# Towards an Extrinsic Formalization of Featherweight Java in Agda

Samuel S. Feitosa
*Departamento de Informática*
*Instituto Federal de Santa Catarina*
Caçador, SC - Brazil
samuel.feitosa [at] ifsc.edu.br

Rodrigo G. Ribeiro
*Prog. Pós Grad. em Ciência da Computação*
*Universidade Federal de Ouro Preto*
Ouro Preto, MG - Brazil
rodrigo.ribeiro [at] ufop.edu.br

Andre R. Du Bois
*Prog. de Pós Grad. em Computação*
*Universidade Federal de Pelotas*
Pelotas, RS - Brazil
dubois [at] inf.ufpel.edu.br

***Abstract*—Featherweight Java is one of the most popular calculi which specify object-oriented programming features. It has been used as the basis for investigating novel language functionalities, as well as to specify and understand the formal properties of existing features for languages in this paradigm. However, when considering mechanized formalization, it is hard to find an implementation for languages with complex structures and binding mechanisms as Featherweight Java. In this paper we formalize Featherweight Java, implementing the static and dynamic semantics in Agda, and proving the main safety properties for this calculus.**

***Index Terms*—Featherweight Java, Mechanized Semantics, Type Safety**

## I. Introduction

Currently, Java is one of the most popular programming languages [1]. It is a general-purpose, concurrent, strongly typed, class-based object-oriented language. Since its release in 1995 by Sun Microsystems, and later acquisition by Oracle, Java has been evolving over time, adding features and programming facilities in its new versions. For example, in a recent major release of Java, new features such as lambda expressions, method references, and functional interfaces, were added to the core language, offering a programming model that fuses the object-oriented and functional styles [2].

Since a programming language evolves, it is important to have mechanisms to ensure that certain behaviors and desired properties are maintained after changing the language's structure and the compiler or interpreter implementation. One way to do that is to formalize the language (or subset of it) in a proof assistant, such as Agda, Coq, or Isabelle, providing formal proofs of the desired properties. Although mechanized proof assistants are powerful tools, proof development can be difficult and time-consuming [3].

In this context, this paper discusses the steps to formalize Featherweight Java (FJ) [4] in Agda, a dependently-typed functional programming language based on Martin-Löf intuitionistic type theory [5]. FJ is a small core calculus with a rigorous semantic definition of the main core aspects of Java. The motivations for using the specification of FJ are that it is very compact, and its minimal syntax, typing rules, and operational semantics fit well for modeling and proving properties for the compiler and programs. We adopt the most used method for proving safety of a programming language: the syntactic approach (sometimes called extrinsic) proposed by Wright and Felleisen [6]. Using this technique, we define first the syntax, and then relations to express both the typing judgments (static semantics), and the evaluation through reduction steps (dynamic semantics). We prove the common theorems of *progress* and *preservation* to link the static and dynamic semantics, guaranteeing that a well-typed term will not get stuck, i.e., it should be a value or be able to take another reduction step, preserving the intended type. As far as we know, this is the first attempt to formalize an extrinsic version of FJ in Agda. Filling this gap, we provide to the interested reader the source-code which can be used to better understanding the approach and Agda, as well as to be extended for future developments.

More concretely, we make the following contributions:

- We specify the static and dynamic semantics of FJ (class table and expressions) in Agda using the syntactic approach [6].
- We prove that the specification is sound, i.e., we can show that the proposed theorems of *progress* and *preservation* hold.
- We define a function to evaluate well-typed terms, by repeating the application of the *progress* and *preservation* proofs [7].

The remainder of this text is organized as follows: Section II summarizes the FJ proposal. Section III shows how we represent types, how we model the class table and expressions, and the specification of the static and dynamic semantics of FJ in Agda. Section IV discusses the proof steps to guarantee type safety of the studied calculus. Section V presents the steps to define evaluation through repeated applications of the *progress* and *preservation* theorems. Section VI discusses related work. Finally, we present the final remarks in Section VII.

All source-code presented in this paper has been formalized in Agda version 2.6.0 using Standard Library 1.0. We present here parts of the Agda code used in our definitions, not necessarily in a strict lexically-scoped order. Some formal proofs were omitted from the text for space reasons, and also to not distract the reader from understanding the high-level structure of the formalization. In such situations we give just proof sketches and point out where all details can be found

in the source code. All source code produced, including the LaTeX source of this paper, are available on-line [8].

## II. Featherweight Java: a Refresher

Featherweight Java (FJ) [4] is a minimal core calculus for Java, in the sense that as many features of Java as possible are omitted, while maintaining the essential flavor of the language and its type system. FJ is to Java what $\lambda$-calculus is to Haskell. It offers similar operations, providing classes, methods, attributes, inheritance and dynamic casts with semantics close to Java's. The Featherweight Java project favors simplicity over expressivity and offers only five ways to create terms: object creation, method invocation, attribute access, casting and variables [4]. This fragment is large enough to include many useful programs.

A program in FJ consists of the declaration of a set of classes and an expression to be evaluated, which corresponds to Java's main method. The following example shows how classes can be modeled in FJ. There are three classes, A, B, and Pair, with constructor and method declarations.

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

In the following example we can see two different kinds of expressions: new A(), new B(), and new Pair(...) are *object constructors*, and .setfst(...) refers to a *method invocation*:

```
new Pair (new A () , new B ()) ∘ setfst (new B ());
```

FJ semantics provides a purely functional view without side effects. In other words, attributes in memory are not affected by object operations [9]. 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 [4]. As the language does not allow side effects, it is possible to formalize the evaluation directly on FJ terms, without the need for auxiliary mechanisms to model the heap [9].

### A. Syntax and Auxiliary Functions

The abstract syntax of FJ is given in Figure 1, where `L` represents classes, `K` defines constructors, `M` stands for methods, and `e` refers to the possible expressions. The metavariables `A`, `B`, `C`, `D`, and `E` can be used to represent class names, `f` and `g` range over field names, `m` ranges over method names, `x` and `y` range over variables, `d` and `e` range over expressions. Throughout this paper, we write $\overline{C}$ as shorthand for a possibly empty sequence $C_1, ..., C_n$ (similarly for $\overline{f}$, $\overline{x}$, etc.). An empty sequence is denoted by $\bullet$, and the length of a sequence $\bar{\mathtt{x}}$ is written $\#\bar{\mathtt{x}}$. We use $\Gamma$ to represent an environment, which is a finite mapping from variables to types, written $\overline{x} : \overline{T}$, and we let $\Gamma$(x) denote the type C such that x: C $\in \Gamma$. We slightly abuse notation by using set operators on sequences. Their meaning is as usual.

**Syntax**

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

Fig. 1. Syntactic definitions for FJ.

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

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

$$C <: C$$

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

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

Fig. 2. Subtyping relation between classes.

The authors in [4] also proposed some auxiliary definitions for working in the typing and reduction rules. These definition are given in Figure 3. The rules for *field lookup* demonstrate how to obtain the fields of a given class. If the class is `Object`, an empty list is returned. Otherwise, it returns a

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

**Field lookup**

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

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

**Method type lookup**

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

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

**Method body lookup**

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

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

Fig. 3. Auxiliary definitions.

### B. Typing and Reduction Rules

This section presents how the typing rules of FJ are used to guarantee type soundness, i.e., well-typed terms do not get stuck, and the reduction rules showing how each step of evaluation should be processed for FJ syntax. Figure 4 shows in the left side, the typing rules for expressions, and in the right side, it shows first the rules to check if methods and classes are well-formed, then the reduction rules for this calculus. We omit here the congruence rules, which can be found in the original paper [4].

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

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

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

There are only three computation rules, indicating which expressions can be used in the main program. The first rule `R-Field` formalizes how to evaluate an *attribute access*. Similarly to the typing rule `T-Field`, it uses the function *fields*, and matches the position $i$ of the field $\texttt{f}_i$ in the resulting list, returning the value $\texttt{v}_i$, which refers to the value in the position $i$ of the actual parameter list. The second rule `R-Invk` shows the evaluation procedure for a *method invocation*, where firstly it obtains the method body expression `m` of class `C` through the function *mbody*, and then performs substitution of the actual parameters and the special variable `this` in the body expression, similar to a beta reduction on $\lambda$-calculus. The last rule `R-Cast` refers to *cast processing*, where the same subexpression `new C(ē)` is returned in case the subject class `C` is subtype of the target class `D`. There are also five congruence rules[1] (omitted from Figure 4), which are responsible for the intermediary evaluation steps for the proposed small-step semantics.

The FJ calculus is intended to be a starting point for the study of various operational features of object-oriented

[1]The congruence rules omitted from the text can be found in p. 407 of [4].

**Expression typing**

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

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

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

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

$$\frac{\Gamma \vdash e_0 : D \qquad D <: C}{\Gamma \vdash (C)\ e_0 : C} \quad \text{[T-UCast]}$$

$$\frac{\Gamma \vdash e_0 : D \qquad C <: D \qquad C \neq D}{\Gamma \vdash (C)\ e_0 : C} \quad \text{[T-DCast]}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : D \qquad C \not<: D \qquad D \not<: C \\ \mathit{stupid\ warning}\end{array}}{\Gamma \vdash (C)\ e_0 : C} \quad \text{[T-SCast]}$$

**Method typing**

$$\frac{\begin{array}{c}\bar{x} : \bar{C},\ \text{this} : C \vdash e_0 : E_0 \qquad E_0 <: C_0 \\ \text{class } C \text{ extends } D\ \{...\} \\ \text{if } \mathit{mtype}(m, D) = \bar{D} \to D_0, \text{ then } \bar{C} = \bar{D} \text{ and } C_0 = D_0\end{array}}{C_0\ m(\bar{C}\ \bar{x})\ \{\ \text{return } e_0;\ \}\ \text{OK in } C}$$

**Class typing**

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

**Evaluation**

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

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

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

Fig. 4. Typing and evaluation rules.

programming in Java-like languages, being compact enough to make rigorous proofs feasible. Besides the rules for evaluation and type-checking, Igarashi et al. [4] present (paper) proofs of type soundness for FJ.

## III. Mechanization of Featherweight Java

This section presents our formalization of a large subset of FJ (without casts) using the usual syntactic (extrinsic) approach proposed by Wright and Felleisen [6]. We use Agda, an advanced programming language based on Type Theory. Agda's type system is expressive enough to support full functional verification of programs [10], giving programmers the power to guarantee the absence of bugs, and thus improving the quality of software in general. By using such extrinsic approach, we first define a bunch of relations (inductive data types) to specify the syntax, auxiliary definitions, the behavior (reduction rules), and the type system of the FJ programming language. Once defined the complete set of rules, we write proofs of properties about them. The proofs are separate external artifacts, which use structural induction to verify the desired properties. In this case, we are interested in mechanically proving the properties defined on paper in the original FJ [4], which include several lemmas, and the properties of *progress* and *preservation*, which together provide guarantees of type safety.

### A. Syntax

The syntax of FJ includes the definition of a class table (CT), which stores all classes in a source-code program, and an expression, which replaces the Java's main method. An expression can refer to information of two sources: (1) a context to deal with variables, which stores the actual parameters during a method invocation; (2) the class table, to perform operations involving attributes or methods. 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 the method body).

Considering all this, we start our formalization in Agda by defining the syntactic elements regarding FJ. A Class is represented by a record with four fields. The class name is stored in cname, the base class is in super, the attributes are in flds, and the methods are in meths. We keep all names abstract, and the only requirement for them is equality. For simplicity, we define Name = ℕ, a simple type which satisfies this requirement.

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

As we 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 we can see next.

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

As we mentioned before, an expression can appear 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. We represent it 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, which receives a variable name as argument. A field access is encoded by Field, receiving two arguments. The first is an Expr which represents the instantiated object, and the second is the attribute name. A method invocation is encoded by Invk and receives three arguments. The first is similar to Field, the second is the method name, and the third is the formal parameters on a method invocation. Lastly, an object instantiation is defined by New, receiving two arguments. The first is the class name being instantiated, and the second is a list of formal parameters for the constructor. The complete BNF grammar is presented in [4].

The only possible value in FJ is encoded in the Val definition.

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

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 Agda's standard library's datatype All, which associates the predicate Val for each element of the given list cp.

### B. 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, we defined two auxiliary definitions. Using the definition fields one can refer to information about the attributes of a class, including the fields inherited by its super class.

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

We use an auxiliary definition (_ ∋ _ : _)[2] which binds a value cd (class definition) from an element c (class name) in a list of pairs Δ (class table). In our encoding, we use this definition several times to lookup information about classes, fields, methods, and variables.

By using the predicate method it is possible to refer information about a specific method in a certain class.

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

Both auxiliary definitions refer to information on a class table Δ, which is defined globally in the working module.

### C. 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, we use two important definitions: 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[3].

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

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

[2]We omit the code of _ ∋ _ : _ predicate, but it can be found in our repository [8].

[3]The details about both definitions can be found online [8].

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 substitution of the 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 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.

```
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′
```

We use an extra predicate _ ⟼ _ (note the different arrow) to evaluate a list of expressions recursively.

### D. 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, we have the typing judgment predicate _ ⊢ _ : _ which encodes the typing rules of FJ, 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, defined as a list of pairs List (Name × Name), aiming to store the types for variables. The second is represented by 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 uses the auxiliary definition _ ∋ _ : _ to lookup the context, binding the type C for a variable x in a context Γ.

```
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 our auxiliary predicate method to obtain the definition of method m in class C. It is used to type-check the method parameters mp[4]. 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 our definition, we use the All datatype 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. We use the expression typing judgment 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.

[4]We use $\text{proj}_2$ to get the second argument of a tuple, and unzip to split a list of tuples.

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

## IV. Proving Safety Properties

We proved type soundness through the standard theorems of *preservation* and *progress* for our formalization of FJ. This section presents only the main proofs, which use several lemmas to fulfill the proof requirements. The interested reader can refer to our source-code repository to see the intricacies of the whole proofs.

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, and we finish 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 we apply the induction hypothesis; (2) the congruence for the list of arguments, where we use an auxiliary proof preservation-list which applies 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[5]. The function subst represents the lemma which states that *Expression substitution preserves typing* [4]. Lastly, T-New has only the congruence case for which we apply the induction hypothesis for each argument of the class constructor.

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 $\tau$ in an empty context [], then it can make *Progress*, i.e., or e is a value, or it can take another reduction step. We use the inductive datatype Progress to hold the result of our proof, with two constructors: Done when e is a value, and Step when e reduces to an e′.

```
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))) p
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, and the reader should understand without further explanation. 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.

## V. Evaluation

Following Wadler's recipe [7] to automate evaluation for the Simple Typed Lambda Calculus (STLC), we also define an evaluator for FJ, by the repeated application of the proofs of *progress* and *preservation*, using an Agda function that computes the reduction sequence from any given closed, well-typed expression to its value.

First we present the inductive datatype _ ↠ _, which represents the multi-step relation, or the reflexive and transitive closure of the step relation. This relation is defined as a sequence of zero (refl) or more steps (multi) of the underlying relation.

```
data _ ↠ _ : Expr → Expr → Set where
  refl : ∀ {e} → e ↠ e
  multi : ∀ {e e′ e″} → e ⟶ e′ → e′ ↠ e″ → e ↠ e″
```

Then, we can implement the function eval. Since Agda is a total language, we use a Fuel (represented as a natural number) to avoid non-termination. We also use two other inductive

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

datatype definitions[6] (Finished which has two constructors: done indicating that the computation is successfully finished, and out-of-gas indicating that the fuel ran out; and Steps, with one constructor: steps which combines the multi-step relation _ ↠ _ and the Finished datatype) to proceed with the evaluation.

```
eval : ∀ {e τ} → Fuel → [] ⊢ e : τ → Steps e
eval zero t = steps refl out-of-gas
eval (suc fuel) t with progress t
... | Done vl = steps refl (done vl)
... | Step stp with eval fuel (preservation t stp)
... | steps stp′ fin = steps (multi stp stp′) fin
```

The eval function receives the fuel and evidence that e is a well-typed expression, and produces the Steps to evaluate the given expression. It starts with a closed and well-typed term. By progress, it is either a value, in which case we are Done, or it reduces to some other expression. By preservation, that other expression will be closed and well-typed. This process is repeated until we reach a value, or the fuel rans out [7].

## VI. Related Work

There are several papers describing the mechanization of programming languages in proof assistants. For example, in their book, Pierce et al. [11] describe the formalization of STLC in Coq, and Wadler [7] present the formalization of STLC in Agda. We used their ideas to build the foundations of our encoding. Besides these books, there are several other papers mechanizing different versions of $\lambda$-calculus, among other languages [12], [13].

Regarding Featherweight Java, there are some projects describing its formalization. Feitosa et al. [14] provided an intrinsically-typed formalization for FJ. Mackay et al. [15] developed a mechanized formalization of FJ with assignment and immutability in Coq, proving type-soundness for their results. Delaware et al. [3] used FJ as basis to describe how to engineer product lines with theorems and proofs built from feature modules, also carrying the formalization Coq. All these papers inspired us in our modeling of FJ. The first difference between these works and ours is that we encoded the semantic rules and proofs in Agda, which is being used more frequently nowadays. Another difference is that we do not use any proof automation, since Agda's system is not as powerful as Coq's. As far as we know, our work is the first to formalize FJ in Agda using the extrinsic approach. We believe that this formalization can be used as basis to study properties of object-oriented programming languages by other researchers.

## VII. Conclusion

In this paper, we presented a formalization of Featherweight Java using the Wright and Felleisen's syntactic approach to specify the static and dynamic semantics, proving the common soundness properties. As we could notice, although FJ is a small core calculus, its non-trivial binding structures and intricate relation between class tables and expressions give rise to challenges during its formalization. The Agda language has shown to be a good tool for such work, although it does not provide proof automation, which can make the maintainability process difficult for large subsets of languages and bigger proofs. During the development of this work, we have changed our definitions many times, both as a result of correcting errors and streamlining the presentation. The possibility of testing the changes by running the evaluation function helps to reason about the impact right away.

As future work, we intend to extend the formalization to embed more features of Java, like dynamic dispatch, $\lambda$-expressions and default methods, studying which features do enjoy the safety properties. We can also explore different approaches to formalize the language and extensions, and prove the equivalence with the result presented in this paper.

[6]For brevity, we just discuss their constructors, however the complete source-code is available on-line [8].


## Acknowledgment

We would like to thank Wouter Swierstra and Alejandro Serrano Mena from the Software Technology Group at Utrecht University for the valuable suggestions during the development of this paper.


## References


[1] tiobe.com, "TIOBE Index," https://www.tiobe.com/tiobe-index/, 04 2019, accessed: 2019-03-12.

[2] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The Java lang. specification, Java SE 8," 2014.

[3] B. Delaware, W. Cook, and D. Batory, "Product lines of theorems," *SIGPLAN Not.*, vol. 46, no. 10, pp. 595–608, Oct. 2011.

[4] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight java: A minimal core calculus for java and gj," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 3, pp. 396–450, May 2001.

[5] P. Martin-Löf, "An intuitionistic theory of types," in *25 years of constructive type theory*, ser. 1. Oxford Univ. Press, New York, 1998, vol. 36, pp. 127–172.

[6] A. Wright and M. Felleisen, "A syntactic approach to type soundness," *Inf. Comput.*, vol. 115, no. 1, pp. 38–94, Nov. 1994.

[7] P. Wadler, "Programming language foundations in agda," in *Formal Methods: Foundations and Applications*, T. Massoni and M. R. Mousavi, Eds. Cham: Springer International Publishing, 2018, pp. 56–73.

[8] S. Feitosa, A. Mena, R. Ribeiro, and A. D. Bois, "Extrinsic Formalization of Featherweight Java in Agda - On-line repository." https://github.com/fjpub/ettfj, 2019.

[9] B. C. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[10] A. Stump, *Verified Functional Programming in Agda*. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool, 2016.

[11] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach, and B. Yorgey, "Software foundations volume 2: Programming language foundations. electronic textbook. version 5.7," 2018.

[12] R. Ribeiro, L. Figueiredo, and C. Camarão, "Mechanized metatheory for a lambda-calculus with trust types," *J. Brazilian Computer Society*, vol. 19, no. 4, pp. 433–443, Nov 2013.

[13] K. Donnelly and H. Xi, "A formalization of strong normalization for simply-typed lambda-calculus and System F," *Elec. Not. Theor. Comp. Sci.*, pp. 109–125, Jun. 07.

[14] S. d. S. Feitosa, A. S. Mena, R. G. Ribeiro, and A. R. D. Bois, "An inherently-typed formalization for featherweight java," in *Proc. of the XXIII Brazilian Symposium on Prog. Languages*, ser. SBLP 2019. New York, NY, USA: ACM, 2019, pp. 11–18.

[15] J. Mackay, H. Mehnert, A. Potanin, L. Groves, and N. Cameron, "Encoding Featherweight Java with assignment and immutability using the Coq proof assistant," in *14th Works. on Formal Tech. for Java-like Prog.*, ser. FTfJP '12, 2012, pp. 11–19.