Towards a formal definition of the Foc language

Stéphane Fechter[1] and Catherine Dubois[2]

[1] stephane.fechter@lip6.fr
Laboratoire d'Informatique de Paris VI
8, rue du Capitaine Scott
75015 Paris, France
[2] dubois@iie.cnam.fr
Cedric-IIE
18, allée Jean Rostand
91025 Evry, France

Abstract. The FOC project develops a formal language to implement certified components called collections. These collections are specified and implemented step by step: the programmer describes formally the properties of the algorithms, the context in which they are executed, the data representation and proves formally that the implemented algorithms satisfies the specified properties. This programming paradigm implies the use of classic oriented-object features and the use of module features like interfaces and encapsulation of data representation. In this paper we formalize a kernel of the FOC language whose main ingredients are multiple inheritance, late binding, overriding, interfaces and encapsulation of the data representation. We specify formally the semantics, the type system, the soundness of the typing discipline.

Résumé Le projet FOC développe un langage formel pour implanter des composants certifiés appelés collections. Ces collections sont spécifiées et implantées pas à pas : le programmeur décrit formellement les propriétés des algorithmes, le contexte dans lequel ils sont exécutés, la représentation des données et prouve formellement que les algorithmes implantés satisfont les propriétés spécifiées. Ce paradigme de programmation implique l'utilisation de traits orientés objets classiques et l'utilisation de certain traits des modules comme les interfaces et l'encapsulation de la représentation des données. Dans ce papier on formalise un noyau du langage FOC dont les ingrédients principaux sont le multi-héritage, la liaison retardée, les interfaces et l'encapsulation de la représentation des données. On spécifie formellement la sémantique, le système de type, la sûreté du typage.

1 Introduction

The FOC project[3] [1] develops a formal language to implement certified components called collections. These collections are specified and implemented step by

[3] French acronym for Formel OCAML and COQ

step: the programmer describes formally the properties of the algorithms, the context in which they are executed and the data representation. The language allows the correctness of the code with respect to the specified properties: the programmer can write formal proofs that are verified by the proof checker Coq. The properties and the algorithms are organized hierarchically in structures with an object oriented flavor: inheritance, late binding, encapsulation, refinement. This makes the specification reuse easier. These object-oriented features are at the same time powerful but limited with respect to the state of the art of object oriented languages. Our purpose is to deliver certified components equipped with correctness proofs. And consequently it has an impact on the structure of the development, on the dependencies we can accept. A static analysis ensures that nasty dependencies are rejected [2].

The final code, written in OCaml, is obtained by translation of the ultimate specifications contained in the collections. The generated code is quite efficient thanks to the optimizations discovered by the static analysis.

Up to now, the Foc project has applied the language and methodology to the computer algebra domain. More precisely, computer algebra served as a model and gave the principal guidelines to implement the Foc language [1]. Thus the Foc approach is validated by a wide computer algebra library, developed by Rioboo [3], that includes some complex algorithms with performance comparable to the best existing computer algebra systems. For example, the library provides algorithms to compute the polynomial resultant of two polynomials with some original polynomial representations.

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

Carrier type, multiple inheritance, late binding, encapsulation, refinement are the elementary ingredients of our approach that ensure that the generated code satisfies the specified properties. The purpose of this article is to formalize these elementary ingredients. We formally define the type system and semantics of the core language we call [4]. The typing discipline is proven sound with respect to the semantics.

In section 2, we present informally the core features of the Foc language and illustrate them with examples coming from computer algebra. The terminology fits well to this domain and then can be intuitively understood. However knowledge in computer algebra is not required to read this paper. We compare the Foc concepts with notions coming form other paradigms. Then we detail Otarie: syntax (section 4), type discipline (section 5) and semantics (section 6). In the last section, we conclude and propose perspectives.

2 An overview of Foc

In this section, we illustrate the main features of Foc with the help of computer algebra examples. The Foc environment allows us to describe general algebraic structures such as setoids. A setoid is a set equipped with a reflexive, symmetric and transitive binary relation. At this level of description, the representation of the elements of a setoid, is let abstract. The species `setoid` is written in Foc as follows:

```
species setoid =
  rep ;

  sig equal in  self->self->bool;

  property equal_reflexive : all x in self,
    !equal(x,x);

  property equal_symmetric : all x y in self,
    !equal(x,y) -> !equal (y,x);

  property equal_transitive : all x y z in self,
    !equal(x,y) -> !equal(y,z) -> !equal(x,z);

end
```

In the previous example, the keyword `rep` introduces the carrier type which is not yet defined. The sentence `sig equal in self->self->bool` constitutes

[4] otarie is the French word for sea-lion. The model name comes from a joke about the language name Foc which is an homonym for the French translation of the English word seal

the declaration of the relation `equal`: it is a binary relation whose parameters are two elements of the setoid (`self` is their type). This method is only declared, not yet defined, it can be compared to a virtual method in an object oriented language. The properties about the `equal` relation are introduced by `equal_reflexive`, `equal_symetric` and `equal_transitive`. The symbol `!` in front of `equal` has the same signification than the variable *self* used in class-based languages like OCAML [4]. Thus `!equal` is some syntactic sugar for `self!equal` that denotes the `equal` method of the collection that implements the species (represented by `self`). In the definition of the properties, `all x in self` means for any element `x` of the collection built from `setoid`. In the construction `property`, `->` denotes the logical implication.

Now, we can describe an additive monoid from a setoid by adding an operation `plus` and a neutral element `zero`. For this purpose, we construct `additive_monoid` by inheritance of `setoid`. Then we add the properties `zero_is_neutral` (`zero` is the right neutral element) and `plus_is_associative` (`plus` is associative). Since `additive_monoid` inherits from `setoid` (and consequently, it inherits the relation `equal`), we can use `!equal` to describe these properties.

```
species additive_monoid
        inherits setoid =

 sig zero in self;
 property zero_is_unique : all x o in self,
  !equal(x,!plus(x,o)) -> !equal(!plus(o,x),x)
  -> !equal(o,!zero) ;

 sig plus in self-> self -> self;

 property zero_is_neutral : all x in self,
  !equal(!plus(x,!zero),x) and
  !equal(!plus(!zero,x),x) ;

 property plus_is_associative : all x y z in self,
  !equal(!plus(x,!plus(y,z)),!plus(!plus(x,y),z));

end
```

Now, we create an additive monoid whose elements are integers. As above, we create a species `additive_monoid_integers` that inherits from `additive_monoid`. Then we define the carrier type with `rep=int;` where `int` is the type of integers. And we give a definition for every declaration introduced previously. We also add the constant `one`. In the species and in all the "daughter" species, the entities are implemented as integers and the programmer can use this information.

```
species additive_monoid_integers
        inherits additive_monoid =
 rep=int;

 let zero in self = 0;
 let equal( x in self, y in self) in bool =
     #int_eq(x,y);
 let plus ( x in self, y in self) in self =
     #int_plus(x,y);
 let is_zero ( x in self) in bool =
     #int_eq(x,!zero);

 let one in self = 1;

 proof of equal_reflexive =     (* proof *);
 proof of equal_symmetric =     (* proof *);
 proof of equal_transitive =    (* proof *);

 proof of plus_is_associative = (* proof *);
 proof of zero_is_neutral =     (* proof *);

end
```

In the above species, `#int_plus` and `#int_eq` are predefined operations (the OCaml ones) for integer addition and equality.
Now the equality and the operations are defined, it is possible to prove the properties. The formal proofs (not detailed in the example) are introduced by `proof of ...`. These formal proofs can be done directly with the Coq prover or with an adhoc prover under development in the project. However some proofs will not be done because the involved operations are too low level. In this case, we trust OCaml.

At this level, we could derive another species from `additive_monoid_integers` in order to redefine the method `plus`. This would imply to prove again the properties `plus_is_associative` and `zero_is_neutral` because they depend on the definition of `plus`. In this context, let us define `plus` as an operation manipulating and computing integers modulo two. A classical approach would consist in testing the parameters in order to reject integers different from 0 and 1. However it is not very efficient because of the supplementary tests. So we define a predicate `is_modulo_2` (with the construction `letprop`) to express that the only entities of modulo 2 integers are the integers 0 and 1. This property is the representation invariant. Then we formulate and prove the property `plus_modulo_2`: for all integers `x` and `y` satisfying the representation invariant, the result of the addition of `x` and `y` satisfies also the representation invariant. Thus, `one` and `zero`, satisfying themselves the representation invariant, (plus zero one) is again an integer modulo 2.

```
species modulo_2_integers
        inherits additive_monoid_integers =

  let plus (x in self, y in self) in self =
      let r=#int_plus(x,y) in
       if (!equal(r,2)) then 0 else r;

  proof of plus_is_associative = (* new proof *);
  proof of zero_is_neutral     = (* new proof *);

  letprop is_modulo_2 (x in self) = !equal(x,0) or !equal(x,1);

  theorem plus_modulo_2:
   all x y in self,
    !is_modulo_2(x) -> !is_modulo_2(y) -> !is_modulo_2(!plus(x,y))
  proof: ...

  theorem zero_modulo_2: !is_modulo_2(!zero)
  proof: ...
  theorem one_modulo_2: !is_modulo_2(!one)
  proof: ...

end
```

The species modulo_2_integers is totally defined. Then we can build a collection c from this species.

```
collection c implements modulo_2_integers
```

To prevent the end user from using the entities *in a bad manner*, as for example in the expression c!plus(2,5), the carrier type is made abstract. The creation of the collection c comes with this encapsulation. Then the type of plus becomes c -> c -> c where c is the name of the collection. And consequently the entities denoted by one and two have the type c. Thus, plus may take as parameters one and two. Generally speaking, only elements generated by the operations defined in the collection c, that is, having the result type c, can be used with to plus:

```
let r=c!plus(c!one,c!zero);;
c!plus(r,c!one);;
```

As one and zero satisfy the representation invariant, we are sure that plus returns a modulo two integer stored in the variable r. And r can be used again as a parameter. Thus the representation invariant is never broken.

In a species, every method has an associated type more or less imposed by the programmer (with type annotations parameters or declarations). For example, the method `plus` is defined by the user with the type `self -> self -> self`. Such a type is called the **method interface**. The set of method interfaces associated with their method names, for a given species, is called the **interface of the species**. A collection has also an interface, provided by the interface , the species from the collection derives. And every occurrence of `self` is replaced by the name of the collection.

Furthermore, the FOC language offers multiple inheritance. The convention, in case of conflicts, is to choose the definition from the right most species.
The language is more restrictive about the carrier type. For example, in the species `modulo_2_integers`, we cannot redefine `rep` as `bool`. Moreover, if a species inherits from several other species, then the inherited carrier types must be the same. We can explain easily this restriction in the computer algebra domain: a species represents an algebraic structure that relies on a carrier set. This set is given once for all, so changing the representation of its elements would change the nature of this set. Moreover, in the framework of FOC, instead of changing the carrier type from `int` to `bool`, we would create a species with an abstract carrier type and two derived species (inheriting from the first one), one with `int` as its carrier type and another one with `bool` as its carrier type.

Lastly, FOC provides parameterized species:

```
species cartesian_setoid (c1 is setoid, c2 is setoid) =
  rep = c1 * c2;

  let fst ( x in self ) in c1 = #first(x) ;
  let snd ( x in self ) in c2 = #scnd(x)  ;
end
```

The species `cartesian_setoid` has two parameters `c1` and `c2` representing two collections whose interface is derived from `setoid`.
As the collection name can be considered as a type, we can use `c1` and `c2` to define the carrier type of `cartesian_setoid`. Here, an entity of `cartesian_setoid` is a pair made of an entity of `c1` and an entity of `c2`.
In `cartesian_setoid`, we provide two methods `fst` and `snd` to access the components of an entity. These methods use the OCAML projections, `#first` and `#snd`.
Let suppose two species `bool_setoid` and `int_setoid`, totally defined, with `bool` and `int` as the carrier type definitions. Let `c_bool_setoid` and `c_int_setoid` the collections created respectively from the species `bool_setoid` and `int_setoid`. Thus `c_bool_setoid` and `c_int_setoid` have interfaces derived from the `setoid` interface. Therefore, we can use `c_bool_setoid` and `c_int_setoid` as parameters for `cartesian_setoid`. As this application provides a new species totally defined, we can create a new collection from it:

```
collection c implements
```

```
cartesian_setoid(c_bool_setoid, c_int_setoid)
```

Neither species nor collections are first class objects, even if collection may be used as parameters of species.

3 Relative works

Inheritance, late binding, method redefinition are features common to Foc and class based objects oriented languages. However, there are differences. First of all, the Foc carrier type, fundamental ingredient of our approach, has no counter-part in the OO world. Another important difference is that a Foc species has no state. We could consider the carrier type as a built-in method, usually virtual in the early stages of the development. Any method of a species or a collection is applied to entities which have a Caml type, not an object type.

Nevertheless with objects we cannot obtain the Foc encapsulation when we create a collection: the carrier type must be made abstract while it is manifest (in the same way as [5]) in the species that allowed to derive the collection.

A collection can be compared to a module of functional languages. In this context, a collection is close to a structure providing a type whose definition is hidden, that is an opaque type, and functions to handle elements of that opaque type.

Species are also close to mixin modules (see [6]). Both have defined components and deferred components (declared but not yet defined). Defining a deferred method in the Foc context can be compared to the operation of the sum of two mixins.

The ingredients found in Foc and formalized in our core language named Otarie are not new, they come from class based languages and modules. They are consistently mixed to provide a framework to develop certified components by taking advantage of specification and code reuse.

4 Presentation of Otarie

The previous section has presented different features of Foc, in particular those related to the carrier type, the interface and its abstraction. A first formalization has been presented in [7,8]. However this work was not incorporating encapsulation and interfaces. In this paper we come back to this formal model and adapt it to take into account the interfaces and the encapsulation of entities.

Our formal definition of Otarie has been inspired by Objective ML [9], a class-based model that serves as the foundations of the programming language Ocaml. And, even if a collection is closer to a module than an object, a species can be considered as a class, more precisely a virtual class since it may contain deferred methods. A collection can be seen as an object, an instance of a class, but it is not a first class value. For example, it's impossible to write a function taking as a parameter, any collection having at least a method m.

In Otarie, we consider a set of constants $cst \in \mathcal{K}$, a set of variables $x \in \mathcal{X}$, a set of collections $c \in \mathcal{C}$, a set of species names $z \in \mathcal{Z}$, a set of name methods $m \in \mathcal{M}$. All these sets are enumerable.

Fig. 1. carrier type definition and method interface

carrier type definition :

$t ::=$	τ_{cst}	atomic type like `int`, `bool`, etc...
$\mid$	$\mathtt{In}(c)$	carrier type reference of the collection c
$\mid$	$t \rightarrow t \mid t * t$	

method interface :

$i ::=$	`rep`	abstraction annotation
$\mid$	$\tau_{cst} \mid \mathtt{In}(c)$	
$\mid$	$i \rightarrow i \mid i * i$	

The syntax to define a carrier type is given in the figure 1. A carrier type as defined by the developer can mix atomic types, constructors and collection names. In this latter case, the collection name is considered as a type name. Although, in a FOC program, we use indifferently the name c to denote both the collection and irs carrier type, we prefer to adopt in Otariea disambiguous syntax. So we write $\mathtt{In}(c)$ instead of c in a carrier type. Thus, by using $\mathtt{In}(c)$, the developer refers to the carrier type of the collection c. However he has an abstract vision from it. In other words, an expression of type $\mathtt{In}(c)$, must be considered as an encapsulated entity of the collection c.

To define a method interface, the syntactic category i is used. In the same way, an interface is composed of regular types and collection names (tagged with $\mathtt{In}()$) and incorporate generally occurrences of `rep` (`self` in FOC). By using `rep`, the developer specifies that the corresponding parameter or result is an entity of the species that he is writing. Thanks to the annotation `rep`, even if the carrier type definition is `int`, in the method interface `rep` $\rightarrow$ `int`, we can do the distinction between an integer that is an entity and an integer that is not.This information will help when creating and abstracting a collection.

Although the syntactic category t is included in i, we do the distinction between the two categories because the annotation `rep` must not be used to define a carrier type. Indeed, the use of `rep` to define a carrier type has no meaning since `rep` refers to the carrier type itself. Moreover, this syntactic distinction avoids us not to add supplementary rules in the type system.

The main syntax of Otarie is described in the figure 2. It's an extension of core ML (constant, variable, function, application, pair and local definition) with

Fig. 2. main syntax

$a ::= cst \mid x \mid fun(x).\ a \mid a\ a \mid (a,a) \mid$ `let` $x = a$ `in` a	core ML
$\mid\ col!m$	method invocation
$\mid$ `collection` $c = e$ `in` a	collection definition
$\mid$ `species` $z = e$ `in` a	species definition

three constructions. The first construction $col!m$ is the invocation of a method m on a collection col. The second construction `collection` $c = e$ `in` a, is the creation of the collection c from the species e. The user can access the collection through the name c in the expression a. It reflects the construction `collection c implements species_name` of FOC. In relation to Objective ML, the second construction is close to the creation of an object from a class e. But because of the abstraction mechanism, we provide a scope for any introduced collection.
The third construction, `species` $z = e$ `in` a, aims to name the species e. Thus the collection e is identified by z through the expression a. By definition of a, z can never be used directly in a. Only species expressions or creations of collections will be able to use the species name z.

Fig. 3. collection

$col ::= c$	name collection
$\mid$ `self`	current collection
$\mid\ \langle w \rangle$	executive collection

The syntax for a collection is described in the figure 3. A collection may be a collection name, `self` to denote the current collection or an **executive collection**, that is a list of defined methods.

Fig. 4. definition of fields

$w ::= \varnothing \mid d;\ w$

$d ::= m : i = a$	method
$\mid$ `rep` $= t$	carrier type definition
$\mid$ `inherit` e	inherited species

The fields, described in the figure 4, are used to define executive collections or species body. A field d can be :

– a method $m : i = a$ where i is its type or interface (syntax given in the figure 1) and a its definition (syntax given in the figure 2)

- the definition of the carrier type rep $= t$ where t is a carrier type definition (syntax given in the figure 1)
- an inheritance declaration inherit e (syntax given in the figure 5).

Fig. 5. syntax of species

$e ::= z$	
$\mid$ struct w end	species structure
$\mid$ $fun(c: \ [m : i]\ ).\ e$	parameterized species
$\mid$ $e\ col$	application on a species

Lastly, the species syntax is described in figure 5. The main form of a species, called a **species structure**, is struct w end where w is its body. The body is a list of fields. A parameterized species is written $fun(c: \ [m : i]\ ).\ e$ where c is the collection parameter. The notation $[m : i]$ is a list of method names m associated with their interface i (whose syntax is described in figure 1). This list represents the interface of the parameter c. The Foc species species sp_name (c is oth_sp) = body end is translated in Otarie as follows :

species sp_name $= fun(c: \ [m : i]\ ).$ struct body end in ...

where $[m : i]$ is the corresponding interface of the species oth_sp.

Finally, e *col* is the application the species e on the collection *col*.

The translation of a Foc program into a Otarie program is quite easy. For example:

```
species foo =
  rep;
  sig inc  in self -> self;
  let inc2 (x in self) in self = !inc(x);
end
```

corresponds to the following Otarie program:

```
species foo =
 struct
   inc2 : rep → rep =  fun(x).  self!inc x
 end
in ...
```

where the declaration inc and rep are made implicit. Indeed, Otarie provides implicit declarations and doesn't constrain us to write rep; when the carrier type is not yet defined. Although it was possible to make ones explicit, we use an implicit version in order to simplify the presentation of Otarie.

The species `foo`, for example, can be extended by inheritance in oder to make a collection :

```
species foo2 inherits foo2 =
 rep = int;
 let elt in self = 0;
 let inc (x in self) in self = #int_plus(x,1);
end

collection c implements foo2 ;;

c!inc2(c!elt);;
```

Thus, the corresponding Otarie program is :

```
species foo2 =
 struct
  inherit foo;
  rep = int;
  elt : rep = 0;
  inc : rep → rep =  fun(x). #int_plus x
 end

in collection c = foo2 in

c!inc2 c!elt
```

5 The type system of Otarie

5.1 Type language

Main types. The main types, corresponding to main expressions a, are described in figure 6. A type τ can be a type variable α, an atomic type (e.g. `int`, `bool`, etc ...), a collection name, a functional type or a product type. The occurrences of collection names appearing in a type, are considered as type names.

Fig. 6. Main types

$$\boxed{\tau ::= \alpha \mid \tau_{cst} \mid c \mid \tau \to \tau \mid \tau * \tau}$$

List of field types. Lists of fields have their class type Φ described in figure 7. A type Φ contains two sorts of field types :

– the method type $(m : \iota)$ where m is the name of the method and ι, its interface.
– the carrier type $\mathtt{rep} = \tau$ where τ corresponds to a carrier type t as given by the developer.

On the list of field types Φ, we suppose an axiom of left-commutativity :

$$f_1;\ f_2;\ \Phi = f_2;\ f_1;\ \Phi$$

where f_1 and f_2 are field types. Thanks to this axiom, we can retrieve easily a method name or `rep` in Φ without being constrained by any order.

We define an union operation $\oplus$ on lists of field types. This operation requires the two argument lists are identical on the intersection of their domain. In other words, if there is a field type $m : \iota$ (resp. $\mathtt{rep} = \tau$) in Φ_1 and there is a field type $m : \iota'$ (resp. $\mathtt{rep} = \tau'$) in Φ_2, $\Phi_1 \oplus \Phi_2$ requires that ι and ι' are equal (resp. τ and τ' be equal).

Fig. 7. list of field types

$$\begin{array}{l} \iota\ ::= \alpha \mid \tau_{cst} \mid c \mid \mathtt{rep} \mid \iota \to \iota \mid \iota * \iota \\ \Phi ::= \varnothing \mid m : \iota;\ \Phi \mid \mathtt{rep} = \tau;\ \Phi \end{array}$$

We add meta-notations on Φ (see figure 8) in order to distinguish lists of fields types without occurrences of $\mathtt{rep} = \tau$ (see Φ_d), those with a unique occurrence of $\mathtt{rep} = \tau$ (see Φ_c) and those that may be followed by a row variable ρ (see Φ_e).

Fig. 8. meta-notations

$$\begin{array}{l} \Phi_d \triangleq \Phi \backslash \{\mathtt{rep} = \tau\} \\ \Phi_c \triangleq (\mathtt{rep} = \tau;\ \Phi_d) \\ \Phi_e \triangleq \Phi_d \mid \Phi_d;\ \rho \end{array}$$

Collection types. The syntax of a collection type is described in the figure 9. It is composed of a list of field types Φ_c optionally followed by the row variable ρ. By definition of Φ_c, a collection type has a unique occurrence of $\mathtt{rep} = \tau$. Thus, we impose that a collection has mandatory a unique carrier type. On the other hand, the presence of $\mathtt{rep} = \tau$ allows us to bind occurrences of annotation `rep` in method types. In other words, `rep` plays the role of an existential type whose witness is τ (given by $\mathtt{rep} = \tau$).
The row variable is useful for the parameterized species. It permits to apply a collection whose interface is larger than the one written in the species parameter.

Fig. 9. collection types

$$\boxed{\tau_{col} ::= \langle \Phi_c \rangle \mid \langle \Phi_c;\ \rho \rangle}$$

Species types. The species types γ, in figure 10, can take the form `sig` (τ_{col}) Φ `end` for species structures or $\tau_{col} \rightarrow \gamma$ for parameterized species. The species types `sig` (τ_{col}) Φ `end` is composed of two parts :

- Φ represents the list of field types whose corresponding fields are defined in the species (directly in the structure or by inheritance). We call this type the **list of defined field types** (type of the defined fields).
- τ_{col} represents the type of the underlying executive collection, that is the future collection created from the species. We call this type the **signature**. Among other things, it permits to build a fix point (see typing rules further) in order to resolve the self reference and the late binding. Thus, the variable `self` will be assigned the type τ_{col}.

A method name m present in τ_{col}, but not in Φ, is considered as virtual, that is in Foc words, m is only declared. Similarly, if `rep` $= \tau$ is not present in Φ, it means that the carrier type is not yet defined.
If all method names and `rep` $= \tau$ present in τ_{col} are also declared in Φ, then the species is totally defined. Consequently all methods are defined and a definition for the carrier type is given. Such a species is said "concrete".

The species and collection types are very similar to class and object types of Objective ML. Furthermore, as in Objective ML, methods and carrier types cannot be polymorphic. In other words, the methods in species and collections are monomorph. However our actual experience with Foc shows the polymorphic methods are not indispensable. Generally parameterized species provide the solution. It is also mandatory to forbid free type variables in a carrier type. Indeed, let us consider the following example :

```
species foo =
 rep = 'a                          (* 'a is a free type variable *)
 let elt in self = true
 let m (x in self ) in self = x + 1
end

collection c implements foo = end

c!m c!elt;;
```

Since `'a` is a free type variable, the method definitions are correct according to their specifications. Thus, the application of `c!elt` on `c!m` is correct. However, at run-time, the incorrect result `true + 1` is obtained.
To avoid this kind of problem, the syntactic category t (see figure 1) doesn't

provide the possibility to define a carrier type with type variable occurrences. And in the type system, the free variables are captured in type schemes or eliminated through the typing rules.

Fig. 10. species types

$$
\boxed{\begin{array}{l} \gamma ::= \texttt{sig}\ (\tau_{col})\ \Phi\ \texttt{end} \\ \quad\ \mid\ \tau_{col} \rightarrow \gamma \end{array}}
$$

Type schemes. For the sequel we consider the following type schemes :

$$
\begin{array}{l} \sigma_\tau ::= \forall\bar{\alpha}.\tau \\ \sigma_\gamma ::= \forall\bar{\alpha}\forall\rho.\gamma \end{array}
$$

where $\bar{\alpha}$ denotes for a set of type variables $\alpha_1, \ldots, \alpha_n$ (possibly empty). ρ is a row variable (possibly absent).

We denote by $\tau \leqslant \sigma_\tau$ (respectively $\tau \leqslant \sigma_\tau$) that τ (resp. γ) is a type instance of the type scheme σ_τ (resp. σ_γ).

5.2 Notations

Since there are several syntactic categories we use for the sequel the following meta-notations :

$$
\begin{array}{l} \breve{a} \triangleq a \mid w \mid col \mid e \\ \breve{\tau} \triangleq \tau \mid \Phi \mid \tau_{col} \mid \gamma \end{array}
$$

These meta-notations are used consistently. For instance, $(\breve{a}, \breve{\tau})$ means (a, τ), (w, Φ), etc... but not (e, τ).

5.3 Rules

The typing rules, presented in the figures 11, 12, 13 and 15, allow to certify or not that an expression is well-typed in a given context. This context is a pair of environments.

The first environment is a typing environment defined by :

$$
\begin{array}{ll} A ::= \varnothing & \\ \quad\ \mid\ A + x : \sigma_\tau \mid A + z : \sigma_\gamma & \\ \quad\ \mid\ A + c : \tau_{col} \mid A + \texttt{self} : \tau_{col} & \end{array}
$$

We note A^* the typing environment A deprived of `self`.

The second environment is a collection name environment :

$$
\Omega ::= \varnothing \mid \Omega;\ c \text{ where } c \text{ does not belong to } \Omega
$$

We call a **well-formed** typing environment according to a collection name environment Ω an environment A such that :
for all $x : \sigma_\tau \in A$ (respectively for all $z : \sigma_\gamma \in A$, for all $c : \tau_{col} \in A$, for all $\texttt{self} : \tau_{col} \in A$), all occurrences of collection names in σ_τ (respectively σ_γ, τ_{col}) are declared in Ω.

The typing rules use typing judgments whose form is $A \; ; \; \Omega \vdash \breve{a} : \breve{\tau}$, meaning the expression $\breve{a}$ is well typed and has the type $\breve{\tau}$ with respect to the context $A; \; \Omega$.

We say that a judgment $A \; ; \; \Omega \vdash \breve{a} : \breve{\tau}$ is **well-formed** if A is well-formed according to the collection name environment Ω and all the occurrences of collection names in $\breve{\tau}$ are declared in Ω.

We define the generalisation $Gen(\breve{\tau}, A)$ by $\forall \bar{\alpha}.\breve{\tau}$ where $\bar{\alpha}$ are the variables of $\breve{\tau}$ that are not free in A.

Main typing rules. The rules in figure 11 correspond to expressions of the main syntax. The rules VAR, FUN-ML, APP-ML, PAIR-ML and LET-ML coming from ML, are classical.
The method invocation m of a collection *col* is verified with the rule SEND. This rule is close to the one used for objects in [9]. The expression $col!m$ has a type τ' if the type of *col* contains the field type $(m : \iota)$ and a carrier type $(\texttt{rep} = \tau)$. Since occurrences of `rep` can be in ι, τ' must be equal to ι where all occurrences of `rep` are replaced by τ.
The type verification for a collection definition is done with the rule ABSTRACT. The creation of a collection with $\texttt{collection}\ c = e\ \texttt{in}\ a$, is authorized only if the species e is totally defined: the type of e indicates that the carrier type is well defined and all methods are defined since there are $\texttt{rep} = \tau$ and Φ_d in the signature and in the list of defined field types. The uses of the new collection c in the expression a are verified in the second premise of the rule ABSTRACT. For this, we extend the collection name environment Ω with c. By construction, c must be fresh with respect to Ω. Globally, it means the name c must be different from all the other ones already introduced in the collection name environment Ω. In FOC, every collection has a unique name. So the property is syntactically satisfied.Then, we extend the type environment A with the collection name c associated with the type $\langle \texttt{rep} = c; \; \Phi_d \rangle$. This type is built from the signature of the species type where the carrier type is replaced by the collection name c. By this way, the carrier type becomes abstract (like a private type in ADA, for example). Thus, the collection c in the expression a is abstracted and the type of $\texttt{collection}\ c = e\ \texttt{in}\ a$ is the type τ' of the expression a.
Lastly, the rule SPECIES LET, permitting to check the type of the expression $\texttt{species}\ z = e\ \texttt{in}\ a$, is similar to the rule LET-ML.

Fig. 11. main typing rules

VAR
$$\frac{\tau \leqslant A(x)}{A\ ;\ \Omega \vdash x : \tau}$$

FUN-ML
$$\frac{A + x : \tau_1\ ;\ \Omega \vdash a : \tau_2}{A\ ;\ \Omega \vdash fun(x).\ a : \tau_1 \rightarrow \tau_2}$$

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

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

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

SEND
$$\frac{A\ ;\ \Omega \vdash col : \langle \texttt{rep} = \tau; m : \iota; \Phi_d \rangle}{A\ ;\ \Omega \vdash col!m : \iota[rep \leftarrow \tau]}$$

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

SPECIES LET
$$\frac{A\ ;\ \Omega \vdash e : \gamma \qquad A + z : Gen(\gamma, A)\ ;\ \Omega \vdash a : \tau}{A\ ;\ \Omega \vdash \texttt{species}\ z = e\ \texttt{in}\ a : \tau}$$

Collection typing rules. The rule for collections are presented in the figure 12. For the collection name and the variable `self`, the rules Collection name and Self are respectively used. These simple rules consist in retrieving the type associated with identifier in the typing environment.
The rule for executive collection $\langle w \rangle$ is the same as the one used for the objects in Objective ML. The collection $\langle w \rangle$ has the type $\langle \Phi_c \rangle$ if w has the type Φ_c. The environment A^*, in the premise, is extended with $\texttt{self} : \langle \Phi_c \rangle$ in order to provide the self reference for w.

Fig. 12. collection typing rules

Collection name
$$\frac{}{A\ ;\ \Omega \vdash c : A(c)}$$

Self
$$\frac{}{A\ ;\ \Omega \vdash \texttt{self} : A(\texttt{self})}$$

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

Typing rules for fields. The fields are type-checked with the rules of the figure 13. These rules use the $|\texttt{-}|_A$ forms (see figure 14) that translates any type t in a type when all occurrences of collection names are replaced by their carrier type if such names are found in A.

In figure 13, the rules Method, Carrier type and Inherit type-check (respectively the method, the carrier type definition and the inheritance) field. During the type-checking of a carrier type definition $\texttt{rep} = t$, we must validate the carrier type t given by the programmer. If valid, this type must be the appearing carrier type in the type of `self`.

To type-check a method $m : i = a$, we first verify that the method interface is well-formed. Then we must check that the variable `self` has a collection type $\langle \texttt{rep} = \tau; m : \iota; \Phi_d \rangle$ where the method name m is present with a type equal to the interface. Lastly, we type-check the body a of the method. Its type must be the type ι where all occurrences of `rep` are substituted by τ. By these different verifications, we check that the method m of the underlying collection has a type coherent (in our context, coherent means equal modulo the substitution of $\texttt{In}(c)$) with respect to the interface given by the programmer. Moreover, we check that the definition of the method is correct according to the specification.

The rule for the inheritance `inherit` e is the same as in [9]. The species e must be type-checked in a context where `self` is associated with the underlying collection. In order to take into account the right variable `self`, the signature of the type species e must be the type of variable `self` found in the current environment. Thus the type for `inherit` e, is the list of defined field types from the type of species e.

The type-checking of a list of fields uses the rules BASIC and THEN. The first rule is trivial. The second rule type-checks the head of the list (by using rules INHERIT, CARRIER TYPE and METHOD), that is a field whose type is Φ_1. Then it type-checks the rest of the list whose type is Φ_2. Thus, the type for the entire list is $\Phi_1 \oplus \Phi_2$. Consequently, when a method is redefined, its type cannot be changed. In a similar way, the carrier type cannot be redefined. This is enforced by the $\oplus$ operator which requires that the two arguments share commun types on the intersection of their domains.
The rules BASIC and THEN are almost the ones of Objective ML. The rule THEN of Objective ML must take in account the *super* binders in addition, features not provided by FOC.

Fig. 13. typing rules for fields

BASIC
$$\frac{}{A\ ;\ \Omega \vdash \varnothing : \varnothing}$$

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

CARRIER TYPE
$$\frac{A\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep} \ = |t|_A;\ \Phi_d \rangle}{A\ ;\ \Omega \vdash \texttt{rep} = t : (\texttt{rep} = |t|_A)}$$

METHOD
$$\frac{A\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \tau;\ m : \iota;\ \Phi_d \rangle \qquad A\ ;\ \Omega \vdash a : \iota[\texttt{rep} \leftarrow \tau] \qquad \text{where } |i|_A = \iota}{A\ ;\ \Omega \vdash m : i = a : (m : \iota)}$$

Fig. 14. Verification of interfaces and carrier type definitions

$$\begin{array}{ll}
|\tau_{cst}|_A & = \tau_{cst} \\
|\texttt{rep}|_A & = \texttt{rep} \\
|\texttt{In}(c)|_A & = \tau \text{ if } c : \langle \texttt{rep} = \tau;\ \Phi_d \rangle \in A \\
|t_1 \rightarrow t_2|_A & = |t_1|_A \rightarrow |t_2|_A \\
|t_1 * t_2|_A & = |t_1|_A * |t_2|_A
\end{array}$$

Typing rules of species. The type-checking for the species uses the rules of the figure 15.

To type-check a species structure, we use the rule SPECIES BODY. This rule is identical to the rule for class structure in Objective ML. In the the body w of `struct` w `end`, there are invocation of methods on `self`. Thus we must type-check w on the starry current environment augmented with the variable `self`. As for the rule INHERIT, if the environment is starry, it's to avoid conflict problems. The type for the variable `self` must be the one from the signature of the type of e. The list of defined field types for e, is built with the list of field types of w.

The rule SPECIES FUN is used to check a parameterized species $fun(c : \ [m : i]\ ).\ e$. Its type $\langle \mathtt{rep} = \tau';\ [m : \iota];\ \Phi_e \rangle \rightarrow \gamma'$ ($[m : \iota]$ is the list of method names m associated with their type ι) specifies that the parameter is a collection providing at least the methods m detailed in the interface with types following the ones given in the interface. Then, the rule checks the species e. This is done by increasing the current collection name environment with c, and by increasing the current typing environment A with $c : \langle \mathtt{rep} = c;\ [m : \iota] \rangle$. We use $\mathtt{rep} = c$ in the type of c, in order to see c as an abstract collection different from the other ones used in the species e.
The type τ' seems independent of the rule and chosen randomly. But it's not really exact in most of the time. Indeed, we shall have in mind that other rules intervene on the derivation tree whose the expression $fun(c : \ [m : i]\ ).\ e$ is an element of it. Thus the type τ' is constrained by the other rules employed to derive the tree. On the other hand, we would understand that the name c is quantified universally. From this fact, therefore all substitution of c by other types is available. Thus we can apply any collection of any form on parameterized species seeing that the collection posses the same interface as the one imposed by the parameter.
The Φ_e list appearing in the type, allows to apply a collection whose its interface is greater than $[m : i]$, that is an interface containing $[m : i]$ and other method interfaces.

Lastly, the application of a collection on parameterized species is type-checked by the rule SPECIES APP. This rule is homologous to the rule APP-ML.

6 Semantics

In order to formalize the execution of a FOC program, we provide a reduction semantics with *a call by value* strategy for Otarie. Then we prove our typing discipline is sound with respect to this semantics.

Semantics is described by a set of small-step reduction rules (see figure 17) and a set of contexts (see figure 18). Thus the evaluation of an expression, if it terminates, can be visualized step by step until obtaining an expression that can't be reduced anymore.

Fig. 15. typing rules of species

Species name
$$\frac{\gamma \leqslant A(z)}{A\ ;\ \Omega \vdash z : \gamma}$$

Species Body
$$\frac{A^* + \texttt{self} : \tau_{col}\ ;\ \Omega \vdash w : \Phi}{A\ ;\ \Omega \vdash \texttt{struct}\ w\ \texttt{end} : \texttt{sig}\ (\tau_{col})\ \Phi\ \texttt{end}}$$

Species Fun
$$\frac{A + c : \langle \texttt{rep} = c;\ [m : \iota]\rangle\ ;\ (\Omega;\ c) \vdash e : \gamma \qquad \text{where } |i|_A = \iota}{A\ ;\ \Omega \vdash fun(c : [m : i]\ ).\ e : \langle \texttt{rep} = \tau', [m : \iota];\ \Phi_e\rangle \rightarrow \gamma[c \leftarrow \tau']}$$

Species App
$$\frac{A\ ;\ \Omega \vdash e : \tau_{col} \rightarrow \gamma \qquad A\ ;\ \Omega \vdash col : \tau_{col}}{A\ ;\ \Omega \vdash e\ col : \gamma}$$

The values are described in the figure 16. Every syntactic category has a corresponding category of values. First, we find standard values ML v : constant, abstraction and pair of values. The value of a list of fields is a list where there is no more overriding on method names and `rep` (one occurrence of `rep` at most). In other words, a value for a list of fields is a list where inheritance and redefinition have been resolved. Such a value is used to define a collection value $\langle v_w \rangle$ or a species value, in particular a species structure `struct` v_w `end`. Lastly, the parameterized species are also values.

Fig. 16. Values

$$\begin{array}{lll}
v & ::= & cst \mid fun(x).\ a \mid (v, v) \\
 & & \\
v_{col} & ::= & \langle v_w \rangle \\
 & & \\
v_s & ::= & \texttt{struct}\ v_w\ \texttt{end} \\
 & \mid & fun(c :\ [m : i]\ ).\ e \\
 & & \\
v_w & ::= & \varnothing \mid v_d;\ v_w \\
v_d & ::= & m : i = a \mid \texttt{rep} = t
\end{array}$$

Let us now comment the elementary reduction rules detailed in the figure 17. The rules 1 and 2 are the standard β-reduction ML rules.

The rule 3, very similar to the one provided for objects in [9], reduces the method of an executive collection $\langle v_w(m)\rangle!m$: it returns the body $v_w(m)$ of the method m and replaces every occurrence of `self` in $v_w(m)$ by the executive collection itself. This substitution allows to compute the self reference.

The rule 4 replaces the collection name c by its executive form $\langle v_w \rangle$, in an expression a. It's done if the species, used to instantiate the collection, is a value `struct` v_w `end`, that is a species where inheritance has been resolved and

Fig. 17. reduction rules

1 $(fun(x).\ a)\ v$	$\rightarrow_\epsilon a[v/x]$
2 $\texttt{let}\ x = v\ \texttt{in}\ a$	$\rightarrow_\epsilon a[v/x]$
3 $\langle v_w \rangle!m$	$\rightarrow_\epsilon v_w(m)[\langle v_w \rangle/\texttt{self}]$
4 $\texttt{collection}\ c = (\texttt{struct}\ v_w\ \texttt{end})\ \texttt{in}\ a$	$\rightarrow_\epsilon a[\langle v_w \rangle/c][CT(\langle v_w \rangle)/\texttt{In}(c)]$
5 $\texttt{species}\ z = v_s\ \texttt{in}\ a$	$\rightarrow_\epsilon a[v_s/z]$
6 $m : i = a;\ v_w$	$\rightarrow_\epsilon v_w$ if $m \in dom(v_w)$
7 $\texttt{rep} = t;\ v_w$	$\rightarrow_\epsilon v_w$ if $\texttt{rep} \in dom(v_w)$
8 $\texttt{inherit}\ (\texttt{struct}\ v_w\ \texttt{end});\ w$	$\rightarrow_\epsilon v_w\ @\ w$
9 $(fun(c:\ [m:i]\ ).\ e)\ v_{col}$	$\rightarrow_\epsilon e[v_{col}/c][CT(v_{col})/\texttt{In}(c)]$

where all fields are defined. Moreover, as the collection is now executive, all occurrences of $\texttt{In}(c)$ must be replaced by the carrier type found in v_w, denoted by $CT(\langle v_w \rangle)$.

The rule 5 is analogous to the rule 2: the occurrences of z in a are replaced by the species value v_s.

The rules 6, 7 and 8 are the computation rules for lists of fields. The rules 6 and 7 are related to the redefinition of a field. If a method m already occurs in the list v_w, then the rule 6 returns v_w, its *forgets* the first, that is the old, definition of the method $m : i = a$. The rule 7 does likewise with the $\texttt{rep} = t$ field. The rule 8 is used for resolving inheritance. The inherited species must be a value $\texttt{struct}\ v_w\ \texttt{end}$, the rule concatenates the inherited methods and the possible *rep* field, v_w, with the other methods w.
By combining these previous rules, we resolve the multi-inheritance (by using several times the rule 6) and the method redefinition: the rightmost definition is chosen.

Lastly, the rule 9, very close to the first rule, reduces the application of a parameterized species. However, occurrences of $\texttt{In}(c)$ may appear in the species. These occurrences are replaced by the carrier type of the collection as in the rule 4.

The typing system presented previously is sound with respect to our semantics. Formally, it consists in two properties: the preservation of the type by reduction (also called the subject reduction theorem) and the non-locking of well typed programs. The proof of type soundness follows the proof of type soundness for Objective ML (detailed in [9]). The main difference comes from the construction $\texttt{collection}\ c = e\ \texttt{in}\ a$, a lemma establishing that a well-typed collection is also well-typed under its executive form. We detail the proof in the appendix A. The verification of this proof with the Coq proof assistant [10] has been partly done [11]: at the moment, it does not take into account the entities abstraction, this last aspect is under development.

Fig. 18. Reduction context

Context:

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

where [] is the empty context

7 Conclusion and future works

In the first part of this paper we have presented informally the core features of the FOC language. We have then formalized the main constructions of the language.

The main purposes of Otarie in this paper are to explain the different object oriented features and encapsulation possibilities. But we didn't mention logic aspects. Among other things, the self reference provides a naive recursion making easily logic inconsistent. To avoid this problem, FOC provides a dependency analysis on methods (see [2] and [12]). Thus, every method call is certified to terminate. This analysis looks like the one done for mixins, in particular ones presented in [6]. The authors extend their type system with dependency graphs. If a type derivation tree is built with a graph having at least a cycle, then the tree is considered like inconsistent.
In FOC, the mutual recursion, through the methods, is more or less limited. The user must declare explicitly the methods concerned by this sort of recursion. And he must provide a proof of termination.
Thus, in the future, Otarie will have to be extended with such a dependency analysis.
Lastly, the conception of Otarie has been carried out with constraints coming from computer algebra. Most of these constraints appear naturally and independently of the computer algebra domain. An important perspective is to evaluate the constraints on other domains, in order to understand whether they can be relaxed or not. For example, type carrier redefinition could be visited again.

Acknowledgements

We would like to thank Thérèse Hardin, Luigi Liquori and Véronique Viguié Donzeau-Gouge for helpful discussions about this work. We are also grateful to the referees of an old version of this paper for their constructive remarks.

A Proofs of the type soundness for Otarie

A.1 Introduction

Since Otarie has been inspired by Objective ML, the different proofs for the propositions and lemmas are classical and closed to the ones found in [9]. We just present the most interesting and pertinent cases. The other cases can be easily retrieved.

Since we have multiple syntactic categories for expressions, contexts and types, it is convenient to introduce the following meta-notations:

$$\begin{aligned}
\breve{a} &\triangleq a \mid w \mid col \mid e \\
\breve{\tau} &\triangleq \tau \mid \Phi \mid \tau_{col} \mid \gamma \\
\breve{E} &\triangleq E \mid E_{col} \mid E_e \mid F \mid F_d
\end{aligned}$$

These meta-notations are used consistently. For instance, when writen $A\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$, $(\breve{a}, \breve{\tau})$ means (a, τ), (w, Φ), etc, but not (a, γ).

We introduce the relation $\sigma_\tau \geqslant \sigma'_\tau$ (resp. $\sigma_\gamma \geqslant \sigma'_\gamma$) to say that any instance of σ'_τ (resp. σ'_γ) is an instance of σ_τ (resp. σ_γ).

A.2 Proofs

Lemma 1. *Let $|t|_A = \tau$, c a collection name and τ' a carrier type definition. Then $|A[c \leftarrow \tau']|_{\tau[c \leftarrow \tau']} = t$*

Proof. The proof is by induction on t.

Case t is τ_{cst} :

trivial:

$$|A[c \leftarrow \tau']|_{\tau_{cst}[c \leftarrow \tau']} = \tau_{cst}$$

⋆

Case t is `rep` :

similar to the above case.

$\star$

Case t is $\mathtt{In}(c)$:

We have:

$$|\mathtt{In}(c)|_A = \tau \text{ with } c' : \langle \mathtt{rep} = \tau;\ \Phi_d \rangle \in A$$

thus we have:

$$c' : \langle \mathtt{rep} = \tau[c \leftarrow \tau'];\ \Phi_d[c \leftarrow \tau'] \rangle \in A[c \leftarrow \tau']$$

therefore:

$$|A[c \leftarrow \tau']|_{\tau[c \leftarrow \tau']} = \mathtt{In}(c)$$

$\star$

Case t is $t_1 \rightarrow t_2$:

We have:

$$|t_1 \rightarrow t_2|_A = |t_1|_A \rightarrow |t_2|_A$$

with $|t_1|_A = \tau_1$ and $|t_2|_A = \tau_2$

By induction on $|t_1|_A$ and $|t_2|_A$, we have:

$$\begin{aligned} |A[c \leftarrow \tau']|_{t_1[c \leftarrow \tau']} \rightarrow |A[c \leftarrow \tau']|_{t_2[c \leftarrow \tau']} &= \tau_1[c \leftarrow \tau'] \rightarrow \tau_2[c \leftarrow \tau'] \\ &= (\tau_1 \rightarrow \tau_2)[c \leftarrow \tau'] \end{aligned}$$

therefore:

$$|A[c \leftarrow \tau']|_{(\tau_1 \rightarrow \tau_2)[c \leftarrow \tau']} = t_1 \rightarrow t_2$$

$\star$

Case t is $t_1 * t_2$:

similar to the above case

$\star$

$\square$

Lemma 2. *For this lemma, we use the notations* $\breve{\iota} \triangleq \iota \mid \Phi \mid \tau_{col} \mid \gamma$ *and* $a_t \triangleq$ $\mathit{rep} \mid c$.
Let a_{t1} *and* a_{t2} *distinct. Let* τ *and* τ' *two types such as* τ' *doesn't contain occurences of* a_{t2}*. Then the following equality is verified:*

$$(\breve{\iota}[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] = (\breve{\iota}[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau]$$

Proof. The proof is by induction on $\breve{\iota}$.

Case $\breve{\iota}$ is α :

We have:

$$\begin{array}{ll}(\alpha[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \alpha[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \alpha & \end{array}$$

and:

$$\begin{array}{ll}(\alpha[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] & = \\ \alpha[a_{t1} \leftarrow \tau] & = \\ \alpha & \end{array}$$

Therefore, the equality is verified.

⋆

Case $\breve{\iota}$ is τ_{cst} :

The proof is similar to the previous case.

⋆

Case $\breve{\iota}$ is c :

There are three sub-cases:

- case $c \neq a_{t1}$ and $c \neq a_{t2}$:
  The proof is similar to the previous case.
- case $c = a_{t1}$ (and $a_{t2} \neq c$, by hypothesis) :
  we have:

$$\begin{array}{ll}(c[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \tau[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \tau & \text{because } \tau \text{ doens't have contain of } a_{t2}\end{array}$$

and:

$$\begin{array}{ll}(c[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] & = \\ c[a_{t1} \leftarrow \tau] & = \\ \tau & \end{array}$$

Therefore the equality is verified.

- case $c = a_{t2}$ (and $a_{t1} \neq c$, by hypothesis) :
  We have:

$$\begin{array}{ll}(c[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ c[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \tau'[a_{t1} \leftarrow \tau] & \end{array}$$

  and:

$$\begin{array}{l}(c[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] = \\ \tau'[a_{t1} \leftarrow \tau]\end{array}$$

  Therefore the equality is verified.

⋆

Case $\breve{\imath}$ is `rep` :

There are three sub-cases:

- case $a_{t1} \neq \texttt{rep}$ and $a_{t2} \neq \texttt{rep}$:
  We have:

$$(\texttt{rep}[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] = \texttt{rep}$$

  and:

$$(\texttt{rep}[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] = \texttt{rep}$$

  Therefore the equality is verified.
- case $a_{t1} = \texttt{rep}$ (and $a_{t2} \neq \texttt{rep}$, by hypothesis):
  We have:

$$\begin{array}{ll}(\texttt{rep}[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \tau[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \tau \quad \text{(by knowing that } \tau \text{ doesn't contain occurences of } a_{t2}) & \end{array}$$

  and:

$$\begin{array}{ll}(\texttt{rep}[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] & = \\ \texttt{rep}[a_{t1} \leftarrow \tau] & = \\ \tau & \end{array}$$

  Therefore the equality is verified.
- case $a_{t2} = \texttt{rep}$ (and $a_{t1} \neq \texttt{rep}$, by hypothesis):
  We have:

$$\begin{array}{ll}(\texttt{rep}[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \texttt{rep}[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \tau'[a_{t1} \leftarrow \tau]\ ] & \end{array}$$

  and:

$$\begin{array}{l}(\texttt{rep}[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] = \\ \tau'[a_{t1} \leftarrow \tau]\end{array}$$

  Therefore the equality is verified.

$\star$

Case $\breve{\iota}$ is $\iota_1 \rightarrow \iota_2$:

We have:

$$\begin{array}{ll}(\iota_1 \rightarrow \iota_2)[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ (\iota_1[a_{t1} \leftarrow \tau] \rightarrow \iota_2[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ \iota_1[a_{t1} \leftarrow \tau][a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] \rightarrow \iota_2[a_{t1} \leftarrow \tau][a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & \end{array}$$

By induction on $\tau_i[a_{t1} \leftarrow sup][a_{t2} \leftarrow sup'[a_{t1} \leftarrow sup]\ ]$ (for i equal 1 and 2), we have:

$$\begin{array}{l} = (\iota_1[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] \rightarrow (\iota_2[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] \\ = (\iota_1[a_{t2} \leftarrow \tau'] \rightarrow \iota_2[a_{t2} \leftarrow \tau'])[a_{t1} \leftarrow \tau] \\ = ((\iota_1 \rightarrow \iota_2)[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] \end{array}$$

Therefore the equality is verified.

$\star$

Case $\breve{\iota}$ is $\iota_1 * \iota_2$:

The proof is similar to the previous case.

$\star$

Case $\breve{\iota}$ is $\varnothing$:

trivial:

$$(\varnothing[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] = (\varnothing[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau]$$

$\star$

Case $\breve{\iota}$ is $(m : \iota;\ \Phi)$:

We have:

$$\begin{array}{ll}((m : \iota;\ \Phi)[a_{t1} \leftarrow \tau])\ [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ (m : \iota[a_{t1} \leftarrow \tau];\ \Phi[a_{t1} \leftarrow \tau])[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ] & = \\ (m : \iota[a_{t1} \leftarrow \tau][a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ];\ \Phi[a_{t1} \leftarrow \tau][a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ]) & \end{array}$$

By induction on $\iota[a_{t1} \leftarrow \tau][a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ]$ et $\Phi[a_{t1} \leftarrow \tau][a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau]\ ]$, therefore we have:

$$\begin{array}{l} = m : (\iota[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau];\ (\Phi[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] \\ = (m : \iota[a_{t2} \leftarrow \tau'];\ \Phi[a_{t2} \leftarrow \tau'])[a_{t1} \leftarrow \tau] \\ = ((m : \iota;\ \Phi)[a_{t2} \leftarrow \tau'])\ [a_{t1} \leftarrow \tau] \end{array}$$

$\star$

Case $\breve{\iota}$ is $\texttt{rep} = \tau;\ \Phi$:

The proof is similar to the previous cas, namely the definition of τ is included in the one of ι.

$\star$

Case $\breve{\iota}$ is $\langle \Phi_c \rangle$:

We have:

$$(\langle \Phi_c ; \rangle [a_{t1} \leftarrow \tau]) \; [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau] \;] = \\ \langle (\Phi_c [a_{t1} \leftarrow \tau]) \; [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau] \;] \rangle$$

By induction on $(\Phi_c [a_{t1} \leftarrow \tau]) \; [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau] \;]$, namely the definition of Φ_c is included in the one of Φ, we have:

$$= \langle (\Phi_c [a_{t2} \leftarrow \tau']) \; [a_{t1} \leftarrow \tau] \rangle \\ = (\langle \Phi_c \rangle [a_{t2} \leftarrow \tau']) \; [a_{t1} \leftarrow \tau]$$

Therefore the equality is verified.

$\star$

Case $\breve{\iota}$ is $\langle \Phi_c ; \; \rho \rangle$:

The proof is similar to the above case.

$\star$

Case $\breve{\iota}$ is $\texttt{sig}$ (τ_{col}) Φ $\texttt{end}$:

On a :

$$((\texttt{sig} \; (\tau_{col}) \; \Phi \; \texttt{end}) \; [a_{t1} \leftarrow \tau]) \; [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau] \;] \qquad = \\ (\texttt{sig} \; (\tau_{col}[a_{t1} \leftarrow \tau]) \; \Phi[a_{t1} \leftarrow \tau] \; \texttt{end}) \; [a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau] \;] \qquad = \\ \texttt{sig} \; ((\tau_{col}[a_{t1} \leftarrow \tau])[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau] \;]) \; (\Phi[a_{t1} \leftarrow \tau])[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau] \;] \; \texttt{end}$$

By induction on $(\tau_{col}[a_{t1} \leftarrow \tau])[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau] \;]$ and on $\Phi[a_{t1} \leftarrow \tau])[a_{t2} \leftarrow \tau'[a_{t1} \leftarrow \tau] \;]$ we have:

$$= \texttt{sig} \; ((\tau_{col}[a_{t2} \leftarrow \tau']) \; [a_{t1} \leftarrow \tau]) \; (\Phi[a_{t2} \leftarrow \tau']) \; [a_{t1} \leftarrow \tau] \; \texttt{end} \\ = (\texttt{sig} \; (\tau_{col}[a_{t2} \leftarrow \tau']) \; \Phi[a_{t2} \leftarrow \tau'] \; \texttt{end}) \; [a_{t1} \leftarrow \tau] \\ = (\; (\texttt{sig} \; (\tau_{col}) \; \Phi \; \texttt{end})[a_{t2} \leftarrow \tau'] \;) \; [a_{t1} \leftarrow \tau]$$

The equality is verified.

$\star$

Case $\breve{\iota}$ is $\tau_{col} \rightarrow \gamma$:

The proof is similar to the case for $\breve{\iota}$ is $\iota_1 \rightarrow \iota_2$.

⋆

□

Property 1 (Application of collection names). Let $\Omega;c$ a collection name environment, A a type environment well-formed in relation to $\Omega;c$. Let τ_{car} a type such as:

- τ_{car} doesn't contain occurrence of c and occurrence of type variable.
- all collection name in τ_{car} is declared in Ω.

Then $A\ ;\ (\Omega;\ c) \vdash \breve{a} : \breve{\tau}$ implies $A[c \leftarrow \tau_{car}];\ \Omega \vdash \breve{a} : \breve{\tau}[c \leftarrow \tau_{car}]$

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

Case Var :

We have:

$$\frac{\tau \leqslant A(x)}{A\ ;\ (\Omega;\ c) \vdash x : \tau}$$

By the premise, we have

$$\tau[c \leftarrow \tau_{car}] \leqslant (A(x))[c \leftarrow \tau_{car}]$$

$\tau[c \leftarrow \tau_{car}]$ doesn't contain any occurrence of c and $(A(x))[c \leftarrow \tau_{car}] = A[c \leftarrow \tau_{car}](x)$. Then $\tau[c \leftarrow \tau_{car}] \leqslant A[c \leftarrow \tau_{car}](x)$.

Therefore:

$$\frac{\tau[c \leftarrow \tau_{car}] \leqslant A[c \leftarrow \tau_{car}](x)}{A\ ;\ \Omega \vdash x : \tau[c \leftarrow \tau_{car}]}$$

⋆

Case Fun-ML :

We have:

$$\frac{A + x : \tau_1\ ;\ (\Omega;\ c) \vdash a : \tau_2}{A\ ;\ (\Omega;\ c) \vdash fun(x).\ a : \tau_1 \rightarrow \tau_2}$$

By induction hypothesis on the premise, therefore:

$$\frac{A[c \leftarrow \tau_{car}] + x : \tau_1[c \leftarrow \tau_{car}];\ \Omega \vdash a : \tau_2[c \leftarrow \tau_{car}]}{A[c \leftarrow \tau_{car}];\ \Omega \vdash fun(x).\ a : (\tau_1 \rightarrow \tau_2)[c \leftarrow \tau_{car}]}$$

⋆

Case App-ML :

We have:

$$\frac{A\ ;\ (\Omega;\ c) \vdash a_1 : \tau' \rightarrow \tau \qquad A\ ;\ (\Omega;\ c) \vdash a_2 : \tau'}{A\ ;\ (\Omega;\ c) \vdash a_1\ a_2 : \tau}$$

By induction hypothesis on the premises, therefore:

$$\frac{\begin{array}{c} A[c \leftarrow \tau_{car}];\ \Omega \vdash a_1 : \tau'[c \leftarrow \tau_{car}] \rightarrow \tau[c \leftarrow \tau_{car}] \\ A[c \leftarrow \tau_{car}];\ \Omega \vdash a_2 : \tau'[c \leftarrow \tau_{car}] \end{array}}{A[c \leftarrow \tau_{car}];\ \Omega \vdash a_1\ a_2 : \tau[c \leftarrow \tau_{car}]}$$

$\star$

Case Pair-ML :

This case is similar to the above one.

$\star$

Case Let-ML :

We have:

$$\frac{A\ ;\ (\Omega;\ c) \vdash a_1 : \tau_1\ _{\mathbf{(1)}} \qquad A + x : Gen(\tau_1, E)\ ;\ (\Omega;\ c) \vdash a_2 : \tau_2\ _{\mathbf{(2)}}}{A\ ;\ (\Omega;\ c) \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau_2}$$

By induction hypothesis on the premises ($\mathbf{1}$) et ($\mathbf{2}$) we have:

$$A[c \leftarrow \tau_{car}];\ \Omega \vdash a_1 : \tau_1[c \leftarrow \tau_{car}]$$

and

$$A[c \leftarrow \tau_{car}] + x : (Gen(\tau_1, E))[c \leftarrow \tau_{car}];\ \Omega \vdash a_2 : \tau_2[c \leftarrow \tau_{car}]$$

By hypothesis, τ_{car} doesn't contain occurrence of type variable. Then we have:

$$Gen(\tau_1, E)[c \leftarrow \tau_{car}] = Gen(\tau_1[c \leftarrow \tau_{car}], A[c \leftarrow \tau_{car}])$$

Therefore:

$$\frac{\begin{array}{c} A[c \leftarrow \tau_{car}];\ \Omega \vdash a_1 : \tau_1[c \leftarrow \tau_{car}] \\ A[c \leftarrow \tau_{car}] + x : Gen(\tau_1[c \leftarrow \tau_{car}], A[c \leftarrow \tau_{car}]);\ \Omega \vdash a_2 : \tau_2[c \leftarrow \tau_{car}] \end{array}}{A[c \leftarrow \tau_{car}];\ \Omega \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau_2[c \leftarrow \tau_{car}]}$$

$\star$

Case Send :

We have:

$$\frac{A\ ;\ (\Omega;\ c) \vdash col : \langle \mathtt{rep} = \tau';\ m : \iota;\ \Phi_d \rangle}{A\ ;\ (\Omega;\ c) \vdash col!m : \iota[rep \leftarrow \tau']}$$

By induction hypothesis on the premise we have:

$$\frac{A[c \leftarrow \tau_{car}];\ \Omega \vdash col : \langle \texttt{rep} = \tau'[c \leftarrow \tau_{car}];\ m : \iota[c \leftarrow \tau_{car}];\ \Phi_d[c \leftarrow \tau_{car}]\rangle}{A[c \leftarrow \tau_{car}];\ \Omega \vdash col!m : (\iota[c \leftarrow \tau_{car}])\ [\texttt{rep} \leftarrow \tau'[c \leftarrow \tau_{car}]]}$$

By the lemma 2 (τ_{car} doesn't contain any `rep` by definition) we have:

$$(\iota[c \leftarrow \tau_{car}])\ [\texttt{rep} \leftarrow \tau'[c \leftarrow \tau_{car}]] = (\iota[rep \leftarrow \tau'])\ [c \leftarrow \tau_{car}]$$

Therefore:

$$A[c \leftarrow sup];\ \Omega \vdash col!m : (\iota[rep \leftarrow \tau'])\ [c \leftarrow \tau_{car}]$$

⋆

Case Abstract :

On a :

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

By the premisse (**2**), c' is fresh in relation to $(\Omega;\ c)$. Therefore $c \neq c'$.
By hypothesis, τ_{car} can contain only occurences of collection names belonging to Ω. Then τ_{car} can't contain occurrence of c'.
Therefore by induction hypothesis applied on the premises (**1**) and (**2**):

$$\frac{\begin{array}{c} A[c \leftarrow \tau_{car}]; \Omega \vdash e : \texttt{sig}\ (\langle \texttt{rep} = \tau'[c \leftarrow \tau_{car}]; \Phi_d[c \leftarrow \tau_{car}]\rangle)\ (\texttt{rep} = \tau'[c \leftarrow \tau_{car}]; \Phi_d[c \leftarrow \tau_{car}])\ \texttt{end} \\ A[c \leftarrow \tau_{car}] + c' : \langle \texttt{rep} = c'; \Phi_d[c \leftarrow \tau_{car}]\rangle;\ (\Omega;\ c') \vdash a : \tau[c \leftarrow \tau_{car}] \end{array}}{A[c \leftarrow \tau_{car}];\ \Omega \vdash \texttt{collection}\ c' = e\ \texttt{in}\ a : \tau[c \leftarrow \tau_{car}]}$$

By knowing $\tau[c \leftarrow \tau_{car}]$ doesn't contain any occurrence of c' according to the previous remark.

⋆

Case Species Let :

This case is similar to the Let-ML case.

⋆

Case Collection name :

Trivial

⋆

Case Self :

Trivial

⋆

Case Executive collection :

We have:

$$\frac{A^* + \texttt{self} : \langle \Phi_c \rangle \ ; \ (\Omega; \ c) \vdash w : \Phi_c}{A \ ; \ (\Omega; \ c) \vdash \langle w \rangle : \langle \Phi_c \rangle}$$

By induction hypothesis on the premise, we have:

$$\frac{A^*[c \leftarrow \tau_{car}] + \texttt{self} : \langle \Phi_c[c \leftarrow \tau_{car}] \rangle; \ \Omega \vdash w : \Phi_c[c \leftarrow \tau_{car}]}{A[c \leftarrow \tau_{car}]; \ \Omega \vdash \langle w \rangle : \Phi_c[c \leftarrow \tau_{car}]}$$

⋆

Case Basic :

trivial

⋆

Case Then :

We have:

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

By applying $[c \leftarrow \tau_{car}]$ in the same time on Φ_1 and on Φ_2, $\Phi_1[c \leftarrow \tau_{car}]$ and $\Phi_2[c \leftarrow \tau_{car}]$ stay compatible. Therefore $\Phi_1[c \leftarrow \tau_{car}] \ \oplus \ \Phi_2[c \leftarrow \tau_{car}] = (\Phi_1 \oplus \Phi_2)[c \leftarrow \tau_{car}]$.

Therefore by induction hypothesis applied on the premises, we have:

$$\frac{A[c \leftarrow \tau_{car}]; \ \Omega \vdash d : \Phi_1[c \leftarrow \tau_{car}] \qquad A[c \leftarrow \tau_{car}]; \ \Omega \vdash w : \Phi_2[c \leftarrow \tau_{car}]}{A[c \leftarrow \tau_{car}]; \ \Omega \vdash d; \ w : (\Phi_1 \oplus \Phi_2)[c \leftarrow \tau_{car}]}$$

⋆

Case Inherit :

We have:

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

By induction hypothesis applied on the premises, we have:

$$\frac{\begin{array}{c} A[c \leftarrow \tau_{car}]; \ \Omega \vdash \texttt{self} : \tau_{col}[c \leftarrow \tau_{car}] \\ A[c \leftarrow \tau_{car}]; \ \Omega \vdash e : \texttt{sig} \ (\tau_{col}[c \leftarrow \tau_{car}]) \ \Phi[c \leftarrow \tau_{car}] \ \texttt{end} \end{array}}{A[c \leftarrow \tau_{car}]; \ \Omega \vdash \texttt{inherit} \ e : \Phi[c \leftarrow \tau_{car}]}$$

⋆

Case Carrier Type :

We have:

$$\frac{A\ ;\ (\Omega;\ c) \vdash \texttt{self} : \langle \texttt{rep} = |t|_A ; \Phi_d \rangle\ _{\mathbf{(1)}}}{A\ ;\ (\Omega;\ c) \vdash \texttt{rep} = t : (\texttt{rep} = |t|_A)}$$

By the lemma 1 we have:

$$(|t|_A)[c \leftarrow \tau_{car}] = |A[c \leftarrow \tau_{car}]|_t$$

Thus by induction hypothesis applied on the premise (**1**), we have:

$$\frac{A[c \leftarrow \tau_{car}];\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = |A[c \leftarrow \tau_{car}]|_t;\ \Phi_d[c \leftarrow \tau_{car}] \rangle}{E[c \leftarrow \tau_{car}];\ \Omega \vdash \texttt{rep} = t : (\ \texttt{rep} = |A[c \leftarrow \tau_{car}]|_t\ )}$$

That is:

$$E[c \leftarrow \tau_{car}];\ \Omega \vdash \texttt{rep} = t : (\ \texttt{rep} = |t|_A\ )[c \leftarrow \tau_{car}]$$

⋆

Case Method :

We have:

$$\frac{\begin{array}{c} A\ ;\ (\Omega;\ c) \vdash \texttt{self} : \langle \texttt{rep} = \tau';\ m : \iota;\ \Phi_d \rangle\ _{\mathbf{(1)}} \\ A\ ;\ (\Omega;\ c) \vdash a : \iota[\texttt{rep} \leftarrow \tau']\ _{\mathbf{(2)}} \qquad \text{where } |i|_A = \iota\ _{\mathbf{(3)}} \end{array}}{A\ ;\ (\Omega;\ c) \vdash m : i = a : (m : \iota)}$$

By induction hypothesis applied on the premises (**1**) et (**2**), then by application of the lemma 1 on (**3**) we have:

$$A[c \leftarrow \tau_{car}];\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \tau'[c \leftarrow \tau_{car}];\ m : \iota[c \leftarrow \tau_{car}];\ \Phi_d[c \leftarrow \tau_{car}] \rangle\ ,$$

$$A[c \leftarrow \tau_{car}];\ \Omega \vdash a : (\iota[rep \leftarrow \tau'])[c \leftarrow \tau_{car}]$$

and

$$|A[c \leftarrow \tau_{car}]|_{\iota[c \leftarrow \tau_{car}]} = i$$

Since $\texttt{rep}$ is not contained in τ_{car} by defininition, we have by the lemma 2:

$$(\iota[c \leftarrow \tau_{car}])\ [rep \leftarrow \tau'[c \leftarrow \tau_{car}]\ ] = (\iota[rep \leftarrow \tau'])\ [c \leftarrow \tau_{car}]$$

Therefore:

$$\frac{\begin{array}{c} A[c \leftarrow \tau_{car}];\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \tau'[c \leftarrow \tau_{car}];\ m : \iota[c \leftarrow \tau_{car}];\ \Phi_d[c \leftarrow \tau_{car}] \rangle \\ A[c \leftarrow \tau_{car}];\ \Omega \vdash a : (\iota[c \leftarrow \tau_{car}])\ [rep \leftarrow \tau'[c \leftarrow \tau_{car}]\ ] \\ \text{where } |A[c \leftarrow \tau_{car}]|_{\iota[c \leftarrow \tau_{car}]} = i \end{array}}{A[c \leftarrow \tau_{car}];\ \Omega \vdash m : i = a : (m : \iota)[c \leftarrow \tau_{car}]}$$

⋆

Case Species Name :

This case is similar to the one for Var

⋆

Case Species Body :

We have:

$$\frac{A^* + \mathtt{self} : \tau_{col}\ ;\ (\Omega;\ c) \vdash w : \Phi}{A\ ;\ (\Omega;\ c) \vdash \mathtt{struct}\ \ w\ \ \mathtt{end} : \mathtt{sig}\ (\tau_{col})\ \Phi\ \mathtt{end}}$$

By induction hypothesis on the premise, we have:

$$\frac{A^*[c \leftarrow \tau_{car}] + \mathtt{self} : \tau_{col}[c \leftarrow \tau_{car}];\ \Omega \vdash w : \Phi[c \leftarrow \tau_{car}]}{A[c \leftarrow \tau_{car}];\ \Omega \vdash \mathtt{struct}\ \ w\ \ \mathtt{end} : \mathtt{sig}\ (\tau_{col}[c \leftarrow \tau_{car}])\ \Phi[c \leftarrow \tau_{car}]\ \mathtt{end}}$$

Therefore:

$$A[c \leftarrow \tau_{car}];\ \Omega \vdash \mathtt{struct}\ \ w\ \ \mathtt{end} : (\mathtt{sig}\ (\tau_{col})\ \Phi\ \mathtt{end})[c \leftarrow \tau_{car}]$$

⋆

Case Species Fun :

We have:

$$\frac{A + c' : \langle \mathtt{rep} = c'; [m : \iota] \rangle\ ;\ (\Omega;\ c', c) \vdash e : \gamma\ _{(\mathbf{1})} \qquad \text{where } |i|_A = \iota\ _{(\mathbf{2})}}{A\ ;\ (\Omega;\ c) \vdash fun(c' : [m : i]).\ e : \langle \mathtt{rep} = \tau';\ [m : \iota];\ \Phi_e \rangle \rightarrow \gamma[c' \leftarrow \tau']}$$

We have $c \neq c'$ since c' is fresh in relation to Ω.
By hypothesis, all collection name into τ_{car} is declared in Ω. c' is fresh relation to Ω, then τ_{car} doesn't contain occurrence of c'.
By induction application on the premise ($\mathbf{1}$), then by the lemma 1 applied to the side condition ($\mathbf{2}$), we have:

$$\frac{\begin{array}{c} A[c \leftarrow \tau_{car}] + c' : \langle \mathtt{rep} = c'; [m : \iota[c \leftarrow \tau_{car}]\ ] \rangle;\ (\Omega; c') \vdash e : \gamma[c \leftarrow \tau_{car}] \\ |A[c \leftarrow \tau_{car}]|_{\iota[c \leftarrow \tau_{car}]} = i \end{array}}{\begin{array}{r} A[c \leftarrow \tau_{car}];\ \Omega \vdash fun(c' : [m : i]).\ e : \langle \mathtt{rep} = \tau'[c \leftarrow \tau_{car}];\ [m : \iota[c \leftarrow \tau_{car}]\ ]; \Phi_e[c \leftarrow \tau_{car}] \rangle \\ \rightarrow (\gamma[c \leftarrow \tau_{car}])[c' \leftarrow \tau'[c \leftarrow \tau_{car}]\ ] \end{array}}$$

And we have:

$$\begin{array}{l} \langle \mathtt{rep} = \tau'[c \leftarrow \tau_{car}];\ [m : \iota[c \leftarrow \tau_{car}]\ ];\ \Phi_e[c \leftarrow \tau_{car}]\ \rangle \rightarrow (\gamma[c \leftarrow \tau_{car}])\ [c' \leftarrow \tau'[c \leftarrow \tau_{car}]\ ] \\ = \langle \mathtt{rep} = \tau'; [m : \iota];\ \Phi_e \rangle [c \leftarrow \tau_{car}]\ \rightarrow\ (\gamma[c \leftarrow \tau_{car}])[c' \leftarrow \tau'[c \leftarrow \tau_{car}]\ ] \end{array}$$

By application of the lemma 2 at the right of $\rightarrow$, by knowing τ_{car} doesn't contain any c', we have:

$$\begin{array}{l} = \langle \mathtt{rep} = \tau'; [m : \iota];\ \Phi_e \rangle [c \leftarrow \tau_{car}]\ \rightarrow\ (\gamma[c' \leftarrow \tau'])[c \leftarrow \tau_{car}] \\ = (\ \langle \mathtt{rep} = \tau'; [m : \iota];\ \Phi_e \rangle\ \rightarrow\ \gamma[c' \leftarrow \tau']\ )\ [c \leftarrow \tau_{car}] \end{array}$$

Therefore:

$$A[c \leftarrow sup];\ \Omega \vdash fun(c : [m : i]).\ e : (\ \langle \mathtt{rep} = \tau'; [m : \iota];\ \Phi_e \rangle\ \rightarrow\ \gamma[c' \leftarrow \tau']\ )\ [c \leftarrow \tau_{car}]$$

⋆

Case Species App :

On a :

$$\frac{A\ ;\ (\Omega;\ c) \vdash e : \tau_{col} \to \gamma \qquad A\ ;\ (\Omega;\ c) \vdash col : \tau_{col}}{A\ ;\ (\Omega;\ c) \vdash e\ col : \gamma}$$

By induction hypothesis applied on the premises, we have:

$$\frac{\begin{array}{c} A[c \leftarrow \tau_{car}];\ \Omega \vdash e : \tau_{car}[c \leftarrow \tau_{car}] \to \gamma[c \leftarrow \tau_{car}] \\ A[c \leftarrow \tau_{car}];\ \Omega \vdash col : \tau_{car}[c \leftarrow \tau_{car}] \end{array}}{A[c \leftarrow \tau_{car}];\ \Omega \vdash e\ a : \gamma[c \leftarrow \tau_{car}]}$$

⋆

□

Lemma 3. *Let A and A', two type environment such as:*

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

Then $|A|_\iota = i$ *implies* $|A'|_\iota = i$

Proof. The proof is by simple induction on $|A|_i$

□

Proposition 1 (Typing stability by hypothesis reenforcement). *Let A and A' two type environment well formed in relation to a collection name environment Ω such as:*

- $dom(A) = dom(A')$
- $A'(\breve{x}) \geqslant A(\breve{x})$ *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 by induction on $A\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$

Case Var :

We have:

$$\frac{\tau \leqslant A(x)\ \mathbf{(1)}}{A\ ;\ \Omega \vdash x : \tau}$$

By hypothesis and the premise $(\mathbf{1})$, we have $\tau \leqslant A'(x)$. As A' is well formed in relation to Ω, we have:

$$\frac{\tau \leqslant A'(x)}{A'\ ;\ \Omega \vdash x : \tau}$$

⋆

Case App - ML :

By simple induction on the premises of App-ML:

$$\frac{A' \ ; \ \Omega \vdash a_1 : \tau_1 \to \tau_2 \qquad A' \ ; \ \Omega \vdash a_2 : \tau_1}{A' \ ; \ \Omega \vdash a_1 \ a_2 : \tau_2}$$

⋆

Case Let - ML :

We have:

$$\frac{A \ ; \ \Omega \vdash a_1 : \tau_1 \ _{(\mathbf{1})} \qquad A + x : Gen(\tau_1, A) \ ; \ \Omega \vdash a_2 : \tau_2 \ _{(\mathbf{2})}}{A \ ; \ \Omega \vdash \texttt{let} \ x = a_1 \ \texttt{in} \ a_2 : \tau_2}$$

By induction on the premise $(\mathbf{1})$ we have:

$$A \ ; \ \Omega \vdash a_1 : \tau_1$$

We know $A'(\breve{x}) \geqslant \breve{x}$ for $\breve{x} \in dom(A)$. Thus all type variables of $A(\breve{x})$ belong to $A'(\breve{\tau})$. Moreover we have $dom(A') = dom(A)$. Therefore all free type variables of A are also free type variables of A'.

We have $Gen(\tau_1, A) = \forall \alpha_1 \ldots \alpha_n . \tau_1$ with $\{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau_1) \backslash \mathcal{L}(A)$.
By the previous remark we have $\mathcal{L}(A) = \mathcal{L}(A')$. Thus $\{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau_1) \backslash \mathcal{L}(A')$.
Then $Gen(\tau_1, A) = Gen(\tau_1, A')$

Therefore we have:
$(A' + x : Gen(\tau_1, A'))(x) \geqslant (A + x : Gen(\tau_1, A))(x)$
for all $x \in dom(A + x : Gen(\tau_1, A))$
Then, by hypothesis, we have:

- $dom(A' + x : Gen(\tau_1, A')) = dom(A + x : Gen(\tau_1, A))$
- $(A' + x : Gen(\tau_1, A'))$ and $(A + x : Gen(\tau_1, A))$ are well-formed in relation to Ω.

Therefore, by induction on the premise $(\mathbf{2})$, we have:

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

⋆

Case Abstract :

We have:

$$\frac{\begin{array}{c}A\ ;\ \Omega \vdash e : \mathtt{sig}\ (\langle \mathtt{rep} = \tau ; \Phi_d \rangle)\ (\mathtt{rep} = \tau ; \Phi_d)\ \mathtt{end}\ _{(\mathbf{1})} \\ A + c : \langle \mathtt{rep} = c ; \Phi_d \rangle\ ;\ (\Omega;\ c) \vdash a : \tau\ _{(\mathbf{2})}\end{array}}{A\ ;\ \Omega \vdash \mathtt{collection}\ c = e\ \mathtt{in}\ a : \tau}$$

By induction on the premise (**1**), we have:

$$A'\ ;\ \Omega \vdash e : \mathtt{sig}\ (\langle \mathtt{rep} = \tau ; \Phi_d \rangle)\ (\mathtt{rep} = \tau ; \Phi_d)\ \mathtt{end}$$

By hypothesis, we have:

- $(A' + c : \langle \mathtt{rep} = c ; \Phi_d \rangle)(\breve{x}) \geqslant (A + c : \langle \mathtt{rep} = c ; \Phi_d \rangle)(\breve{x})$ for all $\breve{x}$ of $(A + c : \langle \mathtt{rep} = c ; \Phi_d \rangle)(\breve{x}))$.
- $dom(A' + c : \langle \mathtt{rep} = c ; \Phi_d \rangle) = dom(A + c : \langle \mathtt{rep} = c ; \Phi_d \rangle)$
- $A' + c : \langle \mathtt{rep} = c ; \Phi_d \rangle$ and $A + c : \langle \mathtt{rep} = c ; \Phi_d \rangle$ are well-formed in relation to $(\Omega;\ c)$ since A and A' are well-formed in relation to Ω and c is fresh in relation Ω.

Thus, by induction on the premise (**2**), we have:

$$A' + c : \langle \mathtt{rep} = c ; \Phi_d \rangle\ ;\ (\Omega;\ c) \vdash a : \tau$$

Therefore, we have:

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

⋆

Case Carrier type :

We have:

$$\frac{A\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}\ \ = |t|_A;\ \Phi_d \rangle}{A\ ;\ \Omega \vdash \mathtt{rep} = t : (\mathtt{rep} = |t|_A)}$$

By induction hypothesis on the premise we have:

$$A'\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}\ \ = |t|_A;\ \Phi_d \rangle$$

By the lemma 3 on $|t|_A$ we have $|A'|_t$. Therefore:

$$\frac{A'\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}\ \ = |A'|_t;\ \Phi_d \rangle}{A'\ ;\ \Omega \vdash \mathtt{rep} = t : (\mathtt{rep} = |A'|_t)}$$

⋆

Case Method :

We have:

$$\frac{\begin{array}{c} A\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \tau;\ m : \iota;\ \Phi_d \rangle\ _{(\mathbf{1})} \\ A\ ;\ \Omega \vdash a : \iota[\texttt{rep} \leftarrow \tau]\ _{(\mathbf{2})} \qquad \text{where } |i|_A = \iota \end{array}}{A\ ;\ \Omega \vdash m : i = a : (m : \iota)}$$

By the lemma 3, $|i|_A = \iota$ implies $|A'|_\iota = i$. Then by induction hypothesis on the premises $(\mathbf{1})$ and $_{(\mathbf{2})}$, we have:

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

$\star$

Case Species Fun :

$$\frac{A + c : \langle \texttt{rep} = c;\ [m : \iota] \rangle\ ;\ (\Omega;\ c) \vdash e : \gamma\ _{(\mathbf{1})} \qquad \text{where } |i|_A = \iota}{A\ ;\ \Omega \vdash fun(c : [m : i]\ ).\ e : \langle \texttt{rep} = \tau', [m : \iota];\ \Phi_e \rangle \rightarrow \gamma[c \leftarrow \tau']}$$

By the lemma 3, $|i|_A = \iota$ implies $|A'|_\iota = i$.
By hypothesis, we have:

- $dom(A + c : \langle \texttt{rep} = c;\ [m : \iota] \rangle) = dom(A' + c : \langle \texttt{rep} = c;\ [m : \iota] \rangle)$
- $(A + c : \langle \texttt{rep} = c;\ [m : \iota] \rangle)(\breve{x}) \geqslant (A' + c : \langle \texttt{rep} = c;\ [m : \iota] \rangle)(\breve{x})$ for $\breve{x} \in dom(A + c : \langle \texttt{rep} = c;\ [m : \iota] \rangle)$
- $A' + c : \langle \texttt{rep} = c; [m : \iota] \rangle$ and $A + c : \langle \texttt{rep} = c; [m : \iota] \rangle$ are well-formed in relation to $(\Omega;\ c)$ since A and A' are well-formed in relation to Ω and c is fresh in relation Ω.

Therefore by induction hypothesis, we have:

$$\frac{A' + c : \langle \texttt{rep} = c;\ [m : \iota] \rangle\ ;\ (\Omega;\ c) \vdash e : \gamma \qquad \text{where } |A'|_\iota = i}{A'\ ;\ \Omega \vdash fun(c : [m : i]\ ).\ e : \langle \texttt{rep} = \tau', [m : \iota];\ \Phi_e \rangle \rightarrow \gamma[c \leftarrow \tau']}$$

$\star$

$\square$

Lemma 4. *If $|i|_A = \iota$, then for all type variable substitution θ, we have $|\theta(A)|_{\theta(\iota)} = i$.*

Proof. The proof is done by simple induction on $|i|_A$. $\square$

Property 2. If $A\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$, then for all type variable subsitution θ such as all collection name brought by θ is declared in Ω , we have $\theta(A)\ ;\ \Omega \vdash \breve{a} : \theta(\breve{\tau})$.

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

Case Var :

We have:

$$\frac{\tau \leqslant A(x)}{A\ ;\ \Omega \vdash x : \tau}$$

Let $A(x) = \forall \alpha_1 \ldots \alpha_n.\tau_x$ where α_i are without of reach of θ.
By the premise, we have:
$\tau = \tau_x[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]$
where all collection name occurences in every τ_i, are declared in Ω.

Then we have:
$(\ \theta(A)\ )(x) = \theta(\ A(x)\ )$
$\qquad = \forall \alpha_1 \ldots \alpha_n.\theta(\tau_x)$
and
$\theta(\tau) = \theta(\check{\tau}_x[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n])$
$\qquad = \theta(\check{\tau}_x)[\alpha_1 \leftarrow \theta(\tau_1), \ldots, \alpha_n \leftarrow \theta(\tau_n)]$
since α_i are without of reach of θ.
Thus, we have:
$\theta(\tau) \leqslant (\ \theta(A)\ )(x)$

Since all collection names brought by θ are declared in Ω, therefore we have:

$$\frac{\theta(\tau) \leqslant (\ \theta(A)\ )(x)}{\theta(A)\ ;\ \Omega \vdash x : \theta(\tau)}$$

⋆

Case Carrier type :

We have:

$$\frac{A\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}\ \ = |t|_A;\ \Phi_d\rangle}{A\ ;\ \Omega \vdash \mathtt{rep} = t : (\mathtt{rep} = |t|_A)}$$

By induction on the premise, we have:
$\theta(A)\ ;\ \Omega \vdash \mathtt{self} : \theta(\ \langle \mathtt{rep}\ \ = |t|_A;\ \Phi_d\rangle\ )$

By the lemma 4, we have $\theta(|t|_A) = |\theta(A)|_t$. Thus we have $\theta(A)\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}\ \ = |\theta(A)|_t;\ \theta(\Phi_d)\rangle$ and we obtain:

$$\frac{\theta(A)\ ;\ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}\ \ = |\theta(A)|_t;\ \theta(\Phi_d)\rangle}{\theta(A)\ ;\ \Omega \vdash \mathtt{rep} = t : (\mathtt{rep} = |\theta(A)|_t)}$$

Therefore:

$$\theta(A)\ ;\ \Omega \vdash \mathtt{rep} = t : \theta(\ (\mathtt{rep} = |t|_A)\ )$$

⋆

Case Method :

We have:

$$\frac{\begin{array}{c}A\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \tau;\ m : \iota;\ \Phi_d\rangle\ _{(\mathbf{1})} \\ A\ ;\ \Omega \vdash a : \iota[\texttt{rep} \leftarrow \tau]\ _{(\mathbf{2})} \qquad \text{where } |i|_A = \iota\end{array}}{A\ ;\ \Omega \vdash m : i = a : (m : \iota)}$$

By induction on the premises $(\mathbf{1})$ and $(\mathbf{2})$, we have:
$\theta(A)\ ;\ \Omega \vdash \texttt{self} : \theta(\ \langle \texttt{rep} = \tau;\ m : \iota;\ \Phi_d\rangle\ )$
and
$\theta(A)\ ;\ \Omega \vdash a : \theta(\ \iota[\texttt{rep} \leftarrow \tau]\ )$
that is:
$\theta(A)\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \theta(\tau);\ m : \theta(\iota);\ \theta(\Phi_d)\rangle$
and
$\theta(A)\ ;\ \Omega \vdash a : \theta(\iota)[\texttt{rep} \leftarrow \theta(\tau)]$

Therefore we have:

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

Thus we have:

$$\theta(A)\ ;\ \Omega \vdash m : i = a : \theta(\ (m : \iota)\ )$$

⋆

Case SPECIES FUN :

We have:

$$\frac{A + c : \langle \texttt{rep} = c;\ [m : \iota]\rangle\ ;\ (\Omega;\ c) \vdash e : \gamma\ _{(\mathbf{1})} \qquad \text{where } |i|_A = \iota}{A\ ;\ \Omega \vdash fun(c : [m : i]\ ).\ e : \langle \texttt{rep} = \tau', [m : \iota];\ \Phi_e\rangle \to \gamma[c \leftarrow \tau']}$$

By induction on the premise $(\mathbf{1})$, we have:
$\theta(A) + c : \langle \texttt{rep} = c;\ [m : \theta(\iota)]\rangle\ ;\ (\Omega;\ c) \vdash e : \theta(\gamma)$

And by the lemma 4 on $|i|_A = \iota$ we have $|\theta(A)|_{\theta(\iota)} = i$.

By hypothesis, θ can bring collection names only declared in Ω. Since, all collection names occurence in τ' and in Φ_e are declared in Ω, then all collection names occurences in $\theta(\tau')$ and in $\theta(\Phi_e)$ are also declared in Ω. Thus we have:

$$\frac{\theta(A) + c : \langle \texttt{rep} = c;\ [m : \theta(\iota)]\rangle\ ;\ (\Omega;\ c) \vdash e : \theta(\gamma) \qquad \text{where } |\theta(A)|_{\theta(\iota)} = i}{\theta(A)\ ;\ \Omega \vdash fun(c : [m : i]\ ).\ e : \langle \texttt{rep} = \theta(\tau'), [m : \theta(\iota)];\ \theta(\Phi_e)\rangle \to \theta(\gamma)[c \leftarrow \theta(\tau')]}$$

Since c is fresh in relation to Ω, θ doesn't provide types with occurrences of c. Thus we have:
$\theta(\gamma)[c \leftarrow \theta(\tau')] = \theta(\ \gamma[c \leftarrow \tau']\ )$
Therefore:
$\theta(A)\ ;\ \Omega \vdash fun(c : [m : i]\ ).\ e : \theta(\ \langle \texttt{rep} = \tau', [m : \iota];\ \Phi_e\rangle \to \gamma[c \leftarrow \tau']\ )$

⋆

□

Lemma 5. *Let A and A' two type environments sush as:*

- $|i|_A = \iota$
- $A(c) = A'(c)$ *for all* $\mathit{In}(c) \in i$

Then $|i|_A = \iota$ *implies* $|A'|_\iota = i$.

Proof. The proof is by simple induction on $|i|_A$. □

Property 3. Let A and A' two type environments, Ω a collection name environment and $\breve{a}$ an expression such as:

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

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

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

Case VAR :

We have:

$$\frac{\tau \leqslant A(x)}{A\ ;\ \Omega \vdash x : \tau}$$

By hypothesis we have $A(x) = A'(x)$ since x is free. Thus $\tau \leqslant A'(x)$. Since A' is well-formed in relation to Ω, therefore we have:

$$\frac{\tau \leqslant A'(x)}{A'\ ;\ \Omega \vdash x : \tau}$$

⋆

Case FUN-ML :

We have:

$$\frac{A + x : \tau_1\ ;\ \Omega \vdash a : \tau_2}{A\ ;\ \Omega \vdash fun(x).\ a : \tau_1 \rightarrow \tau_2}$$

By hypothesis we have:

- $(A + x : \tau_1)(\breve{x}) = (A' + x : \tau_1)(\breve{x})$ for all free $\breve{x}$ in a.
- $(A' + x : \tau_1)$ is well-formed in relation to Ω since $(A + x : \tau_1)$ and A' are well-formed in relation to Ω.

Thus by induction hypothesis on the premise, we have:

$$\frac{A' + x : \tau_1 \ ; \ \Omega \vdash a : \tau_2}{A' \ ; \ \Omega \vdash fun(x).\ a : \tau_1 \rightarrow \tau_2}$$

⋆

Case Abstract :

We have:

$$\frac{\begin{array}{c} A \ ; \ \Omega \vdash e : \mathtt{sig}\ (\langle \mathtt{rep} = \tau ; \Phi_d \rangle)\ (\mathtt{rep} = \tau ; \Phi_d)\ \mathtt{end}\ _{\mathbf{(1)}} \\ A + c : \langle \mathtt{rep} = c ; \Phi_d \rangle \ ; \ (\Omega ;\ c) \vdash a : \tau\ _{\mathbf{(2)}} \end{array}}{A \ ; \ \Omega \vdash \mathtt{collection}\ c = e\ \mathtt{in}\ a : \tau}$$

By hypothesis induction on the premise (**1**) we have:

$$A' \ ; \ \Omega \vdash e : \mathtt{sig}\ (\langle \mathtt{rep} = \tau ; \Phi_d \rangle)\ (\mathtt{rep} = \tau ; \Phi_d)\ \mathtt{end}$$

By hypothesis we have:

- $(A + c : \langle \mathtt{rep} = c ; \Phi_d \rangle)(\breve{x}) = (A' + c : \langle \mathtt{rep} = c ; \Phi_d \rangle)(\breve{x})$ for all free $\breve{x}$ in a.
- $(A' + c : \langle \mathtt{rep} = c ; \Phi_d \rangle)$ is well-formed in relation to $(\Omega ; c)$ since $(A + c : \langle \mathtt{rep} = c ; \Phi_d \rangle)$ is well formed in relation to $(\Omega ; c)$ and A' is well-formed in relation to Ω.

Thus by induction hypothesis on the premise (**2**) we have:

$$A' + c : \langle \mathtt{rep} = c ; \Phi_d \rangle \ ; \ (\Omega ;\ c) \vdash a : \tau$$

Therefore we have:

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

⋆

Case Carrier type :

We have:

$$\frac{A \ ; \ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}\ = |t|_A ;\ \Phi_d \rangle}{A \ ; \ \Omega \vdash \mathtt{rep} = t : (\mathtt{rep} = |t|_A)}$$

By induction on the premise, we have:

$$A' \ ; \ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}\ = |t|_A ;\ \Phi_d \rangle$$

By the lemma 5 we have $|t|_A = |A'|_t$. Therefore we have:

$$\frac{A' \ ; \ \Omega \vdash \mathtt{self} : \langle \mathtt{rep}\ = |A'|_t ;\ \Phi_d \rangle}{A' \ ; \ \Omega \vdash \mathtt{rep} = t : (\mathtt{rep} = |t|_A)}$$

⋆

Case Method :

we have:

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

By the lemma 5 we have $|t|_A = |A'|_t$. Thus by induction hypothesis on the premises we have:

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

⋆

Case Species Fun :

we have:

$$\frac{A + c : \langle \texttt{rep} = c ; \ [m : \iota] \rangle \ ; \ (\Omega ; \ c) \vdash e : \gamma \qquad \text{where } |i|_A = \iota}{A \ ; \ \Omega \vdash fun(c : [m : i] \ ). \ e : \langle \texttt{rep} = \tau', [m : \iota]; \ \Phi_e \rangle \rightarrow \gamma[c \leftarrow \tau']}$$

By hypothesis we have:

- $(A + c : \langle \texttt{rep} = c; \ [m : \iota] \rangle)(\breve{x}) = (A' + c : \langle \texttt{rep} = c; \ [m : \iota] \rangle)(\breve{x})$ for all free $\breve{x}$ in e.
- $(A' + c : \langle \texttt{rep} = c; \ [m : \iota] \rangle)$ is well-formed in relation to $(\Omega; \ c)$ since $A + c : \langle \texttt{rep} = c; \ [m : \iota] \rangle)$ is well-formed in relation to $(\Omega; \ c)$ and A' is well-formed in relation to Ω.

Thus by hypothesis induction on the premise we have:

$$A' + c : \langle \texttt{rep} = c; \ [m : \iota] \rangle \ ; \ (\Omega; \ c) \vdash e : \gamma$$

By the lemma 5 we have $|t|_A = |A'|_t$. Therefore we have:

$$\frac{A' + c : \langle \texttt{rep} = c ; \ [m : \iota] \rangle \ ; \ (\Omega ; \ c) \vdash e : \gamma \qquad \text{where } |A'|_\iota = i}{A' \ ; \ \Omega \vdash fun(c : [m : i] \ ). \ e : \langle \texttt{rep} = \tau', [m : \iota]; \ \Phi_e \rangle \rightarrow \gamma[c \leftarrow \tau']}$$

⋆

□

Proposition 2. *For any 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 done by simple induction on the size of $\breve{E}$.
Let $\breve{E}$ be a one-node context. Let A be a type environment and Ω be a collection name environment such that $A\ ;\ \Omega \vdash \breve{E}[\breve{a}_1] : \breve{\tau}$. We show that $A\ ;\ \Omega \vdash \breve{E}[\breve{a}_2] : \breve{\tau}$.

All cases are simple and similar. We show one case for example.

Case $\breve{E}$ is `let` $x = []$ `in` a :

We have:

$$\textsc{Let-ML}\ \frac{A\ ;\ \Omega \vdash E[\breve{a}_1] : \tau_1 \qquad A + x : Gen(\tau_1, A)\ ;\ \Omega \vdash a : \tau_2}{A\ ;\ \Omega \vdash \texttt{let}\ x = E[\breve{a}_1]\ \texttt{in}\ a : \tau_2}$$

By induction hypothesis applied to the first premisse, $A\ ;\ \Omega \vdash E[\breve{a}_2] : \tau_1$. Hence:

$$\textsc{Let-ML}\ \frac{A\ ;\ \Omega \vdash E[\breve{a}_2] : \tau_1 \qquad A + x : Gen(\tau_1, A)\ ;\ \Omega \vdash a : \tau_2}{A\ ;\ \Omega \vdash \texttt{let}\ x = E[\breve{a}_2]\ \texttt{in}\ a : \tau_2}$$

$\star$

$\square$

Lemma 6. *If* $|A + c : \langle \textit{rep} = \tau;\ \Phi\rangle|_\iota = i$ *then* $|\ i[\tau/\textit{In}(c)]\ |_A = \iota$.

Proof. The proof is done by simple induction on $|A + c : \langle \texttt{rep} = \tau;\ \Phi\rangle|_i$. $\square$

Lemma 7 (Variable substitution). *Let A be a type environment well-formed in relation to a collection name environment Ω. Let $\breve{a}_1$ and $\breve{a}_2$ be expressions. We have $A^*\ ;\ \Omega \vdash \breve{a}_1 : \breve{\tau}_1$ and $A + \breve{x} : \forall\alpha_1 \ldots \alpha_n.\breve{\tau}_1\ ;\ \Omega \vdash \breve{a}_2 : \breve{\tau}_2$ such that:*

- *$\alpha_1, \ldots, \alpha_n$ are type variables not free in A*
- *the bind variables in $\breve{a}_2$ are not free in $\breve{a}_1$.*

Then $A\ ;\ \Omega \vdash \breve{a}_2[TS(\breve{a}_1)/\textit{In}(\breve{x})][\breve{a}_1/\breve{x}] : \breve{\tau}_2$ with TS returning the carrier type of $\breve{a}_1$ if it is a collection. Else $[TS(\breve{a}_1)/\textit{In}(\breve{x})]$ must be considerate as a neutral substitution.

Proof. The proof is by induction on $\breve{a}_2$ and by deriving $A + \breve{x} : \forall\alpha_1 \ldots \alpha_n.\breve{\tau}_1\ ;\ \Omega \vdash \breve{a}_2 : \breve{\tau}_2$.
We note $A_{\breve{x}}$ for $A + \breve{x} : \forall\alpha_1 \ldots \alpha_n.\breve{\tau}_1$.

Case $\breve{a}_2$ is x :

There are two cases:

- case $\breve{x}$ is x:
  We have:

$$\frac{\tau \leqslant A_{\breve{x}}(x)}{A_{\breve{x}}\ ;\ \Omega \vdash x : \tau}$$

and

$$x[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] = \breve{a}_1$$

With the premise we have $\tau \leqslant \forall\alpha_1 \ldots \alpha_n.\breve{\tau}_1$. Thus, there is a substitution θ, compatible with Ω, on the α_i such that $\tau = \theta(\breve{\tau}_1)$.

By the property 2 applied on $A^* \; ; \; \Omega \vdash \breve{a}_1 : \breve{\tau}_1$, we have $\theta(A)^* \; ; \; \Omega \vdash \breve{a}_1 : \theta(\breve{\tau}_1)$. The α_i variables are not free in A by hypothesis, then $A^* \; ; \; \Omega \vdash \breve{a}_1 : \theta(\breve{\tau}_1)$. That is $A^* \; ; \; \Omega \vdash \breve{a}_1 : \tau$.
By the property 3, since A is well-formed in relation to Ω , applied on $A^* \; ; \; \Omega \vdash \breve{a}_1 : \tau$ we have $A \; ; \; \Omega \vdash \breve{a}_1 : \tau$ by extending A^* with $\mathtt{self} : \tau_{col}$.
Therefore we have:

$$A \; ; \; \Omega \vdash x[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \tau$$

– case $\breve{x}$ is not x:

Thus we have:

$$x[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] = x$$

Hence by hypothesis we have:

$$A_{\breve{x}} \; ; \; \Omega \vdash x[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \breve{\tau}_2$$

Therefore, by the property 3 we have:

$$A \; ; \; \Omega \vdash x[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \breve{\tau}_2$$

⋆

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

We have:

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

If there are α_i in the types τ and Φ_d, they can be renamed with β_i not free in A and distinct of α_i thanks to the following substution $\theta = [\alpha_i \leftarrow \beta_i]$.
If the α_i variables doesn't occur in τ and Φ_d, then the identity is taken for θ.

By application of the property 2 on the premises, we obtain:

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

Since α_i and β_i are not free in A we have:

$$\frac{A_{\breve{x}}\ ;\ \Omega \vdash e : \mathtt{sig}\ (\ \langle \mathtt{rep} = \theta(\tau'); \theta(\Phi_d)\rangle\ )\ (\ \mathtt{rep} = \theta(\tau'); \theta(\Phi_d)\ )\ \mathtt{end}\ (\mathbf{1}) \qquad A_{\breve{x}} + c : \langle \mathtt{rep} = c; \theta(\Phi_d)\rangle\ ;\ (\Omega;\ c) \vdash a : \theta(\ \tau\ )(\mathbf{2})}{A_{\breve{x}}\ ;\ \Omega \vdash \mathtt{collection}\ c = e\ \ \mathtt{in}\ \ a : \theta(\tau)}$$

By induction hypothesis on e of the premise ($\mathbf{1}$), we have:

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

We note that $\breve{x}$ cannot be c, because c is fresh in relation to Ω. We have:

$$\begin{array}{l} (\ \mathtt{collection}\ c = e\ \ \mathtt{in}\ \ a\ )[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] \qquad = \\ \mathtt{collection}\ c = e[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}]\ \ \mathtt{in}\ \ a[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] \end{array}$$

To apply the induction hypothesis on the expression a of the premise ($\mathbf{2}$), the hypothesis $A^*\ ;\ \Omega \vdash \breve{a}_1 : \breve{\tau}_1$ must be extended with $A^* + c : \langle \mathtt{rep} = c;\ \theta(\Phi_d)\rangle\ ;\ (\Omega;\ c) \vdash \breve{a}_1 : \breve{\tau}_1$. This extension is valid. Indeed, since c is fresh in relation to Ω, the judgment $A^*\ ;\ \Omega \vdash \breve{a}_1 : \breve{\tau}_1$ can be extended with $A^*\ ;\ (\Omega;\ c) \vdash \breve{a}_1 : \breve{\tau}_1$. Then we obtain the final extention by applying the property 3 on $A^*\ ;\ (\Omega;\ c) \vdash \breve{a}_1 : \breve{\tau}_1$. This is possible since $A^* + c : \langle \mathtt{rep} = c;\ \theta(\Phi_d)\rangle$ and A^* are well-formed in relation to $(\Omega; c)$.
Thus by induction hypothesis on the expression a of the presmise ($\mathbf{2}$) we have:

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

Hence we obtain the following result:

$$\frac{A\ ;\ \Omega \vdash e[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \mathtt{sig}\ (\ \langle \mathtt{rep} = \theta(\tau'); \theta(\Phi_d)\rangle\ )\ (\ \mathtt{rep} = \theta(\tau'); \theta(\Phi_d)\ )\ \mathtt{end} \qquad A + c : \langle \mathtt{rep} = c; \theta(\Phi_d)\rangle\ ;\ (\Omega;\ c) \vdash a[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \theta(\tau)}{A\ ;\ \Omega \vdash \mathtt{collection}\ c = e[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}]\ \ \mathtt{in}\ \ a[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \theta(\tau)}$$

By inverse renoming, therefore we have:

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

that is:

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

⋆

Case $\breve{a}_2$ is $fun(c :\ [m : i]\ ).\ e$:

We have:

$$\textsc{Species Fun}\quad \frac{A_{\breve{x}} + c : \langle \mathtt{rep} = c;\ [m : \iota]\rangle\ ;\ (\Omega;\ c) \vdash e : \gamma \qquad \text{where } |A_{\breve{x}}|_\iota = i}{A_{\breve{x}}\ ;\ \Omega \vdash fun(c : [m : i]\ ).\ e : \langle \mathtt{rep} = \tau', [m : \iota];\ \Phi_e\rangle {\rightarrow} \gamma[c \leftarrow \tau']}$$

Since c is fresh in relation to Ω, $\breve{x}$ can't be c. Thus we have:

$$\begin{array}{ll} (\ fun(c : [m : i]\ ).\ e\ )[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] & = \\ fun(c : [m : i\ [TS(\breve{a}_1)/\mathtt{In}(\breve{x})]\ ]\ ).\ e[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] & \end{array}$$

If there are α_i in the type $\langle \mathtt{rep} = \tau', [m : \iota];\ \Phi_e\rangle \rightarrow \gamma[c \leftarrow \tau']$, they can be renamed with β_i not free in A and distinct of α_i thanks to following substitution $\theta = [\alpha_i \leftarrow \beta_i]$. If the α_i variables don't occur in $\langle \mathtt{rep} = \tau', [m : \iota];\ \Phi_e\rangle \rightarrow \gamma[c \leftarrow \tau']$, then the identity is taken for θ.

By the property 2 on the premise we have:

$$\theta(A_{\breve{x}} + c : \langle \mathtt{rep} = c;\ [m : \iota]\rangle)\ ;\ (\Omega;\ c) \vdash e : \theta(\gamma)$$

That is, since α_i and β_i are not free in A:

$$A_{\breve{x}} + c : \langle \mathtt{rep} = c;\ [m : \theta(\iota)]\rangle\ ;\ (\Omega;\ c) \vdash e : \theta(\gamma)$$

In oder to apply the induction on the above judgment, we must extend the hypothesis $A^*\ ;\ \Omega \vdash \breve{a}_1 : \breve{\tau}_1$ by $A^* + c : \langle \mathtt{rep} = c;\ [m : \theta(\iota)]\rangle\ ;\ (\Omega;\ c) \vdash: \breve{\tau}_1$.This extension is valid. Indeed, since c is fresh in relation to Ω, the judgment $A^*\ ;\ \Omega \vdash \breve{a}_1 : \breve{\tau}_1$ can be extended with $A^*\ ;\ (\Omega;\ c) \vdash \breve{a}_1 : \breve{\tau}_1$. Then we obtain the final extention by applying the property 3 on $A^*\ ;\ (\Omega;\ c) \vdash \breve{a}_1 : \breve{\tau}_1$. This is possible since $A^* + c : \langle \mathtt{rep} = c;\ [m : \theta(\iota)]\rangle$ and A^* are well-formed in relation to $(\Omega; c)$.
Thus by induction hypothesis on the expression a of the presmise (**2**) we have:

$$A + c : \langle \mathtt{rep} = c;\ [m : \theta(\iota)]\rangle\ ;\ (\Omega;\ c) \vdash e[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \theta(\gamma)$$

Then, by application of the lemma 4 on $|A_{\breve{x}}|_\iota = i$ we have:

$$|\theta(A_{\breve{x}})|_{\theta(\iota)} = i$$

That is, since α_i and β_i are not free in A:

$$|A_{\breve{x}}|_{\theta(\iota)} = i$$

And by application of the lemma 6 application, we obtain:

$$|A|_{\theta(\iota)} = i[TS(\breve{a}_1)/\mathtt{In}(\breve{x})]$$

Thus, we obtain:

$$\textsc{Species Fun}\ \dfrac{\begin{array}{c} A + c : \langle \mathtt{rep} = c;\ [m : \theta(\iota)]\rangle\ ;\ (\Omega;\ c) \vdash e[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \theta(\gamma) \\ \text{where } |A|_{\theta(\iota)} = i[TS(\breve{a}_1)/\mathtt{In}(\breve{x})] \end{array}}{\begin{array}{c} A\ ;\ \Omega \vdash fun(c : [m : i\ [TS(\breve{a}_1)/\mathtt{In}(\breve{x})]\ ]\ ).\ e[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \\ \langle \mathtt{rep} = \theta(\tau'), [m : \theta(\iota)];\ \Phi_e\rangle \rightarrow \theta(\gamma)[c \leftarrow \theta(\tau')] \end{array}}$$

Therefore, by inverse renaming we obtain:

$$A\ ;\ \Omega \vdash (\ fun(c : [m : i]\ ).\ e\ )[TS(\breve{a}_1)/\mathtt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \langle \mathtt{rep} = \tau', [m : \iota];\ \Phi_e\rangle \rightarrow \gamma[c \leftarrow \tau']$$

⋆

Case $\breve{a}_2$ is $m : i = a$:

We have:

$$\dfrac{\begin{array}{c}\text{METHOD}\\ A_{\breve{x}}\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \tau;\ m : \iota;\ \Phi_d \rangle\ _{(\mathbf{1})} \\ A_{\breve{x}}\ ;\ \Omega \vdash a : \iota[\texttt{rep} \leftarrow \tau]\ _{(\mathbf{2})} \qquad \text{where } |A_{\breve{x}}|_\iota = i\end{array}}{A_{\breve{x}}\ ;\ \Omega \vdash m : i = a : (m : \iota)}$$

By the property 3 on the premisse ($\mathbf{1}$) we have:

$$A\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \tau;\ m : \iota;\ \Phi_d \rangle$$

By induction hypothesis on the premise ($\mathbf{2}$):

$$A\ ;\ \Omega \vdash a[TS(\breve{a}_1)/\texttt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \iota[\texttt{rep} \leftarrow \tau]$$

And by application of the lemma 6 on where $|A_{\breve{x}}|_\iota = i$, we have:

$$|A|_\iota = i[TS(\breve{a}_1)/\texttt{In}(\breve{x})]$$

Therefore we have:

$$\dfrac{\begin{array}{c}\text{METHOD}\\ A\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep} = \tau;\ m : \iota;\ \Phi_d \rangle \\ A\ ;\ \Omega \vdash a[TS(\breve{a}_1)/\texttt{In}(\breve{x})][\breve{a}_1/\breve{x}] : \iota[\texttt{rep} \leftarrow \tau] \qquad \text{where } |A|_\iota = i[TS(\breve{a}_1)/\texttt{In}(\breve{x})]\end{array}}{A\ ;\ \Omega \vdash m : i[TS(\breve{a}_1)/\texttt{In}(\breve{x})] = a[TS(\breve{a}_1)/\texttt{In}(\breve{x})][\breve{a}_1/\breve{x}] : (m : \iota)}$$

Thus we have:

$$A\ ;\ \Omega \vdash (\ m : i = a\ )[TS(\breve{a}_1)/\texttt{In}(\breve{x})][\breve{a}_1/\breve{x}] : (m : \iota)$$

⋆

Case $\breve{a}_2$ is $\texttt{rep} = t$:

We have:

$$\dfrac{\begin{array}{c}\text{CARRIER TYPE}\\ A_{\breve{x}}\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep}\ = |A_{\breve{x}}|_t;\ \Phi_d \rangle\end{array}}{A_{\breve{x}}\ ;\ \Omega \vdash \texttt{rep} = t : (\texttt{rep} = |A_{\breve{x}}|_t)}$$

By the property 3 on the premise, we have:

$$A\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep}\ = |A_{\breve{x}}|_t;\ \Phi_d \rangle$$

By the lemma 6 applied on $|A_{\breve{x}}|_t$, we have:

$$|A_{\breve{x}}|_t = |A|_{t[TS(\breve{a}_1)/\texttt{In}(\breve{x})]}$$

Thus, we obtain:

$$\dfrac{A\ ;\ \Omega \vdash \texttt{self} : \langle \texttt{rep}\ = |A|_{t[TS(\breve{a}_1)/\texttt{In}(\breve{x})]};\ \Phi_d\rangle}{A\ ;\ \Omega \vdash \texttt{rep} = t[TS(\breve{a}_1)/\texttt{In}(\breve{x})] : (\texttt{rep} = |A|_{t[TS(\breve{a}_1)/\texttt{In}(\breve{x})]})}\ \textsc{Carrier type}$$

Hence:

$$A\ ;\ \Omega \vdash \texttt{rep} = t[TS(\breve{a}_1)/\texttt{In}(\breve{x})] : (\texttt{rep} = |A_{\breve{x}}|_t)$$

And we have:

$$\begin{array}{l}(\texttt{rep} = t)[TS(\breve{a}_1)/\texttt{In}(\breve{x})][\breve{a}_1/\breve{x}] = \\ \texttt{rep} = t[TS(\breve{a}_1)/\texttt{In}(\breve{x})]\end{array}$$

Therefore we have:

$$A\ ;\ \Omega \vdash (\texttt{rep} = t)[TS(\breve{a}_1)/\texttt{In}(\breve{x})][\breve{a}_1/\breve{x}] : (\texttt{rep} = |A_{\breve{x}}|_t)$$

⋆

□

Lemma 8 (Concatenation of field lists). *Let A be a type environment and Ω a collection name environment. Le w_1 and w_2 two field lists such as $A\ ;\ \Omega \vdash w_1 : \Phi_1$ and $A\ ;\ \Omega \vdash w_2 : \Phi_2$.*
If Φ_1 and Φ_2 are compatible, then $A\ ;\ \Omega \vdash w_1 @ w_2 : \Phi_1 \oplus \Phi_2$.

Proof. The proof is done simply by induction on w_1.

□

Lemma 9. *Let $\breve{a}$ be col, a or e expressions. Let w be a field list expression. Let A be a type environment and supposed starry. Then we have $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash w : \Phi_c$ and $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$ such that the bind variables of $\breve{a}$ are not free in $\langle w \rangle$. Then $A\ ;\ \Omega \vdash \breve{a}[\langle w \rangle/\texttt{self}] : \breve{\tau}] : \breve{\tau}$*

Proof. The proof is done easily by induction on $\breve{a}$ and by deriving $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash \breve{a} : \breve{\tau}$.
We note E_s for $E + \texttt{self} : \langle \Phi_c \rangle$.

Case $\breve{a}$ is $\texttt{collection}\ c = e\ \texttt{in}\ a$:

We have:

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

By induction hypothesis applied on e of the premise $(\mathbf{1})$, we have:

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

We extend the hypothesis $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash w : \Phi_c$ by $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ (\Omega;\ c) \vdash w : \Phi_c$ since c is fresh in relation to Ω. Then, $A + c : \langle \texttt{rep} = c; \Phi_d \rangle + \texttt{self} : \langle \Phi_c \rangle$ is well formed in relation to $(\Omega;\ c)$ since $A + \texttt{self} : \langle \Phi_c \rangle$ is well formed in relation to Ω and c is fresh in relation to Ω. Hence by the property 3 applied on $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ (\Omega;\ c) \vdash w : \Phi_c$, we have $A + c : \langle \texttt{rep} = c; \Phi_d \rangle + \texttt{self} : \langle \Phi_c \rangle\ ;\ (\Omega;\ c) \vdash w : \Phi_c$.
Thus by induction hypothesis applied on a of the premise (**2**) we have:

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

Thus we have:

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

Therefore:

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

$\star$

Case $\breve{a}$ is $fun(c :\ [m : i]\ ).\ a$:

We have:

$$\textsc{Species Fun}\ \frac{A_s + c : \langle \texttt{rep} = c;\ [m : \iota] \rangle\ ;\ (\Omega;\ c) \vdash e : \gamma\ _{(\mathbf{1})} \qquad \text{where } |A_s|_\iota = i}{A_s\ ;\ \Omega \vdash fun(c : [m : i]\ ).\ e : \langle \texttt{rep} = \tau', [m : \iota];\ \Phi_e \rangle \rightarrow \gamma[c \leftarrow \tau']}$$

We extend the hypothesis $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ \Omega \vdash w : \Phi_c$ by $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ (\Omega;\ c) \vdash w : \Phi_c$ since c is fresh in relation to Ω. Then, $A + c : \langle \texttt{rep} = c; [m : \iota] \rangle + \texttt{self} : \langle \Phi_c \rangle$ is well formed in relation to $(\Omega;\ c)$ since $A + \texttt{self} : \langle \Phi_c \rangle$ is well formed in relation to Ω and c is fresh in relation to Ω. Hence by the property 3 applied on $A + \texttt{self} : \langle \Phi_c \rangle\ ;\ (\Omega;\ c) \vdash w : \Phi_c$, we have $A + c : \langle \texttt{rep} = c; [m : \iota] \rangle + \texttt{self} : \langle \Phi_c \rangle\ ;\ (\Omega;\ c) \vdash w : \Phi_c$.
Thus by induction hypothesis applied on e of the premise (**1**) we have:

$$A + c : \langle \texttt{rep} = c; [m : \iota] \rangle\ ;\ (\Omega;\ c) \vdash e[\langle w \rangle/\texttt{self}] : \gamma$$

And by the lemma 5, applied on $|A_s|_\iota = i$, we have $|i|_A = \iota$.
Thus we have:

$$\textsc{Species Fun}\ \frac{A + c : \langle \texttt{rep} = c;\ [m : \iota] \rangle\ ;\ (\Omega;\ c) \vdash e[\langle w \rangle/\texttt{self}] : \gamma \qquad \text{where } |i|_A = \iota}{A\ ;\ \Omega \vdash fun(c : [m : i]\ ).\ e[\langle w \rangle/\texttt{self}] : \langle \texttt{rep} = \tau', [m : \iota];\ \Phi_e \rangle \rightarrow \gamma[c \leftarrow \tau']}$$

Therefore we have:

$$A\ ;\ \Omega \vdash (\ fun(c : [m : i]\ ).\ e\ )[\langle w \rangle/\texttt{self}] : \langle \texttt{rep} = \tau', x[m : \iota];\ \Phi_e \rangle \rightarrow \gamma[c \leftarrow \tau']$$

$\star$

$\square$

Lemma 10. *If $\breve{a}_1 \rightarrow_\epsilon \breve{a}_2$, then $\breve{a}_1 \subset \breve{a}_2$.*

Proof. The proof is don e indepently for each redex. All cases are easy now that we have proven the right lemmas.
Let assume $A\ ;\ \Omega \vdash \breve{a}_1 : \breve{\tau}$ and A is starry in relation to the context of the $\subset$ relation.

Case $\breve{a}_1$ is $(fun(x).\ a)\ v$:

A derivation for $\breve{a}_1$ is:

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

By the lemma 7 applied on the premises (**1**) and (**2**), we have:

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

$\star$

Case $\breve{a}_1$ is $\texttt{let}\ x = v\ \texttt{in}\ a$:

A derivation for $\breve{a}_1$ is:

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

By the lemma 7 applied on the premises (**1**) and (**2**), we have:

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

$\star$

Case $\breve{a}_1$ is $\langle v_w \rangle !m$:

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

A derivation for $\breve{a}_1$ is:

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

By applying the lemma 9 on the premises (**1**) and (**2**) we obtain:

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

Hence:

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

⋆

Case $\breve{a}_1$ is $\texttt{collection}\ c = \texttt{struct}\ v_w\ \texttt{end in}\ a$:

A derivation for $\breve{a}_1$ is:

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

By applying the rule Executive collection on the premise (**1**) we have:

$$\dfrac{A^* + \texttt{self} : \langle \texttt{rep} = \tau; \Phi_d \rangle \; ; \; \Omega \vdash v_w : (\texttt{rep} = \tau; \Phi_d)}{A \; ; \; \Omega \vdash \langle v_w \rangle : \langle \texttt{rep} = \tau; \Phi_d \rangle\ _{(\mathbf{1'})}}$$

The jugdment of the premise (**1**) is well formed. Thus, all occurences of collection name used in the carrier type τ is declared according to Ω. As c is fresh in relation to Ω, the type τ doesn't contain occurences of c. Then by the property 1 applied on the premise (**2**) we have:

$$(A + c : \langle \texttt{rep} = c; \Phi_d \rangle \,)[c \leftarrow \tau] \; ; \; \Omega \vdash a : \tau'[c \leftarrow \tau]$$

Since the jugdment $A \; ; \; \Omega \vdash \texttt{collection}\ c = \texttt{struct}\ v_w\ \texttt{end in}\ a : \tau'$ is well formed, the type environment and the type τ' don't contain occurences of collection name c.
Thus from the previous jugdment we obtain the judgment (**2'**):

$$A + c : \langle \texttt{rep} = \tau; \Phi_d \rangle \; ; \; \Omega \vdash a : \tau'$$

By the lemma 7 applied on the jugments (**1'**) and (**2'**) we obtain the conclusion:

$$A \; ; \; \Omega \vdash a[CT(\langle v_w \rangle)/\texttt{In}(c)][\langle v_w \rangle / c] : \tau$$

⋆

Case $\breve{a}_1$ is $\texttt{species}\ z = v_e\ \texttt{in}\ a$:

A derivation for $\breve{a}_1$ is:

$$\dfrac{A \; ; \; \Omega \vdash v_e : \gamma\ _{(\mathbf{1})} \qquad A + z : Gen(\gamma, A) \; ; \; \Omega \vdash a : \tau\ _{(\mathbf{2})}}{A \; ; \; \Omega \vdash \texttt{species}\ z = v_e\ \texttt{in}\ a : \gamma}$$

By the lemma 7 applied on the premises (**1**) and (**2**) we obtain the conclusion:

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

⋆

Case $\breve{a}_1$ is $(m : i = a;\ v_w)$ with $m \in dom(v_w)$:

A derivation for $\breve{a}_1$ is:

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

Since $m \in dom(v_w)$, we have $m \in dom(\Phi)$. Then, $(m : \iota)$ and Φ are compatible. Thus $\Phi = (m : \iota) \oplus \Phi$. Therefore, by the premise $(\mathbf{1})$ we have:

$$A\ ;\ \Omega \vdash v_w : (m : \iota) \oplus \Phi$$

⋆

Case $\breve{a}_1$ is $\texttt{rep} = t;\ v_w$ with $\texttt{rep} \in dom(v_w)$:

A derivation for a_1 is:

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

Since $\texttt{rep} \in dom(v_w)$, we have $\texttt{rep} \in dom(\Phi)$. Then, $(\texttt{rep} : \tau)$ and Φ are compatible. Thus $\Phi = (\texttt{rep} : \tau) \oplus \Phi$. Therefore, by the premise $(\mathbf{1})$ we have:

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

⋆

Case $\breve{a}_1$ is $\texttt{inherit (struct}\ v_w\ \texttt{end)};\ w$:

A derivation for $\breve{a}_1$ is:

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

By the premise $(\mathbf{2})$, we have $A = A^* + \texttt{self} : \tau_{col}$. Thus the premise $(\mathbf{1})$ can be rewritten as $A\ ;\ \Omega \vdash v_w : \Phi_1$. As Φ_1 and Φ_2 are compatible (that is, $\Phi_1 \oplus \Phi_2$), the lemma 8 can be apply on the judment $(\mathbf{1})$ and $(\mathbf{3})$. Hence the conclusion:

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

⋆

Case $\breve{a}_1$ is $(fun(c :\ [m : i]\ ).\ e)\ v_{col}$:

A derivation for a_1 is:

$$\dfrac{\dfrac{A + c : \langle \mathtt{rep} = c;\ [m : \iota] \rangle\ ;\ (\Omega;\ c) \vdash e : \gamma\ _{(\mathbf{1})} \qquad \text{where } |i|_A = \iota}{A\ ;\ \Omega \vdash fun(c :\ [m : i]\ ).\ e : (\mathtt{rep} = \tau;\ [m : \iota]) \rightarrow \gamma[c \leftarrow \tau]} \qquad A\ ;\ \Omega \vdash v_{col} : \langle \mathtt{rep} = \tau;\ [m : \iota] \rangle\ _{(\mathbf{2})}}{A\ ;\ \Omega \vdash (fun(c :\ [m : i]\ ).\ e)\ v_{col} : \gamma[c \leftarrow \tau]}$$

⋆

From the jugdment $A\ ;\ \Omega \vdash v_{col} : \langle \mathtt{rep} = \tau;\ [m : \iota] \rangle$, the carrier type τ doesn't contain occurence of c since it is fresh in relation to Ω. Thus by the property 1 applied on the premise ($\mathbf{1}$) we have:

$$(\ A + c : \langle \mathtt{rep} = c;\ [m : \iota] \rangle\ )[c \leftarrow \tau]\ ;\ \Omega \vdash e : \gamma[c \leftarrow \tau]$$

The type environment A is well-formed in relation to Γ and c is fresh in relation to Γ. Thus, c doesn't occur in A. Since we have $|i|_A = \iota$, c doesn't occur also in ι. Thus we obtain the following judgment from the previous one:

$$A + c : \langle \mathtt{rep} = \tau;\ [m : \iota] \rangle\ ;\ \Omega \vdash e : \gamma[c \leftarrow \tau]\ _{(\mathbf{1'})}$$

Now we can apply the lemma 7 on the judgment ($\mathbf{1'}$) and ($\mathbf{2}$) to obtain the conclusion:

$$A\ ;\ \Omega \vdash e[v_{col}/c][CT(v_{col})/\mathtt{In}(c)]$$

□

Lemma 11. *1. Let v be a value. We assume $\varnothing \vdash v : \tau$. If τ is functional type, then v is a function.*

2. Let v_e be a species value. We assume $\varnothing \vdash v_e : \gamma$. If γ is function type, then v_e is parameterized species. Otherwise v_e is a structure.

Proof. 1. If τ is function type, then $\tau = \tau_1 \rightarrow \tau_2$. Since v is a value, in the environment $\varnothing;\ \varnothing$, only the rule FUN-ML can be applied. Then v is a function.

2. Since v_e is a value, in the environment $\varnothing;\ \varnothing$, only the rules SPECIES-FUN and SPECIES BODY can be applied. If γ is a functional type, that is $\gamma = \tau_{col} \rightarrow \gamma'$, then it's only SPECIES-FUN is applied. Then v_e is a parameterized species. Otherwise, if γ is not a functional type, only SPECIES BODY rule can be applied. In this case, v_e is a structure.

□

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

Proof. The proof is done according to the different previous lemmas.

Theorem 2. *Well-typed irreductible normal forms are values (i.e. if $\varnothing \vdash \breve{a} : \breve{\tau}$ and $\breve{a}$ cannot be reduced, then $\breve{a}$ is a value).*

Proof. The proof is done by simultaneous induction on the size of different form of $\breve{a}$. We assume $\varnothing \vdash a : \tau$ (respectively $\varnothing \vdash e : \gamma$ and $A;\ \varnothing \vdash w : \Phi$ with E just containing `self` necessary for the fields w).

Case $\breve{a}$ is cst :

By definition, cst is a value.

⋆

Case $\breve{a}$ is x :

x cannot be typed in the empty environmment.

⋆

Case $\breve{a}$ is $fun(x).\ a$:

By definition, $fun(x).\ a$ is a value.

⋆

Case $\breve{a}$ is $a_1\ a_2$:

A derivation for $\breve{a}$ is:

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

The induction hypothesis applied to expression a_1 shows that it is a value. From the previous derivation, the type $\tau_1 \rightarrow \tau_2$ of a_1 is functional. Thus by the lemma 11, a_1 must be a function $fun(x).\ a_1'$. Then the expression $\breve{a}$ can be reduced. It's contradictory.

⋆

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

The induction hypothesis applied to expressions a_1 and a_2 shows that they are values. Then (a_1, a_2) is a value by definition.

⋆

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

The induction hypothesis applied to expression a_1 shows that it is a value. Then the expression $\breve{a}$ can be reduced. It's contradictory.

⋆

Case $\breve{a}$ is $col!m$:

The induction hypothesis applied to expression *col* shows that it is a value. Then the expression $\breve{a}$ can be reduced. It's contradictory.

$\star$

Case $\breve{a}$ is $\texttt{collection}\ c = e\ \texttt{in}\ a$:

A derivation for $\breve{a}$ is:

$$\frac{\varnothing \vdash e : \texttt{sig}\ (\langle \texttt{rep} = \tau;\ \Phi_d \rangle)\ (\texttt{rep} = \tau;\ \Phi_d)\ \texttt{end} \qquad c : \langle \texttt{rep} = c;\ \Phi_d \rangle;\ c \vdash a : \tau}{\varnothing \vdash \texttt{collection}\ c = e\ \texttt{in}\ a : \tau}$$

The induction hypothesis applied to species expression e shows that it is a value. From the previous derivation, the type $\texttt{sig}\ (\langle \texttt{rep} = \tau;\ \Phi_d \rangle)\ (\texttt{rep} = \tau;\ \Phi_d)\ \texttt{end}$ of e is the one for a structure. Thus by the lemma 11, e must be a structure $\texttt{struct}\ v_w\ \texttt{end}$. Then the expression $\breve{a}$ can be reduced. It's contradictory.

$\star$

Case $\breve{a}$ is $\texttt{species}\ z = e\ \texttt{in}\ a$:

The induction hypothesis applied to species expression e shows that it is a value. Then the expression $\breve{a}$ can be reduced. It's contradictory.

$\star$

Case $\breve{a}$ is c :

c cannot be typed in the empty environmment.

$\star$

Case $\breve{a}$ is `self` :

`self` cannot be typed in the empty environmment.

$\star$

Case $\breve{a}$ is $\langle w \rangle$:

The induction hypothesis applied to species expression $\langle w \rangle$ shows that it is a value. Thus by definition, $\langle w \rangle$ is a value.

$\star$

Case $\breve{a}$ is z :

z cannot be typed in the empty environmment.

$\star$

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

The induction hypothesis applied to species expression `struct` w `end` shows that it is a value. Thus by definition, `struct` w `end` is a value.

$\star$

Case $\breve{a}$ is $fun(c:\ [m:i]\ ).\ e$:

By definition $fun(c:\ [m:i]\ ).\ e$ is a value.

$\star$

Case $\breve{a}$ is $e\ col$:

A derivation for $\breve{a}$ is:

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

The induction hypothesis applied to expression e shows that it is a value. From the previous derivation, the type $\tau_{col} \rightarrow \gamma$ of e is functional. Thus by the lemma 11, e must be a a parameterized species $fun(c:\ [m:i]\ ).\ e'$. Then the expression $\breve{a}$ can be reduced. It's contradictory.

$\star$

Case $\breve{a}$ is $\varnothing$:

By definition, $\varnothing$ is a value.

$\star$

Case $\breve{a}$ is $d;\ w$:

By induction d and w are values. d is a method or `rep` field not overloaded by w. In this case, $\breve{a}$ is a value by definition. Otherwise, $\breve{a}$ can be reduced. And in this case, it's contradictory.

$\star$

Case $\breve{a}$ is $m : i = a$:

By definition, $\breve{a}$ is a value.

$\star$

Case $\breve{a}$ is `rep` $= t$:

By definition, $\breve{a}$ is a value.

$\star$

Case $\breve{a}$ is `inherit` e :

A derivation for $\breve{a}$ is:

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

The induction hypothesis applied to species expression e shows that it is a value. From the previous derivation, the type $\mathtt{sig}\ (\tau_{col})\ \Phi\ \mathtt{end}$ of e is the one for a structure. Thus by the lemma 11, e must be a structure $\mathtt{struct}\ \ v_w\ \ \mathtt{end}$. Then the expression $\breve{a}$ can be reduced. It's contradictory.

$\star$

References

1. Boulm, S., Doligez, D., Dubois, C., Fechter, S., Hardin, T., Jaume, M., Maarek, M., Ménissier-Morain, V., Pons, O., Prevosto, V., Rioboo, R., Donzeau-Gouge, V.V.: The Foc project. (2003) http://www-spi.lip6.fr/~foc.
2. Prevosto, V., Doligez, D.: Inheritance of algorithms and proofs in the computer algebra library foc. Journal of Automated Reasoning **29** (2002) 337–363 Special Issue on Mechanising and Automating Mathematics, In Honor of N.G. de Bruijn.
3. Boulmé, S., Hardin, T., Rioboo, R.: Some hints for polynomials in the Foc project. In: Calculemus 2001 Proceedings. (2001)
4. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system release 3.02 Documentation and user's manual. INRIA. (2001) http://pauillac.inria.fr/ocaml/htmlman/.
5. Leroy, X.: Manifest types, modules, and separate compilation. In: 21st symposium Principles of Programming Languages, ACM Press (1994) 109–122
6. Hirschowitz, T., Leroy, X.: Mixin modules in a call-by-value setting. In: European Symposium on Programming. (2002) 6–20
7. Fechter, S.: Une sémantique pour FoC. Rapport de D.E.A., Université Paris 6 (2001) avaible at http://www-spi.lip6.fr/~fechter.
8. Fechter, S.: An object-oriented model for the certified computer algebra library. Paper presented at FMOODS 2002 PhD workshop (2002) http://www-spi.lip6.fr/~fechter.
9. Rémy, D., Vouillon, J.: Objective ML: An effective object-oriented extension to ML. Theory and Practice of Object Systems **4** (1998) p. 27–50
10. Barras, B., Boutin, S., Cornes, C., Courant, J., Coscoy, Y., Delahaye, D., de Rauglaudre, D., Filliâtre, J.C., Giménez, E., Herbelin, H., Huet, G., Laulhère, H., Munoz, C., Murthy, C., Parent-Vigouroux, C., Loiseleur, P., Paulin-Mohring, C., Saïbi, A., Werner, B.: The Coq Proof-assistant reference manual. INRIA. 6.3.1 edn. (1999) http://pauillac.inria.fr/coq/doc/main.html.
11. Boite, O., Fechter, S.: BBFoC. draft available at http://www-spi.lip6.fr/~fechter (2002)
12. Prevosto, V.: Conception et implantation du langage Foc pour le développement de logiciels certifiés. PhD thesis, Université Paris VI (2003)