Proof Contexts with Late Binding

Virgile Prevosto[1] and Sylvain Boulmé[2]

[1] Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85 – 66123 Saarbrücken – Germany
prevosto@mpi-sb.mpg.de
[2] LSR-IMAG
681, rue de la Passerelle BP-72 – 38402 St-Martin D'Hères – France
Sylvain.Boulme@imag.fr

Abstract. The FOCAL language (formerly FoC) allows one to incrementally build modules and to formally prove their correctness. In this paper, we present two formal semantics for encoding FOCAL constructions in the COQ proof assistant. The first one is implemented in the FOCAL compiler to have the correctness of FOCAL libraries verified with the COQ proof-checker. The second one formalizes the FOCAL structures and their main properties as COQ terms (called mixDrecs). The relations between the two embeddings are examined in the last part of the paper.

1 Introduction

As software applications are growing in size and complexity, it becomes necessary to provide strong, machine-checked guarantees of their correctness. FOCAL is a language (formerly called FoC) initially designed to develop certified computer algebra libraries. In short, a component of a FOCAL library can mix abstract specifications, implementations of operations and proofs that the implementations satisfy their specifications. FOCAL encourages a development process by refinement: concrete implementation can be derived step-by-step from abstract specifications. The validity of each step is bound to various constraints. Some of them can be checked by the compiler, but others lead to proof obligations. These proof obligations can be discharged by an automatic prover or directly by the developer. In FOCAL, refinement is realized through a kind of inheritance mechanism. The correctness of the libraries is verified with the COQ proof assistant [21].

This paper introduces two semantics for the FOCAL constructions. The first one is a *very shallow embedding* into COQ using mainly λ-abstractions. This semantics reflects the current implementation of the FOCAL compiler. However, it is purely operational and does not take into account the global structure of FOCAL libraries. We have defined a denotational semantics that associates to each component of a FOCAL library a COQ type called *mixDrec*. These mixDrecs allow us to state formally in the COQ logic the main properties of these structures. In this paper, we examine the relations between these two semantics.

Let us now introduce a flavor of the main FOCAL concepts. The building blocks of a computer algebra library are algebraic structures. At first glance, an algebraic structure can be seen as a set of functions (and constants, *i.e.* functions with 0 argument) and properties. For instance, a group is built upon a carrier set, **rep**. It has a binary operation over **rep**, *plus*, which is associative and has a neutral element 0. It also provides a unary operation *opp* such that $plus(x, opp(x)) = 0$ etc. As we may notice from this example, each component of such a structure must be expressed in a certain context, where some other elements are present. For instance, *plus* or 0 can only be introduced after **rep** has been specified. This can be captured by the notion of *dependent records* [7, 15, 6], that is records in which the type of each field might depend on the preceding ones. With such a construction, the notion of group is simply a record type, which can be informally represented as

$$\left\{\begin{array}{l} \mathbf{rep}\ :Set \\ plus\ :\mathbf{rep} \rightarrow \mathbf{rep} \rightarrow \mathbf{rep} \\ assoc{:}\forall x, y, z \in \mathbf{rep}, \\ \qquad plus(x,plus(y,z)) = plus(plus(x,y),z) \\ \qquad\qquad\qquad \ldots \end{array}\right\}$$

A given group, such as $(\mathbb{Z}, +, 0, -)$, is a particular instance of this type.

While these records can specify single mathematical structures, they do not capture the relations between them. Indeed, once we have groups, we can add a new operation *mult* (together with its properties) to obtain rings. Then it is possible to define integral domains as a special kind of rings with some additional properties. Thus record types must be extensible.

Furthermore, it is often possible to define some operations at an abstract level. For instance, subtraction can be derived from *plus* and *opp* for any group ($x - y = plus(x, opp(y))$), and by extension for any structure deriving from the groups, such as rings and domains: its definition can be given together with the specification of groups, and reused in any instance of a structure deriving from groups, as well as proofs of its properties (for instance $x - y = opp(y - x)$). Such refinements from one structure into another are very similar to the *inheritance* notion in Object-Oriented Programming (OOP).

In order to increase efficiency of programs, it is often interesting to refine a definition when deriving a new structure. If we want to implement $\mathbb{Z}/_{2\mathbb{Z}}$ with booleans, the generic definition of subtraction may be replaced by *xor*, and all other components of this structure have to use this new definition. This concept is known in OOP as *late binding*. However, on the contrary to OOP, definitions represent here (proofs of) theorems as well as functions. Mixing redefinitions with proofs and dependent types is the main issue of FOCAL inheritance semantics.

This paper is divided as follows. First, we present in the next section the main constructions of the FOCAL language. Then, we propose an embedding of these constructions in COQ, which is used to ensure the correctness of the proofs made within FOCAL (section 3). Section 4 introduces *mixDrecs*, which can be seen as partially defined (dependent) records, and hence constitute a good representation of mathematical structures. Last, sections 5 and 6 show the conformance of respectively the FOCAL language and its translation into COQ with respect to the mixDrecs model.

2 The Focal Language

2.1 Presentation

The main goal of the FoC project[3] was to design a new language for computer algebra having both a clear semantics and an efficient implementation – *via* a translation to OCAML. The resulting language incorporates functional and some restricted object-oriented features. For the certification part, the language provides means for the developers to write formal specifications and proofs of their programs and to have them verified with COQ. The FOCAL library, mostly developed by Rioboo [5, 18], implements mathematical structures up to multivariate polynomial rings with performances comparable to the best CAS in existence.

Designing our own language allows us to express more easily than in a general purpose language some very important concepts of computer algebra, and in particular the *carrier type* of a structure. We have also restricted object-oriented features to what was strictly necessary and avoided constructions which would have hindered our intention to prove the correctness of programs [4].

2.2 Main Constructions of the Language

We describe here informally the main features of the language, and give an overview of its syntax. More in-depth descriptions can be found in [17, 22].

Species. The main nodes of the FOCAL hierarchy are called **species**. They can be compared to the *classes* of an object-oriented language. Each species is composed of *methods*, identified by their names. A method can be either *declared* or *defined*. Declared methods represent the primitive operations of the structure, as well as its specifications. Defined methods represent all the operations that have received an implementation so far, and all theorems that have been proved. Moreover, we distinguish three kinds of methods:

Carrier: It is the type of the elements manipulated by the algebraic structure. A declared carrier is an abstract data type. A defined carrier is bound to a concrete type. For instance, polynomials may be represented by a list of pairs, the first component being the degree and the second one the coefficient.

Programming methods: They represent the constants and the operators of the structure. Declared methods of this category are simply signatures. The definitions are written in a language close to the functional core of ML.

Logical methods: Such methods represent the properties of the programming methods. Following the Curry-Howard isomorphism, the type of such a method is a statement, while its body (when it is defined) is a proof.

To give an example of FOCAL syntax, we express below the specification of groups seen in section 1 as a species (due to space constraints, the statements

[3] `http://focal.inria.fr`

of the properties are given informally in a comment):

```
species group =
  rep;
  sig plus in rep → rep → rep;
  property assoc: (* x+(y+z)=(x+y)+z *);
  sig opp in rep → rep; sig zero in rep;
  property plus_opp: (* -(x+y)=(-y)+(-x) *);
  property opp_opp: (* -(-x)=x *);
  let minus(x,y)= plus(x,opp(y));
  let id(x)=minus(x,zero); ...
  theorem minus_opp: (* x-y=-(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 **rep**. After that, we state a logical **property** over *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*.

Inheritance. the species *group* above is defined "from scratch", by providing the complete list of its methods. It is also possible to build species using *inheritance*. A new species may inherit the declarations and definitions of one or several species. 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: (* x*(y+z)=(x*y)+(x*z) *);
  let zero = minus(one, one); ...
end
```

The new species can use all the methods of its parents, regardless of their origin. It can declare new methods (such as *distrib*) or directly define them, provide a definition for previously declared methods (such as *zero*), or even redefine methods. On the other hand, the type of the methods (or their statement in the case of logical methods) must remain the same. This constraint guarantees that if a species s_2 inherits from a species s_1, then any instance of s_2 is also an instance of s_1. 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**.

Collections and Interfaces. The *interface* of a species s is obtained by hiding the body of all defined methods of s (while keeping the corresponding declarations). It can be seen as the type of s. All implementations of s must adhere to this interface, while they are free to modify some of the definitions of s. A **collection** is an instance of a completely defined species (*i.e.* in which every method is defined). Users of a collection access it only through its interface.

2.3 Dependencies

A method m_1 of a species s can *depend* on a method m_2 of s. The *group* species gives various examples of dependencies:

- The declaration of *plus* depends on **rep**.
- The statement of *assoc* depends on **rep** and *plus*.
- The body of *minus* depends on *plus* and *opp*, and also implicitly on **rep**, since some subexpressions have type **rep**. Its type (**rep** → **rep** → **rep** as it can be inferred by the compiler), depends only on **rep**.
- *id* depends on *minus*, *zero* (and **rep**).
- The statement of *minus_opp* depends on *minus*, *opp*, and **rep**. The proof itself depends in addition on *plus_opp* and *opp_opp*, and on the *definition* of *minus*.

As shown by this last example, we have two kinds of dependencies. There is a *decl-dependency* on x when we need to rely on the declaration of x (*i.e.* its type or its statement). There is a *def-dependency* on x if we need to unfold its exact definition in order to type-check an expression.

A def-dependency may occur not only in the body of a method, but also in the statement of a property or of a theorem. In this case, it is not possible to extract an interface for the species. Consider for instance the following code:

```
species bad =
  rep = int;
  property id: all x in rep, equal(int_plus(x,0),x);
 end
```

The statement of *id* can only be well-typed in a context where **rep** is bound to the *int* type. Otherwise, it is not even possible to type-check the expression *int_plus*(x,0). Allowing such def-dependencies in types would have a major drawback with respect to redefinition. For instance, redefining **rep** requires at least to remove completely the methods with a def-dependency upon it in their type, such as *id*, because their type could not be type-checked anymore. But then, the resulting species does not offer the same functionalities as its parent (*id* does not exist anymore). To avoid this problem, FOCAL does not allow species without a correct interface: species such as *bad* are rejected.

2.4 Inheritance Resolution and Normal Form

If we redefine a method x, all methods that def-depend upon x are no longer accurate, so that we have to erase their definition. In [17] and [16], Doligez and Prevosto describe an algorithm that performs inheritance resolution and takes into account def-dependencies. They use the following notations:

– The *decl*-dependencies of a method x in a species s are written $\wr x \int_s$.
– The *def*-dependencies of a method x in a species s are written $\wr\!\wr x \int\!\!\int_s$.
– Last, it is required that any species s can be put in *normal form*.

A species in normal form has no **inherits** clause, and its methods are ordered according to their dependencies. Namely, a method can only depend on the preceding ones. Hence, a species in normal form is nothing more than an ordered sequence of methods. It is unique modulo reordering of independent methods. In this paper, we will sometimes implicitly identify such a sequence with a species.

Given a species defined by **species** s **inherits** $s_1, \ldots, s_n = \phi_1 \ldots \phi_m$ **end**, the main properties of the algorithm are the following:

1. If the algorithm succeeds then its result $norm(s)$ is in normal form.
2. If a normal form equivalent to s exists then the algorithm succeeds.
3. $norm(s)$ contains the *newest* definition of each method x. This is the last definition of name x found, starting from $norm(s_1)$ and ending with ϕ_m.
4. Only a minimum set of definitions is erased: for each method x declared in $norm(s)$ but defined in one of the s_i, there exists $y \in \wr\!\wr x \int\!\!\int_{s_i}$, such that y is only declared in $norm(s)$, or the definition of y is not the same in s and s_i.

3 Minimal Environment and Method Generators

Dependencies also play an important role during the translation of a species s in the Coq language. Each method x of s introducing a new definition is transformed into a well-typed Coq-term. In addition, we want to reuse this term in any species or collection deriving from s (in which x is neither redefined nor erased), *and* the names of $\wr x \int_s$ to be bound to the newest definitions available, hence emulating a late-binding mechanism. The main issue is that a definition can not come alone. Indeed, we have to take dependencies into account and to provide a whole environment in which the definition can be type-checked. For instance, if we consider again the *group* species, the theorem *minus_opp* has to be verified in an environment containing at least the following methods:

rep; **sig** *plus* **in rep** → **rep** → **rep**; **sig** *opp* **in rep** → **rep**;
property *plus_opp*: ...; **property** *opp_opp*: ...;
let *minus* $(x,y) = plus(x,\ opp(y))$;

Indeed, *minus_opp* def-depends upon *minus* and decl-depends upon *plus_opp* and *opp_opp*. Then in order to define minus (and to state *plus_opp* and *opp_opp* by the way) we have to include the declarations of *plus* and *opp*. Last, **rep** must also be present, since all the functions manipulate elements of the carrier type.

To achieve that, the *decl*-dependencies of x can be replaced by abstractions. This leads to the notion of *method generators*. To obtain the method x for a given collection, we only have to apply the generator of x to the appropriate methods. In our example, we can apply the generator of *minus_opp* to any implementation of **rep**, *plus* and *opp*, together with proofs of *plus_opp* and *opp_opp*.

By definition, *def*-dependencies can not be abstracted. We must put their definition in the environment. These definitions have themselves their own environment constraints: if $y \in \llbracket x \rrbracket_s$, the method generator must be abstracted with respect to the methods of $\lceil y \rfloor_s$, so that we can generate y itself.

In order to properly tackle this issue, we introduce the notion of *minimal environment* in which x can be defined. It is denoted by $s \Cap x$, and corresponds to the smallest subset of $norm(s)$ which is needed to both type-check x *and* have a well formed typing context. The formal definition of $s \Cap x$ relies on the notion of *visible universe* $\mid x \mid$ of a method x. It is the list of method names that will occur in $s \Cap x$. Namely, with $<_s^{def}$ the transitive closure of $\llbracket \cdot \rrbracket_s$ and $\mathcal{T}_s(x)$ denoting the type (or the statement for logical methods) of x in the species s, we have

Definition 1 (visible universe).

$$\frac{y \in \lceil x \rfloor_s}{y \in \mid x \mid} \qquad \frac{y <_s^{def} x}{y \in \mid x \mid} \qquad \frac{z <_s^{def} x \qquad y \in \lceil z \rfloor_s}{y \in \mid x \mid} \qquad \frac{z \in \mid x \mid \qquad y \in \lceil \mathcal{T}_s(z) \rfloor_s}{y \in \mid x \mid}$$

In the following, we write $norm(s)$ as an ordered list of methods Φ, where Φ is defined inductively as either the empty list $\emptyset$ or a non-empty list $\{y : \tau = e; \Phi'\}$. In this last case, τ is a type associated to the field name y, and e is $\perp$ when y is only declared, or else a term corresponding to the definition of y.

Definition 2 (minimal environment). *The minimal environment is defined with the following rules:*

$$\emptyset \Cap x = \emptyset \qquad \frac{y \notin \mid x \mid \qquad \Phi \Cap x = \Sigma}{\{y : \tau = e; \Phi\} \Cap x = \Sigma} \qquad \frac{y <_s^{def} x \qquad \Phi \Cap x = \Sigma}{\{y : \tau = e; \Phi\} \Cap x = \{y : \tau = e; \Sigma\}}$$

$$\frac{y \in \mid x \mid \qquad y \not<_s^{def} x \qquad \Phi \Cap x = \Sigma}{\{y : \tau = e; \Phi\} \Cap x = \{y : \tau = \perp; \Sigma\}}$$

On the Coq side, the method generator is obtained by mapping any declaration of $s \Cap x$ to a λ-abstraction, and any definition to a local binding, obtained by applying the corresponding method generator to the appropriate variables. In [16], we have shown that such a method generator is a well-typed Coq term. For instance, the method generators corresponding to the *minus*, *id* and *minus_opp* methods of our *group* species would be 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: ...) ⇒ ...
Definition id_gen:=
  fun(rep:Set)⇒ fun(zero:rep)⇒ fun(minus: rep → rep → rep)⇒
    fun (x:rep) ⇒ minus(x,zero).
```

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. *id_gen* is abstracted with respect to *minus* and *zero*. Since it does not rely on the definition of *minus*, there is no need to call *minus_gen*.

4 MixDrecs

Method generators provide a convenient way to represent the method definitions of a species s in Coq, and to reuse the corresponding code in each implementation of s. However this translation gives us only a set of method generators: we do not have any representation of s itself, nor of its relations with the other species. In his PhD [4], Boulmé verifies that relations between species can be described in Coq (and are thus compatible with the Coq logic, the calculus of inductive constructions). He proposes Coq constructions to represent species. This section presents these structures and their main properties using human-friendly notations (roughly speaking, inference rules below correspond to inductive definitions in Coq). The whole Coq development is available at `http://www-lsr.imag.fr/Les.Personnes/Sylvain.Boulme/focal.html`

4.1 Dependent Records

Collections can be represented as records, while their interface can be represented as a record type. Given a (countably infinite) set *Lab* of field names, a record is a function (with a finite domain of definition) which associates a definition to a field name. A record signature associates a type to a field name.

Definition 3 (Drecord signature).

$$\mathit{Rec} : \mathrm{List}(\mathit{Lab}) \rightarrow \mathit{Type} \qquad \frac{\textsc{Esig}}{\{\}_{\mathrm{s}} : \mathit{Rec}_{\emptyset}} \qquad \frac{\textsc{Csig} \quad T : \mathit{Type} \qquad a \notin l \qquad F : T \rightarrow \mathit{Rec}_l}{\{a : T; F\}_{\mathrm{s}} : \mathit{Rec}_{a;l}}$$

The function F in Csig expresses the dependencies of the remaining signature with respect to a. We build then a signature from right to left, that is, following the terminology of [15], right associative records. The definition of a Drecord follows the same rules, with the difference that the field a is now bound to an expression and not only to a type. The main operations over signatures are:

- A sub-signature relation between two signatures, noted $s_1 :\succ s_2$[4], which says that s_1 contains *at least* all the fields of s_2 with the same type, but not necessarily in the same order. Two *independent* fields can indeed be swapped.
- Single inheritance corresponds to the extension of a given signature s with new fields. These new fields can depend on the fields of s and thus are appended at the end of the signature.

[4] $:\succ$ corresponds to the usual subtyping relation $<$, but the direction of the relation makes "bigger" the signature which has more fields.

– Multiple inheritance corresponds to the fusion of two signatures s_1 and s_2, provided that the common fields have the same type in both signatures.

4.2 Partially Defined Drecords: MixDrecs

To represent *species*, we must be able to *mix* declared and defined fields. This is done in a structure called *MixDrec*. A mixDrec is a tree whose nodes are:

– Empty nodes, forming the leafs of the tree.
– Abstract nodes, corresponding to declared fields. An abstract node contains a name and its type, and has one son.
– Manifest nodes, corresponding to defined fields. Such a node contains a name x , its type and its definition. It has two sons. In the first one, x is abstracted, *i.e.* we only know its type. In the second one, we have access to the definition of x. Both sons contain the same fields in the same order, but some abstract nodes of the first son may be defined in the second. Intuitively, it corresponds to def-dependencies over x.

$$\begin{tikzcd}
 & & \mathbb{A}_{rep} \arrow[d] & & \\
 & & \mathbb{A}_{plus} \arrow[d] & & \\
 & & \mathbb{A}_{opp} \arrow[d] & & \\
 & & \mathbb{M}_{minus} \arrow[dl] \arrow[dr] & & \\
 & \mathbb{A}_{minus\_opp} \arrow[d] & & \mathbb{M}_{minus\_opp} \arrow[dl] \arrow[dr] & \\
 & \mathbb{M}_{id} & \mathbb{M}_{id} & & \mathbb{M}_{id}
\end{tikzcd}$$

Fig. 1. Tree-structure of a mixDrec

As an example, figure 1 represents the structure of the MixDrec representing groups, with $\mathbb{A}$ being abstract nodes and $\mathbb{M}$ manifest nodes. Due to lack of space, empty nodes (leafs of the tree) have been omitted, as well as some abstract fields. 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 other hand, *id*, which has only a decl-dependency upon *minus*, can be defined. On the right-hand side, we know the definition of *minus*. *minus_opp* can thus be defined. Since *id* does not depend upon *minus_opp*, both sons of $\mathbb{M}_{minus\_opp}$ are manifest nodes containing the definition of *id*.

More formally, the type of mixDrecs has three parameters: a list l of field names, a tree-structure p of type Pre_l (defined figure 2) and a Drecord signature S of type Rec_l, which associates a type to each field. A mixDrec M of structure p and signature S is denoted as $M : Mix_{S,p}$ (l is kept implicit). The definition of this type is mutually recursive with the definition of the relation $\gtrdot$ which expresses that a mixDrec is a more defined view of another one. In particular, if m_1 and m_2 are respectively the left and right son of a manifest node, we must have $m_2 \gtrdot m_1$. Inference rules for Pre_l, $M : Mix_{S,p}$ and $\gtrdot$ are described in figure 2. The side condition for the typing rule of a manifest node captures the notion of balanced mixDrec given in [16]. Intuitively, it says that $(f\ x)$ is the most defined view of m when a is abstract.

$$\frac{}{\emptyset_p : Pre_\emptyset} \qquad \frac{a \notin l \qquad p : Pre_l}{\mathbb{A}\ p : Pre_{a;l}} \qquad \frac{a \notin l \qquad p_1 : Pre_l \qquad p_2 : Pre_l}{\mathbb{M}\ p_1\ p_2 : Pre_{a;l}}$$

$$\frac{}{\{\!\!\{\}\!\!\} : Mix_{\{\}_s,\emptyset_p}} \qquad \frac{f : \Pi x : T.Mix_{(F\ x),p}}{\{\!\!\{a : T; f\}\!\!\} : Mix_{\{a:T;F\}_s,\mathbb{A}p}}$$

$$\frac{x : T \qquad \begin{array}{c} f : \Pi x : T.Mix_{(F\ x),p_1} \\ m : Mix_{(F\ x),p_2} \end{array} \qquad m \gtrdot (f\ x)}{\{\!\!\{a : T = x; f; m\}\!\!\} : Mix_{\{a:T;F\}_s,\mathbb{M}p_1p_2}} \ \begin{array}{l} \forall f' : \Pi x : T.Mix_{(F\ x),p_1}, \\ m \gtrdot (f'\ x) \Rightarrow f\ x \gtrdot f'\ x \end{array}$$

$$\frac{}{\{\!\!\{\}\!\!\} \gtrdot \{\!\!\{\}\!\!\}} \qquad \frac{\forall y : T,\ f_1\ y \gtrdot f_2\ y}{\{\!\!\{a : T; f_1\}\!\!\} \gtrdot \{\!\!\{a : T; f_2\}\!\!\}} \qquad \frac{m \gtrdot (f_1\ x) \qquad \forall y : T,\ f_1\ y \gtrdot f_2\ y}{\{\!\!\{a : T = x; f_1; m\}\!\!\} \gtrdot \{\!\!\{a : T; f_2\}\!\!\}}$$

$$\frac{m_1 \gtrdot m_2 \qquad \forall y : T,\ f_1\ y \gtrdot f_2\ y}{\{\!\!\{a : T = x; f_1; m_1\}\!\!\} \gtrdot \{\!\!\{a : T = x; f_2; m_2\}\!\!\}}$$

Fig. 2. types of mixDrecs

4.3 Operations on MixDrecs

The two main operations on mixDrecs are the embedding of a mixDrec M of signature s in a sub-signature s' of s, $\Uparrow_{s'} M$, and the fusion of two mixDrecs M_1 and M_2 sharing the same signature, $M_1 \oplus M_2$.

$\Uparrow_{s'} M$ is defined by induction on the derivation of $s' :\succ s$. Basically, we add the fields of s' that are not present in s as abstract nodes, and reorder the fields of s according to s' order.

To compute $M_1 \oplus M_2$, we traverse both structures, and consider the corresponding nodes of each side. The definitions of M_1 have precedence over the one of M_2. In other words, if x is defined in both M_1 and M_2 we take the definition of M_1 and follow the left son of the manifest node in M_2 since we provide a new definition: on this branch, fields with a def-dependency on x are abstract.

But even if the node of the first mixDrec is abstract, it might not be possible to use a definition provided by the second mixDrec. Indeed, we have to take into account possible def-dependencies with respect to previously considered fields. For instance, if we take the two mixDrecs:

$$M_1 = \{\!\!\{a : T_1 = e_1; \lambda a : T_1 . \{\!\!\{b : T_2\}\!\!\} ; \{\!\!\{b : T_2 = e_2\}\!\!\}\}\!\!\}$$

$$M_2 = \left\{\!\!\!\left\{ \begin{array}{r} a : T_1 = e_1';\ \lambda a : T_1 . \{\!\!\{b : T_2 = e_2'\}\!\!\} ; \\ \{\!\!\{b : T_2 = e_2'\}\!\!\} \end{array} \right\}\!\!\!\right\}$$

then $M_1 \oplus M_2$ is equal to

$$M = \{\!\!\{a : T_1 = e_1; \lambda a : T_1 . \{\!\!\{b : T_2\}\!\!\} ; \{\!\!\{b : T_2 = e_2\}\!\!\}\}\!\!\}$$

and not to the mixDrec M' obtained by comparing the nodes of M_1 and M_2 placed in the same position:

$$M' = \left\{\!\!\!\left\{ \begin{array}{r} a : T_1 = e_1;\ \lambda a : T_1 . \{\!\!\{b : T_2 = e_2'\}\!\!\} ; \\ \{\!\!\{b : T_2 = e_2\}\!\!\} \end{array} \right\}\!\!\!\right\}$$

Indeed, M' has two different definitions for the field b and thus is not well-formed.

To take into account the global structure of the mixDrecs, Boulmé introduces a list of booleans, the *control list*, indicating for each field whether it is possible or not to take the definition of the mixDrec on the right. Briefly, for a given field x, the corresponding element of the control list is *true* if x is abstract in the rightmost branch of M_1 and defined in M_2, and *false* otherwise, as it is the case for b in the previous example.

5 From Species to MixDrecs

We have just seen two semantics of Focal libraries. The *mixDrecs* semantics formalizes the notion of species. The *method generator* semantics provides a light way to express the environment in which each proof obligation of the Focal library can be verified with Coq. In the next sections, we examine the relations between these two semantics.

5.1 Plain Translation

A species s put in normal form $norm(s)$ can be directly translated into a mixDrec, $\ll s \gg$. Whenever there is a defined method x in $norm(s)$, we compute the left son of the corresponding node in $\ll s \gg$ by erasing the definitions that def-depend upon x. The structure of such a mixDrec, $\mathbb{P}(s)$, is defined the same way. It is also possible to derive a signature of Drecord from s, $[\![s]\!]$, by taking the types of the methods in the order given by $norm(s)$

Theorem 1. $\ll s \gg$: $Mix_{[\![s]\!],\mathbb{P}(s)}$ *Moreover, the names of the fields of* $\ll s \gg$ *are exactly the names of the methods of* $norm(s)$*.*

Proof. By induction on the number of methods of $norm(s)$. □

5.2 Inheritance

Let s_1 be a well-formed species and take $M_1 = \ll s_1 \gg$ the mixDrec associated to its normal form. Let s_2 be a well-formed species defined by

$$\textbf{species } s_2 \textbf{ inherits } s_1 = \phi_1 \ldots \phi_n \textbf{ end}$$

The following properties have been proved in [16]:

Lemma 1. $[\![s_2]\!] :\succ [\![s_1]\!]$

Theorem 2 (Correctness of single inheritance). *With the above notations, the following equality holds:* $\ll s_2 \gg \oplus \left(\Uparrow_{[\![s_2]\!]} tomix s_1\right) = \ll s_2 \gg$

Intuitively, this theorem states that in the single inheritance case, the inheritance resolution algorithm conforms to the mixDrecs semantics: the fusion of $\ll s_2 \gg$ and the embedding of $\ll s_1 \gg$ does not add any definition that existed in s_1 and has been erased during inheritance resolution.

Theorem 3 (Correctness of multiple inheritance).
Let s be defined as **species** s **inherits** s_1, s_2 = **end**. *Then*

$$\ll s \gg = (\Uparrow_{[\![s]\!]} \ll s_2 \gg) \oplus (\Uparrow_{[\![s]\!]} \ll s_1 \gg)$$

6 Method Generators and MixDrecs

First, we define paths inside a MixDrec $\mathcal{M}$ as usual: a path p is a sequence of 0 and 1 of length smaller than the depth of $\mathcal{M}$.

Definition 4 (Definition contexts). *Let $\mathcal{M}$ be a mixDrec, and p a path. The* context *associated to p, noted $\Gamma_p(\mathcal{M})$ is defined according to the following rules:*

$$\begin{array}{ll}
\Gamma_\emptyset(\mathcal{M}) & = \{\!\{\}\!\} \\
\Gamma_{0;p'}(\{\!\{x : T = e; f; m\}\!\}) & = \{\!\{x : T; \lambda x.\Gamma_{p'}(f\ x)\}\!\} \\
\Gamma_{0;p'}(\{\!\{x : T; f\}\!\}) & = \{\!\{x : T; \lambda x.\Gamma_{p'}(f\ x)\}\!\} \\
\Gamma_{1;p'}(\{\!\{x : T = e; f; m\}\!\}) & = \{\!\{x : T = e; \lambda x.\Gamma_{p'}(f\ x); \Gamma_{p'}(m)\}\!\} \\
\Gamma_{1;p'}(\{\!\{x : T; f\}\!\}) & = \{\!\{x : T; \lambda x.\Gamma_{p'}(f\ x)\}\!\}
\end{array}$$

Let x be the last field of $\Gamma_p(\mathcal{M})$. If x is defined in $\Gamma_p(\mathcal{M})$, we say that p is a definition path *of x, and $\Gamma_p(\mathcal{M})$ is a* definition context *of x in $\mathcal{M}$.*

Lemma 2. *Given a mixDrec $\mathcal{M}$ and a defined field x of $\mathcal{M}$, let $\rightsquigarrow^{\mathcal{M}}(x)$ be* the minimal definition path *of x in $\mathcal{M}$ according to the lexicographic ordering over paths. We write $\Gamma(\mathcal{M}) \vdash x$ for $\Gamma_{\rightsquigarrow^{\mathcal{M}}(x)}(\mathcal{M})$. If a field y is defined in $\Gamma(\mathcal{M}) \vdash x$, y is also defined in any definition context associated to x in $\mathcal{M}$.*

In addition, we define an equivalence relation $\Leftrightarrow$ over mixDrecs, such that two well-formed mixDrecs defining the same fields but in different order are equivalent. $\Leftrightarrow$ can be seen as an extension of the congruence associated to the $:\succ$ relation over signatures to the mixDrecs. Given a species s and a defined method x, we can find a mixDrec $\mathcal{M}$ equivalent to $\ll s \gg$, such that $\Gamma(\mathcal{M}) \vdash x$ is equal to $\ll s \Cap x \gg$, and the depth of x in $\mathcal{M}$ is minimal for the whole equivalence class. This shows that $s \Cap x$ is minimal according to mixDrecs semantics: method generators can be extracted from the mixDrecs by selecting the appropriate path.

Theorem 4. *Let $\mathcal{T}_s(x)$ and $\mathcal{B}_s(x)$ be the type and the body of x in s, and $\gamma_s(x) = \ll s \Cap x; x : \mathcal{T}_s(x) = \mathcal{B}_s(x) \gg$. There exists a mixDrec $\mathcal{M}$ such that*

1. $\mathcal{M} \Leftrightarrow \Gamma(\ll s \gg) \vdash x$
2. $\Gamma(\mathcal{M}) \vdash x = \gamma_s(x)$
3. $\forall m$, such that $m \Leftrightarrow \Gamma(\ll s \gg) \vdash x$, *the depth of x in $\mathcal{M}$ is less than or equal to the depth of x in m.*

Proof. First, we use the rules of $\Leftrightarrow$ to build $\mathcal{M}$ such that it is equivalent to $\Gamma(\ll s \gg) \vdash x$, and then prove that it verifies the two other properties. The construction itself is based on the fact that if y_1 and y_2 are two consecutive fields of $\Gamma(\ll s \gg) \vdash x$, such that y_1 is not in $\mid x \mid$, while y_2 is in $\mid x \mid \cup \{x\}$, then it is possible to swap the two elements. We perform recursively all such permutations, until no one is possible (since the depth of the methods of $\mid x \mid$ strictly decrease at each step, this always terminates).

Then by definition, all the fields of $\mathcal{M}$ preceding x are in $\mid x \mid$ (otherwise, we could make an exchange), and their relative order is preserved. Moreover, it follows from the preceding lemma that the only manifest nodes of $\Gamma\,(\ll s \gg) \vdash x$ correspond to the def-dependencies of x, that is the methods defined in $s \Cap x$: $\Gamma\,(\mathcal{M}) \vdash x = \gamma_s(x)$. Last, by induction on the derivation of $y \in \mid x \mid$, it is impossible to swap x with any of the y in $\mid x \mid$. $\square$

7 Related Work

The very idea of mixDrecs can be viewed as an extension of de Bruijn's telescopic mappings [7], in which the context is formed of definitions as well as of abstractions. Roughly, telescopes can be seen as an embedding of contexts as λ-terms, in which each type can depend on the preceding abstractions. More generally, different formalisms have been designed to deal with "incomplete" terms. In particular, the λc calculus [3] offers a very flexible approach in which holes (similar to the declared methods of FOCAL), can be manipulated as normal λ variables, thanks to new binders. Similar approaches have been proposed in particular by Sato, Sakurai and Kameyama [20], Sands [19], and Mason [12]. Last, Lee and Friedman [11] uses contexts of lambda calculus to obtain a notion of separate compilation, the distinguished free variables of a term being the names of the values that must be given by the context. However, none of these calculi deal explicitly with redefinition, a crucial point in FOCAL inheritance resolution.

Pollack [15] and Betarte [2] have given their own embedding of dependent records in Type Theory. Both provide operations to extend existing signatures with new abstract fields, quite similar to the one that have been presented in section 4. Recently, Coquand, Pollack and Takeyama [6] have also presented a notion of records in which some fields can be defined, as in MixDrecs. However, they do not seem to look toward a computational counterpart such as the FOCAL translation into OCAML, so that they do not deal with redefinitions either. This applies also to the definition of records given by Kopylov [10], which is based on the notion of intersection types. On the other hand, a formalization of late binding with specifications and proofs in the proof assistant LEGO is given in [9]. However, they seem to restrict the theorems that can be proved inside a class in order to avoid issues related to def-dependencies.

On the programming side, mixDrecs can be related to the mixin modules of Ancona and Zucca [1], and to their implementation as an extension of the module system of OCAML [8]. Mixins can be seen as structure mixing some features of modules (and in particular type definitions and abstraction), and classes (inheritance). In [13], the νobj calculus introduces dependent types in an object-oriented language. In this approach, classes can have type components. Hence, on the contrary to mixins, νobj adds module features to objects. However, both of them deal mainly with functions. Thus, they do not address the issues specific to properties and theorems, and in particular the notion of def-dependency.

FOCAL programming features have been partly inspired by the "*compiler of computer algebra libraries*" Axiom and its successor Aldor. In order to integrate deduction steps in the system, Thompson and Poll propose in [23, 14] to extend the type system of Aldor with dependent types and properties. However, their project seems to be in a quite initial stage.

8 Perspectives

During the implementation of the FOCAL library [22], some constructions have been provided inside the FOCAL compiler in addition to the core FOCAL methods presented here. Some work is needed to incorporate these constructions at a theoretical level. First, it is often interesting to be able to define "local" methods. Such methods are only visible to the other methods of the species in which they are defined. They can be treated as normal methods provided dependencies upon such method are flagged as def-dependencies (as if their definition was in-lined everywhere). On the mixDrec side, there is nothing to handle them, though.

It would also be convenient to rename some methods during inheritance. For instance, monoids could declare a generic operation *op*, which would be called *plus* for abelian groups, and *mult* in rings. This has not been implemented yet, but since method generators and Drecords signatures implement dependencies by λ-abstractions, renaming might reduce to alpha-conversion.

The most important extension is the possibility to define inner collections inside a species by using **self**, the species currently defined to instantiate the parameters of a given species. This construction is translated to OCAML, and a restricted version has been studied formally (in particular with respect to dependencies analysis) in [16], but the general case as well as the translation into COQ need to be investigated.

9 Conclusion

In this paper, we have presented two embeddings of the main FOCAL constructions into COQ. The first one, which uses "simple" terms made of abstractions and local bindings, provides an efficient encoding, as the produced terms are quite easily type-checked by COQ, while *method generators* let us reuse as much code as possible. On the contrary, *mixDrecs* are a very heavy encoding, but they preserve the structure of the species and the relation between them. Moreover, we have seen that the treatment of inheritance is the same in both embeddings, and that method generators can be extracted from mixDrecs.

Acknowledgment

The authors wish to thank Thérèse Hardin and the anonymous referees for their helpful remarks on earlier versions of this paper.

References

[1] D. Ancona and E. Zucca. An algebra of mixin modules. In *WADT'97*, volume 1376 of *LNCS*, 1998.

[2] G. Betarte. *Dependent Record Types and Formal Abstract Reasoning: Theory and Practice*. PhD thesis, University of Göteborg, 1998.

[3] M. Bognar and R. de Vrijer. A calculus of lambda calculus contexts. *Journal of Automated Reasoning*, 27(1), 2001.

[4] S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*. Thèse de doctorat, Université Paris 6, 2000. `http://www-lsr.imag.fr/Les.Personnes/Sylvain.Boulme/pub/sbthese.ps.gz`.

[5] S. Boulmé, Th. Hardin, and R. Rioboo. Some hints for polynomials in the Foc project. In *Proc. Calculemus*, 2001.

[6] T. Coquand, R. Pollack, and M. Takeyama. A logical framework with dependently typed records. In *Typed Lambda Calculus and Applications, TLCA'03*, volume 2701 of *LNCS*, 2003.

[7] N. G. de Bruijn. Telescopic mappings in typed λ-calculus. *Information and Computation*, 91(2), 1991.

[8] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *ESOP*, volume 2305 of *LNCS*, 2002.

[9] Martin Hofmann et al. Inheritance of proofs. *TAPOS*, 4(1):51–69, 1998.

[10] A. Kopylov. Dependent intersection: A new way of defining records in type theory. In *LICS*, 2003.

[11] S. Lee and D. P. Friedman. Enriching the lambda calculus with contexts: Toward a theory of incremental program construction. In *Proceedings of ICFP*, ACM SIGPLAN notices, 1996.

[12] I. A. Mason. Computing with contexts. *Higher-Order and Symbolic Computation*, 12, 1999.

[13] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *FOOL 10*, 2003.

[14] E. Poll and S. Thompson. Integrating Computer Algebra and Reasoning through the Type System of Aldor. In *FROCOS*, volume 1794 of *LNCS*, 2000.

[15] R. Pollack. Dependently typed records for representing mathematical structures. In *TPHOLs*, volume 1869 of *LNCS*, 2000.

[16] V. Prevosto. *Conception et Implantation du langage FoC pour le développement de logiciels certifiés*. Thèse de doctorat, Université Paris 6, 2003. `http://www.mpi-sb.mpg.de/~prevosto/papiers/these.ps.gz`.

[17] V. Prevosto and D. Doligez. Inheritance of algorithms and proofs in the computer algebra library foc. *Journal of Automated Reasoning*, 29(3-4), 2002.

[18] R. Rioboo. *Programmer le calcul formel, des algorithmes à la sémantique*. Habilitation, Université Paris 6, 2002.

[19] D. Sands. Computing with contexts: A simple approach. In *Second Workshop on Higher-Order Operational Techniques in Semantics*, volume 10 of *ENTCS*, 1997.

[20] M. Sato, T. Sakurai, and Y. Kameyama. A simply typed context calculus with first-class environments. *J. of Functional and Logic Programming*, 2002(4), 2002.

[21] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0*. INRIA-Rocquencourt, 2004.

[22] The Focal development team. *Focal, version 0.2 Tutorial and reference manual*. LIP6 – INRIA – CNAM, 2004. `http://modulogic.inria.fr/focal/download/`.

[23] S. Thompson. Logic and Dependent Types in the Aldor Computer Algebra System. In *Proc. Calculemus*, 2000.