Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Feature Featherweight Java: Implementation & Formalization

Pedro da C. Abreu Jr.

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2016

CIP — Catalogação Internacional na Publicação

Abreu Jr., Pedro da C..

Feature Featherweight Java: Implementation & Formalization / Pedro da C. Abreu Jr.. Brasília : UnB, 2016.
69 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. Verificação Formal, 2. FFJ, 3. Coq

CDU 004.4

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Feature Featherweight Java: Implementation & Formalization

Pedro da C. Abreu Jr.

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Rodrigo B. de Almeida	Prof. Dr. Flávio L. C Moura
CIC/UnB	CIC/UnB

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 6 de junho de 2016

Dedicatória

Agradecimentos

Abstract

Languages formal especification are very effective to detect bugs and, more importantly, provides a deeper understanding of its underlying structure. Formal especification tools, such as Coq, have been target of great research interests in the previous years, one of the main reasons is for its proved soundness. The main goal of this paper is to bring together these two fronts and formally specify Feature Featherweight Java with Coq and prove its soundness.

Keywords: Formal Verification, Coq, Functional Programming, Programming Languages, Java, Featherweight Java, FFJ

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The main goal of this work is to develop an interpreter for Feature Featherweight Java [11], in the future it will be provided a formal proof of its soundness in Coq.

But before delving through the definition of FFJ we will first provide important formal definition to better understand the motivation of our work.

In the Chapter 2 we will provide the theoretical background behind programming languages, Chapter 3 we discuss our implementation decisions for the project. Chapter 4 some implementation details.

Chapter 2

Theoretical Introduction

2.1 Abstract Syntax

Robert Harper in [10] defines Programming Languages as means of expressing computations in a form comprehensible to both people and machines and the syntax of a language specifies the means by which various sorts of phrases (expressions, commands, declarations, and so forth) may be combined to form programs.

To better understand the syntax, it is usually introduced a few other auxiliary concepts, they are *concrete syntax*, *abstract syntax* and *binding*. The *surface*, or *concrete syntax* is concerned with how phrases are entered and displayed on a computer. The *surface syntax* is usually thought of as given by strings of characters from some alphabet. The *structural*, or *abstract syntax* is concerned with the structure of phrases, specifically how they are composed from other phrases. At this level a phrase is a tree, called *abstract syntax tree*, whose nodes are operators that combine several phrases to form another phrase. The *binding* structure of syntax is concerned with the introduction and use of identifiers how they are declared, and how declared identifers are to be used. At this level phrases are abstract binding trees, which enrich *abstract syntax* trees with the concepts of binding and scope.

An *abstract syntax tree*, or *ast* for short, is an ordered tree whose leaves are variables, and whose interior nodes are operators whose arguments are its children. A *variable* stands for an unspecified, or generic piece of syntax of specified sort. Ast's may be combined by an *operator*, which as an *arity* specifying the sort of the operator and the number and sorts of its arguments.

A variable is an unknown object drawn from some domain. The unknown can become known by substitution of a particular object for all occurrences of a variable in a formula, thereby specializing a general formaula to a particular instance. For example, in school, algebra variables range over real numbers, say, 7 for x to obtain $7^2+(2\times 7)+1$, which may be simplified according to the laws of arithmetic to obtain 64, which is indeed $(7+1)^2$ as it ought to be, since $x^2+2x+1=(x+1)^2$ in general, and hence in any particular case. The core idea is that *a variable is an unknown, or a placeholder, whose meaning is given by susbtition*

As an example, consider a language of arithmetic expressions built from numbers addition and multiplication. The abstract syntax of such a language consists of a single sort, Exp, generated by these operations:

- An operator num[n] of sort Exp for each n ∈ N;
- Two operators, plus and times, of sort Exp, each with two arguments of sort Exp.

The expression $2 + (3 * x)$, which involves a variable x, would represent by the ast $plus(num[2]; times(num[3]; x))$, which is written informally as $2 + (3 \times 4)$.

Abstract binding trees, or *abt's* enrich ast's with the means to introduce new variables and symbols, called a *binding*, with a specified range of significance, called its *scope.* The scope of a binding is an abt within which the bound identifier may be used, either as a placeholder (in the case of a variable declaration) or as the index of some operator (in the case of a symbol declaration). Thus the set of active identifiers may be larger within a subtree of an abt than it is within the surrounding tree. Moreover, different subtrees may introduce identifiers with disjoint scopes. The crucial principle is that any use of an identifier should be understood as a reference, or abstract pointer, to its binding. One consequence is that the choice of identifiers is immaterial, so long as we can always associate a unique binding with each use of an identifier.

For an example consider the expression $letxbea_1ina_2$, which introduces a variable x, for use within the expression a_2 to stand for the expression a_1. The variable x is bound by the let expression for use within a_2; any use of x within a_1 refers to a different variable that happens to have the same name. For example, in the expression $letxbe7inx + x$ ocurrences of x in the addition refer to the variable introduced by the let. On the other and in the expression $letbex * xinx + x$, occurrences of x within the multiplication refer to a different variable than occurring whithin the addition.

2.2 Inductive Definitions

An inductive definition consists of a set of *rules* for deriving *judgements*, or *assertion.* Judgments are statements about one or more abt's of some sort. The rules specify necessary and sufficient conditions for the validity of a judgement, and hence determine its meaning.

As examples of *judgements* we have:

Table 2.1: Judgements Example

n nat	n is a natural number
$n = n_1 + n_2$	n is the sum of n_1 and n_2
t type	t is a type
e : t	expression e has type t
e ⇓ v	expression e has type v

A judgment states that one or more abstract binding trees have a property or stand in some relation to one another.

2.3 Inference Rules

An *inductive definition* of a judgment form consists of a collection of *rules* of the form

$$\frac{J_1 \qquad \ldots \qquad J_k}{J}$$

in which J and $J_1, \cdots J_k$ are all judgments of the form being defined. The judgments above the horizontal line are called the *premises* of the rule, and the judgement below the line is called its *conclusion.* If a rule has no premises the rule is called an *axiom*; otherwise it is called a *proper rule.*

For example, the following rules form an inductive definition of the judgement form — nat:

$$\frac{}{zero\ nat}$$

$$\frac{a\ nat}{succ\ (a)\ nat}$$

These rules specify that a nat holds whenever either a is zero, or a is succ (b) where b nat for some b. Taking these rules to be exhaustive, it follows that a nat iff a is a natural number.

2.4 Derivation

To show that an inductively defined judgement holds, it is enough to exhibit a derivation of it. A derivation of a judgement is a finite composition of rules, starting with axioms and ending with that judgment. It may be thought of as a tree in which each node is a rule whose children are derivations of its premises. We sometimes say that a derivation of J is evidence for the validity of an inductively defined judgement J.

We usually depict derivations as trees with the conclusion at the bottom, and with the children of a node corresponding to a rule appearing above it as evidence for the premises of that rule.

For example, this is a derivation of succ (succ (succ (zero))) nat:

$$\frac{\frac{\frac{\frac{}{zero\ nat}}{succ\ (zero)}}{succ\ (succ\ (zero))}}{succ\ (succ\ (succ\ (zero)))}$$

To show that an inductively defined judgment is derivable, we need only find a derivation for it. There are two main methods for finding derivations, called *forward chaining*, or *bottom-up construction*, and *backward chaining*, or *top-down construction*. Forward chaining starts with the axioms and works forward towards the desired conclusion, whereas backward chaining starts with the desired conclusion and works backwards towards the axioms.

More precisely, forward chaining search maintains a set of derivable judgments and continually extends this set by adding to it the conclusion of any rule all of whose premises are in that set. Initially, the set is empty; the process terminates when the desired judgment occurs in the set. Assuming that all rules are considered at every stage, forward chaining will eventually find a derivation of any derivable judgment, but it is impossible (in general) to decide algorithmically when to stop extending the set and conclude that the desired judgment is not derivable. We may go on and on adding more judgments to the derivable set without ever achieving the intended goal. It is a matter of understanding the global properties of the rules to determine that a given judgment is not derivable.

Forward chaining is undirected in the sense that it does not take account of the end goal when deciding how to proceed at each step. In contrast, backward chaining is goal-directed. Backward chaining search maintains a queue of current goals, judgments whose derivations are to be sought. Initially, this set consists solely of the judgment we wish to derive. At each stage, we remove a judgment from the queue and consider all rules whose conclusion is that judgment. For each such rule, we add the premises of that rule to the back of the queue, and continue. If there is more than one such rule, this process must be repeated, with the same starting queue, for each candidate rule. The process terminates whenever the queue is empty, all goals having been achieved; any pending consideration of candidate rules along the way can be discarded. As with forward chaining, backward chaining will eventually find a derivation of any derivable judgment, but there is, in general, no algorithmic method for determining in general whether the current goal is derivable. If it is not, we may futilely add more and more judgments to the goal set, never reaching a point at which all goals have been satisfied.

2.5 Rule Induction

Because an inductive definition specifies the *strongest* judgment form closed under a collection of rules, we may reason about them by *rule induction*. The principle of rule induction states that to show that a property $a\ \mathcal{P}$ holds whenever $a\ \mathsf{J}$ is derivable, it is enough to show that $\mathcal{P}$ is *closed under*, or *respects*, the rules defining the judgment form J. More precisely, the property $\mathcal{P}$ respects the rule

$$\frac{a_1\mathsf{J}\ \ldots\ a_k\mathsf{J}}{a\mathsf{J}}$$

if $\mathcal{P}(a)$ holds whenever $\mathcal{P}(a_1),\ldots,\mathcal{P}(a_k)$ do. The assumptions $\mathcal{P}(a_1),\ldots,\mathcal{P}(a_k)$ are called the *inductive hypotheses*, and $\mathcal{P}(a)$ is called *inductive conclusion* of the inference.

The principle of rule induction is simply the expression of the definition of an inductively defined judgment form as the *strongest* judgment form closed under the rules comprising the definition. Thus, the judgment form defined by a set of rules is both (a) closed under those rules, and (b) sufficient for any other property also closed under those rules. The former means that a derivation is evidence for the validity of a judgment; the latter means that we may reason about an inductively defined judgment form by rule induction

When specialized to the nat rules, the principle of rule induction states that to show $\mathcal{P}(a)$ whenver $a\ nat$, it is enough to show:

1. $P(zero)$
2. for every a, if $\mathcal{P}(a)$, then $\mathcal{P}(succ(a))$.

The sufficiency of these conditions is the familiar principle of *mathematical induction*

2.6 Hypothetical Judgements

A *hypothetical judgment* expresses an entailment between one or more hypotheses and a conclusion. We will consider two notions of entailment, called *derivability* and *admissibility.* Both express a form of entailment, but they differ in that derivability is stable under extension with new rules, admissibility is not. A *general judgment* expresses the universality, or genericity, of a judgment. There are two forms of general judgment, the *generic* and the *parametric.* The generic judgment expresses generality with respect to all substitution instances for variables in a judgment. The parametric judgment expresses generality with respect to renamings of symbols.

The hypothetical judgment codifies the rules for expressing the validity of a conclusion conditional on the validity of one or more hypotheses. There are two forms of hypothetical judgment that differ according to the sense in which the conclusion is conditional on the hypotheses. One is stable under extension with more rules, and the other is not.

2.6.1 Derivability

For a given set $\mathcal{R}$ of rules, we define the *derivability* judgement, written $J_1, \ldots, J_k \vdash_{\mathcal{R}} K$, where each J_i and K are basic judgements, to mean that we may derive K from the *expansion* of the rules $\mathcal{R}$ with the axioms

$$\frac{}{J_1} \quad \cdots \quad \frac{}{J_k}$$

We treat the *hypotheses*, or *antecedents*, of the judgment $J_1, \ldots, J_k$ as "temporary axioms," and derive the conclusion, or consequent, by composing rules in $\mathcal{R}$. Thus, evidence for a hypothetical judgment consists of a derivation of the conclusion from the hypotheses using the rules in $\mathcal{R}$. We use capital Greek letters, usually Γ or Δ, to stand for a finite set of basic judgments, and write $\mathcal{R} \bigcup \Gamma$ for the expansion of $\mathcal{R}$ with an axiom corresponding to each judgment in Γ.The judgment $\Gamma \vdash_{\mathcal{R}} K$ means that K is derivable from rules $\mathcal{R} \bigcup \Gamma$, and the judgment $\vdash_{\mathcal{R}} \Gamma$ means that $\vdash_{\mathcal{R}} J$ for each in Γ. An equivalent way of defining $J_1, \ldots, J_n \vdash_{\mathcal{R}} J$ is to say that the rule

$$\frac{J_1 \quad \ldots \quad J_n}{J}$$

is *derivable* from *mathcalR*, which means that there is a derivation of J composed of the rules in $\mathcal{R}$ augmented by treating $J_1, \ldots, J_n$ as axioms.

For example, consider the derivability judgment.

$$a\ nat \vdash succ(succ(a))nat$$

This judgement is valid for any choice of object a, as shown by the derivation

$$\cfrac{\cfrac{\cfrac{}{a\ nat}}{succ\ (a)}}{succ\ (succ\ (a))}$$

which composes rules ??, starting with *a nat* as an axiom, and ending with $succ\ (succ\ (a))nat$.

It follows directly from the definition of derivability that it is stable under extension with new rules.

Theorem 2.1 (Stability). If $\Gamma \vdash_{\mathcal{R}} J$, then $\Gamma \vdash_{\mathcal{R} \cup \mathcal{R}'} J$

Proof. Any derivation of J from $\mathcal{R} \bigcup \Gamma$ is also a derivation from $(\mathcal{R} \cup \mathcal{R}') \cup \Gamma$, because any rule in $\mathcal{R}$ is also a rule in $\mathcal{R} \cup \mathcal{R}'$. □

Derivability enjoys a number of structural properties that follow from its definition, independently of the rules $\mathcal{R}$ in question.

Property 2.6.1 (Reflexivity). Every judgment is a consequence of itself:$\Gamma, J \vdash_{\mathcal{R}} J$.Each hypothesis justifies itself as conclusion.

Property 2.6.2 (Weakening). If $\Gamma \vdash_{\mathcal{R}} J$,then $\Gamma, K \vdash_{\mathcal{R}} J$.Entailment is not influenced by un-exercised options.

Property 2.6.3 (Transitivity). If $\Gamma, K \vdash_{\mathcal{R}} J$ and $\Gamma \vdash_{\mathcal{R}} K$, then $\Gamma \vdash_{\mathcal{R}} J$. If we replace an axiom by a derivation of it, the result is a derivation of its consequent without that hypothesis

Reflexivity follows directly from the meaning of derivability. Weakening follows directly from the definition of derivability. Transitivity is proved by rule induction on the first premise

2.7 Statics

Most programming languages exhibit a *phase distinction* between the static and dynamic phases of processing. The static phase consists of parsing and type checking to ensure that the program is well-formed; the dynamic phase consists of execution of well-formed programs. A language is said to be *safe* exactly when well-formed programs are well-behaved when executed.

The static phase is specified by a *statics* comprising a set of rules for deriving *typing judgments* stating that an expression is well-formed of a certain type. Types mediate the interaction between the constituent parts of a program by "predicting" some aspects of the execution behavior of the parts so that we may ensure they fit together properly at run-time. Type safety tells us that these predictions are correct; if not, the statics is considered to be improperly defined, and the language is deemed *unsafe* for execution.

2.7.1 Syntax

When defining a language, we shall be primarily concerned with its abstract syntax, specified by a collection of operators and their arities. The abstract syntax provides a systematic, unambiguous account of the hierarchical and binding structure of the language and is considered the official presentation of the language. However, for the sake of clarity, it is also useful to specify minimal concrete syntax conventions, without going through the trouble to set up a fully precise grammar for it. We will accomplish both of these purposes with a *syntax chart*, whose meaning is best illustrated by example. The following chart summarizes the abstract and concrete syntax of.

Typ τ	::=	**num**	**num**	numbers
		str	**str**	strings
Exp e	::=	x	x	variable
		num[n]	n	numeral
		str[s]	"s"	literal
		plus($e_1; e_2$)	$e_1 + e_2$	addition
		times($e_1; e_2$)	$e_1 * e_2$	multiplication
		cat($e_1; e_2$)	$e_1\hat{}e_2$	concatenation
		len(e)	$\|e\|$	length
		let($e_1; x.e_2$)	**let** x **be** e_1**in** e_2	definition

This chart defines two sorts,**Typ**, ranged over by τ, and **Exp**, ranged over by e. The chart defines a set of operators and their arities. For example, it specifies that the operator **let** has arity **(Exp, Exp, Exp)**, which specifies that it has two arguments of sort **Exp**, and binds a variable of sort **Exp** in the second argument.

2.7.2 Type System

The role of a type system is to impose constraints on the formations of phrases that are sensitive to the context in which they occur. For example, whether the expression **plus**(x; **num**[n]) is sensible depends on whether the variable x is restricted to have type **num** in the surrounding context of the expression. This example is, in fact, illustrative of the general case, in that the only information required about the context of an expression is the type of the variables within whose scope the expression lies. Consequently, the statics of **E** consists of an inductive definition of generic hypothetical judgments of the form

$$\overrightarrow{x}|\Gamma \vdash e : \tau$$

where $\overrightarrow{x}$ is a finite set of variables, and Γ is a *typing context* consisting of hypotheses of the form $x : \tau$, one for each $x \in \overrightarrow{x}$. We rely on typographical conventions to determine the set of variables, using the letters x and y to stand for them. We write $x \notin dom(\Gamma)$ to say that ther is no assumption in Γ of the form $x : \tau$ for any type τ, in which case we say that the variable x is *fresh* for Γ.

The rules defining the statics of **E** are as follows:

$$\frac{}{\Gamma, x : \tau \ \vdash \ x : \tau}\ \text{E\_VAR}$$

$$\frac{}{\Gamma \ \vdash \ \mathbf{str}[s] : \mathbf{str}}$$

$$\frac{}{\Gamma \ \vdash \ \mathbf{num}[n] : \mathbf{num}}$$

$$\frac{\Gamma \ \vdash \ e_1 : \mathbf{num}\ \Gamma \ \vdash \ e_2 : \mathbf{num}}{\Gamma \ \vdash \ \mathbf{plus}\ (e_1; e_2) : \mathbf{num}}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{num}\ \Gamma \vdash e_2 : \mathbf{num}}{\Gamma \vdash \mathbf{times}(e_1; e_2) : \mathbf{num}}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{str}\ \Gamma \vdash e_2 : \mathbf{str}}{\Gamma \vdash \mathbf{cat}(e_1; e_2) : \mathbf{str}}$$

$$\frac{\Gamma \vdash e : \mathbf{str}}{\Gamma \vdash \mathbf{len}(e) : \mathbf{str}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let}(e_1; x.e_2) : \tau_2}$$

In rule `E_VAR`, we tacitly assume that the variable x is not already declared in Γ. This condition may always be met by choosing a suitable representative of the α-equivalence class of the let expression.

It is easy to check that every expression has at most one type by *induction on typing*, which is rule induction applied to rules.

2.8 Type Safety

FJ and FFJ are *safe* (or, *type safe*, or *strongly typed*). Informally, this means that certain kinds of mismatches cannot arise during execution. For example, type safety for E states that it will never arise that a number is to be added to a string, or that two numbers are to be concatenated, neither of which is meaningful.

This means that evaluation cannot get stuck in a state for which no transition is possible, corresponding in implementation terms to the absence of “illegal instruction” errors at execution time. This is proved by showing that each step of transition can never “go off into the weeds”, and hence can never encounter an illegal instruction.

2.9 Formal Semantics

In this article we are concearned in studying the formal semantics of programming languages, i.e. we are interested in the tools to understand and reason how programs behave. More specifically the semantics of Feature Featherweight Java (FFJ). For historical reasons the semantics of programming languages is often viewed as consisteing of three strands:

- *Operational semantics* describes the meaning of the programming language by specifying how it executes on an abstract machine;
- *Denotational semantics* is a technique for defining the meaning of programming languages. At one time called “mathematical semantics” it uses the more abstract mathematical concepts of complete partial orders, continous functions and least fixed points;
- *Axiomatic semantics* tries to fix the meaning of a programming contruct by giving proof rules for it whithin a program logic. Thus axiomatic semantics emphasises proof of correctness right from the start.

It would however be wrong to view these three styles as in opposition to each other. They each have their uses. A clear operational semantics is very helpful in implementation. Axiomatic semantics for special kinds of languages can give strikingly elegant proof systems, useful in developing as well as verifying programs. Denotational semantics provides the deepest and most widely applicable thecniques, underpinned by a rich mathematical theory. Indeed, the different styles of semantics are highly dependent on each other.

For this we will need a nice understanding in logic and set theory.

2.10 Featherweight Java

The constructions of FFJ are based on the constructions of *Featherweight Java* (FJ), therefore, we must furst introduce the main characteristics of this language.

Featherweight Java was proposed by Igarash et. al., as a minimal core calculus for modeling Java’s type system [11]. The design of this language favors compactness over completeness, having just five forms of expression:

- Object creation;
- Method invocation;
- Field access;

- Casting;
- Variables

Several studies have introduced lightweight versions of Java The purpose is to omit the maximum number of Java's characteristics while still maintaining the main core, to ease the formalization model. There is a direct correspondence between FJ and a purely functional core of Java, in the sense that every FJ program is literally an executable Java program.

The design of FJ favors compactness over completeness almost obsessivelly. The aim is to omit as many features as possible, even assignment, while retaining the core features of Java typing. Assignment is used exclusively in field assignment inside the constructors using `this`.

FJ is only a little larger than Church's lambda calculus [4] or abadi and Cardelli's object calculus [2], and is significantly smaller than other precious formal models of class-based languages. Being smaller, FJ lets us focus on just a few key issues.

FJ has been successfully used to study Java extentions because it is so compact, we can focus attention on essential aspectos of the extension. Moreover, because the proof of soundness for pure FJ is very simple, a rigorous soundness proof for even a significant extension may remain manageable. The compactness of FJ is important because the soundness proof becomes very simple as the attention can be paid on essential aspects of the language, then a rigorous soundness proof for even a significant extension of FJ remains manageable. This is related to the authors's main goal: to make a proof of type soundness ("well-typed programs do not get stuck") as concise as possible, while still capturing the essence of the soundness argument for the full Java language.

The Java features omitted from FJ include assignment, interfaces, overloading, messages to `super`, *null* pointers, base types (`int`, `bool`, etc.), abstract method declarations, shadowing of superclass fields by subclass fields, access control (`public`, `private`, etc.), and exceptions. The features of Java that it does model include mutually recursive class definitions, method recursion through `this`, subtyping, and casting.

One key simplification in FJ is the omission of assignment. In essence, all fields and method parameters in FJ are implicitly marked `final`: it is assumed that an object's field are initialized by its constructor and never changed aftward. This restricts FJ to a "functional" fragment of Java, in which many common Java idioms, such as use of enumerations, cannot be represented. Nonetheless, this fragment is computationally complete (it is easy to encode the lambda calculus into it), and is large enough to include many useful programs like programs in Felleisen and Friedman's Java text [7] which use a purely functional style. Moreover, most of the tricky typing issues in Java are independent of assignment.

2.10.1 FJ explanation

In FJ, a program consists of a collection of class definitions, later on defined as the class table, and an expression to be evaluated. This expression correspods to the body of the `main` method in full Java. Here is an example of some typical class definitions in FJ and a possible expression for this definitions.

```
class A extends Object {
    A() { super(); }
}

class B extends Object {
    B() { super(); }
}

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

We always

1. Include the supertype (even when it is `Object`);
2. Write out the constructor (even for the trivial classes `A` and `B`;
3. Write the receiver for the field access (as in `this.snd`) or a method invocation, even when the receiver is `this`

Constructor always take the same stylized form: there is one parameter for each field, with the same name as the field; the `super` constructor is invoked on the fields of the supertype; and the remaining fields are initialized to the corresponding parameters. In this example the supertype is always `Object`, which has no fields, so the invocations of `super` have no arguments. Constructors are the only place where `Super` or = appears in an FJ program. Since FJ provides no side-effecting operations, a method body always consists of return followed by an expression, as in the body of `setfst()`.

In the context of the above definitions, the expression

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

evaluates to the expression

```
new Pair(new B(), new B())
```

There are five forms of expression in FJ

1. Object constructors, like `new A()`, `new B()` and `new Pair(e1, e2)`;
2. Method invocation, like `e3.setfst(e4)`;
3. Field access, like `this.snd` inside the body of `setfst`;

4. Variabes, like the occurencies of `newfst` and `this`;
5. Cast, like the expression

   ```
   ((Pair)new Pair(new Pair(new A(), newB()), new A()).fst).snd
   ```

   which evaluates to

   ```
   new B()
   ```

In Java, we may prefix a field or parameter declaration with the keyword `final` to indicate that it cannot be reassigned. However, since FJ has no assignments, there is no need for such a keyword.

Given that FJ functions has no side effects evaluation gets a lot easier to be formalized. Since it can be interely done within the syntax and no need for a heap mechanism whatsoever. With the absence of side effects, the order in which expressions are evaluated does not affect the final outcome of the computation (except when the program does not terminates).

There are three basic computation rules: one for field access, one for method invocation, and one for casts. It is recursive with the stop condition on the `new` expression. That is to say "everything is an object" here in FJ.

The following example shows the rule for field access in action.

```
new Pair(new A(), newB()).snd → new B()
```

Bringing into attention the object constructors, it has one parameter for each field, in the same order that the fields are declared, also taking into account its superclasses. Invocation of methods reduces to the body with the formal parameter replaced by the actual, and the special variable `this` replaced by the receiver object. This is similar to the beta rula of the lambda calculus. The key differences are the fact that the class of the receiver determines where to look for the ody (supporting method override), and the substitution of the receiver for `this` (suporting method override), and the substitution of the receiver for `this`(supporting "recursion through self"). In FJ, as in the lambda calculus and the pure Abadi-Cardeli calculus, if a formal parameter appears more than once in the body it may lead to duplication of the actual, but since there are no side effects this causes no problems.

Here is an example of a reduction of a cast:

```
(Pair)new Pair(new A(), new B()) → new Pair(new A(), new B())
```

When a cast-expression is reduced to an object (i.e. a new-expression), it is easy to check that the class of the constructor is a subclass of the target of the cast. If so, as in this case, then the reduction removes the cast. Otherwise, is in `(A)new B()`, then no rule applies and the computation is *stuck*, raising a runtime error.

There are three ways computation may get stuck: an attempt to access a field not declared for the class; an attempt to invoke a method not declared for the class; or an attempt to cast to something other than superclass of an object's runtime class. The authors provides a proof that the first two never happen in well-typed programs, and the third never happens in well-typed programs that contain no downcasts (and no "stupid casts", which is explained in details below).

As usual, it is allowed to apply reductions to any subexpressions of an expressions. Below is provided a more complex example, underlining the step of the reduction being applied.

$$\begin{aligned}
&\texttt{(Pair)}\underline{\text{new Pair(new Pair(new A(), new B()), new A()).fst}}\texttt{).snd}\\
\rightarrow\ &\underline{\text{(Pair)new Pair(new A(), new B())}}\texttt{.snd}\\
\rightarrow\ &\underline{\text{new Pair(new A(), new B()).snd}}\\
\rightarrow\ &\textbf{new}\ \texttt{B()}
\end{aligned}$$

In the original paper [11] it is provided a proof of a type soundness result for FJ: if a well-typed expression `e` reduces to a normal form, an expression that cannot reduce any further, than the normal form is either a well-typepd value (an expression consisting only of `new`), whose type is a subtype of the type of `e`, or stuck at a failing typecast.

Finally we present the following code with the implementation in FJ of the Natural numbers using only the constructor for zero and the successor at listing 2.1, known as the Peano Numbers [12]

Listing 2.1: Peano Encoding for Naturals in FJ

```
class Nat extends Object{
    Nat(){super();}

    Nat add(Nat rhs){
        return rhs.add(this);
    }
}

class O extends Nat{
    O(){super();}

    Nat add(Nat rhs){
        return rhs;
    }
}

class S extends Nat{
    Nat num;

    S(Nat num){
        super();
        this.num=num;
    }

    Nat add(Nat rhs){
        return this.num.add(new S(rhs));
    }

}
new S(new S(new O())).add(new S(new O()))
```

With this informal introduction we may now proceed to the formal definitions.

2.10.2 Syntax

To present the abstract syntax of FJ, first we need to define the meaning of the following metavariables, which will apear frequently in the rules of FJ's grammar: A, B, C, D and E range over class names; f and g range over field names; m ranges over method names; x ranges over variables; d and e range over expressions; L ranges over class declarations; K range over constructor declarations and M ranges over method declarations.

The abstract syntax of FJ class declarations, constructor declarations, method declarations and expressions is given at Table 2.2.

Table 2.2: Abstract Syntax

$$\begin{array}{lcl}
L & ::= & \mathit{class}\ C\ \mathit{extends}\ C\ \{\bar{C}\ \bar{f};\ K\ \bar{M}\} \\
K & ::= & C\ (\bar{C}\ \bar{f})\ \{\mathit{super}\ (\bar{f});\ \mathit{this}.\bar{f} = \bar{f};\} \\
M & ::= & C\ m\ (\bar{C}\ \bar{x})\ \{\mathit{return}\ e;\} \\
e & ::= & x \mid e.f \mid e.m\ (\bar{e}) \mid \mathit{new}\ C\ (\bar{e}) \mid (C)\ e
\end{array}$$

The variable `this` is assumed to be included in the set of variables, but it cannot be used as the name of an argument to a method.

It is written $\bar{f}$ as a shorthand for a possibly empty sequence $f_1, \ldots, f_n$ (and similary for $\bar{C}$, $\bar{x}$, $\bar{e}$ etc.) and $\bar{M}$ as a shorthand for $M_1 \ldots M_n$ (with no commas).

A class table CT is a mapping from class names C to class declarations L. A program is a pair (CT, e) of a class table and an expression. In FJ, every class has a superclass declared with extends, with the exception of `Object`, which is taken as a distinguished class name whose definition does *not* appear in the class table. The class table contains the subtype relation between all the classes. From the Table 2.3, one can infer that subtyping is the reflexive and transitive closure of the immediate subclass ralation given by the `extends` clauses in CT.

Table 2.3: Subtyping

$$C\ <:\ C \qquad\qquad \frac{C <: D \qquad C <: E}{C <: E} \qquad\qquad \frac{\mathit{class}\ C\ \mathit{extends}\ D\ \{\ \ldots\ \}}{C\ <:\ E}$$

The function *fields* defined at Table 2.4 returns a list of all the fields declared at the current class definition, as well as the fields declared at the superclass of it. It returns an empty list in the case of `Object`.

The method declaration $D\ m\ (\bar{C}\ \bar{x})\ \{\mathit{return}\ e;\}$ introduces a method name m with result type D and parameters $\bar{x}$ of types $\bar{C}$. The body of the method is the single statement $\mathit{returne};$. The variables x and the special variable *this* are bound in e. The rules that define these functions are ilustrated in Tables 2.5 and 2.6.

The given class table is assumed to satisfy the following conditions:

- $CT\ (C) = \mathit{class}\ C \ldots$ for every $C \in \mathit{dom}(CT)$

Table 2.4: Field lookup

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

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

Table 2.5: Method type lookup

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

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

Table 2.6: Method body lookup

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

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

- `Object`$\notin \mathit{dom}(CT)$
- for every class name C (except `Object`) appearing anywhere in CT, we have $C \in \mathit{dom}(CT)$
- there are no cycles in the subtype relation induced by CT, i.e., the relation $<:$ is antisymmetric

2.10.3 Typing

The typing rules for expressions are in Table 2.7. An environment Γ is a finite mapping from variables to types, written $\bar{c} : \bar{C}$. The typing judgment for expressions has the form $\Gamma \vdash e : C$, read "in the environment Γ, expression e has type C".

2.10.4 Computation

The reduction relation is of ther form $e \to e'$, read "expression e reduces to expression e' in one step", We write $\to *$ for the reflexisive and transitive closure of $\to$.

The reduction rules are given in 2.8. There are three reduction rules, one for field acess, one for method invocation, and one for casting. These were already explained above. We write $[\bar{d} = \bar{x}, e = y]e_0$ for the result of replacing x_1 by d_1, x_2 by $d_2, \ldots, x_n$ by d_n, and y by e in the expression e_0.

Table 2.7: Expression typing

$$\Gamma \vdash x : \Gamma(x) \tag{T-Var}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad fields\ (C_0) = \bar{C}\ \bar{f}}{\Gamma \vdash e_0.f_i : C_i} \tag{T-Field}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad mtypes\ (m,\ C_0) = \bar{D} \to C \qquad \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C}\ <:\ \bar{D}}{\Gamma \vdash e_0.m(\bar{e}) : C} \tag{T-Invk}$$

$$\frac{fields(C) = \bar{D}\ \bar{f} \qquad \Gamma \vdash \bar{e} : \bar{C} \qquad \bar{C}\ <:\ \bar{D}}{\Gamma \vdash new\ C(\bar{e}) : C} \tag{T-New}$$

$$\frac{\Gamma \vdash e_0 : D \qquad D\ <:\ C}{\Gamma \vdash (C)\ e_0 : C} \tag{T-UCast}$$

$$\frac{\Gamma \vdash e_0 : D \qquad C\ <:\ D \qquad C \neq D}{\Gamma \vdash (C)\ e_0 : C} \tag{T-DCast}$$

$$\frac{\Gamma \vdash e_0 : D \qquad C\ \not<:\ D \qquad D\ \not<:\ C \qquad stupid\ warning}{\Gamma \vdash (C)\ e_0 : C} \tag{T-SCast}$$

Notice again that with the absense of side effects, there is no need of stack or heap for variable binding.

The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules.

Table 2.8: Expression computation

$$\frac{fields\ (C) = \bar{C}\bar{f}}{(new\ C(\bar{e})).f_i \to e_i} \tag{R-Field}$$

$$\frac{mbody\ (m, C) = \bar{x}.e_0}{(new\ C\ (\bar{e})).m\ (\bar{d}) \to [\bar{d}/\bar{x}, new\ C\ (\bar{e})/this]e_0} \tag{R-Invk}$$

$$\frac{C <: D}{(D)(new\ C\ (\bar{e})) \to new\ C\ (\bar{e})} \tag{R-Cast}$$

2.11 Feature Featherweight Java

This language extends FJ with constructions of *feature-oriented programming* (FOP) by new languages constructs for feature composition, according evaluation and type rules.

Table 2.9: Congruence

$$\frac{e_0 \to e'_0}{e_0.f \to e'_0.f} \qquad \text{(RC-Field)}$$

$$\frac{e_0 \to e'_0}{e_0.m\ (\bar{e}) \to e'_0.m\ (\bar{e})} \qquad \text{(RC-Invk-Recv)}$$

$$\frac{e_i \to e'_i}{e_0.m\ (\ldots, e_i, \ldots) \to e'_0.m\ (\ldots, e_i, \ldots)} \qquad \text{(RC-Invk-Arg)}$$

$$\frac{e_i \to e'_i}{new\ C\ (\ldots, e_i, \ldots) \to new\ C\ (\ldots, e'_i, \ldots)} \qquad \text{(RC-New-Arg)}$$

$$\frac{e_0 \to e'_0}{(C)e_0 \to (C)e'_0} \qquad \text{(RC-Cast)}$$

FFJ was proposed by Apel et. al., in [3].

Feature-oriented programming (FOP) aims at the modularization of software systems in terms of features [13]. A *feature* implements a stakeholder's requirement and is tipically an increment in program functionality. Different variants of a software system are distinguished in terms of their individual features.

Chapter 3

Implementation Decisions

In this chapter we will discuss the tools chosen for implementing the project and why. In the first section we explain why haskell and how it's features may help us to accomplish our first goal which is building an interpreter. In the second section we give a mild introduction to BNF. The third section is about a very important haskell feature which is called monads, and allows to simulate side effects. The fourth section is about coq which will be used to provide the soundess proof later on.

3.1 DSLs

The first problem to address was to find a suitable programming language to model Domain Specific Languages (DSL). In short DSLs are programming languages specialized to a particular application domain. In our case our domain will be the interpreter and our specialization will be an *ast*. Haskell is a great host for DSLs because of flexible overloading, a powerful type system, and lazy semantics [9].

3.2 Haskell

Haskell is a general purpose programming language. It was designed without any application niche in mind, i.e., it does not favour one problem domain over others. While at its core, Haskell encourages, as a default, a pure, lazy style of functional programming, but it's not the only option. It also supports the more traditional models of procedural code code and strict evaluation. The focus of the language is on writing statically typed programs [1].

3.2.1 Quick tutorial

In Haskell is all about *types*. Types, like those in other languages, are constraining summaries of structural values. For example, in Haskell `Bool` is tye type of the values `False`; `Int` is the type of machinezed sized words; `Double` is the type of double preceision IEEE floating-point values; and this list goes on the same manner as C, C+++, Java, and other traditional languages. All of these type names in haskell start with an upper case letter.

On top of these basic types, Haskell has two syntactic forms for expressing compund types. First, pairs, triples, and larger structures can be written using tuple-syntax, comma-separated types inside parentheses. Thus, (`Int`, `Bool`) is a structures with both an `Int` and a `Bool` component. Second, lists have a syntactic shortcut, using square brackets. Thus, `Int`] is a list of `Int`.

There are also other container types. A container that may contain one Int has the type `Maybe Int`, which is read Maybe of Int. These container names also start with upper case letters.

Types can be nested to any depth. For example, you can have a [(`Maybe`(`Int`, `Bool`))], read as list of maybe of (Int and Bool).

Polymorphic values are expressed using lowercase letters and play a similar role to `void*` pointers in C and polymorphic arguments in the Java generics facility. These polymorphic values can have constraints expressed over them, using the haskell equivalent of an object hierarchy.

Finally, a function is written using an arrow from argument type to result type. Thus, in Haskell, a function that takes a list and returns a list is writen as: `[a] -> [a]`.

Here is an example of a Haskell function:

```
sort :: (Ord a) => [a] -> [a]
sort [] = []
sort (x:xs) = sort before
    ++ [x] ++ sort after
  where
    before = filter (<= x) xs
    after = filter (>x) xs
```

This function sorts a list using a variant of quicksort which the pivot is the first element of the list:

- The first line of the type for `sort`. This is $\forall a$, such that a can be ordered (i.e. admits comparisons like `<=`); the function takes and return a list of such as.
- The second line says that an empty list is already sorted.
- The remaining lines state that a non-empty list can be sorted by taking the first and rest of the list (called `x` and `xs`, respectively), sorting the values before this pivot and after this pivot, and concatenating these intermediate values together.
- Finally, intermediate values can be named using the where syntax; in this case the values of `before` and `after`.

Haskell is a concise and direct language. Structures in Haskell are denoted using types, constructed and deconstructed, but never updated. For example, the `Maybe` type can be defined using two Constructors, `Nothing` and `Just`:

```
data Maybe where
    Nothing :: Maybe a
    Just    :: a -> Maybe a
```

`Nothing` is a `Maybe` of anything; `Just`, with an argument, is a maybe with the type of the argument. These constructors can be used to construct and deconstruct structures, but there is never any updating; all structures in Haskell are immutable.

It is possible to give specific types extra powers, such as equality and comparison, using the class-based overloading system. The `Maybe` type, for example, can be given the ability to test for equality, using an instance:

```
instance Eq a => Eq (Maybe a)
  where
    Just a == Just b = a == b
    Nothing == Nothing = True
    _ == _ = False
```

This states that for any type that can be tested for equality, you can also check `Maybe` of the same type. You take the `Maybe` apart, using pattern matching on `Just`, to check the internal value.

3.3 BNFC

Backus-Naur Form, also know as Context-Free Grammars, *Converter* (BNFC) is a tool developded to help the design and implementation of a new programming language [14]. It uses the grammar formalism named *Labelled BNF* (LBNF) defined in [8]. Given a grammar written in LBNF, the BNFC produces a complete compiler front end (with no type checking), i.e. a lexer, a parser, and an abstract syntax definition. Moreover, it also provides a pretty-printer and a language specification LaTeXas well as a template file for the compiler back end. Since LBNF is purely declarative, these files can be generated in any programming language that supports appropriate compiler front-end tools, in particular, Haskell.

3.3.1 Basic LBNF

Briefly, an LBNF grammar is a context-free grammar where every rule is given a label. The label is used for constructing a syntax tree whose subtrees are given by the nonterminals of the rule, in the same order.

As an example of LBNF, consider a triple of rules defining addition expressions with "1".

```
EPlus. Exp ::= Exp "+" Num ;
ENum. Exp ::= Num ;
NOne. Num ::= 1 ;
```

From an LBNF grammar, the BNFC extracts an abstract syntax and a concrete syntax. In Haskell, for instance, the abstract syntax is implemented asa system of datatype definitions.

```
data Exp = EPlus Exp Exp | ENUm Num
data Num = NOne
```

3.4 Monads

The functional programming community divides into two camps [15]. *Pure* languages, such as Haskell, are lambda calculus pure and simple. *Impure* languages extends lambda calculus with a number of possible effects. Pure languages are easier to reason about and may benefit from lazy evaluation, while the impure ones offer efficiency benefits and sometimes make possible a more compact mode of expression.

It has been discussed the difficulties of implementing a evaluator in a pure functional language, it basically envolves:

- To add error handling to it, it is necessary to modify each recursive call to check for and handle errors appropriately, a work easily done with exceptions in an impure language
- To add a count of operations performed, we must modify each recursive call to pass around such counts appropriately, this would be easily done in an impure language using global variables
- To add an execution trace to it, we need to modify each recursive call to pass aroung such traces, also easily done in an impure language

The solution: *monad*.

3.5 Coq

Coq is a formal proof management system based on the Calculus of Inductive Constructions [5]. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment that combines superior order logic and a dependently typed functional language. Coq allows you to, with this language [6]:

- define functions or predicates, that can be efficiently evaluated
- declare mathemematical theorems and software specifcations
- develop formal proofs
- check these proofs by a certificate “kernel”
- extract certified programs to some functional programming languages

Chapter 4

Implementation Details

4.1 BNFC

To generate the *ast*, we used the tool bnfc [8], and used the following specification to automatically generate the data types used.

```
-- Class Declaration

CProgram . Program ::= [ClassDecl] Exp;

CDecl . ClassDecl ::= "class" Id "extends" ClassName "{" [
    ↪ FieldDecl] Constructor [MethodDecl] "}" ;

-- Fields Declaration
FDecl . FieldDecl ::= ClassName Id ";" ;

-- Constructor Declaration
KDecl . Constructor ::= Id "(" [FormalArg] ")" "{" "super" "(" [
    ↪ Argument] ")" ";" [Assignment] "}" ;

FArg . FormalArg ::= ClassName Id;
Arg . Argument ::= Id ;
Assgnmt . Assignment ::= "this" "." Id "=" Id ";" ;

-- Method Declaration
MDecl . MethodDecl ::= ClassName Id "(" [FormalArg] ")" "{" "
    ↪ return" Exp ";" "}" ;

ExpVar . Exp ::= Var ;
ExpFieldAccess . Exp ::= Exp "." Id ;
ExpMethodInvoc . Exp ::= Exp "." Id "(" [Exp] ")" ;
ExpCast . Exp ::= "(" ClassName ")" Exp ;
ExpNew . Exp ::= "new" Id "(" [Exp] ")" ;

This . Var ::= "this" ;
```

```
VarId . Var ::= Id ;

token Id (letter (letter | digit | '_')*) ;

ClassObject . ClassName ::= "Object" ;
ClassId . ClassName ::= Id ;

separator ClassDecl "" ;
separator FieldDecl "" ;
separator MethodDecl "" ;
separator FormalArg "," ;
separator Argument "," ;
separator Assignment "" ;
separator Exp "," ;
```

4.2 The Core

A core set of functionalities were implemented in order to allow set up a better programing environment. These functions, data and classes would not fit into the Syntax, Dynamics nor the Type modules, but actually is the core of them all.

They are the Result data, which allow to program using the do notation, the Referable class which provides a staple implementation for searching in a list.

4.2.1 Result Data

The `Result` data, which instanciates Monad, this will allow us to program using the do notation, and to set up fairly nice error messages using the raise function.

Listing 4.1: Result Data Implementation

```haskell
type Exception = String

data Result a = Ok a
        | Ex Exception
        deriving(Eq, Show, Ord)

raise :: Exception -> Result a
raise x = Ex x

instance Monad Result where
        return a = Ok a
        {-(>>=) :: MOut a -> (a -> MOut b) -> MOut b-}
        (>>=) m f = case m of
                                (Ok a) -> f a
                                (Ex e) -> raise e
```

Most of our implementation functions will return some `Result a`

4.2.2 Referable Class

The `Referable` class provides a default implementation for key searching in a list. All we need now is to instantiate our classes as a Referable to use this nice and polimorphic class. One can think about such a class as an abstract class in Java, i.e., every class that `extends` a given abstract class, will `inherit` all of its functions.

Listing 4.2: The Referable Class Implementation

```haskell
class Referable a where
  ref  :: a -> Id
  find :: Id -> [a] -> Result a

  find key list = -- defaul implementation
    case [x | x <- list , key == (ref x)] of
      []    -> raise \$ "there is no such a key " ++ show key ++
        ↪  " in the list."
      (x:_) -> return x
```

In listing 4.3 we provide an example of a Referable instantiation. In the first line we say that the Var data is an instance of Referable, and we must provide an implementation for the function ref, which must recieve a Var and return an Id, as specified above.

Listing 4.3: A referable Instance

```haskell
instance Referable Var where
  ref This = Id "this"
  ref (VarId id) = id
```

Now we are set to use the find function for any `List of Var`.

References

[1] *Real world haskell.* O'REILLY, 2008. 19

[2] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 396–409. ACM, 1996. 11

[3] Sven Apel, Christian Kästner, and Christian Lengauer. Feature featherweight java: A calculus for feature-oriented programming and stepwise refinement. In *In Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE).* ACM Press, 2008. 18

[4] Hendrik Pieter Barendregt. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984. 11

[5] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer Science & Business Media, 2013. 22

[6] ADT Coq. The coq proof assistant. `http://https://coq.inria.fr/`, 2011. 22

[7] Matthias Felleisen and Daniel P Friedman. *A little Java, a few patterns.* MIT press, 1998. 11

[8] Markus Forsberg and Aarne Ranta. The labelled bnf grammar formalism. *BNF Converter Version 2.2*, 2005. 21, 23

[9] Andy Gill. Domain-specific languages and code synthesis using haskell. *Queue*, 12(4):30, 2014. 19

[10] Professor Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, New York, NY, USA, 2012. 2

[11] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999. 1, 10, 14

[12] Giuseppe Peano. *Arithmetices principia: nova methodo.* Fratres Bocca, 1889. 14

[13] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *In Proc. Europ. Conf. Object-Oriented Programming, volume 1241 of LNCS*, pages 419–443, 1997. 18

[14] Aarne Ranta. Bnf converter tutorial. `http://bnfc.digitalgrammars.com/tutorial.html`, 2007. 21

[15] Philip Wadler. Monads for functional programming. *Advanced Functional Programming*, 1995. 22