Certified mathematical hierarchies: the FoCal system.

Virgile Prevosto[1]

Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
prevosto@mpi-sb.mpg.de

Abstract. The focal language (formerly Foc) allows a programmer to incrementally build mathematical structures and to formally prove their correctness. focal encourages a development process by refinement, deriving step-by-step implementations from specifications. This refinement process is realized using an inheritance mechanism on structures which can mix primitive operations, axioms, algorithms and proofs. Inheritance from existing structures allows to reuse their components under some conditions, which are statically checked by the compiler.
In this paper, we first present the main constructions of the language. Then we show a shallow embedding of these constructions in the Coq proof assistant, which is used to check the proofs made in Focal. Such a proof can be either an hand-written Coq script, made in an environment set up by the Focal compiler, or a Coq term given the zenon theorem prover, which is partly developed within Focal. Last, we present a formalization of focal structures and show that the Coq embedding is conform to this model.

Keywords. specifications, proofs, inheritance, refinement, types, Focal, Coq

1 Introduction

The main goal of the Focal language [1], formerly known as FoC, was to provide a convenient framework to develop certified computer algebra libraries. The building blocks of a Focal program are the *species*, which represent mathematical structures, with their primitive operations, their axioms, and possibly some derived functions and theorems proved from the axioms. Moreover, a *species* can be built upon existing ones, inheriting their operations and properties, as in object-oriented programming. The Focal compiler generates two outputs: the first one, which will not be mentioned any further in this paper, is an ocaml [2] file which reflects the computational part of the Focal source. The second one is a Coq file which can be verified by the Coq proof-assistant [3].

In this paper, we briefly present the main constructions of Focal (section 2), and their representations in Coq (section 3). Section 4 describes how to make a proof within Focal, either interactively or with the help of the zenon theorem

Dagstuhl Seminar Proceedings 05021
Mathematics, Algorithms, Proofs
http://drops.dagstuhl.de/opus/volltexte/2006/274

prover (distributed with Focal). Last, we focus in section 5 on the mixDrecs structures, which are direct representation of *species* in Coq (in contrast with the translation of Sect. 3, which is much more operational).

2 The Focal Language

2.1 Species

As we have just said, a Focal *species* represents an algebraic structure, such as groups or rings. Each species is composed of *methods*, identified by their names. There are three kinds of method: the carrier type, the programming methods and the logical methods.

Carrier: It is the type of the elements manipulated by the algebraic structure. It is introduced by the keyword **rep**. Each species must have an unique carrier.

Programming methods: They represent constants and operators.

Logical methods: Such methods represent the properties of the programming methods. Following the Curry-Howard isomorphism [4], the type of such a method is a statement, while its body is a proof. Statements are constructed over boolean expressions with the usual connectors (**and**, **or**, →, **not**) and universal (**all**) and existential (**ex**) quantification over a Focal type (including the carrier type of the species). The language used for the proofs will be presented below in section 4.1.

Moreover, each method can be either *declared* or *defined*. Declared methods represent the primitive operations of the structure, as well as its axioms. Defined methods represent derived operations, and (some of) the theorems that can be proved inside the structure. The carrier itself can also be declared (in which case it is only an abstract data-type), or defined, that is bound to a concrete type.

To clarify these different kinds of methods and give an example of Focal syntax, we can define a species representing groups:

```
species additive_group =
  rep;
  sig plus in self → self → self;
  property assoc: all x y z in self,
    self!plus(x,self!plus(y,z)) = self!plus(self!plus(x,y),z);
  sig opp in self → self;
  sig zero in self;
  property plus_opp: all x y in self,
  self!opp(self!plus(x,y)) = self!plus(self!opp(x), self!opp(y));
  property opp_opp: all x in self, self!opp(self!opp(x)) = x;
  let minus(x,y)= self!plus(x,self!opp(y));
  let id(x)=self!minus(x,self!zero);
  theorem minus_opp:
    all x y in self, self!minus(x,y) = self!opp(self!minus(y,x))
   proof: by plus_opp, opp_opp def minus;
  ...
end
```

First, there is the declaration of the abstract carrier **rep**. Then we give the **sig**nature of *plus*, a binary operation over **self**: **self** refers to the current species being defined (hence the method calls have the form **self**!plus), and, as often in mathematics, we identify the whole structure and its carrier. After that, we state a logical **property** of *plus* (namely that it is associative). In addition, we declare a unary operation *opp* together with some of its properties. We also *define* (through the **let** keyword) new operations, *minus*, from *plus* and *opp*, and id from minus and zero. In addition, we state a **theorem** about *minus* which can be derived from the properties of *plus* and *opp* and the definition of *minus*.

2.2 Inheritance

In mathematics, we usually do not define an algebraic structure "from scratch", as we have just done for the groups. On the contrary, we build a new structure by refining pre-existing ones with new axioms and/or operators. This is also the case for FOCAL, where a species can inherit from previous ones. For instance, given a species *monoid* with an associative operation *mult* and a neutral element *one*, we can define the species ring as follows:

```
species ring inherits group, monoid =
  property distrib: all x y z in self,
    self!mult(x,self!plus(y,z)) = self!plus(self!mult(x,y), self!mult(x,z));
  let zero = self!minus(self!one, self!one);
  ...
end
```

ring has all the methods that were present in group and monoid. In addition, it declares a new method distrib, and provides a definition for zero, which was only declared in group. In addition, a new species can of course directly define a new method. It can also redefine a previously defined method. For instance, if we want to build the ring $\mathbb{Z}$ with a carrier set to the native int type of FOCAL, it would be far more efficient to define zero as 0 instead of the generic definition provided in ring. On the other hand, the type of the methods must remain the same. This constraint guarantees that the new species is a particular instance of the species it inherits from. Similarly, in case of multiple inheritance, the methods with a same name in the two parents must have the same type. If several methods are defined, we select the definition coming from the rightmost species in the **inherits** clause. This is also true for the carrier, whose implicit name is **rep**.

2.3 Parameters

Another important feature is the parametrization of a species by another one. For instance, we can define univariate polynomials over an arbitrary ring as follows:

```
species univariate_polynomial (a is ring) inherits ring =
rep = list(a * int);
let zero = Nil;
let plus(x,y) =
  match(x,y) with
   | Nil, y → y | x, Nil → x
   | Cons((coeffx,degx) as monomialx,tailx),
            Cons((coeffy,degy) as monomialy,taily) →
              if degx = degy then
                if not(a!plus(coeffx,coeffy) = a!zero) then
                  Cons((a!plus(coeffx,coeffy),degx),
                      self!plus(tailx,taily))
                else self!plus(tailx, taily)
              else . . . ;
end
```

In the body of the species polynomial, a represents a ring. In particular, we can call any method from ring on it, such as a!plus in the definition of the addition of two polynomials. We also define the carrier of polynomial in term of the carrier of a (again, a!**rep** is identified with a itself in a type). Namely, a polynomial is a list of pairs composed of a coefficient and a degree.

2.4 Collections and Interfaces

In the definition of polynomials above, a should represent any instance of ring in which all the methods are defined (*i.e.* in which all functions have an implementation and all properties are proved). Indeed, polynomial can call any method whose name appears in ring, and a has to provide an implementation for it. On the other hand, since we allow redefinitions, polynomial itself can not make any assumption on the particular form of these implementations, even for methods that are defined in ring itself. Indeed, we can instantiate a with a structure that redefines it. These points lead to two dual notions in Focal:

- Each species must implicitly define an *interface*, obtained by replacing its defined methods by the corresponding declarations. Parameters are then seen through the interface of the species they are required to implement.
- When a species has all its methods defined, we can choose to transform it into a *collection*. Only collections can be used to instantiate parameters. Moreover, a collection is itself seen through its interface. In particular, its carrier is abstracted, in order to preserve possible invariants (for instance, polynomials are not arbitrary lists: they are ordered according to the degree, and we don't have coefficients equal to a!zero).

2.5 Dependencies

As we can see from the examples above, the methods of a given species are not independent from each other. Dependency analysis plays an important role in

Focal compilation. It is presented in detail in [5], but we give an overview of the issues in this paper. Intuitively, a method m of a species s depends upon a method m' if there is a call to **self**!m' in m. Such a call can occur in the type of m (with a reference to the carrier) as well as in its body. Of course, statements and proofs have dependencies too. The first point of dependency analysis is to avoid cycle of dependencies between the methods of s outside of definitions explicitly flagged as recursive, which lead to proof obligations in order to ensure termination.

When it comes to proofs, we have to distinguish between two kinds of dependencies. There is a *decl*-dependency from m upon m' if only the type -or the statement- of m is needed to check the proof. On the other hand, there is a *def*-dependency if we have to unfold the definition of m' during the proof. Def-dependencies might also appear in statements, as shown by the following case:

```
species s =
rep = nat;
property foo: all x in self, nat_plus(x,1) > 0
end
```

The statement of foo is correct only if we know that the carrier is bound to nat. Otherwise, we can not give an argument of type **self** to nat_plus (assuming nat_plus is the addition over nat). However, such def-dependencies are rejected by Focal. Indeed, this would prevent us from deriving an interface for species such as s, since we can not abstract their carrier.

When it comes to inheritance, def-dependencies have another drawback. Indeed, if we want to redefine a method m', all the proofs that def-depends upon m' will become invalid in the new species. Focal erases them during inheritance resolution, but it means that we'll have to do them later for the new version of m'. Since doing a proof can be quite time-consuming, it is important to minimize the number of erasure during a development. Some solutions to this issue are proposed in [6], but this topic seems to be worth further investigations.

3 Coq Translation

Focal uses Coq as a back-end to verify the proofs made in the species. In this section, we present briefly the compilation of Focal constructions into Coq. A complete definition of the translation is to be found in [7], as well as proofs that it returns well-typed Coq term (provided the Focal development is itself well-formed of course).

3.1 Overview of Code Generation

The representation of a species is encapsulated in a Coq module. Such a module is composed of the following parts:

- First, we find a record type corresponding to the interface of the species. This type has one field for each method in the species, regardless of its origin (inherited or not).
- Declared methods are represented by variables (also called **Parameter**s in Coq terminology for modules).
- If a method is (re)defined in the species body, a corresponding *method generator* is given on the Coq side. It is a generic definition obtained by performing some abstractions, which allows to simulate the *late binding* mechanisms of Focal, that is the fact that a call to **self**!m must always use the latest definition available for m. They are described more precisely in the next section.
- All defined methods are represented by definitions that are local to the module. Such definitions are obtained by applying the corresponding method generator (that may come from a preceding module) to the arguments specific to the current module. This allows to let Coq verify the correction of Focal inheritance resolution.
- When we want to define a collection from a completely defined species, we just have to bind together these local definitions into a record of the type given at the beginning of the module.
- Inheritance is handled by defining coercions between the record types. Since a species can not change the types of inherited methods, this consists mainly in forgetting the fields that are not present in the old record. Once the coercion is defined, we can rely on the implicit coercions mechanism of Coq.
- Last, parameters are represented as abstractions in the definition of the record type and the method generators. They also give rise to Coq parameters in order to be able to perform the local definitions mentioned above.

As an example, we can see how polynomial is represented in Coq:

```
Module Polynomial.

Record species (a: Ring.species): Type :=
  { rep:> Set; zero: rep; plus: rep → rep → rep; . . . }.

Parameter self_a: Ring.species.

Definition rep_gen: Set := fun (a: Ring.species) ⇒ list (a*int).

Local self_rep: Set := rep_gen self_a.

. . .

Coercion polynomial_to_ring :=
  fun (a: Ring.species) ⇒ fun (p:species a) ⇒
    Ring.build_species (rep p) (zero p) (plus p) . . . .
```

First, we define a new record type, parameterized by a ring. The declaration of the rep field indicates an implicit coercion from such a record to its carrier (corresponding to the syntax of Focal types). Then we declare a parameter of the whole module of type **Ring**.species. After that, we find the method generator of rep, abstracted with respect to a, and the local definition of the carrier, obtained by applying the generator to the parameter self_a. Last, after all methods have been treated (in an order compatible with the dependencies between them), we find a coercion from the species record to **Ring**.species, since polynomials inherits from ring.

3.2 Method Generators

Let us have now a closer look to the *method generators*. As already said, their main purpose is to handle *late binding*. Namely, take a method m is defined in a species s, and decl-depends upon a method m', which is also defined in s. Then, if s' inherits from s and redefines m' but not m, we want that the method m of s' use the new definition of m'. Basically, to obtain a method generator, we just have to abstract the body of m with respect to m', and apply it to the two different definitions of m' in s and s'. However, things are a little bit more complicated. First, we have to take into account the def-dependencies of m. By hypothesis, we can not abstract the definition of m with respect to them. On the contrary, if m def-depends upon d, we must provide the definition of d in s. To achieve that, we can use the corresponding method generator (from s or one of its ancestor), but d has itself dependencies, that must be taken into account when computing the abstractions of the method generator of m. Likewise, if m' type is a statement, then it might also have dependencies, with respect to whom the method generator of m must be abstracted. All this makes the computation of the necessary abstraction a bit tricky, but in the end we come up with well-typed Coq terms that can be used to generate a method m in s and all its descendant (unless m is redefined, of course). For instance, the method generator corresponding to the minus and minus_opp methods of section 2 are the following:

```
Definition minus_gen:=
  fun(rep: Set)⇒ fun(plus: rep → rep → rep)⇒ fun (opp: rep → rep)⇒
    fun (x,y:rep) ⇒ plus(x,opp(y)).
Definition minus_opp_gen:=
  fun(rep:Set)⇒ fun (plus:rep → rep → rep)⇒ fun (opp: rep → rep)⇒
    let minus = (minus_gen rep plus opp) in
      fun (plus_opp: ...) ⇒ ...
```

minus_gen is simply an abstraction over rep, plus and opp. minus_opp_gen contains in addition a definition of minus, obtained by applying minus_gen to the appropriate representations of rep, plus and opp in the context of the theorem generator.

4 Making Proofs in Focal

The preceding section gives an overview of the translation of a Focal species into Coq so that Coq can verify the proof of the theorems made in Focal. It is now time to see how such a proof can be written. There are two possibilities. The first one is to give directly a Coq script or a Coq term. The second one relies on the zenon theorem prover, due mostly to Damien Doligez, which is developed as part of the Focal project. We concentrate on the latter one in the remaining of the section.

4.1 The proof language

The proof language used to communicate with zenon is quite similar to the one proposed by Lamport in [8]. Namely it consists on a hierarchy of proof steps, where each step represents a lemma which can be derived from the steps of the inferior levels. We can take as an example a theorem from the Focal standard library. This theorem is given in the meet_semi_lattice species. This species has a primitive operator, inf, which is associative, commutative and idempotent, and the theorem states that the relation order_inf derived from inf by $\mathit{order\_inf}(x, y) \triangleq \inf(x, y) = x$ is transitive (in fact it is a partial ordering).

```
theorem order_inf_is_transitive : all x y z in self,
    !order_inf(x,y) → !order_inf(y,z) → !order_inf(x,z)
proof:
  <1>1 assume x y z in self
            H1: self!order_inf (x, y)
            H2: self!order_inf (y, z)}
      prove self!order_inf (x, z)
    <2>0 prove self!equal (x, !inf (x, y))
      by <1>:H1 def self!order_inf
    <2>1 prove !equal (x, self!inf (x, !inf (y, z)))
      by <2>0, <1>:H2, ... def self!order_inf
    <2>2 prove !equal (x, self!inf (self!inf (x, y), z))
      by <2>1, self!inf_is_associative, self!equal_transitive ...
    <2>3 prove !equal (x, self!inf (x, z)) by <2>2, <2>0, ...
    <2>4 qed by <2>3 def self!order_inf
  <1>2 qed;
```

The first step of the proof is a transcription of the original statement: we introduce three variables x, y and z, and two hypotheses H1 and H2, and claim that we can prove **self**!order_inf(x, z) from that. To proceed, we have 5 sub-steps at level 2, <2>0 to <2>4. The first three ones state some intermediate lemmata, while the **qed** in the last one says that we have enough results to achieve the proof of the goal at level 1.

The **by** directives give some hints on how the intermediate lemmata can be proved by zenon. They also allows Focal to compute the decl- and def-dependencies of the theorem. Namely, we use here the definition of order_inf (in steps <2>0 and <2>4), as well as two axioms (in step <2>2), the associativity of inf and the transitivity of equal, the equality of the lattice. Note that we can also used the hypotheses made in <1>1, as it is done in step <2>0, or even steps that have already been proved in the same branch of the proof tree: step <2>4 relies on step <2>3, which uses itself <2>2 and <2>0.

The whole proof is not given to zenon directly. Rather, every single step results in a new formula, which takes into account the **by** directive, and each of these formulæ is given separately to the prover.

4.2 The zenon theorem prover

Let us now briefly describe zenon itself. As we have already said, it is developed in the FOCAL project by Damien Doligez and it comes with the FOCAL distribution. It is a tableaux-based first-order theorem prover, which can deal with equality (although this feature is not yet really used by FOCAL since most species rely on an abstract equivalence relation and not on *the* structural equality over terms). Moreover, zenon produces either a COQ script or a COQ term as proof, which can easily be checked by COQ.

Some few words may be added on the relationship between zenon and the dependency analysis of FOCAL. Namely, when a method m of **self** is given in a **by** directive, FOCAL interprets that as a decl-dependency upon m, while strictly speaking, this only means that zenon *can* use m, but there is no obligation that m appears in the final proof. Thus, we have to add an artificial binding to m to ensure that the dependency analysis is consistent with the method generator. Moreover, zenon emits a warning in such a case, since unnecessary decl-dependencies may lead to a dependency cycle between theorems. In many cases, such situation could indeed have been prevented by a more careful analysis of what is really needed to do a particular proof.

This led to a somehow interesting issue: there are some statements for which zenon can find a proof only if another method is in the environment, but this method is not used in the final proof: its only role is seems to help zenon heuristics in making efficient choices. This point suggests that finding the appropriate environment in which an automated theorem prover can succeed on a given goal may be quite subtle and deserves some more investigations.

5 MixDrecs

MixDrecs have been introduced by Sylvain Boulmé in his PhD [9] in order to be able to reason about species within COQ. Indeed, the translation presented in section 3 destroys completely the structure of a FOCAL species: we have a set of method generators and local definitions, but there is no object to represent whole species nor the relations between species. From an operational point of view, this is satisfactory, since the dependency analysis and inheritance resolution are performed by the FOCAL compiler itself, and COQ is left with smaller terms to type-check. On the other hand, mixDrecs can be seen as a denotational semantics for species. Each mixDrec is a direct representation of a species in COQ, so that the main properties of species can be established formally within COQ logic [10]. The relations between both translations have been investigated in [11,7]. We give a brief overview of these results in the remaining of this section.

5.1 Definition

Intuitively, a mixDrec is a tree structure, whose nodes correspond to declared and defined methods. Each node can depend on all of its predecessors. There are three kinds of nodes:

- empty nodes, at the bottom of the tree.
- abstract nodes, corresponding to declared methods. They have a name and a type, and only one successor.
- manifest nodes, corresponding to defined methods. They have in addition a definition, and each of them has two successors. In the first branch, the definition is hidden, while it is accessible in the second branch, so that some nodes can be defined on the second branch, but have to kept abstract on the first one: this correspond to the def-dependencies between methods.

For instance, we can build the mixDrec corresponding to the additive_group species seen above:

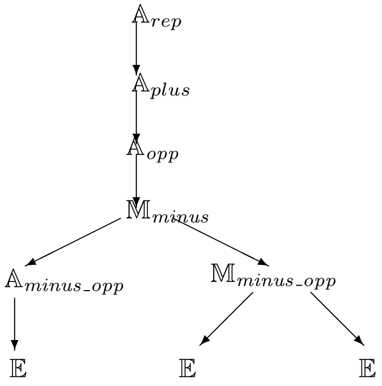

Fig. 1. a mixDrec for groups

In figure 1, $\mathbb{A}$ represent abstract nodes, $\mathbb{M}$ manifest nodes and $\mathbb{E}$ empty nodes. Types and definitions in the label of the nodes have been omitted. In this figure, the first three nodes are abstract. The fourth one is defined and has two sons. In the left-hand side, we do not take into account the definition of *minus*. Thus, *minus_opp* must also be abstracted, because of its def-dependency. On the right-hand side, we know the definition of *minus*. *minus_opp* can thus be defined.

A *signature* can be attached to each mixDrec. It corresponds to the interface of a species, and is simply the list of the fields appearing in the mixDrec with their types. An important notion at this level is the notion of *subsignature*, $s_1 : \succ s_2$. It means that s_1 as at least the same fields as s_2, but not necessarily in the same order (one must respect dependencies, though. For instance, we can not swap minus_opp and minus in the signature corresponding to additive_group).

At the mixDrec level, there are two main operations. The first one is the embedding of a mixDrec M of signature s_1 in s_2 when $s_2 : \succ s_1$: $\Uparrow_{s_2} M$. The nodes of M are reordered according to s_2 and the missing fields are added as abstract nodes. The second operation is the fusion of two mixDrecs M_1 and M_2 sharing the signature s: $M_1 \oplus M_2$. In this case, if two corresponding are manifest in both M_1 and M_2, the one of M_2 is selected, and we continue the fusion using the abstract branch of M_1.

5.2 MixDrecs and species

Briefly speaking, any well-formed FOCAL species s can be transformed in a mixDrec $\ll s \gg$ of signature $\llbracket s \rrbracket$. Indeed, since we do not allow cycle of dependencies, once inheritance has been resolved, we can order the methods of s according to these dependencies.

Moreover, inheritance resolution itself can be expressed through the basic mixDrec operations presented above. More precisely, the following properties hold:

- if s inherits from s', $\llbracket s \rrbracket :\succ \llbracket s' \rrbracket$
- Simple inheritance, s inherits from s_1.

$$\text{Theorem: } \ll s \gg \oplus (\Uparrow_{\llbracket s \rrbracket} \ll s_1 \gg) = \ll s \gg$$

- Multiple inheritance, s inherits from s_1 and s_2, without adding new methods

$$\text{Theorem: } (\Uparrow_{\llbracket s \rrbracket} \ll s_2 \gg) \oplus (\Uparrow_{\llbracket s \rrbracket} \ll s_1 \gg) = \ll s \gg$$

5.3 MixDrecs and Method Generators

Let us now examine the relationships between method generators and MixDrecs. Namely, given a species s and a method x defined in the body of s, we can "extract" the method generator of x from $M = \ll s \gg$ in the following sense. First, we can define the minimal definition context for x in M, $\widetilde{\mathbf{\Gamma_M \vdash x}}$. Intuitively, this is the most abstract branch of M in which x is defined. $\widetilde{\mathbf{\Gamma_M \vdash x}}$ is obtained from M by replacing any manifest node for which x appear as manifest in the abstract branch by an abstract node. Moreover, since we are only interested in x, we replace the successors of the x nodes by empty nodes. In other words, the only manifest nodes in $\widetilde{\mathbf{\Gamma_M \vdash x}}$ correspond to the def-dependencies of x.

We can also define an equivalence relation between mixDrec, $\leftrightarrowtriangle$, which allows us to swap two independent fields. Then it is possible to prove that the method generator of x in s corresponds to the smallest mixDrec equivalent to $\widetilde{\mathbf{\Gamma_M \vdash x}}$. This shows that the construction of method generators is correct with respect to mixDrecs' semantics.

6 Conclusions

To sum up, we can say that FOCAL languages has yet all the ingredients needed to build a hierarchy of mathematical structure and express and prove their properties as well as implement algorithms over them. The COQ translation gives strong guarantees on the correction of the proof made by the zenon theorem prover. Last, FOCAL semantics has solid grounds in the form of the mixDrec model.

References

1. The Focal development team: Focal, version 0.2 Tutorial and reference manual. LIP6 – INRIA – CNAM. (2004) Distribution available at: `http://focal.inria.fr`.
2. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system, release 3.08. (2004)
3. The Coq Development Team: The Coq Proof Assistant Reference Manual Version 8. INRIA-Rocquencourt. (2004)
4. Howard, W.: The formulae-as-type notion of construction. In: To H.B. Curry, Essays on combinatory logics, lambda calculus and formalism. Academic Press (1980) 479–490
5. 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.
6. Prevosto, V., Jaume, M.: Making proofs in a hierarchy of mathematical structures. In: Proceedings of Calculemus. (2003)
7. Prevosto, V.: Conception et Implantation du langage FoC pour le développement de logiciels certifiés. Thèse de doctorat, Université Paris 6 (2003)
8. Lamport, L.: How to write a proof. research report, Digital Equipments Corporation (1993)
9. Boulmé, S.: Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel. Thèse de doctorat, Université Paris 6 (2000)
10. Boulmé, S.: Mixdrec definition (coq development). `http://www-lsr.imag.fr/Les.Personnes/Sylvain.Boulme/focal.html` (2000)
11. Prevosto, V., Boulmé, S.: Proof contexts with late binding. In: TLCA. Volume 3461 of LNCS. (2005) To appear.