Developing Structured Libraries using the Focal Environment

Nicolas Bertaux
CEDRIC/CNAM
292 rue Saint-Martin
75141 Paris Cedex 03, France
Nicolas.Bertaux@lri.fr

David Delahaye
CEDRIC/CNAM
292 rue Saint-Martin
75141 Paris Cedex 03, France
David.Delahaye@cnam.fr

ABSTRACT

We introduce the Focal environment, which is an integrated development environment, offering functional and object-oriented features, and designed to build certified components using theorem proving. In Focal, inheritance provides a suitable notion of refinement, allowing us to go step by step (in an incremental approach) from abstract specifications to concrete implementations while proving that these implementations meet their specifications or design requirements. In addition, inheritance and parameterization offer a high level of reusability. To highlight these features, we present a survey of Focal, with a complete example of formalization in support. Finally, Focal is equipped with a compiler producing OCaml code for execution and Coq code for certification, and we also propose a compilation scheme based on modules, which is supposed to be an alternative to the current scheme using records and aims to provide a higher level view of compiled specifications supplying in particular traceability. This compilation scheme is not only described through an example, but also formally.

Categories and Subject Descriptors

F.3 [**Theory of Computation**]: Logics and Meanings of Programs; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of Programs, Specification Techniques*

General Terms

Languages, Theory, Verification

Keywords

Focal, Objects, Modules, OCaml, Coq

1. INTRODUCTION

Over the last few years, digital libraries have been the subject of a recrudescence of interest, probably due to the possibility of accessing Internet for the great majority of people.

Actually, digital libraries had been introduced more than thirty years ago, with the Project Gutenberg [14] founded in 1971 and which aimed to archive cultural works. This craze for digital libraries can be explained by the numerous advantages they offer, among many others: collections of materials are dematerialized, which saves space and preserves physical collections; the resources can be simultaneously accessed by several users at any time at any place, as long as a network connection is available; information retrieval is possible through the entire digital library using complex patterns of search. There exists a certain number of digital libraries; some of them propose to archive books like the Million Book Project [3], some others offer a wide panel of cultural materials (literature, painting, music, etc) like the World Digital Library, or even an archive of the Web like the Internet Archive [15]. In Computer Science, we are in a quite similar situation (except that materials are already dematerialized), where we intend to design libraries of programs or proofs coming from different languages or formalisms, with the possibility of making information available and especially reusable in different contexts. As examples, we can hold up OpenMath [20], which is an extensible standard for representing the semantics of mathematical objects, or the MoWGLI project [19], which aims to manage and publish mathematical documents. This concern of building appropriate libraries is confirmed by the creation of specific meetings or interest groups, e.g. the DML workshops [9] discuss the means of designing a global mathematical digital library, while the MKM interest group [17] deals with all aspects of mathematical knowledge management. Thus, these days, more and more computer libraries appear, sometimes quite large (for instance, Google Book Search [5] reached the 7 million books in November 2008), and some questions related to practical problems arise. Among these questions, there are the structure of the stored information, the maintainability and the evolution of the library, the reusability of information, and the information retrieval.

In this paper, we plan to address some of the previous issues in the specific framework of libraries of programs and proofs. To do so, we introduce in particular the Focal environment [1, 12], which is an integrated development environment, offering functional and object-oriented features, and designed to build certified components using theorem proving. More precisely, Focal, initiated by T. Hardin and R. Rioboo with S. Boulmé, provides a language in which it is possible to build certified applications step by step (in an incremental approach), going from abstract specifications, called species, to concrete implementations, called collections. These different structures are combined using inheritance and parameterization, inspired by object-oriented programming and which can be seen as a notion of refinement. Moreover, each of these structures is equipped with a carrier set, called representation, providing a typical algebraic specification flavor. V. Prevosto developed a compiler [7] for this language, able to produce OCaml code [10] for execution, Coq code [11] for certification, and code for documentation. D. Doligez also provided a first-order automated theorem prover, called Zenon [2], which helps the user to complete his/her proofs and can directly produce Coq proofs.

Focal is a tool which allows us to develop libraries with a high degree of structure and can be seen as a front-end for more low-level target languages, in particular OCaml and Coq. Basically, this high degree of structure is ensured by the use of parameterizable structures, i.e. species, which can contain both abstract and concrete code, and which can be refined by inheritance to obtain implementations, i.e. collections. Thus, the Focal language defines a methodology of development, which quite accords with the notion of software release life cycle; therefore, Focal appears as a good candidate for the development of safe and/or secure systems, since it allows us to ease the conformance evaluation process w.r.t. different evaluation systems imposed by various standards, like EAL for CC [18] or SIL for IEC 61508 [16]. As a consequence, Focal has been successfully used in several significant applications, such as Computer Algebra [8], airport security regulations [4], or security policies [6]. All these formalizations tend to show that Focal can be considered as a general-purpose specification language, appropriate not only for mathematics but also for real-world applications. To highlight these aspects of Focal, we provide a survey of this environment in this paper, with a complete example of formalization and which basically allows us to reveal the rationale behind this environment. To still underline the high level of structure of Focal specifications, this paper is also focused on the way of building a compilation scheme for Focal which would be of high-level as well. To do so, we present a model of Focal relying on modules, which is supposed to be an alternative to the actual compiler using records [7], and which can be applied both to OCaml and Coq since these two languages offer a module system. As modules are higher level structures than records, such a compilation allows us to preserve, at a certain extent, the structure of Focal specifications in the compiled code and then provides traceability w.r.t. these specifications, which is not possible with the model based on records where the notion of inheritance disappears and where specifications must be flattened.

This paper is organized in two parts as follows: first, we draw an outline of the Focal environment (Section 2), providing a complete example of formalization in particular; second, we describe a compilation scheme from Focal to OCaml and Coq using modules (Section 3), not only by means of an example but also formally.

2. THE FOCAL ENVIRONMENT

In this section, we give an overview of the Focal environment. In particular, we describe the syntax of the specification language and some ideas regarding the underlying semantics. We also provide a complete example of formalization, which will be used in Section 3 when presenting the compilation schemes informally.

2.1 Specification: Species

The first major notion of the Focal language [1, 12] is the structure of *species*, which corresponds to the highest level of abstraction in a specification. A species can roughly be seen as a list of attributes of three kinds: the carrier type, called *representation*, which is the type of the entities that are manipulated by the functions of the species, and which can be either abstract or concrete; the functions, which denote the operations allowed on the entities, and which can be either *declarations* (when only a type is given) or *definitions* (when a body is also provided); the properties, which must be verified by any further implementation of the species, and which can be either simply *properties* (when only the proposition is given) or *theorems* (when a proof is also provided).

The syntax of a species is the following:

```
species <name> =
  rep [= <type>];                 (* representation *)
  sig <name> in <type>;           (* declaration *)
  let <name> = <body>;            (* definition *)
  property <name> : <prop>;       (* property *)
  theorem <name> : <prop>         (* theorem *)
  proof : <proof>;
end
```

where <*name*> is simply a given name, <*type*> a type expression (mainly typing of core-ML without polymorphism but with concrete data types), <*body*> a function body (mainly core-ML with conditional, pattern-matching and recursion), <*prop*> a (first-order) proposition and <*proof*> a proof (expressed by means of a declarative proof language). In the type language, the specific expression "self" refers to the type of the representation and may be used everywhere except when defining a concrete representation. In addition, functions or properties of species or collections are referenced using the "!" prefix, while top-level functions or properties must be used with the "#" prefix.

As said previously, species can be combined using (multiple) inheritance, which works as expected. It is possible to define functions that were previously only declared or to prove properties which had no provided proof. It is also possible to redefine functions previously defined or to reprove properties already proved. However, the representation cannot be redefined and functions as well as properties must keep their respective types and propositions all along the inheritance path. Another way of combining species is to use parameterization. Species can be parameterized either by other species or by entities from species. If the parameter is a species, the parameterized species only has access to the interface of this species, i.e. only its abstract representation, its declarations and its properties. These two features complete the previous syntax as follows:

```
species <name> (<name> is <name>[(<pars>)],
                <name> in <name>, ...)
        inherits <name>, <name> (<pars>), ... =
        ...
end
```

where <*pars*> is a list of <*name*>, which denotes the names used as effective parameters. When the parameter is a species parameter declaration, the "is" keyword is used. When it is an entity parameter declaration, the "in" keyword is used.

To better understand the notion of species, let us give a small example. The selected example concerns the quite standard implementation of stacks. In Focal, every specification starts with the following predefined root species *basic_object*, which provides an abstract representation in particular:

```
species basic_object =
  rep;
  let print (x in self) = "<abst>";
  let parse (x in string) in self =
    #foc_error ("not␣parsable");
end
```

where *#foc_error* is the operator to signal exceptions.

Before giving the specification of stacks, we propose to consider another predefined species allowing us to introduce the notion of equality and which inherits from species *basic_object*. This species is called *setoid* (this is typically a non-empty set supplied with a decidable equality):

```
species setoid inherits basic_object =
  sig equal in self → self → bool;
  sig element in self;
  let different (x, y) = #not_b (!equal (x, y));
  property equal_reflexive : all x in self,
    !equal(x, x);
  theorem same_is_not_different :
    all x y in self,
    !different (x, y) ↔ not (!equal (x, y))
  proof: def !different; ...
end
```

where *#not_b* is the negation over type *bool*.

Using species *setoid*, stacks are seen as a species parameterized by the type of its elements, which must comply the interface of species *setoid*, and also as inheriting from species *setoid* (to compare not only two elements, but also two stacks):

```
species stack (elt is setoid) inherits setoid =
  sig empty in self;
  sig push in elt → self → self;
  sig pop in self → self;
  sig top in self → elt;
  let element = !empty;
  let is_empty (s in self) in bool =
    !equal (s, !empty);
  let has_elements (s in self) in bool =
    #not_b (!is_empty (s));
  theorem ie_empty : !is_empty (!empty)
  proof: by !equal_reflexive def !is_empty;
  theorem he_empty :
    not (!has_elements (!empty))
  proof: by !ie_empty def !has_elements; ...
end
```

2.2 Implementation: Collection

The other main notion of the Focal language is the structure of *collection*, which corresponds to the implementation of a specification. A collection implements a species in such a way that every attribute becomes concrete: the representation must be concrete, functions must be defined and properties must be proved. If the implemented species is parameterized, the collection must also provide implementations for these parameters: either a collection if the parameter is a species or a given entity if the parameter denotes an entity of a species. Moreover, a collection is seen (by the other species and collections) through its corresponding interface; in particular, the representation is an abstract data type and only the definitions of the collection are able to manipulate the entities of this type. Finally, a collection is a terminal item and cannot be extended or refined by inheritance. The syntax of a collection is the following:

```
collection <name> implements
           <name> (<pars>) = ... end
```

To illustrate the notion of collection, let us consider an implementation of our example of stacks seen previously. Once the abstract specification of stacks given, it is possible to provide an implementation based on lists by means of a completely defined species, i.e. a species in which every attribute is concrete. This implementation is given by species *stack_list*:

```
species stack_list (elt is setoid)
        inherits stack (elt) =
  rep = list (elt);
  let rec equal (x, y) = #list_eq (x, y);
  let empty = #Nil;
  let push (e, s) = #Cons (e, s);
  let pop (s) =
    if !is_empty (s) then
      #foc_error ("empty␣stack")
    else #tl (s);
  let top (s) =
    if !is_empty (s) then
      #foc_error ("empty␣stack")
    else #hd (s);
  let is_empty (s) = #list_eq (s, #Nil);
  proof of equal_reflexive = ...;
  proof of ie_empty = ...; ...
end
```

where *list* is the concrete data type of lists, which provides the two constructors *#Nil* and *#Cons*, and where *#list_eq* is the equality over type *list*. The “proof of” clause allows us to provide a proof to an inherited property.

To get executable code, we can build, for instance, the collection of stacks of integers simply implementing the previous species without needing to provide any additional attribute:

```
collection stack_list_int implements
           stack_list (ints) = end
```

where *ints* is the collection of integers which implements species *setoid*.

2.3 Certification: Proving with Zenon

The certification of a Focal specification is ensured by the possibility of proving properties through a declarative proof language and using Zenon [2], a first-order automated theorem prover, which is the reasoning support of Focal. A remarkable feature of Zenon is that it is a certifying automated theorem prover, in the sense that it is able to produce proofs. In particular, Zenon can directly generate Coq proofs which can be reinserted in the Coq specifications produced by the Focal compiler and fully verified by Coq.

2.4 Further Information

For additional information regarding Focal and its applications, the reader can refer to [1, 12]. It should also be noted that a new version of the Focal compiler, called Focalize, has been recently released and is available at [13].

3. COMPILATION USING MODULES

In this section, we present a compilation scheme from Focal to OCaml and Coq, not only through a complete example but also providing a formal description. This compilation scheme is based on modules and appears as an alternative to the actual compiler using records [7], which allows us to preserve the structure of Focal specifications in the compiled code and therefore provides traceability w.r.t. these specifications. An implementation of this compilation scheme is in progress.

3.1 A Complete Example

The considered example concerns the implementation of stacks introduced in Section 2. In the following, the basic idea of the compilation for OCaml and Coq is that a species corresponds to a functor parameterized by some attributes still abstract and a collection corresponds to a module resulting from the application of a functor representing the implemented species to modules representing the actual parameters provided to the species. We suppose that the reader is familiar with OCaml and Coq, and with their respective module systems in particular; otherwise, the reader can refer to [10, 11] for more information regarding these two systems.

3.1.1 Representation

The first species to be compiled is the root species *basic_object*. In OCaml, modules cannot be partially defined, contrary to Focal species where not only representations can be abstract, but also functions or properties. To keep this abstraction in OCaml, the idea is to create a functor parameterized by the attributes still abstract (typically, representations and functions). Thus, the considered species is compiled to the following functor *Basic_object*:

```
module type BASIC_OBJECT =
sig
  type self
  val print : self → string
  val parse : string → self
end

module Basic_object
  (Abs : sig type self end) : BASIC_OBJECT
  with type self = Abs.self =
struct
  type self = Abs.self
  let print (x : self) = "<abst>"
  let parse (x : string) : self =
    failwith "not␣parsable"
end
```

In Coq, the module system offers a quite mixin-oriented approach, in the sense that a module and even a module type may contain abstract and defined attributes (typically, declarations and definitions, but also axioms and theorems). This approach is probably one of the most appropriate to model the semantics of Focal and this allows us to get rid of this notion of module including the abstract attributes (module *Abs* in OCaml), and which appears as a parameter of the functor representing the compiled species. The representation, if abstract, must always be a parameter, but does not need to be included in the module signature representing the interface of the species as required by OCaml (see module type *BASIC_OBJECT*), since we can use a parameterized module signature, which is a feature recently provided by Coq. The Coq compilation is the following:

```
Module Type REP.
  Parameter t : Set.
End REP.

Module Type BASIC_OBJECT (Self : REP).
  Parameter print : Self.t → string.
  Parameter parse : string → Self.t.
End BASIC_OBJECT.

Module Basic_object (Self : REP).
  Definition print (x : Self.t) : string :=
    "<abst>".
  Definition parse (x : string) : Self.t :=
    foc_error Self.t "not␣parsable".
End Basic_object.
```

where *foc_error* is a function encoding the corresponding exception operator.

In the following, we focus on the functor corresponding to the compiled species (typically, *Basic_object* in the previous example), and we do not provide the module signature representing the interface of this species (i.e. *BASIC_OBJECT* in the previous example).

3.1.2 Inheritance

In the example of stacks, inheritance occurs when we introduce species *setoid*. The OCaml compilation of this inheritance is made by means of the inclusion of a module which results from the instantiation of the functor corresponding to the inherited species. The actual parameter of this functor is a module containing the attributes which are abstract in the inherited species and which may be either still abstract or concrete in the sub-species. In our case, this module only includes the representation, which is still abstract. The compilation is as follows:

```
module Setoid
  (Abs : sig
     type self
     val equal : self → self → bool
     val element : unit → self
   end) : SETOID with type self = Abs.self =
struct
  include Basic_object
    (struct type self = Abs.self end)
  let equal = Abs.equal
  let element = Abs.element
  let different x y = not (equal x y)
end
```

The Coq compilation of this inheritance is rather similar and is also realized through the inclusion of the module which corresponds to the instantiation of the functor representing the inherited species. As seen previously, this instantiation only concerns the representation. The compilation is the following:

```
Module Setoid (Self : REP) <: SETOID (Self).
  Include Basic_object (Self).
  Parameter equal : Self.t → Self.t → bool.
  Parameter element : Self.t.
  Definition different (x y : Self.t) : bool :=
    negb (equal x y).
  Axiom equal_reflexive : forall x : Self.t,
    Is_true (equal x x).
  Theorem same_is_not_different :
    forall x y : Self.t,
    Is_true (different x y) ↔
    Is_true (negb (equal x y)).
  Proof. ...
End Setoid.
```

The next species to be compiled is species *stack*. As seen previously, in OCaml, the inheritance is realized through the inclusion of a module representing the application of the functor corresponding to the inherited species to a module containing the instantiations of the attributes of the inherited species previously abstract. Thus, the module of inheritance depends on the actual module of abstractions. However, some other dependencies may appear. For example, it is possible to concretize a function previously abstract using a function which is added in the considered species, as for function *element* in species *stack* which is defined using function *empty*; this implies that the actual module of abstractions may depend on the functions of the compiled species. In addition, a function which is added in the considered species, may depend on a function coming from the inheritance, as for function *is_empty* in species *stack* which is defined using function *equal* coming from species *setoid*; this means that the functions of the compiled species may depend on the module of inheritance. Thus, we have mutual dependencies between the module of inheritance, the actual module of abstractions, and the module gathering the functions of the compiled species. As a consequence, we need to introduce a block of recursive modules as follows:

```
module Stack (Elt : SETOID)
  (Abs : sig
     type self
     val equal : self → self → bool
     val empty : unit → self
     val push : Elt.self → self → self
     val pop : self → self
     val top : self → Elt.self
   end) : STACK with type elt = Elt.self
                and type self = Abs.self =
struct
  type elt = Elt.self
  type self = Abs.self
  module rec M : sig ... end = struct
    let empty = Abs.empty
    let push = Abs.push
    let pop = Abs.pop
    let top = Abs.top
    let element = empty
    let is_empty s = I.equal s (empty ())
    let has_elements s = not (is_empty s)
  end
  and Abs_I : sig ... end = struct
    type self = Abs.self
    let equal = Abs.equal
    let element = M.element
  end
  and I : SETOID with type self = self =
    Setoid (Abs_I)
  let print = I.print
  let parse = I.parse ...
  let empty = M.empty
  let push = M.push ...
end
```

where I is the module of inheritance, *Abs_I* the actual module of abstractions, and M the module of the functions of the compiled species. It should be noted that the previous code does not use the inclusion mechanism of OCaml, since we have to include definitions both from modules I and M, which may overlap.

In Coq, the absence of the module of abstractions allows us to avoid the use of a block of recursive modules. The compilation is made by means of a selective inclusion of the module corresponding to the instantiation of the inheritance functor, and which consists in only including inherited attributes which are not defined or redefined in the compiled species. The attributes added in the compiled species are then also included. The compiled code is the following:

```
Module Stack (Elt : REP) (Std : SETOID (Elt))
  (Self : REP) <: STACK (Elt) (Std) (Self).
  Module I := Setoid (Self).
  Definition print := I.print.
  Definition parse := I.parse.
  Definition equal := I.equal.
  Definition different := I.different.
  Definition equal_reflexive :=
    I.equal_reflexive.
  Definition same_is_not_different :=
    I.same_is_not_different. ...
  Parameter empty : Self.t.
  Parameter push : Elt.t → Self.t → Self.t.
  Parameter pop : Self.t → Self.t.
  Parameter top : Self.t → Elt.t.
  Definition element : Self.t := empty.
  Definition is_empty (s : Self.t) :=
    equal s empty.
  Definition has_elements (s : Self.t) :=
    negb (is_empty s).
  Theorem ie_empty : Is_true (is_empty (empty)).
  Proof. ...
  Theorem he_empty :
    Is_true (negb (has_elements (empty))).
  Proof. ...
End Stack.
```

where I corresponds to the module of inheritance. As for OCaml, this code does not use the primitive inclusion of Coq.

3.1.3 Late Binding

Late binding can be illustrated by means of the compilation of species *stack_list*. To compile this species in OCaml, we must first notice that function *is_empty* is redefined (this new definition is semantically the same than previously and is actually provided just for the purpose of presenting a case of redefinition). This redefinition implies that every function referring to *is_empty* cannot be inherited as it refers to the former definition of *is_empty* and not to the latter. For example, this is the case of function *has_elements* defined in species *stack*. To solve this problem without having to repeat the code of every function referring to *is_empty*, we introduce the notion of function generator. A function generator is a function based on the previous defined function where every reference to another function of the species has been abstracted. The corresponding defined function is then obtained applying its function generator to the actual functions of the species that have been abstracted in the function generator. For each function requiring the use of a function generator, the corresponding function generator is added to the module representing the compiled species and can then be reused later by inheritance. Thus, the compilation is realized as follows:

```
module Stack_list (Elt : SETOID) :
  STACK_LIST with type elt = Elt.self
             and type self = Elt.self list =
struct
  type elt = Elt.self
  type self = elt list
  let pop_gen f s =
    if f s then failwith "empty␣stack"
    else List.tl s
  let top_gen f s =
    if f s then failwith "empty␣stack"
    else List.hd s
  module rec M : sig ... end = struct
    let equal x y = (x = y)
    let empty () = []
    let push e s = e :: s
    let pop = pop_gen M.is_empty
    let top = top_gen M.is_empty
    let is_empty s = (s = [])
  end
  and Abs_I : sig ... end = struct
    type self = elt list
    let equal = M.equal
    let empty = M.empty
    let push = M.push
    let pop = M.pop
    let top = M.top
  end
  and I : STACK with type elt = elt
    and type self = self = Stack (Elt) (Abs_I)
  and Gen : sig ... end = struct
    let has_elements =
      I.has_elements_gen M.is_empty
  end
  let print = I.print
  let parse = I.parse ...
  let equal = M.equal
  let empty = M.empty ...
  let has_elements = Gen.has_elements
  let has_elements_gen = I.has_elements_gen
end
```

where *Gen* is a module where inherited and defined functions are updated using their associated function generators. This module is introduced in the block of recursive modules, as updating inherited functions requires the use of their inherited function generators (from module I) and as functions that are added in the compiled species (from module M) may also require the use of updated functions. It should also be noted that function generators associated with added functions of the considered species, like *pop_gen* for example, are not included in the block of recursive modules as they have no dependency w.r.t. other functions, whereas other function generators associated with inherited functions, e.g. *has_elements_gen*, are inherited as regular functions.

In Coq, the redefinition of function *is_empty* poses the same problem with wider influences. In the same way, we have to use function generators for defined functions using *is_empty*, as for function *has_elements* for instance. However, the dependencies w.r.t. *is_empty* also concerns properties, whose the statements as well as the proofs may depend on this function; this is the case of theorem *ie_empty*, for example. Therefore, we have to introduce the notion of property generator, which actually consists of two generators: a statement generator and a proof generator (if the property is a theorem). As for function generators, these two generators are functions which make an abstraction of the functions, but also of the properties, respectively involved in the statement and the proof of a property. For proof generators, the abstraction of a function is made only if the proof does not depend on the definition of this function, as the proof is invalidated if this function is redefined; for instance, this is the case of *ie_empty* which must be reproved due to the redefinition of *is_empty*. As in OCaml, all the generators are included in the module representing the compiled species. Thus, we obtain the following compilation:

```
Module Stack_list (Elt : REP)
  (Std : SETOID (Elt)) <:
  STACK_LIST (Elt) (Std).
  Module Self := List_of (Elt).
  Module I := Stack (Elt) (Std) (Self).
  Definition print := I.print.
  Definition parse := I.parse.
  Definition element := I.element.
  Definition has_elements_gen :=
    I.has_elements_gen. ...
  Definition equal (s1 s2 : Self.t) : bool :=
    proj1_sig (bool_of_sumbool
      (list_eq_dec elt_dec s1 s2)).
  Definition empty : list Elt.t := nil.
  Definition is_empty (s : Self.t) : bool :=
    proj1_sig (bool_of_sumbool
      (list_eq_dec elt_dec s nil)).
  Definition push (x : Elt.t) (s : Self.t) :
    Self.t := x :: s.
  Definition pop_gen (f : Self.t → bool)
    (s : Self.t) :=
    if (f s) then
      foc_error (Self.t) "empty␣stack"
    else (tail s).
  Definition top_gen (f : Self.t → bool)
    (s : Self.t) :=
    if (f s) then
      foc_error (Elt.t) "empty␣stack"
    else (hd s).
  Definition pop := pop_gen is_empty.
  Definition top := top_gen is_empty.
  Definition has_elements :=
    I.has_elements_gen is_empty.
  Definition ie_empty_gentyp :=
    I.ie_empty_gentyp.
  Theorem ie_empty :
    ie_empty_gentyp empty is_empty.
  Proof. ...
  Definition he_empty_gentyp :
    forall (e : Self.t) (f : Self.t → bool),
    Is_true (negb (f e)) := I.he_empty_gentyp.
  Definition he_empty_genthm :
    forall (e : Self.t) (ie : Self.t → bool)
    (iee : ie_empty_gentyp e ie),
    hs_empty_gentyp e (has_elements_gen ie) :=
    I.he_empty_genthm.
  Definition he_empty :
    he_empty_gentyp empty has_elements :=
    he_empty_genthm empty is_empty ie_empty. ...
End Stack_list.
```

where *List_of* is a functor which allows us to build a concrete representation module based on lists. As in OCaml, defined functions introduce function generators, like *pop_gen*, whereas the function generators of inherited functions are also inherited, like *has_elements_gen*. Regarding property generators, we can see how the two generators of *he_empty*, i.e. statement generator *he_empty_gentyp* and proof generator *he_empty_genthm*, can be used to inherit properly the statement and the proof of *he_empty*. For *ie_empty*, as the inherited proof is invalidated and must be rebuilt, only the corresponding statement generator *ie_empty_gentyp* can be used.

3.1.4 Collections

In our example, an implementation is provided by means of collection *stack_list_int*, which represents the stacks of integers. In OCaml, this collection is a module resulting from the application of the functor corresponding to species *stack_list* to the module representing collection *ints*. This compilation is realized as follows:

```
module Stack_list_int = Stack_list (Ints)
```

where *Ints* is the module corresponding to collection *ints*.

In Coq, the compilation is similar, but as seen previously, we have to additionally provide a module which is supposed to be the representation of the collection also supplied as parameter. The code is the following:

```
Module Stack_list_int :=
  Stack_list (Rep_ints) (Ints).
```

where *Ints* is the module representing collection *ints*, and *Rep_ints* the module corresponding to the representation of this collection.

3.2 Formal Description

The previous example gives an idea of the considered compilation schemes. In this subsection, we aim to formally describe these compilation schemes in the general case. Due to space restrictions, we only deal with the compilation schemes of a species, as it consists of the main difficult point of the compilation. For the same reasons, we also do not deal with the corresponding proofs of correctness.

Given a species S, which has the following general form: $S =$ **species** s $(\mathcal{P})$ **inherits** $\mathcal{I} = rep$; $\mathcal{M}$; $\mathcal{R}$; **end**, where s denotes the name of the species, $\mathcal{P}$ the list of parameters, $\mathcal{I} = S_1 \dots S_m$ the list of inherited species of the form $S_i = s_i(a_1, \dots, a_{n_i})$ where s_i is a species and a_j with $j = 1 \dots n_i$ are actual parameters of s_i, *rep* the representation, $\mathcal{M} = \phi_1 \dots \phi_p$ the list of functions, and $\mathcal{R} = \psi_1 \dots \psi_q$ the list of properties. Given a typing context Γ in which S is well typed, $[\![S]\!]_\Gamma$ denotes the compilation of S and is given by Figures 1 and 2 for OCaml and Coq. Regarding these translations, we do not provide all the details and in OCaml, we mainly focus on the construction of the block of recursive modules, while in Coq, we essentially concentrate on the code generation for properties. In both compilations, we first have the signature associated with the functor representing the species $[\![S]\!]_{\Gamma}^{\text{sig}}$ and then the functor itself $[\![S]\!]_{\Gamma}^{\text{mod}}$. In Coq, preceding the previous signature and functor, we also find two modules $[\![rep]\!]_{\Gamma}^{\text{conc}}$ and $[\![\mathcal{P}]\!]_{\Gamma}^{\text{ent}}$, corresponding respectively to the representation (if concrete) and to the entity parameters.

4. CONCLUSION

In this paper, we have introduced the Focal environment, which allows us to build certified components using theorem proving. The language supported by this environment is object-oriented, and inheritance provides a suitable notion of refinement, going incrementally from abstract specifications to concrete implementations. In addition, inheritance and parameterization offer the capability of creating higher order structures, which can be therefore directly reused and extended. To highlight these features, we have also described a compilation scheme of Focal specifications based on modules and able to generate both OCaml and Coq codes. Thanks to modules, which are high level structures, this scheme can preserve the structure of specifications and ensures a certain traceability. An implementation of this scheme is in progress in the framework of the Focal compiler and should allow us to assess the feasibility of such an approach in practice.

5. REFERENCES

[1] P. Ayrault, M. Carlier, D. Delahaye, C. Dubois, D. Doligez, L. Habib, T. Hardin, M. Jaume, C. Morisset, F. Pessaux, R. Rioboo, and P. Weis. Trusted Software within Focal. In *Computer & Electronics Security Applications Rendez-Vous (C&ESAR)*, pages 142–158, Rennes (France), Dec. 2008.

[2] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165. Springer, Oct. 2007.

[3] Carnegie Mellon University. *The Million Book Project*, Dec. 2001. http://www.rr.cs.cmu.edu/mbdl.htm.

[4] D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Certifying Airport Security Regulations using the Focal Environment. In *Formal Methods (FM)*, volume 4085 of *LNCS*, pages 48–63, Hamilton, Ontario (Canada), Aug. 2006. Springer.

[5] Google. *The Google Book Search*, Oct. 2004. http://books.google.com/.

[6] M. Jaume and C. Morisset. A Formal Approach to Implement Access Control. *Journal of Information Assurance and Security (JIAS)*, 1(2):137–148, June 2006.

[7] V. Prevosto. *Conception et implantation du langage Foc pour le développement de logiciels certifiés*. PhD thesis, Université Pierre et Marie Curie (Paris 6), Sept. 2003.

[8] R. Rioboo. Programmer le Calcul Formel : des algorithmes à la sémantique. Habilitation à diriger des recherches, Université Pierre et Marie Curie (Paris 6), Dec. 2002.

[9] P. Sojka, editor. *Towards a Digital Mathematics Library (DML)*. Masaryk University, July 2008.

[10] The Caml Development Team. *Objective Caml, version 3.11.0*. INRIA, Dec. 2008. http://caml.inria.fr/.

[11] The Coq Development Team. *Coq, version 8.2*. INRIA, Feb. 2009. http://coq.inria.fr/.

[12] The Focal Development Team. *Focal, version 0.3.1*. CNAM, INRIA, and LIP6, May 2005. http://focal.inria.fr/.

[13] The Focalize Development Team. *Focalize, version 0.1 RC 1*. CNAM, INRIA, and LIP6, Apr. 2009. http://focalize.inria.fr/.

[14] The Project Gutenberg, 1971. http://www.gutenberg.org.

[15] The Internet Archive, 1996. http://www.archive.org.

[16] The International Electrotechnical Commission. *Functional Safety of Electrical/Electronic/ Programmable Electronic Safety-Related Systems (IEC 61508)*, Jan. 2005. http://www.iec.ch/.

[17] The Mathematical Knowledge Management (MKM) Interest Group, 2001. http://www.mkm-ig.org/.

[18] The Members of the Common Criteria Recognition Arrangement. *Common Criteria for Information Technology Security Evaluation, version 3.1*, Sept. 2007. http://www.commoncriteriaportal.org/.

[19] The MoWGLI Project, 2002. http://mowgli.cs.unibo.it/.

[20] The OpenMath Society. *OpenMath Version 2.0*, June 2004. http://www.openmath.org/.

$$[\![S]\!]_\Gamma = [\![S]\!]^{\text{sig}}_\Gamma \; [\![S]\!]^{\text{mod}}_\Gamma$$

$$\begin{array}{l}
[\![S]\!]^{\text{mod}}_\Gamma = \\
\quad \textbf{module } \mathit{mod}(s)\; [\![\mathcal{P}]\!]^{\text{param}}_\Gamma \\
\qquad (\text{Abs} : \textbf{sig} \\
\qquad\quad [\![\mathit{rep}]\!]^{\text{abs}}_{\Gamma,S} \\
\qquad\quad [\![\mathit{abs}(S)]\!]^{\text{abs}}_{\Gamma,\mathit{rep},\mathcal{P}} \\
\qquad \textbf{end}) : \\
\qquad \mathit{sig}(s) \textbf{ with } [\![\mathit{rep}]\!]^{\text{mod}}_\Gamma \; [\![\mathcal{P}]\!]^{\text{with}}_\Gamma = \\
\quad \textbf{struct} \\
\qquad [\![\mathcal{P}]\!]^{\text{mod}}_\Gamma \; [\![\mathit{rep}]\!]^{\text{mod}}_{\Gamma,S} \\
\qquad [\![\mathcal{M}]\!]^{\text{gen}}_\Gamma \; [\![\mathcal{M}]\!]^{\text{mod}}_\Gamma \\
\qquad [\![\mathcal{I}]\!]^{\text{mod}}_{\Gamma,\mathcal{M}} \; [\![\mathcal{I}]\!]^{\text{gen}}_\Gamma \\
\qquad [\![\mathcal{I}]\!]^{\text{inc}}_{\Gamma,\mathcal{M}} \; [\![\mathcal{M}]\!]^{\text{inc}}_\Gamma \\
\quad \textbf{end}
\end{array}$$

$$[\![\mathcal{M}]\!]^{\text{gen}}_\Gamma = [\![\phi_1]\!]^{\text{gen}}_\Gamma \ldots [\![\phi_n]\!]^{\text{gen}}_\Gamma$$

$$\begin{array}{l}
[\![\mathcal{M}]\!]^{\text{mod}}_\Gamma = \\
\quad \textbf{module rec } M : \textbf{sig} \\
\qquad [\![\phi_1]\!]^{\text{sig}}_\Gamma \ldots [\![\phi_n]\!]^{\text{sig}}_\Gamma \\
\quad \textbf{end} = \textbf{struct} \\
\qquad [\![\phi_1]\!]^{\text{mod}}_\Gamma \ldots [\![\phi_n]\!]^{\text{mod}}_\Gamma \\
\quad \textbf{end}
\end{array}$$

$$[\![\mathcal{I}]\!]^{\text{mod}}_\Gamma = [\![S_1]\!]^{\text{mod}}_\Gamma \ldots [\![S_n]\!]^{\text{mod}}_\Gamma$$

$$\begin{array}{l}
[\![S_i]\!]^{\text{mod}}_\Gamma = \\
\quad \textbf{and } \text{Abs}i : \textbf{sig} \\
\qquad [\![\mathit{rep}]\!]^{\text{abs}}_{\Gamma,S_i} \; [\![\mathit{abs}(S_i)]\!]^{\text{sig}}_\Gamma \\
\quad \textbf{end} = \textbf{struct} \\
\qquad [\![\mathit{rep}]\!]^{\text{abs-mod}}_{\Gamma,S_i} \; [\![\mathit{abs}(S_i)]\!]^{\text{abs-mod}}_\Gamma \\
\quad \textbf{end} \\
\quad \textbf{and } \text{S}i : \mathit{sig}(s_i) \textbf{ with } [\![\mathit{rep}]\!]^{\text{mod}}_\Gamma \; [\![S_i]\!]^{\text{with}}_{\Gamma,\mathcal{P}} = \\
\qquad\qquad \mathit{mod}(s_i)\; [\![S_i]\!]^{\text{param}}_\Gamma
\end{array}$$

$$\begin{array}{l}
[\![I]\!]^{\text{gen}}_{\Gamma,\mathcal{M}} = \\
\quad \textbf{and } \text{Gen} : \textbf{sig} \\
\qquad [\![S_n]\!]^{\text{gen-sig}}_{\Gamma,\mathcal{M}} \ldots [\![S_1]\!]^{\text{gen-sig}}_{\Gamma,\mathcal{M}} \\
\quad \textbf{end} = \textbf{struct} \\
\qquad [\![S_n]\!]^{\text{gen-mod}}_{\Gamma,\mathcal{M}} \ldots [\![S_1]\!]^{\text{gen-mod}}_{\Gamma,\mathcal{M}} \\
\quad \textbf{end}
\end{array}$$

$$\begin{aligned}
[\![S_i]\!]^{\text{gen-mod}}_{\Gamma,\mathcal{M}} &= [\![\mathit{fun}(S_i)]\!]^{\text{gen-mod}}_{\Gamma,\mathcal{M}} \\
&= [\![\phi_1]\!]^{\text{gen-mod}}_{\Gamma,\mathcal{M}} \ldots [\![\phi_n]\!]^{\text{gen-mod}}_{\Gamma,\mathcal{M}}
\end{aligned}$$

$$[\![\phi_i]\!]^{\text{gen}}_\Gamma = \begin{cases}
\textbf{let } \mathit{m_gen} = [\![\mathit{dep}(m)]\!]^{\text{mod}}_\Gamma \; [\![\mathit{body}]\!]_\Gamma, \\
\quad \text{if } \phi_i = \textbf{let } m \textbf{ in } \tau = \mathit{body}; \\
\quad \text{and } \mathit{dep}(m) \neq \emptyset \\
\emptyset, \text{ otherwise}
\end{cases}$$

$$[\![\mathit{dep}(m)]\!]^{\text{mod}}_\Gamma = \begin{cases}
\emptyset, \text{ if } n = 0 \\
\textbf{fun } m_1 \rightarrow \\
\quad [\![(m_2 : \tau_2) \ldots (m_n : \tau_n)]\!]^{\text{mod}}_\Gamma, \\
\quad \text{otherwise}
\end{cases}$$

$$[\![\phi_i]\!]^{\text{mod}}_\Gamma = \begin{cases}
\textbf{let } m = \text{Abs}.m, \text{ if } \phi_i = \textbf{sig } m \textbf{ in } \tau; \\
\textbf{let } m = [\![\mathit{body}]\!]_\Gamma, \\
\quad \text{if } \phi_i = \textbf{let } m \textbf{ in } \tau = \mathit{body}; \\
\quad \text{and } \mathit{dep}(m) = \emptyset \\
\textbf{let } m = m_\text{gen } m_1 \ldots m_n, \\
\quad \text{if } \phi_i = \textbf{let } m \textbf{ in } \tau = \mathit{body}; \\
\quad \text{and } \mathit{dep}(m) = \{(m_1 : \tau_1), \ldots, (m_n : \tau_n)\}
\end{cases}$$

$$[\![\mathit{abs}(S_i)]\!]^{\text{abs-mod}}_\Gamma = [\![\phi_1]\!]^{\text{abs-mod}}_\Gamma \ldots [\![\phi_n]\!]^{\text{abs-mod}}_\Gamma$$

$$[\![\phi_i]\!]^{\text{abs-mod}}_\Gamma = \begin{cases}
\textbf{let } m = \text{M}.m, \text{ if } \phi_i \in \mathcal{M} \\
\textbf{let } m = \text{Abs}.m, \text{ if } \phi_i \notin \mathcal{M} \\
\quad \text{and } \nexists j = 1 \ldots n \text{ s.t.} \\
\quad \phi_i = \textbf{let } m \textbf{ in } \tau = \mathit{body}; \in \mathit{fun}(S_j) \\
\textbf{let } m = \text{S}_j.m, \text{ if } \phi_i \notin \mathcal{M} \\
\quad \text{and } \exists j = 1 \ldots n \text{ s.t.} \\
\quad \phi_i = \textbf{let } m \textbf{ in } \tau = \mathit{body}; \in \mathit{fun}(S_j) \\
\quad \text{and } \forall\; k > j,\; \phi_i \notin \mathit{fun}(S_k)
\end{cases}$$

$$[\![\mathit{rep}]\!]^{\text{abs-mod}}_{\Gamma,S_i} = \begin{cases}
\textbf{type } \text{self} = \text{Abs.self}, \\
\quad \text{if } \mathit{rep}(S_i) = \emptyset \\
\quad \text{and } \mathit{rep}(S) = \emptyset \\
\textbf{type } \text{self} = [\![\tau]\!]_\Gamma, \\
\quad \text{if } \mathit{rep}(S_i) = \emptyset \\
\quad \text{and } \mathit{rep}(S) = \tau \\
\emptyset, \text{ otherwise}
\end{cases}$$

$$[\![\phi_i]\!]^{\text{gen-mod}}_{\Gamma,\mathcal{M}} = \begin{cases}
\textbf{let } m = \mathit{m_gen}\; X_1.m_1 \ldots X_n.m_n, \\
\quad \text{if } \phi_i \notin \mathcal{M} \cup \mathit{fun}(S_j),\; j = n \ldots i+1 \\
\quad \mathit{dep}(m) = \{(m_1 : \tau_1), \ldots, (m_n : \tau_n)\} \\
\quad \text{and } X_i \in \mathit{Gen}, M, S_j \text{ s.t. } m_i \in X_i \\
\quad \text{and if } X_i = S_j \text{ then } m_i \notin \mathit{Gen}, M, S_k \\
\quad \text{s.t. } k > j \\
\emptyset, \text{ otherwise}
\end{cases}$$

where:

$\mathit{sig}(s)$	returns a module signature name from species name s.
$\mathit{mod}(s)$	returns a module name from species name s.
$\mathit{rep}(S)$	returns the representation type τ of species S if it is concrete, $\emptyset$ otherwise.
$\mathit{fun}(S)$	returns the set of functions ϕ_i of species S.
$\mathit{abs}(S)$	returns the set of functions ϕ_i of species S which are declared.
$\mathit{dep}(m)$	returns the set of couples $(m_i : \tau_i)$, where m_i is a function and τ_i its type, and on which the function of name m depends.

Figure 1: From Focal to OCaml

$$[\![S]\!]_\Gamma = \begin{cases} [\![rep]\!]^{\mathrm{conc}}_\Gamma \\ [\![\mathcal{P}]\!]^{\mathrm{ent}}_\Gamma \\ [\![S]\!]^{\mathrm{sig}}_\Gamma \\ [\![S]\!]^{\mathrm{mod}}_\Gamma \end{cases}$$

$$\begin{array}{l} [\![S]\!]^{\mathrm{mod}}_\Gamma = \\ \quad \textbf{Module}\ \mathit{mod}(s)\ [\![rep]\!]^{\mathrm{param}}_\Gamma\ [\![\mathcal{P}]\!]^{\mathrm{param}}_\Gamma <: \\ \qquad\qquad \mathit{sig}(s)\ [\![rep]\!]^{\mathrm{arg}}_{\Gamma,S}\ [\![\mathcal{P}]\!]^{\mathrm{arg}}_\Gamma. \\ \qquad [\![\mathcal{I}]\!]^{\mathrm{mod}}_{\Gamma,\mathcal{M},\mathcal{R}} \\ \qquad [\![\mathcal{M}]\!]^{\mathrm{mod}}_\Gamma \\ \qquad [\![\mathcal{I},\mathcal{M}]\!]^{\mathrm{gen}}_{\Gamma,<_{dep}} \\ \qquad [\![\mathcal{R}]\!]^{\mathrm{mod}}_\Gamma \\ \qquad [\![\mathcal{I},\mathcal{R}]\!]^{\mathrm{gen}}_{\Gamma,<_{dep}} \\ \quad \textbf{End}\ \mathit{mod}(s). \end{array}$$

$$\begin{aligned} [\![\mathcal{I}]\!]^{\mathrm{mod}}_{\Gamma,\mathcal{M},\mathcal{R}} &= [\![S_1 \ldots S_n]\!]^{\mathrm{mod}}_{\Gamma,\mathcal{M},\mathcal{R}} \\ &= [\![S_1]\!]^{\mathrm{mod}}_{\Gamma,\mathcal{M},\mathcal{R},\{S_2,\ldots,S_n\}} \cdots [\![S_n]\!]^{\mathrm{mod}}_{\Gamma,\mathcal{M},\mathcal{R},\emptyset} \end{aligned}$$

$$\begin{array}{l} [\![S_i]\!]^{\mathrm{mod}}_{\Gamma,\mathcal{M},\mathcal{R},X} = \\ \quad \textbf{Module}\ \mathrm{S}_i := \mathit{mod}(s_i)\ [\![rep]\!]^{\mathrm{arg}}_{\Gamma,S_i}\ [\![a_1 \ldots a_{n_i}]\!]^{\mathrm{arg}}_\Gamma. \\ \quad [\![\mathit{fun}(S_i)]\!]^{\mathrm{inh}}_{\Gamma,\mathcal{M},X} \\ \quad [\![\mathit{prop}(S_i)]\!]^{\mathrm{inh}}_{\Gamma,\mathcal{R},X} \end{array}$$

$$\begin{aligned} [\![\mathit{prop}(S_k)]\!]^{\mathrm{inh}}_{\Gamma,\mathcal{R},X} &= [\![\psi_1 \ldots \psi_n]\!]^{\mathrm{inh}}_{\Gamma,\mathcal{R},X} \\ &= \{[\![\psi_i]\!]^{\mathrm{inh}}_{\Gamma,S_k} \mid \psi_i \notin \mathcal{R} \cup X\} \end{aligned}$$

$$[\![\psi_i]\!]^{\mathrm{inh}}_{\Gamma,S_k} = \begin{cases} \textbf{Definition}\ x := \mathrm{S}k.x.,\ \text{if}\ \mathit{dep}(x) = \emptyset \\ [\![\psi_i]\!]^{\text{gen-typ}}_{\Gamma,S_k}[\![\psi_i]\!]^{\text{gen-thm}}_{\Gamma,S_k},\ \text{otherwise} \end{cases}$$

$$[\![\psi_i]\!]^{\text{gen-typ}}_{\Gamma,S_k} = \begin{cases} \textbf{Definition}\ x\_\mathrm{gentyp} := \mathrm{S}k.x\_\mathrm{gentyp}.,\ \text{if}\ \mathit{dep}_{\mathit{typ}}(x) \neq \emptyset \\ \emptyset,\ \text{otherwise} \end{cases}$$

$$[\![\psi_i]\!]^{\text{gen-thm}}_{\Gamma,S_k} = \begin{cases} \textbf{Definition}\ x\_\mathrm{genthm} := \mathrm{S}k.x\_\mathrm{genthm}.,\ \text{if}\ \psi_i =\ \textbf{theorem}\ x : \mathit{prop}\ \textbf{proof} : \mathit{proof}; \\ \emptyset,\ \text{otherwise} \end{cases}$$

$$[\![\mathcal{R}]\!]^{\mathrm{mod}}_\Gamma = [\![\psi_1]\!]^{\mathrm{mod}}_\Gamma \cdots [\![\psi_n]\!]^{\mathrm{mod}}_\Gamma$$

$$[\![\psi_i]\!]^{\mathrm{mod}}_\Gamma = \begin{cases} [\![\psi_i]\!]^{\text{no-dep}}_\Gamma,\ \text{if}\ \mathit{dep}(x) = \emptyset \\ [\![\psi_i]\!]^{\mathrm{dep}}_\Gamma,\ \text{otherwise} \end{cases}$$

$$[\![\psi_i]\!]^{\text{no-dep}}_\Gamma = \begin{cases} \textbf{Axiom}\ x : [\![prop]\!]_\Gamma.\ ,\ \text{if}\ \psi_i = \textbf{property}\ x : \mathit{prop}; \\ \textbf{Theorem}\ x : [\![prop]\!]_\Gamma.\ \textbf{Proof}.\ [\![proof]\!]_\Gamma\ \textbf{Qed}., \\ \quad \text{if}\ \psi_i = \textbf{theorem}\ x : \mathit{prop}\ \textbf{proof} : \mathit{proof}; \end{cases}$$

$$[\![\psi_i]\!]^{\mathrm{dep}}_\Gamma = \begin{cases} \textbf{Definition}\ x\_\mathrm{gentyp}\ (x_1 : [\![\tau_1]\!]_\Gamma) \ldots (x_n : [\![\tau_n]\!]_\Gamma) := [\![prop]\!]_\Gamma.\ , \\ \quad \text{if}\ \psi_i = \textbf{property}\ x : \mathit{prop};\ \text{and}\ \mathit{dep}_{\mathit{typ}}(x) = \{(x_1 : \tau_1), \ldots, (x_n : \tau_n)\} \\ \textbf{Definition}\ x\_\mathrm{genthm}\ (x_1 : [\![\tau_1]\!]_\Gamma) \ldots (x_n : [\![\tau_n]\!]_\Gamma) := [\![proof]\!]_\Gamma.\ , \\ \quad \text{if}\ \psi_i = \textbf{theorem}\ x : \mathit{prop}\ \textbf{proof} : \mathit{proof};\ \text{and}\ \mathit{dep}_{\mathit{thm}}(x) = \{(x_1 : \tau_1), \ldots, (x_n : \tau_n)\} \end{cases}$$

$$[\![\mathcal{I},\mathcal{R}]\!]^{\mathrm{gen}}_{\Gamma,<_{dep}} = \{[\![\psi_i]\!]^{\mathrm{gen}}_\Gamma \mid \psi_i \in \mathit{prop}(S_j)\backslash\mathcal{R}\ \text{and}\ \psi_i \notin S_k\ \text{with}\ k > j,\ \text{or}\ \psi_i \in \mathcal{R},\ \mathit{dep}(x) \neq \emptyset\}$$

$$[\![\psi_i]\!]^{\mathrm{gen}}_\Gamma = \begin{cases} \textbf{Axiom}\ x : x\_\mathrm{gentyp}\ x_1 \ldots x_n., \\ \quad \text{if}\ \psi_i = \textbf{property}\ x : \mathit{prop};\ \text{and}\ \mathit{dep}_{\mathit{typ}}(x) = \{(x_1 : \tau_1), \ldots, (x_m : \tau_m)\} \\ \textbf{Definition}\ x : x\_\mathrm{gentyp}\ x_1 \ldots x_m := x\_\mathrm{genthm}\ y_1 \ldots y_n., \\ \quad \text{if}\ \psi_i = \textbf{theorem}\ x : \mathit{prop}\ \textbf{proof} : \mathit{proof};\ \text{and}\ \mathit{dep}_{\mathit{typ}}(x) = \{(x_1 : \tau_1), \ldots, (x_m : \tau_m)\}, \\ \quad \mathit{dep}_{\mathit{thm}}(x) = \{(y_1 : \tau_1), \ldots, (y_n : \tau_n)\} \end{cases}$$

where :

$\mathit{sig}(s)$	returns a module signature name from species name s.
$\mathit{mod}(s)$	returns a module name from species name s.
$\mathit{fun}(S)$	returns the set of functions ϕ_i of species S.
$\mathit{prop}(S)$	returns the set of properties ψ_i of species S.
$\mathit{dep}_{\mathit{typ}}(x)$	returns the set of couples $(x_i : \tau_i)$, where x_i is either a function or a property and where τ_i is its type, and on which the statement of property of name x depends.
$\mathit{dep}_{\mathit{thm}}(x)$	returns the set of couples $(x_i : \tau_i)$, where x_i is either a function or a property and where τ_i is its type, and on which the proof of property of name x depends, its possible definition excluded.
$\mathit{dep}(x)$	$= \mathit{dep}_{\mathit{typ}}(x) \cup \mathit{dep}_{\mathit{thm}}(x)$.
$<_{\mathit{dep}}$	sorts the set of properties from $[\![\mathcal{I},\mathcal{M}]\!]^{gen}_{\Gamma,<_{dep}}$ s.t. $\forall\ j > i,\ \psi_j \notin \mathit{dep}(\psi_i)$.

Figure 2: From Focal to Coq