Opérateurs de raffinement sur les structures algébriques

Sylvain Boulmé

Laboratoire d'Informatique de Paris 6,
4 place Jussieu
75252 Paris cedex 05
`Sylvain.Boulme@lip6.fr`

Résumé

Ce papier présente informellement un système, nommé FocKer pour spécifier des bibliothèques de programmes de calcul formel. Dans ce système, les spécifications sont construites les unes à partir des autres, grâce à deux mécanismes : abstraction/application (comme en λ-calcul) et raffinement. Il présente comment "extraire" de ces spécifications des bibliothèques Ocaml. Il montre enfin un début de formalisation de FocKer en Coq.

1. Introduction

Le projet Foc (Formel + Ocaml + Coq) a pour objectif de bâtir un environnement de programmation certifiée pour le calcul formel. Des librairies de programmes de calcul formel, écrites en Ocaml, devraient à terme être certifiables en Coq([3]). En nous basant sur la présentation mathématique des structures algébriques et sur l'expérience d'Axiom (cf. [13]), nous avons mis au point une méthodologie de construction[4] originale des librairies de calcul formel, qui sont écrites en Ocaml[8]. Nous avons profité de la puissance des mécanismes des objets et des modules de Ocaml, de manière à avoir des programmes proches de leur interprétation mathématique, tout en favorisant la réutilisabilité du code, sans nuire toutefois à son efficacité.

Il s'agit maintenant d'associer un système de *spécifications formelles* à cette méthodologie de développement. Cet article propose donc un système de spécification des librairies de calcul formel du système Foc.

Dans ce système, chaque unité de librairies correspond à une "vue" d'une structure algébrique (groupes, anneaux, corps, ...) ou d'un domaine mathématique (comme l'ensemble $\mathbb{Z}$ muni de ses différentes opérations usuelles). Une "vue" peut être totalement abstraite (très proche d'une définition mathématique) ou entièrement concrète (décrivant une implantation particulière) ou mélant des traits encore abstraits et des traits concrets. Les unités de librairies sont bâties à partir de deux mécanismes : abstraction/application (comme en λ-calcul) et raffinement.

La section 2 présente ce système. La section 3 montre comment ce système se place dans Foc. Les sections 4, 5 et 6 montrent un début de formalisation de ce système en Coq. La section 7 indique des extensions à apporter, à la fois au niveau de la formalisation en Coq, et du système de spécification proprement dit.

2. Le système FocKer

Le système FocKer, dont le nom signifie Foc Kernel, constitue le noyau de la formalisation du système Foc.

2.1. Espèces, collections et interfaces

Informellement une *espèce* est une structure algébrique dont le type support est abstrait. Une *collection* correspond à un “domaine mathématique”. Nous allons préciser ces notions.

Espèces de base Une espèce de base est mathématiquement déterminée par un ensemble d'opérations et un ensemble d'hypothèses vérifiées par ces opérations. Dans un esprit “isomorphisme de Curry-Howard”, on désignera indifféremment ces opérations et ces hypothèses sous le nom de *méthodes primitives.* Ces méthodes primitives sont donc données sous la forme d'un couple *(nom, signature).* Pour cette raison, on dit aussi que ces méthodes sont *déclarées.*

À partir de ces méthodes primitives, on peut déduire et définir un certain nombre d'opérations et de théorèmes. On les désigne ici sous le nom de *méthodes dérivées.* Ces méthodes sont données sous la forme *(nom, signature, valeur)*, et sont donc aussi qualifiées de *définies.*

Une *espèce* est donc un ensemble de méthodes déclarées ou définies.

Interfaces de base À chaque espèce de base, on peut associer une interface qui correspond à l'ensemble des couples *(nom, signature)* des méthodes primitives et dérivées : on “oublie” les valeurs associées aux méthodes dérivées. Dans une interface on ne distingue donc plus les méthodes primitives des méthodes dérivées.

Collections de base Une *collection* est un ensemble de méthodes définies. Une collection B *implante* une espèce A, si toutes les méthodes primitives de A ont été implantées dans B. B peut donc être définie uniquement en implantant les méthodes primitives de A : les méthodes dérivées sont définies dans B gratuitement. L'interface associée à l'espèce B peut être considérée comme une signature acceptable pour la collection A.

Espèces & collections paramétrées Une espèce peut prendre en paramètres plusieurs collections. Cela signifie que ces collections sont uniquement spécifiées sous la forme *(nom,interface).* Les méthodes d'une telle espèce ne peuvent dépendre que des couples *(nom,signature)* des méthodes de ses arguments. Il est bien sûr possible d'instancier les espèces paramétrées en passant des collections en argument.

De même, une collection peut prendre en paramètres plusieurs collections.

2.2. Raffinement

On introduit maintenant un ensemble d'opérateurs de raffinement sur les espèces et les collections. Grosso-modo, si A et B sont des collections ou des espèces et si A raffine B, alors l'interface de B est incluse dans celle de A modulo éventuellement renommage de méthodes. Ainsi “tout ce qu'on peut faire avec B, on peut le faire avec A”. Mais, comme précisé ci-dessus, cette notion de raffinement n'est pas une relation entre spécifications, mais le résultat d'opérateurs sur les spécifications. Ainsi le fait que A raffine B est un élément de la spécification de A : cela sert à le construire. Cela rend FocKer extensible car toute opération dérivée ajoutée à B est automatiquement propagée dans A. On gagne aussi en concision, car la définition de A n'a à fournir que les méthodes primitives et les méthodes dérivées à ajouter à B ou à redéfinir.

Le raffinement est construit à partir de 3 primitives : l'enrichissement, la spécialisation, et la convertibilité :

1) La construction par enrichissement prend une ou plusieurs espèces ou une collections, les fusionne et ajoute de nouvelles méthodes primitives ou dérivées. L'interface d'une espèce construite par enrichissement contient strictement les interfaces des espèces utilisées dans la construction.

2) La construction par spécialisation prend une espèce ou une collection E et ne change pas son interface. Elle définit des méthodes primitives de E ou redéfinit des méthodes dérivées.
En cas de redéfinition de méthodes dérivées, certains lemmes de l'espèce obtenue par

spécialisation doivent être redémontrés, car leur démonstration dépendait de la définition des méthodes redéfinies (et ces définitions ont changé). Notons que l'utilisateur a interêt à minimiser le nombre de lemmes dont la démonstration dépend de définitions (et pas uniquement de la signature).

Cela nous amène à une contrainte sur le graphe de dépendances entre méthodes, qui doit acyclique afin de garantir la cohérence logique des spécifications. On pourrait sans doute affaiblir cette contrainte en autorisant certaines formes de récursions mutuelles. Les exemples de 2.3 précisent ces notions de dépendances.

3) Soit A une espèce raffinant B. L'opération de *conversion* permet d'utiliser une collection d'interface A en tant que collection d'interface B : on "oublie" les méthodes ajoutées au cours de l'enrichissement.

Exemple de raffinement Par définition un anneau factoriel (constructif) possède une fonction totale de calcul du pgcd. Un anneau euclidien possède une division euclidienne et le pgcd y est obtenu par l'algorithme d'Euclide. Un anneau euclidien peut être construit par enrichissement d'un anneau factoriel à l'aide de la division euclidienne et par spécialisation à partir de ce même anneau factoriel en définissant le pgcd, opération primitive de l'anneau factoriel.

Lorsqu'on code une collection A (par exemple, la collection des entiers relatifs) implantant l'espèce des anneaux euclidiens, on n'aura pas à expliciter le calcul du pgcd dans cette collection, puisqu'il existe un algorithme général, au niveau des anneaux euclidiens. Par contre, pour passer A comme argument de la collection paramétrée des anneaux factoriels de polynômes, par exemple, il faut convertir A en un anneau factoriel.

2.3. Exemples dans une syntaxe informelle

Pour rendre plus concret ce système au lecteur, on donne quelques exemples dans une syntaxe virtuelle, inspirée de celle de Coq. Les preuves sont donc des λ-termes. On utilise le fait que $\neg P$, où P est une proposition, est défini comme $P \to \texttt{False}$.

Exemple d'espèce L'espèce des setoïdes peut être décrite comme suit (pour la concision de l'exemple, on suppose juste la relation `eq` réflexive) :

$$
\mathit{setoid} := \left\{ \begin{array}{ll}
\texttt{decl} & T : \texttt{Type} \\
\texttt{decl} & \mathit{eq} : T \to T \to \texttt{Prop} \\
\texttt{decl} & \mathit{eq\_refl} : \forall(x:T)(\mathit{eq}\ x\ x) \\
\texttt{def} & \mathit{neq} : T \to T \to \texttt{Prop} \\
 & := \lambda[x,y:T]\neg(\mathit{eq}\ x\ y) \\
\texttt{def} & \mathit{neq\_spec} : \forall(x,y:T)(\mathit{neq}\ x\ y) \to \neg(\mathit{eq}\ x\ y) \\
 & := \lambda[x,y:T; H:\neg(\mathit{eq}\ x\ y)]H \\
\texttt{def} & \mathit{neq\_nrefl} : \forall(x:T)\neg(\mathit{neq}\ x\ x) \\
 & := \lambda[x:T; H:(\mathit{neq}\ x\ x)](\mathit{neq\_spec}\ x\ x\ H\ (\mathit{eq\_refl}\ x))
\end{array} \right\}
$$

Les dépendances entre méthodes apparaissent dans les signatures ou les définitions des méthodes et sont de deux sortes, comme en Coq. On dira que la méthode m_2 dépend de manière *opaque* de m_1, si pour donner la signature ou la définition de m_2, on a besoin de la signature de m_1, mais pas de sa définition. De même, on dira que la méthode m_2 dépend de manière *transparente* de m_1, si pour donner la définition de m_2, on a besoin de la signature et de la définition de m_1.

Par exemple, $\mathit{neq\_nrefl}$ dépend de manière opaque de toutes les autres méthodes de *setoid* (par clôture transitive). Par contre, $\mathit{neq\_spec}$ dépend de manière transparente de neq, mais de manière opaque de eq et de T.

Exemple d'espèce paramétrée A partir de 2 setoïdes, on peut construire l'espèce des morphismes de setoïdes. Dans l'exemple suivant, on suppose que si A désigne une collection, on accède à une méthode T de A, via l'expression $A.T$:

$$map[A, B : setoid] :=$$

$$\left\{ \begin{array}{ll} \texttt{decl} & ap: \ A.T \to B.T \\ \texttt{decl} & ap\_eq: \ \forall(x, y : A.T)(A.eq\ x\ y) \to (B.eq\ (ap\ x)\ (ap\ y)) \\ \texttt{def} & prop: \ \forall(x, y : A.T)(B.neq\ (ap\ x)\ (ap\ y)) \to \neg(A.eq\ x\ y) \\ & := \ \lambda[x, y : A.T; H : (B.neq\ (ap\ x)\ (ap\ y)); H' : (A.eq\ x\ y)] \\ & \qquad\qquad (B.neq\_spec\ (ap\ x)\ (ap\ y)\ H\ (def\ x\ y\ H')) \end{array} \right\}$$

Exemple d'une collection paramétrée avec raffinement et redéfinition On va construire la collection du setoïde de morphismes de setoïdes.

$$map\_setoid[A, B : setoid] :=$$

$$\left\{ \begin{array}{l} \texttt{refine}\ setoid \\ \ \texttt{def} \quad T := (map\ A\ B) \\ \ \texttt{def} \quad eq := \lambda[p, q : T]\forall(x : A.T)(B.eq\ (p.ap\ x)\ (q.ap\ x)) \\ \ \texttt{def} \quad eq\_refl := \lambda[p : T; x : A.T](B.eq\_refl\ (p.ap\ x)) \\ \ \texttt{def} \quad neq := \lambda[p, q : T]\exists[x : A.T](B.neq\ (p.ap\ x)\ (q.ap\ x)) \\ \ \texttt{def} \quad neq\_spec := \lambda[p, q : T; H]... \end{array} \right\}$$

La seule condition pour que la (re)définition d'une méthode m soit valide, est qu'elle respecte le type de cette méthode, si celle-ci a déjà été préalablement déclarée.

Dans l'espèce ci-dessus, comme on raffine *setoid*, on récupère gratuitement neq_nrefl : comme neq_nrefl ne dépend d'aucune méthode de manière transparente, sa définition reste cohérente avec son type, même si les autres méthodes sont redéfinies. On aurait pu aussi récupérer de la même façon neq, mais on choisit ici de le redéfinir. Cette redéfinition impose de redéfinir aussi toutes les méthodes qui dépendaient de manière transparente de neq : soit ici neq_spec.

3. Intégration de FocKer en Ocaml et en Coq

Les spécifications de FocKer doivent être reflétées dans le développement en Coqet en Ocaml. Dans la mesure où le langage de FocKer est proche de celui de Coq, il serait intéressant de traduire les spécifications dans Coq et d'en déduire le code Ocaml, par une démarche analogue à celle de [7]. Nous n'allons pas étudier ce point de vue ici, mais simplement montrer comment le code Ocaml suit les spécifications et comment traduire FocKer en Coq.

3.1. Côté Ocaml

Dans [4], on a détaillé les choix d'implantations de Foc. Les espèces sont codées par des classes virtuelles paramétrées par le type support, et les collections par des classes concrètes. Les interfaces se retrouvent dans les types objets associés à chaque classe. Les opérations primitives sont des méthodes virtuelles, et les opérations dérivées sont des méthodes concrètes. L'enrichissement correspond à l'héritage. La convertibilité est exprimée essentiellement comme instanciation de type polymorphe (instanciation de variables de rangée dans les types objets).

Les dépendances entre méthodes sont exprimées grâce à la liaison tardive, ce qui permet de bien encoder la redéfinition de méthode. Par contre, il n'y a pas de vérification que les dépendances entre méthodes sont acycliques. Toutes les dépendances entre méthodes sont ici opaques, puisqu'ici on ne fait pas de preuves.

L'architecture de FocKer est donc encodée de manière satisfaisante en Ocaml. Cependant, dans 7.2, on montrera les limites de FocKer, qui n'est pas suffisant pour exprimer tous les programmes Ocaml, que l'on souhaiteraient spécifier formellement dans Foc.

3.1.1. Exemples d'espèces et de collections de bases

On illustre ici l'exemple des anneaux factoriels et euclidiens donné en 2.2.

On introduit tout d'abord deux classes virtuelles paramétrées par leur type support, correspondant à l'espèce des anneaux factoriels et l'espèce des anneaux euclidiens.

```
class virtual ['a] anneau_factoriel =
object
  inherit ['a] anneau_integre
  method virtual pgcd: 'a->'a->'a
  ...
end

class virtual ['a] anneau_euclidien =
object (self)
  inherit ['a] anneau_factoriel
  method virtual reste_eucl: 'a->'a->'a
  method pgcd =
    let rec pgcd_rec x y =
      if (self#eq_zero y)
      then x
      else (pgcd_rec y (self#reste x y))
    in pgcd_rec
  ...
end
```

On définit alors la collection des entiers relatifs, qui spécialise la notion d'anneau euclidien.

```
class entiers_relatifs =
object
  inherit [int] anneau_euclidien
  method reste_eucl x y = x mod y
  ...
end
```

Notons que sur cette dernière classe l'implantation du type support est visible de l'extérieur de la classe, ce qui peut induire l'utilisateur en erreur et nuit à la modularité du code. On résoud ce problème en encapsulant les objets associés aux collections dans des modules, et en rendant le type support abstrait. (cf. [4])

3.1.2. Exemples d'espèces et de collections paramétrées

On introduit maintenant l'espèce des algèbres.

```
class virtual ['c,'a] algebre (coef:'c #anneau)
= object (self)
  inherit ['a] anneau
  (* plongement de l'anneau de base dans l'algèbre *)
  method virtual lift: 'c -> 'a
  (* multiplication par un coefficient *)
  method mult_extern c p =
     self#mult (self#lift c) p
  ...
end
```

On peut aussi donner une collection implantant les algèbres de polynômes. Pour simplifier la présentation, on se limite au cas des polynômes en une variable dont les degrés sont les entiers.

```
class ['c] poly_naifs (coef:'c #anneau) =
object
  inherit ['c,('c,int) list] algebre coef
  method lift c = [(c,0)]
  ...
end
```

Là encore, l'implantation du type support sera masquée grâce au système de modules.

3.2. Côté Coq

Pour mesurer si FocKer répond bien à nos attentes, nous l'avons traduit en Coq. Techniquement, on réalise un plongement superficiel ("shallow embedding"). Cela signifie que les types et les termes associés aux méthodes de FocKer sont des termes Coq, manipulés avec la logique de Coq. Cela nous évite de définir notre propre système logique. Cette traduction peut donc être vue comme une "extension" de Coq restant à l'intérieur de Coq. On obtient ainsi assez simplement un système de spécification, dont la cohérence et l'implantation sont garanties par Coq.

Le gros inconvénient de cette technique est que les calculs inhérents à notre système passent toujours par l'intermédiaire de Coq, et coûtent donc relativement cher.

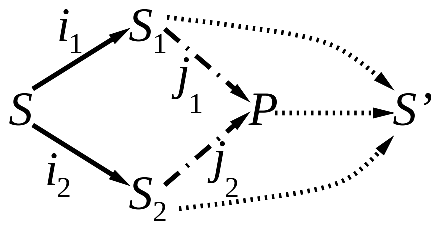
3.2.1. Présentation du problème

Dans un premier temps, on laisse de côté tout le mécanisme de méthodes dérivées et de méthodes primitives. On y reviendra en 7.1. On décrit donc en fait un système d'*enregistrements extensibles avec champs dépendants*. La notion d'espèce ne se retrouve pas dans ce système, par contre les interfaces se traduisent en *signatures d'enregistrements*, et on peut considérer que les enregistrements reflètent la notion de collections.

Dans ce système, l'enrichissement correspond à l'extension de signature. Il n'y a pas de notion d'enrichissement sur les enregistrements eux-mêmes. Comme pour l'enrichissement, l'extension de signature, correspond pour nous à l'ajout de méthodes *(nom,type)*, avec renommage.

Le point un peu délicat concerne en fait *l'extension multiple* de signatures, qui correspond à la notion de colimite de la théorie des catégories (voir par exemple [9]). Cette notion repose sur celle de *somme amalgamée* que nous allons étudier.

On donne trois signatures S, S_1 et S_2, telles que S_1 et S_2 étendent S. P est la *somme amalgamée* de S_1 et S_2 sur S, si P étend S_1 et S_2, et si pour toute signature S' qui étend S_1 et S_2, alors S' étend P.



Lorsqu'un nom de méthode apparaît dans S_1 et S_2, mais pas dans S, cela signifie qu'une des deux méthodes de S_1 ou S_2 doit être renommée, avec un nom nouveau.

Pour simplifier le problème et afin de laisser à l'utilisateur le contrôle sur les noms, on décide de bien distinguer l'extension (ajout de méthodes à une signature) du renommage, en deux opérations distinctes. Ce sera de plus cohérent avec l'implantation en Ocaml, où la conversion correspondant au renommage est beaucoup plus coûteuse que la conversion correspondant à l'oubli de champs.

Ainsi, dans la définition des sommes amalgamées précédente, le “étend” correspondra désormais à une extension sans renommage. On imposera alors que tous les noms de méthodes communs à S_1 et S_2 soient dans S. On appelle cette restriction des sommes amalgamées la *fusion* de S_1 et S_2. Étant données deux signatures quelconques, il n'en existe pas forcément une fusion : il faut pouvoir exhiber une signature S satisfaisant la condition précédente. Mais l'utilisateur a alors toujours la possibilité de renommer les méthodes en conflit avant de faire la fusion.

De même, lorsque l'utilisateur ne veut pas faire la somme amalgamée de deux signatures sur leur intersection, il a toujours la possibilité de renommer les méthodes à distinguer.

Dans la section 4, on présente une théorie des enregistrements. Puis dans la section 5, on présente ces notions d'extension de signatures. Dans la section 6, on donne un exemple d'utilisation des enregistrements. Enfin, dans la section 7, on présente des extensions à cette théorie, notamment afin de mieux exprimer la notion d'espèces du système FocKer.

Mais avant de présenter cette théorie, on indique pourquoi les enregistrements actuels de Coq sont insuffisants dans notre cadre.

3.2.2. Les enregistrements de Coq

En fait, via des mécanismes de macro, Coq offre déjà un système d'enregistrements extensibles. Cela a permis notamment de décrire la théorie des catégories en Coq (cf. [12]) et aussi une partie de l'algèbre (cf. [10]). Mais ces mécanismes sont insuffisants pour le but recherché ici. D'une part, les enregistrements actuels de Coq ne résolvent pas le problème de l'espace de noms. D'autre part, il n'est pas possible de faire de l'extension multiple avec partage de signatures.

Problème de l'espace de noms En Coq, deux enregistrements de type distinct ne peuvent pas partager un même nom de méthode. Plus exactement, comme il est possible de simuler de l'extension de signatures, il est donc possible de faire partager, par ce mécanisme d'extension, un même nom de méthode à des enregistrements de types distincts. Mais deux enregistrements dont les types n'appartiennent pas à la même composante connexe du graphe d'extension de signatures ne peuvent pas partager la même méthode.

Cela signifie que pour rajouter un type-enregistrement nouveau, on peut avoir besoin de revoir complètement la forme du graphe d'extension de signatures.

Problème de l'extension multiple S'il existe une extension multiple en Coq, celle-ci ne permet pas de faire un partage de méthodes des différentes classes qu'on étend. Dans le diagramme précédent, cela signifie qu'on ne sait faire des sommes amalgamées qu'avec $S = \emptyset$, c'est-à-dire finalement, des unions disjointes.

Là encore, on peut s'en sortir (cf. [10]), mais l'insertion d'une classe dans le graphe d'extension peut souvent nécessiter de revoir une bonne partie du graphe.

4. Une théorie Coq des enregistrements avec champs dépendants

D'un point de vue abstrait, un enregistrement n'est en fait rien d'autre qu'une fonction d'un certain sous-ensemble fini d'étiquettes, vers un ensemble de champs. L'application pour un enregistrement correspond alors à *l'accès aux champs.*

Conventions Tout d'abord, on paramètre l'ensemble de la théorie, par le type `A` (de sorte `Set`) des étiquettes, dont on suppose l'égalité décidable. On supposera définis un certain nombre de prédicats "ensemblistes" (qu'on appellera *gardes*) sur les listes d'étiquettes : $\in$ le prédicat qui indique si une étiquette appartient à une liste d'étiquettes ; $\supset$ le prédicat qui indique si une liste en contient une autre ; $\not\pitchfork$ le prédicat qui indique si deux listes n'ont pas d'étiquettes en commun ; $\hat{.}$ le prédicat unaire qui indique si une liste est sans doublon. On suppose aussi définie une fonction infixe $\backslash$ qui retourne une liste des éléments de la liste de gauche qui ne sont pas dans celle de droite.

Le code Coq présenté est en mode "arguments implicites" : les arguments des fonctions qui peuvent être déduits des autres (via les dépendances de types) restent implicites. Ils sont inférés par le vérificateur de type. Comme cet algorithme n'est pas complet, il est parfois nécessaire d'expliciter les arguments normalement implicites. La syntaxe Coq permet de les introduire sous la forme n`!`t où n est la position de l'argument implicite, et t cet argument (voir en 5.1).

4.1. Notre notion d'enregistrement

On introduit le type `sign` de sorte `Type` des *signatures d'enregistrement* et le prédicat `record` de type `sign`$\to$`Type`. Si `s` est un terme de type `sign`, un enregistrement est donc un terme de type `(record s)`. On a une fonction `sign_l` qui retourne la liste des étiquettes d'un enregistrement. Dans la suite on note les expressions `(sign_l s)` avec |`s`|.

L'accès aux champs des enregistrements est défini via deux fonctions, `field` qui prend un enregistrement et une étiquette, et retourne le champ associé à cette étiquette dans l'enregistrement, et `fieldT` qui prend un enregistrement `i` et une étiquette `a`, et qui retourne le type de `(field i a)`. Plus formellement :

```
fieldT: (s:sign)(record s)->A->Type
field:(s:sign; i:(record s); a:A; H:(a∈|s|))(fieldT i a)
```

En fait, dans notre contexte, la fonction `fieldT` n'a de sens que si l'étiquette est bien un nom de champ de l'enregistrement sur lequel on calcule un accès. Ainsi, l'utilisateur n'est pas censé se servir directement de `fieldT`. À priori, il n'en a guère besoin. (Ceci dit, on pourrait facilement rajouter une fonction qui demanderait à décharger une garde avant d'utiliser `fieldT`).

La difficulté est donc maintenant de donner un codage de `sign` et `record` qui exprime les dépendances entre les champs.

4.2. Les têlescopes

Une paire avec champs dépendants peut se coder de la manière suivante :

```
Structure dpair[T:Type; f:T->Type] : Type:={ dproj1: T; dproj2: (f dproj1)}.
```

Les enregistrements que l'on va construire sont une itération de ce type sur lui-même, à l'instar des listes qui sont une itération du type paire sur lui-même. De plus, on va étiqueter chaque champ, et fabriquer une fonction d'évaluation qui permettra d'accéder à un champ, via son étiquette.

Autrement dit, les enregistrements seront des *télescopes étiquetés*, c'est-à-dire des termes de type :

$$\Sigma_{n-1}[x_n : T_n^{a_n} \ ; \ x_{n-1} : (T_{n-1}^{a_{n-1}} x_n) \ ; \dots \ ; \ x_2 : (T_2^{a_2} x_n \dots x_3)](T_1^{a_1} x_n \dots x_2)$$

où Σ_n est un constructeur de type existentiel n-aire, où les T_i sont des fonctions à valeurs dans les types, étiquetées par des a_i (eux-mêmes indépendants des x_i) distincts 2 à 2.

En fait, on introduit le télescope vide, dont on note le type par $\emptyset_T$, qui est l'élément neutre pour l'opération d'extension de signatures. Les télescopes sont des types de la forme :

$$\Sigma_n[x_n : T_n^{a_n} \ ; \ \dots \ ; \ x_1 : (T_1^{a_1} x_n \dots x_2)]\emptyset_T$$

La condition que les "a_i" doivent être distincts 2 à 2 est nécessaire uniquement dans les propriétés "sémantiques" de la fonction d'accès aux champs. Toutes les opérations sur les enregistrements peuvent en fait être construites de manière indépendante de cette condition. On sépare donc ce qui relève de cette condition de ce qui n'en relève pas.

On va donc considérer que les télescopes étiquetés ne vérifient pas forcément cette condition. On appellera *enregistrements*, les télescopes étiquetés qui vérifient cette condition. La notion de télescope a été historiquement introduite dans [6] pour exprimer une notion de contextes relatifs (non "clos"). L'idée d'utiliser les télescopes pour exprimer la notion d'enregistrements extensibles avec champs dépendants en Coq vient de [11].

Les télescopes sont définis au travers de deux types : `sigtel`, le type des "signatures de télescope", et `impltel`, celui des "implantations de télescope". Ces types ont pour sorte `Type` car nos champs peuvent être dans `Set` ou dans `Prop` ou même dans `Type`.

4.2.1. Les signatures de têlescope

On définit le type `sigtel` par récurrence sur les listes, en utilisant deux types de base : `EsigT` le type à un élément qui représente le type du télescope vide, et `(FunIn a T)` le type étiqueté par `a` des fonctions à valeurs dans `T`.

```
Inductive EsigT: Type := Esig_: EsigT.
Structure FunIn[a:A; T:Type]: Type:= {dom: Type; fun:> (dom->T)}.
```

```
Fixpoint sigtel[l:(list A)]:Type :=
      Cases l of
         nil => EsigT
       | (cons a m) => (FunIn a (sigtel m))
       end.
```

4.2.2. Les implantations de télescope

On définit les implantations de télescopes avec la même méthode : en itérant sur les paires dépendantes :

```
Section DependentPair.

Variable T:Type.
Variable f:T->Type.

Structure dpairT[a:A]: Type:=
  { dprojT1: T; dprojT2: (f dprojT1) }.

End DependentPair.
```

```
Fixpoint impltel[l:(list A)]:
                         (sigtel l) -> Type :=
 <[l:(list A)](sigtel l)->Type>Cases l of
       nil => [_]UnitT
    | (cons a m) =>
        [s](dpairT
             [x:(dom s)](impltel (s x)) a)
 end.
```

4.2.3. Accès aux champs

Étant donné un télescope, on veut pouvoir accéder au champ correspondant à un nom d'étiquette. On définit deux fonctions, une pour accéder au type du champ, l'autre pour accéder à la valeur de ce champ. Pour ne pas avoir à gérer ici le fait que ces fonctions d'accès sont partielles (elles n'ont de sens que sur les noms d'étiquettes présents dans le télescope), on introduit un type `Dummy` à un constructeur `foo`, qui capture les cas où ces fonctions sont utilisées hors de leur domaine de définition.

```
Inductive Dummy: Type := foo: Dummy.

Hypothesis eqA_dec:(x, y:A)x=y+~x=y.

Fixpoint fieldsig[a:A; l:(list A)]:(s:(sigtel l))(impltel s)->Type :=
  <[l:(list A)](s:(sigtel l))(impltel s)->Type>Cases l of
      nil => [_;_]Dummy
    | (cons b m) => [s;i]
                  if (eqA_dec a b) then
                    [_](dom s)
                  else
                    [_](fieldsig a (dprojT2 i))
  end.

Fixpoint fieldimpl[a:A; l:(list A)]:(s:(sigtel l); i:(impltel s))(fieldsig a i):=
  <[l:(list A)](s:(sigtel l); i:(impltel s))(fieldsig a i)>Cases l of
      nil => [_;_]foo
    | (cons b m) =>
        [s;i]<[H:a=b+~a=b]if H then [_](dom s)
                                else [_](fieldsig a (dprojT2 i))>
                  if (eqA_dec a b) then
                    [_](dprojT1 i)
                  else
                    [_](fieldimpl a (dprojT2 i))
  end.
```

4.2.4. Passer des télescopes aux enregistrements

Les télescopes sont définis dans un cadre plus général que les enregistrements. Et l'utilisateur a à priori la possibilité de se servir des télescopes comme enregistrements. Le problème est qu'il peut alors écrire des spécifications qui sortiront de la "sémantique des enregistrements", et donc qui n'auront pas le sens que l'utilisateur leur prête, même si Coq leur donne un sens. On est donc amené à mettre des gardes afin de restreindre l'usage des télescopes. Ces gardes sont décidables (sous l'hypothèse de la décidabilité de l'égalité entre étiquettes), et l'utilisateur peut donc les faire décharger par le vérificateur de type de Coq.

Bien sûr, la vérification de ces gardes alourdit la vérification de la spécification de l'utilisateur, et ce dernier peut être tenté de les contourner...

Codage des gardes sur les listes Pour exprimer le fait que les gardes sont des informations non calculatoires de la théorie, mais plutôt de l'ordre de la vérification du respect de la "sémantique des enregistrements" par l'utilisateur, on les a placées dans `Prop`. Ceci dit, étant donné la décidabilité de ces gardes (modulo la décidabilité de l'égalité sur les étiquettes), on ne leur a assigné que deux valeurs de retour possible `True` ou `False`. Lorsque l'utilisateur veut décharger une garde, il lui suffit donc de passer la preuve `I`, qui est le constructeur du type `True`. Si la garde est fausse, le vérificateur de type de Coq lui indiquera un message d'erreur.

En fait, toutes les gardes utilisées dans cette théorie sont des instanciations bien choisies des deux prédicats `AllD` et `ExD` ci-dessous (le suffixe `D` dans leur nom signifiant "décidable").

```
Variable P:A->(list A)->Prop.
Hypothesis P_dec: (x:A; l:(list A))(P x l)+~(P x l).

Fixpoint AllD[l:(list A)]: Prop :=
   Cases l of
      nil => True
    | (cons a m) => if (P_dec a m) then [_](AllD m) else [_]False end.

Fixpoint ExD[l:(list A)]: Prop :=
   Cases l of
      nil => False
    | (cons a m) => if (P_dec a m) then [_]True else [_](ExD m) end.
```

Les enregistrements Les signatures d'enregistrement sont définies par :

```
Structure sign: Type := {
    sign_l: (list A);
    sign_l_df: sign_l^;
    sign_p:> (sigtel sign_l) }.
```

Les enregistrements (ci-dessous) sont donc vraiment des télescopes, pour lesquels on sait trouver une signature d'enregistrements :

```
Definition record[s:sign]:=(impltel s).
```

Le passage d'un télescope à un enregistrement n'est donc qu'une conversion de type. C'est particulièrement agréable pour construire des enregistrements : les constructeurs de télescopes suffisent !

Trivialement, les fonctions d'accès aux champs sont :

```
Definition fieldT:(s:sign)(record s)->A->Type
  :=[s;i;a](fieldsig eqA_dec a i).

Definition field:(s:sign; i:(record s); a:A; H:(a∈|s|))(fieldT i a)
  :=[s;i;a;H](fieldimpl eqA_dec a i).
```

5. Extension de signatures d'enregistrements

Les différentes notions présentées ci-dessous sont en fait implantées dans les télescopes. Pour ne pas entrer dans des détails techniques Coq qui n'ont pas leur place dans ce papier, on se contente ici de donner les spécifications de ces notions au niveau des enregistrements.

5.1. Relation de sous-signature et conversion entre enregistrements

Entre les signatures d'enregistrements, on définit une relation de sous-signature : cette relation caractérise le fait qu'on peut passer d'une signature à une autre en "oubliant" des champs. De façon interne, cette notion de sous-signature est définie par une succession de transformations élémentaires sur les signatures (oubli de champs, permutation de champs, ...) qui permet de construire des opérations par induction sur des "preuves" de sous-signature.

On a une fonction de conversion entre enregistrements associée à cette notion de sous-signature, qui préserve le comportement extensionnel des enregistrements (égalité des champs par accès sur les mêmes étiquettes).

Plus formellement, on suppose définie `eqT_dep`[1] l'égalité dépendante sur les types. On peut alors introduire le théorème `subsign_ext` :

```
subsign: sign -> sign -> Type.
coerce: (s1,s2:(sign))(subsign s1 s2)->(record s1)->(record s2).

Lemma subsign_cont:(s1,s2:sign)(subsign s1 s2)->|s1|⊃|s2|.

Lemma subsign_ext_T: (s1,s2:sign; H:(subsign s1 s2); i:(record s1); a:A)(a∈|s|)
       ->(fieldT i a)==(fieldT (coerce H i) a).

Theorem subsign_ext: (s1,s2:sign; H1:(subsign s1 s2); i:(record s1); a:A; H2:(a∈|s2|))
      (eqT_dep 2![T:Type]T (field i (cont_incl (subsign_cont H1) H2) [2])
                           (field (coerce H1 i) H2)).
```

Cette dernière propriété (démontrée en Coq) garantit la correction de la notion de sous-signature vis-à-vis de la sémantique qu'on lui prête. En particulier, on peut en déduire que quelles soient les "preuves" de sous-signature entre deux signatures, les fonctions de conversion associées au niveau des enregistrements sont équivalentes (vis-à-vis de l'accès aux champs).

La relation `subsign` est un préordre, auquel on peut facilement associer une relation d'équivalence, qui correspond à `subsign` sur les signatures de longueur égale.

```
(* Propriétés de préordre *)
Lemma subsign_refl:(s:sign)(subsign s s).

Lemma subsign_trans:(s1,s2,s3:sign)(subsign s1 s2)->(subsign s2 s3)->(subsign s1 s3).

(* Propriétés de l'équivalence associée *)
Lemma subsign_antisym:(s1,s2:sign)(subsign s1 s2)->(subsign s2 s1)
      ->(length |s1|)=(length |s2|).

Lemma subsign_sym:(s1,s2:sign)(subsign s1 s2)
      ->(length |s1|)=(length |s2|)->(subsign s2 s1).
```

1. Elle a comme définition en mode argument implicite :

```
Inductive eqT_dep[U:Type; P:U->Type; p:U; x:(P p)]: (q:U)(P q)->Prop :=
         eqT_dep_intro: (eqT_dep x x).
```

2. preuve de `a∈|s1|`

5.2. Construction de signature

Extension simple de signatures L'extension simple de signature consiste à ajouter un bout de signature à une signature donnée `s`. Mais les champs de ce bout de signature peuvent éventuellement dépendre des champs de `s`. En toute généralité, ce bout de signature doit donc être exprimé dans un contexte contenant les champs de `s` : c'est-à-dire un enregistrement de signature `s` (n'oublions pas que les télescopes ont été introduits pour formaliser la notion de contexte). Le bout de signature peut donc être exprimé sous la forme d'une fonction de type `(record s)->(sigtel l)`, où `l` est une liste d'étiquettes correspondant à la liste des méthodes ajoutées. L'extension simple a donc comme type :

```
Variable s:sign.
Variable l:(list A)).
Hypothesis H1:l̂.
Hypothesis H2:(l∩̸|s|).
```

```
Definition flatsign: ((record s)->(sigtel l))->sign.

Lemma flatsign_subsign:
  (f:(record s)->(sigtel l))(subsign (flatsign f) s).
```

Cela n'apparaît pas ici, mais par construction |`flatsign`| est la concaténation de |`s`| et de `l`. Comme cas particulier d'extension de signatures, on peut dériver, de l'extension simple, la concaténation de signatures :

```
Variables s1,s2:sign.
Hypothesis H:(|s2|∩̸|s1|).
```

```
Definition concatsign:=
 (flatsign (sign_l_df s2)[3] H [_:(record s1)](sign_p s2)).

Lemma subsign_concatsign_1: (subsign concatsign s1).
Lemma subsign_concatsign_2: (subsign concatsign s2).
```

Fusion de signatures La concaténation de signature précédemment définie ne peut s'appliquer que sur des signatures n'ayant aucune étiquette en commun. Au contraire, la fusion permet aux signatures d'avoir des étiquettes en commun. Mais elle impose que les champs portant le même nom dans les deux signatures soient compatibles.

Cette contrainte est ici exprimée comme l'existence d'une sous-signature (éventuellement vide) commune à ces deux signatures qui contient l'ensemble des noms de champs communs aux deux signatures. Cette contrainte étant non décidable, on demande à l'utilisateur de la décharger complètement. Ultérieurement, on peut envisager d'écrire une tactique (heuristique) qui réussit dans certains cas à décharger seule cette contrainte.

```
Variables s,s1,s2:sign.
Hypothesis H1:(subsign s1 s).
Hypothesis H2:(subsign s2 s).
Hypothesis H3:(|s2|\|s|)∩̸(|s1|\|s|).
```

```
Definition mergesign: sign.

Lemma subsign_mergesign_1: (subsign mergesign s1).
Lemma subsign_mergesign_2: (subsign mergesign s2).
```

Par construction |`mergesign`| est la concaténation de trois listes : |`s`|, (|`s1`|\|`s`|) et (|`s2`|\|`s`|).

6. Exemple d'utilisation de cette théorie

Dans notre exemple, on commence par définir le type des étiquettes. Pour avoir un ensemble dénombrable, on pourrait utiliser les entiers, mais ce ne serait pas lisible, ou utiliser les listes sur un ensemble fini (mots sur un alphabet). Pour rester simple, on se place plutôt sur un type fini énuméré, appelé `Label`, qui contient toutes les étiquettes que l'on va utiliser.

```
Inductive Label: Set := T: Label | EQ : Label | EQ_refl: Label | EQ_sym: Label
  | EQ_trans: Label | OP: Label | OP_EQ_l: Label (* ... *).

Lemma eqLabel_dec : (x,y:Label)x=y+~x=y.
B
  Intros x y; Case x; Case y; Auto; Right; Discriminate.
Defined.
```

3. preuve de $\widehat{|s2|}$

On commence par construire `setoid_sig`, la signature de setoïde, pour construire le type enregistrement `setoid`. En utilisant les grammaires extensibles de Coq, on peut donner une syntaxe concrète agréable à l'utilisateur. Cette syntaxe permet notamment de masquer les mécanismes de garde sur les listes évoqués précédemment : ci-dessous, dans l'exemple de `setoid_sig`, le système vérifie que la liste des étiquettes de `setoid_sig` est sans doublon, et B B déclenche une erreur de type en cas d'échec :

```
Definition setoid_sig:=(<eqLabel_dec>Sign
 T:Type;
 EQ: T->T->Prop;
 EQ_refl: (x:T)(EQ x x);
 EQ_sym: (x,y:T)(EQ x y)->(EQ y x);
 EQ_trans: (x,y,z:T)(EQ x y)->(EQ y z)->(EQ x z)).

Definition setoid:=(record setoid_sig).
```

Grâce à une fonction `Build` de notre théorie des enregistrements, on peut donner une fonction qui permet de construire un setoïde en donnant l'ensemble de ses champs dans l'ordre où ils apparaissent dans la signature :

```
Definition Build_setoid:=(Build setoid_sig).
```

On peut alors construire le setoïde des types muni de l'égalité sur les types :

```
Definition Type_setoid: setoid :=
  (Build_setoid (refl_eqT Type) (sym_eqT Type) (trans_eqT Type)).
```

On accède aux champs via l'opérateur "binaire" `@`, qui masque en fait la vérification que le champ auquel on cherche à accéder est bien dans l'enregistrement :

```
Definition carrier: setoid->Type:=[s](s @ T).
Coercion carrier: setoid >-> SORTCLASS.
```

En utilisant l'extension de signature, on peut construire le type enregistrement `semigroup` qui étend celui des setoïdes en ajoutant une opération binaire interne, compatible avec la relation d'équivalence et associative :

```
Definition semigroup_on:=[s:setoid](Sigtel
     OP:((s @ T)->(s @ T)->(s @ T)) ;
     OP_EQ_l: (x,y,z:(s @ T))((s @ EQ) x y)->((s @ EQ) (OP x z) (OP y z)) ;
     OP_EQ_r: (x,y,z:(s @ T)) ((s @ EQ) x y)->((s @ EQ) (OP z x) (OP z y)) ;
     OP_ass : (x,y,z:(s @ T))((s @ EQ) (OP (OP x y) z) (OP x (OP y z)))).

Definition semigroup_sig:=(Flat [s](semigroup_on s)).
Definition semigroup:=(record semigroup_sig).
```

On peut alors profiter des conversions de la théorie, pour les transformer en cœrcions Coq.

```
Definition sgp_setoid: semigroup -> setoid := (coerce (FlatSub [s](semigroup_on s))).

Coercion sgp_setoid: semigroup >-> setoid.
```

On pourra ensuite définir sur le même modèle que `semigroup`, `order` le type des setoïdes ordonnés puis faire une fusion entre `semigroup_sig` et `order_sig` (en amalgamant sur `setoid_sig`). Ceci nous permettra de créer un type enregistrement `ordsg` qu'on peut étendre afin de décrire les semigroupes ordonnés (il faut ajouter des hypothèses de compatibilité entre l'ordre et l'opération du semigroupe). Malheureusement, ces calculs de types sont trop gros pour l'implantation actuelle de Coq. Cela est dû au moins de façon partielle au fait que les `record` ne sont pas primitifs.

Avec une notion primitive d'enregistrement en Coq, il n'est pas sûr que l'on puisse exprimer les mécanismes de méthodes dérivées et de redéfinition de FocKer, alors que cela semble possible en s'appuyant sur la théorie des télescopes, comme on le présente dans la section suivante.

7. Perspectives

7.1. Extensions à la théorie Coq des télescopes

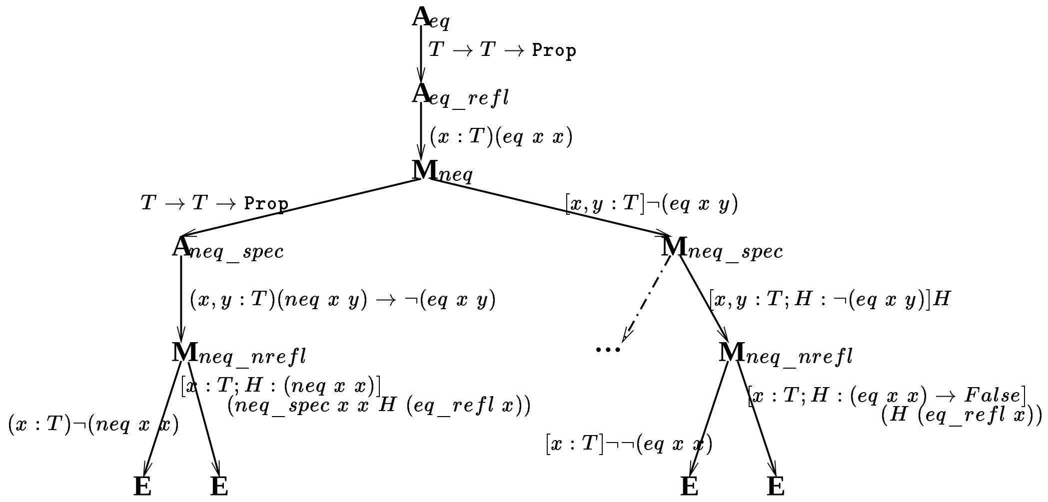
Renommage Le renommage de méthodes dans les enregistrements n'a pas encore été formalisé en Coq. Mais il ne semble pas que cela pose de grosses difficultés.

Méthodes dérivées et redéfinition Nous présentons ici les quelques idées qui sous-tendent le développement en Coq des mécanismes gérant les méthodes dérivées, à partir de la théorie des télescopes. Mais il n'y pas vraiment la place ici de rentrer dans les détails.

Pour représenter les espèces de FocKer, on introduit une nouveau type de donnée : les mixtels. Les mixtels sont des prédicats sur les signatures de télescope. Ainsi, intuitivement, une signature de télescope est aussi une signature de mixtel : les signatures de télescope reflètent la notion d'interface en FocKer. Les collections vont être vues comme des implantations de télescopes, ou comme des mixtels particuliers : des mixtels dont toutes les méthodes sont dérivées. On aura donc une correspondance entre les implantations de télescopes et les mixtels dont toutes les méthodes sont dérivées.

À la différence des signatures de télescopes qui ont une structure linéaire, les mixtels ont une structure arborescente avec deux types de nœuds internes : les nœuds dits "abstraits" qui représentent grosso-modo des méthodes primitives, et ceux dits "manifestes" qui représentent grosso-modo les méthodes dérivées. Les nœuds "abstraits" correspondent aux "nœuds" des signatures de télescopes : ils contiennent la même information, en particulier, ils n'ont qu'un seul fils. Les nœuds "manifestes" contiennent plusieurs informations : le nom de la méthode, son type, son implantation, et deux fils : un mixtel dépendant de manière opaque de cette méthode (comme dans les signatures de télescopes) et un mixtel dépendant de manière transparente de cette méthode (comme dans les implantations de télescopes).

Mixtel correspondant à l'espèce *setoid* de la section 2.3



A :nœuds abstraits ; **M** : nœuds manifestes ; **E** : feuilles (mixtel vide).

Dans un nœud manifeste, les deux fils représentent la même espèce : le fils gauche (opaque) représente cette espèce si on oublie (ou change) l'implantation associée au nœud, le fils droit (transparent) représente cette espèce lorsqu'on conserve cette implantation. Ainsi, on aura besoin d'une notion de mixtel bien formé pour garantir que les deux branches issues des nœuds manifestes représentent bien la même espèce.

Pour finir, précisons un peu les notions de méthodes primitive et dérivée dans un mixtel bien formé. Une méthode sera dérivée dans un mixtel bien formé lorsque dans la branche la plus à droite, le nœud portant son étiquette sera manifeste. Au contraire, une méthode sera primitive, lorsque dans cette branche, ce nœud sera abstrait. Dans ce cas, tous les nœuds du mixtel qui portent cette étiquette seront abstraits (sous l'hypothèse de bonne formation du mixtel). Lorsque toutes les méthodes d'un mixtel sont dérivées, on peut fabriquer directement une implantation de télescope : il suffit de prendre la branche la plus à droite.

La notion de mixtel est en cours d'implantation en Coq. Elle doit encore être étudiée pour éliminer les nombreuses redondances.

7.2. Extensions au système FocKer

De nombreuses extensions sont à envisager. Il faudrait par exemple, introduire une notion de récursivité sur les collections afin de pouvoir certifier l'exemple des polynômes récursifs donné dans [4]. Cette extension semble loin d'être triviale. Pour des raisons de place, on va s'attacher ici à décrire essentiellement une extension, qui semble plus immédiate mais tout aussi fondamentale, car elle étend le raffinement présenté ici.

En fait, on aimerait disposer d'une opération de prolongement de méthode, indépendamment de l'opération de redéfinition. Par exemple, on aimerait pouvoir exprimer l'exemple Ocaml suivant :

On suppose définie dans l'espèce des semigroupes, une méthode dérivée `iter` qui prend une entité du semigroupe, et un entier relatif strictement positif, et qui itère cette entité vis-à-vis de l'opération interne du semigroupe (supposée associative).

```
class virtual ['a] semigroup =
object (self)
  ...
  method virtual plus: 'a->'a->'a
  method iter =
    let rec iter_rec x n acc =
      if n=1 (* n > 0 *)
      then acc
      else (iter_rec x (n-1) (self#plus x acc))
    in (fun x n -> if n>0
                   then (iter_rec x n x)
                   else failwith  "False")
end
```

On veut ensuite étendre successivement le domaine d'application de `iter` sur les entiers relatifs, en réutilisant les définitions précédentes de `iter`.

On définit ainsi l'espèce des monoïdes et celle des groupes :

```
class virtual ['a] monoid =
object (self)
  inherit ['a] semigroup as dad
  method virtual zero: 'a
  method iter x n =
    if n>0
    then (dad#iter x n)
    else if n=0 then self#zero
               else failwith "False"
end

class virtual ['a] group =
object (self)
  inherit ['a] monoid as dad
  method virtual opp: 'a->'a
  method iter x n =
    if n<0
    then (self#opp (dad#iter x (-n)))
    else (dad#iter x n)
end
```

Dans le système FocKer, on peut exprimer cette extension successive de domaine. Il faudrait ajouter l'opération de prolongement. Ci-dessous on montre comment étendre le domaine d'application de `iter`. L'idée est de faire de la garde de `iter` une méthode `Giter` des espèces, qu'on va pouvoir redéfinir au cours des raffinements. Cette garde, de sorte `Prop`, disparaîtrait lors de la traduction vers Ocaml, suivant le mécanisme usuel d'extraction de Coq.

$$
\mathit{semigroup} := \left\{ \begin{array}{ll}
\texttt{decl} & T : \texttt{Type} \\
\texttt{decl} & \mathit{plus} : T \to T \to T \\
\texttt{decl} & \mathit{plus\_assoc} : \ldots \\
\ldots & \ldots \\
\texttt{def} & \mathit{Giter} : \mathit{int} \to \texttt{Prop} := \lambda[n : \mathit{int}] n > 0 \\
\texttt{def} & \mathit{Giter\_spec1} : \forall (n : \mathit{int})(n > 0) \to (\mathit{Giter}\ n) := \lambda[n; H]H \\
\texttt{def} & \mathit{iter} : T \to (n : \mathit{int})(\mathit{Giter}\ n) \to T := \ldots \\
\ldots & \ldots
\end{array} \right\}
$$

On a introduit la spécification `Giter_spec1`, qui permet de décharger la garde lorsqu'on utilise une collection raffinant un `semigroup` en paramètre d'une espèce ou d'une collection : en effet, on n'a alors pas accès à la définition de `Giter`. On remarque que l'implantation d'`iter` correspond au code Ocaml, ci-dessus. Elle dépendra de façon transparente de `Giter`. Ainsi redéfinir `Giter` impliquera l'obligation de redéfinir `iter`.

Pour les monoïdes, on aurait ainsi :

$$
monoid := \left\{ \begin{array}{ll}
\multicolumn{2}{l}{\texttt{refine } semigroup} \\
\ldots & \ldots \\
\texttt{def} & Giter : \ int \rightarrow \texttt{Prop} \ := \ \lambda[n : int] n \geq 0 \\
\texttt{def} & Giter\_spec1 : \ \forall (n : int)(n > 0) \rightarrow (Giter\ n) \ := \ \lambda[n; H].. \\
\texttt{def} & Giter\_spec2 : \ \forall (n : int)(n = 0) \rightarrow (Giter\ n) \ := \ \lambda[n; H].. \\
\texttt{def} & iter : T \rightarrow (n : int)(Giter\ n) \rightarrow T \ := \ \ldots \\
\ldots & \ldots
\end{array} \right\}
$$

8. Conclusion

Cet article présente un système de spécification de librairies de calcul formel, basé sur une méthodologie de description par raffinement. Cette démarche semble bien adaptée au calcul formel. En effet, les relations de dépendances entre spécifications sont établies et validées par les mathématiques depuis longtemps, tandis que le détail des algorithmes est lui en perpétuelle évolution. Dans ce cadre, le raffinement permet d'avoir un système ouvert, mais dont les fondations sont assurées par l'expérience des algébristes. En outre, la thèse de Bruno Barras (voir [2]) montre que le paradigme de spécification par raffinements successifs semble être une approche raisonnable dès qu'on a un logiciel "réel" (comme Coq) à certifier. Avoir une notion de raffinement dans le langage de spécification semble donc un moyen de rendre l'utilisation de ce paradigme plus systématique, et a un intérêt en dehors du calcul formel, comme le montrent les nombreuses applications de la méthode B [1].

Il faut terminer la formalisation de FocKer en Coq, porter l'effort sur l'efficacité de l'implantation, et approfondir les différentes extensions suggérées dans la section précédente. Ensuite, pour offrir un véritable système à l'utilisateur, il faudra fournir des mécanismes interactifs de constructions de preuves (ou plus généralement de termes) lors de la (re)définition de méthodes dérivées. Enfin, il faudra mettre en place un mécanisme de construction semi-automatique de code vers Ocaml. Ce mécanisme devra laisser la possibilité à l'utilisateur d'avoir des librairies Ocaml non certifiées : l'utilisateur devrait en effet pouvoir "jouer" avec ses programmes avant de les certifier (afin, par exemple de s'assurer de leur efficacité). Mais le système devra permettre à l'utilisateur d'être pleinement conscient de ce qui est certifié, et de ce qui ne l'est pas.

Il y a donc encore beaucoup de chemin à parcourir avant de parvenir à un système "fini".

9. Remerciements

Je remercie l'équipe du projet Foc, ce travail en est une émanation. Je remercie également les membres de CFC de leurs échanges fructueux. Je remercie enfin les rapporteurs de ce papier, pour leur lecture attentive, et les nombreuses corrections qu'ils m'ont suggérées.

Références

[1] J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[2] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de Doctorat, Université Paris 7, 1999.

[3] 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. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual.* Projet Coq, INRIA-Rocquencourt – CNRS-ENS Lyon, january 1999.

[4] S. Boulmé, T. Hardin, and R. Rioboo. Modules, objets et calcul formel. *Journées Francophones des Langages Applicatifs*, janvier 1999.

[5] J. Courant. *Un calcul de modules pour les systèmes de types purs.* Thèse de Doctorat, Ecole Normale Supérieure de Lyon, 1998.

[6] N. G. de Bruijn. Telescoping mapping in typed lambda calculus. *Information and Computation*, pages 189–204, April 1991.

[7] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types.* Thèse de doctorat, Université Paris-Sud, July 1999.

[8] X. Leroy. *The Objective Caml system, release 2.02.* Projet Cristal, INRIA-Rocquencourt, 1996.

[9] C. Oriat. *Étude des spécifications modulaires : constructions de colimites finies, diagrammes, isomorphismes.* Thèse de doctorat, Institut National Polytechnique de Grenoble, 1996.

[10] L. Pottier. *Les polynômes comme image par le foncteur adjoint du foncteur d'oubli des algèbres vers les ensembles (D'après D. Lazard).* Contribution Coq, sources disponibles sur http://pauillac.inria.fr/coq/contribs/algebra.html, 1998.

[11] L. Pottier. *Télescopes : des records généralisés définis dans Coq lui-même.* Communication personnelle, sources disponibles sur http://www-sop.inria.fr/croap/CFC/Tel/index.html, 1999.

[12] A. Saïbi. *Outils génériques de modélisation et de démonstration pour la formalisation des mathématiques en théorie des types. Application à la théorie des catégories.* Thèse de doctorat, Université de Paris 6, 1999.

[13] S. Watt, P. Broadbery, S. Dooley P. Iglio, S. Morrison, J. Steinbach, and R. Sutor. *AXIOM Library Compiler User Guide.* NAG Ltd, March 1995.