# Algorithms and Proofs Inheritance in the FOC Language

Virgile Prevosto (`virgile.prevosto@lip6.fr`)
*Laboratoire d'Informatique de Paris 6 – INRIA Rocquencourt*

Damien Doligez (`damien.doligez@inria.fr`)
*INRIA Rocquencourt*

**Abstract.** In this paper, we present the FOC langugage, dedicated to the development of certified computer algebra librairies (*i.e.* sets of programs). These libraries are based on a hierarchy of implementations of mathematical structures. After presenting the core set of features of our language, we describe the static analyses, which reject inconsistent programs. We then show how we translate FOC definitions into OCAML, and COQ, our target languages for the computational part and the proof checking respectively.

## 1. Introduction

### 1.1. THE FOC PROJECT

#### 1.1.1. *Computer algebra systems.*

A computer algebra system (CAS) includes two essential aspects of mathematical knowledge: first, it provides, more or less explicitely, a formalization of the mathematical structures (e.g. the definition of what is a monoid, a group, a ring, etc). Second, it must give efficient implementations of the algorithms used in these structures. Efficiency is extremely important, because CAS are used in many fields of engineering and research to perform arbitrarily complex computations. The range of applications of CAS is only limited by their performance, not by the demands of the users. This explains the emphasis of current CAS on speed of built-in algorithms, and on ease of implementation of new, more complex, faster algorithms.

On the other hand, the formalization of algebraic structures is an essential part of CAS [9], so every CAS must have some way of representing the mathematical structures, which provides the context in which its algorithms will work. This correspondence between the mathematical objects and their computer representation needs to be clearly specified and documented, even if it is not always the case.

#### 1.1.2. *State of the art in CAS.*

The design of current computer algebra systems puts heavy emphasis on the efficient implementation of state-of-the-art algorithms. In contrast, the programming language offered to the user of these systems is often poorly designed. In some systems (such as Maple or Mupad) the language is considered as some kind of scripting language rather than an important feature of the CAS. Even in Axiom [16], Aldor, or Magma, where the user language is a central part of the system, computer algebra issues take precedence over the language design and specification.

Another issue with current CAS is the complexity of the algorithms and their implementations. The algorithms used in computer algebra are generally proved correct with a mathematical proof, but they are complex and hard to understand, and implementing them is far from trivial, which explains the presence of obscure bugs in all current CAS.

These bugs are dangerous because the engineers and scientists tend to trust the answers given by the CAS, and the consequences of a wrong computation can range from a few days of time lost in tracking down the error, to a complete failure of the system designed by the engineer.

### 1.1.3. *The* FOC *approach to computer algebra.*

The FOC project[1] [3], under the direction of Th. Hardin, attempts to deal with these issues by providing a new programming language dedicated to computer algebra. We intend to ground the language on firm theoretical results, with clear semantics and an efficient implementation via translation to OCAML. Our language has functional and object-oriented features carefully tailored to the task at hand. In order to tackle the correctness problem, the language provides means for the programmers to write formal proofs of their code, and to have them verified by a proof checker (COQ).

The programming part of our approach is validated by the FOC library, developped by R. Rioboo [4], which includes some complex algorithms with performance comparable to the best CAS in existence[4].

The fact that we design our own language allows us to express more easily than in a general purpose language some very important concepts of the computer algebra, and in particular the *carrier type* of a structure [12]. On the other hand, we can also restrict object-oriented features to what is strictly necessary (Sec. 3) to computer algebra, and avoid unsound constructions, such as open recursion, which can lead to inconsistencies when used carelessly.

### 1.1.4. *Contents*

In this paper we describe the core features of the FOC language. The remainder of this section introduces informally the fundamental concepts of FOC. Then, we present the concrete syntax (Sec. 2). Some

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

programming errors cannot be avoided at the syntax level, and we have designed a static code analysis to detect them (Sec. 3). We are then able to describe the compilation of the FOC source to OCAML (Sec. 4). Finally, we show how this work is extended to statements and proofs (Sec. 5) and explain the translation into COQ (Sec. 6). Main results are the algorithm that performs the static analysis and the handling of *late binding* (see below) in COQ.

## 1.2. SPECIES

Species correspond to algebraic structures in mathematics and play a primordial role in FOC: they are the nodes of the hierarchy of structures constituting the library. Entities are the elements of species, the objects manipulated by the algorithms.

### 1.2.1. *Entities.*

Entities represent mathematical objects, such as 0 or $X^2 + 3 * X * Y$, in the computer universe. One issue here is that there is no simple relationship between these two worlds. On the one hand, the constant `1` of the `integer` type can be used to represent $1 \in \mathbb{Z}/2\mathbb{Z}$ as well as $1 \in \mathbb{Z}/5\mathbb{Z}$. Of course, it would be a mistake to mix them up, since they do not have the same properties. On the other hand, the polynomial $X+2$ can be viewed as an ordered list of coefficients, `[1;2]`, or as a list of pairs (sparse representation): `[(1,1);(2,0)]`. In this case, adding two polynomials that do not share the same concrete representation is likely to produce an error. To avoid such confusion, we need an abstraction mechanism in the spirit of abstract data types.

### 1.2.2. *Species and Methods*

A *species* can be seen as a set of *methods*, which are identified by their names. A method can be either *declared* or *defined*. *Declared* methods reflect the constants, the primitive operations, and the axioms that define a structure in mathematics. *Defined* methods represent implemented operations (*i.e.* algorithms) and theorems built up (and proved) from these declared methods. There are three different categories of *methods*:

- The *carrier*, or representation type (**rep**) of a species is a type from the FOC type language. In other words (see Sec.2.1), it can be an atomic type, a product, a function type, or a parameterized type (such as `list(int)`). It represents the type of the entities that the species manipulates. The carrier of each species is unique.
- **functions** (when defined) and **signatures** (when only declared) denote the operations that are allowed on the carrier's elements.

– Finally, the developer of a new species can specify the **properties** that further implementations of this species must meet. He may also prove some **theorems** for the current declarations/definitions of functions and properties.

As an example, a monoid is built upon a set which is represented in Foc by its carrier type. It has some declared operations, (specified by their types), namely $+$, and $zero$. These operations must satisfy the axioms of monoids, which are expressed in Foc by *properties*. We can then define a *function*, *dble*, such that $dble(x) = x + x$, and prove some *theorems* about it, for instance that $\forall x \in \mathbf{rep}, dble(zero + x) = dble(x)$. Following Curry-Howard-de Bruijn isomorphism, we can link signatures and properties on the one hand (both are abstract methods), and functions and proofs (as defined methods) on the other one.

### 1.2.3. *Inheritance*

*inheritance* allows to define a new species from previous one(s). The new species inherits all the methods of its parent(s). If two parents have methods that share the same name, they must have the same type. If both methods are defined, then we have to choose the definition that will be exported in the new species. A species can define some methods that were declared in its parents, or even redefine a method. It can also declare a new method but not redeclare an old one with a different type. As said later (p. 18), this restriction ensures that any implementation of a species that inherits from a species `a` has at least the same methods as `a`, with the same type. A species can also declare and define a new method at the same time. These features, along with the parameterization described further, enables the use of a *refinement* methodology to build new species. Thus, multiple inheritance comes with overriding and late binding, which are usual features of object-oriented languages.

## 1.3. Abstraction

### 1.3.1. *Parameters*

In computer algebra, many structures are built upon previously defined algebraic structures by kinds of categorical operations. For example, an algebra of polynomials is built upon a ring $R$ of coefficients and a monomial ordering $D$ of degrees. In fact, to build polynomials, we need only to know the operations provided by $R$, and their specifications, but not their particular implementation. On the other hand, to build an effective implementation of polynomials over $\mathbb{Z}$, $R$ needs to be instantiated by a structure whose all methods are defined. This leads to the two dual notions of *interface* and *collection*.

1.3.2. *Interfaces*
An *interface* is a list of declared methods. It corresponds to the end-user point of view, 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. In FOC, the definition of a *species* must allow the definition of the associated *interface*, by removing all the bodies of the defined methods. While this abstraction is easy within programming languages, it is not always possible when dealing with proofs, as pointed out by S. Boulmé [5]. Sec. 5 deals with this problem.

1.3.3. *Collections*
Assume that we are using $Q$ as an actual parameter for $P$ when building $S$. Suppose that a function $f$ of $Q$ is only declared but is used in $S$ for a computation. Then, there are two possibilities. Either we accept to wait until run-time to obtain a definition for $f$ and then we accept run-time failures. Or, we force any actual species parameter to be a completely defined species. We choose the more restrictive way because it is safer while still having enough expressive power.

A *collection* is a completely defined species. This means that every field must be defined, and every parameter instantiated. It represents a particular mathematical structure, such as $\mathbb{Z}[X]$. Moreover, we can not access directly the entities belonging to a given collection, to avoid breaking the representation invariants. Collections can also be used to introduce a predefined types. For instance, we can assume that there exist a collection `bool` with an (abstract) carrier, two element `true` and `false`, an unary operation `not`, etc.

## 2. 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.

There are three different sets of identifiers:

- $x, y$ denote $\lambda$-bound variables, function and method names.
- $s$ denotes species names.
- $c$ denotes collection names.

There is also a keyword, **self**, which can be used only inside a species $s$ and represents the "current" collection (thus **self** is a collection name).

It is used to handle *late binding* as seen in the following example. (In FOC, `c!m` denotes the method `m` of collection `c`).

**species** `A` = **let** `f(x)` = `body`; **let** `g` = ... **self**!`f` ...; **end**
**species** `B` **inherits** `A` = **let** `f(x)` = `improved_body`; **end**

The new definition of `f` in `B` *overrides* the old one, inherited from `A`. Then, in the value of `g` in `B`, **self**`!f` is not bound to the definition of `f` in the species `A`, where `g` is defined, but to the *actual* value of `f`.

Note that when doing the static analysis of `A`, we have to assume that **self** can be any collection that inherits from `A` itself because we do not know in what context `g` will be used.

### 2.1. Expressions and Types

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

An expression can be a variable, a method $x$ of some collection $c$, a local definition with an expression in its scope, a function application, or a functional abstraction. A type can be a collection name (representing the carrier of that collection), a variable, a function or a product type.

### 2.2. Fields of a Species

| | | |
|---|---|---|
| def_field | ::= | **rep**=*type* \| **let** *declaration* = *expression* |
| | \| | **let rec** { *declaration* = *expression*; }+ |
| decl_field | ::= | **sig** *x* **in** *type* \| **rep** |
| field | ::= | *def_field* \| *decl_field* |

A *field* $\phi$ of a species is a declaration or a definition of a method name. In the case of mutually recursive methods, a single field defines several methods at once (using the **let rec** keywords). The carrier is considered also as a method, introduced by the **rep** keyword. Each species must have exactly one **rep** field, either defined or inherited.

### 2.3. 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: in the declaration of a parameterized species, a formal parameter can be a variable (and its type) or a collection name (and its interface). A species definition is an optional list of parameters, an optional list of inheritance declarations, and a list of fields (its body). Order of inheritance declarations is significant: if a method definition is inherited from several sources, the rightmost one is used. In addition, two different fields in *species_def* must define or declare disjoint sets of method names.

Note that in the complete syntax, we can allow a collection `a` implementing a species `b` to have a body composed of `def_field` entries. This can be translated in the core syntax as

```
species a_spec inherits b = def_field_of_a end
collection a implements a_spec
```

### 2.4. An Example

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. We also use a few predefined operators (`fst`, `snd`, `create_pair`, etc.).

---

```
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)));
end

species cartesian_monoid(a1 is monoid, b1 is monoid)
 inherits monoid, cartesian_setoid(a1,b1) =
```

```
    let plus = 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(a1!plus(x1, y1), b1!plus(x2, y2));
    let neutral = create_pair(a1!zero, b1!zero);
end

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

## 3. Analyzing Species

### 3.1. Informal Description of Static Analysis

Not all syntactically correct definitions are acceptable in Foc. In order to respect the coherence properties, we need to check some semantic constraints on the definitions of species and collections. We impose some additional constraints, especially on mutually recursive methods, to make the proofs easier to write. The restrictions are:

- Typing: all expressions must be well-typed, the arguments passed to parameterized species must have the expected types, 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 outside a **let rec** field is forbidden.

If a collection parameter is required to have interface $A$, the constraints on method types ensure that any implementation of $A$ can be used as an actual parameter.

We want the programmer to explicitate all the mutually-recursive groups of methods because we are interested in certifying the code, which includes proving the termination of every recursive method. If we had implicit recursion between all methods of a species (as usual in object-oriented languages), these termination proofs would become too complex, needlessly involving all the methods (whether defined or inherited) of the species. By forcing the programmer to flag the mutually-recursive groups of methods, we ensure that these groups are as small as possible, which helps making the proofs simpler.

Note that this restriction involves a global analysis, as shown by the two following examples. Let A, B, and C be defined as follows:

```
species A = rep; sig x in self; let y = self!x; end
species B = rep; let x = self!y; sig y in self; end
species C inherits A,B = rep = int; end
collection C_imp implements C;;
```

The species A and B are obviously well defined. At first glance, C also seems to be well defined. However, the evaluation of C_imp!x cannot terminate because of the recursion between x and y. On the contrary, the following example illustrates the need of mutually recursive methods:

```
species odd_and_even =
 rep = int;
 let rec odd (x in self) =
   if x = 0 then false else self!even(x−1)
 and    even(x in self) =
   if x = 0 then true else self!odd(x−1);
end
```

Here, the presence of a **let rec** field means that the user has to provide a proof of the termination of the odd and even methods. Once the proofs have been done, it is safe to use these methods.

As far as computing is concerned, the whole point of dependency analysis is to reject the first example while allowing the second one. When we add properties and proofs, the dependency analysis becomes more complex, as we see in Sec. 5. To summarize, the analysis of a species definition must take care of three issues:

- inheritance lookup, and resolution of multiple-inheritance conflicts.
- dependency analysis
- type-checking of the methods

### 3.2. Basic definitions

First, we define $\mathcal{N}(s)$, the method names that are introduced in a field (declaration or definition), and $\mathcal{D}(s) \subseteq \mathcal{N}(s)$ the names introduced in a field definition. This is then extended to species themselves, by induction on the inheritance graph. This graph is indeed a DAG, since a species can only inherit from already-defined species. Formal definition can be found in appendix A.

A method name cannot be introduced twice in a species body. From a practical point of view, it is always a mistake to give two definitions of the same method in a species body, because one of the definitions would be useless. The following analyses will keep this unicity, so that a species in normal form (see def. 7) will have at most one definition for each of its methods. In the remainder of this section, we consider a species definition of the following form, which will be noted *defspec*:

$$\textbf{species } s \textbf{ inherits } s_1..s_n = \phi_1..\phi_m$$

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

Then, for every $x$ in $\mathcal{N}(s)$, we define $\mathcal{B}_s(x)$ to be the body of $x$ in $s$. If $x$ is inherited from another species, we take the body from that species. If $x$ is inherited from several other species, we take the one that is mentioned last in the **inherits** clause.

DEFINITION 1 (binding of a method x in a species s).
*Let $x \in \mathcal{N}(s)$ be the name of a method of s (defined by defspec)*

- *if $x \notin \mathcal{D}(s)$, then $\mathcal{B}_s(x) = \perp$.*
- *if $x = rep$, and $\exists i \leq m$, $\phi_i$ is* ***rep*** *$= \tau$, then $\mathcal{B}_s(rep) = \tau$*
- *if $\exists i \leq m$, $\phi_i$ is* ***let*** *$x = expr$ then $\mathcal{B}_s(x) = expr$*
- *if $\exists i \leq m$, $\phi_i$ is* ***let rec*** *$\{x_1 = e_1; \ldots; x_n = e_n\}$, and $x_j = x$ then $\mathcal{B}_s(x) = expr_j$*
- *else $\exists i_0 \leq n$, $x \in \mathcal{N}(s_{i_0})$ and $\forall i > i_0, x \notin \mathcal{D}(s_i)$, and $x \notin \bigcup_{i=1}^{m} \mathcal{D}(\phi_i)$ then $\mathcal{B}_s(x) = \mathcal{B}_{s_{i_0}}(x)$*

By definition of $\mathcal{D}(s)$ we do not have other cases.

### 3.3. Well-typed Species

The methods of species and collections are not polymorphic. Instead, we use parameterized species (see 3.8), which provides genericity. With unbounded polymorphism in methods, we could build up inconsistent species, as shown in appendix E. On the contrary, local definitions inside a species body can be polymorphic. We denote by $\mathcal{F}(\tau)$ the set of free type variables that occur in type $\tau$.

DEFINITION 2 (Concrete type). *A type $\tau$ is said to be* concrete *if and only if $\mathcal{F}(\tau) = \emptyset$ ($\tau$ may contain names of collection parameters)*

The typing environment of FOC is composed of four sets: $\Delta, \Pi, \Gamma, \Sigma$, which denote respectively the existing collections, species, variables, and the methods of **self**. Elements of these sets have the following form:

- $c : \langle x_i : \tau_i \rangle \in \Delta$ where the $\tau_i$'s are concrete types
- $s : \{x_i : \tau_i = e_i\} \in \Pi$ where the $\tau_i$'s are concrete types
- $x : \forall \alpha_i, \tau \in \Gamma$
- $x : \tau = e \in \Sigma$

Typing rules for expressions are then basically the same as in the Hindley-Milner type inference algorithm. Fig. 1 presents these rules. We define as usual three auxiliary functions, *mgu*, *Gen* and *Inst*. *mgu* tries to unify two types $\tau_1$ and $\tau_2$, and may instantiate some type variables during this process. In addition to the usual algorithm, we provide two new rules [SELF1] and [SELF2] to cover unification steps between **self** and $\tau$ when **rep** is defined to $\tau$. The *mgu* rules are given in B.

DEFINITION 3 (Generalization and Instantiation). *Let $\Gamma$ be a typing environment, and $\tau$ a type.*
Gen$(\tau, \Gamma) = \forall \alpha_i.\tau$, *where* $\{\alpha_i\} = \mathcal{F}(\tau) \backslash \mathcal{F}(\Gamma)$
Inst$(\forall \alpha_i.\tau, \Gamma) = \tau[\alpha_i \leftarrow \alpha'_i]$ *where the $\alpha'_i$ do no appear free in $\Gamma$.*

We can now formally introduce the notion of well-typed species: every method is well-typed, and inherited methods keep their types.

DEFINITION 4 (Well-typed species). *Given a species $s$ defined by defspec.*

$$\textit{Well-typed-spec}\quad \frac{\begin{array}{c}\forall j,\ \forall x_i \in \mathcal{N}(\phi_j),\ \Delta, \Pi, \Gamma, \{x_i : \tau_i = \mathcal{B}_s(x_i)\} \vdash \mathcal{B}_s(x_i) : \tau_i \\ \forall i,\ \forall j, \text{s.t.} x_i \in \mathcal{N}(s_j), \{x_i : \tau_i = \mathcal{B}_{s_j}(x_i)\} \in \Pi(s_j)\end{array}}{\Delta, \Pi, \Gamma, \emptyset \vdash s : \{x_i : \tau_i = \mathcal{B}_s(x_i)\}}$$

*Given such a species $s$, we define* $\forall x_i \in \mathcal{N}(s)$, $\mathcal{T}_s(x_i) = \tau_i$.

## 3.4. Introducing Dependencies

After a first step of typing, we now define the second step of the static analysis, the detection of dependencies cycle between the methods of a species. A method $m_1$ depends on the method $m_2$ if the name $m_2$ is used in $m_1$'s body. So, we first introduce $\wr e \int$ that takes an expression $e$ and returns the set of the methods of **self** that are used in $e$.

[var]
$$\frac{x : \forall \alpha_i.\tau' \in \Gamma \qquad \tau \leq \mathrm{Inst}(\forall \alpha_i.\tau', \Gamma)}{\Delta, \Pi, \Gamma, \Sigma \vdash x : \tau}$$

[abs]
$$\frac{\Delta, \Pi, \Gamma + x : \tau_1, \Sigma \vdash e : \tau_2}{\Delta, \Pi, \Gamma, \Sigma \vdash \textbf{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

[let]
$$\frac{\Delta, \Pi, \Gamma, \Sigma \vdash e_1 : \tau_1 \qquad \Delta, \Pi, \Gamma + x : \mathrm{Gen}(\tau_1, \Gamma), \Sigma \vdash e_2 : \tau_2}{\Delta, \Pi, \Gamma, \Sigma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2}$$

[let rec]
$$\frac{\begin{array}{c}\Delta, \Pi, \Gamma + x : \mathrm{Gen}(\tau_1, \Gamma), \Sigma \vdash e_1 : \tau_1 \\ \Delta, \Pi, \Gamma + x : \mathrm{Gen}(\tau_1, \Gamma), \Sigma \vdash e_2 : \tau_2\end{array}}{\Delta, \Pi, \Gamma, \Sigma \vdash \textbf{let rec } x = e_1 \textbf{ in } e_2 : \tau_2}$$

[app]
$$\frac{\Delta, \Pi, \Gamma, \Sigma \vdash e_0 : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau \qquad \forall i \ \Delta, \Pi, \Gamma, \Sigma \vdash e_i : \tau_i}{\Delta, \Pi, \Gamma, \Sigma \vdash e_0(e_1, \ldots, e_n) : \tau}$$

[meth call] [self call]
$$c \neq \textbf{self} \, \frac{c \in \Delta \qquad x : \tau \in \Delta(c)}{\Delta, \Pi, \Gamma, \Sigma \vdash c!x : \tau} \qquad \frac{x : \tau = expr \in \Sigma}{\Delta, \Pi, \Gamma, \Sigma \vdash \textbf{self!}x : \tau}$$

*Figure 1.* typing rules for basic expressions

Then we extend this to field definitions, with a distinction between **let** and **let rec** definitions. Namely, in a **let rec** definition $\phi$, we erase the mutual dependencies between the methods defined inside it. Indeed, we only want to detect dependency cycles that occur outside of **let rec** fields. Appendix C gives a formal definition of these dependencies.

Finally, we define $\wr x \int_s$ to be the dependencies of a method $x$ on the methods of the species $s$ in which $x$ is defined. As for $\mathcal{B}_s(x)$, we consider the last definition in the order given by the **inherits** statement.

DEFINITION 5 (dependencies in a species).
*Let $s$ be a species defined by defspec. Then, $\forall x \in \mathcal{D}(s)$:*

- *if* $\exists j \leq m, \ x \in \mathcal{D}(\phi_j)$ *then* $\wr x \int_s = \wr \phi_j \int$
- *if* $\forall j \leq m, \ x \notin \mathcal{D}(\phi_j) \wedge \exists i_0 \leq n, \ x \in \mathcal{D}(s_{i_0}) \wedge \forall i > i_0 \ x \notin \mathcal{D}(s_i)$ *then* $\wr x \int_s = \wr x \int_{s_{i_0}}$

This leads to the notion of *well-formed* species, where there is no cycle of dependencies *outside* a **rec**-structure. Only well-formed *and* well-typed species are acceptable in FOC.

DEFINITION 6 (well-formedness).
$x_1 \blacktriangleleft_s x_2 \widehat{=} \exists \{y_i\}_{i=1\ldots n}$ s.t. $y_1 = x_1, y_n = x_2, \forall i < n,\ y_{i+1} \in \wr y_i \int_s$.
*We say that $s$ is **well-formed** if $\forall x \in \mathcal{N}(s)\ \neg (x \blacktriangleleft_a x)$.*
*In addition, $s$ must inherit only from well-formed species.*

DEFINITION 7 (normal form). *Let nf be a species defined by:*

$$\textbf{\textit{species}}\ nf = \phi_1 \ldots \phi_n\ \textbf{\textit{end}}$$

*nf is said to be in* normal form *if:*

- *There is no **inherits** clause*
- *It is 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 depends only upon previous fields:*

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

### 3.5. Merging two fields

Let $s$ be a species defined by *defspec*. To check that it is well-typed and well-formed, we create a species *nfs* that is equivalent to it, in the sense that it shares the same definitions (and declarations). Intuitively, $s$ and *nfs* cannot be distinguished from each other from "outside": they react in the same way to all method calls.

This is done by induction on the inheritance graph. In the following, we will assimilate a species in normal form and the sequence of all its definitions (its body). $a_1@a_2$ denotes the concatenation of two sequences. If $s$ does not have an **inherits** clause, then reordering its fields and typing each method is straightforward. Otherwise, let $norm(s_i)$ be the normal forms of $s_i$ and $\mathbb{W}_1 = norm(s_1)@...@norm(s_n)@[\phi_1, ..., \phi_m]$. $\mathbb{W}_1$ may contain several occurrences of the same name, due to multiple inheritance or redefinition. So we build a new sequence, $\mathbb{W}_2$, from $\mathbb{W}_1$, in which each name is introduced only once. $\mathbb{W}_2$ is identified to a species $\tilde{s}$: **species** $\tilde{s} = \mathbb{W}_2$ **end**. We prove that $\tilde{s}$ is well-formed if $a$ is well-formed.

To build $\mathbb{W}_2$ from $\mathbb{W}_1$, we must find a precise way to resolve "conflicts" (multiple definitions of the same method) in inheritance. To do that, we provide a function $\otimes$ to merge two fields $\phi_1$ and $\phi_2$ that have some names in common. This is not a total function because a name might be defined with two incompatible types. In this case, the definition is considered ill-typed and rejected by $\otimes$. Two **let rec** fields 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), and also add some new methods to this recursion. In this case, the merging function will take every method that are involved in at least one of the two mutual recursive definitions. This will also imply a new termination proof (see 5.2), involving *all* the mutually defined functions, including the inherited ones that are not redefined. The full definition of $\otimes$ is given in appendix D.

The operator $\otimes$ enjoys two important properties. First, it preserves all the names introduced by $\phi_1$ or $\phi_2$ in one of the definition, and if a method is defined in $\phi_1$ or $\phi_2$, then it is also defined in $\phi_1 \otimes \phi_2$. Second, it is compatible with *late binding*, which requires that a method call always uses the "newest" definition available for it in the inheritance path.

*Proposition 1. (names preservation)* $\forall \phi_1, \phi_2$ st $\mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) \neq \emptyset$

$$\mathcal{N}(\phi_1 \otimes \phi_2) = \mathcal{N}(\phi_1) \cup \mathcal{N}(\phi_2)$$

Same property holds for $\mathcal{D}\,()$

*Proposition 2. (late binding)* $\forall \phi_1, \phi_2$ s.t. $\mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) \neq \emptyset$

$$\begin{cases} \forall x \in \mathcal{D}\,(\phi_2)\,, & \mathcal{B}_{\phi_1 \otimes \phi_2}(x) = \mathcal{B}_{\phi_2}(x) \\ \forall x \in \mathcal{D}\,(\phi_1) \setminus \mathcal{D}\,(\phi_2)\,, & \mathcal{B}_{\phi_1 \otimes \phi_2}(x) = \mathcal{B}_{\phi_1}(x) \end{cases}$$

This property is interesting only if neither $\phi_1$ nor $\phi_2$ is a **sig**. Otherwise, we deal with empty sets, and the property is trivial.

*Proof.* immediate by case analysis on the structure of $\phi_1$ and $\phi_2$. □

We then build a sequence $\mathbb{W}_2$ of definitions from $\mathbb{W}_1$, by analysing its elements one by one in the order of the list. This is done inside a loop, starting with $\mathbb{W}_1 = \phi_1 \dots \phi_n$ and $\mathbb{W}_2 = \emptyset$. At each step, we examine the first field remaining in $\mathbb{W}_1$ and we update $\mathbb{W}_1$ and $\mathbb{W}_2$. The loop ends when $\mathbb{W}_1$ is empty. The loop body is the following:
Let $\mathbb{W}_1 = \phi, \mathbb{X}$ and $\mathbb{W}_2 = \psi_1 \dots \psi_m$

- if $\mathcal{N}(\phi) \cap \mathcal{N}(\mathbb{W}_2) = \emptyset$ 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 $\mathcal{N}(\phi) \cap \mathcal{N}(\psi_{i_0}) \neq \emptyset$, then we do $\mathbb{W}_1 \leftarrow ((\phi \otimes \psi_{i_0}), \mathbb{X})$ and $\mathbb{W}_2 \leftarrow (\psi_1 \dots \psi_{i_0-1}, \psi_{i_0+1} \dots \psi_m)$. This time, we must use $\otimes$. However, in the case of mutually recursive definitions, $\phi$ can have some names in common with more than one $\psi_i$, so that $\phi \otimes \psi_{i_0}$ is kept in $\mathbb{W}_1$.

To ensure the termination of the algorithm, we take the following lexicographic ordering: $(\mathrm{Card}\mathbb{W}_1, \mathrm{Card}\mathbb{W}_2)$. Indeed, let $\widetilde{\mathbb{W}_1}$ and $\widetilde{\mathbb{W}_2}$ be the values computed after one step in the loop. If there wasn't any conflict, then $\mathrm{Card}\,\widetilde{\mathbb{W}_1} < \mathrm{Card}\,\mathbb{W}_1$. Else, we have $\mathrm{Card}\,\widetilde{\mathbb{W}_1} = \mathrm{Card}\,\mathbb{W}_1$, and $\mathrm{Card}\,\widetilde{\mathbb{W}_2} < \mathrm{Card}\,\mathbb{W}_2$. □

We now establish the main properties of this algorithm, in order to show that $\mathbb{W}_2$ defines a well-formed species equivalent to $s$. We use the same notations as above to speak about the fields of $\mathbb{W}_1$ and $\mathbb{W}_2$.

*Proposition 3. (Well-typed merging)* With the notations above, if $s$ is well typed, then $\psi_{i_0} \otimes \phi$ never fails.

*Proof.* This is straightforward with the definition of $\otimes$ and def.4. □

*Proposition 4. (unicity)* $\forall \phi_1, \phi_2 \in \mathit{nfs},\ \mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) = \emptyset$.

*Proof.* By induction on the length of $\mathbb{W}_1$: If there is only one definition, then this is trivial. If there are $n+1$ definitions, we can use the induction hypothesis for the first $n$ steps. It remains to add the last definition, $\phi$.

– If $\forall \psi \in \mathbb{W}_2, \mathcal{N}(\phi) \cap \mathcal{N}(\psi) = \emptyset$ then we can safely add it

– Else, we conclude by induction on the size of $\mathcal{N}(\phi) \cap \left(\bigcup_{\psi \in \mathbb{W}_2} \mathcal{N}(\psi)\right)$: if it concerns only one name $x \in \mathcal{N}(\phi)$, then, $\psi_{i_0}$ is the (*only*) definition in $\mathbb{W}_2$ such that $x \in \mathcal{N}(\psi_{i_0})$, we have

$$\mathcal{N}(\psi_{i_0} \otimes \phi) \cap \left( \bigcup_{\chi \in \mathbb{W}_2 \setminus \psi_{i_0}} \mathcal{N}(\chi) \right) = \emptyset$$

Indeed, by definition of $\otimes$, $\forall x \in \mathcal{N}(\psi_{i_0} \otimes \phi_1),\ x \in \mathcal{N}(\psi_{i_0}) \vee x \in \mathcal{N}(\phi_1)$. In both cases, we cannot find $x$ in any of the remaining $\psi_i$, by induction.

– If there are $m+1$ names involved, then with the same notations as above, $|\ \mathcal{N}(\phi) \cap (\cup_{i \neq i_0} \mathcal{N}(\psi_i))\ | \leq m$. Namely, by induction hypothesis any name introduced in $\psi_{i_0}$ does not appear anywhere else in $\mathbb{W}_2$. So every $x \in \mathcal{N}(\phi_1) \cap \mathcal{N}(\psi_{i_0})$ disappears from the intersection while we do not add any new identifier. □

As said above, we now define $\tilde{s}$ as **species** $\tilde{s} = \mathbb{W}_2$ **end**.

*Proposition 5. (equivalence)* $\mathcal{N}(s) = \mathcal{N}(\tilde{s})$, $\mathcal{D}(s) = \mathcal{D}(\tilde{s})$, and $\forall x \in \mathcal{D}(\tilde{s})\,,\ \mathcal{B}_s(x) = \mathcal{B}_{\tilde{s}}(x)$

*Proof.* In fact, we just have to prove that the following properties hold at each step:

$$\mathcal{N}(s) \ = \ \bigcup_{i=1}^{n} \mathcal{N}(s_i) \ \cup \ \bigcup_{j=1}^{m} \mathcal{N}(\phi_j) = \bigcup_{\phi \in \mathbb{W}_1} \mathcal{N}(\phi) \ \cup \ \bigcup_{\psi \in \mathbb{W}_2} \mathcal{N}(\psi)$$

$$\forall x \in \mathcal{N}(s), \exists \phi \in \mathbb{W}_2 \cup \mathbb{W}_1, \mathcal{B}_s(x) \in \phi$$

At first step, this is true, since $\mathbb{W}_1$ contains all the definitions found in $s$ or its parents. Suppose that the properties are still true after $n$ steps. Let $\phi_1$ be the definition to be analyzed.

If $\forall \phi \in \mathbb{W}_2\ \mathcal{N}(\phi_1) \cap \mathcal{N}(\phi) = \emptyset$ then we just move $\phi_1$ from $\mathbb{W}_1$ to $\mathbb{W}_2$, so that neither the set of names appearing in one of them , nor the associated definitions change.

Else, with $\psi_{i_0}$ such that $\mathcal{N}(\phi_1) \cap \mathcal{N}(\psi_{i_0}) \neq \emptyset$: *names preservation* property says that $\mathcal{N}(\psi_{i_0} \otimes \phi) = \mathcal{N}(\psi_{i_0}) \cup \mathcal{N}(\phi)$, so that the global namespace is left unchanged. Besides, this is true for $\mathcal{D}(\cdot)$.

Moreover, the *late binding* property shows that the methods bodies that are removed are $\left\{\mathcal{B}_{\psi_{i_0}}(x), x \in \mathcal{D}(\psi_{i_0}) \cap \mathcal{D}(\phi_1)\right\}$. Since $\mathbb{W}_1$ is ordered, we have, by definition of $\mathcal{B}_s(x)$

$$\forall x \in \mathcal{D}(\psi_{i_0}) \cap \mathcal{D}(\phi)\, \mathcal{B}_s(x) \neq \mathcal{B}_{\psi_{i_0}}(x)$$

At the end of the construction, $\mathbb{W}_1 = \emptyset$, so that $\mathcal{N}(x) = \bigcup_{\psi \in \mathbb{W}_2} \mathcal{N}(\psi)$. Moreover, we have one definition for each method: $\mathcal{B}_s(x) = \mathcal{B}_{\tilde{s}}(x)$. □

We can now state the main result of this section:

THEOREM 1 (normal form of a species).

*For each well-formed and well-typed species s, there exists a species nfs, which is in* normal form *and enjoys the following properties:*

- *names:* $\mathcal{N}(nfs) = \mathcal{N}(s)$ *and* $\mathcal{D}(nfs) = \mathcal{D}(s)$
- *definitions:* $\forall x \in \mathcal{D}(s), \mathcal{B}_s(x) = \mathcal{B}_{nfs}(x)$

*Proof.* This follows directly from Props. 3, 4 and 5. It just remains to prove that $\tilde{s}$ is well formed.

DEFINITION 8. *Let s be a species without* **inherits** *clause, such that*

$$\forall x \in \mathcal{N}(s), \exists! \phi \in s, x \in \mathcal{N}(\phi)$$

*We will note* $\mathcal{D}_s(x)$ *the (unique) definition in s where x appears.*

*Proposition 6.* Using above notations, $\forall x \in \mathcal{N}(s)$, with $D_x = \{\phi \in \mathbb{W}_1 \mid x \in \mathcal{N}(\phi)\}$, we have $\mathcal{N}(\mathcal{D}_{\tilde{s}}(x)) = \left(\bigcup_{\phi \in D_x} \mathcal{N}(\phi)\right)$

*Proof.* Once again, we will state a property verified at each step of the construction of $\mathbb{W}_2$, namely, that

$$\left(\bigcup_{x \in \mathcal{N}(\phi)} \mathcal{N}(\phi)\right) \cup \mathcal{N}(\mathcal{D}_{\mathbb{W}_2}(x)) = \bigcup_{\phi \in D_x} \mathcal{N}(\phi)$$

This is trivial at first step, when $\mathbb{W}_2$ is still empty and $\mathbb{W}_1$ contains all the fields involved in $s$. If it is still verified after $n$ steps, then with $\phi$ the first field of $\mathbb{W}_1$, we have three possible cases:

- if we can add $\phi$, then nothing is changed for the union.
- $x \notin \phi, x \notin \psi_{i_0}$. Then the definitions where $x$ appear do not change.
- $x \in \phi_1$, or $x \in \psi_{i_0}$. Then we remove $\psi_{i_0}$, but add $\psi_{i_0} \otimes \phi$. Since $\mathcal{N}(\psi_{i_0} \otimes \phi) = \mathcal{N}(\psi_{i_0}) \cup \mathcal{N}(\phi_1)$, by Prop 1, the property is preserved.

At the end, we are left with $\{\mathcal{D}_{\mathbb{W}_2}(x)\}$, and the property holds. □

We can now prove that $\tilde{s}$ is well-formed: if this wasn't the case, consider $x_1$ and $x_2$ in $\mathcal{N}(\tilde{s})$ such that, by definition of *(non) well-formedness*, $x_1 \blacktriangleleft_{\tilde{s}} x_2 \blacktriangleleft_{\tilde{s}} x_1$. Then, since $\forall x \in \mathcal{D}(\tilde{s})$, $\mathcal{B}_{\tilde{s}}(x) = \mathcal{B}_s(x)$, by *mutual rec* $\mathcal{D}_s(x_1) = \mathcal{D}_s(x_2)$, so that $x_2 \in \cup_{D_{x_1}} \mathcal{N}(\phi) = \mathcal{N}(\mathcal{D}_{\tilde{s}}(x_1))$, and $\mathcal{D}_{\tilde{s}}(x_1) = \mathcal{D}_{\tilde{s}}(x_2)>$ Then this common definition is a **let rec** definition, in contradiction with $x_1 \blacktriangleleft_{\tilde{s}} x_2$, so that $\tilde{s}$ is well-formed.

Since $\tilde{s}$ is well-formed, $\blacktriangleleft_{\tilde{s}}$ is a strict ordering. Then, we just have to reorder the fields of $\mathbb{W}_2$ according to $\blacktriangleleft_{\tilde{s}}$. □

### 3.7. Collections

A collection $c$ can only be created from a completely defined species $s$. In addition, we abstract its carrier type and all the methods.

[coll]

$$\frac{s : \{x_i : \tau_i = e_i\} \in \Pi \qquad \forall i\ e_i \neq \bot}{\Delta, \Pi, \Gamma, \Sigma \vdash \textbf{collection}\ c\ \textbf{implements}\ s : \langle x_i : \tau_i[\textbf{self} \leftarrow c]\rangle}$$

As often in mathematics, we denote by the same name the collection, and its carrier type, that is the set on top of which the collection is built: in the types of the interface, **self** is replaced by the collection's name.

### 3.8. Parameterized Species

First, we define a function $\mathcal{A}$ that takes a species $s$ and a name $c$ and returns an interface (abstracting all the methods and replacing **self** by $c$ in the types). Indeed a collection parameter, of the form "`c` **is** `s`" adds a collection `c` of interface $\mathcal{A}(\mathtt{s}, \mathtt{c})$ in the environment.

DEFINITION 9 (abstraction). *Let $s = \{x_i : \tau_i = e_i\}_{i=1..n}$ be a typed species, and $c$ a collection name. Then*

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

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 allow such instantiations.

DEFINITION 10 (sub-species). *Let $s_1$, $s_2$ be two species.*

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

*$\mathcal{T}_{s_i}(x)$ being defined as in def.4*

By def.4, if $s_1$ inherits from $s_2$, then $s_1 \preccurlyeq s_2$. Since only the types of the methods are concerned, the relation is easily extended to interfaces.

We can now present the typing rules for parameterized species.

[ent-prm]

$$\frac{\Delta,\Pi,\Gamma + x : \tau,\Sigma \vdash \textbf{species}\ s(prms)\ expr\_spec : (x\ \textbf{in}\ \tau)type\_spec}{\Delta,\Pi,\Gamma,\Sigma \vdash \textbf{species}\ s(x\ \textbf{in}\ \tau, prms)\ expr\_spec : type\_spec}$$

[coll-prm]

$$\frac{\Delta,\Pi,\Gamma,\Sigma \vdash i : inst \qquad \begin{array}{c}\Delta + c : \mathcal{A}(inst, c),\Pi,\Gamma,\Sigma \vdash \textbf{species}\ s(prms) \\ expr\_spec : type\_spec\end{array}}{\begin{array}{c}\Delta,\Pi,\Gamma,\Sigma \vdash \textbf{species}\ s(c\ \textbf{is}\ i, prms)\ expr\_spec : \\ (c\ \textbf{is}\ \mathcal{A}(inst, c))type\_spec\end{array}}$$

[ent-inst]

$$\frac{(s, (x \in \tau)type\_spec) \in \Pi \qquad \Delta,\Pi,\Gamma,\Sigma \vdash e : \tau}{\Delta,\Pi,\Gamma,\Sigma \vdash s(e) : type\_spec}$$

[coll-inst]

$$\frac{(s, (c_1\ \textbf{is}\ i_1)type\_spec) \in \Pi \qquad c_2, i_2 \in \Delta \qquad i_2 \preccurlyeq \mathcal{A}(i_1, c_2)}{\Delta,\Pi,\Gamma,\Sigma \vdash s(c_2) : type\_spec[c_1 \leftarrow c_2]}$$

[prm-inherit]

$$\frac{\begin{array}{c}\Delta,\Pi,\Gamma,\Sigma \vdash s_i : \{x_{i,j} : \tau_{i,j} = e_{i,j}\} \\ \Delta,\Pi,\Gamma,\Sigma \vdash \textbf{species}\ s\ \textbf{inherits}\{x_{i,j} : \tau_{i,j} = e_{i,j}\} = defs : \\ \{y_j : \sigma_j = e'_j\}\end{array}}{\Delta,\Pi,\Gamma,\Sigma \vdash \textbf{species}\ s\ \textbf{inherits}\ s_i = defs : \{y_j : \sigma_j = e'_j\}}$$

## 4. Effective computation: translation to OCAML

The initial development of FOC has been done in OCAML [17], a functional language of the ML family developed at INRIA. The OCAML produces a very efficient code, while the language has a clear semantics and features that cover the needs of FOC, as objects and modules[3]. Indeed, species can be quite easily translated into **classes**. Species inheritance in FOC is translated by class inheritance in OCAML. A collection is represented as an **object** (*i.e* an instance) of the class that correspond to the species it implements. Type abstraction is achieved through a **module** wrapper around the **object**. Functions outside the module can only access *entities* through the methods of the **object**.

### 4.1. TRANSLATION OF SPECIES

Translation of expressions is straightforward. Each method in FOC is translated into a method in OCAML. There is no difference between a method whose body is a constant, and a method that is a function. This is a problem for constant methods because in the object-oriented context of OCAML they are evaluated at each call. So we can optimize the translation by using instance variables in OCAML to represent constant methods of FOC. These instance variables must be initialized when a new object is created. The order in which variables are initialized is exactly the order given by the dependency analysis of FOC.

### 4.2. PARAMETERS

Parameters in FOC are translated by abstracting the **class** definition with respect to them. For instance, to define modular integers from any possible implementation of integers, we would have the FOC definition:

```
species int_mod (base is int_spec, n in base) ...
```

which is translated into

```
class ['t,'base_carrier,'base_imp] int_mod =
  fun (base:'base_imp) -> fun (n:'base_carrier) ->
    object
     constraint 'base_imp = ['base_carrier]#int_spec
     ...
    end
```

Each collection parameter adds two type parameters in the OCAML translation. the first one (`base_carrier`) correspond to its carrier type. The second one (`base_imp`) allows us to instantiate `int_mod` with any instance of a subclass of `int_spec`. We also add a type constraint which expresses the fact that `'base_carrier` is the carrier of `base`.

## 5. Adding Properties and Proofs

### 5.1. SYNTAX

We now extend the syntax of 2.2 with two new field definitions: **theorem** and **property**. In fact, a proof is a term that is built from other proofs by composition, substitution, etc. This is analogous to the construction of a program. The type verification of this proof term in the COQ language ensures the validity of the proof itself.

| | | |
|---|---|---|
| def_field | ::= | … \| **theorem** $x$ = *prop* **proof:** [ deps ] *proof* |
| deps | ::= | { (**decl:** \| **def:**) { $x_i$ }* }* |
| field | ::= | … \| **property** $x$ = *prop* |
| prop | ::= | *expr* \| *prop* **and** *prop* \| *prop* **or** *prop* \| *prop* → *prop* |
| | \| | **not** *prop* \| **all** x **in** *typ*, *prop* \| **ex** x **in** *typ*, *prop* |

For the time being, a proof is a Coq script that corresponds to the translation of the property it is bound to. A proof language dedicated to Foc is under development.

### 5.2. Def-Dependencies and Decl-dependencies

When talking about proofs, the notion of dependency introduced in the previous sections (see 3.4) becomes too weak. For instance, we can express that `plus` in the species `monoid` is associative:

```
property assoc =
  all x,y,z in self,
   self!eq(self!plus(self!plus(x,y),z)
         self!plus(x,self!plus(y,z)))
```

then we are able to prove this property for the naive implementation of `plus` on Peano's integer: $0 + n = n$ and $succ(m) + n = succ(m + n)$. It is done by induction on $x$, $y$ and $z$, and uses *the exact definition* of `plus`. In other words, the proof of `assoc` that we obtain depends upon *the definition* of `plus`, while dependencies we have seen so far were only upon the type of the methods. We call *def-dependencies* this new kind of dependencies, and speak of *decl-dependency* when only the type of the method is needed. We need to avoid cycles of *dependencies* –both *decl*- and *def*- ones. The distinction between the two occurs during inheritance resolution, when a method is redefined. We must now erase every proof that *def*-depends upon this method, and prove the property for the new implementation. For instance, if we decide to use a more efficient algorithm, that uses internally a binary representation of integers for `plus`, the old proof of `assoc` is not correct anymore.

Apart from explicitly stated properties, some proof obligations are requested by Foc.

- For every **let rec** definition, we have to prove that any call to a method of $\mathcal{N}(rec\_def)$ ends. This proof has def-dependencies upon all the methods in $\mathcal{N}(rec\_def)$. Even if only one of the method is redefined, a new proof for *all* the methods involved is requested.

- For each species that has an equality (that is that derives from setoid), we must prove that each function that uses entities of this species is compatible with this equality.

- For some particular representations, additional proofs are requested. For instance, if we work with the native integers of Ocaml, we must ensure that there is no overflow.

### 5.3. Dependencies upon the carrier

Until now, it is syntactically impossible to define a carrier that depends upon a method, but conversely, "normal" methods often depend upon the carrier. We have to know if we use decl- or def- dependencies. Given an expression $e$, if a sub-expression of $e$ has type **self**, then there is a decl-dependency upon the carrier. Def-dependencies upon the carrier can be detected during the typing phase. More precisely, if an unification must use one of the two SELF rules of *mgu*, then there is a def-dependency. Indeed $e$ cannot be typed in an environment where the carrier is bound to another type. For instance, if we define

```
species counter =
  rep = int;
  let inc in self -> self = fun x -> x + 1;
end
```

`inc` def-depends upon the carrier `int`: in order to type `inc` with **self** $\rightarrow$ **self**, we must know that **self** is bound to `int`.

Such def-dependencies addresses a new issue, since they may occur in statements too. For instance, we can add to `counter` the theorem:

```
theorem inc_spec : all x in self, self!inc(x) >= x + 1 proof: ...
```

The statement of `inc_spec` has a def-dependency on the carrier, so that it would be impossible to build the interface of `counter`: **rep** cannot be abstracted. Since we want to build interfaces for each species (1.3.2), such species definition must be rejected.

In Fig. 5.3, we define $\Sigma^* = \Sigma[\mathcal{B}_s(\mathit{rep}) \leftarrow \perp]$ which hide the concrete representation of **rep** when typing the statements. This ensures that def-dependencies upon **rep** at the property level are rejected as ill-typed. The Expr rule coerce every boolean expression used in a statement into *prop*. Following Curry-Howard-de Bruijn isomorphism, we also extend $\mathcal{T}_s(x)$ to **theorem** and **property** fields as being the statement of $x$ in $s$.

### 5.4. Inheritance Lookup

We extend straightforwardly the definitions of the preceding section. $\wr \cdot \int$ now denotes decl-dependency, while $\wr\wr \cdot \iint$ denotes def-dependencies.

DEFINITION 11 (binding of a method). *Let $s$ be 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* $\mathcal{B}_s(x) = \perp$.

$$\frac{\Delta, \Pi, \Gamma, \Sigma^* \vdash expr : bool}{\Delta, \Pi, \Gamma, \Sigma \vdash expr : prop}\ [\text{expr}] \qquad \frac{\Delta, \Pi, \Gamma, \Sigma \vdash p : prop}{\Delta, \Pi, \Gamma, \Sigma \vdash \mathbf{not}\ p : prop}\ [\text{not}]$$

$$\frac{\Delta, \Pi, \Gamma, \Sigma \vdash p_1 : prop \qquad \Delta, \Pi, \Gamma, \Sigma \vdash p_2 : prop}{\Delta, \Pi, \Gamma, \Sigma \vdash p_1 \ \bullet \ p_2 : prop}\ [\text{and}]$$

$$\frac{\Delta, \Pi, \Gamma + x : \tau, \Sigma \vdash p : prop}{\Delta, \Pi, \Gamma, \Sigma \vdash \perp\ x\ \mathbf{in}\ \tau, p : prop}\ [\text{ex}]$$

with $\bullet \in \{\mathbf{and}, \mathbf{or}, \rightarrow\}$ and $\perp \in \{\mathbf{all},\ \mathbf{ex}\}$

*Figure 2.* Typing rules for statements

- *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* $\mathbb{I}_s(x) = i_0$

DEFINITION 12 (defined methods). $\mathcal{D}(s) = \{x \in \mathcal{N}(s), \mathcal{B}_s(x) \neq \perp\}$

We then define the dependencies of $x$ in a species $s$. Note that not only proofs but also statements may have decl-dependencies, so that we analyze $\mathcal{T}_s(x)$ as well as $\mathcal{B}_s(x)$.

DEFINITION 13 (dependencies inside a species).

- *if* $x$ *is a function, then its decl-dependencies are defined as in def.5*
- *else,* $\wr x \int_s = \wr \mathcal{B}_s(x) \int \cup \wr \mathcal{T}_s(x) \int$ *and* $\wr\wr x \iint_s = \wr\wr \mathcal{B}_s(x) \iint$

DEFINITION 14. $x_1 \blacktriangleleft_s x_2 \widehat{=} \exists y_{i\, i=1 \ldots n}$ s.t. $y_1 = x_1, y_n = x_2, \forall i < n y_{i+1} \in \wr y_i \int_s \cup \wr\wr y_i \iint_s$.

*We say that* $s$ *is* ***well-formed*** *if* $\forall x \in \mathcal{N}(s) \neg x \blacktriangleleft_s x$.

Each well formed and well-typed species still has a normal form. Due to def-dependencies, some inherited proofs must be removed, but we want to erase as few proofs as possible. So the main result of this section is that for each well-formed species $s$, there exist a *normal form* with the same declarations as $s$, and a subset of its definitions which is maximal *wrt* inclusion.

In order to precisely state the theorem, we need a technical definition. $changed(y,x)$ is true if and only if the definition of $y$ has changed since the last definition of $x$ (following the **inherits** clause of $s$).

DEFINITION 15.
*$changed(y,x)$ is a relation over $\mathcal{N}(s)$, $s$ being defined by defspec*

$$\begin{array}{rcl} changed(y,x) & \iff & \left(\exists j > \mathbb{I}_s(x),\, y \in \mathcal{D}\left(s_j\right) \wedge \mathcal{B}_{s_j}(y) \neq \mathcal{B}_{s_{\mathbb{I}_s(x)}}(y)\right) \\ & \vee & \left(\exists k,\, y \in \mathcal{D}\left(\phi_k\right) \wedge \mathbb{I}_s(x) \neq n+1\right) \end{array}$$

$\ominus$ is extended to merge theorems and properties. *Names-preservation* and *late-binding* are preserved by this extension.

To build *nfs*, the main difference is that we must now take def-dependencies into account when merging two definitions. First, we must slightly refine the construction of $\mathbb{W}_1$ to avoid erasing new definitions.

Let $i_1, \ldots, i_n$ be a permutation of $1 \ldots n$ such that $\forall j < k$, $\mathcal{N}(\phi_{i_j}) \cap \llbracket \phi_{i_k} \rrbracket = \emptyset$. Such a permutation exists by definition of well-formedness. Then $\mathbb{W}_1 = norm(a_1)@\ldots@norm(a_n)@[def_{i_1} \ldots def_{i_n}]$

Then, we build $\mathbb{W}_2$ step by step. Now, if there is a conflict between $\phi_1$ and $\psi_{i_0}$, some proofs of $\mathbb{W}_2$ must be erased. Let $\mathbb{N} = \mathcal{N}(\phi_1) \cap \mathcal{N}(\psi_{i_0})$. Each $d \in \mathbb{W}_2$ s.t. $\mathbb{N} \cap \llbracket d \rrbracket \neq \emptyset$ is replaced by **property** $x = \mathcal{T}_d(x)$[2].

We can now prove that we build a normal form. First, *definition unicity* still holds, by induction on the length of $\mathbb{W}_1$. Then, we still have $\mathcal{N}(nfs) = \mathcal{N}(s)$, since there is no change here. It remains to prove that we do not erase too many proofs.

*Proposition 7. (preservation of definitions)*
$\mathcal{D}\left(nfs\right) \subseteq \mathcal{D}\left(s\right)$
$\forall x \in \mathcal{D}\left(nfs\right), \mathcal{B}_s(x) = \mathcal{B}_{nfs}(x)$
$\forall x \in \mathcal{D}\left(s\right) \setminus \mathcal{D}\left(nfs\right), \exists y \in \llbracket x \rrbracket_s$ s.t.
$y \notin \mathcal{D}\left(nfs\right)$ or $y \in \mathcal{D}\left(nfs\right) \wedge changed(y,x)$.

*Proof.* First, $\mathcal{D}\left(nfs\right) \subseteq \mathcal{D}\left(s\right)$. Indeed, if we don't consider the def-dependencies, we would have the equality, as in preceding section. Moreover, since Prop. 2 still holds, the bodies that are not erased are always

[2] Only theorems have def-dependencies.

the latest definition of the method. It remains to prove that we do not erase too much proofs.

Suppose that we have $x \in \mathcal{D}(s) \setminus \mathcal{D}(\mathit{nfs})$. If $\exists \phi_i$ s.t. $x \in \mathcal{D}(\phi_i)$, then $\exists \phi_j$ s.t. $y \in \mathcal{D}(\phi_j)$ such that $x$ def-depends upon $y$ and $\phi_i$ is analyzed before $\phi_j$. Otherwise, $x$ would be defined in *nfs* too. Since we have reordered the list of definitions, this case cannot occur.

Otherwise, $\exists s_{i_0}$ s.t. $x \in \mathcal{D}(s_{i_0})$, and $\forall y \in \llbracket \mathcal{B}_{s_{i_0}}(x) \rrbracket, \mathcal{B}_s(y) = \mathcal{B}_{s_{i_0}}(y)$ Such an $y$ may be redefined in $s_{i_0}$ itself, but since we work with the normal forms of father species, the corresponding definition is analyzed before the one of $x$ in building $\mathbb{W}_2$. If it was redefined after $s_{i_0}$, then the hypothesis wouldn't hold, so that this case cannot occur. □

It remains to prove that *nfs* is well-formed. Prop 3 and 6 are easily extended, so that we can conclude the same way as above:

THEOREM 2 (normal form of well-formed species). *For each well-formed and well-typed species s, there exists a species nfs, which is in* normal form *and enjoys the following properties:*

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

## 6. Certification: the translation into coq

### 6.1. Overview of the translation

Coq is a proof-checker, based on the Calculus of Inductive Constructions [22]. Its language is very expressive, and it was quite natural to choose it as the target language of the certification part of Foc, because it contains as a sublanguage the purely functional part of Ocaml.

In his PhD [5], S. Boulmé describes a complete axiomatization in Coq of inheritance and decl- and def-dependencies (called *opaque* and *transparent* in his work). In this formalism, the size of the terms grows up very fast with the number of methods and inheritance steps, and these terms soon become too complicated for the Coq type-checker. We then designed a new approach that deals with smaller Coq expressions.

As in Boulmé's work, interfaces are represented by **Records** (see below), and collections by instances of the corresponding **Records**. Our species definitions, however, are quite different. In fact, we build a set of *method generators*. A method generator is a lambda-lifting which produces the body of a method. For instance, in the species a below,

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

The *method generator* for neq is (using Coq syntax)

```
[abst_T:Set][abst_eq:abst_T->abst_T->bool]
   [x,y:abst_T](notb (abst_eq x y))
```

The first line corresponds to the abstractions we have made. The second line is the translation of the method's body *in the environment set up by the appropriated abstractions*. In particular, the call to **self!eq** has been replaced by the variable abst_eq.

To create a collection from a completely defined species $s$, we follow the order of the methods in the normal form of $s$. For each method, we select the corresponding generator and apply it to the preceding methods of the collection.

### 6.2. Dependent Records

Coq **Records** are quite similar to the records of most programming languages: n-uples whose components (called *fields*) are named. The main distinction is that the type of one component may *depend* on the preceding ones, *i.e.* use their labels. For each species defined in Foc, we define a **Record** type in Coq, which denotes its interface. If the species has the form $\{s_i : \tau_i = e_i\}$ in Foc, the **Record** is defined as

$$\textbf{Record}\ \texttt{name\_spec} : \texttt{Type} := \texttt{mk\_spec}\ \{s_i : \tau_i\}$$

In contrast with the OCaml translation, here we explicitly give *all* the fields of the **Record**, in the order given by the normal form.

### 6.3. Method generators

A species definition consists in the creation of the *method generators*, for the functions and theorems that are *(re)defined* in the species itself.

To translate the definition of $x$, we consider the minimal environment $\Sigma$ necessary to type the body of $x$. We keep the methods $x_i$ $x$ depends upon, but we need different handling for decl-dependencies (where the body of $x_i$ can be forgotten) and def-dependencies (where this body must be kept in $\Sigma$). Moreover, the bodies of these def-dependencies are expressions themselves, and we also need to keep their dependencies.

DEFINITION 16 (Minimal Environment). *Let* $\Sigma = \{x_i : \tau_i = e_i\}$ *and* $e$ *be an expression.* $\Sigma \Cap e$ *is defined as follows.*

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

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

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

*where* $\{x : \tau = \perp\} \cup \{x : \tau = e\} = \{x : \tau = e\}$

Since the sequence $U_i$ is growing and bounded by $\Sigma$, it has a least upper bound, so the definition of $\Cap$ is correct. For each identifier present in $\Sigma \Cap d$, we define either a $\lambda$-abstraction (for decl-dependencies) or a local binding (for def-dependencies).

DEFINITION 17 (Translation of a method's environment).

$$\begin{aligned} [\![\emptyset, d]\!] &= [\![d]\!] \\ [\![\{x : \tau = \perp; l\}, d]\!] &= [abst\_x : \tau]\, [\![l, d]\!] \\ [\![\{x : \tau = e; l\}, d]\!] &= \textbf{\textit{Let}}\ abst\_x : \tau := (gen\_x\ abst\_s_i)\, \textbf{\textit{in}}\, [\![l, d]\!] \end{aligned}$$

*where* $gen\_x$ *is the method generator of* $x$ *(inherited or not: according to the dependency analysis, it is defined before this translation).*

Of course, we need to apply $gen\_x$ to some arguments, namely the methods $s_i$ from $\Sigma \Cap x$ that are, by def.16, part of the environment we use. The translation of the body itself is straightforward, except that each method call **self**!x is turned into a variable reference, `abst_x`.

## 7. Related works

There exist a certain number of projects in the area of specification of algebraic structures. Among them, CoFI [7] offers a language, CASL (Common Algebraic Specification Language), in which it might be interesting to express the interfaces of the Foc hierarchy. On the other hand, OpenMath [8] can be very useful to represent Foc entities. OpenMath provides a standard for a semantically-rich representation of mathematical objects in XML. OMDoc, an extension of OpenMath which handles whole documents can also help FOC to provide species descriptions in a standard XML format. However, this concerns only specifications, not the concrete implementation of the algorithms.

Algebraic hierarchies have been developed in various theorem-provers or proof-checkers. In particular, Loic Pottier [20] has developed an huge library in COQ about fundamental notions of algebra, up to fields. H. Geuvers and the FTA project [13] are also using the `Records` of COQ to represent algebraic structures, in order to define abstract and concrete representations of reals and complex. In addition, R. Pollack [19] and G. Betarte [2] have given their own embedding of dependent records in Type Theory. At last, a fairly large amount of proofs has been done inside the Mizar project, which attempts to build a database of important theorems of mathematics. We can also mention Imps [11], 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 [15] 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. In other words, the mathematical hierarchy is not fully reflected in this construction.

## 8. Future work

### 8.1. Higher-order methods

In addition to the core language, some new features have been added to cover practical requests from the programmers. Some further work is needed to incorporate them in the formalization desribed above. In particular, we need a way to define a method that returns a "collection", *i.e.* a species with all its methods defined. For instance, we can define the species of multivariate polynomials. Assuming that `polynom` is a species parameterized by the name of the variable, the ring of coefficients and the ordered set of degrees, we will define a method that lifts the whole species to the correct number of variables for a given operation:

```
let updom (s in string)=polynom(s,self,my_degree)
```

It is not clear yet how we can extend dependency analysis to such methods. The most simple solution seems to consider that `updom` decl-depends upon all the methods of `self` that `polynom` can see through the interface `ring`.

8.2. The proof language

As said in sec. 5.1, it is now necessary to define a proof language for Foc that will be independent from Coq scripts. It is yet possible to write some basic proofs directly in Foc but some features are still missing: In particular, proof obligations are not yet generated. In addition, we have to provide more support for induction and rewriting steps.

## 9. Conclusion

As a conclusion, we can say that Foc has now achieved quite good expressive power, at least for its computational part. The static analyses that are discussed in sec. 3 and 5 have been successfully implemented in a compiler that generates Ocaml as well as Coq code, following the ideas of sec 4 and 6.

On the one hand, we are able to provide a good environment to prove the properties that are needed in each species' implementation. We still have to specify a language to build proofs in this environment.

On the other hand, the Ocaml code produced by Foc conforms to the initial requirements of the project. Moreover, the generated code is quite efficient, thanks to the optimizations allowed by static analysis.

## 10. Acknowledgement

The authors wish to thank Thérèse Hardin and Gilles Dowek for her precious help, and the anonymous referees for their precise and useful comments on previous versions of this paper.

## References

1. C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: an interface between isabelle and maple. In A. Levelt, editor, *Proceedings of ISSAC*, pages 150–157, Montréal, Canada, 1995. ACM Press.
2. G. Betarte. *Dependent Record Types and Formal Abstract Reasoning: Theory and Practice*. PhD thesis, University of Göteborg, 1998.
3. 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`.
4. S. Boulmé, T. Hardin, and R. Rioboo. Some hints for polynomials in the Foc project. In *Calculemus 2001 Proceedings*, June 2001.
5. S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*. PhD thesis, Université Paris 6, 2000.

6. B. Buchberger and all. A survey on the theorema project. In W. Kuechlin, editor, *Proceedings of ISSAC'97*. ACM Press, 1997.
7. M. Cerioli, P. Mosses, and G. Reggio, editors. *Proceedings of the 15th International Workshop on Algebraic Development Techniques and the General Workshop of the CoFI WG*, Genova, Italy, April 2001.
8. S. Dalmas, M. Gaëtano, and S. Watt. An openmath 1.0 implementation. In W. Kuechlin, editor, *Proceedings of ISSAC 97*. ACM Press, 1997.
9. J. Davenport, Y. Siret, E. Tournier, and D. Lazard. *Computer Algebra*. Masson, 1993.
10. M. Dunstan, H. Gottliebsen, T. Kelsey, and U. Martin. Computer algebra meets automated theorem proving: A maple-pvs interface. In *Proceedings of the Calculemus Workshop*, 2001.
11. W. M. Farmer, J. D. Guttman, and F. J. Thayer. The IMPS user's manual. Technical Report M-93B138, The MITRE Corporation, November 1995. Available at `ftp://math.harvard.edu/imps/doc/`.
12. S. Fechter. Une sémantique pour foc. Rapport de D.E.A., Université Paris 6, Septembre 2001.
13. H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The algebraic hierarchy of the fta project. In *Proceedings of the Calculemus Workshop*, 2001.
14. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
15. P. Jackson. Exploring abstract algebra in constructive type theory. In *Proceedings of 12th International Conference on Automated Deduction*, 1994.
16. R. D. Jenks and R. S. Stutor. *AXIOM, The Scientific Computation System*. Springer-Verlag, 1992.
17. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.00 Documentation and user's manual*. INRIA, 2000. `http://pauillac.inria.fr/ocaml/htmlman/`.
18. L. Mandel. Factorisation de polynômes sur les corps finis. Rapport de magistère, Université Paris 6, 2001.
19. R. Pollack. Dependently typed records for representing mathematical structures. In *TPHOLs'00*. Springer-Verlag, 2000.
20. L. Pottier. contrib algebra. `http://coq.inria.fr/contribs-eng.html`.
21. V. Prevosto. Vers une interface utilisateur pour foc. Rapport de D.E.A., Université Paris 6, Septembre 2000.
22. The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Projet LogiCal, INRIA-Rocquencourt – LRI Paris 11, Nov. 1996.

## Appendix

## A. Names introduced in a field and a species

DEFINITION 18. *$\mathcal{N}(def)$ and $\mathcal{D}(def)$ are defined in the following way:*

$$\begin{array}{rcl}
\mathcal{N}(\boldsymbol{let}\ x = e) = \mathcal{N}(\boldsymbol{sig}\ x\ \boldsymbol{in}\ \tau) & = & \{x\} \\
\mathcal{N}(\boldsymbol{let}\ \boldsymbol{rec}\ \{x_1 = e_1; \ldots ; x_n = e_n\}) & = & \{x_i\}_{i=1\ldots n} \\
\mathcal{N}(\boldsymbol{rep}) = \mathcal{N}(\boldsymbol{rep}\ =\ \tau) & = & rep
\end{array}$$

$$\begin{array}{rcl}
\mathcal{D}(def) & = & \emptyset \quad if\ def = \left\{ \begin{array}{c} \boldsymbol{sig}\ s\ \boldsymbol{in}\ \tau \\ rep \end{array} \right. \\
& = & \mathcal{N}(def) \quad else
\end{array}$$

$$\mathcal{N}(\boldsymbol{theorem}\ x = ...) = \{x\} \qquad \mathcal{N}(\boldsymbol{property}\ x = ...) = \{x\}$$

$$\mathcal{D}(\boldsymbol{theorem}\ x = ...) = \{x\} \qquad \mathcal{D}(\boldsymbol{property}\ x = ...) = \emptyset$$

*Consider the following statement:*
***species*** $s$ ***inherits*** $s_1, \ldots s_n = \phi_1 \ldots \phi_m$ *such that*

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

*Then, we define $\mathcal{N}(s)$ and $\mathcal{D}(s)$:*

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

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

## B. Most General Unifier

DEFINITION 19 (Most General Unifier).
*Let $\sigma$ be either a concrete type $\upsilon$ or $\perp$,*

*[Eq]*
$mgu_\sigma(\tau, \tau) = \tau, id$

*[Var1]*
$mgu_\sigma(\alpha, \tau) = \tau, [\alpha \leftarrow \tau]$ $\alpha$ not free in $\tau$

*[Var2]*
$mgu_\sigma(\tau, \alpha) = \tau, [\alpha \leftarrow \tau]$ $\alpha$ not free in $\tau$

*[Arrow]*

$$\frac{mgu_\sigma(\tau_1, \tau_1') = \tau_1'', \theta \qquad mgu_\sigma(\tau_2\theta, \tau_2'\theta) = \tau_2'', \phi}{mgu_\sigma(\tau_1 \to \tau_2, \tau_1' \to \tau_2') = \tau_1'' \to \tau_2'', \phi\theta}$$

*[Prod]*

$$\frac{mgu_\sigma(\tau_1, \tau_1') = \tau_1'', \theta \qquad mgu_\sigma(\tau_2\theta, \tau_2'\theta) = \tau_2'', \phi}{mgu_\sigma(\tau_1 * \tau_2, \tau_1' \to \tau_2') = \tau_1'' \to \tau_2'', \phi\theta}$$

*[Self1]* $mgu_\upsilon(\textbf{\textit{self}}, \upsilon) = \textbf{\textit{self}}, id$

*[Self2]* $mgu_\upsilon(\upsilon, \textbf{\textit{self}}) = \textbf{\textit{self}}, id$

*In every other case,* $mgu_\sigma$ *fails.*

*Given* $\Delta, \Pi, \Gamma, \Sigma$ *a typing environment, and* $\tau_1, \tau_2$ *two types such that* $mgu_{\Sigma(rep)}(\tau_1, \tau_2) = \tau_3, \theta$ *we define* $Mgu(\tau 1, \tau 2) \widehat{=} \tau_3\theta$.

## C. Dependencies

DEFINITION 20. $\wr \cdot \int$ *is defined by induction on the expressions of the language:*

$$\begin{array}{rcl}
\wr x \int & = & \emptyset \\
\wr \textbf{\textit{let}}\ x = \ e_1\ \textbf{\textit{in}}\ e_2 \int & = & \wr e_1 \int \oplus \wr e_2 \int \\
\wr \textbf{\textit{let rec}}\ x = \ e_1\ \textbf{\textit{in}}\ e_2 \int & = & \wr e_1 \int \oplus \wr e_2 \int \\
\wr \lambda x.e \int & = & \wr e \int \\
\wr c.x \int & = & \begin{cases} \emptyset & \text{when } c \neq \textbf{\textit{self}} \\ \{x\} & \text{when } c = \textbf{\textit{self}} \end{cases} \\
\wr e_0(e_1, \ldots, e_n) \int & = & \wr e_0 \int \cup (\bigcup_{i=1}^n \wr e_i \int)
\end{array}$$

DEFINITION 21 (dependencies of prop). *If a proposition p consists of an expression then its dependencies are already defined in def.20. Otherwise, we define them by induction on the structure of p:*

$$\wr p_1\ \textbf{\textit{and}}\ p_2 \int = \wr p_1 \int \oplus \wr p_2 \int \qquad \wr p_1\ \textbf{\textit{or}}\ p_2 \int = \wr p_1 \int \oplus \wr p_2 \int$$

$$\wr p_1 \ \to \ p_2 \int = \wr p_1 \int \oplus \wr p_2 \int \qquad \wr \textbf{\textit{not}}\ p \int = \wr p \int \qquad \wr \textbf{\textit{all}}\ x\ \textbf{\textit{in}}\ \tau, p \int = \wr p \int$$

$$\wr \textbf{\textit{ex}}\ x\ \textbf{\textit{in}}\ \tau, p \int = \wr p \int$$

DEFINITION 22.

$$\begin{array}{rcl}
\wr \textbf{\textit{let}}\ x = e \int & = & \wr e \int \\
\wr \textbf{\textit{let rec}}\ \{x_1 = e_1\ \ldots x_n = e_n\} \int & = & \cup_{i=1}^n \wr e_i \int \setminus \{x_i\}_{i=1\ldots n} \\
\wr \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau \int & = & \emptyset \\
\wr \textbf{\textit{property}}\ x = p \int & = & \wr p \int \\
\wr \textbf{\textit{theorem}}\ x = p\ \textbf{\textit{proof:}} \ldots \textbf{\textit{decl}}\ x_1, \ldots, x_n; \ldots \int & = & \wr p \int \cup \{x_i\}_{i=1..n}
\end{array}$$

DEFINITION 23 (def-dependencies).

$$\lbag \textbf{\textit{proof:}} \ldots \textbf{\textit{def:}} x_1 \ldots x_n \ldots \rbag = \{x_i\}_{i=1\ldots n}$$

*Let $\phi$ be a method definition*

- *if $\phi$ is* ***theorem*** *$x = \ldots proof$ then $\lbag \phi \rbag = \lbag proof \rbag$*
- *otherwise $\lbag \phi \rbag = \emptyset$*

## D. Merging two fields

DEFINITION 24.

*Let $\phi_1$ and $\phi_2$ be two fields, with $\mathcal{N}(\phi_1) \cap \mathcal{N}(\phi_2) \neq \emptyset$.*

*We define $\phi_1 \ogreaterthan \phi_2$ the following way. Note that this is a partial function, since calls to $Mgu$ may fail.*

$$\begin{array}{llcl}
\textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau & \ogreaterthan\ \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau & = & \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau \\
\textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau_1 & \ogreaterthan\ \textbf{\textit{let}}\ x\ in\ \tau_2 =\ e_2 & = & \textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ Mgu(\tau_1,\tau_2) =\ e_2 \\
\textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau_1 & \ogreaterthan\ \textbf{\textit{let rec}} & = & \textbf{\textit{let rec}} \\
 & \quad (x\ in\ \tau_2 = e_2) \cup C & & (x\ in\ Mgu(\tau_1,\tau_2) = e_2) \cup C \\
\textbf{\textit{let rec}} & \ogreaterthan\ \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau_2 & = & \textbf{\textit{let rec}} \\
(x\ in\ \tau_1 = e_1) \cup C & & & (x\ in\ Mgu(\tau_1,\tau_2) = e_1) \cup C \\
\textbf{\textit{let}}\ x\ in\ \tau_1 = e_1 & \ogreaterthan\ \textbf{\textit{let}}\ x\ in\ \tau_2 = e_2 & = & \textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ Mgu(\tau_1,\tau_2) = e_2 \\
\textbf{\textit{let}}\ x\ in\ \tau_1 =\ e_1 & \ogreaterthan\ \textbf{\textit{sig}}\ x\ \textbf{\textit{in}}\ \tau_2 & = & \textbf{\textit{let}}\ x\ \textbf{\textit{in}}\ Mgu(\tau_1,\tau_2)\ =\ e_1 \\
\textbf{\textit{let rec}} & \ogreaterthan\ \textbf{\textit{let rec}} & = & \textbf{\textit{let rec}} \\
\quad \{x_i\ [in\ \tau_i] =\ ex_i & \quad \{x_i\ [in\ \sigma_i] =\ ex'_i & & (\{x_i\ in\ Mgu(\tau_i,\sigma_i) = ex'_i\} \\
\quad y_i\ [in\ \rho_i] = ey_i\} & \quad z_i\ [in\ \pi_i] = ez_i\} & & \cup\{y_i\ in\ \rho_i =\ ey_i\} \\
 & & & \cup\{z_i\ in\ \pi_i =\ ez_i\})
\end{array}$$

DEFINITION 25 (merging properties and theorems).

$$\begin{array}{rclcl}
\textbf{\textit{property}}\ x\ = p & \ogreaterthan & \textbf{\textit{property}}\ x\ = p & = & \textbf{\textit{property}}\ x\ = p \\
\textbf{\textit{property}}\ x\ = p & \ogreaterthan & \textbf{\textit{theorem}}\ x = p & = & \textbf{\textit{theorem}}\ x = p \\
 & & \textbf{\textit{proof:}} prf & & \textbf{\textit{proof:}} prf \\
\textbf{\textit{theorem}}\ x = p & \ogreaterthan & \textbf{\textit{property}}\ x\ = p & = & \textbf{\textit{theorem}}\ x = p \\
\textbf{\textit{proof:}} prf & & & & \textbf{\textit{proof:}} prf \\
\textbf{\textit{theorem}}\ x = p & \ogreaterthan & \textbf{\textit{theorem}}\ x = p & = & \textbf{\textit{theorem}}\ x = p \\
\textbf{\textit{proof:}} prf_1 & & \textbf{\textit{proof:}} prf_2 & & \textbf{\textit{proof:}} prf_2
\end{array}$$

*Other cases are irrelevant: the two field definitions must share the same name and the same statement (*i.e. *$\mathcal{T}_{def}(x)$)*

## E. The Issues of Polymorphic Species

```
species polymorph = rep; let id = fun x -> x; end

species prm (x is polymorph) =
  rep = unit;
  let elt = x!id(true);
end

species imp inherits polymorph =
  rep = unit; let id = fun x -> x + 1;
end

collection Imp implements imp

collection error implements prm(Imp)
```

Here, `polymorph` has a polymorphic method `id`, with type `'a->'a`. Then `prm` takes any implementation `x` of `polymorph` as parameter, and uses `x!id` with type `bool -> bool`. `imp` is a sub-species of `polymorph` which redefines `id` with type `int->int`. Both `bool -> bool` and `int->int` are valid instances of `'a->'a`, so that the definition appears to be correct. However, the collection `error` is not well-typed: Its method `elt` would evaluate in `true + 1`.