Algebraic Structures and Dependent Records

Virgile Prevosto\textsuperscript{1,2}, Damien Doligez\textsuperscript{1}, and Thérése Hardin\textsuperscript{1,2}

\textsuperscript{1} I.N.R.I.A – Projet Moscova  
B.P. 105 – F-78153 Le Chesnay, France  
Damien.Doligez@inria.fr  
\textsuperscript{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 \textit{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 \cite{boume1}. 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\textsuperscript{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 \cite{boume1}, S. Boulmé gives a formal specification of both the hierarchy of the library and the tools used to extend it. This formalization of the

\textsuperscript{1} http://www-spl.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 $\times$, $+$, and $\text{zero}$. These operations must satisfy the axioms of monoids, which are expressed in Foc by properties. We can then define a function, double, such that $\text{double}(x) = x+x$, and prove some theorems about it, for instance that $\text{double}(\text{zero}) = \text{zero}$.

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 $(\mathbb{Z}, +, \cdot)$ 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 \( \text{rep} \), an abstract equality \( \text{eq} \), and a property \( \text{eq_refl} \) stating that \( \text{eq} \) is reflexive. From \( \text{eq} \), we define its negation \( \text{neq} \), and prove by the theorem \( \text{neq_nrefl} \) that it is irreflexive. Using a Coq-like syntax, we can represent setoid like this:

\[
\begin{align*}
\text{rep} : \text{Set} \\
\text{eq} : \text{rep} \to \text{rep} \to \text{Prop} \\
\text{neq} : \text{rep} \to \text{rep} \to \text{Prop} := [x,y:\text{rep}][\neg (\text{eq} \ x \ y)] \\
\text{eq_refl} : (x:\text{rep})(\text{eq} \ x \ x) \\
\text{neq_nrefl} : (x:\text{rep})(\neg (\text{neq} \ x \ x)) := \\
\quad [x:\text{rep}; H:(\neg (\text{eq} \ x \ x))](H (\text{eq_refl} \ x))
\end{align*}
\]

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 \( \text{eq} \), we must know that there exists a \( \text{Set} \) called \( \text{rep} \). Similarly, the body of \( \text{neq} \) refers to \( \text{eq} \), as does the property \( \text{eq_refl} \). 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_nref1`, 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_nref1` must match the definition of `neq`. Thus, `neq` must be unfolded during the typechecking of `neq_nref1`. 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_nref1`. 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`).

```
{rep: Set := nat; p: (x:rep | x = 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 ::= \( x \mid \text{in type} \)
extension ::= \( x \mid c; x \mid \text{fin declaration } \rightarrow \text{expression} \)
expression ::= \( \text{expression(} \text{expression} \}\{, \text{expression}\}^{*} \)
let \( \text{rec } \) 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 \mid prop and prop \mid prop or prop \mid prop \rightarrow prop \mid \text{not } prop \mid \text{all } x \text{ in } type, prop \mid \text{ex } x \text{ in } type, prop

3.3 Fields of a Species

def_field ::= let declaration = expression

let rec \{ declaration = expression; \}+
rep = type \mid \text{theorem } x : \text{prop proof} : [ dep_\text{proof} ]

decl_field ::= sig \( y \) \text{in type} \mid \text{rep} \mid \text{property } x : \text{prop}

field ::= def_field \mid decl_field
dep_\text{proof} ::= \{ (\text{decl}; \text{def}) \{ x \}^{*} \}^{*}

A field \( \phi \) 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 \text{rec} keyword).

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

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

3.4 Species and Collection Definitions

species_def ::= species \( s \mid (\text{parameter} \{, \text{parameter}\}^{*}) \mid \)\n\mid \text{inherits species_expr} \{, species_expr\}^{*} \]\n\mid \text{end}
collection_def ::= collection \( c \text{ implements species_expr} \)

parameter ::= \( x \text{ in type} \mid c \text{ is species_expr} \)

species_expr ::= s \mid s (\text{expr_or_coll} \{, \text{expr_or_coll}\}^{*})

A \text{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 \textit{species\_expr}). A collection definition is simply giving a name to a \textit{species\_expr}.

### 3.5 Method and Variable Names

As pointed out in [?], method names cannot be $\alpha$-converted, so that they must be distinguished from variables. The notation \texttt{self\_x} syntactically enforces this distinction, as we can remark in the following example.

\begin{verbatim}
let $y = 3$;
species $a (x \text{ in } \text{int}) - \text{let } y = 4; \text{let } z = x; \text{let } \text{my\_y} = \text{self\_y}; \text{end collection } a_{\text{imp}} \text{ implements } a(y)
\end{verbatim}

Here, $a_{\text{imp}}\_z$ returns 3, while $a_{\text{imp}}\_\text{my\_y}$ returns 4.

### 3.6 A Complete Example

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

\begin{verbatim}
species cartesian\_setoid(a is setoid, b is setoid)
inherits setoid =
  rep = a * b;
  let eq = fun x \rightarrow fun y \rightarrow \text{and}(a\_eq(fst(x),fst(y)),b\_eq(snd(x),snd(y)));
theorem refl : \text{all } x \text{ in } self, \text{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 \rightarrow fun y \rightarrow
    let x1 = \text{fst}(x) in let x2 = \text{snd}(x) in
    let y1 = \text{fst}(y) in let y2 = \text{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)
\end{verbatim}
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. Boumlé 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

![Possible Dependencies Between Methods](image)
and the carrier. Indeed, the example of section 2.2 can be written in FOC:

```plaintext
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.

- `m₁` is a function body and `m₂` is a function (either declared or defined): `m₁` decl-depends upon `m₂` if `self m₂` is a sub-expression of `m₁`.
- `m₁` is a statement and `m₂` a function: `m₁` decl-depends upon `m₂` if `self m₂` is a sub-expression of `m₁`.
- `m₁` is a proof and `m₂` a function or a statement: `m₁` decl-depends upon `m₂` if `m₂` appears in the `decl` clause of `m₁`. If `m₁` appears upon `m₂` if `m₂` appears in the `def` clause of `m₁`.

**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`.

- `m₁` decl-depends upon `rep` if the type of a subexpression of `m₁` contains `self`.
- `m₁` def-depends upon `rep` if `rep` is defined to `τ`, and when typing `m₁`, a unification step uses the equality `self = τ`. In this case, the unification returns `self`.

**Notations.** \(\Gamma_{m₁} s\) is the set of names upon which `m₁`, considered as a method of the species `s`, decl-depends. Similarly, \(\Psi_{m₁} s\) is the set of names upon which `m₁` def-depends. `rep` is considered as a name. Note that \(\Psi_{m₁} s \subseteq \Gamma_{m₁} s\).

### 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 $\phi$ (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:

$$\text{species } s \text{ inherits } s_1, \ldots, s_m = \phi_1 \ldots \phi_m,$$

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 $\forall i \leq n, x \notin \mathcal{D}(s_i) \land \forall j \leq m, x \notin \mathcal{D}(\phi_j)$ then $\mathcal{B}_s(x) = \bot$.
- if $\exists i \leq n, x$ is let $x = \text{expr}$ then $\mathcal{B}_s(x) = \text{expr}$, and $\mathbb{I}_s(x) = n + 1$.
- if $\exists i \leq m, \phi_i$ is let rec $\{ x_1 = \text{expr}_1 \ldots x_l = \text{expr}_l \}$, and $x_j = x$ then $\mathcal{B}_s(x) = \text{expr}_j$ and $\mathbb{I}_s(x) = n + 1$.
- if $\exists i \leq m, \phi_i$ is theorem $x : \ldots \text{proof}$ then $\mathcal{B}_s(x) = \text{proof}$, 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 $\mathbb{I}_s(x) = i_0$.

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

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 $\phi$, the mutual dependencies between the methods $m_i$ of $\phi$ are not recorded in $\{ m_i \}_{\phi}$. 

$^2$ Note that this termination proof def-depends upon $m_1$ and $m_2$. 

Definition 2 (well-formedness).
A species \( s \) defined by \( \text{specs} \) is 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 \( \text{decl} \)-depends only upon previous fields:
  \( \forall i \leq n, \forall x \in \mathcal{N}(\phi_i), [x]_s \subseteq \bigcup_{j=1}^{i-1} \mathcal{N}(\phi_j) \)

Requiring that definitions are well-typed implies that \( \text{def}\)-dependencies are correctly handled. Indeed, type-checking 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 \( \text{inherits} \) clause.

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

\[ \begin{array}{c}
\text{species } s \text{ inherits } \phi_1 \ldots \phi_m \text{ end}
\end{array} \]

\[ \text{changed}(y,x) \iff (\exists j > \sharp_x(x), y \in \mathcal{D}(s_j) \land B_{s_j}(y) \neq B_{s_j}(x)(y)) \]

\[ \lor (\exists k, y \in \mathcal{D}(\phi_k) \land \sharp_x(x) \neq n + 1) \]

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: \( \mathcal{N}(nfs) = \mathcal{N}(s) \)
- defined names: \( \mathcal{D}(nfs) \subseteq \mathcal{D}(s) \)
- definitions: \( \forall x \in \mathcal{D}(nfs), B_{s}(x) = B_{nfs}(x) \)
- \( \forall x \in \mathcal{D}(s) \setminus \mathcal{D}(nfs), \exists y \in \mathcal{N}(nfs) \) s.t.
  - \( y \notin \mathcal{D}(nfs) \) or
  - \( y \in \mathcal{D}(nfs) \) and \( \text{changed}(y,x) \).

The last clause ensures that we erase as few method bindings as possible, namely only the ones that \( \text{def}\)-depend upon methods that have \( \text{changed} \) 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 \( \text{ordered} \) sequence of all its methods. \( s_1 \oplus s_2 \) denotes the concatenation of two sequences.

Let \( \text{norm}(s_i) \) be a normal form of \( s_i \). We first build the following sequence:

\[ \mathcal{W}_1 = \text{norm}(s_1) \oplus \ldots \oplus \text{norm}(s_n) \oplus [\phi_1, \ldots, \phi_m] \]

\( \mathcal{W}_1 \) may contain several occurrences of the same name, due to multiple inheritance or redefinition. To solve such conflicts, we introduce a function \( \oplus \), which merges two fields sharing some names.
- If the two fields $\phi_1$ and $\phi_2$ are declarations, $\phi_1 \odot \phi_2$ is a declaration too. If only one of the field is defined, $\odot$ takes this definition. If both $\phi_1$ and $\phi_2$ are definitions, then $\odot$ selects $\phi_2$.

- Two let rec fields $\phi_1$ and $\phi_2$ 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 $\phi_1 \odot \phi_2$.

Our analysis builds a sequence $\mathbb{W}$ of definitions from $\mathbb{W}_1 = \phi_1 \ldots \phi_n$, 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 $\mathbb{W}_1 = \phi_1, X$ and $\mathbb{W}_2 = \psi_1 \ldots \psi_m$

- if $\mathcal{N}(\phi_1) \cap (\bigcup_{i=1}^m \mathcal{N}(\psi_i)) = \emptyset$ then $\mathbb{W}_1 \leftarrow X$ and $\mathbb{W}_2 \leftarrow (\psi_1 \ldots \psi_m, \phi_1)$: 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 $\mathbb{W}_1 \leftarrow ((\phi_1 \odot \psi_{i_0}), X)$ and $\mathbb{W}_2 \leftarrow (\psi_1 \ldots \psi_{i_0-1}, \psi_{i_0+1} \ldots \psi_m)$. In the case of mutually recursive definitions, $\phi_1$ can have some names in common with more than one $\psi_i$, so that $\phi_1 \odot \psi_{i_0}$ is kept in $\mathbb{W}_1$. In addition, we abstract all the fields $\{\psi_i\}_{i > i_0}$ such that $\exists x \in \mathcal{N}(\phi_1), y \in \mathcal{N}(\psi_i), y \prec^s x$, where $\prec^s$ is the transitive closure of $\sqsubseteq$.

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 $\text{species } s(c \text{ is } a) \ldots$ 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

$$\mathcal{A}(a, c) = \{x_i : \tau_i | \text{self } \leftarrow c\}_{i=1..n}$$

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 $\preceq$ 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.

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

Thanks to the type constraints during inheritance lookup, if $a$ inherits from $b$, then $a \preceq b$. 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-ple in which every component is explicitly named:

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

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

```coq
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

```coq
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 [7], a species \( s \) is not represented by a \texttt{MixDRec}, that is a \texttt{Record} that mix concrete and abstract fields. For any method \( m \) defined in \( s \), we introduce a \texttt{method generator}, \texttt{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

\[
\text{species a =}
\begin{align*}
\text{sig eq in self} & \rightarrow \text{self} \rightarrow \text{bool}; \\
\text{let neq = fun x \rightarrow fun y \rightarrow notb(self!eq(x,y));}
\end{align*}
\]

The \texttt{method generator} for \texttt{neq} is

\[
\lambda \text{abstract}_T: \text{Set. abstract}_e: \text{abstract}_T \rightarrow \text{abstract}_T \rightarrow \text{bool}. \\
\lambda x, y: \text{abstract}_T. \text{notb(abstract}_e \ x \ y)
\]

Then, each species that inherits from \texttt{setoid} can use this definition of \texttt{neq}, instantiating \texttt{abstract}_\texttt{eq} with its own definition of \texttt{eq}. This way, we can handle \texttt{late-binding}.

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

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

\[
\Sigma \sqcap e = \{x_j: \tau_j = \text{new}_e \ e_j \ \mid \ e_j \in \Sigma \ \land \ (x_j: \tau_j = e_i) \in \Sigma\}
\]

where \( \text{new}_e \ e_j = \begin{cases} e_j & \text{if } x_j \in \Sigma \ \land \ (x_j: \tau_j = e_i) \in \Sigma \\ \bot & \text{otherwise} \end{cases} \)

\[
U_1 = \Sigma \sqcap e \\
U_{k+1} = U_k \cup \bigcup_{(x_j: \tau_j = e_j) \in U_k} \Sigma \sqcap e_j \\
\Sigma \sqcap e = \bigcup_{k > 0} U_k
\]

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 \( \Sigma \otimes d \). This is done by recursion on the structure of the environment. \( [d]_{\text{coq}} \) is the straightforward translation of \( d \) in Coq, each call \texttt{self} being replaced by the introduced variable \texttt{abst \_ s}.

**Definition 7 (Method Generator).**

\[
\begin{align*}
[\emptyset, d] &= [d]_{\text{coq}} \\
\{x : \tau = c ; l\}, d &= \text{Let abst \_ x : } \tau := (\text{gen}_x \text{ abst \_ } x_i) \text{ in } [l, d] \\
\{x : \tau = \bot ; l\}, d &= \lambda \text{ abst \_ x : } \tau. \ [l, d]
\end{align*}
\]

where \( \text{gen}_x = [\Sigma \otimes \mathcal{B}_s(x), \mathcal{B}_s(x)] \) and \( \text{abst \_ x} \) is a fresh name.

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

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:

**Definition** \( c \_ x_1 := \text{gen} \_ x_1 \).

\[ \cdots \]

**Definition** \( c \_ x_n := (\text{gen} \_ x_n (\{x_n\}_s)) \).

**Definition** \( c := (\text{mk} \_ s c \_ x_1 \ldots c \_ x_n) \).

where \( \{x\}_s = \{x \in \mathcal{N}(s) \mid (x_i, \tau_i, \bot) \in \Sigma \otimes \mathcal{B}_s(x)\} \). \( \{x\}_s \) represents the definitions that must be provided to the method generator in order to define \( x \). \( \text{mk} \_ s \) 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 \texttt{Records} as arguments and return \texttt{Records}. For instance, (the interface of) a cartesian product can be defined like this:

**Record** cartesian [ A, B : basic\_object] : Type :=

\[ \{ T : \text{Set}; \text{fst} : T \to A \ldots \} \]

Another solution is to take the parameters as the first fields of the \texttt{Record}:

**Record** cartesian : Type :=

\[ \{ A : \text{basic\_object}; B : \text{basic\_object}; \ldots \} \]
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: \(\text{A} \text{ and } \text{B} \) 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 \(\text{A} \) and \(\text{B} \) must be defined as well as \(\text{T} \) and \(\text{fst} \).

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