Prototype d'interface utilisateur de la librairie FoC

Virgile Prevosto[1]

1: Laboratoire d'Informatique de Paris 6
8 rue du Cne Scott
75015 Paris
`virgile.prevosto@lip6.fr`

Résumé

Ce papier présente le noyau d'une interface permettant de construire des bibliothèques de programmes de calcul formel avec leurs certifications. Ces définitions sont traduites vers OCAML et COQ. Après avoir rapidement présenté la syntaxe concrète utilisée, nous étudions des moyens de représenter en OCAML et surtout en COQ les différentes structures utilisées.

1. Introduction

Le projet FoC (Formel, Ocaml, Coq) a pour objectif de fournir un environnement de calcul formel certifié. Depuis la création du projet, un important travail de réflexion a permis de dégager un certain nombre de notions, pour pouvoir parler avec précision des représentations possibles des différentes structures mathématiques (2.2). La réflexion s'appuie aussi sur l'utilisation du langage AXIOM [5], logiciel de calcul formel reposant sur une hiérarchie de structures algébriques. Toutefois, là où AXIOM "fait confiance" à l'utilisateur qui définit une nouvelle implantation, FoC a pour but d'amener cet utilisateur à fournir en même temps une preuve que cette implantation possède les propriétés requises (par exemple que la multiplication d'un groupe est bien associative, qu'elle admet un élément neutre,...). Pour ce faire, on utilise deux langages : le premier dédié à la vérification des preuves – en l'occurence COQ– et le second destiné à faire des calculs – OCAML. Après avoir exploré différentes voies (cf.[4]), les membres du projet se sont fondés sur les classes et les modules de OCAML pour représenter les notions manipulées en calcul formel. A ce jour une librairie assez importante existe en OCAML (cf.[3]). Du côté COQ, la formalisation de la théorie permettant de passer d'une structure algébrique à une autre (comme on peut passer d'un groupe à un anneau) a été réalisée par Sylvain Boulmé dans son travail de thèse.

A mesure que le projet a grandi, la nécessité de développer une interface utilisateur s'est fait jour. Plusieurs raisons l'expliquent. Tout d'abord, on ne peut exiger des utilisateurs une connaissance parfaite des langages utilisés pour le développement de la librairie. Ensuite, il est important, pour des gros développements, de ne pas avoir à écrire des définitions une fois en OCAML et une fois en COQ. Il faut de plus veiller à la cohérence entre le code OCAML et les preuves COQ. Pour cela, nous avons défini un noyau de syntaxe concrète (3), qui est traduit à la fois vers OCAML (5) et vers COQ (6). De plus, nous proposons une vision de la hiérarchie des structures (4), inspirée de celle que S. Boulmé décrit dans [2], qui conduit à une analyse préalablement à la traduction.

2. Vocabulaire du projet

Tout d'abord, nous allons introduire informellement un certain nombre de concepts fréquemment utilisés au sein du projet.

2.1. Représentation concrète vs. abstraction mathématique

La construction de cet environnement certifié repose sur une représentation "concrète" (*i.e* représentable ultimement sous forme d'une suite de bits) de différents concepts mathématiques : en premier lieu, on a affaire à des *individus*, abstraits, comme le polynôme $X + 2$, représentés, pour l'ordinateur, par des **entités**, par exemple `[(1,1);(2,0)]`. Il n'y a pas bijection entre les deux. Un individu peut être représenté de plusieurs manières différentes (polynômes pleins ou creux), et inversement une entité peut désigner, suivant le contexte, différents individus. (`0` peut désigner le zéro des entiers modulo 2, ou modulo 3, ou ...) Pour bien séparer ces deux notions, il est possible de s'appuyer sur un mécanisme d'abstraction de types (cf.[4]), qui offre en outre l'avantage de masquer à l'utilisateur final l'implantation réelle des algorithmes, séparant aussi utilisation et développement.

2.2. Structures mathématiques

Espèces Les individus mathématiques peuvent appartenir à différentes *structures algébriques* (anneaux, groupes,...). De la même manière, qu'un *individu* est représenté par une **entité**, une *structure algébrique* est représentée par une **espèce**. Pour cela, on doit décrire les *opérations* et les *propriétés*, propres à cette structure. On regroupe sous le nom de **méthodes**. On distingue des **méthodes primitives** et des **méthodes dérivées**. Les méthodes primitives sont celles qu'il faut fournir pour caractériser la structure mathématique considérée (par exemple l'opération "+" et la propriété qu'elle est commutative pour un groupe abélien). Les méthodes dérivées sont celles qu'on peut construire de manière générique pour la structure considérée (ainsi, on peut définir la soustraction en s'appuyant sur la donnée des méthodes primitives que sont l'addition et le passage à l'opposé dans un groupe). De plus, chaque espèce contient le type des **entités** qu'on y manipule, appelé le **type-support**. Enfin, une espèce peut prendre en paramètre des collections (définies ci-après) et des entités (ainsi les matrices sont paramétrées par l'anneau de leurs coefficients, défini par une collection, et deux autres paramètres, entités donnant leurs dimensions).

Interfaces A chaque espèce correspond une **interface** : il s'agit de son "type", dans lequel on efface toutes les définitions des méthodes dérivées, pour ne conserver que leur type. De même, le type-support est alors abstrait. L'interface correspond à ce que voit l'utilisateur : un ensemble de fonctions ayant des propriétés particulières, mais dont l'implantation exacte n'est pas forcément connue, pas plus que la structure interne des objets manipulés.

Collections Enfin, les **collections** sont des espèces complètement instanciées, qui de plus représentent des constructions complètement achevées. Pour reprendre une terminologie objet, on ne peut plus les étendre. Elles représentent des instances particulières des structures générales qu'on décrit à travers les espèces. Ainsi, on peut définir une collection représentant l'anneau $\mathbb{Z}$ des entiers relatifs, à partir de l'espèce représentant les anneaux, en se donnant le type-support et les méthodes adaptées.

3. Syntaxe concrète

3.1. Espèces et collections

Si on s'en tient à la présentation de la section 2.2, la construction d'une espèce passe par la données de méthode primitives et dérivées, d'un type support, et de paramètres. Comme on l'a dit, ces derniers se présentent de deux manières différentes. On distingue donc les paramètres de collection (`prm1 is interface1`) et ceux d'entité (`prm2 in type2`). D'autre part, comme les structures qu'elles représentent, les espèces sont définies les unes à partir des autres, selon un mécanisme qui s'apparente à l'héritage des langages objet.

Un exemple simple permettra de présenter rapidement les différentes étapes de la définition d'une nouvelle espèce dans la syntaxe concrète. On suppose qu'on a défini précédemment une espèce `semi_group`, qui possède deux opérations primitives, `eqn` et `mult`, et les propriétés associées (`eqn` est une relation d'équivalence et `mult` est associative), et une espèce `unitary_set` qui contient un élément distingué, `one`, et l'opération `eqn`. On peut définir l'espèce des monoïdes, dans laquelle `one` est un élément neutre pour `mult`, de la manière suivante :

```
species monoid inherits semi_group,unitary_set =
On ajoute les deux propriétés primitives. Dans l'énoncé de neutral_left, self sert en plusieurs endroits. Dans la
première occurence, il dénote le type-support de l'espèce, alors que dans la suite, il représente l'espèce toute entière, et
signale un appel à une méthode de monoid
  property neutral_left = all x in self (self.eqn(x,self.mult(self.one,x)));
  property neutral_right = all x in self (self.eqn(x,self.mult(x,self.one)));
Deux théorèmes qu'on peut démontrer à partir des propriétés précédentes
  theorem unique_one_l = all x in self ((all y in self self.eqn(self.mult(x,y),y))->self.eqn(x,self.one))
  demo :
la démonstration est effectuée en Coq.
   Intros.
   Apply (self_eqn_trans0 x (self_mult0 x self_one0) self_one0);
   Auto.
  Qed.;
      ;;
```

En plus des propriétés primitives requises par la définition mathématique de la structure, introduites par **property**, on trouve des théorèmes (propriétés dérivées), définis par **theorem** valables pour tous les monoïdes. Deux autres mots-clés permettent d'introduire des définitions de méthodes calculatoires. Il s'agit de **sig** pour les fonctions primitives, et de **let** pour les fonctions dérivées (cf ci-dessous).

Cet exemple des monoïdes est défini grâce à un héritage multiple dans lequel les deux parents partagent une méthode (`eqn`). Pour réaliser l'héritage, il est nécéssaire (cf. p.67) que les types des deux méthodes soient identiques. Le cas auquel on est confronté ici est simple, car les deux méthodes des espèces parent sont primitives : la méthode correspondante dans l'espèce fille est également primitive. Dans le cas où on trouve une méthode primitive et une méthode dérivée, il suffit de garder la seule définition présente. Reste la situation où on a deux définitions différentes. Les travaux sur l'héritage multiple montrent qu'il n'y a pas de méthode optimale simple pour sélectionner une définition plutôt qu'une autre. Dès lors, on est amené à faire un choix a priori. Dans le cas de FoC, nous avons suivi la sémantique de OCAML sur les objets, qui consiste à prendre systématiquement la définition de la dernière espèce parent, dans l'ordre de la ligne de commande.

Pour prolonger notre exemple, on peut implanter `monoid` par une collection sur les entiers (`int` de OCAML et `Z` de COQ) :

```
collection int_mnd implements monoid =
  rep T = caml_type int with coqrep Z;
  let eqn (x, y) = caml eqn with coq eqn;
  demo eqn_refl = ...
  ...
;;
```

rep permet de définir le type-support (ici, il est importé depuis OCAML et COQ, tout comme la fonction `eqn`). Le mot-clé **demo** permet lui de donner directement la démonstration de `eqn_refl`, qui est une propriété primitive de `monoid` sans avoir auparavant à réécrire explicitement son énoncé. Une collection n'implante qu'une seule espèce, et on ne peut évidemment y définir de méthodes primitives. Par contre, on peut introduire des méthodes qui n'étaient pas présentes dans l'espèce implantée. Ces méthodes seront alors invisibles en-dehors de la collection elle-même : il s'agit de fonctions utilitaires locales. Pour plus de clarté, on peut les introduire par **local**, mais ce mot clé est un synonyme de **let**[1]. Pour plus de détails, on pourra se reporter aux sources exemples

[1]les lemmes locaux sont à définir à l'intérieur des scripts COQ. La situation sera reconsidérée lors de l'élaboration du langage de preuves

fournis dans les sections 5 et 6, qui mettent en parallèle des définitions de la syntaxe concrète et leur pendant en OCAML et COQ

3.2. Types

Il est possible de définir des types concrets en FoC. Ces derniers peuvent préexister en OCAML et COQ. Par exemple, les booléens sont construits ainsi :

```
type foc_bool =
  caml_link bool;
  coq_link bool;
  true in foc_bool;
  false in foc_bool;
;;
```

On fait le lien avec les types `bool` préexistant dans les deux langages. On peut donner aussi un exemple de type paramétré, les listes :

```
type foc_list(a) =
 Nil in foc_list(a);
 Cons in a -> foc_list(a) -> foc_list(a)
;;
```

Ces types concrets forment, avec les types-support des collections, les atomes du langage de type de FoC. A partir de ces types atomiques, on peut construire les types des fonctions (`typ1->typ2`) ou des paires d'éléments (`(typ1*typ2)`).

3.3. Expressions et propriétés

Le langage utilisé dans FoC est un langage à deux sortes : expressions et propositions logiques. Les expressions ont une syntaxe proche de CAML-LIGHT :

expr::= **if** `expr` **then** `expr` **else** `expr` | **match** `expr` **with** *(*`cons` -> `expr`*)** | `coll.name` | `expr`*(*`expr1,expr2,...`*)* | **let** `name` **in** `type1` = `expr1` **in** `expr2`

Les propositions logiques sont, elles, construites de la manière suivante :

prop::= **all** `x` **in** `xtype prop` | **ex** `x` **in** `xtype prop` | `prop1` **and** `prop2` | `prop1` **or** `prop2` | `prop1` -> `prop2` | **not** `prop` | `prop_atomique`

Une proposition atomique est construite à partir d'un prédicat ou d'une expression de type `bool` et d'une conversion, qui reste implicite dans la syntaxe concrète, `Is_true`.

4. La hiérarchie des espèces

La définition des espèces utilise un certain nombre de traits de la programmation par objets (en particulier l'héritage). Cela a conduit à utiliser les classes de OCAML pour les représenter. Un problème se pose avec COQ qui lui n'offre pas d'équivalent à ce système de classes. Deux solutions sont possibles. On peut construire un mécanisme général dans COQ lui-même qui émulerait les traits objets dont on a besoin. Cette approche a été réalisée dans le cadre du projet par Sylvain Boulmé [2] dans son travail de thèse. Pour cela, il définit une théorie des enregistrements avec champs dépendants, dans laquelle les notions même d'espèce, d'interface et de collection sont définies par des termes COQ précis. De même il a défini en COQ la plupart des concepts évoqués par la suite (ensemble de noms de champs, opérateurs de raffinement pour passer d'une espèce à l'autre). Tout le mécanisme d'héritage peut ainsi être simulé. On constate toutefois une explosion du temps d'accès aux champs, dûe aux nombreuses vérifications que fait COQ. Une autre approche possible consiste à déléguer une partie des tâches au traducteur, de manière à réduire la taille des structures utilisées ensuite. Dans cette section, nous allons présenter un certain nombre d'opérations primitives sur les espèces, dont la composition permet de retrouver les mécanismes dont nous avons besoin. C'est sur ce formalisme, dont une analyse plus complète est faite dans [15], qu'est fondée l'analyse des définitions d'espèces dans le traducteur.

4.1. Ensembles de champs

Soient $\mathcal{N}$ un ensemble infini de noms, $\mathcal{L}$ un langage, $\mathcal{T}$ l'ensemble des types que peuvent prendre les expressions de $\mathcal{L}$, et $\perp$ un symbole spécial, n'appartenant pas à $\mathcal{L}$ et pouvant prendre n'importe quel type. On notera classiquement $e : t$ la relation "l'expression e a le type t". On suppose de plus l'existence d'une fonction $abst : \mathcal{T} \rightarrow \mathcal{T}$. Cette fonction modélise une abstraction de type réalisée lors du passage d'une espèce à une collection. En premier lieu, on peut voir les espèces comme des ensembles de champs.

Définition 1 *Une espèce est un ensemble fini de triplets de la forme :*

$$S = \{(n_i, t_i, d_i)\}_{i \in I}$$

vérifiant les propriétés suivantes :
- *I un ensemble fini, éventuellement vide.*
- $\forall i \in I \ (n_i, t_i, d_i) \in (\mathcal{N} * \mathcal{T} * (\mathcal{L} \cup \{\perp\}))$
- *la projection $p_1 : S \rightarrow \mathcal{N}$ suivant la première composante est injective.*
- $\forall i \in I,\ d_i : t_i$

Pour prolonger l'exemple de la section précédente, l'espèce des monoïdes sera représentée (cf fig.1) par un ensemble contenant tous ses champs, ceux présents explicitement dans la syntaxe concrète comme (`neutral_left`, ... , $\perp$) comme ceux qui ont été hérité, tel (`eqn`, `self` $\rightarrow$ `self` $\rightarrow$ `bool`, $\perp$).

Interfaces et collections ont sensiblement la même définition que les espèces, mis à part les conditions sur d_i.

Définition 2 *Une interface est un ensemble fini de couples $\{(n_i, t_i)\}_{i \in I}$ tels que $\forall i \in I \ (n_i, t_i) \in (\mathcal{N} * \mathcal{T})$*

La première projection est injective.

Définition 3 *Une collection est un ensemble fini de triplets $\{(n_i, t_i, d_i)\}_{i \in I}$, avec $\forall i \in I (n_i, t_i, d_i) \in (\mathcal{N} * \mathcal{T} * \mathcal{L})$ (i.e. $d_i \neq \perp$)*

La première projection est injective, et $\forall i \in I \ (d_i : t_i)$.

Pour simplifier, on note $\mathfrak{S}$ l'ensemble des espèces, $\mathfrak{I}$ celui des interfaces et $\mathfrak{C}$ celui des collections. Par ailleurs, on se donne deux fonctions, permettant de distinguer les différentes familles d'objets utilisées.

Définition 4

$$freeze : \mathfrak{S} \rightarrow \mathfrak{C}$$

freeze est défini en E si et seulement si $\forall (n, t, d) \in E,\ d \neq \perp$

$$freeze\left(\{(n_i, t_i, d_i)\}_{i \in I}\right) = \{(n_i, abst(t_i), d_i)\}_{i \in I}$$

$spec\_type : \mathfrak{S} \rightarrow \mathfrak{I}$

$$spec\_type\left(\{(n_i, t_i, d_i)\}_{i \in I}\right) = \{(n_i, t_i)\}_{i \in I}$$

Le type des propriétés est leur énoncé et leur définition est leur preuve éventuelle (via l'isomorphisme de Curry-Howard). Les éventuels paramètres, de même que le type support sont également représentés par des triplets. Pour vérifier qu'on instancie les paramètres par des collections ayant la bonne interface, on définit deux relations, `is_prm` $\subset \mathfrak{I} \times \mathfrak{S}$ et `inst_prm` $\subset \mathfrak{C} \times \mathfrak{S}$. `is_prm`$(I, E)$ exprime le fait qu'une espèce E attend un paramètre ayant comme interface I, et `inst_prm`(C, E) signale qu'on instancie un des paramètres de E par la collection C. Ces relations permettent d'exprimer facilement les conditions qu'on doit vérifier (voir p.66) lors de l'instantiation des paramètres de collection. En effet, la collection qu'on passe en paramètre n'a pas forcément exactement l'interface attendue, mais une interface dérivée de celle-ci par héritage. Par exemple, les matrices ont comme paramètre l'anneau de leurs coefficients. On peut très bien l'instancier par un corps (qui est un anneau particulier), mais pas par un groupe, qui ne possède pas tous les champs requis.

$$\text{monoïde}\left\{\begin{array}{c}(S, Set, \bot)\\(eq, S \to S \to bool, \bot)\\(eqn\_refl, \forall x \in S\ S.eqn(x,x), \bot)\\(one, S, \bot)\\(is\_one, S \to bool, fun\ x \to S.eqn(S.one, x))\\(power, S \to int \to S, ...)\\...\end{array}\right\} \Rightarrow \begin{array}{c}\text{opérations réalisées}\\\boxed{\begin{array}{c}\text{déclaration de } \texttt{inv}\\\text{définition directe de } \texttt{div}\\\text{effacement de } \texttt{is\_one}\\\text{instantiation de } \texttt{eq}\\\text{redéfinition de } \texttt{power}\\...\end{array}}\end{array}$$

$$\Longrightarrow \quad \text{groupe}\left\{\begin{array}{c}(S, Set, \bot)\\(inv, S \to S, \bot)\\(div, S \to S \to S, fun\ x \to fun\ y \to S.mult(x, S.inv(y)))\\(is\_one, S \to bool, \bot)\\(eq, S \to S \to bool, S.is\_one(S.div(x,y))\\(power, S \to int \to S, ...)\\...\end{array}\right\}$$

FIG. 1 – exemple d'héritage entre deux espèces

4.2. Passage d'une espèce à l'autre

Les opérations d'héritage ne concernent que les espèces. Certaines de ces opérations agissent sur les interfaces associées, mais leur comportement se déduit alors de celui qu'elles ont sur les espèces, par une application de *spec_type*. De plus, par définition, les collections ne peuvent être étendues par héritage. Collections et interfaces n'interviennent, en fait, qu'en tant que paramètres pour les espèces.

Dans notre représentation, chaque champ peut apparaître sous trois états différents : absent, déclaré ou défini. Définir une espèce à partir des espèces déjà connues amène à modifier l'état de certains de ces champs. Plus précisément, on peut déclarer ou définir un champ absent, définir un champ déclaré, et modifier ou effacer la définition d'un champ défini. Les changements d'états impossibles à réaliser aboutiraient à modifier le type du champ concerné, ou même à le faire disparaître. En effet, toutes les opérations applicables (donc toutes les étiquettes) dans l'espèce parent existent *a fortiori* dans l'espèce fille, qui en est, d'un certain point de vue, une spécialisation. Etudions par exemple (fig.1) le passage d'un monoïde à un groupe (cet exemple est simplement destiné à présenter l'ensemble des changements d'états susceptibles d'intervenir). Un groupe est un monoïde dans lequel chaque élément possède un inverse. On est donc amené à considérer un nouveau champ déclaré, *inv*. De plus, on est alors à même de définir une division ($div(x, y) = x{*}inv(y)$), champ qui n'existait pas dans un monoïde. Par ailleurs, on définit l'égalité, qui jusqu'alors n'était que déclarée (eq) à partir du test à un ($eq(x, y) = is\_one(div(x, y))$). Pour cela, il faut oublier la définition de *is_one* donnée dans les monoïdes, car elle fait elle-même appel à *eq*. Bien sûr, il faudra par la suite donner une nouvelle définition pour *is_one*. Enfin, on redéfinit `power`, qui peut maintenant accepter des exposants négatifs, ce qui était impossible dans un monoïde.

Dans l'exemple que nous venons de décrire, toutes les opérations ont lieu "en même temps", au moment où l'utilisateur décrit la nouvelle espèce. Nous allons maintenant introduire un jeu d'opérations primitives, ne portant que sur une seule méthode et une seule espèce. La création d'une nouvelle espèce se fait en composant ces opérations primitives.

Définition 5 (les opérations primitives) *Dans chacune des définitions ci-dessous, I représente un ensemble fini (éventuellement vide)*

- *Soient* $E = \{(n_i, t_i, d_i)\}_{i \in I} \in \mathfrak{S}$ *et* $n \in \mathcal{N}$ *tel que* $\forall i \in I\ n_i \neq n$*, on a*

$$\mathbf{decl}(n, t, E) = \{(n_i, t_i, d_i)\}_{i \in I} \cup \{(n, t, \bot)\}$$

FIG. 2 – exemple de chemin

$$\{monoide\} \xrightarrow{\texttt{decl}(inv, S \rightarrow S)} E_1 \xrightarrow{\texttt{decl}(div, \ldots)} E_2 \xrightarrow{\texttt{def}(div, \ldots)} E_3 \xrightarrow{\ldots} \{groupe\}$$

- *Soient* $E = \{(n_i, t_i, d_i)\}_{i \in I} \cup \{(n, t, \perp)\} \in \mathfrak{S}$, $n \in \mathcal{N}$ *et* $d \in \mathcal{L}$ *de type* t*, on a*
$$\texttt{def}(n, d, E) = \{(n_i, t_i, d_i)\}_{i \in I} \cup \{(n, t, d)\}$$
- *Soit* $E = \{(n_i, t_i, d_i)\}_{i \in I} \cup \{(n, t, d)\} \in \mathfrak{S}$*, où* $d \neq \perp$*, on a*
$$\texttt{undef}(n, E) = \{(n_i, t_i, d_i)\}_{i \in I} \cup \{(n, t, \perp)\}$$

Cela aboutit à représenter le passage d'une espèce, E, à une autre, F, par un chemin.

Définition 6 *Un chemin de E à F (noté* $\texttt{path}(E, F)$*) est un ensemble fini d'espèces* $E_0, E_1, \ldots, E_n$ *vérifiant les conditions suivantes.*
- $E_0 = E$ *et* $E_n = F$.
- $\forall i < n$, E_{i+1} *est l'image de* E_i *par une des opérations primitives.*

Ainsi, on peut décrire le début d'un chemin possible pour aller des monoïdes jusqu'au groupes : on commence par déclarer *inv*, puis on déclare *div*, qu'on définit juste après. Ensuite, on effacera la définition de *power*, avant d'en redonner une nouvelle, ...

4.3. Equivalence entre chemins

L'utilisateur ne définit que les extrémités de ces chemins. Il convient donc de s'assurer que les espèces "intermédiaires" introduites par les opérations primitives restent proches de celles qui sont présentes dans le programme. Pour cela, on se donne une relation d'équivalence "hors" d'un sous-ensemble de $\mathcal{N}$. Pour avoir un modèle correct, on vérifie que les espèces intermédiaires définies lors de la modélisation d'un héritage entre E_1 et E_2 (les deux espèces définies par l'utilisateur) sont toutes équivalentes hors des champs qui diffèrent entre E_1 et E_2.

Définition 7 *Soit* $A \subset \mathcal{N}$ *fini. Deux espèces* E_1 *et* E_2 *sont* équivalentes hors de A *(avec A fini) si et seulement si*

$$\forall n \in \mathcal{N} \backslash A, \forall t \in \mathcal{T}\, \forall d \in \mathcal{L},\ (n, t, d) \in E_1 \iff (n, t, d) \in E_2$$

On utilisera la notation suivante : $E_1 \cong_A E_2$

Propriété 1 $\forall E_1, E_2 \in \mathfrak{S}$*, il existe un plus petit sous ensemble de* $\mathcal{N}$*, noté* $E_1 \triangle E_2$ *tel que* $E_1 \cong_{E_1 \triangle E_2} E_2$ $E_1 \triangle E_2$ *est appelé divergence de* E_1 *et* E_2.

Démonstration On note
$$H = \{A \subset \mathcal{N}, E_1 \cong_A E_2\}$$
Il faut montrer que H possède un plus petit élément pour l'inclusion :
- $H \neq \emptyset$: il contient au moins la réunion M des champs de E_1 et de ceux de E_2.
- H est stable par intersection :
  $\forall A, B \in H$ on a, en rassemblant les deux définitions
  $$\forall n \in (\mathcal{N} \backslash A) \cup (\mathcal{N} \backslash B),\ (n, t, d) \in E_1 \iff (n, t, d) \in E_2$$
  et avec l'égalité $(\mathcal{N} \backslash A) \cup (\mathcal{N} \backslash B) = \mathcal{N} \backslash (A \cap B)$, on obtient finalement $E_1 \cong_{A \cap B} E_2$

Il suffit alors de prendre l'intersection de tous les éléments de H. Avec ce qui précède, cette intersection existe (H est non vide) et elle est dans H. (H est infini, mais comme on considère des ensembles discrets, on peut se ramener à calculer l'intersection d'un nombre fini d'ensembles, par exemple ceux de $\{A \cap M, A \in H\}$).

Cela permet d'introduire les équivalences sur les chemins. Intuitivement, deux chemins sont équivalents si ils ont les mêmes extrémités, et si les espèces intermédiaires sont "relativement proches". Plus précisément, il faut que tout triplet présent dans une espèce d'un des chemins existe aussi dans une espèce de l'autre.

Définition 8 (équivalence entre chemins) *Soient deux chemins,* $S = \{O_i\}_{0\leq i\leq n}$ *et* $S' = \{O'_j\}_{0\leq j\leq n'}$. S *et* S' *sont dits* équivalents *si et seulement si*
- $O_0 = O'_0$
- $O_n = O'_{n'}$
- $\forall i \leq n \bigcap_{0\leq j\leq n'} O_i \Delta O'_j = \emptyset$
- $\forall j \leq n\prime \bigcap_{0\leq i\leq n} O_i \Delta O'_j = \emptyset$

On notera $S \simeq S'$ *l'équivalence entre* S *et* S'

On étend enfin cette notion d'équivalence pour des graphes. Un graphe est un ensemble d'espèces de collections et d'interfaces, reliés par les différentes opérations et relations que nous avons vues (freeze, spec_type, `is_prm`, `inst_prm`, `decl`, `def` et `undef`)

Définition 9 (graphe) *Un graphe* G *est constitué d'un ensemble* $\mathcal{E}(G)$ *d'espèces,* $\mathcal{C}(G)$ *de collections et* $\mathcal{I}(G)$ *d'interfaces, Les espèces peuvent être reliées entre elles par des chemins. On distingue de plus un ensemble* $\mathcal{E}^*(G) \subset \mathcal{E}(G)$ *d'espèces, tel que :*
- $\forall E \in \mathcal{E}(G)((\exists C \in \mathcal{C}(G)\, C = freeze(E)) \Rightarrow E \in \mathcal{E}^*(G))$
- $\forall I \in \mathcal{I}(G)((\exists E \in \mathcal{E}(G)\, \textit{\textbf{is\_prm}}(I,E)) \Rightarrow (\exists F \in \mathcal{E}^*(G)\, I = spec\_type(F)))$

On notera $E_1 \rightarrow^*_G E_2$ *pour signaler qu'il existe un chemin entre* E_1 *et* E_2 *dans le graphe* G. G *doit de plus vérifier les propriétés suivantes, qui assurent la correction des instantiations des paramètres :*

$$\begin{aligned}&\forall C, E \in \mathcal{C}(G) \times \mathcal{E}(G), \textit{\textbf{inst\_prm}}(C,E) \Rightarrow\\ &\quad \exists (E_1,E_2) \in \mathcal{E}(G)^2 \exists I \in \mathcal{I}(G) \quad (\textit{\textbf{is\_prm}}(I,E)) \wedge (I = spec\_type(E_1)) \wedge\\ &\qquad\qquad\qquad\qquad\qquad\qquad (C = freeze(E_2)) \wedge (E_1 \rightarrow^*_G E_2)\end{aligned}$$

En pratique, $\mathcal{E}^*(G)$ sera constitué des espèces définies explicitement par l'utilisateur. Une des conditions de l'équivalence de deux graphes sera l'égalité des deux ensembles distingués d'espèces. En effet, cela garantira que les manipulations effectuées ne modifieront pas le comportement des espèces “vu de l'extérieur”. Ainsi, toutes les relations d'héritage entre les espèces définies explicitement par l'utilisateur doivent être conservées, ainsi que les collections effectivement implantées.

Définition 10 *deux graphes* G *et* G' *sont* équivalents *si et seulement si*
- $\mathcal{C}(G) = \mathcal{C}(G')$
- $\mathcal{E}^*(G) = \mathcal{E}^*(G')$
- *Soient* $C \in \mathcal{C}(G)$ *et* $I \in \mathcal{I}(G) \cap \mathcal{I}(G')$

$$\begin{aligned}&(\exists E, F \in \mathcal{E}(G)^2\, (I = spec\_type(E)) \wedge (C = freeze(F)) \wedge (E \rightarrow^*_G F) \iff\\ &\quad ((\exists E', F' \in \mathcal{E}(G')^2 \quad (I = spec\_type(E')) \wedge (C = freeze(F')) \wedge\\ &\qquad\qquad\qquad\qquad (E' \rightarrow^*_G F') \wedge (\textit{\textbf{path}}(E,F) \simeq \textit{\textbf{path}}(E',F')))\end{aligned}$$

4.4. Ordonner les champs

La vision des espèces comme un ensemble de champs n'est pas suffisante pour réaliser une implantation correcte : il faut faire intervenir une notion d'ordre entre les champs, qui correspond au fait que ceux-ci dépendent les uns des autres. Par exemple, l'opération *div* définie dans les groupes est construite à partir des champs *mult* et *inv* de ces mêmes groupes : il est difficile de concevoir une espèce intermédiaire où *div* serait définie sans que *inv* ne soit déclaré. Plus précisément, on dira qu'un champ A *dépend de manière opaque* d'un champ B s'il faut connaître le type de B pour pouvoir définir (ou déclarer) A. En revanche, la définition de B n'est pas nécessaire à la déclaration (ou définition) de A. Ainsi, dans l'exemple ci-dessus, on a juste besoin de savoir que *inv* est une opération interne pour pouvoir définir la division, et cette définition sera valable quelle que soit l'implantation de *inv* qui sera faite dans la suite[2]. Si au contraire on a besoin aussi de la définition de B (comme c'est le cas dans un certain nombre de preuves,

[2] à condition bien sûr qu'elle respecte les propriétés associées à *inv*

l'associativité de `mult` dans un groupe, par exemple, dépend étroitement de l'implantation exacte de cette opération), on parle de *dépendance transparente*. Les dépendances liées à des définitions mutuellement récursives sont étudiées globalement par rapport à l'ensemble de ces définitions (c'est pourquoi on impose que de telles définitions soient introduites par le mot-clé `let rec` dans la syntaxe concrète). Traiter ainsi les définitions mutuellement récursives permet de définir un ordre strict en prenant la clôture transitive de la réunion des dépendances transparentes et opaques.

Désormais, une opération primitive ne peut être mise en œuvre que si certaines conditions de compatibilité avec cet ordre sont respectées. En particulier, il ne sera pas possible de définir un champ en faisant référence à un autre qui n'a été déclaré nulle part. L'analyse de la hiérarchie des espèces effectuée au cours de l'analyse du source utilisateur par le compilateur a donc pour but de construire un chemin qui soit compatible avec l'ordre induit par les dépendances. Pour ce faire, nous modélisons la définition donnée par l'utilisateur en deux temps : tout d'abord de manière "naïve", sans tenir compte des dépendances, en prenant simplement l'ordre dans lequel les champs se présentent (en partant des champs des espèces parent et en ajoutant les définitions nouvelles). Ensuite, on cherche à construire un chemin équivalent au précédent, mais compatible avec les dépendances présentes (le fait d'avoir une relation d'ordre garantit l'existence d'un tel chemin). C'est ce dernier qu'on utilisera dans la suite de la traduction vers OCAML et COQ (cf. [15]).

4.5. Héritage multiple

Comme cela a déjà été évoqué (p.61), un problème particulier se pose avec l'héritage multiple. En effet, un même nom de champ peut être présent dans deux (ou plusieurs) parents. A première vue, cela semble demander un traitement différent de ce qu'on a vu jusqu'à présent, et en particulier la définition d'une nouvelle opération prenant deux espèces en entrée. Cela n'est en fait pas nécessaire, et nous en donnons une interprétation à l'aide des seules opérations précédentes. On procède en deux temps.

Tout d'abord, on élimine les conflits de définitions. En effet, des méthodes peuvent être présentes avec des définitions différentes dans plusieurs espèces parent. Il va falloir alors sélectionner une définition, et effacer toutes les autres. Si en théorie on peut "panacher" les définitions des méthodes en conflit (en retenant certaines définitions de la première espèce, certaines de la deuxième, ...), la méthode retenue dans le traducteur consiste à garder la dernière définition (celle qui se trouve dans l'espèce située la plus à droite dans la liste des espèces parent). Il y a deux raisons à cela. D'une part, cela correspond à la sémantique de OCAML pour l'héritage multiple entre les classes, qui représentent nos espèces dans la traduction (p.68). D'autre part, permettre à l'utilisateur de reprendre des définitions de différentes espèces obligerait à vérifier que ses choix sont cohérents avec les dépendances entre méthodes. Par exemple, si une méthode a dépend de manière transparente de b dans deux espèces E_1 et E_2, on ne peut pas demander à conserver la définition de a de E_1 et celle de b de E_2.

Une fois qu'on a effectué les effacements nécessaires, il reste encore à ajouter pour chaque espèce ainsi obtenue, les méthodes présentes dans les autres espèces. Comme on a résolu tous les conflits lors de la phase précédente, cela se résume à une suite de déclarations et/ou de définitions, chaque espèce voyant les autres comme un simple "réservoir" de champs (ainsi, les monoïdes héritent leur champ *one* de *unitary_set* et leur champ *mult* de *semi_group*).

Enfin, il reste à effectuer des transformations sur les champs spécifiques à l'espèce fille, mais il ne s'agit plus d'opérations propres à l'héritage multiple.

5. Traduction en OCAML

Les expressions ont une traduction immédiate en OCAML. Les propriétés ne sont traduites que vers COQ. Il reste à préciser la représentation des espèces et des collections.

5.1. Espèces

La traduction en OCAML des espèces s'appuie sur l'implantation des espèces par Renaud Rioboo ([3]). Dans cette implantation, elles sont représentées par des classes, qui sont paramétrées par le type support de l'espèce et, comme on peut le voir sur l'exemple suivant, les types d'éventuels paramètres. :

source* FoC *original

`cartesian` prend deux collections en paramètre, et en fait le produit cartésien, le munissant d'une égalité

```
species cartesian(a is setoid,b is setoid) inherits setoid =
```
le type manipulé peut être défini en fonction de la représentation de a et b
```
  rep T = (a * b);
  let eqn(x in self,y in self) in bool =
      and_b (a.eqn (first(x),first(y)),b.eqn(scnd(x),scnd(y))) ;
```
il reste à faire les démonstrations correspondantes, déclarées dans `setoid`, mais le traducteur vers OCAML les ignore.
```
 demo eqn_refl = (*...*)
 demo eqn_sym = (*...*)
      ;;
```

***traduction en* OCAML**

on utilise 5 paramètres de type : un pour la classe elle-même, et 2 par paramètre d'espèce déclaré en FoC : le premier représente le type-support, le second le type de la classe correspondante
```
class [ 't , 'a, 'spec_a, 'b, 'spec_b]cartesian((a, b) :('spec_a*'spec_b)) =
  object(sobj :'self)
```
il convient de signaler les liens entre les différents paramètres de type
```
constraint 'spec_a=( 'a)#setoid
constraint 'spec_b=( 'b)#setoid
```
ici, le type support est connu, du moins en partie. On pourrait donc se passer du paramètre 't, mais cela rendrait impossible l'abstraction de type dans les modules définis ensuite
```
constraint 't = ('a*'b)

inherit [ ('a*'b)]setoid
```
la méthode eqn est définie à partir de celles de a et b : les contraintes jouent donc un rôle important à deux titres : garantir le typage et l'existence des méthodes
```
method eqn x y=
( and_b(a#eqn (first x ) ( first y )) (b#eqn ( scnd x)(scnd y)))
      end;;
```

5.2. Collections

C'est à ce niveau que se pose le problème de l'abstraction des types de données. Pour la réaliser, on va s'appuyer sur le système de modules de OCAML, en mettant en pratique les idées antérieures du projet [4]. Cependant, il apparaît que cette abstraction n'est pas si facile que cela à réaliser, en particulier dans l'implantation des espèces paramétrées, comme le montre l'exemple suivant.

Source FoC

```
collection int_int implements cartesian(int_mnd,int_mnd)=
```
Tout les définitions ont été données dans l'espèce `cartesian`. Il reste juste à passer les paramètres. `int_mnd` implante `monoid`, dont un des ancêtres est `setoid` : c'est donc un paramètre acceptable.
```
      ;;
```

fichier OCAML

```
module Int_int_ =
 struct
```
on commence par donner les types qui vont nous servir : ici, on utilise deux fois la collection `int_mnd`, et on récupère à la fois son type (abstrait) de représentation et le type-objet de l'espèce qu'elle représente
```
  type a_t = Int_mnd.rep
  type aspec = a_t setoid
  type b_t = Int_mnd.rep
```

```
  type bspec = b_t setoid
  type rep =(a_t*b_t)
fonctions pour définir et afficher les entités manipulées
  let  parse s = (*...*);;
  let print a = (*...*);;
  type stype = (rep , a_t,aspec, b_t,bspec)cartesian
le coeur du module : c'est ici qu'on définit l'objet composé des méthodes de la collection définie. La coercion est
obligatoire en vue des abstractions, pour que les paramètres soient liés correctement par OCAML.
  let obj = (new cartesian(Int_mnd.obj ,Int_mnd.obj ) :>stype)
end;;
La définition suivante correspond à l'interface à travers laquelle l'utilisateur verra la nouvelle collection
module type Int_int__ =
 sig
  type a_t
aspec n'est pas complètement abstrait : on se rappelle qu'on manipule un setoïde
  type aspec = a_t setoid
  type b_t
  type bspec = b_t setoid
la représentation non plus ne l'est pas : on sait encore qu'elle se compose d'une paire, dont les composantes, elles, sont
cachées
  type rep = (a_t*b_t)
  val parse  : string -> rep
  val print  : rep -> string
  type stype = (rep , a_t,aspec, b_t,bspec)cartesian
  val obj : stype
      end;;
```

L'abstraction telle qu'elle est réalisée ici, sur le type-support des paramètres plutôt que sur celui de l'espèce elle-même[3] est nécessaire, dans cette approche, pour que le typeur de OCAML puisse vérifier les contraintes exprimées dans la classe correspondante. Elle offre aussi l'avantage d'être relativement simple à mettre en place. Par ailleurs, comme une partie de l'information est masquée, il est effectivement impossible de créer un objet de type `rep` sans utiliser les fonctions prévues à cet effet, ce qui correspond à notre exigence de sécurité. Toutefois, ce n'est pas la seule manière de répondre à cette exigence, et le choix de la méthode adéquate est encore une discussion ouverte au sein du projet.

6. Traduction en COQ

Là encore, les expressions se traduisent à peu près directement en COQ. Par contre, la traduction des espèces pose problème, puisqu'on est loin d'avoir un système de classes aussi puissant que celui de OCAML, malgré la présence de macros permettant de gérer des **Records** et des **Coercions** entre ces derniers (cf. [1])

6.1. Espèces

Pour résoudre ces difficultés, on considère une espèce comme un ensemble ordonné de champs, tel qu'on l'a vu précédemment.

Toutes les informations relatives à une même espèce sont encapsulées dans une *Section* de COQ. On peut prendre l'exemple de l'espèce `monoid`, dont la définition en FoC est donnée p.61.

fichier COQ extrait

```
Section Monoid.
On définit un enregistrement, qui est l'interface de l'espèce considérée. Remarquons que tous les champs sont présents
dans l'enregistrement, ceux qui ont été hérité comme ceux qu'on ajoute dans la définition FoC.
 Record monoid : Type :=mk_monoid_ {
   monoid_T : Set;
   monoid_eqn : (monoid_T)->(monoid_T)->bool;
```

[3] sauf bien sûr si elle n'est pas paramétrée

```
   monoid_eqn_refl : (*...*)}.
```
Viennent ensuite les déclarations et définitions de champs. Là aussi, on décrit le statut de toutes les méthodes : celles qui sont déclarées sont représentées par des variables, et celles qui sont définies, par des définitions locales
```
  Variable self_T : Set.
  Variable self_eqn : (self_T)->(self_T)->bool.
  Hypothesis self_eqn_refl : ( x : self_T)( Is_true (( self_eqn x x ))).
  (*...*)
  Variable self_one : self_T.
```
is_one provient d'un parent (ici unitary_set) : on applique la fonction unitary_set__is_one aux champs de monoid dont dépend is_one. Ce comportement sera étudié en détail dans la suite du texte (p.71).
```
  Local self_is_one : (self_T)->bool := (unitary_set__is_one self_T
      self_eqn self_one ).
```
Ici, unique_one_l est une nouvelle définition. On va donc la définir en deux temps : tout d'abord une fonction, dont les arguments sont les champs dont unique_one_l dépend de manière opaque...
```
  Definition monoid__unique_one_l : ( self_T : Set)
   (self_eqn : self_T->self_T->bool)
   (self_eqn_refl :(x :self_T)(Is_true (self_eqn x x )))
   (*...*)
     (x : self_T)(y :self_T)(Is_true (self_eqn (self_mult x y) y ))->
      (Is_true ( self_eqn x self_one)).
  Proof.
    Intros.
    Apply (self_eqn_trans0 x (self_mult0 x self_one0) self_one0);Auto.
  Defined.
```
...Puis un théorème local obtenu en appliquant la fonction précédente aux champs de monoid
```
        Local self_unique_one_l : (x : self_T)(y : self_T)
            (Is_true (self_eqn (self_mult x y) y ))->
              ( Is_true (self_eqn x self_one )) :=
           (monoid__unique_one_l self_T self_eqn self_eqn_refl
             self_eqn_sym self_eqn_trans self_mult self_assoc self_one
             self_neutral_right ).

 (*...*)
```
Une fois tous les champs décrits, il n'y a plus qu'à créer l'enregistrement correspondant à monoid
```
        Definition mk_monoid := ( mk_monoid_ self_T self_eqn (*...*)
        Class monoid.

    End Monoid.
```
On définit des coercions par rapport aux espèces parents
```
  Definition monoid_semi_group[S :monoid] :=(mk_semi_group (monoid_T S) (monoid_eqn S) (*...*).
  Coercion monoid_semi_group : monoid>->semi_group.

Definition monoid_unitary_set[S :monoid] :=(mk_unitary_set (monoid_T S) (*...*).
      Coercion monoid_unitary_set : monoid>->unitary_set.
```

D'éventuels paramètres seront représentés par des champs du **Record** et des variables. Une première version de la traduction conservait des paramètres à l'extérieur de l'enregistrement, comme en OCAML et dans la syntaxe de départ, mais posait différents problèmes. D'une part, ils pouvaient être "oubliés" à la fermeture de la section, s'il n'apparaissaient pas explicitement dans les fonctions exportées[4]. D'autre part, il était difficile de trouver une représentation satisfaisante de l'héritage entre espèces paramétrées. En effet, les paramètres n'apparaissant pas explicitement au sein des **Record**, nous n'avions aucun moyen de lier, lors d'une coercion, les paramètres de l'espèce fille avec ceux de l'espèce parent. Or cette opération est nécessaire. Par exemple, si on définit le produit cartésien de deux monoïdes, qui va hériter du produit de deux ensembles quelconques, on aura :

```
species cartesian_mnd (A is monoid, B is monoid)
  inherits cartesian(A,B) (*...*)
```

[4]quand on ferme une section, COQ transforme les définitions globales en fonctions des variables dont elles dépendent et *d'elles seules*

Il faut avoir un moyen en Coq d'exprimer que c'est le monoïde `A` (ou plutôt le setoïde sous-jacent) qui sert lorsqu'on utilise `cartesian_mnd` comme `cartesian`.

Enfin, il faut accorder une certaine attention à la gestion correcte de la liaison tardive. Elle joue un rôle certain en permettant aux méthodes d'utiliser la dernière définition des méthodes dont elles dépendent de manière opaque, sans avoir à les redéfinir elles-mêmes. Pour réaliser cela, nous avons choisi de "personnaliser" les définitions : chacune est transformée en une fonction qui attend un certain environnement – les méthodes dont elle dépend de manière opaque – et que chaque espèce va fournir. Cette approche est assez similaire à la représentation d'un objet sous forme d'un lambda-terme, comme on peut la trouver par exemple dans [6]. Dans cette représentation, un objet est en fait le point fixe d'une fonction dont l'argument est un enregistrement de même type que l'objet, pour pouvoir gérer correctement les méthodes mutuellement dépendantes. Ici, grâce à l'analyse des dépendances qu'on a faite au préalable, on peut se contenter d'utiliser une version simplifiée On n'a en effet pas besoin de prendre toutes les méthodes de l'espèce en argument, et de les définir toutes à la fois. Au contraire, on peut se permettre de donner les définitions une par une, en ne prenant à chaque fois que les méthodes strictement nécéssaires en argument. Cela permet de fournir à Coq des termes relativement plus simples à typer.

```
Section example.
...
On se place à l'intérieur d'une section. Un champ défini le sera en deux temps
Definition abstr_field :=[depend_opaq_1 :type_1][depend_opaq_2 :type2]...
    body.
Dans le corps, les appels aux méthodes définies précédemment sont remplacés par des appels à depend_opaq_i
Local self_field :=(abstr_field self_depend_opaq_1 self_depend_opaq_2 ...).
...
End example.
En cas de redéfinition d'une des méthodes dont dépend field, il suffira de refaire un appel à abstr field.
Section new_def.

Local self_field :=(abstr_field new_self_depend_opaq_1 new_self_depend_opaq_2 ...).
on utilise la définition des champs depend_i présente dans new def
      End new_def.
```

6.2. Collections

Contrairement au cas de Ocaml la traduction des collections en Coq est moins compliquée que celle des espèces : il suffit d'indiquer, que chacun des champs représentant le type-support des espèce est `Opaque`. Ainsi, Coq ne cherchera plus à l'évaluer.

7. Perspectives

Un certain nombre d'améliorations doivent être apportées au système tel qu'il se présente actuellement. En premier lieu, si le début de la hiérarchie mathématique a été décrit en FoC, on est encore loin d'avoir atteint la bibliothèque existante en Ocaml. Il convient donc de créer un certain nombre d'espèces et de collections. Cela permettra de vérifier que la traduction s'effectue correctement. De même, il faudra à cette occasion améliorer la gestion des fonctions (mutuellement) récursives. Cela devrait passer par des constructions analogues à la tactique `Program` de Coq, telle qu'on peut la voir dans [12]. Cela demandera sans doute d'ailleurs de créer un certain nombre d'espèces et collections pour les arguments de ces fonctions. Dans ces collections apparaîtront des propriétés dédiées aux preuves de terminaison, comme l'existence d'un ordre bien-fondé. Le développement de l'interface utilisateur et de la librairie sont donc loin d'être indépendants.

De plus, divers points sémantiques doivent être éclaircis. Ainsi, il convient d'examiner de près les problèmes que posent d'éventuelles collections paramétrées. Ces dernières semblent en effet avoir une utilité certaine dans différents développements, mais ne s'intègre pas forcément très bien dans le modèle qu'on a vu. Parallèlement, on pourrait être tenté d'autoriser les collections à implanter plusieurs espèces (les entiers peuvent être vus comme un anneau, un ensemble totalement ordonné, ...). D'autres questions se posent pour d'éventuelles méthodes locales au sein des espèces (et en particulier sur leur comportement vis-à-vis de l'héritage).

Par ailleurs, il conviendra à terme de prendre en compte des représentations plus complexes pour les collections, en utilisant comme type-support des sous-ensembles de types existants, définis par des prédicats. Par exemple, une représentation des entiers modulo n pourra exiger que les entités qu'elle manipule, non seulement soient de type `int` ou `nat`, mais encore qu'elles soient comprises entre `0` et `n-1`. en question. On pourra s'inspirer de travaux similaires en HOL et PVS, comme [8], pour réaliser l'implantation de ces concepts en Coq. En Ocaml on devrait sans doute n'avoir à faire des tests que lors de la création des entités par `parse`, puisque Coq nous garantira que les autres fonctions renvoient un résultat ayant les bonnes propriétés.

Enfin, et c'est sans doute là le point le plus important, il va falloir s'atteler à la tâche de définir un langage de preuve simple, et facilement traduisible en Coq. Il devrait s'inspirer des travaux de Damien Doligez, mais il reste encore à le définir, et surtout à examiner les moyens que l'on aura d'interagir avec Coq.

8. Conclusion

En conclusion, on peut espérer à l'heure actuelle avoir une syntaxe relativement solide et des mécanismes de traduction de la hiérarchie des espèces assez bien établis. L'interprétation de l'héritage en terme d'opérations plus primitives impliquant un seul champ à la fois semble avoir porté ses fruits en nous permettant d'obtenir une représentation simple – du moins assez pour que Coq la comprenne dans des délais raisonnables – des espèces. Par rapport au travail de Sylvain Boulmé on perd certes, avec la formalisation complète de la théorie, la preuve Coq (et il faudra donc montrer la correction de la traduction tant en Ocaml qu'en Coq) que toutes les opérations que nous faisons subir aux espèces sont correctes par rapport à la sémantique qu'on s'est fixée. En revanche, cela nous permet de manipuler des objets plus petits, que le vérificateur de type de Coq arrive à typer rapidement, même quand le nombre de champs augmente notablement. Partant de ces bases, on peut donc envisager avec un certain optimisme de faire subir à ces structures le test à grande échelle que sera le portage de la librairie actuelle en Ocaml vers FoC, d'autant plus que les essais existant actuellement fonctionnent parfaitement.

9. Remerciements

Je remercie les membres de l'équipe FoC, auxquels ce travail doit énormément, et plus généralement les membres des équipes SPI et CALFOR. Je remercie également les habitants des bâtiments 8 et 11 de l'INRIA Rocquencourt pour leur disponibilité. Enfin, je remercie les rapporteurs de ce papier, pour leur relecture attentive et les nombreuses améliorations qu'ils ont suggérées.

Références

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Munoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof-assistant reference manual.* INRIA, 6.3.1 edition, 12 1999. http ://pauillac.inria.fr/coq/doc/main.html.

[2] S. Boulmé. Opérateurs de raffinement sur les structures algébriques. In *Actes des JFLA 2000*, Janvier 2000.

[3] S. Boulmé, T. Hardin, V. Ménissier-Morain, and R. Rioboo. Bilan du projet foc. Technical report, Laboratoire d'Informatique de Paris 6, 1999.

[4] S. Boulmé, T. Hardin, and R. Rioboo. Modules, objets et calcul formel. In *Actes des JFLA'1999*, 1999.

[5] M. Bronstein, W. Burge, T. Daly, P. Gianni, J. Grabmeier, and R. Genks. *AXIOM User Guide*. NAG.

[6] G. Castagna. Subtyping and object-oriented programming. notes de cours du D.E.A. Programmation, 2000.

[7] J.-C. Filliâtre. Proof of Imperative Programs in Type Theory. Research Report 97–24, LIP - ENS Lyon, July 1997. ftp ://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR97/RR97-24.ps.Z.

[8] M. D. Jones. Representing abstract theories using predicates type. Master's thesis, Brigham Young University, June 1997.

[9] L. Lamport. How to write a proof. research report, Digital Equipments Corporation, February 1993.

[10] 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/.

[11] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st symp. Principles of Programming Languages*, pages 109–122. ACM press, 1994.

[12] C. Parent. Developing certified programs in the system Coq - the program tactic. In H. Barendregt and T. Nipkow, editors, *Proceedings of Types'93*, 1994. également rapport LIP-ENS Lyon RR 93-29 and by anonymous ftp from lip.ens-lyon.fr file pub/Rapports/RR/RR93/RR93-29.ps.Z.

[13] R. Pollack. Dependently typed records for reprensenting mathematical structure. Technical report, Durham University, June 2000.

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

[15] V. Prevosto. Vers une interface utilisateur pour foc. Rapport de dea, Université Paris 6, Septembre 2000.

[16] B. Werner. An encoding of zermolo-fraenkel set theory in coq. http ://pauillac.inria.fr/coq/contribs-eng.html.