Mini-Foc A Kernel Calculus for Certified Computer Algebra [Ongoing work]

Stéphane Fetcher, Luigi Liquori

Mini-Foc: A Kernel Calculus for Certified Computer Algebra

[Ongoing Work]

Stéphane Fechter [a] and Luigi Liquori [b]

[a] *LIP6, Université Paris VI*

[b] *INRIA Sophia Antipolis*

Abstract

The Foc language is designed to bring solutions on the reliability of the software, in particular on the development and the reusing of certified libraries, especially for Certified Computer Algebra.

The Foc project aims at building an environment to develop certified computer algebra libraries. The project develops a language called Foc, where any implementation must come with a proof of its correctness. This includes of course pre- and post- condition statements, but also proofs of purely mathematical theorems. In this context, reusability of the *code*, but also of the *correctness proofs* is of very important concern: a tool written for mathematical *Groups* should be available for the mathematical *Rings*, provided the system knows that every Ring is a Group, wich can be faithfully modeled by suitable subtyping relation.

For this, this formal language allows to implement certified components called *Collections*. These collections are specified and implemented step by step: the programmer describes formally – the properties of the algorithm, – the context in which they are executed, – the data representation and proves formally that the implemented algorithms satisfies the specified properties. This programming paradigm implies the use of classic oriented-object features and the use of module features like interfaces and encapsulation of data representation.

This conception of the object oriented programming brings the question where the Foc language is situated in relation to others more classic object-oriented languages. To answer to this question, we propose a kernel of Foc called Mini-Foc. With this kernel, we are interested by the programming aspect of Foc (proof aspect are left to suitable extensions)

Thus, the main ingredients of Mini-Foc, are multiple inheritance, late binding, overriding, interfaces and encapsulation of the data representation. We specify

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

formally the syntax, the semantics and the type system.

Key words: Certified Computer Algebra, Existential-types,
Object Orientation, Syntax, Operational semantics, Type system

---

1 Introduction

This paper is in relation to the problem of the reliability of the software. In particular, we are interested by the development of certified libraries. Moreover, we like to develop them by reusing other certified libraries. The Foc language provides solutions for this problem. Its purpose is to give the possibility to build algebraic structures with a stop of specification, a step of development and a step of proof, and by using object oriented features. These structures may be used in a safety way. For this, Foc provides a mechanism of encapsulation to protect the representation of invariants described in the structures.

The *collections* and *species* are the two notions of package units provided by Foc. A collection can be seen as an abstract data type, that is a module containing the definition of a type, called the *carrier-type*, a set of functions manipulating values (whose the types is the same of the carrier-type), called the *entities* of the species and a set of properties with their proofs. The concrete definition of the carrier-type is hidden for the end users: it is encapsulated. This encapsulation is fundamental to ensure that the invariant on the data representation associated with the collection (*e.g. the entities are even natural numbers*) is never broken. A collection is the ultimate refinement of specifications introduced step by step with different abstraction levels. Such a specification unit is called a species: it specifies a carrier-type, functions and properties (both called the methods of the species). Carrier-type and methods may be defined or only declared. In the latter case, the definition of the function is given later in more concrete species, and similarly the proof of a property can be deferred. Species come with late binding: the definition of a function may use a function that is only declared at this level. A complete species is a species whose the methods are defined and for each property, a proof must be provided. A collection built from a complete species. A species B refines a species A if the methods introduced in A and/or the carrier-type of A are made more concrete (more defined) in B. This form of refinement is completed with the inheritance mechanism, that allows us to build a new species from one or more existing species. The new species inherits the carrier-type and the methods of the inherited species. The new species can also specify new methods or redefine inherited ones.

Carrier-type, multiple inheritance, late binding, encapsulation, refinement are the elementary ingredients of our approach that ensure that the generated code satisfies the specified properties. The purpose of this paper is to formalize these elementary ingredients in order to fit with oriented-object languages. For

this, we formally define the type system and semantics of the core language we call Mini-Foc.

The presentation is intentionally kept informal, with few definitions, no full type systems in appendix and no theorems. Quite simply, the main aim of this paper is to introduce and set the formal basis of the Foc language through a minimal but still powerful kernel calculus Mini-Foc, and compare existing systems concerning languages/calculi suitable for certified computer algebra.

Road Map.

The Section 1 presents the pseudo-terms and types of Mini-Foc. Section 2 present an informal overview of Mini-Foc; Section 3 presents an operational semantics, while Section 4 presents the typing rules. Lastly, Section 5 presents related works and conclude. A full page dedicated to the Foc project can be found in

http://www-spi.lip6.fr/foc/index-en.html.

2 An overview of Mini-Foc

In this section, we illustrate some features of Foc trough Mini-Foc. The examples of this section are simple sets equipped with some operations.

Abuse of notations for the examples

To simplify the examples, in addition to Lambda-calculus expressions, we use local definitions `let x=e and y=e and ... in e` *à la* Caml. And we use the conditional expression `if x then y else z` (if the expression `x` is true then the expression `y` is evaluated else it is the expression `z`). Moreover, in order to name the collection we use a global `let`. Also we add the symbols `+` and `*`, which are respectively the addition and the multiplication operations on the integers. And the `==` is added for the equality test on two integers.

The `op_set` species

The Mini-Foc environment allows to describe these sets by following a generic way:

```
spec op_set in {
 rep α = int;
 sig neutral : α;
 sig op      : α -> α -> α;
 def id      : α -> α = λself.λx.x; }
```

The species `op_set` represents a set having an element `neutral`, equipped with a binary operation `op` and the identity function `id`. The representation of elements for this set is given by the type `int` (the type for the integers) introduced by the notation `rep α = int`. This type is called the carrier type.

At this level of abstraction, `neutral` and `op` are declarations of the species `op_set`. On the other hand, a definition is given for `id`. In this case, `id` is a method of the species `op_set`. The expression to define a method, begins always by a λx where x represents the self reference. To avoid confusions, this variable is written with `self` for the examples. For the rest of examples, when `self` is not used, to simplify, we omit λ`self` in the method definitions.
The type variable α, introduced with `rep` α `= int`, is an alias for the carrier type. It is used to define the types of declarations and methods. Also, it allows to distinguish an integer of our working set from any integers.
The types of declarations and methods don't take in account the variable `self`. For example the definition of `id` (that is λ`self.`λ`x.x`) have the type α `->` α, that is the type for λ`x.x` expression.

The `add_set` and `add_set` species

Mini-Foc allows to write other species by inheritance. For the examples, we write species in order to give a definition for the declarations `neutral` and `op`.

```
spec add_set inh op_set in {
  rep α = int;
  def neutral : α = 0 ;
  def op      : α -> α -> α = λx.λy.x+y; }
```

The species `add_set` inherits of the method `id`. And the declarations `neutral` and `op` from `add_set`, are defined respectively by the number `0` and the function λ`x.`λ`y.x+y`.

The method `neutral` have the type `int`, the method `op` have the type `int -> int -> int` and the method `id` have the type `int -> int`. These types, obtained by replacing α by `int`, give an internal vision of the species `add_set`. But for an outside vision, we just know that the method `neutral` have the type α, the method `op` have the type α `->` α `->` α and the method `id` have the type α `->` α. The set of name methods with their types, by forgetting to replace α by the carrier type, is called the **interface** of the species.

For our examples, we give an other derivation of the species `op_set`:

```
spec mult_set inh op_set in {
  rep α = int;
  def neutral : α = 1 ;
  def op      : α -> α -> α = λx.λy.x*y ; }
```

The `modulo_2` species

Mini-Foc allows to redefine methods as it shows below:

```
spec modulo_2 inh add_set in {
 rep α = int;
 def op : α -> α -> α = λself.λx.λy.
```

```
   let r = x+y in  if r == 2 then self!neutral else r ;
 def one : α = 1;  }
```

The method `op` has been redefined with a new expression. In this definition, the method `neutral` is used by invoking it on the variable `self`. Mini-Foc provides the late binding for the methods. That is the last definition of `neutral`, will be considered when the method `op` will be invoked. Thus, it's not necessarily that `self!neutral` refers to the definition situated in the species `add_set`. On the other hand, Mini-Foc doesn't provide the possibility to regive an instance for the variable α unless it's the same that the previous one. Indeed, if we write `rep` α `= bool` in the species `modulo_2`, then the typing for the methods is broken. An other intuitive reason is that we work on a underlying set whose the representation of elements is fixed. As this representation is given by a type (the carrier type), then this type will never change.

The `some_set` species

Mini-Foc provides also multi-inheritance possibilities:

```
spec some_set inh add_set; mult_set in {
 rep α = int;
 def eq : α -> α -> bool=λx.λy.x == y; }
```

The species `some_set` inherits from `add_set` and `mult_set`. The methods of `add_set` are redefined in `mult_set`. By convention, they are the methods of `mult_set` which are conserved.

The `cartesian` species

Lastly, Mini-Foc provides the possibility to write parameterized species as the species `cartesian`:

```
spec cartesian (c1 is op_set, c2 is op_set) inh op_set in {
 rep α = c1!rep * c2!rep;
 def first   : α -> c1!rep = λx.fst(x);
 def second  : α -> c1!rep = λx. snd(x);
 def neutral : α = (c1!neutral , c2!neutral);
 def op      : α ->  α -> α = λx.λy.
   let l1 = self!first x  and  l2 = self!first y
   and r1 = self!second x and  r2 = self!second x
   in ( c1!op l1 l2 , c2!op r1 r2);  }
```

The above species gets two parameters `c1` and `c2` precised by the species `op_set`. On `c1` and `c2`, we can invoke the methods of `op_set` as it is done in the methods `neutral` and `op`. On the other hand, through `c1` and `c2`, we have just access to the interface of the species `op_set`. Moreover the interface linked to `c1` and the one linked to `c2` are considered as different. Thus, for example, the expression `c1!id c2!neutral` is badly typed. On the other hand, `c1!id`

`c1!neutral` is well typed since the invocation methods are done on the same parameter.

In the species `cartesian`, we define the carrier type with the cartesian product of `c1!rep` and `c2!rep`. The annotation `c1!rep` means that we refers to the carrier type linked the parameter `c1`. `c1!rep` and `c2!rep` can be also used to define the types of declarations and methods (e.g. the methods `first` and `second`).

Creation collections from above species

The species `add_set`, `mult_set`, `modulo_set` and `some_set` are complete species. That is species whose the carrier type is given and all methods are defined (there are not any declarations). From such species, we can create collection. For example:

```
let col_add  = impl add_set  ;;
let col_mult = impl mult_set ;;
```

The collections `col_add` and `col_mult` are created respectively from the species `add_set`, `mult_set`. On the collection `col_add`, we know just the interface of the species `add_set`. Likewise for `col_mult` whose the interface is different of the one for `col_add`. On a collection, we can invoke methods:

```
col_add!op col_add!neutral col_add!neutral
```

On the other hand, since `col_add` have the interface of `add_set`, it is forbidden to give integers in argument for the methods `op`. In spite of the carrier type definition is `int` in the species `add_set`.

The interfaces of `col_add` and `col_mult` is the same that the species `op_set`. Thus, we can create a new collection from `cartesian` by applying it these two collections:

```
let col_cart = impl cartesian(coll_add,coll_mult)  ;;
```

Also, it is possible to apply on `cartesian`, collections having most methods than the interface of `op_set`. For example, we can apply collections created from `modulo_2` whose the method `one` is not included in the interface of `op_set`.

3 Pseudo-terms and types

This section present the syntax and the operational semantics of Mini-Foc.

Notational conventions

In this paper the symbols $x, y, \ldots$ range over the set $\mathcal{X}$ of variables, the symbol S ranges over the set $\mathcal{S}$ of species names, and ST over species table. The symbols $\mathsf{m}, \mathsf{n}, \ldots$ range over the set $\mathcal{M}$ of method names. The symbol κ ranges over the set $\mathcal{K}$ of constants. The symbols $\alpha, \beta, \gamma, \ldots$ range over the set

$\mathcal{V}$ of type-variables. All symbols can be indexed.
Also, for two vectors $\overline{\mathsf{m}}_r$ and $\overline{\mathsf{m}}_t$, we use the letter m and t in index in order to distinguish these two vectors.

Syntax

An Mini-Foc program is a pair $(\mathsf{ST}, \mathrm{e})$ of a species table, and an expression. The syntax of type in Mini-Foc is as follows:

$$\tau ::= \iota \mid \alpha \mid \tau \to \tau \mid \tau * \tau \mid \qquad \text{First-Order Types}$$

$$x\,!\,\mathsf{rep} \mid \exists\alpha.\tau \mid \langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}\rangle \qquad \text{Mini-Foc Types}$$

The syntax of types is composed of two parts: first-order types and types peculiar to Mini-Foc. A first-order type can be an atomic type ι for the constants, a type-variable (for existential-types, to be defined below), or an arrow type or a cartesian type. The types proper to Mini-Foc can be a record-type, a representation-type or an existential-type. Intuitively:

- a record-type has the form $\langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}\rangle$ that is the type of collection, which is typically an instance of a species. The terms $\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}$ are the types of defined methods. And the terms $\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}$ are the types of declared methods (a kind of `virtual`). The keyword rep introduces the *carrier-type* τ of the collection. In the types $\overline{\tau}$ of $\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}$ and $\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}$, α can occurs free (*i.e.* rep acts as a binder for α). Note that the carrier-type is not recursive.
- an existential-type $\exists\alpha.\tau$ is typically the type of a species (the type τ being a record-type). Thanks to this type, we can abstract the carrier-type of collection, a particularity of a collection.
- a $x\,!\,\mathsf{rep}$ type is generally used in the carrier-type definition of *parameterized species*. The variable x denotes a collection in the current context. The variable x in $x\,!\,\mathsf{rep}$ denotes (with a little abuse of notation) the carrier type of x. Thus when the variable x will be instanced by a collection, $x\,!\,\mathsf{rep}$ will have to replaced by the carrier-type of this collection.

The syntax of species tables and expression are as follows:

$$\mathsf{ST} ::= \mathsf{spec}\ \mathsf{S}\ (\overline{x}\ \mathsf{is}\ \overline{\mathsf{S}}\,,\overline{y}{:}\overline{\tau})\ \mathsf{inh}\ \overline{\mathsf{S}}\ \mathsf{in}\ \{\mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\}$$

$$e \quad ::= e\,!\,\mathsf{m} \mid \mathsf{impl}\ \mathsf{S}(\overline{e}\,,\overline{e}) \mid \langle \mathsf{rep}\ \alpha{=}\tau\,,\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\rangle$$

$$\kappa \mid x \mid \lambda x.e \mid e\,e \mid e\,,e \mid \mathsf{fix}(\lambda x.e)$$

- A species S is made of (formal) parameters $(\overline{x}\ \mathsf{is}\ \overline{\mathsf{S}}\,,\overline{y}{:}\overline{\tau})$, *i.e.* a list of inherited species $\overline{\mathsf{S}}$ introduced by inh and a body $\{\mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\lambda x.\overline{e}\}$. The body of species introduces the definition of the carrier-type by $\mathsf{rep}\ \alpha{=}\tau$,

the declarations by sig $\overline{\mathsf{m}}{:}\overline{\tau}$ (m is the name of the declaration and τ its type), and methods by def $\overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}$ (m is the name of the method, $\lambda x.e$ is its definition and τ its type). The variable x, bound by λ, in the definition of a method, is used for the self reference. Like the type of collection, all occurrences of α in the type of declarations and methods, are bound by rep and can be substituted by the carrier-type τ, in order to obtain a "runnable" collection.

A species S take two types of parameters. The first type of parameter is $\overline{x}$ is $\overline{\mathsf{S}}$. An instance of this parameter will have to be a collection whose the interface is the one of the species S. And, in the body of the species, we have just an abstract vision of S when we use the parameter x.

The second type of parameter has the classical meaning as in any constructor *à la* `new` and have the form $\overline{x}{:}\overline{\tau}$. An instance $x \in \overline{x}$ of this parameter will have to be an expression of type $\tau \in \overline{\tau}$.

- The expressions e of the Mini-Foc language is divided in Lambda-calculus expressions and the proper expressions of the language. The Lambda-calculus expressions are classical. There are constants given by κ, variables x, abstractions $\lambda x.e$, applications $e\,e$, products $e\,,e$ and the recursive function $\mathsf{fix}(\lambda x.e)$.
- The main Mini-Foc expression is the collection $\langle \mathsf{rep}\ \alpha{=}\tau\,,\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\rangle$ that can be viewed like a *complete object*. Like a species, def $\overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}$ are the methods of the collection. On the other hand, unlike species, a collection doesn't possess declaration (*i.e.* no virtual methods). Moreover, the methods are distinct. The carrier-type of a collection is also introduced by rep $\alpha{=}\tau$. And all occurrences of α in the types of methods are bound by rep. A collection is destined to be abstracted, on the carrier-type, for its final users.
- The second Mini-Foc expression is impl $\mathsf{S}(\overline{e}_1, \overline{e}_2)$. It allows to create a new collection from species S. Since a species take some parameters, then we must pass in argument the list $\overline{e}_1$ of collections, and the list $\overline{e}_2$ of Lambda-calculus expressions.
- Lastly, as in any "decent" object-oriented calculus, the $e\,!\,\mathsf{m}$ allows to invoke a method m on a collection.

4 Operational Semantics

We present a classical small-step operational semantics for the pure functional call-by value fragment of Mini-Foc. The semantics is described by a set of small-step reduction rules (see Figure 3 and 4) and a set of evaluation contexts (see Figure 2). Thus the evaluation of an expression, if it terminates, can be visualized step-by-step until obtaining an expression that can't be reduced anymore. The values are described in the Figure 1; a value can be a constant κ, a variable x, an abstraction $\lambda x.e$ or a pair of value $v\,,v$. A value can be also

$$v, w ::= \kappa \mid x \mid \lambda x.e \mid v\,,v \mid \langle \mathsf{rep}\ \alpha{=}\tau\,, \mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{v}\rangle$$

Fig. 1. Mini-Foc Values

$$\mathsf{E} ::= [\cdot] \mid \mathsf{E}\,e \mid v\,\mathsf{E} \mid \mathsf{E}\,,e \mid v\,,\mathsf{E} \mid \mathsf{fst}(\mathsf{E}) \mid \mathsf{snd}(\mathsf{E}) \mid \mathsf{E}\,!\,\mathsf{m} \mid \mathsf{impl}\ \mathsf{S}(\mathsf{E}\,,e) \mid \mathsf{impl}\ \mathsf{S}(v\,,\mathsf{E})$$

$$\mathsf{E}[e] \to \mathsf{E}[e'] \quad \text{if} \quad e \rightsquigarrow e'$$

Fig. 2. Reduction Contexts and Contextual Rule

$$\begin{array}{lll}
(\lambda x.e)\,v & \rightsquigarrow e[v/x] & \beta_{(\mathrm{Fun})} \\
\mathsf{fst}(v_1\,,v_2) & \rightsquigarrow v_1 & \delta_{(\mathrm{Fst})} \\
\mathsf{snd}(v_1\,,v_2) & \rightsquigarrow v_2 & \delta_{(\mathrm{Snd})} \\
\mathsf{fix}(\lambda x.e) & \rightsquigarrow e[\mathsf{fix}(\lambda x.e)/x] & \delta_{(\mathrm{Fix})}
\end{array}$$

Fig. 3. Mini-Foc Lambda-like Rules

a collection (*i.e.* an object) $\langle \mathsf{rep}\ \alpha{=}\tau\,, \mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{v}\rangle$ whose the methods are also values (recall that method-bodies are functions whose first parameter is the object itself, and that functions are values). There are two types of small-step reduction rules :

- The first type of rules, in the Figure 3, are the classic rules of Lambda-calculus. The rule $\beta_{(\mathrm{Fun})}$ is the beta reduction. The expression $(\lambda x.e)\,v$ is reduced by replacing all free occurrence of x in e, by the value v. The rules $\delta_{(\mathrm{Fst})}$ and $\delta_{(\mathrm{Snd})}$ are respectively, the left and right projection on the pair of values. And the rule $\delta_{(\mathrm{Fix})}$ apply the fix point operator on the an abstraction $\lambda x.e$ by replacing all free occurrences of x in e by $\mathsf{fix}(\lambda x.e)$ itself.
- The second type of rules, in the Figure 4, evaluates proper Mini-Foc expressions. The rule $\delta_{(\mathrm{Self})}$ allows to reduce an invocation of a method m_i on a collection $\langle \mathsf{rep}\ \alpha{=}\tau; \mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\rangle$. For this, we retrieve the definition $\lambda x.e_i$ corresponding to the method m_i in the list $\overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}$. The free occurrences of x in e_i represents the self reference. Thus, these occurrences of x in e_i are replaced by the collection itself in order to obtain the output expression.
- The $\delta_{(\mathrm{Coll})}$ rule is used to reduce the creation of collection from a species S. This rule must retrieve the species S in the species table ST. By using the function meth (see Figure ??) on the species S, we return all methods $\overline{\mathsf{m}}{:}\overline{\tau}'{=}\overline{e}'$ of the species. Thanks to the function meth, the returned methods are distinct and correspond to the multi-inheritance of the species. Free

$$\langle \mathsf{rep}\ \alpha{=}\tau; \mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\rangle\,!\,\mathsf{m}_i \rightsquigarrow e_i[\langle \mathsf{rep}\ \alpha{=}\tau; \mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\rangle/x] \qquad \delta_{(\mathrm{Self})}\ (\mathsf{m}_i \in \overline{\mathsf{m}})$$

$$\frac{\mathsf{ST}(\mathsf{S}) = \mathsf{spec}\ \mathsf{S}\ (\overline{x}\ \mathsf{is}\ \overline{\mathsf{S}}\,, \overline{y}{:}\overline{\tau})\ \mathsf{inh}\ \overline{\mathsf{S}'}\ \mathsf{in}\ \{\mathsf{rep}\ \alpha{=}\tau; \ldots\} \quad \mathsf{meth}(\mathsf{S}) = \langle \mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}'{=}\overline{e}'\rangle \quad \mathsf{grep}(\overline{v}) = \overline{\tau}''}{\mathsf{impl}\ \mathsf{S}(\overline{v}\,, \overline{v}') \rightsquigarrow \langle \mathsf{rep}\ \alpha{=}\tau[\overline{\tau}''/\overline{x\,!\,\mathsf{rep}}]\,, \mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}'[\overline{\tau}''/\overline{x\,!\,\mathsf{rep}}]{=}\overline{e}'[\overline{v}/\overline{x}, \overline{v}'/\overline{y}]\rangle}\ \delta_{(\mathrm{Coll})}$$

Fig. 4. Mini-Foc *ad hoc* Rules

occurrences of parameters $\overline{x}$ and $\overline{y}$ may be in these methods. In this case, for the output result, $\overline{x}$ or replaced by the collection values $\overline{v}$ and the $\overline{y}$ are replaced by the Lambda-calculus values $\overline{v}'$. Then, free occurrences of $\overline{x\,!\,\mathsf{rep}}$ (the carrier-type references of parameter $\overline{x}$) may be in the carrier-type τ and the method types of the species S. Thus, these special variables must be replaced by the carrier-types of collections $\overline{v}$. The function grep allows to return these carrier-types from $\overline{v}$.

In the Figure **??**, we define formally the functions grep and meth.

- The function grep takes in parameter a list of collection in order to return the list of their carrier-types.
- The function meth takes in parameter a species S in order to return the list of all its declarations and methods. Moreover, declarations and methods are distinct. If this list have not declarations, then the species S is complete. This property is used in the rule $\delta_{(\mathrm{Coll})}$ (see Figure 4) in order to precise that the creation of a collection must be done from a complete species.
- The function meth collects the inherited method according to the choosen politics to deal with multiple inheritance (in case of conflict right-most method is selected by the dynamic lookup algorithm). Thus, when a method is redefined, it's the right-st definition that is chosen. For this, the definition of meth uses $\bigsqcup$ binary operation defined according to the rules (R2L), which returns the union of two lists of declarations and methods. In this returned list, if a method is redefined, then the most at the right one is preserved. And if a declaration possesses a correspondant method, then it is removed.

5 Type system

The typing rules, presented in the Figures 6 and 7, allow to certify or not that an expression is well-typed in a given context. Formally, it is given by the relation $\Gamma \vdash e : \tau$: the expression e is well typed with the type τ under the context Γ. This context is a typing environment defined by:

$$\Gamma ::= \emptyset \mid \Gamma, \kappa{:}\tau \mid \Gamma, x{:}\tau \mid \Gamma, \iota{:}\star \mid \Gamma, \alpha{:}\star$$

$$\frac{\Gamma(\iota) = \star}{\Gamma \vdash \iota}\ (\iota\text{−type}) \qquad \frac{\Gamma(\alpha) = \star}{\Gamma \vdash \alpha}\ (\alpha\text{−type})$$

$$\frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}\ (\text{arrow−type}) \qquad \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 * \tau_2}\ (\text{cartesian−type})$$

$$\frac{\Gamma(x) = \langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{e}\rangle \quad \Gamma \vdash \tau}{\Gamma \vdash x\,!\,\mathsf{rep}}\ (\text{rep−type})$$

Fig. 5. Mini-Foc well Formed Types

$$\frac{\Gamma(\kappa) = \iota}{\Gamma \vdash \kappa : \iota}\ (\text{Cst}) \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (\text{Var})$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ (\text{Abst}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ (\text{Apply})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\,, e_2 : \tau_1 * \tau_2}\ (\text{Pair}) \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}(\lambda x.e) : \tau}\ (\text{Fix})$$

Fig. 6. Lambda-calculus First-order Typing Rules

A typing environment is composed of variables x associated with their Mini-Foc type and composed of type-variables associated with their sort $\star$. It is also composed of constants κ associated with their Mini-Foc type and composed of the constant types ι assiocated with their sort $\star$.
We provide a function fresh, that takes in argument a typing environment Γ, in order to return a fresh type variable in relation to Γ.

The collections and species are expressions that use types in order to define the carrier-type and specify the types of methods. In relation to the context, these types must valid. For this, the relation $\Gamma \vdash \tau$, presented in Figure 5 verify that the type τ is valid under the typing environment Γ. Not any types can be used to define the carrier-type or the type of a method. Only constant types, type-variables, carrier-type references and their arrows and cartesian types, are accepted. Moreover, a type variable α is valid if α is present in the actual typing environment. And a carrier-type reference $x\,!\,\mathsf{rep}$ is valid if the collection variable x is present in the actual typing environment. The type of x, that is $\langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{e}\rangle$, introducing a carrier type τ must be also valid.

The rules for first-order typing of λ-expressions, described in Figure 6, are standard and they need no comment. The rules for Foc expressions are described in Figure 7. More precisely:

- The rule (Coll) allows to type a collection $\langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\rangle$. Intuitively, a collection is well type under two conditions. The first condition is the carrier-type and the method types must be valid in relation to the

$$\frac{\Gamma \vdash \tau \qquad \Gamma, \alpha{:}\star \vdash \overline{\tau} \qquad \Gamma, \alpha{:}\star, x{:}\langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}\rangle \vdash e : \tau'[\tau/\alpha] \qquad \forall x \in \overline{x}, e \in \overline{e}, \tau' \in \overline{\tau}}{\Gamma \vdash \langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\rangle : \langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}\rangle}\ (\text{Coll})$$

$$\frac{\Gamma \vdash e : \langle \mathsf{rep}\ \alpha{=}\tau'\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}\rangle \quad \mathsf{m}{:}\tau \in \overline{\mathsf{m}}{:}\overline{\tau}}{\Gamma \vdash e\,!\,\mathsf{m} : \tau[\tau'/\alpha]}\ (\text{Send})$$

$$\frac{\begin{array}{l}\mathsf{ST}(\mathsf{S}) = \mathsf{spec}\ \mathsf{S}\ (\overline{x}\ \mathsf{is}\ \overline{\mathsf{S}}'\,, \overline{y}{:}\overline{\tau})\ \mathsf{inh}\ \overline{\mathsf{S}}\ \mathsf{in}\ \langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\rangle \triangleq \mathcal{S} \\ \Gamma \vdash \mathsf{S}_i : \exists\alpha.\langle \mathsf{rep}\ \alpha_p{=}\alpha\,;\mathsf{sig}\ \overline{\mathsf{n}}_p{:}\overline{\tau}_p\,;\mathsf{def}\ \overline{\mathsf{m}}_p{:}\overline{\tau}_p\rangle \quad \forall \mathsf{S}_i \in \overline{\mathsf{S}}' \\ \Gamma \vdash e : \langle \mathsf{rep}\ \beta'{=}\tau'\,;\mathsf{def}\ \overline{\mathsf{m}}_j{:}\overline{\tau}_j\rangle \quad \Gamma \vdash e' : \tau'' \qquad \forall e \in \overline{e}, e' \in \overline{e}' \\ \Gamma \vdash \mathcal{S}\,\overline{e}\,\overline{e}' : \exists\delta.\langle \mathsf{rep}\ \beta{=}\delta\,;\mathsf{def}\ \overline{\mathsf{m}}_k{:}\overline{\tau}_k\rangle \\ \langle \mathsf{rep}\ \beta'{=}\tau'\,;\mathsf{def}\ \overline{\mathsf{m}}_j{:}\overline{\tau}_j\rangle <: \langle \mathsf{rep}\ \alpha{=}\gamma\,;\mathsf{def}\ \overline{\mathsf{m}}_i{:}\overline{\tau}_i\rangle\end{array}}{\Gamma \vdash \mathsf{impl}\ \mathsf{S}(\overline{e}\,, \overline{e}') : \langle \mathsf{rep}\ \beta{=}\mathsf{fresh}(\Gamma)\,;\mathsf{def}\ \overline{\mathsf{m}}_k{:}\overline{\tau}_k\rangle}\ (\text{CollN})$$

$$\frac{\begin{array}{l}\mathsf{methbis}(\mathsf{S}) = \langle \mathsf{rep}\ \alpha_r{=}\tau'_r\,;\mathsf{sig}\ \overline{\mathsf{n}}_r{:}\overline{\tau}_r\,;\mathsf{def}\ \overline{\mathsf{m}}_r{:}\overline{\tau}_r{=}\overline{\lambda z_r.e_r}\rangle \\ \Gamma \vdash \mathsf{S}_i : \exists\alpha.\langle \mathsf{rep}\ \alpha_p{=}\alpha\,;\mathsf{sig}\ \overline{\mathsf{n}}_p{:}\overline{\tau}_p\,;\mathsf{def}\ \overline{\mathsf{m}}_p{:}\overline{\tau}_p\rangle \quad \forall \mathsf{S}_i \in \overline{\mathsf{S}} \\ \Delta \triangleq \Gamma, \beta{:}\star, x{:}\langle \mathsf{rep}\ \alpha_p{=}\beta\,;\mathsf{def}\ \overline{\mathsf{n}}_p{:}\overline{\tau}_p\,;\mathsf{def}\ \overline{\mathsf{m}}_p{:}\overline{\tau}_p\rangle \quad \forall x \in \overline{x} \quad \beta \in \mathsf{fresh}(\Gamma) \\ \Delta \vdash \overline{\tau}_l \quad \Delta \vdash \tau'_r \quad \Delta, \alpha_r{:}\star \vdash \overline{\tau}_r \\ \Delta, \alpha_r{:}\star, z_r{:}\langle \mathsf{rep}\ \alpha_r{=}\tau'_r\,;\mathsf{def}\ \overline{\mathsf{n}}_r{:}\overline{\tau}_r\,;\mathsf{def}\ \overline{\mathsf{m}}_r{:}\overline{\tau}_r\rangle \vdash e_r : \tau_r[\tau'_r/\alpha_r] \quad \forall z_r \in \overline{z}_r, e_r \in \overline{e}_r \\ \tau \triangleq \overline{\exists\gamma.\langle \mathsf{rep}\ \alpha{=}\gamma\,;\mathsf{def}\ \overline{\mathsf{m}}_i{:}\overline{\tau}_i\rangle} \to \overline{\tau}_l \to \exists\delta.\langle \mathsf{rep}\ \beta{=}\delta\,;\mathsf{def}\ \overline{\mathsf{m}}_k{:}\overline{\tau}_k\rangle\end{array}}{\Gamma \vdash \mathsf{spec}\ \mathsf{S}\ (\overline{x}\ \mathsf{is}\ \overline{\mathsf{S}}\,, \overline{y}{:}\overline{\tau}_l)\ \mathsf{inh}\ \overline{\mathsf{S}}'\mathsf{in}\ \{\mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\} : \tau}\ (\text{Species})$$

$$\frac{J \subseteq I \quad \tau_i = \tau'_i[\beta/\alpha] \quad \forall i \in J}{\langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{def}\ \overline{\mathsf{m}}_i{:}\overline{\tau}_i\rangle^{i\in I} <: \langle \mathsf{rep}\ \beta{=}\tau'\,;\mathsf{def}\ \overline{\mathsf{m}}_j{:}\overline{\tau}'_j\rangle^{j\in J}}\ (<:)$$

Fig. 7. Mini-Foc Typing Rules

$$\frac{\mathsf{SL}(\mathsf{S}) = \mathsf{spec}\ S\ (\overline{x}\ \mathsf{is}\ \overline{\mathsf{S}}'\,, \overline{y}{:}\overline{\tau})\ \mathsf{inh}\ \overline{\mathsf{S}}\ \mathsf{in}\ \{\mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}} : \overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{e}\}}{\mathsf{methbis}(\mathsf{S}) = \biguplus_{S_i \in \overline{\mathsf{S}}} \mathsf{methbis}(\mathsf{S}_i) \uplus \langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{e}\rangle}\ (\text{Methb})$$

Fig. 8. Aux typing rules

actual context. The second condition is every method definition must be well typed. And the type of these definitions must correspond to the types given by the developer. As every method definition can refer to the collection itself, the typing is done in the actual typing environment extended

with the type of the collection itself.

Formally, we verify the carrier-type τ in the current typing environment Γ. And we verify the method types given by the developer, by extending Γ with $\alpha{:}\star$. This extension is necessary to indicate that the type-variable α refers to the carrier-type τ and it's not free. Then, every method definition $\lambda x.e$ must be verified. As the bound variable x represents the collection itself, the expression e must be typed in the current environment extended with the collection type $\langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}\rangle$. The type of the expression e must be the type given by the user whose all free occurrences of α are replaced by the carrier-type τ. That is the type of e must be $\tau'[\tau/\alpha]$. Lastly, the returned type for the collection is $\langle \mathsf{rep}\ \alpha{=}\tau\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}\rangle$.

- The rule (Send) is used to type an invocation of a method m on an expression e. Intuitively, an invocation of m on e, is valid if it exists a correspondent definition brought by e. Formally, the type of the expression e must be the one of a collection $\langle \mathsf{rep}\ \alpha{=}\tau'\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}\rangle$. Then, the method m must be in $\overline{\mathsf{m}}$ in order to retrieve the correspondent type τ. In this type, free occurrences of α that refers to the carrier-type, may exist. Thus the type returned for $e\,!\,m$ is τ with all occurrences of α replaced by the carrier-type τ'.
- The rule (CollN) is used to type the creation of a collection from a species S eventually applied with collections $\overline{e}$ and an expressions $\overline{e}'$. Intuitively, a collection can be created from $\mathsf{impl}\ \mathsf{S}(\overline{e},\overline{e}')$ if the species S is complete. Moreover, for every parameter x $\mathsf{is}\ \mathsf{S}'$ of the species S, the interface of e (from $\overline{e}$) must correspond to the one of the species S'. More precisely, we fetch the species S in the species table ST, and then we typecheck it by providing all needed actual parameters. As expected, the type of a species must be an existential-type $\exists\delta.\langle \mathsf{rep}\ \beta{=}\delta\,;\mathsf{def}\ \overline{\mathsf{m}}_k{:}\overline{\tau}_k\rangle$ representing the type of body of the species S. Of course, the arguments must be well typed according to the type of the species S. Moreover, a "width-subtyping" relation must exist between $\langle \mathsf{rep}\ \alpha_p{=}\alpha\,;\mathsf{sig}\ \overline{\mathsf{n}}_p{:}\overline{\tau}_p\,;\mathsf{def}\ \overline{\mathsf{m}}_p{:}\overline{\tau}_p\rangle$ (from the type of the species S' used for the parameters) and $\langle \mathsf{rep}\ \beta'{=}\tau'\,;\mathsf{def}\ \overline{\mathsf{m}}_j{:}\overline{\tau}_j\rangle$ (the type of collection arguments). For this, we use the relation $<:$ defined in the Figure 7 according to the rule $(<:)$. It imposes that $\overline{\mathsf{n}}_n$ and $\overline{\mathsf{m}}_n$ are included in $\overline{\mathsf{m}}_j$. Then, for any $\mathsf{m}{:}\tau$ of $\overline{\mathsf{m}}_n{:}\overline{\tau}_n$ and $\overline{\mathsf{m}}_n{:}\overline{\tau}_n$, and for the correspondent $\mathsf{m}{:}\tau'$ of $\overline{\mathsf{m}}_j{:}\overline{\tau}_j$, τ and τ' must be equal modulo α,β'. Thanks to this relation, we verify that the interface of the collection passed in argument, is the one expected by the correspondent parameter. Moreover, the collection may have more methods than expected ones by the parameter. Lastly, the type returned for the new collection is $\langle \mathsf{rep}\ \beta{=}\mathsf{fresh}(\Gamma)\,;\mathsf{def}\ \overline{\mathsf{m}}_k{:}\overline{\tau}_k\rangle$ that is the type $\exists\delta.\langle \mathsf{rep}\ \beta{=}\delta\,;\mathsf{def}\ \overline{\mathsf{m}}_k{:}\overline{\tau}_k\rangle$ where δ is replaced by a fresh variable in relation to the actual typing environment.
- Lastly, the rule (Species), allows to type a species

$$\mathsf{spec}\ \mathsf{S}\ (\overline{x}\ \mathsf{is}\ \overline{\mathsf{S}}\,,\overline{y}{:}\overline{\tau})\ \mathsf{inh}\ \overline{\mathsf{S}}'\ \mathsf{in}\ \{\mathsf{rep}\ \alpha{=}\tau\,;\mathsf{sig}\ \overline{\mathsf{m}}{:}\overline{\tau}\,;\mathsf{def}\ \overline{\mathsf{m}}{:}\overline{\tau}{=}\overline{\lambda x.e}\}$$

Intuitively, a species is well typed if the carrier-type is valid and not rede-

fined through the multi-inheritance. Then, the types of declarations and methods must be also valid. If the methods are redefined, or the declarations re-given, then their type must be preserved. Indeed, if there isn't these constraints, the type soundness may be broken. Lastly, the definitions of methods must be have types respecting ones given by the developer.

Formally, we must retrieve the carrier-type, all declarations and methods of the species. For this, the function $\mathsf{methbis}(\mathsf{S})$ is used. Intuitively, this function resemble and works similarly to the function meth. It returns the carrier type, all declarations and method of the species S. But it verifies for multiple method names (from declarations and/or methods) that the types are the same. Moreover, if a method is redefined, the old definition is conserved. It allows to verify that every definition, for a given method, preserves the type of the method.

Then we must type every definition method of the species. This typing must be done in the actual typing environment extended with variables corresponding to parameters, and with the variable representing the underlying collection of the species. But before to do these extension, we must verify for the parameters $\overline{x}$ is $\overline{\mathsf{S}}$, that every S_i of $\overline{\mathsf{S}}$ is well typed. And we must verify for the parameters $\overline{y}{:}\overline{\tau}_l$, that $\overline{\tau}_l$ are valid. Then, we must verify that the carrier-type τ'_r and the declaration and method types $\overline{\tau}_r$ of the species S are valid.

Thus, the type of S_i must be $\exists\alpha.\langle \mathsf{rep}\ \alpha_p{=}\alpha\,;\mathsf{sig}\ \overline{\mathsf{n}}_p{:}\overline{\tau}_p\,;\mathsf{def}\ \overline{\mathsf{m}}_p{:}\overline{\tau}_p\rangle$. Thus we can verify the validity of $\overline{\tau}_l$, τ'_r and $\overline{\tau}_r$, in the environment Γ extended with collection variables $x{:}\langle \mathsf{rep}\ \alpha_p{=}\beta\,;\mathsf{def}\ \overline{\mathsf{n}}_p{:}\overline{\tau}_p\,;\mathsf{def}\ \overline{\mathsf{m}}_p{:}\overline{\tau}_p\rangle$ with $x \in \overline{x}$, where β are fresh type-variables in relation to Γ. Indeed, occurrences of $x\,!\,\mathsf{rep}$ can appear in $\overline{\tau}_l$, and τ'_r and $\overline{\tau}_r$. We remark, that the $\mathsf{sig}\ \overline{\mathsf{n}}_p{:}\overline{\tau}_p$ are transformed in $\mathsf{def}\ \overline{\mathsf{n}}_p{:}\overline{\tau}_p$ in order that the variables $x \in \overline{x}$ represent the collections.

As it is stated previously, every definition method e_r from $\mathsf{def}\ \mathsf{m} : \tau{=}\lambda z_r.e_r$ correspondent, must be typed in the actual environment Γ extended with the variables corresponding to parameters and with the variable representing the underlying collection of the species. In particular, this last variable must z_r with the type $\langle \mathsf{rep}\ \alpha_r{=}\tau'_r\,;\mathsf{def}\ \overline{\mathsf{n}}_r{:}\overline{\tau}_r\,;\mathsf{def}\ \overline{\mathsf{m}}_r{:}\overline{\tau}_r\rangle$. On this type, we remark that $\mathsf{sig}\ \overline{\mathsf{n}}_r{:}\overline{\tau}_r$ has been transformed in $\mathsf{def}\ \overline{\mathsf{n}}_r{:}\overline{\tau}_r$ in order to the type of z_r represents well a collection type.

The type returned for e_r must equal to τ_r where all free occurrences of α_r is replaced by the carrier-type τ'_r.

Lastly, the type returned for the species, is an arrow type whose the first components correspond to the collection parameters, the second components correspond to the Lambda-calculus parameter and the last component is the an existential-type.

6 Relative works and conclusion

In the first part of this paper, we have informally presented Mini-Foc. We have been interested in classic object oriented features like multi-inheritance, late binding, redefinition methods, self references. But also other features like the carrier type, the interface and the abstraction have been presented. These features are particular to Mini-Foc.

The main difference with the classical object oriented languages, is that the species (comparable to classes) and the collections (comparable to objects) don't provide state. Instead of that, the species and collections provides constructions around representation of elements for sets. This representation is given by a type called the carrier type. For the outside world of the sets, the carrier type is abstract. In practice, it allows to avoid to break invariant representation. In order to provide these abstraction mechanism, we use the existential type. We have been inspired by ideas in [PT94] where the autors use the existential type to encapsulate the state of objects.

This version of Mini-Foc doesn't provide all features of Foc. For future versions, Mini-Foc will be extended to provide the proof aspects. In particular, these future version must provide more control on self reference. Indeed, it brings inconsistences. Already, analyses are provided in [Pre03,PD02] to avoid it. Thanks to these analyses, every method call is certified to terminate. These analyses look like the ones done for mixins, in particular ones presented in [HL02]. The authors extend their type system with dependency graphs. If a type derivation tree is built with a graph having at least a cycle, then the tree is considered like inconsistent.

Some relative works can be also found in [Fechter01,Fechter02,FD04]. The aims of these papers was to bring the Foc conceptions to provide a model near of Objective ML [RV98]. Lastly, a tentative to proof formally the type soundness of a minimalist model with some Foc features has been proposed in [FB04].

References

[FB04] Stéphane Fechter and Olivier Boite. BBFoC. Rapport de recherche LIP6 2004/002, Laboratoire d'Informatique de Paris VI, 2004.

[FD04] Séphane Fechter and Catherine Dubois. Vers une définition formelle du langage Foc. Rapport de recherche LIP6 2004/001, Laboratoire d'Informatique de Paris VI, 2004.

[Fechter01] Stéphane Fechter. Une sémantique pour FoC. Rapport de D.E.A., Université Paris 6, Septembre 2001. http://www-spi.lip6.fr/~fechter.

[Fechter02] Stéphane Fechter. An object-oriented model for the certified computer algebra library. Paper presented at FMOODS 2002 PhD workshop, March 2002. http://www-spi.lip6.fr/~fechter.

[HL02] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *European Symposium on Programming*, pages 6–20, 2002.

[PD02] Virgile Prevosto and Damien Doligez. Inheritance of algorithms and proofs in the computer algebra library foc. *Journal of Automated Reasoning*, 29(3-4):337–363, 2002. Special Issue on Mechanising and Automating Mathematics, In Honor of N.G. de Bruijn.

[Pre03] Virgile Prevosto. *Conception et implantation du langage* Foc *pour le développement de logiciels certifiés*. PhD thesis, Université Paris VI, 2003.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[RV98] Didier Rémy and Jérome Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):p. 27–50, 1998.