Rapport de stage de L^3

Rémi Nollet

4 juin 2012 – 27 juillet 2012

1 Apercu

Le but du projet *moca* (pour « Caml modulo ») est de quotienter des types de donnée *OCaml* par des relations d'équivalence. Par exemple considérer que l'ordre des éléments dans une liste n'a pas d'importance. On souhaite alors ne manipuler que des éléments d'un système particulier de représentants pour cette relation. Dans notre exemple, le but est de ne manipuler que des listes triées. Afin de garantir cet invariant, pour des questions de sécurité, la construction du type sera alors cachée à l'utilisateur (grâce à la commande `private` de *OCaml*) : ce dernier a toujours la possibilité d'observer le type de donnée par filtrage ; il faut en revanche lui fournir, en remplacement des constructeurs du type, des fonctions de construction de ce type. Dans notre exemple : un élément représentant la liste vide et une fonction permettant d'ajouter un élément à une liste, *tout en conservant l'invariant.*

À partir d'une syntaxe enrichie des définitions de type de *OCaml*, *moca* produit automatiquement le code de ces fonctions de construction.

Mon rôle dans ce stage est de prouver en *Coq* la correction du code produit par *moca* pour une structure très particulière : les groupes commutatifs libres ; ce code étant produit par la déclaration moca suivante :

```
type 'a t = private
   | Zero
   | Gen of 'a
   | Opp of 'a t
   | Add of 'a t * 'a t
       begin
         associative
         commutative
         neutral left (Zero)
         neutral right (Zero)
```

```
    opposite (Opp)
end
```

2 Environnement

J'effectuai ce stage à l'INRIA [1] [2] Rocquencourt, au sein de l'équipe-projet Pomdapi [3], qui s'intéresse à la construction et à l'analyse d'outils de simulation pour la modélisation de problèmes environnementaux et énergétiques.

Mes maîtres de stages furent Pierre Weis (équipe-projet Pomdapi, entre autres) et Frédéric Blanqui (équipe FORMES, située à Pékin).

Merci à eux. [4]

1. Institut national de recherche en informatique et en automatique
2. Chez Inria ! Pardon ; il faut dire Chez Inria.
3. http://www.inria.fr/equipes/pomdapi
4. Ainsi qu'à tous les membres de l'INRIA — d'Inria ! Pardon. — qui m'ont conseillé, soutenu et / ou distrait dans mon travail.

Table des matières

Première partie
Rapport

3 Rappels — prérequis

Nous supposons que le lecteur est déjà familier avec le paradigme de la programmation fonctionnelle, les notions de récursivité, terminaison d'une fonction, les types de donnée récursifs, *etc.* [5]

Cette section contient quelques explications sur le langage *OCaml* et le système *Coq*. Ceux des lecteurs qui sont déjà au courant les passeront sans problème.

3.1 *OCaml*

OCaml est un langage de programmation fonctionnelle développé depuis environ 1984 par les équipes Formel puis Cristal et, actuellement, Gallium de l'INRIA. [6]

Deux traits de *OCaml* sont particulièrement intéressants dans ce rapport.

1. Les filtrages de *OCaml* offrent la possibilité d'associer à un motif une condition signalée par le mot-clef `when`, qui doit être vérifiée pour que la clause soit sélectionnée. Par exemple dans un filtrage, la clause

   ```
   (* ... *)
   | (x, Add (y, z)) when x = y -> (* code *)
   (* ... *)
   ```

   ne sera sélectionné que sur un motif de la forme `(x, Add (x, z))`.
2. Lors de la déclaration d'un type récursif (type somme), il est possible de le rendre *privé* (grâce à la commande `private`. De la sorte, si ce type est exporté pour être utilisé dans un autre module que celui où il a été déclaré, l'utilisateur n'aura plus accés aux constructeurs du type. Il pourra toujours pratiquer le filtrage de motifs sur ses éléments, mais pour construire des valeurs de ce type, il sera limité aux fonctions définies dans le même module que le type.

Pour plus de précisions concernant ces deux points, ou d'autres, du langage *OCaml*, consultez son manuel [5].

Le but de *OCaml* est, entre autres, de permettre le développement de l'assistant à la preuve *Coq*.

5. J'aurais volontier fait autrement, mais ce rapport ne pouvait pas dépasser 20 pages.
6. Ou d'Inria. Bref!...

3.2 *Coq*

Est un logiciel qui permet l'écriture de preuves mathématiques de manière interactive. C'est-à-dire que je commence par dire à *Coq* ce que je veux prouver puis, grâce à un ensemble de *tactiques*, je fais avancer la preuve pendant que *Coq* me signale s'il est d'accord et ce qu'il me reste à prouver. *Coq* possède deux moyens puissants pour automatiser ses preuves :

- Un langage de programmation interne, fonctionnel, nommé *Gallina*, qui permet de faire des preuves sous forme de calculs. [7]
- Un langage d'écriture de tactiques, *Ltac*, qui permet de les combiner et de les appliquer automatiquement.

L'essentiel des renseignements sur *Coq* peut être trouvé dans le manuel [6]. J'ai aussi beaucoup compulsé, lors de mon stage, le *Coq'art* [1], accessible au débutant complet et dans lequel, mettons, un étudiant en stage de fin de licence d'informatique fondamentale pourra encore apprendre beaucoup de choses...

4 Contexte

4.1 Concernant *moca*

Pour aller un peu plus loin que le résumé de la section 1 : un des buts recherchés par les fonctions de construction est la coïncidence, dans le type `t`, de la relation définie par l'utilisateur avec l'égalité syntaxique. C'est-à-dire que si, prenant comme exemple le code sur lequel j'ai travaillé, si donc le constructeur `Add : (t * t) -> t` est déclaré commutatif et associatif, alors les termes produits par

```
add (gen 3, add (gen 5, gen (-1)))
```

et

```
add (add (gen (-1), gen 5), gen 3)
```

doivent être syntaxiquement égaux. En l'occurrence ils vaudront tous les deux `Add (Gen (-1), Add (Gen 3, Gen 5))`, qui est appelé la *forme normale* de ces termes. Cette propriété est appelée *complétude* des fonctions de construction. Ceci permet notamment de décider si deux termes sont congrus modulo la relation induite par la déclaration *moca*.

7. D'ailleurs en réalité, lorsqu'on utilise des tactiques pour écrire une preuve, *Coq* construit un programme *Gallina* qui représente la preuve.

La deuxième chose essentielle à prouver sur les fonctions produites par *moca* est leur *correction*. C'est, pour reprendre le même exemple, le fait que le terme construit par `add (add (gen (-1), gen 5), gen 3)` sera bel et bien congru, modulo la relation voulue, au terme `Add (Add (Gen (-1), Gen 5), Gen 3)`.

Si vous voulez en savoir plus sur l'utilisation de *moca*, référez-vous au manuel — très clair — présent sur la toile [3]. Vous pourrez également y trouver les sources du programme ainsi que l'article [2] qui en détaille le fonctionnement et les fondements théoriques.

Pour comprendre mon rôle là-dedans, il faut aussi comprendre ceci : à terme, le but est que *moca* produise, en plus des fonctions de constructions attendues, en *OCaml*, les preuves *Coq* de terminaison et de correction de ces fonctions. Actuellement, nous en sommes au stade où nous essayons de voir, sur des exemples particuliers, comment ces preuves doivent être construites, de manière à pouvoir généraliser ensuite ce travail.

Frédéric Blanqui avait déjà écrit des preuves *Coq* pour des déclarations correspondant aux théories des monoïdes et des groupes non commutatifs (dans des fichiers `monoid.v` et `group.v`) et il me fallait écrire celle concernant les groupes commutatifs (donc `group_commutative.v`).

4.2 Concernant mon stage

Du moment que j'eus vraiment réalisé que le but de ces preuves est d'être, un jour, productible automatiquement, j'ai toujours privilégié une plus grande généralité, et réutilisabilité et une plus grande facilité de génération automatique du code à d'autres considérations telles que la taille du code des preuves ou des termes de preuves qu'elles engendrent.

Le code *OCaml* à partir duquel j'ai travaillé se trouve en annexe C. Le code *Coq* que j'ai écrit est trop volumineux pour avoir subi le même traitement...

4.3 Concernant ce rapport

J'essaierai de m'attacher, dans le présent rapport, aux principaux points de la preuve, à ceux sur lesquels j'ai le plus travaillé et réfléchi, à ceux, enfin, qui ne nécessitent pas d'entrer dans les détails de *Coq* et de commenter du code ligne à ligne.

Amoureux de *OCaml*, j'ai écris ce rapport dans un souci de concision, de clarté et de simplicité. Cependant j'ai aussi tâché de le rendre accessible aux non-spécialistes, d'où les quelques explications de la section 3, et les diverses

notes explicatives qui parsèment le rapport. Ceux des lecteurs qui savent déjà tout cela pourront les passer en rigolant.

5 Encodage des fonctions *OCaml*

Le but de mon stage est de prendre les fonctions *OCaml* présentes dans `group_commutative.ml` et de prouver, dans *Coq*, des théorèmes les concernant. Si je veux prouver un théorème sur une fonction *dans* Coq, je dois pouvoir raisonner sur cette fonction *dans* Coq, il faut donc que j'ai à disposition dans *Coq* un objet qui représente cette fonction.

Par la suite, il me sera également très utile d'avoir à disposition un principe de réurrence adapté à la fonction, qui permette de raisonner sur cette fonction.

5.1 Première idée (naïve ?) : recopier les fonctions dans *Gallina*

Coq possède un langage d'écriture de preuves, à l'aide de tactiques, mais possède également un langage de programmation, qui permet d'écrire de véritables programmes informatiques, dans une syntaxe assez proche de celle de *OCaml*. De plus *Coq* possède un mécanisme d'extraction qui permet, à partir d'un programme écrit dans *Gallina*, d'obtenir le même programme écrit, cette fois, en *OCaml*[8].

L'idée, plutôt séduisante, était donc la suivante : ces fonctions *OCaml*, sur lesquelles je dois travailler, je les recopie dans *Gallina*, en me débrouillant pour que, quand j'utilise le mécanisme d'extraction, j'obtiennes à nouveau mes fonctions *OCaml*. L'avantage étant alors que je suis *sûr* que les fonctions sur lesquelles j'aurai prouvé des théorèmes dans *Coq* sont *exactement* les fonctions *OCaml* sur lesquelles je suis censé travailler.

En outre lorsqu'une fonction est ainsi écrite dans *Gallina*, il existe plusieurs manières, dans *Coq*, d'obtenir automatiquement le principe de récurrence adapté à cette fonction.

C'est donc une idée très attirante. Évidemment elle ne fonctionne pas — puisque j'en parle.

D'abord, en effet, plusieurs traits distinguent *OCaml* et *Gallina*, empêchant la translittération immédiate des fonctions :

1. Les clauses *when*, utilisées dans les fonctions *OCaml* produites par *moca*, n'ont pas d'équivalent en *Gallina*.

8. L'inverse n'est, hélas, pas faisable de manière aussi automatique.

2. *Coq* n'autorise la définition d'une fonction récursive que si ses appels récursifs font décroître structurellement un des arguments, ce qui n'est pas toujours le cas pour les fonctions produites par *moca*.

Voyons comment il est possible de parer à ceci, tout en conservant la même approche d'encodage des fonctions.

5.1.1 Élimination brutale des clauses *when*

Considérons un filtrage à la M.L. tel qu'en contiennent les fonctions produites par *moca* et dont un bon exemple est précisément le code de la fonction `add`, dans l'annexe C.2.

Et considérons, dans un tel filtrage, une clause particulière :

```
match x with
  début du filtrage
  | motif when condition -> résultat
  suite du filtrage
```

Ce `when` est éliminable en même temps que ceux qui suivent de la manière suivante :

1. Éliminer récursivement les `when` dans la *suite du filtrage*.
2. Remplacer cette clause par :

```
| motif ->
    if condition then résultat
    else begin match x with
      suite du filtrage
      end
```

En démarrant ce processus au premier `when` du filtrage, on les élimine tous. [9] [10]

L'inconvénient est que, puisque l'élimination d'un seul `when` duplique la suite du filtrage, ce processus augmente la taille du code de manière exponentielle. D'un point de vue théorique ce n'est peut-être pas si grave, puisqu'à

9. Notez bien que la *suite du filtrage*, traitée récursivement, est ainsi dupliquée, puisqu'elle apparaît maintenant dans la clause considérée en plus de constituer la suite effective du filtrage, après cette clause.

10. Notez également que ce processus n'est utile que si la dernière clause du filtrage accepte *tous* les motifs — ce qui est le cas pour les filtrages engendrés par *moca*. En effet, dans le cas contraire, le filtrage ajouté à la clause, et dont le corps est *suite du filtrage*, peut n'être pas exhaustif ; ceci provoquerait un avertissement de la part de *OCaml*, ce qui n'est pas si grave, et un refus de la part de *Coq*, ce qui l'est déjà plus.

terme ces preuves devront être produites automatiquement ; mais tant que nous devons les écrire à la main, cette solution est difficilement praticable.

En outre cela ne résout pas le deuxième problème soulevé, et sûrement le plus embêtant.

5.1.2 Prouver la terminaison

Dans le cas où une fonction récursive ne fait pas décroître structurellement un de ses arguments lors des appels récursifs, il existe plusieurs manières, en *Coq*, de trafiquer cette fonction pour qu'elle le fasse. Ces méthodes reviennent, invariablement, à prouver que cette fonction termine.

Le *Coq'art* [1] présente quatre de ces méthodes dans un chapitre intitulé « Récursivité générale ». Les trois premières sont peu satifaisantes dans l'optique que nous étudions car le code des fonctions extraites y diffère sensiblement de l'original en *OCaml*. La quatrième méthode, proposée par Ana Bove, correspond sensiblement plus à ce que nous cherchons. Elle consiste à encoder le domaine de la fonction sous forme d'un prédicat [11] récursif, puis à utiliser ce prédicat comme argument décroissant. Cette méthode n'est cependant pas applicable à une fonction présentant des appels récursifs imbriqués. [12] Et les fonctions produites par *moca* en ont.

5.2 La solution de Frédéric Blanqui [13]

D'après un commentaire par lui écrit dans ses fichiers de preuves, cette manière de procéder est liée aux travaux d'Ana Bove (auteur de la quatrième methode juste au dessus) et de Venanzio Capretta [14].

Il a commencé par éliminer les clauses *when* à la main. C'est-à-dire non pas par la méthode systématique — et explosive — que j'ai détaillée en 5.1.1, mais simplement en réfléchissant… Je crois que l'idée était que, dans une prochaine version, *moca* serait capable de produire de telles fonctions, débarassées des clauses *when*, et donc que nous n'avions pas à nous en préoccuper pour le travail sur les preuves.

Certaines des fonctions produites par *moca* ne sont pas gênées par le fait de faire comprendre à *Coq* qu'elles terminent ; ces fonctions peuvent alors être écrites directement dans *Gallina*, et le sont.

11. Proposition logique dépendant d'un argument.
12. Lorsqu'un argument passé récursivement à une fonction contient lui-même un appel récursif à la même fonction.
13. qui était mon maître de stage depuis Pékin.
14. Notamment l'article [4]

C'est pour les fonctions restantes que ça devient intéressant. Je décris ici la méthode de manière assez concise ; un exemple concret est donné en annexe A.1.

Pour chaque fonction restante, on suppose d'abord son existence. Ensuite on encode sa sémantique sous forme d'une autre hypothèse, exprimant le fait que l'application de la fonction à ses arguments est égal à ce qu'elle calcule alors.

Ceci permet de manipuler la fonction, il reste à obtenir un principe de récurrence adapté qui permette de raisonner dessus. Pour ce faire on commence par supposer l'existence d'une relation d'ordre bien fondée[15], selon laquelle les arguments de la fonction décroissent (strictement) lors des appels récursifs. À partir de cela et grâce aux outils fournis par *Coq* sur ces relations, on énonce et démontre successivement trois principes de récurrence, le dernier étant celui qu'on attendait.

Cette méthode fonctionne très bien — suffisamment bien en tout cas pour avoir permis à mon maître de stage de mener à bout les preuves concernant les monoïdes et les groupes non commutatifs.

Elle présente cependant plusieurs inconvénients à mes yeux :

1. L'élimination des clauses *when* demande de la réflexion. Je veux dire par là une réflexion encore trop humainement intelligente pour être faite automatiquement. Or le but de ces preuves est bien, à terme, d'être produites automatiquement.[16]
2. Elle nécessite, pour chaque fonction encodée de cette manière, de formuler dans *Coq* cinq hypothèses, dont deux ont chacune la taille de la fonction *après élimination des clauses* when[17] Tout ceci a pour effet d'encombrer les contextes des preuves interactives de *Coq*.[18]

15. C'est-à-dire qu'il n'existe pas de suite infinie strictement décroissante au sens de cet ordre. L'exemple le plus classique de relation d'ordre bien fondée est celui de l'ordre habituel $\leq$ sur l'ensemble des entiers positifs, $\boldsymbol{N}$: il n'existe pas de suite infinie $u_0 > u_1 > \ldots \geq 0$. Ce principe, découvert par Fermat, permet de faire des preuves par récurrence sur les entiers positifs.

16. Pour être tout à fait honnête, je ne comprenais, personnellement, pas tout-à-fait
   - comment cette transformation avait été faite ;
   - le code qui en résultait ;
   - comment j'allais faire la même chose aux fonctions, encore plus compliquées, de `group_commutative.ml`

et je n'aime pas travailler avec des choses que je ne comprends pas...

17. Ce qui fait plus qu'avant.

18. Car toutes ces hypothèses appraissent au début de ces contextes, obligeant à faire défiler deux ou trois écrans après chaque commande afin de *voir* le but à prouver. D'un point de vue théorique, ou au niveau de la génération automatique de ces preuves, c'est de peu d'importance, mais tant que quelqu'un (moi surtout) dois les écrire à la main, ça

3. Elle suppose (explicitement) que la fonction est terminante au lieu de le prouver. En fait ça n'est pas gênant parce que, à terme, la terminaison des fonctions devra être prouvée séparément, par un prouveur automatisé de terminaison (quelque chose sur quoi Frédéric Blanqui travaille également). Mais je n'ai appris cela qu'au bout de six semaines de stage...

J'ai donc cherché — et trouvé — une autre manière de faire.

5.3 Ma solution

Je repartit également des travaux d'Ana Bove. [19] L'idée est d'encoder par un prédicat récursif le graphe de la fonction. [20]

Pour cela, tout d'abord, écrire deux prédicats par clause du filtrage *OCaml* : une signifiant que l'argument de la fonction *n'est pas* capturé par cette clause, et un pour signifier qu'il n'est capturé par aucune des clauses jusqu'à celle-là.

À partir de ceci il est aisé d'encoder le graphe de la fonction sous forme d'un prédicat récursif ; les clauses *when* s'écrivant alors très naturellement.

Un inconvénient de cette méthode est qu'au lieu d'utiliser directement le filtrage de motifs de *Coq* pour encoder celui de *OCaml* (ce que faisait Frédéric — après avoir éliminé les clauses *when*) je le réencode par des prédicats.

Son avantage, outre le fait que les clauses *when* ne sont plus un problème, est que le principe de récurrence permettant de raisonner sur la fonction est *automatiquement* produit par *Coq*.

6 Prouver la correction des règles induites par le filtrage

Regardez à nouveau le code des fonctions produites par *moca* en annexe C. Pour prouver la correction [21] d'une fonction, il est nécessaire de prouver la correction de chaque clause du filtrage, c'est-à-dire de prouver que le motif de gauche est bien congru, modulo la relation considérée [22], à celui qui est reconstruit à droite.

---

m'embête.

19. Du moins de l'aperçu que j'en avais eu par l'entremise du *Coq'art*[1]
20. Le graphe d'une fonction est l'ensemble des couples $(argument, résultat)$ pour tous les arguments possibles de cette fonction.
21. Au sens décris en section 4.1
22. C'est-à-dire intuitivement égal. En théorie des groupes.

Par exemple pour la cinquième clause de la fonction `add` (ligne 33)[23], il s'agit de prouver que

$$Add(x, Add(Opp(x), z)) \sim z$$

C'est-à-dire, intuitivement, que

$$x + (-x + z) = z$$

.

Ces règles ne sont pas extrêmement dures à prouver, dans la mesure où il s'agit d'algèbre élémentaire. Cependant le problème, ici, est de mécaniser le plus possible la réflexion. Or il se trouve que ces règles ont été produites, précisément, par un algorithme[24], lequel algorithme est correct et pourrait être prouvé tel ! Mon idée était donc — et est toujours — d'utiliser les outils de *Coq* pour reproduire, dans une moindre mesure peut-être, le comportement de cet algorithme, afin d'obtenir les preuves cherchées.

Je n'ai pas — pas encore — réussi à aller au bout de ceci, essentiellement à cause des limitations de *Coq*. C'est qu'il ne s'agit pas, ici, de manipuler de vrais termes du type `t` (comme `Add (Zero, Gen 3)`, mais des expressions contenant encore des variables libres (comme `Add (Zero, x)`.

J'ai tout de même pu écrire des tactiques qui simplifient et rendent plus mécanique la rédaction de ces preuves, l'essentiel des simplifications algébriques étant fait automatiquement.

Pour aller au bout de l'idée il sera peut-être intéressant d'écrire une tactique adaptée, directement en *OCaml*. (Je sais que *Coq* le permet et je sais que c'est peu documenté...)

7 Conclusion

Il existe encore d'autres points sur lesquels mon approche diffère légèrement de celle utilisée par mon maître de stage dans les deux preuves précédentes, ceci découlant de la nouvelle manière dont j'ai encodé les fonctions de constructions[25], mais j'ai, de manière globale, tâché de suivre la même démarche.

23. `Pervasives.compare x y = 0` équivaut à `x = y`

24. Une variante de la procédure de complétion de Knuth-Bendix, pour ceux que ça intéresse.

25. Par exemple, pour écrire le prédicat affirmant qu'un terme est en forme normale, j'ai préféré — encore — utiliser un prédicat récursif à une fonction définie par filtrage.

En ce qui concerne *moca*, la preuve sur laquelle j'ai travaillé n'est pas encore finie. Cependant j'ai déjà pu y apporter de nouvelles idées qui, on peut l'espérer, donneront quelque chose de bien...

En ce qui me concerne, j'ai énormément approfondi mes connaissances concernant *Coq*[26] et ai découvert ce que peut être la participation à un projet plus vaste que le seul sujet qui m'est confié. Notamment à quel point les échanges d'idées entre chercheurs travaillant sur un même projet peuvent être productifs.

26. Je ne partais peut-être pas de beaucoup, mais tout de même.

Deuxième partie
Annexes

A Application à des exemples des méthodes d'encodage des fonctions

A.1 La méthode de Frédéric Blanqui

J'illustrerai le mécanisme sur la fonction suivante, tirée de `monoid.ml` [27] :

```
let rec add moca_z =
  match moca_z with
  | (Zero, moca_x) -> moca_x
  | (moca_x, Zero) -> moca_x
  | (Add (moca_x, moca_y), moca_z) -> add (moca_x, (add (moca_y, moca_z)))
  | (moca_x, moca_y) -> Add (moca_x, moca_y)
;;
```

Première étape, donc, admettre l'existence de cette fonction :

```
Variable add : t -> t -> t. (* add is curryfied *)
```

Ensuite encoder la sémantique par une autre hypothèse, qui prend la forme d'une égalité :

```
Hypothesis add_eq : forall moca_u moca_v, add moca_u moca_v =
  match (moca_u, moca_v) with
    | (Zero, moca_x) => moca_x
    | (moca_x, Zero) => moca_x
    | (Add moca_x moca_y, moca_z) => add moca_x (add moca_y moca_z)
    | (moca_x, moca_y) => Add moca_x moca_y
  end.
```

L'étape suivante est l'obtension d'un principe de récurrence adapté à la fonction. Pour ce faire on suppose donnée une relation (d'ordre) sur les arguments de `add` :

```
Variable add_lt : relation (t*t).
```

Et on utilise les outils fournis par la bibliothèque standard de *Coq* pour supposer en outre que cette relation est *bien fondée* [28] :

27. Le code *Coq* qui suit est donc tiré de `monoid.v`, écrit par Frédéric Blanqui.
28. *c. f. note 15, page 10*

```
Require Import Relation_Operators.

Hypothesis add_lt_wf : well_founded add_lt.
```

Enfin on signifie que les arguments de `add` décroissent selon cette relation lors des appels recursifs :

```
Hypothesis add_wf : forall moca_u moca_v,
  match (moca_u, moca_v) with
    | (Zero, moca_x) => True
    | (moca_x, Zero) => True
    | (Add moca_x moca_y, moca_z) =>
      add_lt (moca_y, moca_z) (moca_u, moca_v)
      /\ add_lt (moca_x, add moca_y moca_z) (moca_u, moca_v)
    | (moca_x, moca_y) => True
  end.
```

Ceci permet ensuite de prouver successivement trois principes de récurrence, de plus en plus pratiques, le troisième étant celui attendu :

```
Lemma add_ind_pair_match : forall P : t*t -> t -> Prop,
  (forall moca_u moca_v,
    match (moca_u, moca_v) with
      | (Zero, moca_x) => P (moca_u, moca_v) moca_x
      | (moca_x, Zero) => P (moca_u, moca_v) moca_x
      | (Add moca_x moca_y, moca_z) =>
        P (moca_y, moca_z) (add moca_y moca_z) ->
        P (moca_x, add moca_y moca_z) (add moca_x (add moca_y moca_z)) ->
        P (moca_u, moca_v) (add moca_x (add moca_y moca_z))
      | (moca_x, moca_y) => P (moca_u, moca_v) (Add moca_u moca_v)
    end) -> forall p, P p (add (fst p) (snd p)).

Lemma add_ind_match : forall P : t -> t -> t -> Prop,
  (forall moca_u moca_v,
    match (moca_u, moca_v) with
      | (Zero, moca_x) => P moca_u moca_v moca_x
      | (moca_x, Zero) => P moca_u moca_v moca_x
      | (Add moca_x moca_y, moca_z) =>
        P moca_y moca_z (add moca_y moca_z) ->
        P moca_x (add moca_y moca_z) (add moca_x (add moca_y moca_z)) ->
        P moca_u moca_v (add moca_x (add moca_y moca_z))
      | (moca_x, moca_y) => P moca_u moca_v (Add moca_u moca_v)
    end) -> forall u v, P u v (add u v).
```

```
Lemma add_ind : forall P : t -> t -> t -> Prop,
  (forall moca_x, P Zero moca_x moca_x) ->
  (forall moca_x, P moca_x Zero moca_x) ->
  (forall moca_x moca_y moca_z,
    P moca_y moca_z (add moca_y moca_z) ->
    P moca_x (add moca_y moca_z) (add moca_x (add moca_y moca_z)) ->
    P (Add moca_x moca_y) moca_z (add moca_x (add moca_y moca_z))) ->
  (forall moca_x moca_y, nZero moca_x -> nZero moca_y -> nAdd moca_x ->
    P moca_x moca_y (Add moca_x moca_y)) ->
  forall u v, P u v (add u v).
```

A.2 Ma méthode

Je l'illustrerai à l'aide du code que j'ai personnellement étudié et écrit pendant mon stage.

On a, dans `group_commutative.ml`, la fonction suivante (dont je ne garde que deux clauses pour les besoins de l'exemple, la deuxième et la troisième du filtrage) :

```
let rec add moca_z =
  match moca_z with
    (* ... *)
    | (moca_x, Zero) -> moca_x
    | (moca_x, Opp moca_y) when Pervasives.compare moca_x moca_y = 0 -> zero
    (* ... *)
```

Ces deux clauses engendrent les quatres prédicats suivants, pour encoder le filtrage :

```
  Definition safe_add_case2 (z1 z2 : t) :=
    match z1, z2 with
      | x, Zero => False
      | _, _ => True
    end.
  Definition safe_add_until2 z1 z2 :=
    safe_add_until1 z1 z2 /\ safe_add_case2 z1 z2.

  Definition safe_add_case3 (z1 z2 : t) :=
    match z1, z2 with
      | x, Opp y =>
          match Pervasives_compare x y with
            | Eq => False
```

```
          | _ => True
        end
    | _, _ => True
  end.
Definition safe_add_until3 z1 z2 :=
  safe_add_until2 z1 z2 /\ safe_add_case3 z1 z2.
```

Ce dernier, par exemple, signifiant que `(z1, z2)` ne rentre dans aucune des trois premières clauses du filtrage.

Enfin on encode la fonction par :

```
Inductive add_gr : t -> t -> t -> Prop :=
  (* ... *)
  | add_gr2 : forall x, safe_add_until1 x Zero -> add_gr x Zero x
  | add_gr3 : forall x y, safe_add_until2 x (Opp y) ->
      Pervasives_compare x y = Eq -> add_gr x (Opp y) zero
  (* ... *)
```

La validation de ce dernier prédicat entraîne la production par *Coq* du théorème `add_gr_ind`, un principe de récurrence qui suit la structure récursive de `add`.

B Élimination des récursions mutuelles dans des fonctions *OCaml*

Il est un (éventuel) problème qui n'a pas été mentionné jusqu'ici : celui des récursions croisées. À savoir que de manière générale, *moca*, tel que décrit dans l'article [2] peut produire des fonctions mutuellement récursives. Le problème ne s'est pas posé — et n'est pas détaillé dans le présent travail — parce que dans les cas particuliers de `monoid.mlm`, `group.mlm` et `group_commutative.mlm`, les fonctions de construction produites par moca ne sont pas réellement mutuellement récursives.

Dans le cas général, il n'est pas évident que *Coq* permette d'encoder et de travailler facilement avec de telles fonctions. (C'est cependant possible, hein ! Encore une fois nous ne nous sommes pas penché plus que cela sur la question.)

Pour le cas où cela poserait problème, voici une méthode générale qui permettrait d'éliminer les récursions croisées : si on a, par exemple, deux fonctions `f : int -> t -> t` et `g : t -> t` mutuellement récursives, créer le type suivant :

```
type args =
  | F of int * t
  | G of t
```

puis une unique fonction

```
let rec total : args -> t = function
  | F (n, x) -> (* le code de [f] *)
  | G x -> (* le code de [g] *)
```

où, dans les codes de `f` et `g`, les appels récursifs de la forme `f m y` ou `g y` sont respectivement remplacés par `total (F (m, y))` et `total (G y)`. On n'a ainsi plus qu'une fonction récursive : `total`. Cette solution est d'autant plus facilement applicable au code produit par *moca*, que l'équivalent du type `args` y existe déjà.

C Les codes étudiés

C.1 Code *moca*

Code contenu dans `group_commutative.mlm`, fichier passé en argument à *moca* :

```
0   (* Abelian group or Z-module freely generated by argument type 'a. *)
1
2   type 'a t = private
3      | Zero
4      | Gen of 'a
5      | Opp of 'a t
6      | Add of 'a t * 'a t
7          begin
8            associative
9            commutative
10           neutral left (Zero)
11           neutral right (Zero)
12           opposite (Opp)
13         end
14  ;;
```

C.2 Code *OCaml*

À partir du code précédent, *moca* produit celui-ci, qui est utilisable par le compilateur *OCaml* :

```
type 'a
   t =
   | Zero
   | Gen of 'a
   | Opp of 'a t
   | Add of 'a t * 'a t
   (* begin
        associative
        commutative
        neutral left (Zero)
        neutral right (Zero)
        inverse (Opp)
      end *)
;;
let zero = Zero
;;

let gen moca_x = Gen moca_x
;;

let rec compare_add moca_x moca_y =
  match (moca_x, moca_y) with
  | (Opp moca_x, Opp moca_y) -> compare_add moca_x moca_y
  | (Opp moca_x, moca_y) -> compare_add moca_x moca_y
  | (moca_x, Opp moca_y) -> compare_add moca_x moca_y
  | (moca_x, moca_y) -> Pervasives.compare moca_x moca_y

and add moca_z =
  match moca_z with
  | (Zero, moca_x) -> moca_x
  | (moca_x, Zero) -> moca_x
  | (moca_x, Opp moca_y) when Pervasives.compare moca_x moca_y = 0 -> zero
  | (Opp moca_x, moca_y) when Pervasives.compare moca_x moca_y = 0 -> zero
  | (moca_x, Add ((Opp moca_y), moca_z))
      when
      Pervasives.compare moca_x moca_y = 0 ->
    moca_z
  | (Opp moca_x, Add (moca_y, moca_z))
      when
      Pervasives.compare moca_x moca_y = 0 ->
    moca_z
```

```
41      | (Add (moca_x, moca_y), moca_z) -> add (moca_x, (add (moca_y, moca_z)))
42      | (moca_x, Add (moca_y, moca_z)) ->
43        if compare_add moca_x moca_y > 0
44        then add (moca_y, (add (moca_x, moca_z)))
45        else Add (moca_x, (Add (moca_y, moca_z)))
46      | (moca_x, moca_y) when compare_add moca_x moca_y > 0 ->
47        add (moca_y, moca_x)
48      | (moca_x, moca_y) -> Add (moca_x, moca_y)
49
50    and opp moca_x =
51      match moca_x with
52      | Opp moca_x -> moca_x
53      | Add (moca_x1, moca_x2) -> add ((opp moca_x2), (opp moca_x1))
54      | Zero -> zero
55      | _ -> Opp moca_x
56    ;;
57
58    external eq_t : 'a t -> 'a t -> bool = "%equal"
59    ;;
60
```

Références

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[2] Frédéric Blanqui, Thérèse Hardin, and Pierre Weis. On the implementation of construction functions for non-free concrete data types. In *16th European Symposium on Programming - ESOP'07*, volume 4421, Braga, Portugal, 2007.

[3] Frédéric Blanqui, Pierre Weis, and Richard Bonichon. *Moca : un générateur de modules pour les types à relations.* INRIA. Version 0.7.0. http://moca.inria.fr/.

[4] Venanzio Capretta. A polymorphic representation of induction-recursion. http://www.cs.ru.nl/~venanzio/publications/induction_recursion.pdf.

[5] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system. Documentation and user's manual*. INRIA, France. Release 3.12. Homepage at http://caml.inria.fr.

[6] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.