# Une nouvelle extension de ML avec des traits orientés objets de FOCAL

Stéphane Fechter

# A new extension for ML with object oriented features of Focal

Stéphane Fechter[1]

stephane.fechter@lip6.fr
Laboratoire d'Informatique de Paris VI
8, rue du Capitaine Scott
75015 Paris, France

## 1 Introduction

New paradigms are appearing in relation to object oriented programming like mixins [1–4] for instance. The framework FOCAL belongs to this tendency. In particular, it brings object oriented solutions to develop certified softwares. Moreover, FOCAL has the originality to allow to write in a same program specifications, code and proofs.
FOCAL allows to develop certificate components called **collections** containing a set of functions working on an only data representation called **carrier**. Elements manipulated by functions of a collection are called **entities**. To avoid to break representation invariants, an abstraction barrier is imposed on collections. Hence, a collection can be considered as an algebraic abstract type. The contribution of FOCAL is the possibility to create collections with the help of powerful tools in order to ease the proof of properties and the reusing.

To obtain step by step a collection, FOCAL provides constructions called **species**. A species allows to specify the carrier with a type : **the carrier type**. FOCAL allows to build an hierarchy of species bound by an inheritance relation. At the top of the hierarchy, the carrier type and functions called **methods** are only declared. The declarations are like virtual methods of object oriented languages. By inheritance, the definitions of carrier type and methods are postponed. The developer can also declare properties. However, he will have to prove them. Finally, the different steps of specification, implementation and proofs end up at species called **complete species**. More precisely, the carrier type and every method of a complete species are defined, and every property is proved. Lastly, collection is obtained from a complete species.

In addition to the multi-inheritance, FOCAL provides other object oriented features as the late binding, method redefinition and the self reference. Moreover, species can be defined with parameters whose instances are collections. However, the object oriented features of FOCAL must be used carefully in order to avoid logical inconsistencies. On the one hand, along of a hierarchy of species, the carrier type must not be redefined, but it can be instantiated. On the other hand, parameterization, inheritance imply the existence of strong dependencies between methods and properties. The compilation of FOCAL checks that this dependency graph is acyclic.

In this paper, only operational aspects of FOCAL are considerate. An extension of ML with notions of species and collections is given in order to exhibit an operational semantics. Thus, the model provided here will be able to be compared with other ML extensions dedicated to object oriented programming. Thus, in a general way, the object oriented approach of FOCAL will be able to be compared with the ones provided by more traditional object oriented languages such as JAVA [5] or OCAML[6].

This paper is organized as follows: the section 2 is an overview of the model, called here $\mathcal{FML}$[1]. Abstract syntax is presented in section 3. Type system is presented in section 4. The operational semantics of $\mathcal{FML}$ is presented in section 5. In section 5, type soundness is proved. In section 6, we present relative works and we conclude.

*Notational conventions* In this paper, the symbols $x, y$ range over the set $Var$ of term variables, the symbols $\mathsf{m}, \mathsf{n}, \mathsf{p}, \mathsf{q}$ range over the set $\mathcal{M}$ of method names, the symbols $\mathsf{S}$ ranges over the set $\mathcal{S}$ of species names and the symbols $\mathfrak{c}, \mathfrak{d}, \mathfrak{e}$ range over the set $\mathcal{C}$ of collection names. Term variables are $\alpha$-convertibles oppositely to method names. Also all identifiers used can be indexed.

## 2 An overview of $\mathcal{FML}$

In this section, simple examples are written with the concrete syntax (near to the one of OCAML [6]) of $\mathcal{FML}$ in order to introduce the notions of FOCAL. They start at top of a hierarchy to finish with complete species and collections.

A basic species is defined at the top of the hierarchy:

```
species Basic =
struct
 rep;
 parse : string -> rep;
end;;
```

Such a species declares a carrier type, here still abstract with the field `rep`. The method `parse` is declared and be seen as a virtual method of object oriented languages. It returns an element of `rep`, that is an entity of the future collection. For the sequel, the type associated to a method is called **specification**. Specifications given by the programmer allow to build a structure in order to indicate elements exported by the species. This structure is called **interface** and presented as follows:

```
I_Basic =
struct
 rep;
 parse : string -> rep;
end
```

[1] $\mathcal{FML}$ meaning FOCAL - ML

The interface exports the carrier type and the methods that are only declared. The carrier type of an interface is always opaque in order to abstract it.
For the sequel of the presentation, names of species are prefixed with `I_` to indicate that their interfaces are considered.

The species in FOCAL are refined by inheritance. With $\mathcal{FML}$, `Basic` can be extended as follows:

```
species Basic_add =
struct
  inherits Basic;

  add : rep -> rep -> rep;
end;;
```

`Basic_add` inherits carrier type and methods from BASIC. The interface bound to `Basic_add` is written as follows:

```
I_Basic_add =
struct
 rep;
 parse : string -> rep;
 add : rep -> rep -> rep;
end
```

Species can be parameterized:

```
species ProdCart = fun (c is Basic) -> fun  (d is Basic) ->
struct
 inherits Basic ;
 rep = c!rep * d!rep;
 parse : string -> rep = fun x ->
     let (c1,c2) = decompose(x) in (c!parse c1 , d!parse c2);
end;;
```

`ProdCart` uses two parameters `c` and `d` specified by the interface of `Basic`. We can access to methods of `c` and `d` and to their carrier type. However, the carrier type of `c` and the one of `d` are incompatible.
The carrier type of `ProdCart` is defined as the cartesian product of the carrier type of `c` (`c!rep`) and the one of `d` (`d!rep`).
The interface of `ProdCart` is defined as follows:

```
I_ProdCart = fun (c:I_Basic) -> fun (d:I_Basic) ->
struct
 rep;
 parse : string -> rep;
end
```

This interface erases the definition of the carrier type and the definition of `parse`. It is also parameterized by `c` and `d`.

In order to parameterize a species $A$ by a parameterized species $B$, every parameter of the species $B$ has to be instantiated.

A parameterized species can be also refined by inheritance. For instance, `ProdCart` is derived as follows:

```
species ProdCart_R = fun (c is Basic_add) -> fun (d is Basic_add) ->
struct
 inherits (ProdCart c d);

 add : rep -> rep -> rep = fun x -> fun y ->
   let (e1 , e2) = x and (e3 , e4) = y in
   ( (c!add e1 e3) , (d!add e2 e4) );

 const : c!rep -> d!rep -> rep = fun x -> fun y -> ( x , y )
end;;
```

Since the interface of BASIC is included in the one of BASIC_ADD, the parameters of PRODCART can be instantiated vy D and `d`. So, the methods of `I_Basic` are retrieved in `Basic_add`.
The interface of `ProdCart_R` is obtained by unfolding the inheritance of `ProdCart(c,d)`:

```
I_ProdCart_R = fun (c : I_Basic_add) -> fun (d : I_Basic_add ) ->
 rep;
 parse : string -> rep ;
 add : rep -> rep -> rep ;
 const : c!rep -> d!rep -> rep ;
end
```

The definition of methods `add` and `const` are erased from the interface.

Toward the bottom of a hierarchy, complete species can be obtained. For $\mathcal{FML}$, a complete species is a species with no parameters, all methods defined and `rep` associated to a type. From `Basic_add`, we can obtain the complete species `Integers`:

```
species Integers =
struct
 inherits Basic_add ;

 rep = int;
 parse = fun x -> #int_of_string x;
 add   = fun x -> fun y -> x + y;
end;;
```

From a complete species, a collection can be created as follows:

```
collection My_Integers = Integers ;;
```

A collection has also an interface. The interface of `My_Integers` is given by the one of `Integers`:

```
I_My_Integers =
  rep;
  parse : string -> rep;
  add   : rep -> rep -> rep;
end
```

As the carrier type is opaque in a interface, an abstraction of the carrier type for a collection is obtained. Thus, we can only apply `add` on entities of `My_Integers`.

In the same way, a complete species `Booleans` can be obtained from `Basic_add` in order to implement algorithms for booleans. Then, a collection `My_Booleans` can be created from `Booleans`. Thus, the interface of `Basic_add` is included in the ones of `My_Integers` and `My_Booleans`. So, a new collection can be created as follows:

```
collection IntXBool = Prod_Cart_R  My_Integers My_Booleans ;;
```

`Prod_Cart_R` applied to the collections `My_Integers` and `My_Booleans` provides a complete species. Thus the collection `IntXBool` can be created. Its interface is:

```
I_IntXBool =
struct
 rep;
 parse : string -> rep ;
 add   : rep -> rep -> rep ;
 const : My_Integers!rep -> My_Booleans!rep -> rep ;
end
```

Above, the parameters `c` and `d` of `Prod_Cart_R` are respectively instantiated by `My_Integers` and `My_Booleans`.

The collections previously created can now be used in the following simple ML program:

```
let one_true    = IntXBool!parse "1 and true"  ;;
let zero_false  = IntXBool!parse "0 and false" ;;
let resu        = IntXBool!add one_true zero_false ;;
```

By using the method `parse`, entities `one_true` and `zero_false` of the collection `IntXBool` are created. In relation to the type of `parse` exported by the interface of `IntXBool`, `one_true` and `zero_false` have the type `IntXBool!rep`, that is the carrier type of `IntXBool`. The type of `add`, in relation to the interface of `IntXBool`, is `IntXBool!rep` → `IntXBool!rep` → `IntXBool!rep`. Thus, `one_true` and `zero_false` can be applied to `add`.
The species `(Prod_Cart_R My_Integers My_Booleans)` has the carrier type defined with `int` (carrier type of `My_Integers`) and `bool` (carrier type of `My_Booleans`). Thus, the type of method `add` has the type `int*bool` → `int*bool` → `int*bool`. However, the collection `IntXBool` has abstracted the type `int*bool`. Thus, in the above example, we cannot directly call `add` with `(1,#t)` and `(0,#f)`.

In addition to above features, FOCAL and $\mathcal{FML}$ allow multi-inheritance. To resolve the diamond problem, if a method is defined several times in a list of inherited species, by convention, it is the rightest definition that is retained. FOCAL and $\mathcal{FML}$ also provides method redefinition, the self reference and the late binding by default. Self reference is obtained by a variable `self` used to represent underlying collection inside a species.
Moreover, specifications of methods must not change during the inheritance, even if a method is (re)defined. In the same way, once the carrier type is defined, it must not be changed. These constraints avoid to break type soundness.
Lastly, collections are not first class objects: a method cannot be parameterized by collections. Collections are only applied to species.

## 3 Syntax

In this section, type expressions used to define specifications of method and interfaces are introduced. Then, abstract syntax to define a $\mathcal{FML}$ program is presented.

### 3.1 Type expressions and interfaces

*Type expressions* The type expressions are divided in the two following categories:

| | |
|---|---|
| carrier type definition: | |
| $t ::= \iota$ | atomic type |
| $\mid$ c!`rep` | carrier type of collection |
| $\mid t \to t \mid t * t$ | functional and product type |
| | |
| specification syntax: | |
| $i ::=$ `rep` | reference to carrier type |
| $\mid \iota \mid$ c!`rep` $\mid i \to i \mid i * i$ | |

The first category is used to define the carrier type. The second category allows to give a specification for a method.

A carrier type $t$ can be an atomic type $\iota$ (that is the type of constants), a carrier type c!`rep` of a collection c, a functional type $t \to t$ or a product type $t * t$. The definition for $i$ is included in the one of $t$. The only difference between the two definitions is `rep`. Using `rep` to define the carrier type does not have any sense in $\mathcal{FML}$.

*Interfaces* In the abstract syntax of $\mathcal{FML}$, parameters of species are directly specified by interfaces instead of using species. It avoids to overload the model too much. As we previously said, an interface associated to parameters is not parameterized. Thus, such an interface is simply represented here by a list of method names associated to specifications:

$$I ::= \emptyset \mid \mathsf{m}{:}i;\ I$$

$\emptyset$ represents empty interfaces.

### 3.2 Language of terms

The main terms are defined by extending ML with constructions for method invocations and constructions for species and collections:

| | | |
|---|---|---|
| $a$ | $::= \kappa \mid x \mid \lambda x.\ a \mid a\ a \mid (a, a) \mid \texttt{let}\ x = a\ \texttt{in}\ a$ | Core ML |
| | $\mid col!\mathsf{m}$ | Method invocation |
| | $\mid \texttt{species}\ \mathsf{S} = e\ \texttt{in}\ a$ | Species definition |
| | $\mid \texttt{collection}\ \mathfrak{c} = e\ \texttt{in}\ a$ | Creation of collection |
| $col$ | $\triangleq \mathfrak{c}$ | Collection |
| | $\mid \texttt{self}$ | *Self reference* |

In the first line, the core ML is defined from constants $\kappa$, term variables $x$, functions $\lambda x.\ a$, applications $a\ a$, pairs $(a, a)$ and locale declarations $\texttt{let}\ x = a\ \texttt{in}\ a$. We add the invocation method $col!\mathsf{m}$, the species definition $\texttt{species}\ \mathsf{S} = e\ \texttt{in}\ a$ and the creation of collection $\texttt{collection}\ \mathfrak{c} = e\ \texttt{in}\ a$.
The invocation method is limited only either on a collection $\mathfrak{c}$ or on the variable `self`. This syntactic constraint allows to forbid the collection to be a first class object. Here, the variable `self` is used for the self reference. In a species, this variable represents the family of underlying collections. However, in a species, a concrete implementation for the future collection is given. This implementation must be coherent in relation to the type of `self`. Hence, `self` represents the underlying collection that will be used during the execution. As it will be showed in the semantics, `self` is not substituted by the identifier of a collection, but by the value of this collection.
The construction $\texttt{species}\ \mathsf{S} = e\ \texttt{in}\ a$ is used to define a species $\mathsf{S}$ with a species expression $e$ (defined below). The scope of $\mathsf{S}$ is limited to the expression $a$.
The construction $\texttt{collection}\ \mathfrak{c} = e\ \texttt{in}\ a$ is used to defined a collection $\mathfrak{c}$ from the species $e$. The scope of $\mathfrak{c}$ is limited to the one of the expression $a$. Every collection name used to define a new collection must be fresh. This constraint provides the abstraction of elements of the collection.

The species expression is defined as follows:

| | | |
|---|---|---|
| $e$ | $::= \mathsf{S}$ | Species name |
| | $\mid \texttt{struct}\ w\ \texttt{end}$ | Structure |
| | $\mid \lambda(\mathfrak{c} : I).\ e$ | Parameterized species |
| | $\mid e\ col$ | Species application |
| $w$ | $::= \emptyset \mid d;\ w$ | |
| $d$ | $::= \texttt{rep}{=}t$ | Carrier type |
| | $\mid \mathsf{m}{:}i{=}a$ | Methode |
| | $\mid \texttt{inherit}\ e$ | Inherited species |

A species expression $e$ is used to explicitly define a species. It can be a species name $\mathsf{S}$, a structure $\texttt{struct}\ w\ \texttt{end}$, a parameterized species $\lambda(\mathfrak{c} : I).\ e$ or an application $e\ col$ of an species expression $e$ on a collection $a$.

In order to define the body of a species, a structure `struct` $w$ `end` (called for the sequel **species structure**) is used where $w$ is the list (eventually empty) of its components $d$. A component $d$ can be a carrier type definition `rep`=$t$, a method `m`:$i$=$a$ or a inherited species `inherit` $e$. In the previous introduction of $\mathcal{FML}$, the inherited species have appeared before the others components. Here, we are less restrictive on this order to ease the presentation. Thus, methods can appear before inherited species. However in this case, these methods cannot access to ones appearing in the inherited species.
In the section 2, we said that there is at most one declaration of carrier type in a species. However, thanks to the definition of $w$, several `rep`=$t_i$ can appear. It is the type system which restrains all $t_i$ to be the same. Thus, it is amounted to have one declaration of carrier type at maximum.

## 4 Type system

In addition to validate expressions, the type system presented here builds a fix point to obtain the self reference and offers an abstraction mechanism for entities of collections.

For the type system, the symbol $\alpha$ ranges over the set $Tvar$ of type variables. Moreover, the row variable $\rho$ is also used.

### 4.1 Type language

The type language is formally defined as:

**Basic type:**

| | | |
|---|---|---|
| $\tau$ | $::= \alpha$ | type variable |
| | $\mid \iota$ | atomic type |
| | $\mid$ `c!rep` | carrier type of a collection `c` |
| | $\mid \tau \rightarrow \tau \mid \tau * \tau$ | functional and product type |

**List of specification fields:**

$\omega \quad ::= \texttt{rep} \mid \alpha \mid \iota \mid \texttt{c!rep} \mid \omega \rightarrow \omega \mid \omega * \omega$
$\Phi \quad ::= \emptyset \mid m{:}\omega;\ \Phi \mid \texttt{rep}{=}\tau;\ \Phi$

Notations :
$\Phi_d \triangleq \Phi \backslash \{\texttt{rep}{=}\tau\}$
$\Phi_c \triangleq (\texttt{rep}{=}\tau;\ \Phi_d)$
$\Phi_e \triangleq \Phi_d \mid \Phi_d;\ \rho$

**Collection type:**

$\xi \quad ::= \langle \Phi_c \rangle \mid \langle \Phi_c;\ \rho \rangle$

**Species type:**

| | | |
|---|---|---|
| $\gamma$ | $::=$ `sig` $(\xi)\ \Phi$ `end` | structure type |
| | $\mid \xi \rightarrow \gamma$ | functional type |

The basic types $\tau$ are the types of expressions $a$. They can be a type variable $\alpha$, an atomic type (`int`, `bool`, etc ...), a carrier type $\mathfrak{c}$!`rep` of a collection $\mathfrak{c}$, a functional type $\tau \rightarrow \tau$ or product type $\tau * \tau$.

In the type system, the list $\Phi$ is used to represent the interface of a collection or a species. In order to return the method types, the list $\Phi$ contains at least a field `rep` $= \tau$. For every method with a specification $\omega$, we just have to replace `rep` by the type $\tau$ in $\omega$ in order to get the type $\tau'$ of the method.
Formally, a list $\Phi$ can be empty or contain a method specification m:$\omega$ and a carrier type definition `rep` $= \tau$. A specification $\omega$ can be a reference `rep` to the carrier type, a type variable $\alpha$, a carrier type $\mathfrak{c}$!`rep` of a collection $\mathfrak{c}$, a functional type or a product type $\omega * \omega$.
*We suppose that in a list $\Phi$, all the `rep` fields are associated to the same type $\tau$. This constraint comes from the interdiction to redefine the carrier type. Likewise, if several occurrences of a method* m *appear in a list $\Phi$, the specification $\omega$ associated is the same for all.*
We define a binary operator $\oplus$ to concatenate a list $\Phi_1$ to a list $\Phi_2$. Moreover, $\oplus$ checks the preservation of specifications and the carrier type. With the verification of the preservation of specifications, the preservation of types associated to methods are also verified in the same time. Thus, the type soundness is not broken.
Lastly, some notations are introduced for the list $\Phi$ in order to ease the presentation of rules and to avoid to add supplementary verification rules. Then $\Phi_d$ is a list deprived of all `rep` $= \tau$ occurrences. The notation $\Phi_c$ is a list $\Phi$ defined with an only one occurrence of `rep` $= \tau$. Then the list $\Phi_e$ denotes a list $\Phi_d$ or a list $\Phi_d$ followed by the row variable $\rho$.
In order to avoid to be constrained by the order induced by a list $\Phi$, a permutation axiom is given as follows:

$$s \triangleq \texttt{rep}{=}\tau \mid \texttt{m} : \omega$$
$$s_1; s_2; \Phi_1 = s_2; s_1; \Phi_1$$

The type $\xi$ is reserved for the collection and to define the type for the variable `self`. More precisely, $\xi$ can be the $\langle \Phi_c \rangle$ shape or the $\langle \Phi_c; \rho \rangle$ shape. The first shape gives a type to the variable `self` or a collection $\mathfrak{c}$. If $\langle \Phi_c \rangle$ denotes the type of a collection $\mathfrak{c}$, we use `rep` $= \mathfrak{c}$!`rep` in $\Phi_c$ in order to abstract the entities of the collection. The second shape is used for parameterized species. Here the row variable $\rho$ is used in case of a collection with an interface greater than expected by the specification of parameter.

The type $\gamma$ is the one of species. It can be either a type of structure `sig` $(\xi)$ $\Phi$ `end` or a type of parameterized species $\xi \rightarrow \gamma$.
The type of a structure `sig` $(\xi)$ $\Phi$ `end` is composed of a type $\xi$ and a list $\Phi$. The type $\xi$ will allow to assign a type for the variable `self`. It represents the type of the underlying collection but with a concrete form. So the `rep` field is defined with a $\tau$ type compatible with the one defined in the species. It allows to verify the compatibility of the interface of the underlying collection with the set of method types. For the sequel, we call the type $\xi$, the **signature** of species

type.
As for the list $\Phi$, it corresponds to the one of defined fields in the species. More precisely, the fields of $\Phi$ are the ones of the structure, but also the ones of inherited species. Lastly, the fields of $\Phi$ must appear in the type $\xi$. On the other hand, a declared method in $\xi$ does not necessarily appear in the list $\Phi$. In this case, this method corresponds to a declaration. In an analogous way, if the field `rep` is not present in $\Phi$, then the carrier type is not yet defined in the species. Lastly if the all fields declared in $\xi$ belong to $\Phi$, then the species is complete.
Afterwards, $\Phi$ is called **list of types of defined fields**.

The type schemas are defined as follows:

$$\sigma_\tau ::= \forall \bar{\alpha}.\tau$$
$$\sigma_\gamma ::= \forall \bar{\alpha}.\forall \rho.\gamma$$

where $\bar{\alpha}$ denotes the set (eventually empty) of type variables $\alpha_1 \ldots \alpha_n$. The row variable $\rho$ may be omitted.
We denote by $\tau \leq \sigma_\tau$ (respectively $\gamma \leq \sigma_\gamma$), that $\tau$ (resp. $\gamma$ ) is an instance of $\sigma_\tau$ (resp. $\sigma_\gamma$).

In $\mathcal{FML}$, several syntactic categories are used. In order to homogenize them, we introduce the following notations:

$$\breve{a} \triangleq a \mid w \mid col \mid e$$
$$\breve{\tau} \triangleq \tau \mid \Phi \mid \xi \mid \gamma$$

These notations must be used in a consistent way. For example $(\breve{a}, \breve{\tau})$ means $(a, \tau)$, $(w, \Phi)$, etc... but not $(e, \tau)$.

### 4.2 Typing rules

The typing rules for $\mathcal{FML}$ use two environments defined as follows:

- the typing environment $A$:

  $$A ::= \emptyset \mid A + x{:}\sigma_\tau \mid A + \mathsf{S}{:}\sigma_\gamma \mid A + \mathfrak{c}{:}\xi \mid A + \texttt{self}{:}\xi$$

  The environment $A$, possibly empty, is composed of term variables $x{:}\sigma_\tau$, species variables $\mathsf{S}{:}\sigma_\gamma$, collection names $\mathfrak{c}{:}\xi$ and $\texttt{self}{:}\xi$ variable.
- the collection name environment $\Omega$ defined like a set of collection names possibly empty.

We denote $A^*$ the typing environment $A$ deprived of any occurrence of `self` .
We call this environment, **starry environment**. Starry environment allows to control the scope of `self`. It allows to avoid the capture of `self` by an other species different to the one is typing.

An environment $A$ is considered as a partial function. Notations are used as follows:

- $A(x)$ for the type scheme $\sigma_\tau$ associated to the variable $x$ declared in $A$,

– $A(\mathsf{S})$ for the type scheme $\sigma_\gamma$ associated to the variable $\mathsf{S}$ declared in $A$,
– $A(\mathfrak{c})$ for the type $\xi$ associated to the collection name $\mathfrak{c}$ declared in $A$,
– $A(\texttt{self})$ for the type $\xi$ associated the `self` variable declared in $A$.

The environment $\Omega$ contains collection names declared with constructions `collection` $\mathfrak{c} =$ $e$ `in` $a$ or corresponding to parameters of a species. The types of variables declared in $A$ can only use collection names declared in $\Omega$ as types. That implies some coherence between $A$ and $\Omega$. So, a notion of well formed environment is introduced:

**Definition 1.** A typing environment $A$ is a `well formed` typing environment in relation to a collection name environment $\Omega$ iff for all $x{:}\sigma_\tau \in A$ (respectively for all $\mathsf{S}{:}\sigma_\gamma \in A$, for all $\mathfrak{c}{:}\xi \in A$, for all $\texttt{self}{:}\xi \in A$), all occurrences of collection names in $\sigma_\tau$ (respectively in $\sigma_\gamma$, in $\xi$) are declared in $\Omega$.

A generalization operator $Gen$ is defined:

$$Gen(\breve{\tau}, A) = \forall \alpha_1 \ldots \alpha_n . \breve{\tau}$$

with $\alpha_1 \ldots \alpha_n$ the free variable types of $\breve{\tau}$, except in $A$.

The typing rules must validate well typed expressions, but also type expressions used in terms. Indeed, for all carrier type $\mathfrak{c}!\texttt{rep}$ used in expression types, the collection name $\mathfrak{c}$ must be declared in the collection name environment $\Omega$. So, there is the function $|\texttt{-}|_\Omega : \{i\} \to \{\omega\} \cup \bot$ defined by induction as follows:

$$\begin{array}{ll} |\iota|_\Omega & = \iota \\ |\texttt{rep}|_\Omega & = \texttt{rep} \\ |\mathfrak{c}!\texttt{rep}|_\Omega & = \mathfrak{c}!\texttt{rep} \text{ si } \mathfrak{c} \in \Omega \\ |\mathfrak{c}!\texttt{rep}|_\Omega & = \bot \text{ si } \mathfrak{c} \notin \Omega \\ |i_1 \to i_2|_\Omega & = |i_1|_\Omega \to |i_2|_\Omega \\ |i_1 * i_2|_\Omega & = |i_1|_\Omega * |i_2|_\Omega \end{array}$$

The function $|\texttt{-}|_\Omega$ applied to a specification $i$, returns a type $\omega$ containing eventually occurrences of an error symbol $\bot$. The symbol $\bot$ corresponds to occurrences of $\mathfrak{c}!\texttt{rep}$ in $i$ for the ones $\mathfrak{c} \notin \Omega$. Thus the validate specifications correspond to the types without occurrence of $\bot$.
Note, the function $|\texttt{-}|_\Omega$ can be applied also on a carrier type definition $t$. Indeed the set $\{t\}$ is included in the set $\{i\}$.

The judgement used in the typing rules have the shape $A \ ; \ \Omega \vdash \breve{a} : \breve{\tau}$: the expression $\breve{a}$ has the type $\breve{\tau}$ in the environments $A$ and $\Omega$. To be validated, the environment $A$ of a judgement must be well formed in relation to $\Omega$. Moreover, collection names used like type in $\breve{\tau}$ must be declared in $\Omega$. For these constraints, a notion well formed judgement is introduced:

**Definition 2.** *A judgement $A \ ; \ \Omega \vdash \breve{a} : \breve{\tau}$ is **well formed** iff $A$ is well formed in relation to $\Omega$ and all occurrences of collection names in $\breve{\tau}$ are declared in $\Omega$.*

In the rules presented below, all judgments used must be considered well formed in relation to the environment $\Omega$. If a typing derivation tree created from typing rules, uses a judgment not well formed, then it is rejected.

Among the typing rules, ones to type ML expressions are retrieved:

$$\textsc{Cst}\ \frac{}{A\ ;\ \Omega \vdash \kappa : \iota} \qquad \textsc{Var}\ \frac{\tau \leq A(x)}{A\ ;\ \Omega \vdash x : \tau} \qquad \textsc{Fun-ML}\ \frac{A + x : \tau_1\ ;\ \Omega \vdash a : \tau_2}{A\ ;\ \Omega \vdash \lambda x.\ a : \tau_1 \rightarrow \tau_2}$$

$$\textsc{App-ML}\ \frac{A\ ;\ \Omega \vdash a_1 : \tau_1 \rightarrow \tau_2 \qquad A\ ;\ \Omega \vdash a_2 : \tau_1}{A\ ;\ \Omega \vdash a_1\ a_2 : \tau_2}$$

$$\textsc{Pair-ML}\ \frac{A\ ;\ \Omega \vdash a_1 : \tau_1 \qquad A\ ;\ \Omega \vdash a_2 : \tau_2}{A\ ;\ \Omega \vdash (a_1, a_2) : \tau_1 * \tau_2}$$

$$\textsc{Let-ML}\ \frac{A\ ;\ \Omega \vdash a_1 : \tau_1 \qquad A + x : Gen(\tau_1, A)\ ;\ \Omega \vdash a_2 : \tau_2}{A\ ;\ \Omega \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau_2}$$

Then add rules:

$$\textsc{Send}\ \frac{A\ ;\ \Omega \vdash col : \langle \texttt{rep}{=}\tau; \mathsf{m}{:}\omega; \Phi_d \rangle}{A\ ;\ \Omega \vdash col!\mathsf{m} : \omega[\texttt{rep} \leftarrow \tau]}$$

$$\textsc{Species Let}\ \frac{A\ ;\ \Omega \vdash e : \gamma \qquad A + \mathsf{S} : Gen(\gamma, A)\ ;\ \Omega \vdash a : \tau}{A\ ;\ \Omega \vdash \texttt{species}\ \mathsf{S} = e\ \texttt{in}\ a : \tau}$$

$$\textsc{Abstract}\ \frac{\begin{array}{c} A\ ;\ \Omega \vdash e : \texttt{sig}\ (\langle \texttt{rep}{=}\tau'; \Phi_d \rangle)\ (\texttt{rep} = \tau'; \Phi_d)\ \texttt{end} \\ A + \ \mathfrak{c} : \langle \texttt{rep}{=}\mathfrak{c}!\texttt{rep}; \Phi_d \rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash a : \tau \end{array}}{A\ ;\ \Omega \vdash \texttt{collection}\ \mathfrak{c} = e\ \texttt{in}\ a : \tau}$$

The rule Send is used to type the invocation of a method $\mathsf{m}$ on an expression *col* (collection or `self` variable). Intuitively, this rules consist in verifying the existence of the method $\mathsf{m}$. More formally, the type of *col* in the environment $A$ and $\Omega$, must have the form $\langle \texttt{rep}{=}\tau; \mathsf{m}{:}\omega; \Phi_d \rangle$ with $\mathsf{m}$ declared in this type. The type returned for $col!\mathsf{m}$, is the specification $\omega$ with every occurrence of `rep` replaced by the definition of the carrier type $\tau$ found in the type of *col*. In other words, the rule Send returns the type $\omega[\texttt{rep} \leftarrow \tau]$.

To type an expression $\texttt{species}\ \mathsf{S} = e\ \texttt{in}\ a$, we use the rule Species Let. It is similar to the one used to type an expression $\texttt{let}\ x = a\ \texttt{in}\ a$. The species $e$ must have the type $\gamma$ in environments $A$ and $\Omega$. Then, the expression $a$ is typed in the environment $A$ extended with $\mathsf{S} : Gen(\gamma, A)$ (the environment $\Omega$ stays fixed). The type of $a$, that is $\tau$, is the type returned by Species Let.

The rule ABSTRACT is used to type the creation of a collection `collection` $\mathtt{c} = e$ `in` $a$. Intuitively this rule must verify that the species $e$ is complete. Then, it must type the expression $a$ with the environment $A$ and $\Omega$ extended with the collection $\mathtt{c}$. During the typing phase of the expression $a$, the carrier type $\mathtt{c!rep}$ of the collection $\mathtt{c}$ plays the role of a formal variable different of all other variables and all types declared in the environments. Moreover, the scope of type $\mathtt{c!rep}$ is limited to the one of the expression $a$.
Formally, in environments $A$ and $\Omega$, the type of species $e$ must have the form $\mathtt{sig}\ (\langle \mathtt{rep}{=}\tau'; \Phi_d\rangle)\ \langle \mathtt{rep}{=}\tau'; \Phi_d\rangle\ \mathtt{end}$. In other words, the signature of this type must be the same than the list of types of defined fields. Hence, we guarantee that the species $e$ is complete. Then, the expression $a$ is typed in the environment $(\Omega; \mathtt{c})$ and the environment $A$ extended with $\mathtt{c}{:}\langle \mathtt{rep}{=}\mathtt{c!rep}; \Phi_d\rangle$. The type for the name $\mathtt{c}$ is descended from the signature of the species type. However, the definition of carrier type $\tau'$ has been replaced by $\mathtt{c}\ \mathtt{!rep}$ in order to abstract it.
Lastly, the rule ABSTRACT returns the type $\tau$ of the expression $a$. Moreover, the type $\tau$ must not possess any occurrence of type $\mathtt{c!rep}$. To guarantee this constraint, the environment $\Omega$ in the judgement of the conclusion, does not contain the collection name $\mathtt{c}$. *In other words, we constrain the judgement of the collection to be well formed only in relation to* $\Omega$.

For the collection names and the `self` variable, the rules are defined as follows:

COLLECTION NAME
$$\frac{}{A\ ;\ \Omega \vdash \mathtt{c} : A(\mathtt{c})}$$

SELF
$$\frac{}{A\ ;\ \Omega \vdash \mathtt{self} : A(\mathtt{self})}$$

To type species expressions, the rules are the following:

SPECIES NAME
$$\frac{\gamma \leq A(\mathsf{S})}{A\ ;\ \Omega \vdash \mathsf{S} : \gamma}$$

SPECIES BODY
$$\frac{A^* + \mathtt{self} : \xi\ ;\ \Omega \vdash w : \Phi}{A\ ;\ \Omega \vdash \mathtt{struct}\ w\ \mathtt{end} : \mathtt{sig}\ (\xi)\ \Phi\ \mathtt{end}}$$

SPECIES FUN
$$\frac{A + \ \mathtt{c} : \langle \mathtt{rep}{=}\mathtt{c!rep};\ \mathsf{m}_k{:}\omega_k\rangle\ ;\ (\Omega;\ \mathtt{c}) \vdash e : \gamma \qquad \text{where}\ \ |i_k|_\Omega = \omega_k}{A\ ;\ \Omega \vdash \lambda(\mathtt{c} : \mathsf{m}_k{:}i_k).\ e : \langle \mathtt{rep}{=}\alpha, \mathsf{m}_k{:}\omega_k;\ \rho\rangle{\rightarrow}\gamma[\mathtt{c!rep} \leftarrow \alpha]}$$

SPECIES APP
$$\frac{A\ ;\ \Omega \vdash e : \xi{\rightarrow}\gamma \qquad A\ ;\ \Omega \vdash \mathit{col} : \xi}{A\ ;\ \Omega \vdash e\ \mathit{col} : \gamma}$$

The rule SPECIES NAME and SPECIES APP are relatively classic.
The rule SPECIES NAME simply returns an instance $\gamma$ of type scheme $A(\mathsf{S})$. As for the rule SPECIES APP, for an expression $e$ *col*, checks that $e$ has the type $\xi \rightarrow \gamma$ and *col* has the type $\xi$ in the environments $A$ and $\Omega$.

The rule SPECIES FUN allows to type a parameterized species $\lambda(\mathtt{c} : \mathsf{m}_k{:}i_k).\ e$. Intuitively, like a ML function, the expression $e$ must be well typed in the environments $A$ and $\Omega$ extended with the identifier $\mathtt{c}$. So, the type associated to $\mathtt{c}$ must be coherent in relation to its interface $\mathsf{m}_k{:}i_k$. Formally, $e$ is typed in the

environment $A$ extended with $\mathtt{c} : \langle \mathtt{rep}{=}\mathtt{c!rep};\ \mathsf{m}_k{:}\omega_k \rangle$. On the one hand, in the type of $\mathtt{c}$, the field `rep` is equal to $\mathtt{c!rep}$ in order to get an abstract vision of parameter in $e$. And on the other hand, the $\mathsf{m}_k$ correspond to method names given in the interface associated to the parameter of species. Their types $\omega_k$ must correspond to types returned by the application of $|\text{-}|_\Omega$ on the specification $i_k$. For $|i_k|_\Omega$, $\Omega$ is not extended with the name $\mathtt{c}$. Indeed, the identifier $\mathtt{c}$ is bound in the expression $e$, but not in the interface of parameter.
For a standard function, the type of the expression $\lambda(\mathtt{c} : \mathsf{m}_k{:}i_k).\ e$ would have to be $\langle \mathtt{rep}{=}\mathtt{c!rep}, \mathsf{m}_k{:}\omega_k \rangle \to \gamma$ with $\gamma$, the type of the class $e$. However, we type the expression $e$ for all collection $\mathtt{c}$ whose the interface is $\mathsf{m}_k{:}\omega_k$. Thus, we must replace all occurrences of $\mathtt{c!rep}$ in $\gamma$, by a type variable $\alpha$. This type variable will be instantiated by the carrier type of the collection instantiating the parameter $\mathtt{c}$.
Lastly, an instance of parameter $\mathtt{c}$ must contain the interface $\mathsf{m}_k{:}\omega_k$ completed with a row variable $\rho$. The row variable will be instantiated by the types of supplementary methods of the collection instantiating the parameter $\mathtt{c}$.
In final, for the expression $\lambda(\mathtt{c} : \mathsf{m}_k{:}i_k).\ e$, the rule Species Fun returns the type $\langle \mathtt{rep}{=}\alpha, \mathsf{m}_k{:}\omega_k;\ \rho \rangle \to \gamma[\mathtt{c!rep} \leftarrow \alpha]$.

The rule Species Body allows to type the structure `struct` $w$ `end` of a species. The list $w$ of fields of the structure is typed in the environment $A^*$ extended with the variable `self`. Here the use of the starry environment allows to not interfere with the use of `self` reserved to type an other species.
The returned type for the structure is `sig` $(\xi)$ $\Phi$ `end`, with $\Phi$ the type of the list $w$. The type of `self` in the premise corresponds to the signature $\xi$ from returned type. By proceeding in this way, a fix point is built in order to obtain the self reference.

To finish, the list of fields of a species is typed with the following rules:

Basic
$$\frac{}{A\ ;\ \Omega \vdash \emptyset : \emptyset}$$

Then
$$\frac{A\ ;\ \Omega \vdash d : \Phi_1 \qquad A\ ;\ \Omega \vdash w : \Phi_2}{A\ ;\ \Omega \vdash d;\ w : \Phi_1 \oplus \Phi_2}$$

Inherit
$$\frac{A\ ;\ \Omega \vdash \mathtt{self} : \xi \qquad A^*\ ;\ \Omega \vdash e : \mathtt{sig}\ (\xi)\ \Phi\ \mathtt{end}}{A\ ;\ \Omega \vdash \mathtt{inherit}\ e : \Phi}$$

Carrier type
$$\frac{A\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}{=}\tau;\ \Phi_d \rangle \qquad \text{where } |t|_\Omega = \tau}{A\ ;\ \Omega \vdash \mathtt{rep} = t : (\mathtt{rep}{=}\tau)}$$

Method
$$\frac{\begin{array}{c} A\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}{=}\tau;\ \mathsf{m}{:}\omega;\ \Phi_d \rangle \\ A\ ;\ \Omega \vdash a : \omega[\mathtt{rep} \leftarrow \tau] \qquad \text{where } |i|_\Omega = \omega \end{array}}{A\ ;\ \Omega \vdash (\mathsf{m}{:}i = a) : (\mathsf{m} : \omega)}$$

The rule Then allows to type a list $d; w$ in relation to the order induced by the declaration of fields of this list. The head $d$ of this list is typed by the rules

INHERIT, CARRIER TYPE and METHOD. The rest $w$ is typed again by the rule THEN. This process is reiterated until to obtain the empty list $\emptyset$. The end is marked with the rule BASIC by returning simply the empty list.
The rule THEN returns the concatenation of the list $\Phi_1$, type of $d$, with the list $\Phi_2$, type of $w$. We use for this concatenation the operator $\oplus$ in order to check the type preservation in case of method redefinition. Likewise, the carrier type preservation must be verified.

The rule `Inherit` allows to type the field `inherit` $e$. For this, the species $e$ is typed in the environment $A^*$. The use of the starry environment is explained by the possibility to use a same inherited species in order to define different species. For example, suppose two species $e_1$ and $e_2$. These two species inherit from the species $e$. However the type of `self` in $e_1$ and in $e_2$ is not the same. Thus to type $e$, the correct type for `self` must be taken. This correct type corresponds to the species that is typing actually. In order to make the good choice, the rule `Inherit` imposes that the signature $\xi$ of the species type must be identical to the type of the variable `self` declared in the environment $A$.
Lastly the rule `Inherit` returns the list $\Phi$ of the type of the species $e$. The list $\Phi$ corresponds to defined methods (thus different of declarations) within the species $e$.

The rule CARRIER TYPE checks simply the agreement between the definition $t$ of `rep` $= t$ and the carrier type declared in the type of the variable `self`. The function $|\text{-}|_\Omega$ is applied to $t$ in order to do it. The returned type must be the type $\tau$ associated to the field `rep` in the type of `self`.

The rule `Method` allows to type the definition $a$ of a method `m`. Intuitively, the coherence between the underlying collection represented by `self` and the definition $a$ of the method must be verified. Moreover we must check the coherence of $a$ in relation to the specification $i$ associated to the method `m`. To do it, the type of `self` must be $\langle$`rep`$=\tau$; `m`$:\omega$; $\Phi_d\rangle$ in $A$. More precisely, the method `m`$:\omega$ must appear in the type of `self`. Then in the environment $A$, the type of $a$ must correspond to $\omega$ with the occurrences of `rep` replaced by the carrier type $\tau$, declared in the type of the variable `self`. Lastly the specification $i$ declared with `m`, must correspond to the specification $\omega$ declared in the type of `self`. To do it, the function $|\text{-}|_\Omega$ is applied on $i$. The returned type must be $\omega$ then.

## 5 Semantics

An operational semantics with a call by value strategy is proposed for $\mathcal{FML}$. The choice of call by value strategy comes from the one done for FOCAL.
A set of elementary rules is provided in order to reduce expressions in head term. In order to reduce deeply, contextual rules are provided also.

Classically, type expressions do not intervene in operational semantics. However, in order to prove the type soundness, types expressions must be used. More precisely, for a collection `c` created from a type species whose the carrier type is $t$, all occurrences of `c!rep` must be replaced by $t$. It allows to avoid to get free occurrences of `c!rep` after reduction of expressions. Thus after reduction,

expressions would always be well typed.
Here, we have decided to present the semantics with operations on type expressions. In order to retrieve a more traditional semantics, it is enough to remove all substitution operations on $\mathfrak{c}!\texttt{rep}$.

### 5.1 Values

The values of $\mathcal{FML}$ as defined as follows:

$$\begin{array}{lll} v & ::= & \kappa \mid \lambda x.\ a \mid (v, v) \\ \\ v_{col} & ::= & \langle v_w \rangle \\ \\ v_s & ::= & \texttt{struct}\ \ v_w\ \ \texttt{end} \\ & \mid & \lambda(\mathfrak{c} : I).\ e \\ \\ v_w & ::= & \emptyset \mid v_d;\ v_w \\ v_d & ::= & \mathsf{m}{:}i = a \mid \texttt{rep}{=}t \end{array}$$

Conditions on $v_w$ are imposed as follows:

- field names are not overloaded,
- there is one occurrence of $\texttt{rep} = t$ at most.

There are three categories of values corresponding respectively to ML values ($v$), values for collections ($v_{coll}$) and values for species ($v_s$).

The value $v_{col}$ is the one for a collection and called **collection value** for the sequel. Collection value $\langle v_w \rangle$ is analogous to the formalization of objects in [7] and ones used in [8].
The list $v_w$ comes from the species used to create the collection. This version of the collection is no more abstract. Indeed, it contains the field $\texttt{rep} = t$ coming from the species, that is not used nevertheless in the semantic rules.

For the species, there are two types of values $v_s$ called **species values** for the sequel. The value $\texttt{struct}\ \ v_w\ \ \texttt{end}$ corresponds to a species whose inheritance has been entirely resolved. Thus in the list $v_w$, no more field $\texttt{inherit}\ \ e$ is present yet. The value $\texttt{struct}\ \ v_w\ \ \texttt{end}$ does not necessarily correspond to a complete species. Thus, in $v_w$ the field $\texttt{rep} = t$ is not necessarily present. Lastly, the other species value is the parameterized species $\lambda(\mathfrak{c} : I).\ e$.

In the list $v_w$, method expressions are not values. Indeed, late binding and declarations (analogous to virtual methods of OO languages) are provided by $\mathcal{FML}$. Thus, the evaluation of definitions of methods is not possible.

At last, a list $v_w$ can be seen as a partial function. Thus we denote $v_w(\mathsf{m})$ for the definition associated to the method $\mathsf{m}$ declared in $v_w$. And we denote $v_w(\texttt{rep})$ the type expression associated to the field $\texttt{rep}$ declared in $v_w$. By language abuse, we denote $v_{col}(\texttt{rep})$ for $v_w(\texttt{rep})$ so that $v_{col} = \langle v_w \rangle$.
Then the domain $dom(v_w)$ of the list $v_w$ is defined as follows:

$$\begin{array}{ll} dom(\emptyset) & = \emptyset \\ dom(\texttt{rep} = t;\ v_w) & = \{\texttt{rep}\} \cup dom(v_w) \\ dom(\mathsf{m} = a;\ v_w) & = \{\mathsf{m}\} \cup dom(v_w) \end{array}$$

### 5.2 Rules

Among the elementary rules, there are two classic rules using $\beta$-reduction operation for the ML expressions :

$$\begin{array}{ll} 1\ (\lambda x.\ a)\ v & \rightarrow_\epsilon a[v/x] \\ 2\ \texttt{let}\ x = v\ \texttt{in}\ a & \rightarrow_\epsilon a[v/x] \end{array}$$

For the method invocation, the following rule is used:

$$3\ \langle v_w \rangle !\mathsf{m} \rightarrow_\epsilon v_w(\mathsf{m})[\langle v_w \rangle / \texttt{self}]$$

The reduction of the invocation of a method $\mathsf{m}$ is done on a collection value $\langle v_w \rangle$. The returned value corresponds to the expression $v_w(m)$ with every occurrence of `self` replaced by the collection value $\langle v_w \rangle$ itself. This rule is analogous to the one for the invocation of a method on an object as it is described in [7].

The next rule is destined for species definitions:

$$4\ \texttt{species}\ \mathsf{S} = v_s\ \texttt{in}\ a \rightarrow_\epsilon a[v_s/\mathsf{S}]$$

This rule is similar to the rule 2. Every occurrence of $\mathsf{S}$ in the expression $a$ is substituted by the species value $v_s$.

In order to evaluate the application of a parameterized species on a collection value, use the rule defined as follows:

$$5\ (\lambda(\mathfrak{c} : I).\ e)\ v_{col} \rightarrow_\epsilon e[v_{col}/\mathfrak{c}][v_{col}(\texttt{rep})/\mathfrak{c}!\texttt{rep}]$$

This rule is similar to the rule 1. Every occurrence of $\mathfrak{c}$ in the species $e$ is substituted by the collection value $v_{col}$. Moreover, in order to avoid to get free $\mathfrak{c}!\texttt{rep}$ in $e$ after reduction, they are substituted by the carrier type $v_{col}(\texttt{rep})$ from the collection $v_{col}$.

The reduction of creation of collection from complete species is done with the rule defined as follows:

$$6\ \texttt{collection}\ \mathfrak{c} = (\texttt{struct}\ v_w\ \texttt{end})\ \texttt{in}\ a \rightarrow_\epsilon a[\langle v_w \rangle/\mathfrak{c}][v_w(\texttt{rep})/\mathfrak{c}!\texttt{rep}]$$

To create a collection, the species must be value such that `struct` $v_w$ `end`. More precisely, inheritance and method redefinitions must have been resolved in order to create the collection. Note that it is not checked if `struct` $v_w$ `end` corresponds to a complete species but it is done by the typing.
Then every occurrence $\mathfrak{c}$ in the expression $a$ is substituted by the collection value $\langle v_w \rangle$. Moreover, in order to have no more free $\mathfrak{c}!\texttt{rep}$ in $e$ after reduction, they are replaced by the carrier type $v_w(\texttt{rep})$ defined in the species `struct` $v_w$ `end`.

Lastly, there is a set of rules in order to reduce inside a species:

$$\begin{array}{ll} 7\ \mathsf{m}{:}i{=}a;\ v_w & \rightarrow_\epsilon v_w \text{ if } \mathsf{m} \in dom(v_w) \\ 8\ \texttt{rep}{=}t;\ v_w & \rightarrow_\epsilon v_w \text{ if } \texttt{rep} \in dom(v_w) \\ 9\ \texttt{inherit}\ (\texttt{struct}\ v_w\ \texttt{end});\ w & \rightarrow_\epsilon v_w\ @\ w \end{array}$$

The two first rules respectively allow to treat the redefinition of methods and the overloading of `rep`. If the method `m` is already present in the rest $v_w$ of the list, then $v_w$ is returned. For the field $\texttt{rep} = t$, the principle is the same.
The last rule is used to resolve inheritance. For this, the species must be a structure value form in order to retrieve its list $v_w$. Thus $v_w$ can be concatenated to the rest $w$ of the list.
Lastly, in the case of redefinition of method or overloading of `rep`, the combined use of these three rules allows to choose the rightest definition among the ones proposed in the list of inherited species.

Lastly, the context rules are defined as follows:

$$\frac{a \rightarrow_\epsilon a'}{E[a] \rightarrow E[a']} \qquad \frac{e \rightarrow_\epsilon e'}{E[e] \rightarrow E[e']} \qquad \frac{w \rightarrow_\epsilon w'}{E[w] \rightarrow E[w']}$$

The context is defined as follows:

$$\begin{array}{lll} E & ::= & [] \mid \texttt{let } x = E \texttt{ in } a \mid E\ a \mid v\ E \mid (E, a) \mid (v, E) \\ & & \mid E_{col}!m \\ & & \mid \texttt{collection c} = E_e \texttt{ in } a \mid \texttt{species S} = E_e \texttt{ in } a \\ \\ E_{col} & ::= & \langle F \rangle \\ \\ E_e & ::= & [] \mid \texttt{struct } F \texttt{ end} \\ & & \mid E_e\ col \mid v_s\ E_{col} \\ \\ F & ::= & [] \mid F_d;\ w \mid v_w;\ F \\ F_d & ::= & \texttt{inherit } E_e \end{array}$$

$E$ is defined by sub contexts in order to take in consideration the different syntactic classes.

## 6 Type soundness

The type soundness is the property establishing that every well typed expression can be reduced to a value. Moreover, the type soundness assures that every invoked method possesses a definition.
Here, we present a short version of the proof of the type soundness. A complete version can be found in [9].

Just before present the proof, some considerations and notations are needed. The types for ML values are the ones given by the type system. Likewise, the types for the species values and for the lists $v_w$ (used for different values) are the ones given in the type system. On the other hand, the notion of type for collection value is defined here:

**Definition 3.** Under the hypothesis of a typing environment $A$ well formed in relation to an environment $\Omega$, a collection value $\langle v_w \rangle$ have for type $\langle \Phi_c \rangle$, denoted $A\ ;\ \Omega \vdash \langle v_w \rangle : \langle \Phi_c \rangle$, iff the judgement $A^* + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash v_w : \Phi_c$ is valid.

For commodity reasons, the above definition is summarized with this rule:

$$\textsc{Executive collection}\quad \frac{A^* + \texttt{self} : \langle \Phi_c \rangle \ ; \ \Omega \vdash v_w : \Phi_c}{A \ ; \ \Omega \vdash \langle v_w \rangle : \Phi_c}$$

Moreover, a typing rule is needed for invocation method on collection value:

$$\textsc{Send}\quad \frac{A \ ; \ \Omega \vdash \langle \texttt{rep}{=}t; v_w \rangle : \langle \texttt{rep}{=}\tau; m{:}\omega; \Phi_d \rangle \qquad \texttt{where } |t|_\Omega = \tau}{A \ ; \ \Omega \vdash \langle \texttt{rep}{=}t; v_w \rangle ! m : \omega[\texttt{rep} \leftarrow \tau]}$$

Lastly, some definitions, relations and notations are necessary for the proof.

**Definition 4.** *A type schema $\sigma$ is more general than a type schema $\sigma'$ iff all instance of $\sigma$ is also an instance of $\sigma'$.*

**Definition 5.**

- $a_1 \subset a_2$ (resp. $e_1 \subset e_2$) iff for all $A^*$, $\Omega$ and $\tau$, $(A^* \ ; \ \Omega \vdash a_1 : \tau)$ (resp. $(A^* \ ; \ \Omega \vdash e_1 : \gamma)$) implies $(A^* \ ; \ \Omega \vdash a_2 : \tau)$ (resp. $(A^* \ ; \ \Omega \vdash e_2 : \gamma)$).
- $w_1 \subset w_2$ iff for all $A$, $\Omega$ and $\Phi$, $(A \ ; \ \Omega \vdash w_1 : \Phi)$ implies $(A \ ; \ \Omega \vdash w_2 : \Phi)$.

For the sequel, substitutions on type variables are needed. However the types brought by such substitutions must be coherent in relation to the collection name environment. Thus on the one hand, a notion of well formed type in relation to a collection name environment must be defined. On the other hand, a notion of well formed substitution in relation to a collection name environment must be defined:

**Definition 6.** A type $\breve{\tau}$ is said **well formed** in relation to a collection name environment $\Omega$, if for all occurrences $\mathfrak{c}!\texttt{rep}$, the collection names $\mathfrak{c}$ are declared in $\Omega$.

**Definition 7.** A substitution $\theta = [\alpha_1 \rightarrow \tau_1, \ldots, \alpha_n \rightarrow \tau_n]$ is said **well formed** in relation to a collection name environment $\Omega$, if all $\tau_1 \ldots \tau_n$ are well formed in relation to $\Omega$.

In order to homogenize the different categories of contexts, the following notation is introduced:

$$\breve{E} \triangleq E \mid E_{col} \mid E_e \mid F \mid F_d$$

Before to prove the type soundness, some classic propositions are given:

**Proposition 1 (Stability of typing by reinforcement of hypotheses).** *Let $A$ and $A'$ two environments well formed in relation to a collection name environment $\Omega$ such that :*

- $dom(A) = dom(A')$
- $A'(\breve{x}) \geq A(\breve{x})$ *for all* $\breve{x} \in dom(A)$.

*Then* $(A\ ;\ \Omega \vdash \breve{a} : \breve{\tau})$ *implies* $(A'\ ;\ \Omega \vdash \breve{a} : \breve{\tau})$.

*Proof.* The proof is done by induction on $(A\ ;\ \Omega \vdash \breve{a} : \breve{\tau})$.

**Lemma 1.** *For all substitution* $\theta$ *and all environment* $A$*, one have:*

$$\theta(A^*) = \theta(A)^*$$

*Proof.* By definition $A = A^* + \texttt{self} : \xi$. Thus $\theta(A) = \theta(A^*) + \texttt{self} : \theta(\xi)$. Therefore $\theta(A)^* = \theta(A^*)$.

**Proposition 2.** *If* $A\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$*, then for all substitution of type variables* $\theta$ *well formed in relation to* $\Omega$*, then* $\theta(A)\ ;\ \Omega \vdash \breve{a} : \theta(\breve{\tau})$.

*Proof.* The proof is done by induction on $A\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$ from different typing rules.

**Proposition 3 (Typing indifference in relation to useless hypotheses).** *Let* $A$ *and* $A'$ *two typing environments,* $\Omega$ *a collection name environment and* $\breve{a}$ *an expression such that:*

- $A$ *and* $A'$ *are well formed in relation to* $\Omega$*.*
- $A(\breve{x}) = A'(\breve{x})$ *for all free variable* $\breve{x}$ *of* $\breve{a}$*.*

*Then* $A\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$ *implies* $A'\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$

*Proof.* The proof is done by induction on $A\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$ from different typing rules.

**Proposition 4.** *For all context* $\breve{E}$*, if* $\breve{a}_1 \subset \breve{a}_2$ *then* $\breve{E}[\breve{a}_1] \subset \breve{E}[\breve{a}_1]$.

*Proof.* The proof is simply done by induction on $\breve{E}$.

For the sequel, notations are introduced as follows:

$$\tilde{a} \triangleq a \mid e$$
$$\tilde{\tau} \triangleq \tau \mid \gamma$$

These notations must be used in consistent way. Thus $(\tilde{a}, \tilde{\tau})$ must be understand as $(a, \tau)$, $(e, \gamma)$ but not as $(a, \gamma)$ and $(e, \tau)$.

**Lemma 2 (Substitution des variables de terme).** *Let* $\Omega$ *a collection name environment,* $A$ *a typing environment well formed in relation to* $\Omega$*,* $\tilde{a}_1$ *and* $\breve{a}_2$ *expressions,* $\tilde{\tau}_1$ *and* $\breve{\tau}_2$ *well formed in relation to* $\Omega$*.*
*If* $A^*\ ;\ \Omega \vdash \tilde{a}_1 : \tilde{\tau}_1$*,* $A + \tilde{x} : \forall \alpha_1 \ldots \alpha_n . \breve{\tau}_1\ ;\ \Omega \vdash \breve{a}_2 : \breve{\tau}_2$*,* $\alpha_1, \ldots, \alpha_n$ *are not free type variables in* $A$ *and bound variables in* $\breve{a}_2$ *are not free in* $\tilde{a}_1$*, then* $A\ ;\ \Omega \vdash \breve{a}_2[\tilde{a}_1/\tilde{x}] : \breve{\tau}_2$.

*Proof.* The proof is done by induction on $\breve{a}_2$ and by derivation of $A + \tilde{x} : \forall\alpha_1 \ldots \alpha_n.\tilde{\tau}_1 \ ; \ \Omega \vdash \breve{a}_2 : \tilde{\tau}_2$. It is relatively classic, however some less for constructions of $\mathcal{FML}$.
We show some examples in relation to $\mathcal{FML}$. For this, we note $A_{\tilde{x}}$ for $A + \tilde{x} : \forall\alpha_1 \ldots \alpha_n.\tilde{\tau}_1$.

**Case** $\breve{a}_2 \triangleq col!\mathtt{m}$ :

Let the following derivation:

$$\frac{\textsc{Send}\quad A_{\tilde{x}} \ ; \ \Omega \vdash col : \langle \mathtt{rep}{=}\tau; m{:}\omega; \Phi_d\rangle \ \ (\mathbf{1})}{A_{\tilde{x}} \ ; \ \Omega \vdash col!m : \omega[\mathtt{rep}{\leftarrow}\tau]}$$

By induction hypothesis applied on the expression *col* of the premiss (**1**), we get:

$$\frac{\textsc{Send}\quad A \ ; \ \Omega \vdash col[\tilde{a}_1/\tilde{x}] : \langle \mathtt{rep}{=}\tau; m{:}\omega; \Phi_d\rangle}{A \ ; \ \Omega \vdash col[\tilde{a}_1/\tilde{x}]!m : \omega[\mathtt{rep}{\leftarrow}\tau]}$$

Thus $A \ ; \ \Omega \vdash (col!m)[\tilde{a}_1/\tilde{x}] : \omega[\mathtt{rep}{\leftarrow}\tau]$

**Case** $\breve{a}_2 \triangleq \mathtt{collection\ c} = e \ \mathtt{in}\ a$ :

Let the following derivation:

$$\frac{\begin{array}{c}\textsc{Abstract}\\ A_{\tilde{x}} \ ; \ \Omega \vdash e : \mathtt{sig}\ (\langle \mathtt{rep}{=}\tau'; \Phi_d\rangle)\ (\mathtt{rep} = \tau'; \Phi_d)\ \mathtt{end} \\ A_{\tilde{x}} + \mathfrak{c}{:}\langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep}; \Phi_d\rangle \ ; \ (\Omega; \ \mathfrak{c}) \vdash a : \tau\end{array}}{A_{\tilde{x}} \ ; \ \Omega \vdash \mathtt{collection\ c} = e \ \mathtt{in}\ a : \tau}$$

If $\alpha_i$ appear in $\Phi_d$, then they can be renamed in $\beta_i$ not free in $A$ and distinct of $\alpha_i$ thanks to the substitution $\theta = [\alpha_i \leftarrow \beta_i]$. And if the $\alpha_i$ don't appear in the type $\Phi_d$, then the identity function is taken for the substitution $\theta$.
By the proposition 2 applied on the premises of the above derivation, we get:

$$\frac{\begin{array}{c}\textsc{Abstract}\\ \theta(A_{\tilde{x}}) \ ; \ \Omega \vdash e : \theta(\ \mathtt{sig}\ (\langle \mathtt{rep}{=}\tau'; \Phi_d\rangle)\ (\mathtt{rep} = \tau'; \Phi_d)\ \mathtt{end}\ ) \\ \theta(A_{\tilde{x}} + \mathfrak{c}{:}\langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep}; \Phi_d\rangle) \ ; \ (\Omega; \ \mathfrak{c}) \vdash a : \theta(\tau)\end{array}}{\theta(A_{\tilde{x}}) \ ; \ \Omega \vdash \mathtt{collection\ c} = e \ \mathtt{in}\ a : \theta(\tau)}$$

As $\alpha_i$ and $\beta_i$ are not free $A$, then:

$$\frac{\begin{array}{c}\textsc{Abstract}\\ A_{\tilde{x}} \ ; \ \Omega \vdash e : \mathtt{sig}\ (\langle \mathtt{rep}{=}\theta(\tau'); \theta(\Phi_d)\rangle)\ (\mathtt{rep} = \theta(\tau'); \theta(\Phi_d))\ \mathtt{end}\ \ (\mathbf{1}) \\ A_{\tilde{x}} + \mathfrak{c}{:}\langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep}; \theta(\Phi_d)\rangle \ ; \ (\Omega; \ \mathfrak{c}) \vdash a : \theta(\tau)\ \ (\mathbf{2})\end{array}}{A_{\tilde{x}} \ ; \ \Omega \vdash \mathtt{collection\ c} = e \ \mathtt{in}\ a : \theta(\tau)}$$

By induction hypothesis applied on the expression $e$ of the premiss (**1**):

$$A \ ; \ \Omega \vdash e[\tilde{a}_1/\tilde{x}] : \mathtt{sig}\ (\langle \mathtt{rep}{=}\theta(\tau'); \theta(\Phi_d)\rangle)\ (\mathtt{rep} = \theta(\tau'); \theta(\Phi_d))\ \mathtt{end}$$

However, the induction hypothesis can't be applied on the expression $a$ of the premiss (**2**). The hypothesis $A^*\ ;\ \Omega \vdash \tilde{a} : \tilde{\tau}_1$ is extended so that $A^*\ ;\ (\Omega;\ \mathfrak{c}) \vdash \tilde{a} : \tilde{\tau}_1$. It is valid because the name $\mathfrak{c}$ is fresh in relation to the environment $\Omega$. Then $A^*\ ;\ (\Omega;\ \mathfrak{c}) \vdash \tilde{a} : \tilde{\tau}_1$ is extended so that $A^* + \ \mathfrak{c}{:}\langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep}; \theta(\Phi_d)\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash \tilde{a} : \tilde{\tau}_1$. This extension is possible thanks to the proposition 3. Indeed, the environment $A^* + \mathfrak{c}{:}\langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep}; \theta(\Phi_d)\rangle$ is well formed in relation to $(\Omega;\ \mathfrak{c})$.
Thus the induction hypothesis can be applied on the expression $a$ of the premiss (**2**):

$$A + \ \mathfrak{c}{:}\langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep}; \theta(\Phi_d)\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash a[\tilde{a}_1/\tilde{x}] : \theta(\tau)$$

Thus a derivation is obtained as follows:

$$\frac{\begin{array}{c}\textsc{Abstract}\\ A\ ;\ \Omega \vdash e[\tilde{a}_1/\tilde{x}] : \mathtt{sig}\ (\langle \mathtt{rep}{=}\theta(\tau'); \theta(\Phi_d)\rangle)\ (\mathtt{rep} = \theta(\tau'); \theta(\Phi_d))\ \mathtt{end} \\ A + \ \mathfrak{c}{:}\langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep}; \theta(\Phi_d)\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash a[\tilde{a}_1/\tilde{x}] : \theta(\tau)\end{array}}{A\ ;\ \Omega \vdash \mathtt{collection}\ \mathfrak{c} = e[\tilde{a}_1/\tilde{x}]\ \mathtt{in}\ a[\tilde{a}_1/\tilde{x}] : \theta(\tau)}$$

By the proposition 2 applied on the conclusion of the above derivation, with the inverse renaming of $\theta$, we get:

$$A\ ;\ \Omega \vdash \mathtt{collection}\ \mathfrak{c} = e[\tilde{a}_1/\tilde{x}]\ \mathtt{in}\ a[\tilde{a}_1/\tilde{x}] : \tau$$

That is:

$$A\ ;\ \Omega \vdash (\mathtt{collection}\ \mathfrak{c} = e\ \mathtt{in}\ a)[\tilde{a}_1/\tilde{x}] : \tau$$

**Case** $\breve{a}_2 \triangleq \mathtt{struct}\ \ w\ \ \mathtt{end}$ :

Let the following derivation:

$$\frac{\begin{array}{c}\textsc{Species Body}\\ A^*_{\tilde{x}} + \mathtt{self} : \xi\ ;\ \Omega \vdash w : \Phi\ \ {}_{(\mathbf{1})}\end{array}}{A_{\tilde{x}}\ ;\ \Omega \vdash \mathtt{struct}\ \ w\ \ \mathtt{end} : \mathtt{sig}\ (\xi)\ \Phi\ \mathtt{end}}$$

Remark:

$$(A^* + \mathtt{self} : \xi)^* = A^*$$

Thus the hypothesis $A^*\ ;\ \Omega \vdash \tilde{a}_1 : \tilde{\tau}_1$ is equivalent to $(A^* + \mathtt{self} : \xi)^* \vdash \tilde{a}_1 : \tilde{\tau}_1$. Thus, the induction hypothesis on the expression $w$ of the premiss (**1**). Then:

$$\frac{\begin{array}{c}\textsc{Species Body}\\ A^* + \mathtt{self} : \xi\ ;\ \Omega \vdash w[\tilde{a}_1/\tilde{x}] : \Phi\end{array}}{A\ ;\ \Omega \vdash \mathtt{struct}\ \ w[\tilde{a}_1/\tilde{x}]\ \ \mathtt{end} : \mathtt{sig}\ (\xi)\ \Phi\ \mathtt{end}}$$

That is:

$$A\ ;\ \Omega \vdash (\mathtt{struct}\ \ w\ \ \mathtt{end})[\tilde{a}_1/\tilde{x}] : \mathtt{sig}\ (\xi)\ \Phi\ \mathtt{end}$$

**Case** $\breve{a}_2 \triangleq \lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e$ :

Let the following derivation:

$$\frac{A_{\tilde{x}} + \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash e : \gamma \qquad \text{where}\ \ |i_k|_\Omega = \omega_k\ \ \textbf{(1)}}{A_{\tilde{x}}\ ;\ \Omega \vdash \lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e : \langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e\rangle{\rightarrow}\gamma[\mathfrak{c}!\mathtt{rep} \leftarrow \tau']}\ \textsc{Species Fun}$$

The induction hypothesis can't be applied directly on the expression $e$ of the premiss (**1**). At begin, the hypothesis $A^*\ ;\ \Omega \vdash \breve{a}_1 : \breve{\tau}_1$ is extended so that $A^*\ ;\ (\Omega;\ \mathfrak{c}) \vdash \breve{a}_1 : \breve{\tau}_1$. It is valid because the name $\mathfrak{c}$ is fresh in relation to the environment $\Omega$. Then, $A^*\ ;\ (\Omega;\ \mathfrak{c}) \vdash \breve{a}_1 : \breve{\tau}_1$ is extended so that $A^* + \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash \breve{a}_1 : \breve{\tau}_1$. This extension is possible thanks to the proposition 3. Indeed the environment $A^* + \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle$ is well formed in relation to $(\Omega;\ \mathfrak{c})$.
Lastly, the types $\omega_k$ don't possess free type variable. Indeed, on the one hand, there is $|i_k|_\Omega = \omega_k$. On the other hand, the specifications $i_k$ don't possess type variable. Thus the variables $\alpha_1 \ldots \alpha_n$ are not free in $A^* + \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle$. Thus the induction hypothesis can be applied on the expression $a$ of the premiss (**1**). Then:

$$\frac{A + \ \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash e[\tilde{a}_1/\tilde{x}] : \gamma \qquad \text{where}\ \ |i_k|_\Omega = \omega_k}{A\ ;\ \Omega \vdash \lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e[\tilde{a}_1/\tilde{x}] : \langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e\rangle{\rightarrow}\gamma[\mathfrak{c}!\mathtt{rep} \leftarrow \tau']}\ \textsc{Species Fun}$$

That is:

$$A\ ;\ \Omega \vdash (\lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e)[\tilde{a}_1/\tilde{x}] : \langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e\rangle{\rightarrow}\gamma[\mathfrak{c}!\mathtt{rep} \leftarrow \tau']$$

**Lemma 3.** *Let $\Omega$ a collection name environment, $\mathfrak{c}$ a collection name, $\tau$ a well formed type in relation $\Omega$, $\omega$ a well formed type in relation to $(\Omega;\ \mathfrak{c})$.*
*If $|i|_\Omega = \tau$ et $|i|_{(\Omega;\ \mathfrak{c})} = \omega$, then $|i[t/\mathfrak{c}!\mathtt{rep}]|_\Omega = \omega[\tau/\mathfrak{c}!\mathtt{rep}]$.*

*Proof.* The proof is done by induction on $|i|_{(\Omega;\ \mathfrak{c})} = \omega$. There is an only case that can be delicate:

**Case** $i \triangleq \mathfrak{c}'!\mathtt{rep}$ :

There are two cases:

- case $\mathfrak{c}'$ is different to $\mathfrak{c}$:
  There is $|\mathfrak{c}'!\mathtt{rep}|_{(\Omega;\ \mathfrak{c})} = \mathfrak{c}'!\mathtt{rep}$ with $\mathfrak{c}' \in (\Omega;\ \mathfrak{c})$. As $\mathfrak{c}'$ is different to $\mathfrak{c}$, then $\mathfrak{c}' \in \Omega$. Thus $|\mathfrak{c}'!\mathtt{rep}|_\Omega = \mathfrak{c}'!\mathtt{rep}$.
  On the other hand, there is $\mathfrak{c}'!\mathtt{rep}[t/\mathfrak{c}!\mathtt{rep}] = \mathfrak{c}'!\mathtt{rep}$ and $\mathfrak{c}'!\mathtt{rep}[\tau/\mathfrak{c}!\mathtt{rep}] = \mathfrak{c}'!\mathtt{rep}$. Thus $|\mathfrak{c}'!\mathtt{rep}[t/\mathfrak{c}!\mathtt{rep}]|_\Omega = \mathfrak{c}'!\mathtt{rep}[\tau/\mathfrak{c}!\mathtt{rep}]$.
- case $\mathfrak{c}'$ is equal to $\mathfrak{c}$ :
  In this case, there are $\mathfrak{c}'!\mathtt{rep}[t/\mathfrak{c}!\mathtt{rep}] = t$ and $\mathfrak{c}'!\mathtt{rep}[\tau/\mathfrak{c}!\mathtt{rep}] = \tau$.
  Then by hypothesis, there is $|t|_\Omega = \tau$. Thus $|\mathfrak{c}'!\mathtt{rep}[t/\mathfrak{c}!\mathtt{rep}]|_\Omega = \mathfrak{c}'!\mathtt{rep}[\tau/\mathfrak{c}!\mathtt{rep}]$.

**Lemma 4 (substitution des noms de collection).** *Let $\Omega$ a collection name environment, $\mathfrak{c}$ a collection name, $A$ a typing environment well formed in relation to $(\Omega;\ \mathfrak{c})$, $\breve{a}_2$ an expression, $\breve{\tau}_2$ a well formed type in relation to $\Omega$, $\langle \mathit{rep}{=}t; v_w\rangle$ a collection value, $\tau$ a well formed type in relation to $\Omega$, $\Phi_c$ a list of well formed types in relation to $\Omega$.*
*If $A^*\ ;\ (\Omega;\ \mathfrak{c}) \vdash \langle \mathit{rep}{=}t; v_w\rangle : \langle \mathit{rep}{=}\tau; \Phi_c\rangle$, $A+\ \mathfrak{c} : \langle \mathit{rep}{=}\mathfrak{c}!\mathit{rep}; \Phi_c\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash \breve{a}_2 : \breve{\tau}_2$, $|t|_\Omega = \tau$, bound variables of $\breve{a}_2$ are not free in $\langle \mathit{rep}{=}t; v_w\rangle$, then:*

$$A[\tau/\mathfrak{c}!\mathit{rep}]\ ;\ \Omega \vdash \breve{a}_2[\langle rep{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathit{rep}] : \breve{\tau}_2[\tau/\mathfrak{c}!\mathit{rep}]$$

*Proof.* The proof resembles to the one of lemma 2. *A priori*, we would be able to combine these two lemmas together. However, we have preferred to separate them in order to avoid to overload notations too much.
The lemma is proved by induction on $\breve{a}_2$ and by derivation of $A+\ \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep}; \Phi_c\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash \breve{a}_2 : \breve{\tau}_2$. The lemma 4 is used as needed.
For the proof, note $A_\mathfrak{c}$ for $A + \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep}; \Phi_c\rangle$ and $[A]$ for $A[\tau/\mathfrak{c}!\mathtt{rep}]$.
Here, show some cases for examples.

**Case** $\breve{a}_2 \triangleq \mathfrak{c}'$ :

Let a derivation for $A_\mathfrak{c}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \breve{a}_2 : \breve{\tau}_2$:

$$\frac{}{A_\mathfrak{c}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \mathfrak{c}' : A_\mathfrak{c}(\mathfrak{c}')}\ \textsc{Collection name}$$

There are two cases:

- case $\mathfrak{c}$ is equal to $\mathfrak{c}'$:
  Then $\mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] = \langle \mathtt{rep}{=}t; v_w\rangle$. Thus by hypothesis:

  $$A\ ;\ (\Omega;\ \mathfrak{c}) \vdash \mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle$$

  As $\langle \mathtt{rep}{=}\tau; \Phi_c\rangle$ is well formed in relation to $\Omega$ and doesn't contain occurrence of $\mathfrak{c}!\mathtt{rep}$, then:

  $$\langle \mathtt{rep}{=}\tau; \Phi_c\rangle = \langle \mathtt{rep}{=}\tau; \Phi_c\rangle[\tau/\mathfrak{c}!\mathtt{rep}]$$

  Thus:

  $$A\ ;\ (\Omega;\ \mathfrak{c}) \vdash \mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle[\tau/\mathfrak{c}!\mathtt{rep}]$$

  By application of $[\tau/\mathfrak{c}!\mathtt{rep}]$ on $A$, then:

  $$[A]\ ;\ \Omega \vdash \mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle[\tau/\mathfrak{c}!\mathtt{rep}]$$

- case $\mathfrak{c}$ is different to $\mathfrak{c}'$ :
  Then:

  $$\mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] = \mathfrak{c}'$$

  Thus by hypothesis:

  $$A_\mathfrak{c}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : A_\mathfrak{c}(\mathfrak{c}')$$

More precisely:

$$A_{\mathfrak{c}} \ ; \ (\Omega; \ \mathfrak{c}) \vdash \mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : A(\mathfrak{c}')$$

By application of $[\tau/\mathfrak{c}!\mathtt{rep}]$ on $A_{\mathfrak{c}}$, then:

$$A[\tau/\mathfrak{c}!\mathtt{rep}] + \ \mathfrak{c}{:}\langle \mathtt{rep}{=}\tau; \Phi_d\rangle \ ; \ (\Omega; \ \mathfrak{c}) \vdash \mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (A[\tau/\mathfrak{c}!\mathtt{rep}])(\mathfrak{c}')$$

By hypothesis, the environment $A[\tau/\mathfrak{c}!\mathtt{rep}]$ is well formed in relation to $(\Omega; \ \mathfrak{c})$. Thus by the proposition 3:

$$A[\tau/\mathfrak{c}!\mathtt{rep}] \ ; \ (\Omega; \ \mathfrak{c}) \vdash \mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (A[\tau/\mathfrak{c}!\mathtt{rep}])(\mathfrak{c}')$$

By application $[\tau/\mathfrak{c}!\mathtt{rep}]$ on the environment $A$, the environment $A[\tau/\mathfrak{c}!\mathtt{rep}]$ and the type $(A[\tau/\mathfrak{c}!\mathtt{rep}])(\mathfrak{c}')$ are well formed in relation to $\Omega$. Indeed, there are no any occurrences of $\mathfrak{c}!\mathtt{rep}$. Then:

$$[A] \ ; \ \Omega \vdash \mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (A[\tau/\mathfrak{c}!\mathtt{rep}])(\mathfrak{c}')$$

Thus:

$$[A] \ ; \ \Omega \vdash \mathfrak{c}'[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : A(\mathfrak{c}')[\tau/\mathfrak{c}!\mathtt{rep}]$$

**Case** $\breve{a}_2 \triangleq col!\mathsf{m}$ :

Let a derivation for $A_{\mathfrak{c}} \ ; \ (\Omega; \ \mathfrak{c}) \vdash \breve{a}_2 : \breve{\tau}_2$:

$$\frac{\textsc{Send} \quad A_{\mathfrak{c}} \ ; \ (\Omega; \ \mathfrak{c}) \vdash col : \langle \mathtt{rep}{=}\tau'; \mathsf{m}{:}\omega'; \Phi'_d\rangle}{A_{\mathfrak{c}} \ ; \ (\Omega; \ \mathfrak{c}) \vdash col!\mathsf{m} : \omega'[\mathtt{rep}{\leftarrow}\tau']}$$

By induction hypothesis applied on $col$, then:

$$\frac{\textsc{Send} \quad [A] \ ; \ \Omega \vdash col[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \langle \mathtt{rep}{=}\tau'[\tau/\mathfrak{c}!\mathtt{rep}]; \mathsf{m}{:}\omega'[\tau/\mathfrak{c}!\mathtt{rep}]; \Phi'_d[\tau/\mathfrak{c}!\mathtt{rep}]\rangle}{[A] \ ; \ \Omega \vdash col[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}]!\mathsf{m} : \omega'[\tau/\mathfrak{c}!\mathtt{rep}][\mathtt{rep}{\leftarrow}\tau'[\tau/\mathfrak{c}!\mathtt{rep}]]}$$

That is:

$$[A] \ ; \ \Omega \vdash (col!\mathsf{m})[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (\omega'[\mathtt{rep}{\leftarrow}\tau'])[\tau/\mathfrak{c}!\mathtt{rep}]$$

**Case** $\breve{a}_2 \triangleq \mathtt{collection} \ \mathfrak{c}' = e \ \mathtt{in} \ a$ :

Let a derivation for $A_{\mathfrak{c}} \ ; \ (\Omega; \ \mathfrak{c}) \vdash \breve{a}_2 : \breve{\tau}_2$:

$$\frac{\textsc{Abstract} \quad \begin{array}{c} A_{\mathfrak{c}} \ ; \ (\Omega; \ \mathfrak{c}) \vdash e : \mathtt{sig} \ (\langle \mathtt{rep}{=}\tau'; \Phi'_d\rangle) \ (\mathtt{rep} = \tau'; \Phi_d) \ \mathtt{end} \ {}_{(1)} \\ A_{\mathfrak{c}} + \mathfrak{c}'{:}\langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep}; \Phi'_d\rangle \ ; \ (\Omega; \ \mathfrak{c}; \ \mathfrak{c}') \vdash a : \tau'' \ {}_{(2)} \end{array}}{A_{\mathfrak{c}} \ ; \ (\Omega; \ \mathfrak{c}) \vdash \mathtt{collection} \ \mathfrak{c}' = e \ \mathtt{in} \ a : \tau''}$$

**Remark**: By definition on $\Omega$, the collection name $\mathfrak{c}'$ is different to $\mathfrak{c}$.
For the sequel, note $\Phi''_d$ for $\Phi'_d[\tau/\mathfrak{c}!\mathtt{rep}]$ and $\tau'''$ for $\tau'[\tau/\mathfrak{c}!\mathtt{rep}]$.

By induction hypothesis applied on $e$ of the premiss (**1**), we get:

$$[A]\ ;\ \Omega \vdash e[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \mathtt{sig}\ (\langle \mathtt{rep}{=}\tau'''; \Phi''_d\rangle)\ (\mathtt{rep} = \tau'''; \Phi''_d)\ \mathtt{end}$$

In order to apply the induction hypothesis on the expression $a$ of the premiss (**2**), the hypothesis $A^*\ ;\ (\Omega;\ \mathfrak{c}) \vdash \langle \mathtt{rep}{=}t; v_w\rangle : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle$ is extended so that $A^*\ ;\ (\Omega;\ \mathfrak{c}; \mathfrak{c}') \vdash \langle \mathtt{rep}{=}t; v_w\rangle : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle$ ($\Omega$ has been extended with the name $\mathfrak{c}'$). This extension is valid because $A$ is well formed in relation to $(\Omega;\ \mathfrak{c})$. Thus $A$ doesn't possess occurrences of $\mathfrak{c}'!\mathtt{rep}$. Thus $A$ is well formed in relation to $(\Omega;\ \mathfrak{c};\ \mathfrak{c}')$ also. Then, $A^*\ ;\ (\Omega;\ \mathfrak{c}; \mathfrak{c}') \vdash \langle \mathtt{rep}{=}t; v_w\rangle : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle$ is extended so that $A^* + \ \mathfrak{c}':\langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep}; \Phi'_d\rangle\ ;\ (\Omega;\ \mathfrak{c}; \mathfrak{c}') \vdash \langle \mathtt{rep}{=}t; v_w\rangle : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle$. These extension is valid by the proposition 3. Indeed $A^* + \ \mathfrak{c}':\langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep}; \Phi'_d\rangle$ is well formed in relation to $(\Omega;\ \mathfrak{c};\ \mathfrak{c}')$. Thus by induction hypothesis applied on $a$, then:

$$[A] + \ \mathfrak{c}':\langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep}; \Phi''_d\rangle\ ;\ (\Omega;\ \mathfrak{c}') \vdash a[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \tau''[\tau/\mathfrak{c}!\mathtt{rep}]$$

Thus, there is the derivation as follows:

Abstract
$$\frac{\begin{array}{c}[A]\ ;\ \Omega \vdash e[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \mathtt{sig}\ (\langle \mathtt{rep}{=}\tau'''; \Phi''_d\rangle)\ (\mathtt{rep} = \tau'''; \Phi''_d)\ \mathtt{end} \\ [A] + \ \mathfrak{c}':\langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep}; \Phi''_d\rangle\ ;\ (\Omega;\ \mathfrak{c}') \vdash a[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \tau''[\tau/\mathfrak{c}!\mathtt{rep}]\end{array}}{[A]\ ;\ \Omega \vdash \mathtt{collection}\ \mathfrak{c}' = e[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}]\ \ \mathtt{in}\ \ a[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \tau''[\tau/\mathfrak{c}!\mathtt{rep}]}$$

Then:

$$[A]\ ;\ \Omega \vdash (\mathtt{collection}\ \mathfrak{c}' = e\ \ \mathtt{in}\ \ a)[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \tau''[\tau/\mathfrak{c}!\mathtt{rep}]$$

**Case** $\breve{a}_2 \triangleq \lambda(\mathfrak{c}' : \mathsf{m}_k{:}i_k).\ e$ :

Let a derivation for $A_{\mathfrak{c}}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \breve{a}_2 : \breve{\tau}_2$:

Species Fun
$$\frac{A_{\mathfrak{c}} + \mathfrak{c}' : \langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle\ ;\ (\Omega;\ \mathfrak{c};\ \mathfrak{c}') \vdash e : \gamma\ _{(1)} \qquad \text{where}\ \ |i_k|_{(\Omega;\ \mathfrak{c})} = \omega_k}{A_{\mathfrak{c}}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \lambda(\mathfrak{c}' : \mathsf{m}_k{:}i_k).\ e : \langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e\rangle{\rightarrow}\gamma[\mathfrak{c}'!\mathtt{rep} \leftarrow \tau']}$$

**Remark**: by definition on $\Omega$, the collection name $\mathfrak{c}$ is different to $\mathfrak{c}'$.

In order to apply the induction hypothesis on the expression $e$ of the premiss (**1**), the hypothesis $A^*\ ;\ (\Omega;\ \mathfrak{c}) \vdash \langle \mathtt{rep}{=}t; v_w\rangle : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle$ must be extended so that $A^*\ ;\ (\Omega;\ \mathfrak{c};\ \mathfrak{c}') \vdash \langle \mathtt{rep}{=}t; v_w\rangle : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle$ ($\Omega$ has been extended with $\mathfrak{c}'$). This extension is valid because $A$ is well formed in relation to $(\Omega;\ \mathfrak{c})$. Thus $A$ doesn't possess occurrence of $\mathfrak{c}!\mathtt{rep}$. Thus $A$ is well formed in relation to $(\Omega;\ \mathfrak{c};\ \mathfrak{c}')$ also. Then $A^*\ ;\ (\Omega;\ \mathfrak{c};\ \mathfrak{c}') \vdash \langle \mathtt{rep}{=}t; v_w\rangle : \langle \mathtt{rep}{=}\tau; \Phi_c\rangle$ is extended so

that $A^* + \ \mathfrak{c}' : \langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k \rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash \langle \mathtt{rep}{=}t; v_w \rangle : \langle \mathtt{rep}{=}\tau; \Phi_c \rangle$. This extension is valid by the proposition 3. Indeed $A^* + \ \mathfrak{c}' : \langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k \rangle$ is well formed in relation to $(\Omega;\ \mathfrak{c};\ \mathfrak{c}')$. Thus by induction hypothesis applied on $e$, then:

$$[A]+ \ \mathfrak{c}' : \langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k[\tau/\mathfrak{c}!\mathtt{rep}] \rangle\ ;\ (\Omega;\ \mathfrak{c}') \vdash e[\langle \mathtt{rep}{=}t; v_w \rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \gamma[\tau/\mathfrak{c}!\mathtt{rep}]$$

By hypothesis there is $|t|_\Omega = \tau$. Thus by the lemma 3 applied on $|i_k|_{(\Omega;\ \mathfrak{c})} = \omega_k$, there is:

$$|i_k[t/\mathfrak{c}!\mathtt{rep}]|_\Omega = \omega_k[\tau/\mathfrak{c}!\mathtt{rep}]$$

Then the derivation is obtained as follows:

$$\frac{\textsc{Species Fun}\quad [A] + \ \mathfrak{c}' : \langle \mathtt{rep}{=}\mathfrak{c}'!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k[\tau/\mathfrak{c}!\mathtt{rep}] \rangle\ ;\ (\Omega;\ \mathfrak{c}') \vdash e[\langle \mathtt{rep}{=}t; v_w \rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \gamma[\tau/\mathfrak{c}!\mathtt{rep}] \quad \text{where } |i_k[t/\mathfrak{c}!\mathtt{rep}]|_\Omega = \omega_k[\tau/\mathfrak{c}!\mathtt{rep}]}{\begin{array}{l}[A]\ ;\ \Omega \vdash \lambda(\mathfrak{c}' : \mathsf{m}_k{:}i_k[t/\mathfrak{c}!\mathtt{rep}]).\ e[\langle \mathtt{rep}{=}t; v_w \rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \\ \qquad \langle \mathtt{rep}{=}\tau'[\tau/\mathfrak{c}!\mathtt{rep}], \mathsf{m}_k{:}\omega_k[\tau/\mathfrak{c}!\mathtt{rep}];\ \Phi_e[\tau/\mathfrak{c}!\mathtt{rep}] \rangle \rightarrow (\gamma[\tau/\mathfrak{c}!\mathtt{rep}])[\mathfrak{c}'!\mathtt{rep} \leftarrow \tau'[\tau/\mathfrak{c}!\mathtt{rep}]]\end{array}}$$

By hypothesis $\tau$ is well formed in relation to $\Omega$. Thus it doesn't contain occurrence of $\mathfrak{c}'!\mathtt{rep}$. Then:

$$(\gamma[\tau/\mathfrak{c}!\mathtt{rep}])[\mathfrak{c}'!\mathtt{rep} \leftarrow \tau'[\tau/\mathfrak{c}!\mathtt{rep}]] = (\gamma[\mathfrak{c}'!\mathtt{rep} \leftarrow \tau'])[\tau/\mathfrak{c}!\mathtt{rep}]$$

Thus:

$$\begin{array}{l}\langle \mathtt{rep}{=}\tau'[\tau/\mathfrak{c}!\mathtt{rep}], \mathsf{m}_k{:}\omega_k[\tau/\mathfrak{c}!\mathtt{rep}];\ \Phi_e[\tau/\mathfrak{c}!\mathtt{rep}] \rangle \rightarrow (\gamma[\tau/\mathfrak{c}!\mathtt{rep}])[\mathfrak{c}'!\mathtt{rep} \leftarrow \tau'[\tau/\mathfrak{c}!\mathtt{rep}]] = \\ (\langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e \rangle)[\tau/\mathfrak{c}!\mathtt{rep}] \rightarrow (\gamma[\mathfrak{c}'!\mathtt{rep} \leftarrow \tau'])[\tau/\mathfrak{c}!\mathtt{rep}] = \\ (\langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e \rangle \rightarrow \gamma[\mathfrak{c}'!\mathtt{rep} \leftarrow \tau'])[\tau/\mathfrak{c}!\mathtt{rep}]\end{array}$$

By the conclusion of the derivation that is previously obtained, there is:

$$\begin{array}{l}[A]\ ;\ \Omega \vdash (\lambda(\mathfrak{c}' : \mathsf{m}_k{:}i_k).\ e)[\langle \mathtt{rep}{=}t; v_w \rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \\ \qquad (\langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e \rangle \rightarrow \gamma[\mathfrak{c}'!\mathtt{rep} \leftarrow \tau'])[\tau/\mathfrak{c}!\mathtt{rep}]\end{array}$$

**Case** $\breve{a}_2 \triangleq \mathtt{rep}{=}t'$ :

Let a derivation for $A_\mathfrak{c}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \breve{a}_2 : \breve{\tau}_2$:

$$\frac{\textsc{Carrier type}\quad A_\mathfrak{c}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \mathtt{self} : \langle \mathtt{rep}{=}\tau';\ \Phi'_d \rangle\ _{(\mathbf{1})} \qquad \text{where } |t'|_{(\Omega;)} = \tau'}{A_\mathfrak{c}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \mathtt{rep} = t' : (\mathtt{rep}{=}\tau')}$$

By induction hypothesis applied on the expression $\mathtt{self}$ of the premiss ($\mathbf{1}$), there is:

$$[A]\ ;\ \Omega \vdash \mathtt{self}[\langle \mathtt{rep}{=}t; v_w \rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : \langle \mathtt{rep}{=}\tau'[\tau/\mathfrak{c}!\mathtt{rep}];\ \Phi'_d[\tau/\mathfrak{c}!\mathtt{rep}] \rangle$$

There is $\mathtt{self}[\langle \mathtt{rep}{=}t; v_w \rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] = \mathtt{self}$, thus more precisely there is:

$$[A]\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}{=}\tau'[\tau/\mathfrak{c}!\mathtt{rep}];\ \Phi'_d[\tau/\mathfrak{c}!\mathtt{rep}] \rangle$$

By hypothesis there is $|t|_{\Omega} = \tau$. Thus by the lemma 3 applied on $|t'|_{(\Omega;\mathfrak{c})} = \tau'$, there is $|t'[t/\mathfrak{c}!\mathtt{rep}]|_{\Omega} = \tau'[\tau/\mathfrak{c}!\mathtt{rep}]$. Then:

$$\frac{\begin{array}{c}\textsc{Carrier type}\\ [A]\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}{=}\tau'[\tau/\mathfrak{c}!\mathtt{rep}];\ \Phi'_d[\tau/\mathfrak{c}!\mathtt{rep}]\rangle\\ \text{where } |t'[t/\mathfrak{c}!\mathtt{rep}]|_{\Omega} = \tau'[\tau/\mathfrak{c}!\mathtt{rep}]\end{array}}{[A]\ ;\ \Omega \vdash \mathtt{rep} = t'[t/\mathfrak{c}!\mathtt{rep}] : (\mathtt{rep}{=}\tau'[\tau/\mathfrak{c}!\mathtt{rep}])}$$

Thus:

$$[A]\ ;\ \Omega \vdash (\mathtt{rep} = t')[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (\mathtt{rep}{=}\tau')[\tau/\mathfrak{c}!\mathtt{rep}]$$

**Case** $\breve{a}_2 \triangleq \mathsf{m}{:}i = a$ :

Let a derivation for $A_{\mathfrak{c}}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \breve{a}_2 : \breve{\tau}_2$:

$$\frac{\begin{array}{c}\textsc{Method}\\ A_{\mathfrak{c}}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \mathtt{self} : \langle \mathtt{rep}{=}\tau';\ \mathsf{m}{:}\omega';\ \Phi'_d\rangle\ {}_{(\mathbf{1})}\\ A_{\mathfrak{c}}\ ;\ (\Omega;\ \mathfrak{c}) \vdash a : \omega'[\mathtt{rep} \leftarrow \tau']\ {}_{(\mathbf{2})} \qquad \text{where } |i|_{(\Omega;\ \mathfrak{c})} = \omega'\end{array}}{A_{\mathfrak{c}}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \mathsf{m}{:}i = a : (\mathsf{m} : \omega')}$$

By hypothesis induction applied on the expressions $\mathtt{self}$ and $a$, respectively of premises (**1**) and (**2**), then:

$$[A]\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}{=}\tau'[\tau/\mathfrak{c}!\mathtt{rep}];\ \mathsf{m}{:}\omega'[\tau/\mathfrak{c}!\mathtt{rep}];\ \Phi'_d[\tau/\mathfrak{c}!\mathtt{rep}]\rangle$$

and

$$[A]\ ;\ \Omega \vdash a[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (\omega'[\mathtt{rep} \leftarrow \tau'])[\tau/\mathfrak{c}!\mathtt{rep}]$$

that is:

$$[A]\ ;\ \Omega \vdash a[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (\omega'[\tau/\mathfrak{c}!\mathtt{rep}])[\mathtt{rep} \leftarrow \tau'[\tau/\mathfrak{c}!\mathtt{rep}]]$$

By hypothesis there is $|t|_{\Omega} = \tau$. By the lemma 3 applied on $|i|_{(\Omega;\ \mathfrak{c})} = \omega'$, then $|i[t/\mathfrak{c}!\mathtt{rep}]|_{\Omega} = \omega'[\tau/\mathfrak{c}!\mathtt{rep}]$.
Thus the obtained derivation as follows;

$$\frac{\begin{array}{c}\textsc{Method}\\ [A]\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}{=}\tau'[\tau/\mathfrak{c}!\mathtt{rep}];\ \mathsf{m}{:}\omega'[\tau/\mathfrak{c}!\mathtt{rep}];\ \Phi'_d[\tau/\mathfrak{c}!\mathtt{rep}]\rangle\\ [A]\ ;\ \Omega \vdash a[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (\omega'[\tau/\mathfrak{c}!\mathtt{rep}])[\mathtt{rep} \leftarrow \tau'[\tau/\mathfrak{c}!\mathtt{rep}]]\\ \text{where } |i[t/\mathfrak{c}!\mathtt{rep}]|_{\Omega} = \omega'[\tau/\mathfrak{c}!\mathtt{rep}]\end{array}}{A_{\mathfrak{c}}\ ;\ (\Omega;\ \mathfrak{c}) \vdash \mathsf{m}{:}i[t/\mathfrak{c}!\mathtt{rep}] = a[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (\mathsf{m} : \omega'[\tau/\mathfrak{c}!\mathtt{rep}])}$$

Thus:

$$A_{\mathfrak{c}}\ ;\ (\Omega;\ \mathfrak{c}) \vdash (\mathsf{m}{:}i = a)[\langle \mathtt{rep}{=}t; v_w\rangle/\mathfrak{c}][t/\mathfrak{c}!\mathtt{rep}] : (\mathsf{m} : \omega')[\tau/\mathfrak{c}!\mathtt{rep}]$$

**Lemma 5.** *Let $\Omega$ a collection name environment, $A$ a typing environment well formed in relation to $\Omega$, $w_1$ and $w_2$ two lists of fields, $\Phi_1$ and $\Phi_2$ two compatible lists and well formed in relation to $\Omega$.*
*If $A\ ;\ \Omega \vdash w_1 : \Phi_1$ et $A\ ;\ \Omega \vdash w_2 : \Phi_2$ then $A\ ;\ \Omega \vdash w_1 \ @\ w_2 : \Phi_1 \oplus \Phi_2$.*

*Proof.* The proof is done by induction on $w_1$.

For the lemma described after, it is given:

$$\tilde{a} \triangleq a \mid col \mid e$$
$$\tilde{\tau} \triangleq \tau \mid \langle \Phi \rangle \mid \gamma$$

These notations must be used in a consistence way. Thus $(\tilde{a}, \tilde{\tau})$ must be understood like $(a, \tau)$, $(col, \langle \Phi \rangle)$ , $(e, \gamma)$ but not like $(a, \gamma)$ or $(e, \tau)$ for example.

**Lemma 6.** *Let $\Omega$ a collection name environment, $A$ a starry environment (that is $A$ is $A'^*$) well formed in relation to $\Omega$, a list of fields $v_w$, a collection value $\langle v_w \rangle$, a type $\langle \Phi_c \rangle$ well formed in relation to $\Omega$.*
*If $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash v_w : \Phi_c$, $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash \tilde{a} : \tilde{\tau}$, bound variables of $\tilde{a}$ are not free in $\langle v_w \rangle$, then $A\ ;\ \Omega \vdash \tilde{a}[\langle v_w \rangle / \texttt{self}] : \tilde{\tau}$.*

*Proof.* The proof is by induction on $\tilde{a}$ and by derivation of $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash \tilde{a} : \tilde{\tau}$.
Some cases are shown for examples. We note $A_s$ for $A + \ \texttt{self} : \langle \Phi_c \rangle$.

**Case** $\tilde{a} \triangleq \texttt{collection c} = e\ \texttt{in}\ a\ :$

Let a possible derivation for $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash \tilde{a} : \tilde{\tau}$:

$$\frac{\begin{array}{l}\textsc{Abstract} \\ A_s\ ;\ \Omega \vdash e : \texttt{sig}\ (\langle \texttt{rep}{=}\tau' ; \Phi_d \rangle)\ (\texttt{rep} = \tau' ; \Phi_d)\ \texttt{end}\ {}_{(1)} \\ \qquad A_s + \texttt{c} : \langle \texttt{rep}{=}\texttt{c!rep} ; \Phi_d \rangle\ ;\ (\Omega ;\ \texttt{c}) \vdash a : \tau\ {}_{(2)}\end{array}}{A_s\ ;\ \Omega \vdash \texttt{collection c} = e\ \texttt{in}\ a : \tau}$$

By induction hypothesis applied on the expression $e$ of the premiss (**1**), there is:

$$A\ ;\ \Omega \vdash e[\langle v_w \rangle / \texttt{self}] : \texttt{sig}\ (\langle \texttt{rep}{=}\tau' ; \Phi_d \rangle)\ (\texttt{rep} = \tau' ; \Phi_d)\ \texttt{end}$$

In order to apply the induction hypothesis on the expression $a$ of the premiss (**2**), the hypothesis $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash v_w : \Phi_c$ must be extended so that $A + \ \texttt{self} : \langle \Phi_c \rangle\ ;\ (\Omega ;\ \texttt{c}) \vdash v_w : \Phi_c$ ($\Omega$ is extended with $\texttt{c}$). This extension is valid because $\texttt{c}$ is fresh in relation to $\Omega$. Then, $A + \ \texttt{self} : \langle \Phi_c \rangle\ ;\ (\Omega ;\ \texttt{c}) \vdash v_w : \Phi_c$ so that $A + \ \texttt{self} : \langle \Phi_c \rangle + \texttt{c} : \langle \texttt{rep}{=}\texttt{c!rep} ; \Phi_d \rangle\ ;\ (\Omega ;\ \texttt{c}) \vdash v_w : \Phi_c$. This extension is valid by the proposition 1. Indeed, the environment $A + \ \texttt{self} : \langle \Phi_c \rangle + \texttt{c} : \langle \texttt{rep}{=}\texttt{c!rep} ; \Phi_d \rangle$ is well formed in relation to $\Omega$.
Thus by induction hypothesis applied on the expression $a$:

$$A + \ \texttt{c} : \langle \texttt{rep}{=}\texttt{c!rep} ; \Phi_d \rangle\ ;\ (\Omega ;\ \texttt{c}) \vdash a[\langle v_w \rangle / \texttt{self}] : \tau$$

Then the following derivation is obtained:

$$\frac{\begin{array}{l}\textsc{Abstract}\\ A\ ;\ \Omega \vdash e[\langle v_w\rangle/\mathtt{self}] : \mathtt{sig}\ (\langle \mathtt{rep}{=}\tau';\Phi_d\rangle)\ (\mathtt{rep} = \tau';\Phi_d)\ \mathtt{end} \\ \qquad A + \ \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\Phi_d\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash a[\langle v_w\rangle/\mathtt{self}] : \tau\end{array}}{A\ ;\ \Omega \vdash \mathtt{collection}\ \mathfrak{c} = e[\langle v_w\rangle/\mathtt{self}]\ \ \mathtt{in}\ \ a[\langle v_w\rangle/\mathtt{self}] : \tau}$$

Thus:

$$A\ ;\ \Omega \vdash (\mathtt{collection}\ \mathfrak{c} = e\ \ \mathtt{in}\ \ a)[\langle v_w\rangle/\mathtt{self}] : \tau$$

**Case** $\tilde{a} \triangleq \mathtt{struct}\ \ w\ \ \mathtt{end}$ :

In a structure, `self` is bound variable. Thus:

$$(\mathtt{struct}\ \ w\ \ \mathtt{end})[\langle v_w\rangle/\mathtt{self}] = \mathtt{struct}\ \ w\ \ \mathtt{end}$$

Then by hypothesis:

$$A_s\ ;\ \Omega \vdash (\mathtt{struct}\ \ w\ \ \mathtt{end})[\langle v_w\rangle/\mathtt{self}] : \tilde{\tau}$$

Then by the proposition 3 applied on the above judgement:

$$A\ ;\ \Omega \vdash (\mathtt{struct}\ \ w\ \ \mathtt{end})[\langle v_w\rangle/\mathtt{self}] : \tilde{\tau}$$

**Case** $\tilde{a} \triangleq \lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e$ :

Let a possible derivation for $A + \mathtt{self} : \langle \Phi_c\rangle\ ;\ \Omega \vdash \tilde{a} : \tilde{\tau}$:

$$\frac{\begin{array}{l}\textsc{Species Fun}\\ A_s + \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash e : \gamma\ _{(\mathbf{1})} \qquad \mathrm{o}\ \ |i_k|_\Omega = \omega_k\end{array}}{A_s\ ;\ \Omega \vdash \lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e : \langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e\rangle{\rightarrow}\gamma[\mathfrak{c}!\mathtt{rep} \leftarrow \tau']}$$

In order to apply the induction hypothesis on the expression $e$ on the premiss ($\mathbf{1}$), the hypothesis $A + \mathtt{self} : \langle \Phi_c\rangle\ ;\ \Omega \vdash v_w : \Phi_c$ is extended so that $A + \ \mathtt{self} : \langle \Phi_c\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash v_w : \Phi_c$ ($\Omega$ is extended $\mathfrak{c}$). This extension is valid because $\mathfrak{c}$ is fresh in relation to $\Omega$. Then, $A + \ \mathtt{self} : \langle \Phi_c\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash v_w : \Phi_c$ is extended so that $A + \ \mathtt{self} : \langle \Phi_c\rangle + \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash v_w : \Phi_c$. This extension is valid by the proposition 1. Indeed the environment $A + \ \mathtt{self} : \langle \Phi_c\rangle + \ \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle$ is well formed in relation to $\Omega$.
Then by induction hypothesis applied on the expression $a$:

$$\frac{\begin{array}{l}\textsc{Species Fun}\\ A + \ \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k{:}\omega_k\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash e[\langle v_w\rangle/\mathtt{self}] : \gamma\ _{(\mathbf{1})} \qquad \mathrm{o}\ \ |i_k|_\Omega = \omega_k\end{array}}{A\ ;\ \Omega \vdash \lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e[\langle v_w\rangle/\mathtt{self}] : \langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e\rangle{\rightarrow}\gamma[\mathfrak{c}!\mathtt{rep} \leftarrow \tau']}$$

Thus:

$$A\ ;\ \Omega \vdash (\lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e)[\langle v_w\rangle/\mathtt{self}] : \langle \mathtt{rep}{=}\tau', \mathsf{m}_k{:}\omega_k;\ \Phi_e\rangle{\rightarrow}\gamma[\mathfrak{c}!\mathtt{rep} \leftarrow \tau']$$

**Lemma 7 (Prservation du typage).** *If* $\breve{a}_1 \rightarrow_\epsilon \breve{a}_2$ *then* $\breve{a}_1 \subset \breve{a}_2$.

*Proof.* The proof is done case by case thanks to previous lemmas.
We suppose $A\ ;\ \Omega \vdash \breve{a}_1 : \breve{\tau}_1$ with $A$ starry in relation to the context of use of the relation $\subset$.

**Case** $(\lambda x.\ a)\ v \rightarrow_\epsilon a[v/x]$ :

Let a derivation for $(\lambda x.\ a)\ v$:

$$\dfrac{\dfrac{A + x : \tau'\ ;\ \Omega \vdash a : \tau'\ \textbf{(1)}}{A\ ;\ \Omega \vdash \lambda x.\ a : \tau \rightarrow \tau'} \qquad A\ ;\ \Omega \vdash v : \tau\ \textbf{(2)}}{A\ ;\ \Omega \vdash (\lambda x.\ a)\ v : \tau'}$$

Hence by the lemma 2 applied on the premises (**1**) and (**2**):

$$A\ ;\ \Omega \vdash a[v/x] : \tau'$$

**Case** $\texttt{let}\ x = v\ \texttt{in}\ a \rightarrow_\epsilon a[v/x]$ :

Let a derivation for $\texttt{let}\ x = v\ \texttt{in}\ a$:

$$\dfrac{A\ ;\ \Omega \vdash v : \tau'\ \textbf{(1)} \qquad A + x : Gen(\tau', A)\ ;\ \Omega \vdash a : \tau\ \textbf{(2)}}{A\ ;\ \Omega \vdash \texttt{let}\ x = v\ \texttt{in}\ a : \tau}$$

Hence by the lemma 2 applied on the premises (**1**) and (**2**):

$$A\ ;\ \Omega \vdash a[v/x] : \tau$$

**Case** $\langle v_w \rangle !\mathsf{m} \rightarrow_\epsilon v_w(\mathsf{m})[\langle v_w \rangle / \texttt{self}]$ :

For this case, let $v_w = (m : i = v_w(m))\ @\ v'_w$ with $m \notin dom(v'_w)$.
Then note $A_s$ for $A^* + \texttt{self} : \langle \texttt{rep} = \tau;\ m : \iota;\ \Phi_d \rangle$

Let derivation for $\langle v_w \rangle$:

$$\dfrac{\dfrac{\dfrac{\dfrac{A_s\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \tau;\ \mathsf{m} : \omega;\ \Phi_d \rangle}{A_s\ ;\ \Omega \vdash v_w(\mathsf{m}) : \omega[\texttt{rep} \leftarrow \tau]\ \textbf{(1)}} \qquad \mathrm{o}\ |i|_\Omega = \omega}{A_s\ ;\ \Omega \vdash (\mathsf{m}{:}i = v_w(\mathsf{m})) : (\mathsf{m} : \omega)} \qquad A_s\ ;\ \Omega \vdash v'_w : (\texttt{rep} = \tau;\ \Phi_d)}{A^* + \texttt{self} : \langle \texttt{rep} = \tau;\ \mathsf{m} : \omega;\ \Phi_d \rangle\ ;\ \Omega \vdash (\mathsf{m} : i = v_w(\mathsf{m}))\ @\ v'_w : (\texttt{rep} = \tau;\ \mathsf{m} : \omega;\ \Phi_d)\ \textbf{(2)}}}{A\ ;\ \Omega \vdash \langle (\mathsf{m}{:}i = v_w(\mathsf{m}))\ @\ v'_w \rangle : \langle \texttt{rep} = \tau;\ \mathsf{m} : \omega;\ \Phi_d \rangle}}{A\ ;\ \Omega \vdash \langle (\mathsf{m}{:}i = v_w(\mathsf{m}))\ @\ v'_w \rangle !\mathsf{m} : \omega[\texttt{rep} \leftarrow \tau]}$$

Thus by the lemma 6 applied on the premises (**1**) and (**2**):

$$A\ ;\ \Omega \vdash v_w(m)[\langle(\mathsf{m}{:}i = v_w(\mathsf{m}))\ @\ v'_w\rangle/\mathtt{self}] : \omega[\mathtt{rep} \leftarrow \tau]$$

Hence:

$$A\ ;\ \Omega \vdash v_w(\mathsf{m})[\langle v_w\rangle/\mathtt{self}] : \omega[\mathtt{rep} \leftarrow \tau]$$

**Case** $\mathtt{species}\ \mathsf{S} = v_s\ \mathtt{in}\ a \rightarrow_\epsilon a[v_s/\mathsf{S}]$ :

Let a derivation for $\mathtt{species}\ \mathsf{S} = v_s\ \mathtt{in}\ a$:

$$\frac{A\ ;\ \Omega \vdash v_s : \gamma\ _{(\mathbf{1})} \qquad A + z : Gen(\gamma, A)\ ;\ \Omega \vdash a : \tau\ _{(\mathbf{2})}}{A\ ;\ \Omega \vdash \mathtt{species}\ \mathsf{S} = v_s\ \mathtt{in}\ a : \gamma}$$

Thus by the lemma 2 applied on the premises (**1**) and (**2**):

$$A\ ;\ \Omega \vdash a[v_s/z] : \gamma$$

**Case** $(\lambda(\mathfrak{c} : I).\ e)\ v_{col} \rightarrow_\epsilon e[v_{col}/\mathfrak{c}][v_{col}(\mathtt{rep})/\mathfrak{c}!\mathtt{rep}]$ :

Here, put $\mathsf{m}_k : i_k$ for $I$.

Let a derivation for $(\lambda(\mathfrak{c} : I).\ e)\ v_{col}$:

$$\frac{\dfrac{A + \ \mathfrak{c} : \langle \mathtt{rep}{=}\mathfrak{c}!\mathtt{rep};\ \mathsf{m}_k : \omega_k\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash e : \gamma\ _{(\mathbf{1})} \qquad \text{where } |i_k|_\Omega = \omega_k}{\begin{array}{c} A\ ;\ \Omega \vdash \lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e : (\mathtt{rep}{=}\tau;\ \mathsf{m}_k : \omega_k) \rightarrow \gamma[\mathfrak{c}!\mathtt{rep} \leftarrow \tau] \\ A\ ;\ \Omega \vdash v_{col} : \langle \mathtt{rep}{=}\tau;\ \mathsf{m}_k : \omega_k\rangle\ _{(\mathbf{2})} \end{array}}}{A\ ;\ \Omega \vdash (\lambda(\mathfrak{c} : \mathsf{m}_k{:}i_k).\ e)\ v_{col} : \gamma[\mathfrak{c}!\mathtt{rep} \leftarrow \tau]}$$

From the premiss (**2**), the environment $A$ and the type $\langle \mathtt{rep}{=}\tau;\ \mathsf{m}_k : \omega_k\rangle$ are well typed in relation to $\Omega$. As $\mathfrak{c}$ is fresh in relation to $\Omega$, then $A$ and $\langle \mathtt{rep}{=}\tau;\ \mathsf{m}_k : \omega_k\rangle$ are well formed in relation to $(\Omega;\ \mathfrak{c})$. Then $A\ ;\ (\Omega;\ \mathfrak{c}) \vdash v_{col} : \langle \mathtt{rep}{=}\tau;\ \mathsf{m}_k : \omega_k\rangle$ is valid. Hence by the lemma 4 applied on this judgement and the premiss (**1**):

$$A\ ;\ \Omega \vdash e[v_{col}/\mathfrak{c}][v_{col}(\mathtt{rep})/\mathfrak{c}!\mathtt{rep}] : \gamma[\mathfrak{c}!\mathtt{rep} \leftarrow \tau]$$

**Case** $\mathtt{collection}\ \mathfrak{c} = (\mathtt{struct}\ \ v_w\ \ \mathtt{end})\ \mathtt{in}\ a \rightarrow_\epsilon a[\langle v_w\rangle/\mathfrak{c}][v_w(\mathtt{rep})/\mathfrak{c}!\mathtt{rep}]$ :

Let a derivation for $\mathtt{collection}\ \mathfrak{c} = (\mathtt{struct}\ \ v_w\ \ \mathtt{end})\ \mathtt{in}\ a$:

$$\frac{\dfrac{A^* + \mathtt{self} : \langle \mathtt{rep} = \tau; \Phi_d\rangle\ ;\ \Omega \vdash v_w : (\mathtt{rep} = \tau; \Phi_d)\ _{(\mathbf{1})}}{\begin{array}{l} A\ ;\ \Omega \vdash \mathtt{struct}\ \ v_w\ \ \mathtt{end} : \\ \qquad \mathtt{sig}\ (\langle \mathtt{rep} = \tau; \Phi_d\rangle)\ (\mathtt{rep} = \tau; \Phi_d)\ \mathtt{end} \end{array}} \qquad A + \ \mathfrak{c} : \langle \mathtt{rep} = \mathfrak{c}!\mathtt{rep}; \Phi_d\rangle\ ;\ (\Omega;\ \mathfrak{c}) \vdash a : \tau'\ _{(\mathbf{2})}}{A\ ;\ \Omega \vdash \mathtt{collection}\ \mathfrak{c} = \mathtt{struct}\ \ v_w\ \ \mathtt{end}\ \mathtt{in}\ a : \tau'}$$

From the premiss (**1**), a derivation can be obtained as follows:

$$\frac{A^* + \texttt{self} : \langle \texttt{rep} = \tau; \Phi_d \rangle \ ; \ \Omega \vdash v_w : (\texttt{rep} = \tau; \Phi_d)}{A \ ; \ \Omega \vdash \langle v_w \rangle : \langle \texttt{rep} = \tau; \Phi_d \rangle}$$

From the conclusion of the above derivation, the environment $A$ and the type $\langle \texttt{rep} = \tau; \Phi_d \rangle$ are well formed in relation to $\Omega$. Knowing $\mathfrak{c}$ is fresh in relation to $\Omega$, thus $A$ and $\langle \texttt{rep} = \tau; \Phi_d \rangle$ are well formed in relation to $(\Omega; \ \mathfrak{c})$ also. Hence the judgement $A \ ; \ \Omega \vdash \langle v_w \rangle : \langle \texttt{rep} = \tau; \Phi_d \rangle$ is valid. Then by application of lemma 4 on this judgement and the premiss (**2**):

$$A \ ; \ \Omega \vdash a[\langle v_w \rangle / \mathfrak{c}][v_w(\texttt{rep})/\mathfrak{c}!\texttt{rep}] : \tau'[\tau/\mathfrak{c}!\texttt{rep}]$$

Lastly, by the judgement $A \ ; \ \Omega \vdash \texttt{collection } \mathfrak{c} = \texttt{struct } v_w \texttt{ end in } a : \tau'$, conclusion of the initial derivation, the type $\tau'$ is well formed in relation to $\Omega$. However $\mathfrak{c}$ is fresh in relation to $\Omega$. Thus $\tau'$ doesn't contain occurrence of type $\mathfrak{c}!\texttt{rep}$. Then $\tau'[\tau/\mathfrak{c}!\texttt{rep}] = \tau'$. Hence:

$$A \ ; \ \Omega \vdash a[\langle v_w \rangle / \mathfrak{c}][v_w(\texttt{rep})/\mathfrak{c}!\texttt{rep}] : \tau'$$

**Case** $\mathsf{m}{:}i{=}a; \ v_w \rightarrow_\epsilon v_w$ si $\mathsf{m} \in dom(v_w)$ :

Let a derivation for $\mathsf{m}{:}i{=}a; \ v_w$:

$$\frac{A \ ; \ \Omega \vdash \mathsf{m}{:}i = a : (\mathsf{m} : \omega) \qquad A \ ; \ \Omega \vdash v_w : \Phi \ _{(\mathbf{1})}}{A \ ; \ \Omega \vdash (\mathsf{m}{:}i = a; \ v_w) : (\mathsf{m} : \omega) \oplus \Phi}$$

$\mathsf{m} \in dom(\Phi)$ since $\mathsf{m} \in dom(v_w)$. Then, $(\mathsf{m} : \omega)$ and $\Phi$ are compatible. Hence by the premiss (**1**):

$$A \ ; \ \Omega \vdash v_w : (\mathsf{m} : \omega) \oplus \Phi$$

**Case** $\texttt{rep}{=}t; \ v_w \rightarrow_\epsilon v_w$ si $\texttt{rep} \in dom(v_w)$ :

Let a derivation for $\texttt{rep}{=}t$:

$$\frac{A \ ; \ \Omega \vdash \texttt{rep}{=}t : (\texttt{rep} : \tau) \qquad A \ ; \ \Omega \vdash v_w : \Phi \ _{(\mathbf{1})}}{A \ ; \ \Omega \vdash \texttt{rep}{=}t; \ v_w : (\texttt{rep} : \tau) \oplus \Phi}$$

Since $\texttt{rep} \in dom(v_w)$, then $\texttt{rep} \in dom(\Phi)$. Indeed $\texttt{rep} \in dom(v_w)$. Thus $(\texttt{rep} : \tau)$ and $\Phi$ are compatible. Thus $\Phi = (\texttt{rep} : \tau) \oplus \Phi$. Hence by the premiss (**1**):

$$A \ ; \ \Omega \vdash v_w : (\texttt{rep} : \tau) \oplus \Phi$$

**Case** $\texttt{inherit } (\texttt{struct } v_w \texttt{ end}); \ w \rightarrow_\epsilon v_w \ @ \ w$ :

Let a derivation for $\texttt{inherit}\ (\texttt{struct}\ v_w\ \texttt{end});\ w$:

$$\dfrac{\dfrac{A\ ;\ \Omega \vdash \texttt{self} : \xi\ _{(\mathbf{2})} \qquad \dfrac{A^* + \texttt{self} : \xi\ ;\ \Omega \vdash v_w : \Phi_1\ _{(\mathbf{1})}}{A^*\ ;\ \Omega \vdash \texttt{struct}\ v_w\ \texttt{end} : \texttt{sig}\ (\xi)\ \Phi_1\ \texttt{end}}}{A\ ;\ \Omega \vdash \texttt{inherit}\ (\texttt{struct}\ v_w\ \texttt{end}) : \Phi_1} \qquad A\ ;\ \Omega \vdash w : \Phi_2\ _{(\mathbf{3})}}{A\ ;\ \Omega \vdash \texttt{inherit}\ (\texttt{struct}\ v_w\ \texttt{end});\ w : \Phi_1 \oplus \Phi_2}$$

By the premiss (**2**), $A$ is equivalent to $A^* + \texttt{self} : \xi$. Thus the premiss can be rewritten as $A\ ;\ \Omega \vdash v_w : \Phi_1$. As $\Phi_1$ and $\Phi_2$ are compatible (that is $\Phi_1 \oplus \Phi_2$), the lemma 5 can be applied on the premises (**1**) and (**3**). Hence:

$$A\ ;\ \Omega \vdash v_w\ @\ w : \Phi_1 \oplus \Phi_2$$

**Lemma 8.**

1. *Let $v$ a value such that $\emptyset\ ;\ \Omega \vdash v : \tau$.*
   - *If $\tau$ is a functional type, then $v$ is a function.*
   - *If $\tau$ is product type, then $v$ is a pair.*
2. *Let $v_s$ a species value such that $\emptyset\ ;\ \Omega \vdash v_s : \gamma$*
   - *If $\gamma$ is a type of species structure, then $v_s$ is species structure.*
   - *If $\gamma$ is functional type, then $v_s$ is a parameterized species.*

*Proof.* 1. If $\tau$ is functional type, then it have the shape $\tau_1 \to \tau_2$. In the empty environment, only the rules Cst and Fun-ML can be applied on the value $v$. However only the rule Fun-ML returns a functional type. Hence $v$ is a function.

2. In the empty environment, only the rules Species Body and Species Fun can be applied on the value $v_s$.
   In the case where $\gamma$ is a type of species structure of shape $\texttt{sig}\ (\xi)\ \Phi\ \texttt{end}$, only the rule Species Body returns a such type. In this case, $v_s$ is a species structure.
   Otherwise in the case where $\gamma$ is a functional type of shape $\xi \to \gamma'$, only the rule Species Fun returns a such type. In this case, $v_s$ a parameterized species.

**Theorem 1 (Prservation du typage).** *Reduction preserves typing (i.e. for any $A$, if $A^*\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$ and $\breve{a} \to \breve{a}'$, then $A^*\ ;\ \Omega \vdash \breve{a}' : \breve{\tau}$).*

*Proof.* The proof is done thanks to the different properties established previously.

**Theorem 2 (Formes normales bien types sont les valeurs).** *Well-typed irreducible normal forms are values (i.e. if $\emptyset\ ;\ \emptyset \vdash \breve{a} : \breve{\tau}$ and $\breve{a}$ cannot be reduced, then $\breve{a}$ is a value).*

*Proof.* The proof is by structural induction simultaneously on the different forms of $\breve{a}$. Suppose $\emptyset\ ;\ \emptyset \vdash a : \tau$ (respectively, $\emptyset\ ;\ \emptyset \vdash e : \gamma$ and $A\ ;\ \emptyset \vdash w : \Phi$ with $A$ containing only the variable `self` needed to get the type of $w$).

**Case** $\breve{a} \triangleq \kappa$ :

By definition $\kappa$ is a value.

**Case** $\breve{a} \triangleq x$ :

The term variable $x$ cannot be typed in an empty environment.

**Case** $\breve{a} \triangleq \lambda x.\ a$ :

By definition $\lambda x.\ a$ is a value.

**Case** $\breve{a} \triangleq a_1\ a_2$ :

Let a typing derivation for $a_1\ a_2$:

$$\frac{\emptyset\ ;\ \emptyset \vdash a_1 : \tau_1 \rightarrow \tau_2 \qquad \emptyset\ ;\ \emptyset \vdash a_2 : \tau_1}{\emptyset\ ;\ \emptyset \vdash a_1\ a_2 : \tau_2}$$

By induction hypothesis applied on $a_1$, this expression is a value. From the above derivation, its type is functional. By the lemma 8, $a_1$ is a function. Then $a_1\ a_2$ can be reduced. It is contradictory.

**Case** $\breve{a} \triangleq (a_1, a_2)$ :

By induction hypothesis applied on $a_1$ and $a_2$, these expressions are values. Thus by definition $(a_1, a_2)$ is a value.

**Case** $\breve{a} \triangleq$ `let` $x = a_1$ `in` $a_2$ :

By induction hypothesis applied on $a_1$, this expression is a value. Thus the expression `let` $x = a_1$ `in` $a_2$ can be reduced. It is contradictory.

**Case** $\breve{a} \triangleq col!\mathsf{m}$ :

By induction hypothesis applied on $col$, this expression is a value. Thus the expression $col!\mathsf{m}$ can be reduced. That is contradictory.

**Case** $\breve{a} \triangleq$ `species` $\mathsf{S} = e$ `in` $a$ :

By induction hypothesis applied on $e$, this expression is a value. Thus the expression `species` $\mathsf{S} = e$ `in` $a$ can be reduced. That is contradictory.

**Case** $\breve{a} \triangleq$ `collection` $\mathfrak{c} = e$ `in` $a$ :

Let a typing derivation for $\texttt{collection}\ \mathfrak{c} = e\ \texttt{in}\ a$:

$$\frac{\begin{array}{c}\emptyset\ ;\ \emptyset \vdash e : \texttt{sig}\ (\langle \texttt{rep} = \tau;\ \Phi_d \rangle)\ (\texttt{rep} = \tau;\ \Phi_d)\ \texttt{end} \\ \mathfrak{c} : \langle \texttt{rep} = \mathfrak{c}!\texttt{rep};\ \Phi_d \rangle;\ \mathfrak{c} \vdash a : \tau\end{array}}{\emptyset\ ;\ \emptyset \vdash \texttt{collection}\ \mathfrak{c} = e\ \texttt{in}\ a : \tau}$$

By induction hypothesis applied on $e$, this expression is a value. From the above derivation, its type is the one of species structure. By the lemma 8, $e$ is a structure $\texttt{struct}\ v_w\ \texttt{end}$. Thus $\texttt{collection}\ \mathfrak{c} = e\ \texttt{in}\ a$ can be reduced. It is contradictory.

**Case** $\breve{a} \triangleq \mathfrak{c}$ :

A collection name cannot be typed in an empty environment.

**Case** $\breve{a} \triangleq \texttt{self}$ :

The variable `self` cannot be typed in an empty environment.

**Case** $\breve{a} \triangleq \mathsf{S}$ :

The variable $\mathsf{S}$ cannot be typed in a empty environment.

**Case** $\breve{a} \triangleq \texttt{struct}\ w\ \texttt{end}$ :

By induction hypothesis applied on $w$, this expression is a value. Thus by definition $\texttt{struct}\ w\ \texttt{end}$ is a values.

**Case** $\breve{a} \triangleq \lambda(\mathfrak{c} : I).\ e$ :

By definition $\lambda(\mathfrak{c} : I).\ e$ is a value.

**Case** $\breve{a} \triangleq e\ col$ :

Let a typing derivation for $e\ col$:

$$\frac{\emptyset\ ;\ \emptyset \vdash e : \tau_{col} \rightarrow \gamma \qquad \emptyset\ ;\ \emptyset \vdash col : \tau_{col}}{\emptyset\ ;\ \emptyset \vdash e\ a : \gamma}$$

By induction hypothesis applied on $e$, this expression is a value. From the above derivation, its type is functional. By the lemma 8, $e$ is a parameterized species. Thus $e\ col$ can be reduced. It is contradictory.

**Case** $\breve{a} \triangleq \emptyset$ :

By definition $\emptyset$ is a value.

**Case** $\breve{a} \triangleq d;\ w$ :

By induction hypothesis applied on $d$ and $w$, these expressions are values.
For the case where the expression $d$ is either a method or field `rep`, not overloaded by $w$, then by definition $d;\ w$ is a value. On the other hand $d;\ w$ can be reduced in the contrary case. Then there is contradiction.

**Case** $\breve{a} \triangleq \texttt{rep}{=}t$ :

By definition `rep`$=t$ is a value.

**Case** $\breve{a} \triangleq \mathsf{m}{:}i = a$ :

By definition $\mathsf{m}{:}i = a$ is a value.

**Case** $\breve{a} \triangleq \texttt{inherit}\ e$ :

Let a typing derivation for `inherit` $e$:

$$\frac{A\ ;\ \Omega \vdash \texttt{self} : \xi \qquad A^*\ ;\ \Omega \vdash e : \texttt{sig}\ (\xi)\ \Phi\ \texttt{end}}{A\ ;\ \Omega \vdash \texttt{inherit}\ e : \Phi}$$

By induction hypothesis applied on $e$, this expression is a value. From the above derivation, its type is the one of specie structure. By the lemma 8, $e$ is a species structure `struct` $v_w$ `end`. Then `inherit` $e$ can be reduced. It is contradictory.

## 7 Relative works and conclusion

FOCAL is a powerful language allowing to develop certificated components. It provides new aspects for object oriented programming such as collections and species. $\mathcal{FML}$ has been designed to extend ML with notions of collections and species. $\mathcal{FML}$ also provides abstraction of entities. This abstraction is given by the construction `collection` $\mathfrak{c}$ $=$ $c$ `in` $a$ and controlled by the type system. Indeed, the type system imposes that the carrier type of $\mathfrak{c}$ is different to others types and other formal variables. Moreover, the type system guarantees that the carrier type of $\mathfrak{c}$ is limited to the scope of $a$. Thus, it is no possible to confuse $\mathfrak{c}$`!rep` with other types beyond the scope of $a$. The semantics of `collection` $\mathfrak{c} = e$ `in` $a$ and constraints imposed on it, recall PACK and ABSTYPE operators related to existential types described by J.C. Mitchell and G.D. Plotkin in [10]. Indeed, their operators have similar constraints in order to guarantee abstraction of data types.
Our extension of ML is successful since $\mathcal{FML}$ has the type soundness proved. Hence, operational aspects of collections and species in a language extended with these constructions are coherent.

The design of $\mathcal{FML}$ may seem near to the one of Objective ML [8]. Indeed, a similar syntax is retrieved. Hence, collections may seem similar to objects and species similar to classes. Moreover, type system and operational semantics are also similar. The type system also uses a starry environment in order to correctly build a fix point for the self reference and objects. However, objectives of two

models are not the same. Indeed, Objective ML aims to extend ML with objects and classes by using the formalization given in [7]. Moreover, the objects are provided as first class objects. $\mathcal{FML}$ has not such an aim. Indeed, collections are not first class objects although objects are used to give them an operational semantics. Moreover, collections have not states as objects. Instead, we focus on entities of collections which are first class objects.
Lastly, $\mathcal{FML}$ does not provide sub-typing as Objective ML. However, $\mathcal{FML}$ allows species to be applied on collections whose interfaces are greater than ones expected. This form of sub-typing is sufficient for our purposes and the one of FOCAL. Moreover since collections are not first class object, there are no reason to add sub-typing as in Objective ML.

$\mathcal{FML}$ only takes in account the operational aspects of FOCAL. It is not designed to provide specification and proof aspects. Indeed, $\mathcal{FML}$ uses the object calculus [7] in order to easily give a semantics for the self reference and collections. Objects provides implicit mutual definitions that cannot be easily controlled and can bring logical inconsistencies. To avoid it, FOCAL imposes syntactic restrictions on recursive definitions and ones are not recursive. Moreover, an analysis is done on the graph of method dependencies [11, 12]. Mini-Foc is an other model taking into accounts these considerations [9]. It provides features of species and collection without using the object calculus. However, the approach provided by Mini-Foc is quite different to the one of $\mathcal{FML}$. It is near of modules [13–15] and mixins [16, 1–3]. Comparisons of FOCAL with modules and mixins can be found [9]. In [17], FOCAL examples are coded with mixins.

Lastly, species and collection notions have been used successfully to implement a certified library for computer algebra [18]. Also, they are used to implements security politics destinate for data bases [19]. These two examples how helpful can be these new notions in programming. Then, it would be interesting to bring species and collection notions in another language, as ADA for instance.

## Acknowledgements

I would like to thank Thérèse Hardin, Catherine Dubois, Luigi Liquori for helpful discussions about this work and Charles Morisset for second reading. I am also grateful to the referees of an old version of this paper for their constructive remarks.

## References

1. Bracha, G.: The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. PhD thesis, University of Utah (1992)
2. Ancona, D.: Modular Formal Frameworks for Module Systems. PhD thesis, Dipartimento di Informatica, Universita‘ di Pisa-Genova-Udine (1998)
3. Ancona, D., Zucca, E.: A calculus of module systems. Journal of functional programming **12**(2) (2002) 91–132
4. Duggan, D., Sourelis, C.: Mixin modules. In: ICFP. (1996) 262–273

5. Sun Microsystems: Java (2006) http://java.sum.com.
6. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system release 3.08 Documentation and user's manual. INRIA. (2004) http://pauillac.inria.fr/ocaml/htmlman/.
7. Abadi, M., Cardelli, L.: A Theory of objects. Springer (1996)
8. Rémy, D., Vouillon, J.: Objective ML: An effective object-oriented extension to ML. Theory and Practice of Object Systems **4**(1) (1998) p. 27–50
9. Fechter, S.: Sémantique des traits orientés objet de Focal. Thèse de doctorat, Université Paris 6 (2005)
10. Mitchell, J., Plotkin, G.: Abstract types have existential types. ACM Trans. on Programming Languages and Systems **10**(3) (1988) 470–502 Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.
11. Prevosto, V.: Conception et implantation du langage Foc pour le développement de logiciels certifiés. PhD thesis, Université Paris VI (2003)
12. Prevosto, V., Doligez, D.: Inheritance of algorithms and proofs in the computer algebra library foc. Journal of Automated Reasoning **29**(3-4) (2002) 337–363 Special Issue on Mechanising and Automating Mathematics, In Honor of N.G. de Bruijn.
13. MacQueen, D.B.: Modules for standard ml. Lisp and Functional Programmaming (1984) 198–207
14. Leroy, X.: Manifest types, modules, and separate compilation. In: 21st symposium Principles of Programming Languages, ACM Press (1994) 109–122
15. Harper, R., Lillibridge, M.: A type-theoretic approach to higher-order modules with sharing. In Press, A., ed.: 21st symposium Principles of Programming Languages, Portland, OR (1994) 123–137
16. Hirschowitz, T.: Modules mixins, modules et rcursion tendue en appel par valeur. PhD thesis, Universit Paris VII (2003)
17. Hirschowitz, T.: Mixin modules, modules, and extended recursion in a call-by-value setting (extended version) (2003)
18. Boulm, S., Hardin, T., Rioboo, R.: Some hints for polynomials in the Foc project. In: Calculemus 2001 Proceedings. (2001)
19. Jaume, M., Morisset, C.: Formalisation and implementation of access control models. In: Information Assurance and Security (IAS'05) International Conference on Information Technology, ITCC, IEEE CS Press (2005) 703–708