Algebraic Structures and Dependent Records

Virgile Prevosto[1,2], Damien Doligez[1], and Thérèse Hardin[1,2]

[1] I.N.R.I.A – Projet Moscova
B.P. 105 – F-78153 Le Chesnay, France
Damien.Doligez@inria.fr
[2] L.I.P. 6 – Equipe SPI
8 rue du Cap. Scott – 75015 PARIS, France
[therese.hardin,virgile.prevosto]@lip6.fr

Abstract. In mathematics, algebraic structures are defined according to a rather strict hierarchy: rings come up after groups, which rely themselves on monoids, and so on. In the FOC project, we represent these structures by *species*. A species is made up of algorithms as well as proofs that these algorithms meet their specifications, and it can be built from existing species through inheritance and refinement mechanisms.
To avoid inconsistencies, these mechanisms must be used carefully. In this paper, we recall the conditions that must be fulfilled when going from a species to another, as formalized by S. Boulmé in his PhD [?]. We then show how these conditions can be checked through a static analysis of the FOC code. Finally, we describe how to translate FOC declarations into COQ.

1 Introduction

1.1 The FOC Project

Although computer algebra is based upon strong mathematical grounds, errors are not so rare in current computer algebra systems. Indeed, algorithms may be very complex and there is an abundance of corner cases. Moreover, preconditions may be needed to apply a given algorithm and errors can occur if these preconditions are not checked.

In the FOC language[1], any implementation must come with a proof of its correctness. This includes of course pre- and post- condition statements, but also proofs of purely mathematical theorems. In a computer algebra library, a single proof is of little use by itself. Indeed numerous algorithms, and thus their proofs, can be reused in slightly different contexts. For example a tool written for groups can be used in rings, provided that the system knows every ring is a group. Thus, we need a completely formalized representation of the relations between the mathematical structures, which will serve as a common framework for both proofs and implementations.

In his PhD thesis [?], S. Boulmé gives a formal specification of both the hierarchy of the library and the tools used to extend it. This formalization of the

[1] http:www-spi.lip6.fr/~foc

specification, briefly presented below (Sec. 2), points out that some invariants must be preserved when extending an existing structure. In particular, the *dependencies* between the functions and the properties of a given structure must be analyzed carefully, as well as *dependencies* between structures.

We have elaborated a syntax that allows the user to write programs, statements and proofs. This syntax is restrictive enough to prevent some inconsistencies, but not all. In this paper we describe the core features of this syntax (Sec. 3), and present code analyses to detect remaining inconsistencies (Sec. 4). Then, we show how to use the results of this analysis to translate FOC sources into COQ, in order to have FOC proofs verified by the COQ system (Sec 5).

1.2 FOC's Ground Concepts

Species. Species are the nodes of the hierarchy of structures that makes up the library. They correspond to the algebraic structures in mathematics. A species can be seen as a set of *methods*, which are identified by their names. In particular, there is a special method, called the *carrier*, which is the type of the representation of the underlying set of the algebraic structure.

Every method can be either *declared* or *defined*. *Declared* methods introduce the constants and primitive operations. Moreover, axioms are also represented by declared methods, as would be expected in view of the Curry-Howard isomorphism. *Defined* methods represent implementations of operations and proofs of theorems. The declaration of a method can use the carrier.

As an example, a monoid is built upon a set represented by its carrier. It has some declared operations, (specified by their *signature*), namely $=$, $+$, and *zero*. These operations must satisfy the axioms of monoids, which are expressed in FOC by *properties*. We can then define a *function*, *double*, such that $double(x) = x+x$, and prove some *theorems* about it, for instance that $double(zero) = zero$.

Interface. An *interface* is attached to each species: it is simply the list of all the methods of the species considered as only declared. As S. Boulmé pointed out, erasing the definitions of the methods may lead to inconsistencies. Indeed, some properties may depend on previous definitions, and become ill-typed if these definitions are erased. This is explained in more detail in section 2.2. Interfaces correspond to the point of view of the end-user, who wants to know which functions he can use, and which properties these functions have, but doesn't care about the details of the implementation.

Collection. A *collection* is a completely defined species. This means that every field must be defined, and every parameter instantiated. In addition, a collection is "frozen". Namely, it cannot be used as a parent of a species in the inheritance graph, and its carrier is considered an abstract data type. A collection represents an implementation of a particular mathematical structure, such as $(\mathbb{Z}, +, *)$ implemented upon the GMP library.

Parameters. We also distinguish between "atomic" species and "parameterized" species. There are two kinds of parameters: entities and collections. For instance, a species of matrices will take two integers (representing its dimensions) as parameters. These integers are entities of some collection. For its coefficients, the species of matrices will also take a collection as argument, which must have at least the features specified by the interface of `ring`. Of course, it can be a richer structure, a `field` for instance.

A species s_1 parameterized by an interface s_2 can call any method declared in s_2. Thus, the parameter must be instantiated by a completely defined species, *i.e.* a collection.

2 Constraints on Species Definition

S. Boulmé, in [?], specified different conditions that must be fulfilled when building the species hierarchy. These conditions are required to define a model of the hierarchy in the calculus of inductive constructions. By building a categorical model of the hierarchy, S. Boulmé also showed that they were necessary conditions. One of the objectives of this paper is to show how the implementation of Foc fulfills these conditions.

2.1 Decl- and Def- Dependencies

We will now present these conditions through an example. We can take for instance the species of `setoid`, a set with an equality relation. More precisely, the species has the following methods: a carrier `rep`, an abstract equality `eq`, and a property `eq_refl` stating that `eq` is reflexive. From `eq`, we define its negation `neq`, and prove by the theorem `neq_nrefl` that it is irreflexive. Using a Coq-like syntax, we can represent `setoid` like this:

$$\left\{\begin{array}{l}
\texttt{rep} : \textbf{Set} \\
\texttt{eq} :\ \texttt{rep} \texttt{->} \texttt{rep} \texttt{->} \textbf{Prop} \\
\texttt{neq}\ :\ \texttt{rep} \texttt{->} \texttt{rep} \texttt{->} \textbf{Prop}\ :=\ [\texttt{x},\texttt{y} : \texttt{rep}](\texttt{not}\ (\texttt{eq x y})) \\
\texttt{eq_refl}\ :\ (\texttt{x} : \texttt{rep})(\texttt{eq x x}) \\
\texttt{neq_nrefl}\ :\ (\texttt{x} : \texttt{rep})(\texttt{not}\ (\texttt{neq x x}))\ := \\
\qquad\qquad [\texttt{x} : \texttt{rep};\ \texttt{H} : (\texttt{not}\ (\texttt{eq x x}))](\texttt{H}\ (\texttt{eq_refl x}))
\end{array}\right\}$$

Thanks to the Curry-Howard isomorphism, functions and specifications are treated the same way. We must first verify that the methods have well-formed types. In addition, the body of every defined method must have the type given in its declaration. We can remark that the order in which we introduce the methods of the species is important: in order to write the type of `eq`, we must know that there exists a `Set` called `rep`. Similarly, the body of `neq` refers to `eq`, as does the property `eq_refl`. These three cases are very similar: a method $m2$ uses $m1$, and in order to typecheck $m2$, we need $m1$ in the typing environment. In this case, we speak of a *decl-dependency* of $m2$ upon $m1$.

On the other hand, in order to typecheck `neq_nrefl`, it is not enough to have the type of `neq` in the environment. Indeed, we must know that it is defined as `(not (eq x y))`, because hypothesis `H` in the body of `neq_nrefl` must match the definition of `neq`. Thus, `neq` must be unfolded during the typechecking of `neq_nrefl`. We identify this case as a *def-dependency*. When dealing with inheritance, this new kind of dependency has a major drawback: if we want to redefine `neq` in a species that inherits from `setoid`, then we will have to provide a new proof for `neq_nrefl`. There is no such problem with *decl-dependencies*: the definition of `neq` remains valid for any definition of `eq`, provided it has the right type.

2.2 Purely Abstract Interface

Def-dependencies do not occur only in proofs. They can also appear at the level of types. For instance, take the following species definition (again in a Coq-like syntax, `0` being a constant of type `nat`).

$$\{\mathtt{rep} : \ \mathtt{Set} \ := \ \mathtt{nat};\ \mathtt{p} : \ (\exists \mathtt{x} : \mathtt{rep} \mid \mathtt{x} \ = \ \mathtt{0})\}$$

Here, in order to accept the property `p` as well-typed, we have to know that `rep` is an alias of `nat`. If we remove the definition of `rep`, then the resulting interface is clearly inconsistent. Thus we cannot accept such a species definition, because any species must receive a valid interface. In a correctly written species, the type of a method cannot def-depend upon another method. This restriction was identified by S. Boulmé when representing species by records with dependent fields.

3 Syntax

In this section, we present the core syntax of Foc and an intuitive explanation of its semantics. The complete syntax is built upon the core syntax by adding syntactic sugar without changing its expressive power, so the properties of the core language are easily extended to the full language. In the rest of the paper, we will use the following conventions concerning variable names. Lambda-bound variables, function names and method names are usually denoted by x or y. Species names are denoted by s, and collection names by c. There is also a keyword, **self**, which can be used only inside a species s. It represents the "current" collection (thus **self** is a collection name), allowing to call its methods by **self**!x (see 3.5).

3.1 Types

type ::= c | α | *type* −> *type* | *type* * *type*

A type can be a collection name c (representing the carrier of that collection), a type variable α, or a function or a product type.

3.2 Expressions and Properties

identifier ::= *x*, *y*
declaration ::= *x* [**in** *type*]
expression ::= *x* | *c*!*x* | **fun** *declaration* −> *expression*
| *expression*(*expression* { ,*expression* }*)
| **let** [**rec**] *declaration* = *expression* **in** *expression*

An expression can be a variable (x, y), a method x of some collection c, a functional abstraction, a function application, or a local definition with an expression in its scope.

Properties are boolean expressions with first-order quantifiers:

prop ::= *expr* | *prop* **and** *prop* | *prop* **or** *prop* | *prop* → *prop* | **not** *prop*
| **all** *x* **in** *type*, *prop* | **ex** *x* **in** *type*, *prop*

3.3 Fields of a Species

def_field ::= **let** *declaration* = *expression*
| **let rec** { *declaration* = *expression*; }+
| **rep** = *type* | **theorem** *x* : *prop* **proof:** [deps] *proof*
decl_field ::= **sig** *x* **in** *type* | **rep** | **property** *x* : *prop*
field ::= *def_field* | *decl_field*
deps ::= { (**decl:** | **def:**) { *x* }* }*

A *field* ϕ of a species is usually a declaration or a definition of a method name. In the case of mutually recursive functions, a single field defines several methods at once (using the **let rec** keyword).

The carrier is also a method, introduced by the **rep** keyword. Each species must have exactly one **rep** field.

The proof language (the *proof* entry of the grammar) is currently under development. For the time being, proofs can be done directly in Coq, although the properties themselves are translated automatically. The dependencies (Sec. 2) of a proof must be stated explicitely in the *deps* clause of a theorem definition.

3.4 Species and Collection Definitions

species_def ::= **species** *s* [(*parameter* { , *parameter* }*)]
[**inherits** *species_expr* { , *species_expr* }*]
= { *field*; }* **end**
collection_def ::= **collection** *c* **implements** *species_expr*
parameter ::= *x* **in** *type*
| *c* **is** *species_expr*
species_expr ::= *s* | *s* (*expr_or_coll* { , *expr_or_coll* }*)
expr_or_coll ::= *c* | *expression*

A *species_expr* is a species identifier (for an atomic species), or a species identifier applied to some arguments (for a parameterized species). The arguments can be collections or expressions. Accordingly, in the declaration of a parameterized species, a formal parameter can be a variable (with its type) or a collection name (with its interface, which is a *species_ expr*). A collection definition is simply giving a name to a *species_ expr*.

3.5 Method and Variable Names

As pointed out in [?], method names can not be α-converted, so that they must be distinguished from variables. The notation **self**!x syntactically enforces this distinction, as we can remark in the following example.

```
let y = 3;;
species a (x in int) = let y = 4; let z = x; let my_y = self!y; end
collection a_imp implements a(y)
```

Here, a_imp!z returns 3, while a_imp!my_y returns 4.

3.6 A Complete Example

We will illustrate the main features of FOC with an example of species definition. Assume that the species `setoid` and `monoid` have already been defined, and that we have a collection `integ` that implements $\mathbb{Z}$. We now define the cartesian products of two setoids and of two monoids.

```
species cartesian_setoid(a is setoid, b is setoid)
  inherits setoid =
   rep = a * b;
   let eq = fun x -> fun y -> and(a!eq(fst(x),fst(y)),b!eq(snd(x),snd(y)));
   theorem refl : all x in self, self!eq(x,x)
     proof:
       def: eq;
       {* (* A Coq script that can use the definition of self!eq *) *} ;
end

species cartesian_monoid(a1 is monoid, b1 is monoid)
  inherits monoid, cartesian_setoid(a1,b1) =
    let bin_op = fun x -> fun y ->
       let x1 = fst(x) in let x2 = snd(x) in
       let y1 = fst(y) in let y2 = snd(y) in
         create_pair(a!bin_op(x1,y1),b!bin_op(x2,y2));
    let neutral = create_pair(a!neutral,b!neutral);
end

collection z_square implements cartesian_monoid(integ,integ)
```

4 Finding and Analyzing Dependencies

As said above, the syntax of FOC prevents some kinds of inconsistencies, but not all. To eliminate the remaining ones, we perform a static analysis on the species definitions.

4.1 Informal Description of Static Analysis

Given a species definition, we must verify that it respects the following constraints.

- All expressions must be well-typed in an ML-like type system. Redefinitions of methods must not change their type.
- When creating a collection from a species, all the fields of the species must be defined (as opposed to simply declared).
- The **rep** field must be present or inherited in every species.
- Recursion between methods is forbidden, except within a **let rec** field.

4.2 Classifying Methods

As said in section 2, when defining a species s, it is important to find the dependencies of a method x upon the other methods of s, in order to check the correctness of s. It is syntactically impossible for some dependencies to occur in FOC source. For instance, we can not write a type that depends upon a function or a property, so that the carrier of s never depends upon another method. Thus, while in the work of S. Boulmé there is only one sort of method, we distinguish here three kinds of methods: the carrier, the functions, and the specifications. Each of these can be declared or defined.

All the dependencies that can be found in a FOC definition are summed up in Fig. 1. In particular, note that a def-dependency can occur between a statement

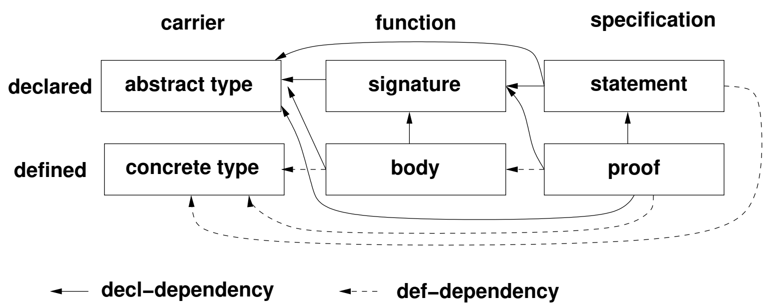

Fig. 1. Possible Dependencies Between Methods

and the carrier. Indeed, the example of section 2.2 can be written in Foc:

```
species a =
  rep = nat;
  property p : ex x in self, base_eq(x,0);
end
```

where `base_eq` is the built-in equality primitive.

Since we want to have a fully abstract interface for each species written in Foc, such a species definition will be rejected by the dependency analysis.

4.3 Identifying the Dependencies

The dependencies between the various kinds of methods cannot be computed in a uniform way. For instance, the decl-dependencies of a function body b are found by simply listing all sub-expressions of the form **self**!`m` in b. On the other hand, to identify a def-dependency of b upon `rep`, we need to typecheck b. We will now describe the computation of the dependencies.

Syntactic Criterion. We mention here all the dependencies that are found by simply looking at the Abstract Syntax Tree (AST) of the method.

- m_1 is a function body and m_2 is a function (either declared or defined): m_1 decl-depends upon m_2 if **self!**m_2 is a sub-expression of m_1.
- m_1 is a statement and m_2 a function: m_1 decl-depends upon m_2 if **self!**m_2 is a sub-expression of m_1.
- m_1 is a proof and m_2 a function or a statement: m_1 decl-depends upon m_2 if m_2 appears in the **decl** clause of m_1. It def-depends upon m_2 if m_2 appears in the **def** clause of m_1.

Typing Criterion. Some dependencies require a finer analysis to be caught. This is done in the typing phase (Sec. 4.7) and concerns the dependencies upon **rep**.

- m_1 decl-depends upon **rep** if the type of a subexpression of m_1 contains **self**.
- m_1 def-depends upon **rep** if **rep** is defined to τ, and when typing m_1, a unification step uses the equality **self** $= \tau$. In this case, the unification returns **self**.

Notations. $\wr m_1 \int_s$ is the set of names upon which m_1, considered as a method of the species s, decl-depends. Similarly, $\wr\wr m_1 \iint_s$ is the set of names upon which m_1 def-depends. **rep** is considered as a name. Note that $\wr\wr m_1 \iint_s \subseteq \wr m_1 \int_s$.

4.4 Name unicity

A name can not belong to two distinct fields of a species body. We take this condition as an invariant, which is easy to check syntactically. From a programming point of view, such situation would be an error, since one of the two declarations (or definitions) would be ignored.

Notations. Let $\mathcal{N}(\phi)$ be the names of methods introduced in a field ϕ (only one name when no mutual recursion), and $\mathcal{D}(\phi)$, the names that are introduced in a field definition. In the following, we will consider this general form of a species definition (*defspec*), which respects the invariant:

$$\textbf{species } \mathrm{s} \textbf{ inherits } s_1, \ldots s_n = \phi_1 \ldots \phi_m, \\ \text{such that } \forall i, j \leq m, \mathcal{N}(\phi_i) \cap \mathcal{N}(\phi_j) = \emptyset.$$

Then we define the set of names of the species s by

$$\mathcal{N}(s) = \left(\bigcup_{i=1}^{n} \mathcal{N}(s_i) \right) \cup \left(\bigcup_{j=1}^{m} \mathcal{N}(\phi_j) \right)$$

4.5 Binding of a method

Let $x \in \mathcal{N}(s)$. The binding $\mathcal{B}_s(x)$ of x is, roughly speaking, the body of the definition of x, if any. But, in case of multiple inheritance, x may be associated to several inherited definitions. Then $\mathcal{B}_s(x)$ is the last such definition in the order specified by the **inherits** clause.

Definition 1 (binding of a method). *Let s be a species defined by defspec, and $x \in \mathcal{N}(s)$. $\mathcal{B}_s(x)$, $\mathbb{I}_s(x)$ and $\mathcal{D}(s)$ are recursively defined as follows.*

- *if $\forall i \leq n,\ x \notin \mathcal{D}(s_i) \wedge \forall j \leq m,\ x \notin \mathcal{D}(\phi_j)$ then $\mathcal{B}_s(x) = \perp$.*
- *if $\exists i \leq m,\ \phi_i$ is **let** $x = expr$ then $\mathcal{B}_s(x) = expr$, and $\mathbb{I}_s(x) = n+1$.*
- *if $\exists i \leq m,\ \phi_i$ is **let rec** $\{x_1 = expr_1 \ldots x_l = expr_l\}$, and $x_j = x$ then $\mathcal{B}_s(x) = expr_j$ and $\mathbb{I}_s(x) = n+1$*
- *if $\exists i \leq m,\ \phi_i$ is **theorem** $x : ...proof$ then $\mathcal{B}_s(x) = proof$, and $\mathbb{I}_s(x) = n+1$*
- *else let i_0 be the greatest index such that $x \in \mathcal{D}(s_{i_0})$ then $\mathcal{B}_s(x) = \mathcal{B}_{s_{i_0}}(x)$, and $\mathbb{I}_s(x) = i_0$*

$\mathcal{D}(s) = \{x \in \mathcal{N}(s), \mathcal{B}_s(x) \neq \perp\}$

4.6 Normal Form of a Species

To ensure that a species s meets all the constraints, we compute its *normal form* (def. 3), in which inheritance resolution, dependency analysis and typing are performed. A species in normal form has no **inherits** clause, and all its fields are ordered in such a way that a field depends only upon the preceding ones.

Since **rep** has no dependencies, we choose **rep** as the first field of the normal form. Then, any other field may depend upon **rep**. To study dependencies between functions, we distinguish between **let** and **let rec** definitions. If m_1 and m_2 are defined inside the same **let rec** field, they are allowed to mutually depend upon each other – provided that a termination proof is given[2]. Thus, for a **let rec** definition ϕ, the mutual dependencies between the methods m_i of ϕ are not recorded in $\wr m_i \int_s$.

[2] Note that this termination proof def-depends upon m_1 and m_2.

Definition 2 (well-formedness).
A species s defined by defspecis said to be well-formed *if:*

- *the s_i are well-formed.*
- *All the definitions are well-typed.*
- *The different fields introduce different names:*

$$\forall i, j, i \neq j \Rightarrow \mathcal{N}(\phi_i) \cap \mathcal{N}(\phi_j) = \emptyset$$

- *A given definition decl-depends only upon previous fields:*

$$\forall i \leq n, \forall x \in \mathcal{N}(\phi_i),\ \wr x \int_s \subset \bigcup_{j=1}^{i-1} \mathcal{N}(\phi_j)$$

Requiring that definitions are well-typed implies that def-dependencies are correctly handled. Indeed, typechecking will fail if the definition is missing.

Definition 3 (normal form). *A species s is said to be in normal form if it is well-formed and it has no **inherits** clause.*

Definition 4. *changed(y, x) is a relation over $\mathcal{N}(s)$, s being defined by*

$$\textbf{\textit{species}}\ s\ \textbf{\textit{inherits}}\ s_1 \ldots s_m = \phi_1 \ldots \phi_n\ \textbf{\textit{end}}$$

$$\begin{aligned} \mathit{changed}(y, x) \iff & \big(\exists j > \mathbb{I}_s(x),\, y \in \mathcal{D}(s_j) \wedge \mathcal{B}_{s_j}(y) \neq \mathcal{B}_{\mathbb{I}_s(x)}(y)\big) \\ \vee \quad & (\exists k,\, y \in \mathcal{D}(\phi_k) \wedge \mathbb{I}_s(x) \neq n + 1) \end{aligned}$$

Theorem 1 (normal form of well-formed species). *For each well-formed species s, there exists a species nfs, which is in* normal form *and enjoys the following properties:*

- *names:* $\mathcal{N}(\mathit{nfs}) = \mathcal{N}(s)$
- *defined names:* $\mathcal{D}(\mathit{nfs}) \subseteq \mathcal{D}(s)$
- *definitions:* $\forall x \in \mathcal{D}(\mathit{nfs}), \mathcal{B}_s(x) = \mathcal{B}_{\mathit{nfs}}(x)$
- $\forall x \in \mathcal{D}(s) \setminus \mathcal{D}(\mathit{nfs}), \exists y \in \wr\wr x \int\int_s$ s.t.
  - $y \notin \mathcal{D}(\mathit{nfs})$ *or*
  - $y \in \mathcal{D}(\mathit{nfs})$ *and changed(y, x).*

The last clause ensures that we erase as few method bindings as possible, namely only the ones that def-depend upon methods that have *changed* during inheritance lookup, or upon a method that must itself be erased.

The proof gives all the steps of the static analysis performed on species (inheritance lookup, dependency analysis and typing). In the proof, we assimilate a species in normal form and the *ordered* sequence of all its methods. $s_1 @ s_2$ denotes the concatenation of two sequences.

Let $\mathit{norm}(s_i)$ be a normal form of s_i. We first build the following sequence: $\mathbb{W}_1 = \mathit{norm}(s_1)@...@\mathit{norm}(s_n)@[\phi_1, ..., \phi_m]$. $\mathbb{W}_1$ may contain several occurrences of the same name, due to multiple inheritance or redefinition. To solve such conflicts, we introduce a function $\ominus$, which merges two fields sharing some names.

- If the two fields ϕ_1 and ϕ_2 are declarations, $\phi_1 \circledequal \phi_2$ is a declaration too. If only one of the field is defined, $\circledequal$ takes this definition. If both ϕ_1 and ϕ_2 are definitions, then $\circledequal$ selects ϕ_2.
- Two **let rec** fields ϕ_1 and ϕ_2 can be merged even if they do not introduce exactly the same sets of names, because you can inherit a **let rec** field and then redefine only some of its methods (keeping the inherited definition for the others), or even add some new methods to this recursion. Merging two **let rec** fields is not given for free, though. Indeed, it implies that the user provides a new termination proof, that involves *all* the methods defined in $\phi_1 \circledequal \phi_2$.

Our analysis builds a sequence $\mathbb{W}_2$ of definitions from $\mathbb{W}_1 = \phi_1 \dots \phi_n$, starting with $\mathbb{W}_2 = \emptyset$. This is done with a loop; each iteration examines the first field remaining in $\mathbb{W}_1$ and updates $\mathbb{W}_1$ and $\mathbb{W}_2$. The loop ends when $\mathbb{W}_1$ is empty. The loop body is the following:

Let $\mathbb{W}_1 = \phi_1, \mathbb{X}$ and $\mathbb{W}_2 = \psi_1 \dots \psi_m$

- if $\mathcal{N}(\phi_1) \cap (\cup_{i=1}^{m} \mathcal{N}(\psi_i)) = \emptyset$ then $\mathbb{W}_1 \leftarrow \mathbb{X}$ and $\mathbb{W}_2 \leftarrow (\psi_1 \dots \psi_n, \phi_1)$: if the analyzed field does not have any name in common with the ones already processed, we can safely add it at the end of $\mathbb{W}_2$.
- else let i_0 be the smallest index such that $\mathcal{N}(\phi_1) \cap \mathcal{N}(\psi_{i_0}) \neq \emptyset$, then we do $\mathbb{W}_1 \leftarrow ((\phi_1 \circledequal \psi_{i_0}), \mathbb{X})$ and $\mathbb{W}_2 \leftarrow (\psi_1 \dots \psi_{i_0-1}, \psi_{i_0+1} \dots \psi_m)$. In the case of mutually recursive definitions, ϕ_1 can have some names in common with more than one ψ_i, so that $\phi_1 \circledequal \psi_{i_0}$ is kept in $\mathbb{W}_1$. In addition, we abstract all the fields $\{\psi_i\}_{i>i_0}$ such that $\exists x \in \mathcal{N}(\phi_1)$, $y \in \mathcal{N}(\psi_i), y <_s^{def} x$, where $<_s^{def}$ is the transitive closure of $\llbracket \cdot \rrbracket_s$.

The complete proof that this algorithm computes effectively a well-formed normal form that satisfies the conditions of theorem 1 can be found in [?]. In fact, the algorithm can be applied to any species, provided that fields can be reordered according to the last clause of def. 3. If it succeeds, then s is indeed well-formed. If it fails, the definition of s is inconsistent, and thus rejected.

4.7 Typing a Normal Form

Once inheritance resolution and dependency analyses have been done, we have to type the definitions of a species in normal form. The typing algorithm for functions is basically the same as the Hindley-Milner type inference algorithm used in the ML family of languages. We also check that specifications are well-typed, but the verification of proofs is left to Coq (Sec. 5).

The only trick here is that types must be preserved through inheritance so that, if a method is redefined, we have to check that the inferred type for the new definition is compatible with the old one. Moreover, we may detect a def-dependency upon **rep**, as said in 4.3, while typing the statement of a property or a theorem. In this case, we must reject the species definition, as explained in Sec. 2.2, since such a species can not have a fully abstract interface.

The typing inference rules are given in [?].

4.8 Parameterized Species

Let s be a parameterized species, written **species** $s(c\ \mathbf{is}\ a)\ldots$ where c is a fresh name. The typing environment of the body of s contains a binding $(c, \mathcal{A}(a, c))$, where $\mathcal{A}(a, c)$ is an interface defined as follows.

If $a = \{x_i : \tau_i = e_i\}_{i=1..n}$, then

$$\mathcal{A}(a, c) = \langle x_i : \tau_i[\mathbf{self} \leftarrow c] \rangle_{i=1..n}$$

A collection parameter may be instantiated by a richer structure than expected. For instance, polynomials must be defined over a ring, but may perfectly be given a field instead. So we define a *sub-species* relation $\preccurlyeq$ in order to instantiate a collection parameter with arguments of the right interface.

Definition 5 (sub-species). *Let $\mathcal{T}_s(x)$ be the type of x in s. Let s_1, s_2 be two species.*

$$s_1 \preccurlyeq s_2 \iff \mathcal{N}(s_2) \subset \mathcal{N}(s_1) \wedge \forall x \in \mathcal{N}(s_2), \mathcal{T}_{s_1}(x) = \mathcal{T}_{s_2}(x)$$

Thanks to the type constraints during inheritance lookup, if a inherits from b, then $a \preccurlyeq b$. Since only the types of the methods are concerned, the relation is easily extended to interfaces.

5 Certification: the translation into Coq

5.1 Interfaces

As in [?] interfaces are represented by Coq's `Records`, and collections by instances of the corresponding `Records`. In Coq, a `Record` is a n-uple in which every component is explicitly named:

Record `my_record` := { `label_1` : `type_1`; `label_2` : `type_2`; ... }.

The main issue here is that we are dealing with *d*ependent `Records`: `type_2` can use `label_1`, as in the following example:

Record `comparable` :=
{ `my_type` : **Set**; `less_than` : `my_type` -> `my_type` -> **Prop** }.

So the order in which the different labels appear is important.

We define a `Record` type in Coq, which denotes the interface of the species. If the species is $\{x_i : \tau_i = e_i\}$, then the `Record` is defined as

Record `name_spec` : `Type` := `mk_spec`$\{x_i : \tau_i\}$

We explicitly give *all* the fields of the `Record`, including the inherited ones. They have to be given in the order of the normal form because decl-dependencies can be present even at the level of types.

We also provide `coercions` between the `Record` we have just built and the `Record(s)` corresponding to the interface(s) of the father species. Such `coercions` reflect the inheritance relations of Foc.

5.2 Species

Unlike [?], a species s is not represented by a *MixDRec*, that is a `Record` that mix concrete and abstract fields. For any method m defined in s, we introduce a *method generator*, gen_m. If a method is inherited, the corresponding generator has been defined in a preceding species, and does not need to be recompiled. This offers a kind of modularity.

For instance, in the following species

```
species a =
 sig eq in self -> self -> bool;
 let neq = fun x -> fun y -> notb(self!eq(x,y));
end
```

The *method generator* for `neq` is

$$\lambda abst\_T : Set.\lambda abst\_eq : abst\_T- > abst\_T- > bool.$$

$$\lambda x, y : abst\_T.notb(abst\_eq\ x\ y)$$

Then, each species that inherits from `setoid` can use this definition of `neq`, instantiating `abst_eq` with its *own* definition of `eq`. This way, we can handle *late-binding*.

More formally, Let Σ be a normal form of a species s, (sequence $\Sigma = \{x_i : \tau_i = e_i\}$ of methods). Let e be an expression occuring in a field ϕ (e being a declaration, a statement, a binding, or a proof). We define below $\Sigma \Cap e$, which is the minimal environment needed to typecheck e (or ϕ). Due to def-dependencies, $\Cap$ can not simply select the methods ϕ depends upon. Each time ϕ def-depends upon ψ, we must also keep the methods upon which ψ itself decl-depends.

Definition 6 (Minimal Environment). *Let $\Sigma = \{x_i : \tau_i = e_i\}$ and e be an expression. $\Sigma \Cap e$ is the environment needed to typecheck e, and is defined as follows.*

$$\Sigma \sqcap e = \{x_j : \tau_j = new\_e_j | x_j \in \wr e \int \wedge (x_j : \tau_j = e_i) \in \Sigma\}$$

where $new\_e_j = \begin{cases} e_j & \text{if } x_j \in \wr\wr e \iint \\ \bot & \text{otherwise} \end{cases}$

$$\begin{aligned} U_1 &= \Sigma \sqcap e \\ U_{k+1} &= U_k \cup \bigcup_{(x_j : \tau_j = e_j) \in U_k} \Sigma \sqcap e_j \\ \Sigma \Cap e &= \bigcup_{k>0} U_k \end{aligned}$$

where $\{x : \tau = \bot\} \cup \{x : \tau = e\} = \{x : \tau = e\}$

We turn now to the translation of the definition d of a method y in Coq, according to the environment $\Sigma \Cap d$. This is done by recursion on the structure of the environment. $[d]_{coq}$ is the straightforward translation of d in Coq, each call **self!**x being replaced by the introduced variable $abst\_s$.

Definition 7 (Method Generator).

$$
\begin{aligned}
[\![\emptyset, d]\!] &= [d]_{coq} \\
[\![\{x : \tau = e; l\}, d]\!] &= \textbf{\textit{Let}}\ abst\_x : \tau := (gen_x\ \ abst\_x_i)\, \textbf{\textit{in}}\, [\![l, d]\!] \\
[\![\{x : \tau = \bot; l\}, d]\!] &= \lambda abst\_x : \tau .\, [\![l, d]\!]
\end{aligned}
$$

where $gen_x = [\![\Sigma \Cap \mathcal{B}_s(x), \mathcal{B}_s(x)]\!]$ *and* $abst\_x$ *is a fresh name.*

The second case treats def-dependencies. The method x being defined in Σ as $\{x : \tau = e\}$ has already been compiled to Coq. Thus its method generator gen_x has been obtained by abstracting the names x_i of $\Sigma \Cap \mathcal{B}_s(x)$ (note that $\Sigma \Cap \mathcal{B}_s(x) \subseteq \Sigma \Cap d$). Here, gen_x is applied to the corresponding $abst\_x_i$.

The third case concerns –simple– decl-dependencies. We only abstract x.

5.3 Collections

Collections are defined using the method generators. Namely, if c implements $s = \{x_i : \tau_i = e_i\}$, the Coq translation is the following:

$$
\begin{gathered}
\textbf{Definition}\ c\_x_1 := gen\_x_1. \\
\cdots \\
\textbf{Definition}\ c\_x_n := (gen\_x_n\ ([\![x_n]\!]_s). \\
\textbf{Definition}\ c := (mk\_s\ c\_x_1\ \ \ldots c\_x_n).
\end{gathered}
$$

where $[\![x]\!]_s = \{x_i \in \mathcal{N}(s) |\, (x_i, \tau_i, \bot) \in \Sigma \Cap \mathcal{B}_s(x)\}$. $[\![x]\!]_s$ represents the definitions that must be provided to the method generator in order to define x. mk_s is the function that generates the record corresponding to the interface of s.

5.4 Parameters

A natural way to handle parameters in Coq would be to create functions that take `Records` as arguments and return `Records`. For instance, (the interface of) a cartesian product can be defined like this:

```
Record cartesian [ A, B : basic_object] : Type :=
    { T : Set; fst : T -> A ...}
```

Another solution is to take the parameters as the first fields of the `Record`:

```
Record cartesian : Type :=
    { A : basic_object; B: basic_object; ...}
```

These two translations are quite similar for Coq. In the first one, `cartesian` will be a parameterized type, while it is not the case in the second one: `A` and `B` are only the first two arguments of its unique constructor. The second solution seems to have some practical advantages over the first one:

- Parameters can be accessed directly as fields of the record
- Fields accesses (the equivalent of methods call) do not need extra arguments, as it would be the case in the first solution.
- Coercions between parameterized records are easier to define too.
- More important, it reflects the fact that collections can not have parameters: in an implementation of `cartesian`, the fields `A` and `B` must be defined as well as `T` and `fst`.

6 Related Work

Other projects use Coq's `Records` to represent algebraic structure. In particular, L. Pottier [?] has developed quite a large mathematical library, up to fields. H. Geuvers and the FTA project [?] have defined abstract and concrete representations of reals and complex numbers. In addition, R. Pollack [?] and G. Betarte [?] have given their own embedding of dependent records in Type Theory. We can also mention Imps [?], a proof system which aims at providing a computational support for mathematical proofs. However, none of these works include a computational counterpart, similar to the Ocaml translation of Foc. P. Jackson [?] implemented a specification of multivariate polynomials in Nuprl. His approach is quite different from Foc, as in his formalism, a group can not be directly considered as a monoid, for instance.

7 Conclusion and Future Work

To sum up, we can say that Foc has now achieved a quite good expressive power. The static analyses that are discussed in Sec. 4 have been implemented [?] in a compiler that generates Coq and Ocaml code. An important number of mathematical structures have been implemented, and performances are good.

It seems to us that we provide a well-adapted framework to prove the properties needed for each species' implementation. It it is now necessary to define a proof language for Foc, dedicated to users of computer algebra systems. This is currently under development.

Building mathematical structures requires the whole power of the Calculus of Inductive Constructions, but higher-order features are mostly needed only to handle dependencies. Once we have succeeded to build an appropriate environment, the proofs themselves stay in first order logic most of the time. This may lead to a quite high level of automatization in the proof part of the project, leading to proofs in deduction modulo [?, ?]. We could then try to delegate some part of the proofs to rewriting tools. Similarly, it would be interesting to offer powerful tools that allow the user of Foc to define his own Foc proof *tactics*.

D. Delahaye's PhD [?] presents very promising developments in this area and may be of great help here. From the Curry-Howard point of view this future work is the counterpart in the proof universe of the basic expressions of the Foc language.

References

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

[2] S. Boulmé, T. Hardin, and R. Rioboo. Polymorphic data types, objects, modules and functors: is it too much ? Research Report 14, LIP6, 2000. available at <http://www.lip6.fr/reports/lip6.2000.014.html>.

[3] S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel.* PhD thesis, Université Paris 6, december 2000.

[4] B. Buchberger and all. A survey on the theorema project. In W. Kuechlin, editor, *Proceedings of ISSAC'97.* ACM Press, 1997.

[5] D. Delahaye. *Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve.* PhD thesis, Université Paris 6, 2001.

[6] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. Research Report 3400, INRIA, 1998.

[7] G. Dowek, T. Hardin, and C. Kirchner. Hol-$\lambda\sigma$: an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):21–45, 2001.

[8] W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, 202 Burlington Road, Bedford, MA 01730-1420, USA, November 1995. Available at ftp://math.harvard.edu/imps/doc/.

[9] H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The algebraic hierarchy of the fta project. In *Proceedings of the Calculemus Workshop*, 2001.

[10] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st Symposium on Principle of Programming Languages*, 1994.

[11] P. Jackson. Exploring abstract algebra in constructive type theory. In *Proceedings of 12th International Conference on Automated Deduction*, July 1994.

[12] R. Pollack. Dependently typed records for representing mathematical structures. In *TPHOLs'00.* Springer-Verlag, 2000.

[13] L. Pottier. contrib algebra pour coq, mars 1999. <http://pauillac.inria.fr/coq/contribs-eng.html>.

[14] V. Prevosto, D. Doligez, and T. Hardin. Overview of the Foc compiler. to appear as a research report, LIP6, 2002. available at <http://www-spi.lip6.fr/~prevosto/papiers/foc2002.ps.gz>.

[15] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* Projet LogiCal, INRIA-Rocquencourt – LRI Paris 11, Nov. 1996.