax directed judgment for generating type correct programs, for both STLC and FJ, proving soundness with respect to their typing rules. We implemented the proposed algorithms and applied property-based testing using QuickCheck to verify soundness properties against hand-written interpreters. Indeed, we explored the approach even further, proposing a formal semantics for FJ extended with Java 8 features (such as functional interfaces, λ-expression, and default methods), generating well-typed programs and applying property-based testing to check whether soundness properties were still valid for the extended calculus. The lightweight approach provided by QuickCheck allowed us 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 (specification and implementation) 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.

The second branch of this thesis discussed the formalization of the studied programming languages using the syntactic (extrinsic) and functional (intrinsic) approaches with the Agda programming language. When formalizing the STLC, we were able to analyze the differences between each approach to prove type soundness in a small setting. The formalization of FJ brought some extra challenges, mostly because of its mutual relation between expressions and class tables, and because of its complex binding mechanisms. For both languages and scenarios we could observe the complexities of each approach, and their advantages. By using an extrinsic formalization, we can have access to all the steps when processing the static and dynamic semantics, while using the intrinsic version we avoid repetitions and receive type soundness proofs (almost) for free.

We believe that our project gives one step further in the research of programming languages, and that the combination of property-based testing with formal verification can be very useful to reduce the amount of work when mechanically proving properties, since it is the most costly and time consuming task.

6.2 Limitations

As the reader could note, this thesis was focused in working with only two programming language subsets, limiting the range of our work. Obviously, it would be impractical to cover all existent programming languages and paradigms, however we tried to cover important aspects of two programming language paradigms. Our generation algorithms were presented together with informal (non-mechanized) proofs that they are sound according to their specifications, which could be improved by machine-checking the presented proofs. The use of FJ does not reflect the exactly semantics of Java (and also our extension with λ-expressions) since several features are missing in this subset. When considering our formalizations in Agda, we limited FJ even more to be able to contrast more clearly the differences between the extrinsic and intrinsic approaches. It is hard to say how well the techniques explored in this thesis would behave when exploring different programming language paradigms. Several limitations presented here are listed as future work and explained in the next section.

6.3 Future Work

There is plenty of room for improvement and deepening upon the work developed in this thesis. In this section we discuss a few of the possibilities.

Studying different languages. To be able to produce a text related to the subject

of this thesis, we had to limit the amount of languages to work on. A good starting point to work in this area is the study or specification of different programming language semantics, and the application of the techniques presented here. The acquired knowledge when doing this task can be useful to apply on bigger and more complex projects.

Testing compilers. Using the test generators produced in this thesis, one can apply differential testing in order to check the existence of bugs in real compilers, with few adaptations in our approach. Furthermore, the same technique can be explored to generate programs for different languages, expanding the horizon of this research. We also consider the development of a composable random program generator in the style of “Data types a la carte” (SWIERSTRA, 2008), to be able to reuse the generation procedures of similar constructors for different programming languages.

Soundness and completeness. An important step regarding our generation method is to mechanically prove the soundness properties presented in our text, as well as proving the completeness property to demonstrate that all values of a type will eventually be generated when performing our test suite. We gave a first step in this direction by checking the code coverage of our interpreters, however a formal proof would bring elegance and conciseness impossible to achieve only by testing.

Extrinsic and intrinsic verification. We should explore further the presented techniques in our case studies, expanding the STLC subset with more complex constructors, as well as to formalize the complete calculus of FJ, since we focused the formalization in the most important object-oriented constructors in this thesis. Furthermore, we intend to provide machine checked proofs for the Java 8 constructors also using the extrinsic approach, following the presented specification and the ideas explored using property-based testing. Lastly, by using our case studies one can be able to apply both the extrinsic and intrinsic approaches in order to verify different language subsets.

Equivalence between formalization approaches. Another interesting subject to be explored is to mechanically prove equivalence between the extrinsic and intrinsic versions of language semantics. Indeed, we have done a step in this direction, allowing the elaboration of an extrinsic well-typed expression into an intrinsically-typed one. Having a proof like that, one can safely choose the approach that fits best for each part of a project.

REFERENCES

AFFELDT, R.; SAKAGUCHI, K. An Intrinsic Encoding of a Subset of C and its Application to TLS Network Packet Processing. **Journal of Formalized Reasoning**, [S.l.], v.7, n.1, p.63–104, 2014.

ALLWOOD, T. O. R.; EISENBACH, S. Tickling Java with a Feather. **Electron. Notes Theor. Comput. Sci.**, Amsterdam, The Netherlands, The Netherlands, v.238, n.5, p.3–16, Oct. 2009.

ALTENKIRCH, T.; KAPOSI, A. Type Theory in Type Theory Using Quotient Inductive Types. **SIGPLAN Not.**, New York, NY, USA, v.51, n.1, p.18–29, Jan. 2016.

ALTENKIRCH, T.; REUS, B. Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In: INTERNATIONAL WORKSHOP AND 8TH ANNUAL CONFERENCE OF THE EACSL ON COMPUTER SCIENCE LOGIC, 13., 1999, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 1999. p.453–468. (CSL '99).

AMIN, N.; ROMPF, T. Type Soundness Proofs with Definitional Interpreters. **SIGPLAN Not.**, New York, NY, USA, v.52, n.1, p.666–679, Jan. 2017.

AUGUSTSSON, L.; CARLSSON, M. An exercise in dependent types: A well-typed interpreter. In: IN WORKSHOP ON DEPENDENT TYPES IN PROGRAMMING, GOTHENBURG, 1999. **Anais...** [S.l.: s.n.], 1999.

BACH POULSEN, C. et al. Intrinsically-typed Definitional Interpreters for Imperative Languages. **Proc. ACM Program. Lang.**, New York, NY, USA, v.2, n.POPL, p.16:1–16:34, Dec. 2017.

BARENDREGT, H. P. In: ABRAMSKY, S.; GABBAY, D. M.; MAIBAUM, S. E. (Ed.). **Handbook of Logic in Computer Science (Vol. 2)**. New York, NY, USA: Oxford University Press, Inc., 1992. p.117–309.

BAZZICHI, F.; SPADAFORA, I. An automatic generator for compiler testing. **IEEE Transactions on Software Engineering**, [S.l.], n.4, p.343–353, 1982.

BELLIA, M.; OCCHIUTO, M. E. Properties of Java Simple Closures. **Fundam. Inf.**, Amsterdam, The Netherlands, The Netherlands, v.109, n.3, p.237–253, Aug. 2011.

BENTON, N.; HUR, C.-K.; KENNEDY, A. J.; MCBRIDE, C. Strongly Typed Term Representations in Coq. **J. Autom. Reason.**, Secaucus, NJ, USA, v.49, n.2, p.141–159, Aug. 2012.

BETTINI, L.; BONO, V.; DEZANI-CIANCAGLINI, M.; VENNERI, B. Java & Lambda: a Featherweight Story. **CoRR**, [S.l.], v.abs/1801.05052, 2018.

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. **Anais...** [S.l.: s.n.], 2017.

BOGDANAS, D.; ROSU, G. K-Java: A Complete Semantics of Java. **SIGPLAN Not.**, New York, NY, USA, v.50, n.1, p.445–456, Jan. 2015.

BRUIJN, N. de. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. **Indagationes Mathematicae (Proceedings)**, [S.l.], v.75, n.5, p.381 – 392, 1972.

CELENTANO, A. et al. Compiler testing using a sentence generator. **Software: Practice and Experience**, [S.l.], v.10, n.11, p.897–918, 1980.

CHURCH, A. A Set of Postulates for the Foundation of Logic. **Annals of Mathematics**, [S.l.], v.33, n.2, p.346–366, 1932.

CLAESSEN, K.; HUGHES, J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: FIFTH ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, 2000, New York, NY, USA. **Proceedings...** ACM, 2000. p.268–279. (ICFP '00).

DANIEL, B.; DIG, D.; GARCIA, K.; MARINOV, D. Automated Testing of Refactoring Engines. In: JOINT MEETING OF THE EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, 6., 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.185–194. (ESEC-FSE '07).

DANIELSSON, N. A. Operational Semantics Using the Partiality Monad. In: ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, 17., 2012, New York, NY, USA. **Proceedings...** ACM, 2012. p.127–138. (ICFP '12).

DELAWARE, B.; COOK, W.; BATORY, D. Product Lines of Theorems. **SIGPLAN Not.**, New York, NY, USA, v.46, n.10, p.595–608, oct 2011.

DONNELLY, K.; XI, H. A Formalization of Strong Normalization for Simply-Typed Lambda-Calculus and System F. **Electron. Notes Theor. Comput. Sci.**, Amsterdam, The Netherlands, The Netherlands, v.174, n.5, p.109–125, June 2007.

DRIENYOVSZKY, D.; HORPáCSI, D.; THOMPSON, S. Quickchecking Refactoring Tools. In: ACM SIGPLAN WORKSHOP ON ERLANG, 9., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.75–80. (Erlang '10).

DROSSOPOULOU, S.; EISENBACH, S. Java is Type Safe - Probably. In: IN EUROPEAN CONFERENCE ON OBJECT ORIENTED PROGRAMMING, 1997. **Anais...** Springer-Verlag, 1997. p.389–418.

DROSSOPOULOU, S.; EISENBACH, S. Describing the Semantics of Java and Proving Type Soundness. In: FORMAL SYNTAX AND SEMANTICS OF JAVA, 1999, London, UK, UK. **Anais...** Springer-Verlag, 1999. p.41–82.

DROSSOPOULOU, S.; EISENBACH, S.; KHURSHID, S. Is the Java Type System Sound? **Theor. Pract. Object Syst.**, New York, NY, USA, v.5, n.1, p.3–24, Jan. 1999.

FARZAN, A.; CHEN, F.; MESEGUER, J. Formal analysis of Java programs in JavaFAN. In: IN PROCEEDINGS OF CAV, 2004. **Anais...** [S.l.: s.n.], 2004. p.501–505.

FEITOSA, S. d. S. **Strategies for Testing and Formalizing Modern Programming Languages - Online repository.** `https://github.com/fjpub/stfmpl`.

FEITOSA, S. d. S.; MENA, A. S.; RIBEIRO, R. G.; BOIS, A. R. D. An Inherently-Typed Formalization for Featherweight Java. In: XXIII BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 2019, New York, NY, USA. **Proceedings...** ACM, 2019. p.11–18. (SBLP 2019).

FEITOSA, S. d. S.; RIBEIRO, R. G.; DU BOIS, A. R. Formal Semantics for Java-like Languages and Research Opportunities. **Revista de Informática Teórica e Aplicada**, [S.l.], v.25, n.3, p.62–74, 2018.

FEITOSA, S. d. S.; RIBEIRO, R. G.; DU BOIS, A. R. Property-based Testing for Lambda Expressions Semantics in Featherweight Java. In: XXII BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, 2018, New York, NY, USA. **Proceedings...** ACM, 2018. p.43–50. (SBLP '18).

FEITOSA, S. d. S.; RIBEIRO, R. G.; DU BOIS, A. R. A Type-Directed Algorithm to Generate Well-Typed Featherweight Java Programs. In: FORMAL METHODS: FOUNDATIONS AND APPLICATIONS, 2018, Cham. **Anais...** Springer International Publishing, 2018. p.39–55.

FEITOSA, S. d. S.; RIBEIRO, R. G.; DU BOIS, A. R. Generating Random Well-Typed Featherweight Java Programs Using QuickCheck. **Electronic Notes in Theoretical Computer Science**, [S.l.], v.342, p.3 – 20, 2019. The proceedings of CLEI 2018, the XLIV Latin American Computing Conference.

FLATT, M.; KRISHNAMURTHI, S.; FELLEISEN, M. Classes and Mixins. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 25., 1998, New York, NY, USA. **Proceedings...** ACM, 1998. p.171–183. (POPL '98).

FLATT, M.; KRISHNAMURTHI, S.; FELLEISEN, M. A Programmer's Reduction Semantics for Classes and Mixins. In: FORMAL SYNTAX AND SEMANTICS OF JAVA, 1999, London, UK, UK. **Anais...** Springer-Verlag, 1999. p.241–269.

GILL, A.; RUNCIMAN, C. Haskell Program Coverage. In: ACM SIGPLAN WORKSHOP ON HASKELL WORKSHOP, 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.1–12. (Haskell '07).

GLIGORIC, M. et al. Test generation through programming in UDITA. In: ACM/IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING-VOLUME 1, 32., 2010. **Proceedings...** [S.l.: s.n.], 2010. p.225–234.

GOOGLE. **Google Scholar**. Accessed: 2018-01-12, `https://scholar.google.com.br/`.

HARPER, R. **Practical Foundations for Programming Languages**. 2nd.ed. New York, NY, USA: Cambridge University Press, 2016.

HINDLEY, J. R.; SELDIN, J. P. **Lambda-Calculus and Combinators**: An Introduction. 2.ed. New York, NY, USA: Cambridge University Press, 2008.

IGARASHI, A.; PIERCE, B. C.; WADLER, P. Featherweight Java: A Minimal Core Calculus for Java and GJ. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.23, n.3, p.396–450, May 2001.

KAHN, G. Natural Semantics. In: ANNUAL SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SCIENCES ON STACS 87, 4., 1987, London, UK, UK. **Anais...** Springer-Verlag, 1987. p.22–39.

KLEIN, C.; FLATT, M.; FINDLER, R. B. Random Testing for Higher-order, Stateful Programs. In: ACM INTERNATIONAL CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.555–566. (OOPSLA '10).

KLEIN, G.; NIPKOW, T. A Machine-checked Model for a Java-like Language, Virtual Machine, and Compiler. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.28, n.4, p.619–695, July 2006.

KLEIN, G.; NIPKOW, T. **Jinja is not Java - Archive of Formal Proofs**. Accessed: 2017-09-06, `https://www.isa-afp.org/entries/Jinja.html`.

LAMPROPOULOS, L.; PARASKEVOPOULOU, Z.; PIERCE, B. C. Generating Good Generators for Inductive Relations. **Proc. ACM Program. Lang.**, New York, NY, USA, v.2, n.POPL, p.45:1–45:30, Dec. 2017.

LE, V.; AFSHARI, M.; SU, Z. Compiler Validation via Equivalence Modulo Inputs. **SIGPLAN Not.**, New York, NY, USA, v.49, n.6, p.216–226, June 2014.

LINDIG, C. Random Testing of C Calling Conventions. In: SIXTH INTERNATIONAL SYMPOSIUM ON AUTOMATED ANALYSIS-DRIVEN DEBUGGING, 2005, New York, NY, USA. **Proceedings...** ACM, 2005. p.3–12. (AADEBUG'05).

LIPOVACA, M. **Learn you a haskell for great good!**: a beginner's guide. [S.l.]: no starch press, 2011.

MACKAY, J. et al. Encoding Featherweight Java with Assignment and Immutability Using the Coq Proof Assistant. In: WORKSHOP ON FORMAL TECHNIQUES FOR JAVA-LIKE PROGRAMS, 14., 2012, New York, NY, USA. **Proceedings...** ACM, 2012. p.11–19. (FTfJP '12).

MARTIN-LÖF, P. An intuitionistic theory of types. In: **Twenty-five years of constructive type theory (Venice, 1995)**. [S.l.]: Oxford Univ. Press, New York, 1998. p.127–172. (Oxford Logic Guides, v.36).

MCBRIDE, C. Outrageous but Meaningful Coincidences: Dependent Type-safe Syntax and Evaluation. In: ACM SIGPLAN WORKSHOP ON GENERIC PROGRAMMING, 6., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.1–12. (WGP '10).

MIDTGAARD, J. et al. Effect-driven QuickChecking of Compilers. **Proc. ACM Program. Lang.**, New York, NY, USA, v.1, n.ICFP, p.15:1–15:23, Aug. 2017.

MONGIOVI, M. et al. Scaling testing of refactoring engines. In: SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), 2014 IEEE INTERNATIONAL CONFERENCE ON, 2014. **Anais...** [S.l.: s.n.], 2014. p.371–380.

MOURA, L. de et al. The Lean theorem prover (system description). In: INTERNATIONAL CONFERENCE ON AUTOMATED DEDUCTION, 2015. **Anais...** [S.l.: s.n.], 2015. p.378–388.

NIELSON, H. R.; NIELSON, F. **Semantics with Applications**: An Appetizer (Undergraduate Topics in Computer Science). Berlin, Heidelberg: Springer-Verlag, 2007.

NIPKOW, T. Jinja: Towards a comprehensive formal semantics for a Java-like language. In: IN PROCEEDINGS OF THE MARKTOBERDORF SUMMER SCHOOL. NATO SCIENCE SERIES, 2003. **Anais...** Press, 2003.

NIPKOW, T.; OHEIMB, D. von. Javalight is Type-safe&Mdash;Definitely. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 25., 1998, New York, NY, USA. **Proceedings...** ACM, 1998. p.161–170. (POPL '98).

NORELL, U. **Dependently Typed Programming in Agda**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. p.230–266.

ORACLE.COM. **The Java Language Specification**. Accessed: 2018-05-03, `https://docs.oracle.com/javase/specs/jls/se10/html/`.

OWENS, S.; MYREEN, M. O.; KUMAR, R.; TAN, Y. K. Functional Big-Step Semantics. In: EUROPEAN SYMPOSIUM ON PROGRAMMING LANGUAGES AND SYSTEMS - VOLUME 9632, 25., 2016, New York, NY, USA. **Proceedings...** Springer-Verlag New York: Inc., 2016. p.589–615.

PAŁKA, M. H.; CLAESSEN, K.; RUSSO, A.; HUGHES, J. Testing an Optimising Compiler by Generating Random Lambda Terms. In: INTERNATIONAL WORKSHOP ON AUTOMATION OF SOFTWARE TEST, 6., 2011, New York, NY, USA. **Proceedings...** ACM, 2011. p.91–97. (AST '11).

PIERCE, B. C. **Types and Programming Languages**. 1st.ed. [S.l.]: The MIT Press, 2002.

PIERCE, B. C. et al. **Software Foundations Volume 2**: Programming Language Foundations. Electronic textbook. Version 5.7.

PLOTKIN, G. D. **A structural approach to operational semantics**. [S.l.: s.n.], 1981.

PLOTKIN, G. D. A structural approach to operational semantics. **J. Log. Algebr. Program.**, [S.l.], v.60-61, p.17–139, 2004.

POTTIER, F. Hindley-milner Elaboration in Applicative Style: Functional Pearl. In: ACM SIGPLAN INTERNATIONAL CONFERENCE ON FUNCTIONAL PROGRAMMING, 19., 2014, New York, NY, USA. **Proceedings...** ACM, 2014. p.203–212. (ICFP '14).

REYNOLDS, J. C. Definitional Interpreters for Higher-order Programming Languages. In: ACM ANNUAL CONFERENCE - VOLUME 2, 1972, New York, NY, USA. **Proceedings...** ACM, 1972. p.717–740. (ACM '72).

REYNOLDS, J. C. **What do Types Mean? - From Intrinsic to Extrinsic Semantics**.

RIBEIRO, R.; FIGUEIREDO, L.; CAMARÃO, C. Mechanized metatheory for a lambda-calculus with trust types. **Journal of the Brazilian Computer Society**, [S.l.], v.19, n.4, p.433–443, Nov 2013.

SERRANO MENA, A. **Beginning Haskell**: A Project-Based Approach. [S.l.]: Apress, 2014.

SILVA, T. D. da; SAMPAIO, A.; MOTA, A. Verifying Transformations of Java Programs Using Alloy. In: BRAZILIAN SYMPOSIUM ON FORMAL METHODS, 2015. **Anais...** [S.l.: s.n.], 2015. p.110–126.

SOARES, G.; GHEYI, R.; MASSONI, T. Automated behavioral testing of refactoring engines. **IEEE Transactions on Software Engineering**, [S.l.], v.39, n.2, p.147–162, 2013.

SØRENSEN, M. H.; URZYCZYN, P. **Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)**. New York, NY, USA: Elsevier Science Inc., 2006.

STARK, R. F.; BORGER, E.; SCHMID, J. **Java and the Java Virtual Machine**: Definition, Verification, Validation with Cdrom. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.

STUMP, A. **Verified Functional Programming in Agda**. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool, 2016.

SWIERSTRA, W. Data Types à La Carte. **J. Funct. Program.**, New York, NY, USA, v.18, n.4, p.423–436, July 2008.

SYME, D. Proving Java Type Soundness. In: FORMAL SYNTAX AND SEMANTICS OF JAVA, 1999, London, UK, UK. **Anais...** Springer-Verlag, 1999. p.83–118.

TIOBE.COM. **TIOBE Index**. Accessed: 2019-09-11, `https://www.tiobe.com/tiobe-index/`.

WADLER, P. Programming Language Foundations in Agda. In: FORMAL METHODS: FOUNDATIONS AND APPLICATIONS, 2018, Cham. **Anais...** Springer International Publishing, 2018. p.56–73.

WRIGHT, A.; FELLEISEN, M. A Syntactic Approach to Type Soundness. **Inf. Comput.**, Duluth, MN, USA, v.115, n.1, p.38–94, Nov. 1994.

YANG, X.; CHEN, Y.; EIDE, E.; REGEHR, J. Finding and Understanding Bugs in C Compilers. **SIGPLAN Not.**, New York, NY, USA, v.46, n.6, p.283–294, June 2011.