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.3.1 [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 collector.
tions. 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 Coq his/her proofs and can directly produce prover, called code [11] for certification, and code for documentation. Coq this language, able to produce 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. Moreover, each of these structures is equipped with a car-
tance and parameterization, inspired by object-oriented pro-
tions. These different structures are combined using inheri-
tance and parameterization, inspired by object-oriented pro-

2. THE FOCAL ENVIRONMENT

In this section, we give an overview of the Focal environment. In particular, we describe the syntax of the speci-
fication language and some ideas regarding the underlying semantics. We also provide a complete example of formal-
ization, 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 (mul-
tiple) 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 pos-
sible 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 param-
eter is a species, the parameterized species only has access to the interface of this species, i.e. only its abstract representa-
tion, its declarations and its properties. These two features complete the previous syntax as follows:

```
species <name> (<name> is <name>[<name>],<name> in <name>[<name>], . . .)
    inherits <name>, <name> (<name>[<name>], . . .) = . . .
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 spec-
ification starts with the following predefined root species
To illustrate the notion of collection, let us consider an implementation of our example of stacks given 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:

```plaintext
species stack_list (elt in 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 !s_empty (s) then
    #for_error ("empty_stack")
    else #tl (s);
  let top (s) =
    if !s_empty (s) then
    #for_error ("empty_stack")
    else #hd (s);
  let is_empty (s) = #list_eq (s, #Nil);
  proof of equal_reflexive = ...
  proof of is_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:

```plaintext
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].
3.1 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:

```ocaml
module type BASIC_OBJECT =
  sig
    type self
    val print : self -> string
    val parse : string -> self
  end

module Basic_object
  (Abs : sig type self end) : BASICOBJECT
  with type self = Abs.self =
struct
  type self = Abs.self
  let print (x : self) = "<abst>"
  let parse (x : string) : self =
    fastwith "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:

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


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

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:

```coq
module Setoid (Abs : sig type self end) :
  with type self = Abs.self =
struct
  include Basic_object

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:

```coq
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. x = x.
  Theorem same_is_not_different : forall x y : Self.t. Is_true (equal x y) <-> Is_true (different 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
    val pop : self → self
    val top : self
end) : STACK with type elt = Elt.self and type selt = Abs.self = struct
eq selt = Elt.self
eq selt = Abs.self
module rec M : sig ...
    let empty = Abs.empty
    let push = Abs.push
    let pop = Abs.pop
    let top = Abs.top
    let element = empty
    let is-empty s = l.equal s (empty ())
    let has-elements s = not (is-empty s)
end

```

where I is the module of inheritance, AbsI 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 is-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 selt = Elt.self list = struct
eq selt = Elt.self

type elt = Elt.self
type selt = elt list
module rec M : sig ...
    let empty = M.empty
    let push = M.push
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 is_empty where 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 I := Stack (Elt) (Std) (Self).
Definition print := I.print.
Definition parse := I.parse.
Definition element := I.element.
Definition has_elements_gen :=
   l. 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
      for_error (Self t) "empty_stack"
   else (hd s).
Definition top_gen (f : Self t → bool)
   (s : Self t) :=
   if (f s) then
      for_error (Elt t) "empty_stack"
   else (hd s).
Definition pop := pop_gen is_empty.
Definition top := top_gen is_empty.
Definition has_elements := l. has_elements_gen is_empty.
Definition is_empty_gen_typ :=
   ic_empty_gen_typ empty is_empty := empty.
   ... End Stack_list.
```

where List_of is a function 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 is_empty, i.e. statement generator has_elements_gen and proof generator IC_empty_gen, can be used to inherit properly the statement and the proof of is_empty. For IC_empty_gen, as the inherited proof is invalidated and must be rebuilt, only the corresponding statement generator IC_empty_gen 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: $S = \text{species } s (P) \text{ inherits } I = \text{rep; } M; R; \text{ end}$, where $s$ denotes the name of the species, $P$ the list of parameters, $I = S_1, \ldots, S_m$, the list of inherited species of the form $S_i = s_1(a_1, \ldots, a_n)$ where $s_1$ is a species and $a_j$ with $j = 1 \ldots n_i$ actual parameters of $s_i$, rep the representation, $M = \psi_1, \ldots, \psi_p$ the list of functions, and $X = \psi_1, \ldots, \psi_q$ the list of properties. Given a typing context $\Gamma$ in which $S$ is well typed, [S] 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 \([S]^{\text{con}}\) and then the functor itself \([S]^{\text{con}}\). In Coq, preceding the previous signature and functor, we also find two modules \([\text{rep}]^{\text{con}}\) and \([\text{P}]^{\text{con}}\), 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


\[
\begin{align*}
[S]_\Gamma &= [S]_\Gamma^{\text{param}} \\
[S]_\Gamma^{\text{mod}} &= \text{module} \mod(s) [P]_\Gamma^{\text{param}} \text{ (Abs : \sig)} \\
&\quad \text{[rep]}_S^{\text{abs}} [\text{abs}(S)]_\Gamma^{\text{rep},p} \\
&\quad \text{end} \\
&\quad \text{sig(s) with} [\text{rep}]_\Gamma^{\text{mod}} [P]_\Gamma^{\text{with} =} \\
\end{align*}
\]

\[
\begin{align*}
[M]_\Gamma^{\text{gen}} &= [\phi_1]_\Gamma^{\text{gen}} \ldots [\phi_n]_\Gamma^{\text{gen}} \\
[M]_\Gamma &= \text{module} \text{rec } M : \sig \\
&\quad [\phi_1]_\Gamma^{\text{sig}} \ldots [\phi_n]_\Gamma^{\text{sig}} \\
&\quad \text{end} \\
&\quad \text{struct} \\
&\quad [\phi_1]_\Gamma^{\text{mod}} \ldots [\phi_n]_\Gamma^{\text{mod}} \\
\end{align*}
\]

\[
\begin{align*}
[S]_\Gamma^{\text{mod}} &= [\text{abs}(S)]_\Gamma^{\text{abs-mod}} \\
[S]_\Gamma^{\text{mod}} &= \text{and} \text{Abs : \sig} \\
&\quad [\text{rep}]_\Gamma^{\text{abs-mod}} [\text{abs}(S)]_\Gamma^{\text{abs-mod}} \\
&\quad \text{end} \\
&\quad \text{and } S \text{: sig(s) with} [\text{rep}]_\Gamma^{\text{mod}} [S]_\Gamma^{\text{with} =} \\
\end{align*}
\]

\[
\begin{align*}
[J]_\Gamma^{\text{gen}} &= \text{and} \text{Gen : \sig} \\
&\quad [S_1]_\Gamma^{\text{gen-sig}} \ldots [S_n]_\Gamma^{\text{gen-sig}} \\
&\quad \text{end} \\
&\quad \text{struct} \\
&\quad [S_1]_\Gamma^{\text{gen-mod}} \ldots [S_n]_\Gamma^{\text{gen-mod}} \\
\end{align*}
\]

\[
\begin{align*}
[S]_\Gamma^{\text{gen-mod}} &= [\text{fun}(S)]_\Gamma^{\text{gen-mod}} \\
&\quad [\phi_1]_\Gamma^{\text{gen-mod}} \ldots [\phi_n]_\Gamma^{\text{gen-mod}} \\
\end{align*}
\]

where:

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

\[
\begin{align*}
[\phi_1]_\Gamma^{\text{gen}} &= \begin{cases} 
\text{let } m, \text{gen} = \{\text{dep}(m)\}_\Gamma^{\text{mod}} \{\text{body}\}_\Gamma, \\
\text{if } \phi_i = \text{let } m \text{ in } \tau = \text{body}; \\
\text{and } \text{dep}(m) \neq \emptyset \\
\emptyset, \text{ otherwise}
\end{cases} \\
\end{align*}
\]

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

\[
\begin{align*}
[\phi_1]_\Gamma^{\text{mod}} &= \begin{cases} 
\text{let } m = \text{Abs.m, if } \phi_i = \text{sig m in } \tau; \\
\text{let } m = \{\text{body}\}_\Gamma, \\
\text{if } \phi_i = \text{let } m \text{ in } \tau = \text{body}; \\
\text{and } \text{dep}(m) = \emptyset \\
\text{let } m = \text{m-gen m}_1 \ldots \text{m}_n, \\
\text{if } \phi_i = \text{let } m \text{ in } \tau = \text{body}; \\
\text{and } \text{dep}(m) = \{\text{m}_1 : \tau_1\}, \ldots \{\text{m}_n : \tau_n\} &
\end{cases}
\end{align*}
\]

\[
\begin{align*}
[\text{abs}(S_i)]_\Gamma^{\text{abs-mod}} &= [\phi_1]_\Gamma^{\text{abs-mod}} \ldots [\phi_n]_\Gamma^{\text{abs-mod}} \\
[\phi_1]_\Gamma^{\text{abs-mod}} &= \begin{cases} 
\text{let } m = \text{M.m, if } \phi_i \in M \\
\text{let } m = \text{Abs.m, if } \phi_i \notin M \\
\text{and } \exists j = 1 \ldots n \text{ s.t.} \\
\phi_i = \text{let } m \text{ in } \tau = \text{body}; \in \text{fun}(S_j) \\
\text{let } m = \text{S.j.m, if } \phi_i \notin M \\
\text{and } \exists j = 1 \ldots n \text{ s.t.} \\
\phi_i = \text{let } m \text{ in } \tau = \text{body}; \in \text{fun}(S_j) \\
\text{and } \forall k > j, \phi_i \notin \text{fun}(S_k) &
\end{cases}
\end{align*}
\]

\[
\begin{align*}
[\text{rep}]_\Gamma^{\text{abs-mod}} &= \begin{cases} 
\text{type self = Abs.self,} \\
\text{if } \text{rep}(S) = \emptyset \\
\text{and } \text{rep}(S) = \emptyset &
\end{cases}
\end{align*}
\]

Figure 1: From Focal to OCaml
\[ S_\Gamma \cap \text{mod} = \begin{cases} \text{Module } \text{mod}(s) \mid \text{rep}_{\Gamma}^{\text{param}} \mid \text{param} <_{\cdot} \text{sig}(s) \mid \text{rep}_{\Gamma}^{\text{inh}} \mid \text{I}_{\Gamma}^{\text{thm}} \mid \text{I}_{\Gamma}^{\text{gen}} \mid \text{R}_{\Gamma}^{\text{thm}} \mid \text{I}_{\Gamma, \text{R}}^{\text{gen}} \cap_{\text{dep}} \\ \text{End } \text{mod}(s) \end{cases} \]

\[ [S_{\Gamma}]^{\text{mod}} = \left\{ \begin{array}{ll} \text{Definition } x_{\text{gentyp}} := \text{Sk} . x_{\text{gentyp}}, & \text{if } \text{dep}_{\text{typ}}(x) \neq 0 \\
0, & \text{otherwise} \end{array} \right. \]

\[ [S_{\Gamma}]^{\text{dep}} = \left\{ \begin{array}{ll} \text{Theorem } x : \text{prop} \text{ proof : } \text{proof}; & \text{if } \psi_i = \text{theorem } x : \text{prop } \text{ proof : } \text{proof}; \\
0, & \text{otherwise} \end{array} \right. \]

\[ [I_{\Gamma}^{\text{thm}}]^{\text{dep}} = \left\{ \begin{array}{ll} \text{Definition } x_{\text{gentyp}}(x_1 : [\tau_1] \Gamma \ldots (x_n : [\tau_n] \Gamma) := [\text{prop}] \Gamma \cdot \Gamma; & \text{if } \psi = \text{property } x : \text{prop } \text{ and } \text{dep}_{\text{typ}}(x) = \{x_1 : \tau_1, \ldots, (x_n : \tau_n)\} \\
0, & \text{otherwise} \end{array} \right. \]

\[ [I_{\Gamma}^{\text{gen}}]^{\text{dep}} = \left\{ \begin{array}{ll} \text{Axiom } x : [\text{prop}] \Gamma \cdot \Gamma; & \text{if } \psi = \text{property } x : \text{prop } \text{ and } \text{dep}_{\text{typ}}(x) = \{x_1 : \tau_1, \ldots, (x_m : \tau_m)\} \\
0, & \text{otherwise} \end{array} \right. \]

\[ \text{where:} \]

\[ \text{sig}(s) \] returns a module signature name from species name \( s \).

\[ \text{mod}(s) \] returns a module name from species name \( s \).

\[ \text{fun}(S) \] returns the set of functions \( \phi \) of species \( S \).

\[ \text{prop}(S) \] returns the set of properties \( \psi \) of species \( S \).

\[ \text{dep}_{\text{typ}}(x) \] returns the set of properties of \( \psi \) of species \( S \).

\[ \text{dep}_{\text{thm}}(x) \] returns the set of couples \( (x_i : \tau_i) \), where \( x_i \) is either a function or a property and where \( \tau_i \) is its type, and on which the statement of property of name \( x \) depends.

\[ \text{dep}_{\text{prop}}(x) \] returns the set of couples \( (x_i : \tau_i) \), where \( x_i \) is either a function or a property and where \( \tau_i \) is its type, and on which the proof of property of name \( x \) depends, its possible definition excluded.

\[ \text{dep}(x) = \text{dep}_{\text{typ}}(x) \cup \text{dep}_{\text{thm}}(x). \]

\[ <_{\cdot} \] sorts the set of properties from \( [I, M]^{\text{gen}}_{\Gamma, \cdot} \) s.t. \( \forall j > i, \psi_j \notin \text{dep}(\psi_i). \)

Figure 2: From Focal to Coq