Réutilisation de preuves formelles : Une étude pour le système FoC

C. Dubois[1] & J. Grandguillot[2] & M. Jaume[3]

1: CEDRIC IIE-CNAM, Evry
dubois@iie.cnam.fr
2: CEDRIC IIE-CNAM, Evry
grandguillot@iie.cnam.fr
3: SPI LIP6, Université Paris 6,
Mathieu.Jaume@lip6.fr

Résumé

Le projet Foc est consacré au développement d'un environnement de programmation, dédié au calcul formel, permettant à la fois d'écrire des programmes et de prouver certaines propriétés qu'ils vérifient. Au sein de ce projet, un effort porte sur les aspects liés à la réutilisation de développements de preuves mécanisées de mathématiques. En effet, de nombreux développements existent et il est souhaitable de pouvoir les réutiliser dans le cadre de l'environnement Foc. Pour cela, il faut d'abord cerner ce que peut signifier le mot réutilisation. Une preuve s'appuie sur un énoncé, effectué dans un cadre logique précis, mais également sur certains choix de présentation des données. Par exemple, alors que les structures de Foc sont paramétrées par une relation d'équivalence, la majorité des développements que l'on peut trouver utilisent l'égalité syntaxique, qui est une forme plus contrainte d'égalité, sans toutefois utiliser dans les preuves les propriétés plus fortes que cette relation vérifie. S'agit-il alors de réutiliser uniquement la preuve d'une propriété sur une structure particulière ou bien peut-on envisager de généraliser une preuve afin qu'elle corresponde à la preuve cherchée? Nous présentons ici, au travers d'un exemple sur les groupes, le développement d'un outil permettant d'une part de généraliser une preuve en la paramétrant par une relation d'équivalence et un ensemble de propriétés que doit vérifier cette relation (il s'agit essentiellement de propriétés de congruence) et d'autre part de réutiliser cette preuve dans l'environnement Foc.

1. Introduction – Motivations

L'intérêt principal du calcul formel par rapport aux méthodes numériques est de donner des résultats exacts, sans erreur d'arrondi. Mais le calcul formel perd beaucoup de son intérêt si ces résultats ne sont pas fiables. En effet, comme tous les programmes, les programmes de calcul formel sont sujets à des erreurs d'implantation qui peuvent provenir soit d'une incorrection mathématique, soit d'une mauvaise interprétation de la sémantique. Pour assurer la correction des programmes de calcul formel il faut se tourner vers des méthodes fondées sur des techniques de preuves. L'environnement Foc propose d'intégrer les méthodes formelles au processus de développement des bibliothèques de programmes de calcul formel. Cet environnement fournit un langage construit sur Ocaml et Coq [4] pour spécifier, programmer et prouver des unités de bibliothèques de calcul formel. Ces unités sont traduites par le compilateur Foc vers du code source Ocaml pour aboutir à des programmes exécutables et vers du code source Coq afin de garantir les propriétés implantées.

L'environnement Foc propose un langage d'expression de propriétés qui peuvent être vérifiées soit à l'aide de Coq, soit à l'aide des outils de preuve propres à Foc. Ceux-ci sont en cours de développement et ont pour objectif de construire un terme Coq en s'appuyant sur la structure particulière des objets spécifiés dans l'environnement Foc. Les spécifications mathématiques sont issues de l'algèbre classique et les propriétés sont souvent des théorèmes bien connus. Certains d'entre eux ont déjà été formalisés dans un autre cadre, soit dans Coq, soit dans un autre système comme Mizar, PVS, Theorema, etc. Il serait évidemment intéressant de comprendre comment réutiliser toutes ces preuves dans le cadre de Foc. La proximité entre Foc et Coq suggère de commencer à étudier le vaste thème de la réutilisation par la réutilisation de preuves Coq dans Foc. En ce qui nous concerne, la réutilisation consiste donc à faire "communiquer" deux assistants à la preuve dans le but de faire bénéficier à l'un des preuves implantées par le second.

Cet article présente les problèmes posés par le développement d'une tactique de réutilisation de certaines preuves Coq dans le langage de preuve Foc. La méthode que nous esquissons ici pour parvenir à cet objectif a été implantée et utilisée avec succès sur un exemple (l'outil obtenu est encore à l'état de prototype). Notre étude s'est appuyée sur un exemple concernant les groupes : nous avons d'un côté une hiérarchie Foc de structures mathématiques aboutissant à la définition des groupes, et de l'autre une implantation Coq des groupes [3]. Il s'agit alors de fournir les outils nécessaires à la réutilisation d'un théorème du développement Coq existant pour prouver une propriété de la spécification Foc.

Nous allons voir que la principale difficulté provient du fait que les deux systèmes utilisent la plupart du temps des notions différentes de l'égalité. Le système Foc, orienté calcul formel, utilise exclusivement l'égalité des sétoïdes alors que le système Coq, laissant le choix de la spécification à l'utilisateur, incite celui-ci à utiliser l'égalité de Leibniz déjà implantée au cœur du système.

Le reste de l'article est organisé comme suit : nous présentons tout d'abord les systèmes Foc et Coq au travers de l'implantation des groupes dans ces systèmes, puis nous discutons des problèmes posés par la notion d'égalité dans ces deux formalisations et enfin nous montrons comment nous réutilisons dans l'environnement Foc les propriétés sur les groupes prouvées dans Coq.

2. Implantation des groupes dans les systèmes Coq et Foc

De manière classique, on peut définir un groupe additif comme un ensemble non vide muni d'une relation d'équivalence $\doteq$ et d'une loi de composition interne $\oplus$ associative, admettant un élément neutre 0, telle que $\doteq$ soit une congruence pour $\oplus$ et telle que chaque élément admette un opposé pour $\oplus$. C'est cette spécification des groupes qui est implémentée dans le système Foc. Afin de faire bénéficier à ce développement de résultats déjà prouvés dans le système Coq, nous allons réutiliser une formalisation des groupes, obtenue par P. Castéran [3], et qui établit qu'un groupe peut aussi être défini comme un ensemble E non vide muni d'une loi de composition interne $\oplus$ telle que pour tout x dans E, les deux opérations $f_x = (\lambda y : E.\ x \ \oplus\ y)$ et $g_x = (\lambda y : E.\ y \ \oplus\ x)$ soient surjectives.

2.1. Implantation des groupes dans le système FoC

Le projet Foc [8] a pour but de construire un environnement permettant à la fois de développer et de prouver des bibliothèques de calcul formel. Les structures mathématiques sont représentées par des *espèces* qui forment l'équivalent des classes en langage orienté-objet. L'implantation des bibliothèques peut être effectuée étape par étape grâce à la notion d'héritage qui relie les espèces entres elles. Le tableau suivant dresse un parallèle entre les notions usuelles du langage Foc et celles de l'algèbre et des langages orientés-objet :

ALGÈBRE	FOC	L.O.O.
structure	**espèce**	classe
élément	**entité**	élément d'un type atomique
loi, constante	**méthode**	méthode
domaine de définition	**signature**	type de méthode
ensemble, domaine	**collection**	objet

Les entités sont les éléments (les objets mathématiques) manipulés à l'intérieur des espèces. Le type support (carrier en anglais, introduit par le mot clé **rep**) d'une espèce est le type de ses entités. Chaque espèce contient un unique type support déclaré ou hérité. Les méthodes représentent les opérations de base ou les propriétés des structures mathématiques. Il y a quatre catégories de méthodes : le *type support* – les *signatures* représentant les opérations *déclarées* (introduction du nom et du type de l'opération) – les *méthodes* représentant les opérations *définies* (introduction du nom, du type et de la définition de l'opération) – les *propriétés* et les *théorèmes* vérifiés par l'espèce et impliquant des obligations de preuve. Une espèce définit une structure mathématique par un ensemble de méthodes. À l'intérieur d'une espèce, certaines méthodes peuvent être uniquement déclarées (donc non définies), ce qui permet de les manipuler sans connaître leur implantation réelle : l'espèce correspond alors à une classe virtuelle en langage orienté-objet. La notion d'héritage, proche de la notion d'héritage en programmation orienté-objet, permet de définir une espèce à partir d'autres espèces préalablement définies. Une espèce hérite de toutes les méthodes (dont le type support) de ses parents. Les collections permettent de coder des domaines mathématiques tels que les entiers, les anneaux de polynômes, etc. Les espèces permettent donc de définir une famille de collections comme les classes permettent de décrire une famille d'objets en programmation orienté-objet. Une collection est associée à une espèce complètement définie. Toutes ses méthodes doivent être définies et toutes ses obligations de preuve déchargées.

Nous présentons à présent une implantation des groupes dans l'environnement Foc sans entrer dans les détails de la syntaxe du langage Foc et des mécanismes mis en œuvre dans ce langage (pour plus de détail voir [16, 17]). Tout d'abord, nous définissons la notion de sétoïde (un ensemble non vide muni d'une relation d'équivalence) comme suit :

```
species setoide inherits basic_object =
sig equal in self->self->Prop;
sig element in self;
property eq_refl :  all x in self,  !equal(x,x);
property eq_sym :  all x,y in self,  !equal(x,y) -> !equal(y,x);
property eq_trans :  all x,y,z in self,
                    !equal(x,y) -> !equal(y,z) -> !equal(x,z);
end
```

Dans le langage Foc, `self` désigne la collection "courante" et `c!m` dénote la méthode `m` de la collection `c` (si celle-ci n'est pas précisée, il s'agit de `self`). L'espèce `setoide` hérite de la racine de la hiérarchie `basic_object`. Cette espèce contient trois méthodes : le type support **rep** et deux utilitaires `parse` et `print`. Le type support n'est que déclaré dans l'espèce `basic_object`, il sera instancié au fil de l'héritage entre espèces. La fonction `parse` de type `string->self` est elle aussi abstraite à ce stade, elle sera définie ultérieurement comme une fonction qui convertit une chaîne de caractères en un élément de la collection (`print` réalise la conversion inverse). L'espèce `setoide` introduit des méthodes qui lui sont propres : `equal` et `element` (qui sert à préciser que l'ensemble de base contient au moins un élément). D'autre part, on énonce les propriétés `eq_refl`, `eq_sym` et `eq_trans` qui traduisent que `equal` est une relation d'équivalence.

Nous définissons à présent la notion d' "ensemble 0-aire" qui hérite de la structure de sétoïde (l'élément `element` est alors défini comme étant l'élément déclaré `zero`), puis la structure de monoïde additif (un ensemble 0-aire muni d'une loi de composition interne `plus` associative pour laquelle `zero` est neutre et telle que la relation `equal` est une congruence pour cette opération).

```
species ensemble_zero_aire inherits setoide =
sig zero in self;
let element = self!zero;
end

species monoide_additif inherits ensemble_zero_aire =
sig plus in self-> self -> self;
property plus_congr :
     all x1,x2,y1,y2 in self
     !equal(x1,x2) -> !equal(y1,y2) -> !equal(!plus(x1,y1),!plus(x2,y2));
property plus_assoc : all x,y,z in self,
     !equal(!plus(x,!plus(y,z)),!plus(!plus(x,y),z));
```

```
property right_zero :  all x in self, !equal(!plus(x,!zero),x);
property left_zero :  all x in self, !equal(!plus(!zero,x),x);
end
```

Nous pouvons à présent définir la structure de goupe additif comme suit :

```
species groupe_additif inherits monoide_additif =
sig oppose in self -> self;
property right_opp :  all x in self, !equal(!plus(x,!oppose(x)),!zero);
property left_opp :  all x in self, !equal(!plus(!oppose(x),x),!zero);
theorem opp_unique :  all x,y in self,
       !equal(!plus(x,y),!zero) -> !equal(y,!oppose(x))
       proof : – voir plus loin –
theorem opp_invol :  all x in self,  !equal(x,!oppose(!oppose(x)))
       proof : – voir plus loin –
```

Un groupe additif est donc un monoïde additif muni d'une opération `oppose`. Les propriétés `right_opp` et `left_opp` spécifient les propriétés attendues de cette opération. Nous allons voir, qu'à partir de cette spécification des groupes, les deux théorèmes vont pouvoir être prouvés en réutilisant le développement Coq existant présenté dans la section suivante.

2.2. Implantation des groupes dans le système Coq

Le système Coq est un assistant à la démonstration qui peut être utilisé comme environnement de programmation certifié pour concevoir des logiciels sûrs, mais aussi comme outil de preuve pour élaborer et vérifier des preuves formelles. La théorie sous-jacente à cet outil est le calcul des constructions inductives. Nous présentons ici une formalisation des groupes dans le système Coq : il s'agit d'un développement réalisé par P. Castéran [3] que nous avons légèrement modifié (principalement en supprimant les dépendances entre sections et en modifiant le paramétrage de la structure de groupe) afin de satisfaire les contraintes imposées par l'outil de réutilisation que nous présentons par la suite.

Le système Coq dispose d'un mécanisme de sections qui permet de structurer les développements : lorsqu'une section est fermée, tous les termes définis dans cette section sont automatiquement paramétrés par les termes déclarés localement à la section par les mots clés `Variable` et `Hypothesis`. Ainsi chaque utilisation ultérieure du terme défini devra être instanciée par les variables qui étaient implicites à l'intérieur de la section.

Une première section, nommée `Groups`, contenant les variables suivantes :

```
Variable E : Set.         (* l'ensemble sous-jacent        *)
Variable o : E -> E -> E. (* la loi de composition interne *)
Variable e : E.           (* un élément du groupe          *)
Variable s : E -> E.      (* la fonction opposé            *)
```

permet de formaliser les groupes à l'aide d'une structure d'enregistrement comme suit :

```
Record group [p:E->E->E;a:E;f:E->E] : Set := mkgroup {
prf_gp_assoc     : (associative p);       (* preuve de l'associativité de p *)
prf_gp_l_neutral : (left_neutral p a);    (* propriétés de neutralité       *)
prf_gp_r_neutral : (right_neutral p a);
prf_gp_l_inverse : (left_inverse p a f);  (* propriétés de l'opposé         *)
prf_gp_r_inverse : (right_inverse p a f)}.
```

La variable `G` de type `(group o e s)` est alors introduite et les propriétés d'unicité de l'opposé d'un élément quelconque et d'involutivité de la fonction d'opposé sont prouvées.

```
Lemma s_unique : (x,y:E)((o x y)=e)->(y=(s x)).
Lemma s_involutive : (x:E)(x=(s (s x))).
```

Une deuxième section, nommée `Casteran_groups`, permet de formaliser la notion de groupe de manière différente (en utilisant les deux propriétés de surjectivité présentées plus haut). Tout d'abord les variables suivantes sont introduites :

```
Variable E : Set.         (* l'ensemble sous-jacent        *)
Variable o : E -> E -> E. (* la loi de composition interne *)
Variable a : E.           (* un élément du groupe          *)
Variable g : E -> E -> E.
Variable d : E -> E -> E.
```

La propriété :

```
Definition conto := [f:E->E;h:E->E](y:E)(y=(f (h y))).
```

est alors définie afin de pouvoir poser les hypothèses suivantes :

```
Hypothesis o_onto_r : (x:E)(conto [y:E](o x y) (d x)).
Hypothesis o_onto_l : (x:E)(conto [y:E](o y x) (g x)).
```

De plus on introduit l'hypothèse `o_assoc` exprimant l'associativité de la loi de composition interne `o`. Plusieurs définitions sont alors obtenues : preuves de l'existence d'un élément neutre "à droite", nommé `e`, preuve de la neutralité "à gauche" de cet élément `e`, définition d'une fonction `s` qui permet d'obtenir l'opposé d'un élément quelconque et preuves des propriétés de cette fonction. A partir de ces définitions, il est alors possible de prouver un théorème `E_is_a_group` exprimant le fait qu'à partir de `o`, `e` et `s` il est possible d'obtenir un groupe en utilisant la formalisation des groupes obtenue dans la section `Groups`.

3. Réutilisation

Alors que les développements effectués dans le système Foc reposent essentiellement sur la structure de sétoïde, les formalisations existantes en Coq utilisent principalement l'égalité syntaxique interne au système. À partir de l'objectif premier de réutilisation de preuves Coq pour Foc, nous allons développer un schéma de réutilisation tenant compte de ces différences de représentation et de raisonnement. Ce schéma est composé de deux parties distinctes (voir figure 3).

La première a pour but de transformer une preuve utilisant l'égalité de Leibniz en une preuve utilisant une égalité de sétoïde : il s'agit de la généralisation qui consiste en une traduction intermédiaire permettant au système Foc de réutiliser des preuves effectuées dans un contexte similaire au sien. Passer de l'égalité de Leibniz à celle des sétoïdes revient à généraliser l'égalité de Leibniz en une relation d'équivalence particulière qui valide la propriété de compatibilité de toutes les opérations sur son domaine. De nombreuses propriétés de compatibilité alors engendrées ne sont pas utilisées dans les preuves à réutiliser et l'étape de généralisation portera uniquement sur les propriétés $I_1, \cdots, I_n$ dont on a réellement besoin. Alors que ces propriétés correspondent à des instances de la propriété de Leibniz dans le développement initial, elles devront être prouvées lors de la généralisation.

La seconde partie du schéma a pour but d' "encapsuler" un terme de preuve Coq de manière à l'utiliser comme preuve dans le système Foc. Ceci se traduit par la définition d'une tactique de preuve Foc qui se fonde sur la compilation en Coq d'une spécification Foc.

3.1. Généralisation

3.1.1. Différentes approches de l'égalité

Dans les deux développements que nous considérons, nous sommes confrontés à deux approches différentes de l'égalité.

Egalité de Leibniz. L'égalité de Leibniz, aussi appelée égalité propositionnelle, est caractérisée par les deux axiomes suivants où $=_A$ dénote l'égalité propositionnelle sur le type A :

$$\forall A : \texttt{Set}, \forall x : A, x =_A x$$
$$\forall A : \texttt{Set}, \forall P : (A \rightarrow \texttt{Prop}), \forall x : A, \forall y : A, x =_A y \rightarrow (P\ x) \rightarrow (P\ y)$$

Si un prédicat P est vérifié en x, alors il l'est aussi en y si $x =_A y$. En Coq, l'égalité est définie de manière inductive comme la plus petite relation binaire réflexive :

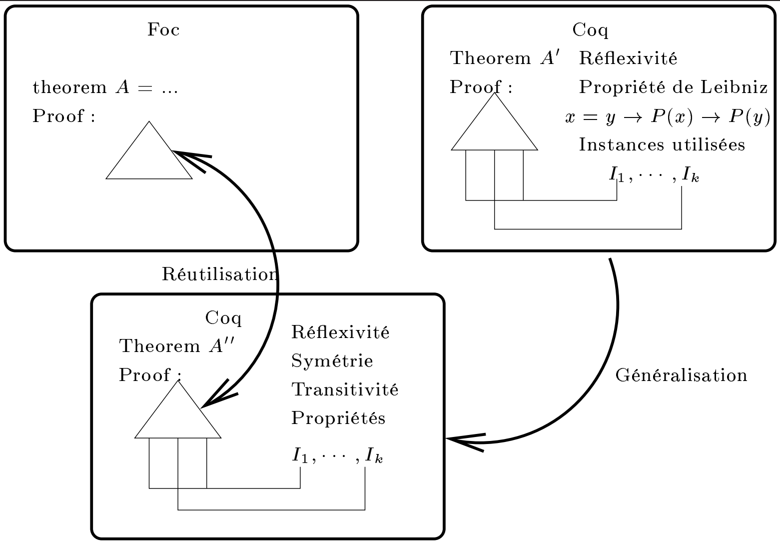

Figure 1: Schéma de réutilisation

```
Inductive eq [A:Set;x:A]: A -> Prop := refl_equal : (eq A x x).
```

Un schéma d'élimination `eq_ind` (spécialisé au cas des prédicats non dépendants) est automatiquement engendré à partir de cette définition :

```
eq_ind : (A:Set; x:A; P:(A->Prop))(P x)->(y:A)x=y->(P y)
```

où `x=y` est un racourci syntaxique de `(eq A x y)`. A partir de ce schéma d'élimination, on établit aisément que cette relation est symétrique et transitive. Une des principales propriétés de cette égalité est que toute fonction f de A dans B préserve `eq` :

```
f_equal:(A,B:Set)(f:A->B)(x,y:A) x=y -> (f x)=(f y)
```

Egalité sur les sétoïdes. Un sétoïde peut être représenté en Coq par un enregistrement contenant un type, une relation et les preuves de réflexivité, symétrie et transitivité de cette relation :

```
Record Setoid : Type :=
 { Carrier   : Type;
   Equal     : Carrier -> Carrier -> Prop;
   Prf_refl  : (Reflexive Carrier Equal);
   Prf_sym   : (Symmetric Carrier Equal);
   Prf_trans : (Transitive Carrier Equal) }
```

Si St_A (resp. St_B) est un sétoïde construit sur le type A (resp. B) et la relation d'équivalence eq_A (resp. eq_B), un morphisme de sétoïdes de St_A dans St_B est une fonction f de A dans B qui préserve l'égalité, c'est-à-dire qui vérifie la propriété de compatibilité :

$$(x, y : A)\ (eq_A\ x\ y) \to (eq_B\ (f\ x)\ (f\ y))$$

Avec l'égalité de Leibniz, toute fonction $f : A \to B$ préserve l'égalité à travers la propriété `f_equal`. Au contraire, un terme f de type $A \to B$ est un morphisme de sétoïdes si et seulement si il est accompagné d'une preuve de compatibilité `f_comp`, de type $(x, y : A)\ (eq_A\ x\ y) \to (eq_B\ (f\ x)\ (f\ y))$.

3.1.2. Un outil de réécriture sur les sétoïdes

Deux tactiques du système Coq sont directement concernées par l'égalité. La tactique `Rewrite` du système Coq permet de mettre en œuvre la propriété de Leibniz `eq_ind`. Etant donné un but courant à prouver dans un contexte dans lequel figure en hypothèse le terme $h : (y = x)$, l'application de la tactique `Rewrite` h va remplacer y par x dans le but courant. La tactique `Replace` t_1 `with` t_2 permet de remplacer les occurrences d'un terme t_1 par un terme t_2 dans un but et engendre un but supplémentaire qui consiste à prouver l'égalité $t_1 = t_2$. Lorsque la preuve en cours de construction met en jeu des sétoïdes et des morphismes de sétoïdes, il n'est plus possible d'utiliser ces tactiques : seules les propriétés de réflexivité, symétrie et transitivité qui définissent les relations d'égalité ainsi que les propriétés de compatibilité des morphismes (`f_comp`) peuvent être appliquées.

Dans [18], C. Renard propose un module d'extension Coq, appelé `Setoid`, qui permet de manipuler les égalités des sétoïdes avec de nouveaux outils qui se comportent de façon analogue aux outils de l'égalité du système Coq. A l'aide de ce développement, pour déclarer un sétoïde, il suffit d'utiliser la commande `Add Setoid` avec en paramètre les termes A de type `Set` ou `Type` (correspondant au type des éléments du sétoïde), eq_A de type A `->` A `-> Prop` (correspondant à la relation d'égalité considérée) et St_A de type sétoïde (contenant une preuve que la relation eq_A est bien une relation d'équivalence).

La définition d'un morphisme f dans le système Coq se fait en utilisant la commande `Add Morphism` f : *ident*. Coq passe alors en mode preuve et demande à l'utilisateur de prouver la propriété de compatibilité de f. Une fois la preuve terminée, le système enregistre f comme morphisme et *ident* comme propriété de compatibilité. L'algorithme qui engendre l'obligation de preuve relative à la propriété de compatibilité est décrit dans [18]. La seule contrainte imposée au terme f est de ne pas contenir de produit dépendant.

L'extension `Setoid` fournit alors deux tactiques de réécriture sur les sétoïdes `Setoid_rewrite` et `Setoid_replace`. Si eg est une égalité de sétoïde, la tactique `Setoid_rewrite` appliquée à un terme de la forme $(eg\ t_1\ t_2)$ remplace toute occurrence du terme t_1 par le terme t_2 à condition que dans le but courant toutes les opérations "traversées" soient des morphismes.

3.1.3. Paramétrage d'une preuve par une relation d'équivalence

Le mécanisme de généralisation que nous présentons ici impose certaines contraintes sur le développement formel à généraliser :

- Afin d'éviter une analyse de type, et pour pouvoir effectuer la généralisation sur les scripts de preuve, nous supposons que la généralisation ne concerne qu'une unique relation d'égalité. Plus précisément, nous faisons l'hypothèse que le symbole `=` dénote le même terme $(eq\ E)$ sur l'ensemble du développement. Ainsi il n'existe aucune ambiguïté de typage : toute égalité est une égalité sur le type E.
- Du fait de la contrainte sur les morphismes définis en utilisant l'extension `Setoid`, nous supposons que chaque opération destinée à être définie comme morphisme n'est pas exprimée sous forme de produit dépendant. De plus, lors de l'implantation du prototype, nous avons fixé arbitrairement le nombre maximum de morphismes à quatre (la plupart des implantations n'utilisent qu'un ou deux morphismes).
- Le fichier Coq ne doit pas contenir de sections dépendantes. En effet, la généralisation introduit de nombreuses variables implicites telles que la relation d'équivalence et les propriétés de compatibilité. Effectuer l'instanciation des termes dépendant à l'extérieur des sections augmenterait de manière considérable la complexité du programme de généralisation.

Le mécanisme de généralisation procède comme suit :

1. L'extension `Setoid.v` est importée en introduisant en tête du fichier Coq la commande suivante `Require Export Setoid`.
2. Une fois que l'ensemble **E** que l'on souhaite considérer comme un sétoïde a été déclaré, la relation d'équivalence sur cet ensemble est introduite :

```
Variable eg :  E -> E -> Prop.              la relation
Variable st :  (Setoid_Theory E eg).        les propriétés d'équivalence

Local eg_refl := (Seq_refl E eg st).        extraction de la réflexivité
Local eg_sym := (Seq_sym E eg st).          extraction de la symétrie
Local eg_trans := (Seq_trans E eg st).      extraction de la transitivité

Add Setoid E eg st.                         déclaration du sétoïde

Infix 2 "=S" eg.                            notation infixe = S
```

3. Chaque occurrence de l'égalité (de Leibniz) `=` dans le développement Coq est remplacée par une occurrence de l'égalité de sétoïde `=S`.
4. Pour définir une opération **o** du fichier Coq comme morphisme à partir de sa propriété de compatibilité **o_comp**, on introduit le code suivant :

```
Hypothesis :  congr_o :  o_comp.            la propriété de compatibilité

Add Morphism o :  morph_o.                  déclaration du morphisme
   Exact congr_o.
   Qed.
```

5. Les trois tactiques `Reflexivity`, `Symmetry` et `Transitivity H` sont remplacées, dans les scripts de preuve, par les applications des propriétés équivalentes :

   (`Apply eg_refl`), (`Apply eg_sym`), et (`Apply eg_trans with H`).

   De même, les occurrences (`Rewrite H`) et (`Replace m with n`) sont remplacées par les tactiques équivalentes de l'extension `Setoid` :

   (`Setoid_rewrite H`) et (`Setoid_replace m with n`).

Le prototype réalisé automatise la généralisation de l'égalité de Leibniz au sein d'un développement Coq. L'utilisation de cet outil est détaillée dans [9].

3.1.4. Application : l'exemple des groupes

La généralisation effectuée sur le code source des deux sections Coq présentées dans la section 2.2 s'obtient en spécifiant d'une part l'ensemble que l'on souhaite considérer comme un morphisme, c'est-à-dire `E`, et d'autre part la propriété de congruence qui va être utilisée dans les preuves. Ces deux fichiers, nommés `groups_morphism.txt` et `casteran_groups_morphism.txt` contiennent donc les deux informations suivantes :

```
o : E -> E -> E
(x1,x2,y1,y2:E)(x1 =S x2)->(y1 =S y2)->((o x1 y1)=S(o x2 y2))
```

La mise en œuvre de l'outil de généralisation permet alors d'obtenir les deux fichiers **`groups_setoid.v`** et **`casteran_groups_setoid.v`** qui correspondent à une formalisation des groupes basée sur la structure de sétoïde qui vont être "réutilisables" dans le contexte de l'environnement Foc.

3.2. Réutilisation d'une preuve Coq dans une spécification Foc

3.2.1. Preuves en Foc

Il existe deux moyens d'introduire une preuve en Foc : soit en définissant un théorème, soit en démontrant une propriété déclarée (dans l'espèce courante ou dans une espèce héritée). L'une des formes syntaxiques possibles est la suivante :

theorem T : *theoreme*
proof : [def : $def\_dep_1 \dots def\_dep_n$];
[decl : $decl\_dep_1 \dots decl\_dep_n$];
preuve;

Les termes def_dep_k désignent les noms des méthodes définies (directement dans l'espèce ou par héritage) et les termes $decl\_dep_k$ désignent les méthodes déclarées. Le mot-clé `decl` indique la preuve du théorème T dépend uniquement des types des méthodes $decl\_dep_1 \dots decl\_dep_n$. En revanche, elle dépend également de la définition des méthodes $def\_dep_1 \dots def\_dep_n$. Par conséquent toute redéfinition ultérieure d'une de ces méthodes imposera de reconstruire la preuve du théorème T. Ces indications de dépendances explicitent donc de manière claire le contexte de la preuve.

Un langage de preuve pour Foc dédié aux applications de calcul formel est en cours de développement, il permettra de décrire les preuves. L'exécution du script ainsi défini pourra par exemple produire un terme de preuve qui sera ensuite vérifié par le système Coq. Nous proposons un autre moyen d'associer une preuve à une propriété Foc, à savoir une tactique de preuve Foc qui a pour but d'*encapsuler* un terme de preuve Coq de manière à l'utiliser comme preuve dans le système Foc. Ceci se traduit par la définition d'une tactique de preuve Foc qui se fonde sur la compilation en Coq d'une spécification Foc. Pour réaliser la preuve, on souhaite pouvoir mettre en oeuvre une tactique de réutilisation faisant référence à un fichier Coq, à un terme de preuve Coq défini dans ce fichier et aux termes Foc def_dep_k et $decl\_dep_k$ introduits par les dépendances.

Dans la suite de cette partie, nous présentons rapidement les principes de la compilation vers Coq d'une spécification Foc. Pour une présentation plus formelle, on pourra consulter [17] ainsi que [9] pour l'analyse d'un exemple. Nous présentons ensuite notre tactique de réutilisation et en montrons l'effet sur l'exemple des groupes.

3.2.2. Compilation Foc vers Coq

Une espèce est compilée en un type enregistrement dont les champs dénotent l'ensemble des méthodes, que celles-ci soient définies ou déclarées dans l'espèce courante ou héritées. Dans les méthodes nous incluons les propriétés logiques et le type support. Une collection dont l'interface est une espèce s sera traduite en un enregistrement de type s.

Une méthode m définie dans une espèce s est compilée en un *générateur de méthode* gen_m, utilisable dans les espèces qui hériteront ultérieurement de l'espèce s. La fonction gen_m est une fonction générique paramétrée par les méthodes dont elle dépend (seules les dépendances de déclarations sont concernées ici, voir [17] et [2] pour plus de détails sur le calcul des dépendances).

Par exemple la méthode `element` définie dans l'espèce `ensemble_zero_aire` (voir la section 2.1) par :

```
let element = self!zero;
```

est compilée en :

```
Definition ensemble_zero_aire__element :
  (abst_T : Set) (abst_zero : abst_T) abst_T :=
  [abst_T : Set] [abst_zero : abst_T] abst_zero.
```

Ainsi chaque espèce qui hérite de l'espèce `ensemble_zero_aire` pourra utiliser la méthode `ensemble_zero_aire__element` en instanciant `abst_T` avec son propre type support et `abst_zero` avec son propre zéro. Cette technique permet d'implanter la liaison retardée.

Enfin des fonctions de coercion vers chacune des espèces héritées sont générées de manière à refléter les relations d'héritage spécifiées en Foc dans l'espèce s. Il s'agit alors de ne sélectionner dans un enregistrement de type s que les champs présents dans le type enregistrement correspondant à l'espèce héritée.

3.2.3. La tactique de réutilisation

Syntaxiquement, la tactique de réutilisation prend la forme

{* Reuse (*theoreme_Coq instances*). *}

et se place à l'endroit où la preuve est attendue. Par exemple :

proof of P =[def : $def\_dep_1 \dots def\_dep_n$];
[decl : $decl\_dep_1 \dots decl\_dep_n$];

{* Reuse ($theoreme\_Coq\ \ t_1\ t_2 \dots t_n$). *};

Le terme $theoreme\_Coq$ est le théorème Coq à réutiliser. Les t_i dénotent des termes Coq. Ce morceau de code Foc est compilé en un lemme Coq : le script généré commence par procéder à l'introduction des dépendances de déclarations et des paramètres puis applique la tactique `Exact` ($theoreme\_Coq\ \ t_1\ t_2 \dots t_n$).

Par exemple, la propriété `opp_invol` de l'espèce Foc `groupe_additif` est prouvée en utilisant la propriété `s_involutive` de la spécification Coq obtenue après l'étape de généralisation. Nous donnons ci-dessous le type du lemme à réutiliser :

$$
\begin{array}{l}
s\_involutive\ : \\
\quad (E : Set;\ eg : (E \to E \to Prop)) \\
\quad (Setoid\_Theory\ E\ eg) \\
\quad \to\ (o : (E \to E \to E))\ ((x1, x2, y1, y2 : E) \\
\quad\quad (eg\ x1\ x2) \to (eg\ y1\ y2) \to (eg\ (o\ x1\ y1)\ (o\ x2\ y2))) \\
\quad \to\ (e : E;\ s : (E \to E))\ (group\ E\ eg\ o\ e\ s) \\
\quad \to\ (x : E)\ (eg\ x\ (s\ (s\ x)))
\end{array}
$$

Côté Foc, nous écrivons donc :

```
theorem opp_invol :  all x in self,
                     !equal(x,!oppose(!oppose(x)))
proof :  decl :  equal plus zero oppose
                 eq_refl eq_sym eq_trans
                 plus_congr plus_assoc
                 right_zero left_zero right_opp left_opp;

{* Reuse (
s_involutive                              théorème Coq
                 abst_T                   type support E
                 abst_equal               relation eg
                 (Build_Setoid_Theory     sétoïde
                     abst_T
                     abst_equal
                     abst_eq_refl
                     abst_eq_sym
                     abst_eq_trans)
                 abst_plus                opération o
                 abst_plus_congr          propriété de compatibilité
                 abst_zero                élément neutre e
                 abst_oppose              inverse s
                 (mkgroup                 groupe
                     abst_T
                     abst_equal
                     abst_plus
                     abst_zero
                     abst_oppose
                     abst_plus_assoc
                     abst_left_zero
                     abst_right_zero
                     abst_left_opp
                     abst_right_opp)
).  *};
```

De la même façon, la propriété `opp_unique` de l'espèce Foc `groupe_additif` est prouvée en utilisant la propriété `s_unique` de la spécification Coq obtenue après l'étape de généralisation. Les termes Coq paramètres de la tactique `Reuse` sont les mêmes que précédemment. D'autre part, toujours en utilisant la même technique, quatre des résultats prouvés sur les groupes définis dans la section `Casteran_groups` ont été réutilisés afin d'obtenir des preuves pour une spécification des groupes dans le système Foc mettant en jeu les deux propriétés de surjectivité présentées plus haut.

Les termes $t_1\ t_2 \ldots t_n$ mentionnés dans la spécification Foc (dans l'exemple `abst_T`, `abst_equal` ...) font référence directement aux objets manipulés par la spécification Coq obtenue par compilation. Pour utiliser cette tactique il faut donc connaître assez précisément cette traduction ou du moins quelques correspondances simples à établir et quasi-syntaxiques que nous résumons dans le tableau ci-dessous :

Correspondances

termes Foc	instances de la tactique
`self`	`abst_T`
def_dep_k	`abst_`def_dep_k
$decl\_dep_k$	`abst_`$decl\_dep_k$

Il est clair que ce procédé n'est guère acceptable : d'une certaine manière, le code source fait référence au code objet. Il s'agit ici d'une première expérience qu'il convient d'approfondir. Une des perspectives de ce travail est de proposer une syntaxe qui permettrait de relier les paramètres du théorème Coq à instancier avec les objets de la spécification Foc. Néanmoins la correspondance n'est pas toujours aisée à exprimer car elle peut nécessiter de mixer les constructions des deux mondes, c'est par exemple le cas du troisième argument de l'exemple précédent.

La tactique `Reuse` a été incorporée au langage Foc. On pourra consulter [9] pour quelques détails sur le prototype réalisé.

4. Conclusion – Perspectives

L'étude de réutilisation de preuves que nous avons réalisée dans le cadre du projet Foc a permis de fournir deux outils de preuve autonomes. Le premier est un programme de généralisation de développements Coq automatisant un changement d'égalité. Le second est une tactique de preuve Foc réutilisant des termes de preuve Coq. La combinaison de ces deux outils nous a permis de réutiliser des preuves Coq, comme celles implantées par Pierre Castéran dans son exercice sur les groupes (distribuées avec le système Coq), dans des développements Foc.

Formaliser une théorie, c'est décrire dans un langage formel les objets de cette théorie, exprimer, dans ce même langage, les propriétés que ces objets peuvent vérifier et prouver certaines d'entre elles, souvent à l'aide d'un langage de tactiques. Il s'agit donc du développement d'un logiciel et la problématique de la réutilisation de développements formels est naturellement proche de celle de la réutilisation de bibliothèques de programmes. On retrouve donc les notions classiques de modularité, de généricité, d'abstraction ... qui sont des critères importants de réutilisabilité. Toutefois, réutiliser un développement formel existant soulève des difficultés propres à ce type de développement. En effet, tandis que les spécifications formelles des définitions des objets et des propriétés sont généralement facilement réutilisables, la réutilisation des preuves est par contre plus délicate. Ceci est principalement dû au fait, qu'au niveau du développeur, les preuves sont seulement exprimées via une succession de tactiques qui indiquent comment les construire. La complexité et la taille d'une preuve ne permettent pas d'envisager humainement l'écriture de la preuve elle-même qui ne figure donc pas dans le code source du développement. La manipulation (parcours, recherche, modification, transformation ...) d'une preuve est alors difficile si l'on reste au niveau du langage de tactiques et peu d'outils existent encore (citons toutefois [15]). Cependant, l'accès à la structure d'une preuve, au niveau de l'utilisateur, est rarement aisée et nécessite une connaissance des représentations internes des preuves, donc du code source de l'outil de développement formel utilisé. Nous avons donc choisi ici de développer un outil de réutilisation qui ne porte que sur le code source d'un développement effectué dans Coq. Evidemment, en simplifiant l'utilisation de notre outil, cette approche impose quelques contraintes sur le code source à réutiliser (par exemple, chaque occurrence du symbole d'égalité `=` doit être remplacée "à la main" par `eq T` où `T` est le type sur lequel porte l'égalité – cette information de type se trouve bien sûr dans le terme de preuve interne à Coq) mais reste totalement effective.

D'autre part, une difficulté supplémentaire provient du fait que, comme tout logiciel, un tel outil évolue avec le temps et il est possible que les représentations internes utilisées évoluent elles aussi. Il est à noter que le passage d'une version à une autre d'un outil de développement formel pose aussi le problème de la maintenance des preuves existantes. En effet, si la sémantique des constructions d'un langage de programmation reste, dans la plupart des cas, inchangée dans les versions successives d'un compilateur, cette stabilité ne se retrouve pas pour les langages de tactiques : les constructions produites par certaines tactiques évoluent souvent avec l'outil.

La réutilisation de développements formels pose des problèmes difficiles – tant du point de vue théorique que du point de vue technique – et les perspectives de ce travail sont nombreuses.

Concernant l'outil présenté dans cet article, et au delà de l'aspect réutilisation, un tel outil devrait également permettre d'apporter une information sur la preuve réutilisée, en fournissant les hypothèses sur la relation d'égalité qui sont réellement utilisées.

D'autre part, pour utiliser cet outil, une connaissance approfondie de la traduction Coq réalisée par le compilateur Foc est requise de la part de l'utilisateur. Une intégration totale de la tactique de réutilisation dans le futur langage de preuve de Foc pourrait permettre à l'utilisateur de travailler exclusivement sur des termes du langage Foc tout en lui masquant la totalité du travail de traduction. Cette voie nécessiterait un langage permettant de mettre en correspondance des objets de la spécification Foc avec des objets de la spécification Coq à réutiliser et demanderait un travail de compilation plus important. Ceci est la condition nécessaire pour offrir à l'utilisateur un réel outil de réutilisation.

À plus long terme, il serait intéressant d'étudier la réutilisation de manière plus approfondie. Plusieurs directions sont possibles :

- Plusieurs représentations équivalentes d'un même objet peuvent coexister et les problèmes de changement de représentation méritent d'être étudiés tant d'un point de vue pratique, comme dans [12], que d'un point de vue théorique, comme dans [1].
- Abstraire un concept dans une preuve pour l'instancier de manière différente – comme nous l'avons fait pour la notion d'égalité – nécessite une étude plus poussée et une comparaison avec d'autres travaux comme par exemple le développement présenté dans [11].
- Un piste plus ambitieuse concerne la "communication" entre divers outils de développements formels. Ici encore, des problèmes théoriques se posent (il s'agit de faire "coïncider les logiques mises en jeu") [10, 13] et la mise en pratique pose également des problèmes "techniques" [5, 14, 7, 6].

Bibliographie

[1] G. Barthe and O. Pons. Type isomorphisms and proof reuse in dependent type theory. In F. Honsell and M. Miculan, editors, *Proc. of 4th Int. Conf. on Found. of Software Science and Computation Structures, FoSSaCS'01*, volume 2030 of *LNCS*, pages 57–71. Springer-Verlag, 2001.

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

[3] P. Castéran. *An exercice on groups*, 1997. `http://Coq.inria.fr/contribs/groups.tar.gz`.

[4] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.2*. INRIA-Rocquencourt, December 2001.

[5] D. J. Howe. Importing mathematics from HOL into nuprl. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Ninth International Conference on Theorem Proving in Higher Order Logics TPHOLs'96*, volume 1125 of *LNCS*, pages 267–282. Springer Verlag, 1996.

[6] E. Denney. A prototype proof translator from HOL to Coq. In M. Aagaard and J. Harrison, editors, *13th International Conference on Theorem Proving in Higher Order Logics TPHOLs'2000*, volume 1869 of *LNCS*, pages 108–125. Springer-Verlag, 2000.

[7] A. Felty and D. Howe. Hybrid interactive theorem proving using Nuprl in HOL. In W. McCune, editor, *14th International Conference on Automated Deduction, CADE'14*, volume 1249 of *LNAI*, pages 351–365. Springer Verlag, 1997.

[8] Groupe Foc. *Présentation du projet Foc.* `http://www-calfor.lip6.fr/~foc`.

[9] J. Grandguillot. *Réutilisation de preuves : une étude pour le système Foc.* Mémoire de DEA, IIE-CNAM – Université d'Evry, 2002.

[10] D. J. Howe. Semantics foundations for embedding HOL in Nuprl. In M. Wirsing and M. Nivat, editors, *Proceedings of the Algebraic Methodology and Software Technology Conference AMAST'96*, volume 1101 of *LNCS*, pages 85–101. Springer-Verlag, 1996.

[11] T. Kolbe and C. Walther. Proof analysis, generalization, and reuse. In W. Bibel and P. H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume II, Systems and Implementation Techniques.* Kluwer Academic Publishers, 1998.

[12] N. Magaud and Y. Bertot. Changing data structures in type theory:a study of natural numbers. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs TYPES'2000*, volume 2277 of *LNCS*, pages 181–196. Springer-Verlag, 2000.

[13] P. Naumov. Formalization of isabelle meta logic in nuPRL. Technical Report TR99-1769, Cornell University, Computer Science, 1999.

[14] P. Naumov. Importing isabelle formal mathematics into nuPRL. Technical Report TR99-1734, Cornell University, Computer Science, 1999.

[15] O. Pons. Ingénierie de la preuve. In INRIA, editor, *Onzièmes Journées Francophones des Langages Applicatifs, JFLA'2000*, 2000.

[16] V. Prevosto and D. Doligez. *Inheritance of Algorithms and Proofs in the Computer Algebra Library FoC*, 2002. To appear in Journal of Automated Reasoning.

[17] V. Prevosto, D. Doligez, and Th. Hardin. Algebraic structure and dependent records. In S. Tahar C. Munoz and V. Carreno, editors, *Proceedings of TPHOLs 02*. Springer-Verlag, August 2002.

[18] C. Renard. *Un peu d'extensionnalité en Coq - une tactique pour les sétoïdes.* Mémoire de DEA, Université Paris 7, 2001.