`Field`: une procédure de décision pour les nombres réels en Coq

David Delahaye[1] & Micaela Mayero[2]

INRIA Rocquencourt, Projet LogiCal,
Domaine de Voluceau BP 105,
78153 Le Chesnay Cedex

1. David.Delahaye@inria.fr
2. Micaela.Mayero@inria.fr

Résumé

Nous nous proposons d'automatiser les preuves d'égalités sur les nombres réels dans le système Coq en utilisant la théorie des corps commutatifs. L'idée de l'algorithme consiste à se débarrasser des inverses afin de pouvoir se brancher sur la procédure de décision déjà existante sur les anneaux abéliens (`Ring`). L'élimination des inverses se fait de manière complètement réflexive et la réflexion est réalisée au moyen d'un nouveau langage de tactiques intégré au système Coq (version V7). Nous pensons étendre cette tactique à tous les corps commutatifs bien qu'actuellement, seuls les nombres réels soient concernés.

1. Introduction

La théorie des nombres réels dans le système Coq est axiomatique[1] et les preuves se font à grand renfort de réécritures[2], ce qui fait grossir la taille des scripts ainsi que les termes preuves. La tactique `Ring` qui permet de décider d'égalités sur les anneaux abéliens a permis de résoudre partiellement ce problème et se révèle très utile dans les preuves ne faisant pas intervenir l'inverse. On peut l'utiliser dans des théorèmes auxiliaires triviaux servant à compléter la théorie comme :

$$\forall x, y \in \mathbb{R}. x + y = 0 \rightarrow y = (-x)$$

où pas moins de cinq réécritures sont nécessaires pour résoudre sans la tactique `Ring`. Mais, on peut aussi l'utiliser dans des théorèmes non triviaux où l'inverse n'intervient pas, comme le théorème des 3 intervalles ([11]).

Cependant, `Ring` ne règle pas le cas des égalités avec inverses qui, si elles ne sont pas difficiles à montrer, n'en restent pas moins très fastidieuses à résoudre. Ces preuves sont d'autant plus pénibles à construire qu'elles ralentissent de manière conséquente le développement de la théorie des nombres réels. En effet, les théorèmes sur les limites et les dérivées font systématiquement intervenir des égalités avec des inverses dans les "preuves ε"[3] lorsque l'on veut composer, additionner, ... Ces égalités sont relativement triviales et alourdissent considérablement les preuves fondamentales de limites et de dérivées. Un exemple typique est celui de la dérivée de l'addition où l'on doit montrer que la somme des dérivées est égale à la dérivée de la somme. Pour ce faire, on prend deux fonctions f et g ainsi que leurs dérivées en x_0, c'est-à-dire $f'(x_0)$ et $g'(x_0)$. Par définition, les deux dérivées peuvent s'exprimer comme suit :

[1]Ce choix provient d'un souci non seulement de simplicité mais aussi de rapidité dans le développement de la théorie.

[2]Une construction des nombres réels ne changerait pas ce phénomème car $\mathbb{R}$ ne serait pas non plus un type inductif sur lequel on pourrait calculer. Voir, par exemple, [8] pour s'en rendre compte.

[3]On appelle preuves ε, les preuves utilisant la définition explicite de la limite sous la forme : $\forall \varepsilon > 0. \exists \alpha.....$ Dans la littérature anglaise, ces preuves sont plutôt connues sous la dénomination de preuves ε/δ.

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

$$g'(x_0) = \lim_{x \to x_0} \frac{g(x) - g(x_0)}{x - x_0}$$

En utilisant le théorème d'addition des limites, on obtient directement :

$$f'(x_0) + g'(x_0) = \lim_{x \to x_0} \left(\frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} \right)$$

En utilisant la définition de la limite, on a pour tout domaine D de $\mathbb{R}$:

$$\forall \varepsilon > 0,\ \exists \alpha > 0,\ \forall x \in D \backslash x_0,\ \text{si } |x - x_0| < \alpha \text{ alors}$$

$$\left| \frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} - (f'(x_0) + g'(x_0)) \right| < \varepsilon$$

Maintenant, il suffit de montrer l'égalité suivante :

$$\frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} = \frac{f(x) + g(x) - (f(x_0) + g(x_0))}{x - x_0}$$

pour conclure que $f'(x_0) + g'(x_0)$ et $(f + g)'(x_0)$ coïncident. Cette dernière égalité est clairement triviale mais la preuve formelle l'est beaucoup moins[4]. Il faut d'abord réduire au même dénominateur (4 réécritures) puis montrer l'égalité sur les numérateurs (6 réécritures). L'utilisation de `Ring` sur les numérateurs économise des réécritures et au total, cette égalité nécessite 4 réécritures + 1 tactique (`Ring`). Nous pensons que ce genre d'égalité doit être résolue directement par une tactique, à la fois pour un gain de temps considérable dans les développements mais aussi de concision dans les scripts où l'on a plutôt envie de rendre implicites de telles preuves.

Dans ce papier, nous présenterons, tout d'abord, l'algorithme que nous avons utilisé pour décider les égalités sur les corps commutatifs (sur $\mathbb{R}$ en particulier) modulo certaines inégalités (conditions que les dénominateurs doivent être non nuls). Ensuite, nous donnerons une idée assez globale de l'implantation qui se fait de manière totalement réflexive et qui utilise des spécificités propres à la version V7 de `Coq`, à savoir le langage de tactiques. Enfin, nous nous appuierons sur plusieurs exemples, faciles pour certains et plus réalistes pour d'autres afin de mettre en évidence la correction de la tactique ainsi que ses performances.

2. Algorithme

2.1. Principe

L'idée de l'algorithme est de minimiser les opérations de simplification de manière à se brancher le plus tôt possible sur la procédure de décision sur les anneaux abéliens (`Ring`). Cela signifie qu'il faut éliminer tous les inverses intervenant dans l'égalité qu'il s'agit de résoudre.

Pour ce faire, nous proposons la suite d'étapes suivantes :

- Transformer les expressions $x - y$ en $x + (-y)$ et x/y en $x * 1/y$.
- Chercher tous les inverses apparaissant dans l'égalité pour en faire un produit.
- Distribuer totalement à gauche et à droite de l'égalité, excepté dans les inverses.
- Associer à droite chaque monôme, excepté dans les inverses[5].

[4]Il ne s'agit pas ici d'entamer le leitmotiv bien connu que les preuves formelles sont bien plus difficiles que les preuves que l'on peut trouver dans les meilleurs ouvrages de mathématiques mais de mettre en évidence une lacune d'automatisation qui, si elle venait à être comblée, pourrait nous permettre certaines "imprécisions" dans le sens où l'utilisateur n'aurait plus à montrer, à la main, certaines propriétés considérées comme triviales.

[5]Cette étape est clairement non nécessaire mais elle permet un gain d'efficacité en évitant un double appel récursif pour toutes les fonctions manipulant ces expressions.

- Multiplier à gauche et à droite par le produit d'inverses, que l'on a construit précédemment, en générant la condition que tous les inverses doivent être non nuls.
- Distribuer seulement le produit sur la somme de monômes à gauche et à droite sans réassocier à droite.
- Éliminer les inverses des monômes en utilisant la règle de corps $x.1/x = 1$, si $x \neq 0$ et en permutant les éléments du monôme si nécessaire, c'est-à-dire s'il reste des inverses et que la règle de corps ne peut pas s'appliquer[6].
- Recommencer le processus s'il reste encore des inverses.

La dernière étape qui consiste à réitérer le processus s'explique par le fait qu'il peut y avoir d'autres inverses dans les inverses et ce, dans des expressions pouvant être compliquées. Pour éviter la réitération, une idée serait de se lancer dans une simplification directe en utilisant la règle $1/1/x = x$, si $x \neq 0$. Toutefois, l'expérience a montré que le codage de cette simplification était plutôt complexe et générait un lemme de correction difficile. Par ailleurs, les expressions devant être différentes de 0 n'étaient pas exactement les mêmes que celles nécessaires pour l'élimination des inverses dans les monômes. On pouvait certes les déduire mais le lemme de correction correspondant à l'élimination des inverses devenait alors plus compliqué à montrer. On a donc tout avantage à se limiter à la règle $x.1/x = 1$, si $x \neq 0$ qui tend à simplifier grandement l'algorithme et ce pour une perte d'efficacité négligeable en pratique[7].

Après ces étapes, nous obtenons une expression débarrassée de tous ses inverses et il suffit d'appeler `Ring` pour conclure.

2.2. Exemple

Considérons un petit exemple en détaillant les étapes de preuve afin de voir comment la procédure fonctionne ; étant donné x et y, deux variables réelles, on se propose de montrer l'égalité suivante :

$$x * (\frac{1}{x} + \frac{x}{x+y}) = (-\frac{1}{y}) * y * (-(\frac{x*x}{x+y}) - 1)$$

On commence par transformer les moins binaires et les divisions :

$$x * (\frac{1}{x} + x * \frac{1}{x+y}) = (-\frac{1}{y}) * y * (-(x*x) * \frac{1}{x+y} + (-1))$$

On construit le produit d'inverses que l'on appellera p :

$$p = x * ((x+y) * (y * (x+y)))$$

On distribue totalement à gauche et à droite sauf dans les inverses :

$$x * \frac{1}{x} + x * \frac{1}{x+y} = (-1) * \frac{1}{y} * y * ((-1) * (x*x) * \frac{1}{x+y}) + (-1) * \frac{1}{y} * (-1)$$

On associe à droite chaque monôme sauf dans les inverses :

$$x * \frac{1}{x} + x * \frac{1}{x+y} = (-1) * (\frac{1}{y} * (y * ((-1) * (x * (x * \frac{1}{x+y}))))) + (-1) * (\frac{1}{y} * (-1))$$

On multiplie à gauche et à droite par p en générant la condition de correction :

[6]Il n'est pas nécessaire ici de vérifier que $x \neq 0$ car la condition a déjà été générée lors de la multiplication par le produit de tous les inverses.

[7]On estime, en effet, que les expressions contenant des empilements d'inverses d'inverses seront plutôt rares.

$$(x*((x+y)*(y*(x+y))))*(x*\frac{1}{x}+x*\frac{1}{x+y})$$
$$=$$
$$(x*((x+y)*(y*(x+y))))*((-1)*(\frac{1}{y}*(y*((-1)*(x*(x*\frac{1}{x+y})))))+(-1)*(\frac{1}{y}*(-1)))$$

Avec $x*((x+y)*(y*(x+y)))\neq 0$.

On distribue ce produit sur les monômes sans réassocier à droite :

$$(x*((x+y)*(y*(x+y))))*(x*\frac{1}{x})+$$
$$(x*((x+y)*(y*(x+y))))*(x*\frac{1}{x+y})$$
$$=$$
$$(x*((x+y)*(y*(x+y))))*(((-1)*(\frac{1}{y}*(y*((-1)*(x*(x*\frac{1}{x+y}))))))+$$
$$(x*((x+y)*(y*(x+y))))*((-1)*(\frac{1}{y}*(-1)))$$

On élimine les inverses en permutant si nécessaire :

$$((x+y)*(y*((x+y))))*x+(x*(y*(x+y)))*x$$
$$=$$
$$(x*(x+y))*((-1)*(y*((-1)*(x*x))))+(x*((x+y)*(x+y)))*((-1)*(-1))$$

On obtient alors une égalité sur les anneaux abéliens que `Ring` sait résoudre.

2.3. Remarques

Nous n'avons pas encore formalisé la preuve que cet algorithme décide bien des égalités sur les nombres réels et sur les corps commutatifs plus généralement modulo certaines preuves d'inégalités. Il semble clair, cependant, qu'il est correct dans la mesure où l'on n'utilise que des axiomes de corps. On peut également se convaincre de la terminaison de la procédure puisque le nombre d'inverses décroît à chaque étape.

Par ailleurs, il est important de souligner que notre démarche ne vise pas à résoudre le problème global de décision sur les corps commutatifs dont on ne sait pas, *a priori*, s'il est décidable ou non. En effet, nous ne cherchons pas à prouver les conditions sur les inverses qui sont laissées à l'utilisateur. Ainsi, nous nous plaçons dans une optique où l'inverse est une fonction totale.

Dans un souci de généralité, notre méthode a pour vocation, à terme, de traiter tous les corps commutatifs. Si on avait voulu traiter seulement les nombres réels, notre approche aurait été bien différente et on aurait certainement opté pour des algorithmes résolvant au premier ordre tels que, entre autres, la méthode de Tarski[8] ([12]), l'algorithme de Kreisel-Krivine[9] ([9]) ou la décomposition cylindrique de Collins ([3]).

Toujours dans cette optique plus générale, on peut citer le travail du projet Fundamental Theorem of Algebra (Herman Geuvers, Freek Wiedijk, Jan Zwanenburg, Randy Pollack et Henk Barendregt), avec, dans le cadre d'une axiomatisation constructive des nombres réels en `Coq`, le codage de la tactique réflexive `Rational` ([6]), traitant des égalités similaires à celles que nous nous proposons de résoudre. Notre approche se démarque essentiellement du fait de choix différents dans la formalisation des nombres réels. Tout d'abord, le fait que la fonction inverse soit totale permet de faire une réflexion totale des expressions de $\mathbb{R}$ ce qui n'est pas le cas dans `Rational` où tout inverse contient aussi la preuve que le dénominateur est non nul. La réflexion doit donc aussi être

[8]Cet algorithme ne fonctionne qu'en logique classique ([5]) mais ce n'est pas gênant dans la mesure où il en est de même pour les nombres réels en `Coq` à cause de l'axiome d'ordre total.

[9]Kreisel et Krivine se sont aussi intéressés à un algorithme dans d'autres structures telles que les corps algébriquement clos, les anneaux de Boole séparables, ...

partielle ce qui rend le processus plus complexe. Enfin, nous considérons l'égalité de Leibniz, ce qui permet à l'utilisateur d'appliquer des tactiques de réécriture à n'importe quel prédicat alors que dans `Rational`, l'égalité est plus large (setoïde) et il est nécessaire de prouver des lemmes de compatibilité afin de passer au contexte.

3. Implantation

Comme nous l'avons dit précédemment, l'implantation de cette procédure de décision sur les nombres réels, que nous avons appelée `Field`, a été réalisée dans la version V7 de Coq afin de pouvoir profiter des nouvelles possibilités du langage de tactiques. Bien que la version V7 de Coq soit encore très expérimentale et, de ce fait, non disponible, le code de ce développement peut être récupéré en ligne à l'adresse suivante :

ftp ://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/Field/Field.v

3.1. À propos de la réflexion

Pour coder `Field`, il y a globalement deux choix possibles. Un codage explicite en utilisant la réécriture ou un codage par réflexion en utilisant la réduction. Le codage explicite (approche à la LCF) est très coûteux du fait de l'utilisation de la réécriture qui prend du temps mais aussi et surtout de la place dans le terme preuve[10]. Le codage par réflexion est une alternative complètement satisfaisante pour laquelle nous avons opté. En effet, les réécritures sont remplacées par des phases de réduction plus efficaces et la taille du terme preuve est de l'ordre de celle du but à résoudre. Par ailleurs, on peut formaliser clairement la correction globale de la tactique ainsi que sa complétude (même si nous ne l'avons pas fait ici) alors que dans l'approche explicite, c'est bien plus difficile, voire impossible.

Avant de donner une idée de l'implantation de `Field`, rappelons rapidement le principe d'une tactique codée par réflexion. Soit un langage C des termes concrets (typiquement un type quelconque) et un langage A des termes abstraits (typiquement un type inductif). Comme on ne peut pas manipuler les termes du langage C comme on voudrait (on ne peut pas filtrer), l'idée est de le réfléchir dans le langage A qui lui est isomorphe. Une première phase, appelée métaification par Samuel Boutin ([2]), consiste donc à traduire les termes de C vers les termes de A. Plus précisément, cela consiste, pour un terme c de C, à trouver le terme a de A tel que $(f\ v\ a) = c$, où f est la fonction d'interprétation de A vers C (codable dans Coq), v est une liste d'associations contenant les parties de C que l'on ne réfléchit pas (atomes) et $=$ est l'égalité de Leibniz. On peut se passer de la liste d'associations à condition que l'égalité sur les atomes soit décidable car on a généralement besoin de comparer les termes de A. La métaification s'assimile exactement à une phase d'analyse syntaxique comme on pourrait la trouver dans un langage de programmation.

Ensuite, on peut coder la fonction t de transformation des termes de A. Pour l'utiliser, il suffit de prouver un lemme de correction[11] de la forme :

$$\forall a \in A.(f\ v\ (t\ a)) = (f\ v\ a)$$

Enfin, après avoir appliqué ce lemme de correction, il suffit de réduire totalement (`Compute`) pour transformer le terme abstrait (de A) et revenir à un terme concret (de C). Pour pousser l'analogie avec les langages de programmation, on pourrait voir ces deux étapes comme une phase d'évaluation suivie d'une phase de "pretty-print".

On peut résumer la situation au moyen du schéma suivant :

[10] La taille du terme preuve est un point auquel il faut être très sensible car il n'est pas rare de rencontrer des scripts de preuves corrects pour lesquels on ne peut pas construire le terme preuve, faute de mémoire suffisante.

[11] Il est intéressant de voir ici que dans le processus de réflexion, la tactique et la preuve de sa correction sont indissociables.

$$\begin{tikzcd}[column sep=huge, row sep=huge]
C \arrow[r, "\text{Métaification}"] \arrow[d, "\text{Codage explicite}"'] & A \arrow[d, "t"] \\
C & A \arrow[l, "f"]
\end{tikzcd}$$

Réflexion

Pour une description complète de la réflexion, on pourra se reporter à [2] et [7].

3.2. Codage de la tactique

Nous allons maintenant entrer dans les détails de l'implantation de l'algorithme donné précédemment. Nous utiliserons une syntaxe spécifique au langage Coq dont on pourra trouver une documentation complète dans [1]. Les seules spécificités de la version V7, que nous utiliserons, concernent le langage de tactiques. On pourra consulter [4] pour un aperçu de ce langage, ainsi que le document suivant pour une documentation complète :

ftp ://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/ltac/ltac-doc.ps

3.2.1. Structure de réflexion

Dans le cas de `Field`, les termes concrets sont exactement les expressions réelles (dans $\mathbb{R}$) et C correspond donc à `R`. Les termes abstraits correspondent à la partie corps commutatif de `R` et se définissent très simplement par un type inductif que nous avonc appelé `ExprR` :

```
Inductive ExprR : Set :=
| ER0    : ExprR
| ER1    : ExprR
| ERplus : ExprR -> ExprR -> ExprR
| ERmult : ExprR -> ExprR -> ExprR
| ERopp  : ExprR -> ExprR
| ERinv  : ExprR -> ExprR
| ERvar  : nat -> ExprR.
```

Les variables sont des expressions réelles quelconques pour lesquelles on ne peut clairement pas décider l'égalité. On les remplace donc par des indices entiers (`nat`) pour pouvoir décider l'égalité entre variables et on associe à un terme une liste d'associations entre les indices et les expressions de `R`.

3.2.2. Métaification

Pour traduire les expressions de `R` vers `ExprR` (métaification), il faut utiliser le métalangage de Coq. Jusqu'à la V7 exclue, le seul moyen était de coder cette traduction dans Objective Caml (le langage d'implantation et le métalangage de Coq, [10]) en utilisant un fichier ML que l'on compilait avec le système et que l'on pouvait importer dans un toplevel bytecode de Coq. Ce protocole était un peu lourd à mettre en œuvre[12], d'autant que le processus de métaification est

[12] En effet, il faut se procurer Objective Caml, Camlp4, compiler les sources de Coq, coder la métaification en comprenant la structure abstraite des termes Coq et enfin compiler le fichier en question.

extrêmement simple. Dans la V7, le langage de tactiques permet de se libérer de ce genre de contraintes au moyen d'un noyau fonctionnel et d'opérateurs de filtrage élaborés. Par ailleurs, un autre atout intéressant est, qu'étant intégré au toplevel de Coq, il est possible de faire tourner le code au moyen d'un toplevel compilé en natif sans perdre la modularité[13] du sytème.

Dans `Field`, pour métaifier, on utilise d'abord une tactique appelée `BuildVarList` qui construit la liste d'associations en évitant les doublons puis on appelle la fonction d'interprétation de `R` dans `ExprR` définie de la manière suivante :

```
Recursive Tactic Definition interp_R lvar trm :=
  Match trm With
  | [R0] -> ER0
  | [R1] -> ER1
  | [(Rplus ?1 ?2)] ->
    Let e1 = (interp_R lvar ?1)
    And e2 = (interp_R lvar ?2) In
    '(ERplus e1 e2)
  | [(Rmult ?1 ?2)] ->
    Let e1 = (interp_R lvar ?1)
    And e2 = (interp_R lvar ?2) In
    '(ERmult e1 e2)
  | [(Ropp ?1)] ->
    Let e = (interp_R lvar ?1) In
    '(ERopp e)
  | [(Rinv ?1)] ->
    Let e = (interp_R lvar ?1) In
    '(ERinv e)
  | [?1] ->
    Let idx = (Assoc ?1 lvar) In
    '(ERvar idx).
```

où `Assoc` est une tactique qui donne l'indice correspondant à la variable réelle. À ce niveau là, on suppose que les constantes `Rminus` et `Rdiv`, à savoir respectivement les constantes du moins binaire et de division, ont déjà été dépliées.

3.2.3. Construction du multiplicateur

Ici, il s'agit de construire un produit de facteurs sans doublons inutiles (doublons qui apparaîtront entre monômes après distribution), constitué des inverses de l'égalité (on ne prend pas en compte les inverses dans les inverses qui seront traités dans d'autres passes de `Field`). Pour ce faire, on a le choix entre utiliser une fonction de Coq ou une tactique que l'on peut plus facilement écrire dans la V7. Pour des raisons d'aisance de programmation, nous avons opté pour une tactique puisque l'on est moins limité, entre autres, dans la récursivité. Le multiplicateur nous est donc donné par la tactique `GiveMult` utilisant la tactique `RawGiveMult` qui donne la liste des inverses :

[13]En effet, le chargement dynamique de fichiers bytecode dans un exécutable natif n'est pas encore très standard et le chargement dynamique de fichiers natifs dans du natif n'est, quant à lui, clairement pas compris. La commande `coqmktop` permet de créer un toplevel "customisé" éventuellement en natif en indiquant une liste de fichiers à inclure au moment de l'édition de liens. Toutefois, le processus est purement statique et on perd la possibilité d'appeler d'autres tactiques qui n'ont pas été "linkées" au moment du `coqmktop`.

```
Recursive Tactic Definition RawGiveMult trm :=
  Match trm With
  | [(ERinv ?1)] -> '(cons ExprR ?1 (nil ExprR))
  | [(ERopp ?1)] -> (RawGiveMult ?1)
  | [(ERplus ?1 ?2)] ->
    Let l1 = (RawGiveMult ?1)
    And l2 = (RawGiveMult ?2) In
    (Union l1 l2)
  | [(ERmult ?1 ?2)] ->
    Let l1 = (RawGiveMult ?1)
    And l2 = (RawGiveMult ?2) In
    Eval Compute in (app ExprR l1 l2)
  | _ -> '(nil ExprR).

Tactic Definition GiveMult trm :=
  Let ltrm = (RawGiveMult trm) In
  '(mult_of_list ltrm).
```

où `nil`, `cons` et `app` sont les constructeurs et la concaténation des listes polymorphes. `Union` est une tactique qui concatène deux listes en éliminant les doublons (éléments communs aux deux listes). `mult_of_list` est une fonction Coq qui rend un produit associé à droite à partir d'une liste d'expressions de `ExprR`. Le `Eval Compute in` utilisé dans `RawGiveMult` permet de réduire les `app` pour obtenir une liste canonique pouvant être correctement filtrée (par `mult_of_list`).

3.2.4. Distributivité et associativité

Il s'agit maintenant de distribuer totalement (sauf dans les inverses) et d'associer à droite (par rapport à l'addition et à la multiplication) dans les membres de l'égalité. Ces deux fonctions se font obligatoirement dans Coq car l'idée est de passer du terme initial au terme transformé via la réduction de Coq et un lemme de correction à prouver pour chaque fonction.

La distributivité ne peut pas être codée directement et facilement dans Coq. En effet, les conditions de garde assurant la normalisation forte obligent à découper le problème de manière à faire des appels récursifs respectant la décroissance de la mesure (ordre sous-terme). Les fonctions peuvent donc sembler un peu compliquées mais il s'agit surtout de respecter ces conditions syntaxiques. Pour distribuer, l'idée est de distribuer d'abord tous les moins unaires `ERopp`. Ensuite, on distribue totalement dans les sous-termes et, pour le cas de la multiplication `ERmult`, on distribue d'abord à gauche puis à droite. Nous ne donnerons pas ici les fonctions en question qui ne présentent pas un intérêt particulier. Le lecteur intéressé pourra se reporter au code source. Pour utiliser la fonction de distributivité `distrib`, on a prouvé le lemme de correction suivant :

```
Lemma distrib_correct:
  (e:ExprR)(lvar:(list (Sprod R nat)))
    (interp_ExprR lvar (distrib e))==(interp_ExprR lvar e).
```

où `interp_ExprR` est la fonction d'interprétation de `ExprR` vers `R` (écrite en Coq) et `lvar` la fonction d'associations des variables.

L'associativité ne pose pas de problèmes dans son codage moyennant quelques précautions. De même que pour `distrib`, nous ne donnerons pas le code de la fonction d'associativité, nommée `assoc`, et il nous a fallu prouver le lemme de correction suivant :

```
Lemma  assoc_correct:
  (e:ExprR)(lvar:(list (Sprod R nat)))
    (interp_ExprR lvar (assoc e))==(interp_ExprR lvar e).
```

Après avoir appliqué `distrib` et `assoc`, on obtient à gauche et à droite de l'égalité à prouver, deux termes qui sont des sommes de monômes associés à droite.

3.2.5. Multiplier les membres de l'égalité

Pour pouvoir effectuer la simplification des inverses, on multiplie ensuite par le multiplicateur qui a été construit précédemment (produit de tous les inverses excepté ceux qui sont dans d'autres inverses). Pour ce faire, il suffit de montrer le lemme suivant sur les expressions de ExprR et qui a directement son équivalent dans R :

```
Lemma mult_eq:
  (e1,e2,a:ExprR)(lvar:(list (Sprod R nat)))
    ~((interp_ExprR lvar a)==R0)->
      (interp_ExprR lvar (ERmult a e1))==(interp_ExprR lvar (ERmult a e2))->
        (interp_ExprR lvar e1)==(interp_ExprR lvar e2).
```

Ce lemme permet de générer le but (que l'utilisateur devra prouver) que le multiplicateur doit être non nul. Ceci équivaut à dire que tous ses facteurs doivent être nuls ce qui est cohérent dans la mesure où l'on est sûr de devoir simplifier ces expressions.

Une fois le produit effectué, on distribue ce produit sur les monômes sans réassocier à droite, ce qui est trivialement fait par une fonction Coq appelée `multiply` dont le lemme de correction est complètement similaire à ceux donnés précédemment pour la distributivité et l'associativité.

3.2.6. Élimination des inverses

L'élimination des inverses se fait monôme par monôme. À ce stade, un monôme est un produit du multiplicateur et d'une expression qui est un produit associé à droite (monôme obtenu après la phase de distributivité et d'associativité). Étant donné que la simplification doit se faire modulo permutation des membres du monôme (de manière à faire apparaître $x.1/x$), le plus simple consiste à manipuler des listes. On transforme donc le multiplicateur en une liste l_p et le reste du monôme en une liste l_m. Pour que les simplifications soient correctes, il faut considérer une troisième liste l_n qui est une liste d'expressions de ExprR non nulles (différentes de ER0). Ainsi, l'algorithme de simplification consiste à parcourir l_p et pour chaque élément de l_p, s'il est dans l_n et que son inverse est dans l_m alors enlever l'inverse de l_m sinon on rajoute cet élément à la fin de l_m. Finalement, on rend l_m que l'on retransforme en monôme. Dans notre cas, on aura toujours $l_p = l_n$ puisque les expressions du multiplicateur sont exactement les expressions que l'on va simplifier.

Concrètement, le travail est effectué par les fonctions Coq suivantes :

```
Definition monom_simplif [l:(list ExprR);e:ExprR] : ExprR :=
  Cases (list_of_monom e) of
  | (pairT l1 l2) =>
    (monom_of_list (remove_list ExprR eqExprR inverse l l1 l2))
  end.

Fixpoint inverse_simplif [l:(list ExprR);e:ExprR] : ExprR :=
  Cases e of
  | (ERplus e1 e2) => (ERplus (monom_simplif l e1) (inverse_simplif l e2))
  | _ => (monom_simplif l e)
  end.
```

où `list_of_monom` et `monom_of_list` permettent respectivement de transformer un monôme en une liste de ses facteurs et une listes d'expressions en un monôme. `remove_list` réalise exactement l'algorithme que l'on vient de donner avec l_n=l, l_p=l1 et l_m=l2. `inverse` est la fonction, qui étant donnée une expression, rend son inverse. Elle permet de tester si l'inverse d'un élément de l1 est bien dans l2.

Le lemme de correction d'élimination des inverses s'exprime comme suit :

```
Lemma inverse_correct:
  (e:ExprR)(l:(list ExprR))(lvar:(list (Sprod R nat)))(make_mult lvar l)->
    (interp_ExprR lvar (inverse_simplif l e))==(interp_ExprR lvar e).
```

où make_mult génère la condition que l'interprétation du produit des expressions de l doit être non nulle (différente de R0). Cette condition permet d'assurer la correction des simplifications et ce pour tout l. Pour appliquer ce lemme et comme on l'a dit précédemment, on utilisera la liste des facteurs du multiplicateur pour l.

3.2.7. La tactique globale

La tactique Field combine toutes les phases que nous venons de voir. Elle s'exprime directement dans le langage de tactiques de la V7 comme suit :

```
Tactic Definition Field :=
  Unfold Rminus;Unfold Rdiv;
  Match Context With
  | [|-?1== ?2] ->
    Let lvar = (BuildVarList '(Rplus ?1 ?2)) In
    Let trm1 = (interp_R lvar ?1)
    And trm2 = (interp_R lvar ?2) In
    Let mul = (GiveMult '(ERplus trm1 trm2)) In
    Let lmul = Eval Compute in (list_of_mult mul) In
    Cut (interp_ExprR lvar trm1)==(interp_ExprR lvar trm2);
    [Compute;Auto
    |(ApplySimplif ApplyDistrib);(ApplySimplif ApplyAssoc);
     (Multiply mul);[(ApplySimplif ApplyMultiply);
       (ApplySimplif (ApplyInverse lmul));
       (Let id = (GrepMult ()) In Clear id);Compute;
       First [(InverseTest ());Ring|Field]|Idtac]].
```

La première ligne de Field (série d'Unfold) permet de déplier les moins binaires et les divisions. Ensuite, on crée la liste d'associations des variables (dans lvar) avec BuildVarList pour interpréter les deux membres de l'égalité, ce qui donne deux termes trm1 et trm2 de ExprR. Le multiplicateur mul est donné par GiveMult en prenant soin de sommer les deux termes pour tenir compte des inverses des deux membres de l'égalité. On transforme alors le multiplicateur en une liste de facteurs au moyen de list_of_mult, qui sera donnée au lemme de correction concernant l'élimination des inverses. Le Cut permet d'insérer les termes de ExprR dans le but à prouver. Par la suite, on peut leur appliquer les différentes transformations dont nous avons parlé précédemment au moyen de tactiques. ApplyDistrib applique la distributivité, ApplyAssoc l'associativité, Multiply la multiplication par le multiplicateur à gauche et à droite de l'égalité, ApplyMultiply la distribution du multiplicateur et ApplyInverse l'élimination des inverses. ApplySimplif permet d'appliquer la tactique à gauche et à droite de l'égalité pour les tactiques qui travaillent sur un terme. On se débarrasse de l'hypothèse que l'interprétation du multiplicateur doit être non nulle au moyen de GrepMult, qui rend le nom de cette hypothèse pouvant être ainsi effacées (Clear). Enfin, on teste s'il reste encore des inverses dans les termes de l'égalité grâce à la tactique InverseTest qui, soit ne fait rien s'il ne reste pas d'inverses permettant ainsi l'appel à Ring, soit échoue dans le cas contraire impliquant une nouvelle application de Field.

4. Exemples

Nous donnons ici quelques exemples, accompagnés du temps d'exécution. Ces exemples sont tirés de preuves faisant partie ou allant faire partie du développement[14] des nombres réels où l'utilisation de cette tactique sera la bienvenue (autant par commodité que dans le but d'alléger les preuves).

Les tests sont effectués sur un Pentium III à 450Mhz sous Linux (RedHat 5.2) et la version (V7) de Coq utilisée a été compilée en natif. En effet, comme nous l'avons dit précédemment, l'utilisation du langage de tactiques de Coq pour coder Field s'adapte parfaitement à la compilation native.

[14]Plus précisément, deux fichiers sont principalement concernés : Rlimit.v et Rderiv.v.

L'appel à Ring ne pose aucun problème puisque le code ML correspondant est linké par défaut dans la version native.

4.1. Exemple 1

L'exemple que nous donnons ici appartient à une famille d'égalités que nous pouvons qualifier de simples. Néanmoins, ce type d'égalités revient assez souvent dans les preuves et l'accumulation de toutes ces petites preuves devient rapidement fastidieuse et fini par produire un terme preuve plus important qu'il ne devrait. Pour cette raison, il est intéressant de pouvoir utiliser `Field` fréquemment, tout comme `Ring` afin de minimiser le nombres de réécritures.

Nous considérons, par exemple, les parties de preuves présentes de manière récurrente telles que : $b = \frac{b}{a} \times a$.

```
Welcome to Coq 7.00 (December 1999)

Coq < Goal (a,b:R)``b == b*(1/a)*a``.

Unnamed_thm < Intros. Time Field.
1 subgoal

  a : R
  b : R
  ============================
   ``a <> 0``
Finished transaction in 0 secs (0.35u,0s)
```

Sans aucune réécriture, il ne nous reste plus qu'à prouver que $a \neq 0$, propriété que nous devions impérativement prouver même dans le cas où nous procédions par réécritures.

4.2. Exemple 2

Nous voulons prouver que $\frac{\varepsilon}{2+2} + \frac{\varepsilon}{2+2} = \frac{\varepsilon}{2}$. Cette opération est utilisée dans la preuve concernant la multiplication des limites (`limit_mul`). La preuve du but énoncé ci-dessous fait actuellement environ 25 lignes de Coq, alors qu'après l'application de `Field`, il nous reste à prouver uniquement que $2 + 2$ et 2 sont non nuls.

```
Coq < Goal (eps:R)``eps*1/(2+2)+eps*1/(2+2) == eps*1/2``.

Unnamed_thm < Intro. Time Field.
1 subgoal

  eps : R
  ============================
   ``(2+2)*2 <> 0``
Finished transaction in 0 secs (0.51u,0s)
```

Comme dit précédemment, pour pouvoir simplifier par $2 + 2$ et par 2, nous utilisons le fait que $2 + 2 \neq 0$ et $2 \neq 0$. Ces deux conditions sont générées sous la forme d'un unique sous-but traduisant ces deux propriétés : $(2 + 2) \times 2 \neq 0$. En effet, si un produit est non nul alors chacun de ses facteurs est également non nul.

4.3. Exemple 3

Revenons sur l'exemple cité en introduction, tiré de la preuve concernant l'addition des dérivées :

```
Coq < Goal (f,g:(R->R); x0,x1:R)
    ``((f x1)-(f x0))/(x1-x0)+((g x1)-(g x0))/(x1-x0) ==
      ((f x1)+(g x1)-((f x0)+(g x0)))/(x1-x0)``.

Unnamed_thm < Intros. Time Field.
1 subgoal

  f : R->R
  g : R->R
  x0 : R
  x1 : R
  ============================
   ``x1+-x0 <> 0``
Finished transaction in 2 secs (2.17u,0s)
```

Il ne reste plus qu'à prouver que $x1 - x0 \neq 0$, ce qui est une hypothèse de notre lemme d'addition des dérivées.

4.4. Exemple 4

Nous nous intéressons ici aux preuves concernant les séries entières, utilisées pour définir les fonctions transcendantes (en cours de développement) telles que exp, sin ou cos.

Considérons, par exemple, l'application du critère de d'Alembert à la fonction exponentielle. Rappelons, avant tout, les définitions d'une série entière, de la fonction exponentielle ainsi que l'énoncé du critère de d'Alembert.

Une série entière réelle est une série de la forme : $\sum_{n=0}^{+\infty} a_n.x^n$

La fonction $\exp(x)$ peut se définir ainsi : $\sum_{i=0}^{+\infty} \frac{x^i}{i!}$

Une forme du critère de d'Alembert est : si $\left|\frac{a_{n+1}}{a_n}\right| \xrightarrow[n \to +\infty]{} 0$ alors $\exists l$ t.q. $\sum_{i=0}^{+\infty} a_n.x^n \ \to l$

Une manière de définir l'exponentielle est donc d'appliquer le critère de d'Alembert avec $a_n = \frac{1}{n!}$ et il faut alors montrer que $\left|\frac{\frac{1}{(n+1)!}}{\frac{1}{n!}}\right| \xrightarrow[n \to +\infty]{} 0.$

Dans ce but, nous pouvons montrer l'égalité suivante et l'appliquer ultérieurement dans la preuve avec $a = (S\ n)$ et $b = n!$:

$$\frac{\frac{1}{a.b}}{\frac{1}{b}} = \frac{1}{a}$$

```
Coq < Goal (a,b:R)``a <> 0``->``b <> 0``->``1/(a*b)/(1/b) == 1/a``.

Unnamed_thm < Intros. Time Field.
2 subgoals

  a : R
  b : R
  H : ``a <> 0``
  H0 : ``b <> 0``
  ============================
   ``b <> 0``

subgoal 2 is:
 ``a*b*(1/b*a) <> 0``
Finished transaction in 0 secs (0.79u,0s)
```

Nous remarquons la génération de deux nouveaux sous-buts. Conformément à l'algorithme utilisé, la première passe génère le sous-but 2 tandis que la seconde génère le sous-but 1. En effet, la première passe commence par multiplier par $a.b \times \frac{1}{b} \times a$ et, après simplifications, il reste alors un $\frac{1}{b}$. La seconde passe multiplie donc par b. Cela équivaut alors à montrer $a.b \neq 0$, $\frac{1}{b} \neq 0$, $a \neq 0$ et $b \neq 0$, ce qui se déduit des hypothèses.

4.5. Exemple 5

Enfin, nous pouvons donner l'exemple de la section 2.2, qui n'a pas de sens particulier mais qui est un bon test pour `Field` :

```
Coq < Goal (x,y:R)``x*(1/x+x/(x+y)) == -1/y*y*(-(x*x)/(x+y)-1)``.

Unnamed_thm < Intros. Time Field.
1 subgoal

  x : R
  y : R
  ============================
   ``x*((x+y)*y) <> 0``
Finished transaction in 1 secs (0.95u,0s)
```

4.6. Observation

À la vue de ces quelques exemples, la principale observation concerne le temps d'exécution de la tactique. En effet, sur certains exemples, nous pouvons constater des performances assez moyennes.

Après quelques tests rapides, nous avons éliminé des sources potentielles d'inefficacité et isolé quelques causes probables. La perte de temps a lieu au sein des fonctions Coq et n'est donc pas due au métalangage. Les fonctions chargées de distribuer tous les termes (exponentielle en le nombre de noeuds de l'expression distribuée) afin d'obtenir des monômes ainsi que la fonction de simplification des inverses semblent être les principales mises en cause. En particulier, l'algorithme de simplification des inverses, utilisant deux listes non triées, peut certainement être optimisé en effectuant un tri préalable.

Néanmoins, des tests plus poussés doivent être effectués, car l'optimisation de ces deux fonctions ne reduirait, au maximun, le temps d'exécution que d'environ 30%.

5. Conclusion

5.1. Synthèse

La tactique `Field` contribue grandement au développement de la théorie des nombres réels en Coq. Elle permet une économie de temps précieux ainsi qu'un gain non négligeable de concision dans les scripts de preuves. Par ailleurs, étant intégralement réflexive, elle permet la construction de termes preuves plus petits que dans l'approche directe en utilisant les réécritures. L'utilisateur peut maintenant se désintéresser de certaines parties de preuves comme il le ferait dans une preuve informelle. On peut voir ce travail comme s'inscrivant dans une optique plus globale qui est de créer, à terme, une tactique plus puissante capable de gérer aussi les inéquations comme le fait la tactique `Omega` pour les entiers naturels et relatifs.

D'un point de vue plus technique, `Field` est un bon test pour le nouveau langage de tactiques de la V7, non pas en ce qui concerne l'implantation mais plutôt le type de situations où il peut être utile. En regardant le cas de `Field`, il semblerait que les tactiques réflexives soient typiquement le genre de tactiques que l'on souhaite écrire à toplevel. En effet, seule la métaification nécessite l'utilisation, comme son nom l'indique, du métalangage. Étant particulièrment simple à concevoir et nécessitant du filtrage sur les termes Coq (d'un type non inductif), il est clair que la métaification tombe dans le contexte de ce langage de tactiques qui possède tous les opérateurs de filtrage nécessaires et dont l'objectif est d'automatiser des petites parties de preuves.

5.2. Travaux futurs

Dans un futur proche, nous pensons faire une étude un peu plus poussée des performances de la tactique. En effet, comme nous l'avons vu précédemment, certains cas mettent en évidence des sources d'inefficacité et il est important de savoir s'il est possible de les contrôler ou non.

Une extension facile de `Field` est de généraliser la tactique à tous les corps commutatifs, tout comme `Ring` peut s'appliquer à n'importe quel anneau abélien. Cette transformation peut s'effectuer exactement comme pour `Ring` et ne semble pas poser de problèmes supplémentaires[15].

Pour finir, il serait intéressant d'avoir une option similaire à celle de `Ring` où l'on peut lui donner un terme, lequel est normalisé et remplacé dans le but courant. On pourrait faire de même avec `Field` qui simplifierait tous les inverses du terme avant de le remplacer dans le but courant. Étant donné une égalité, on est sûr de pouvoir simplifier tous les inverses, mais, pour un terme, il se peut que des inverses ne se simplifient pas et le test d'arrêt de `Field` devra être différent dans ce cas.

Références

[1] Bruno Barras et al. *The Coq Proof Assistant Reference Manual Version 6.3.1.* INRIA-Rocquencourt, May 2000.
http ://coq.inria.fr/doc-eng.html.

[2] Samuel Boutin. *Réflexions sur les quotients.* PhD thesis, Université Paris 7, Avril 1997.

[3] G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Second GI Conference on Automata Theory and Formal Languages*, volume 33, pages 134–183. Springer-Verlag LNCS, 1976.

[4] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island*, volume 1955, pages 85–95. Springer-Verlag LNCS/LNAI, November 2000. ftp ://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/LPAR2000-ltac.ps.gz.

[5] D. Gabbay. The Undecidability of Intuitionnistic Theories of Algebraically Closed Fields and Real Closed Fields. In *Journal of Symbolic Logic*, volume 38, pages 86–92, 1973.

[6] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. Equational Reasoning via Partial Reflection. In *Proceedings of TPHOL.* Springer-Verlag, August 2000.

[7] John Harrison. Metatheory and Reflection in Theorem Proving : a Survey and Critique. Technical Report CRC-053, SRI Cambridge, UK, 1995.
http ://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz.

[8] John Harrison. *Theorem Proving with the Real Numbers.* Springer-Verlag, 1998.

[9] G. Kreisel and J.-L. Krivine. *Éléments de logique mathématique : théorie des modèles.* Dunod, 1964.

[10] Xavier Leroy et al. *The Objective Caml system release 3.00.* INRIA-Rocquencourt, April 2000.
http ://caml.inria.fr/ocaml/htmlman/.

[11] Micaela Mayero. The Three Gap Theorem (Steinhauss Conjecture). In *Proceedings of TYPES'99, Lökeberg.* Springer-Verlag LNCS, 1999. To appear.
ftp ://ftp.inria.fr/INRIA/Projects/coq/Micaela.Mayero/PS/three-gap.ps.tar.gz.

[15] D'un point de vue technique, cette abstraction s'effectue au moyen du mécanisme de sections de Coq. Tout le code peut être encapsulé dans une section où l'ensemble support, les constantes et les opérateurs s'y rapportant sont des paramètres. Le langage de tactiques s'adapte très bien à cette généralisation dans la mesure où il est également possible de filtrer sur des paramètres.

[12] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1951. Previous version published as a technical report by the RAND Corporation, 1948 ; prepared for publication by J. C. C. McKinsey.