Calcul de formules affines et de séries entières en arithmétique exacte avec types co-inductifs

Yves Bertot

Calcul de formules affines et de séries entières en arithmétique exacte avec types co-inductifs

Yves Bertot

INRIA Sophia Antipolis
bertot@inria.fr

Résumé

Nous décrivons une extension du travail de P. Di Gianantonio sur l'utilisation de types co-inductifs dans la description de librairies de calcul exact sur les nombres réels. Notre extension permet de représenter les séries formelles convergentes et de calculer sur ces représentations dans les systèmes de démonstration sur ordinateur qui disposent de réductions directes pour les fonctions structurelles récursives et les fonctions co-récursives gardées, comme le système Coq [14, 4, 16].

1. introduction

Nous reprenons les travaux de P. Di Gianantonio et A. Ciaffaglione [9] qui décrivent les nombres réels à l'aide de types co-inductifs et les opérations élémentaires sur ces nombres à l'aide de fonctions co-récursives. Nous rappelons briévement comment l'addition de deux nombres réels peut être représentées dans ce cadre, puis nous montrons que des opérations binaires plus générales, comme des opérations affines peuvent être programmées. Ces transformations plus générales ne reposent pas seulement sur le concept de co-récursion gardée, mais nécessitent de combiner co-récursion et récursion bien fondée de façon réminiscente des techniques proposées dans [12], mais avec une approche qui nous semble moins obscure. Nous montrons que la démonstration de correction de ces algorithmes est abordable dans le cadre des systèmes de preuve actuels.

Dans une deuxième partie, nous décrivons le calcul de sommes infinies. En particulier, nous décrivons la fonction co-récursive qui permet de calculer le nombre d'Euler e et nous donnons une esquisse de la preuve de correction pour ce calcul que nous avons vérifiée à l'aide du système de preuve Coq. Le résultat remarquable est que nous sommes alors capable de calculer le nombre e jusqu'à une précision assez grande (plusieurs dizaines de décimales) en utilisant seulement le mécanisme de réduction présent dans le système de preuve. En particulier, nous pouvons calculer la représentation décimale du nombre e jusqu'à une précision d'une centaine de décimales en moins d'une minute, même sans faire appel aux techniques de compilation proposées récemment [18].

Enfin, nous montrons que la multiplication de deux nombres se modélise également facilement comme le calcul d'une série entière, en utilisant la même infrastructure.

2. Travaux similaires

Dans les calculs numériques sur ordinateur, les nombres réels sont traditionnellement représenté sous forme approchée à l'aide de nombres à virgule flottante. Le terme "virgule flottante" signifie que l'on n'impose pas à priori le nombre de chiffres après la virgule, mais quand même que ce nombre de chiffre est fini, ce qui impose que les nombres représentables directement en machine sont en nombre

fini et que l'on doit arrondir les nombres réels pour trouver le nombre représentable en machine le plus proche. Pour cette raison, les calculs effectués par ordinateur et concernant des grandeurs réelles ne sont seulement que des approximations et les erreurs provenant des arrondis peuvent s'accumuler au point de rendre certains calculs grossièrement faux [25].

Les nombres à virgule flottante sont néanmoins très utiles : la majeure partie des processeurs fournissent directement l'implémentation des opérations de base (addition, soustraction, multiplication, division), en se référant à un standard qui permet de donner un sens mathématique très précis à la notion d'arrondi et qui permet aux programmeurs d'implémenter des calculs numériques avec une précision garantie [25, 20], parfois même avec des démonstrations de correction vérifiables à l'outils de preuve sur ordinateur [10, 21, 7, 30, 6].

Une direction de recherche alternative se concentre sur des calculs avec des représentations des nombres qui permettent d'éviter les erreurs d'arrondi. Parmi les approches possibles les plus connues sont basées sur les fractions continues [17, 31] ou les représentation avec nombres à virgule dans une base donnée [24]. Dans ce dernier cas, les représentations utilisées sont très proches des représentations en virgule flottante usuelle, sauf que le nombre de chiffres après virgule n'est pas figé pour une valeur donnée : on l'imagine alors comme une donnée infinie, qui n'est pas réellement présente dans l'ordinateur mais dont on connait déjà une partie finie, la partie infinie étant "matérialisée" au fur et à mesure que le besoin s'en fait sentir. On parle alors d'arithmétique réelle exacte. L'arithmétique exacte est étudiée assez largement [22, 26], en particulier dans le domaine de la programmation fonctionnelle [23, 5, 2, 15].

Lorsque l'on cherche à modéliser des calculs sur des structures de données infinies, les types co-inductifs [16] apparaissent comme un outil de choix. Les travaux précurseurs sur ce sujet sont ceux de di Gianantonio et Ciaffaglione [9] qui ont montré que l'on pouvait représenter les suites infinies de chiffres à l'aide de types co-inductifs et les opérations usuelles de l'arithmétique réelle (addition, multiplication, comparaison) à l'aide de fonction co-récursives simples. Niqui [28] fournit également une étude formelle assez proche, qui présente l'avantage d'aborder sous un même angle les deux approches de fractions continues et de séquences infinies de chiffres. Nos travaux utilisent une approche très similaire à celle Ciaffaglione et di Gianantonio, sauf que nous utilisons une collection de chiffre différents.

Puisque l'on attend d'elles qu'elles produisent des résultats infinis, les fonctions co-récursives sont habituellement des fonctions qui présentent une forme de récursion infinie. Pourtant, on préfère généralement éviter la récursion générale et l'on contraint les fonctions co-récursive à devoir produire un fragment de la structure infinie à chaque appel récursif. Cette contrainte semble interdire la programmation de fonctions récursives générales. Toutefois, nous avons montré dans [3] que la co-récursion pouvait être combinée avec des formes générales de récursion bien fondée pour définir des fonctions qui ne satisfont pas nécessairement cette contrainte. di Gianantonio et Miculan on été également proposé des approches pour assouplir cette contrainte [12, 13], mais les travaux présentés ici sont indépendants de leurs résultats.

3. Représentation redondante des nombres réels

Nous utilisons une représentation redondante des nombres réels compris entre 0 et 1, basés sur trois "chiffres" dont deux ont une signification intuitive proche des chiffres utilisés en représentation fractionnaire binaire :

- le chiffre `L` est utilisé comme le chiffre 0. Si x est une suite infinie de chiffres représentant le nombre v, `L`x représente le nombre $v/2$. En représentation binaire usuelle,
- le chiffre `R` est utilisé comme le chiffre 1. Si x est une suite infinies de chiffre représentant le nombre v, `R`x représente le nombre $v/2 + 1/2$.
- le chiffre `C` n'a pas de correspondant en représentation binaire usuelle. Si x est une suite infinie représentant le nombre v, alors `C`x représente le nombre $v/2 + 1/4$.

Dans la suite nous ferons l'abus de notation qui consiste à parler de la même manière d'une séquence de chiffres infinie et du nombre réel qu'elle représente. Nous assimilerons de même les chiffres à des fonctions. Par exemple, la fonction `L` est la fonction qui à x associe $x/2$.

Expliquons pourquoi le chiffre `C` est ajouté dans la représentation. En représentation binaire usuelle, les chiffres peuvent également être compris comme des informations d'intervalle. Un nombre binaire qui commence par 0.1 est forcément dans l'intervalle entre 1/2 et 1, tandis qu'un nombre qui commence par 0.01 est dans l'intervalle entre 1/4 et 1/2. Aucun nombre plus grand que 1/2 ne peut être représenté par une suite de chiffres commençant par 0.0 et aucun nombre plus petit que 1/2 ne peut être représenté par une suite commençant commençant par 0.1. Lorsque l'on additionne des nombres commençant par 0.010101010 et 0.001010101, on sait que la valeur obtenue est dans l'intervalle entre 0.011111111 et 0.100000001, mais on n'en sait pas assez pour déterminer le premier chiffre après la "virgule" (ici c'est un point car nous adoptons la notation anglo-saxonne) du résultat. Le chiffre `C` permet de résoudre ce problème. Le résultat de l'addition est donné avec assez de précision par la séquence de chiffres `CCCCCCCCC`.

Avec les nouveaux chiffres, un nombre qui s'écrit `CL`x s'écrit également `LR`x, et un nombre qui s'écrit `CR`x s'écrit également `RL`x, ce que l'on pourra vérifier aisément en utilisant les interprétations fonctionnelles des chiffres. Il sera donc possible d'enlever les utilisations du chiffre redondant `C` dans la représentation infinie d'un nombre, sauf pour les nombres utilisant une suite infinie de chiffres `C`. En particulier, le nombre `CC...` contenant une infinité de `C` et aucun autre chiffre représente le nombre 1/2, qui peut aussi être représenté par `LR...` avec un seul `L` et une infinité de `R` et `RL...` avec un seul `R` et une infinité de `L`.

3.1. Addition directe

Dans cette section nous décrivons un algorithme pour effectuer l'addition de deux nombres dont la représentation est donnée par une des séquences des trois chiffres `L`, `R`, et `C`.

Nous commençons par décrire la moyenne arithmétique de deux nombres (la demi-somme), puis nous utiliserons une multiplication par deux. Les résultats se justifient aisément par un raisonnement rapide sur les intervalles.

- Si l'on addition un nombre `L`x' avec un autre qui commence par `L`y', on est sur que le résultat est dans [0,1] et on peut affirmer que ce résultat est dans [0,1/2] et le résultat peut être de la forme `L`$(\frac{x'+y'}{2})$. On programme donc ce cas par un appel r'ecursif sur les s'equences en queue de x et y.
- Le raisonnement précédent est encore valable si l'on remplace toutes les occurrences de `L` par `R` et [0,1/2] par [1/2,1] dans le paragraphe précédent, puis avec `C` et [1/4,3/4],
- Si l'on addition un nombre `L`x' avec un nombre `R`y', alors le résultat est `C`$(\frac{x'+y'}{2})$
- Dans tous les autres cas, il faut observer le deuxième chiffre d'au moins l'un des deux nombres additionnés, il suffit souvent d'appliquer les équivalences `LR` $\sim$ `CL` et `RL` $\sim$ `CR` pour se ramener à un cas déjà étudié.
- Il reste alors peu de cas difficiles dont un cas représentatif est la demi-somme de `CC`x'' et `LL`y''. Dans ce cas, on peut remarquer `CC`x'' représente le même nombre que `LC`x'' plus 1/4 et que `LL`y'' représente le même nombre que `LR`y'' moins 1/4. Il est donc possible d'emprunter à l'un des arguments pour ajouter à l'autre, pour se ramener à un cas facile.

Le principe de l'algorithme de demi-somme est ainsi exprimable en quelque lignes, en pratique il existe 25 cas différents. Une particularité amusante de notre développement est que l'implémentation de la fonction d'addition est effectué par une recherche automatique à l'aide des outils de preuves, d'une manière qui exprime les cas ci-dessus : nous ne donnons pas directement l'algorithme, nous laissons le système de preuve le trouver en le guidant par des vérifications arithmétiques.

Pour la multiplication par 2, l'interprétation des opérations est plus directe. Nous ne décrivons que

des nombres dans l'intervalle [0,1], et la multiplication par 2 n'est valide que si l'argument est inférieur à 1/2.

- puisque `L` représente une division par 2, multiplier par 2 un nombre dont la représentation commence par `L` consiste en la suppression de ce `L`,
- puisqu'un nombre qui commence par `R` est supérieur à 1/2, la seule valeur significative possible est 1 (qui est la bonne valeur si l'argument représente exactement 1/2),
- le double d'un nombre $\mathtt{C}x'$ peut être représenté par $\mathtt{R}(x' \times 2)$, il y a donc un appel récursif.

La combinaison de ces deux fonctions nous fournit l'addition qui n'est valide que si la somme des deux arguments est inférieure à 1.

On peut croire que l'addition permet directement d'implémenter la multiplication, en utilisant les remarques suivantes :

- $(\mathtt{L}x) \times y = \mathtt{L}(x \times y)$,
- $(\mathtt{R}x) \times y = \mathtt{L}(x \times y) + (\mathtt{L}y)$,
- $(\mathtt{C}x) \times y = \mathtt{L}(x \times y) + (\mathtt{LL}y)$.

Mais dans cette approche, l'appel récursif à la multiplication apparait comme argument d'une opération d'addition et ne respecte pas la discipline élémentaire de la programmation co-récursive (les appels récursifs ne sont pas gardés par des constructeurs). La solution de ce problème est déjà décrite dans les travaux [9], en programmant l'opération plus générale qui calcule $x \times y + z$. Dans ce travail nous traiterons cette opération comme un cas particulier de calcul sur les séries entières.

3.2. Modélisation et vérification

Il existe plusieurs approches vers une arithmétique réelle exacte, qui varient par leur adhérence aux principes des mathématiques constructives. La première approche est de ne supposer que l'existence de nombre réels et de nombres rationnels, et de montrer que les nouveaux nombres représenter forment une complétion de l'ensemble des nombres rationnels qui respecte les opérations de base. C'est l'approche utilisée dans [9] La seconde approche est de supposer que l'ensemble des nombres réels existe et de réutiliser les différents résultats déjà établis ou postulés pour ces nombres. C'est cette approche, probablement débatable pour les puristes, que nous avons choisie. Ceci permet de se placer dans un cadre plus proche des mathématiques usuelles et donc de mieux utiliser nos capacités humaines de raisonnement.

Par ailleurs, il s'agit de représenter des structures infinies, en fait des listes infinies de chiffres. Pour ce type de représentation le cadre de choix est fourni par les types co-inductifs [16, 29]. De tels types co-inductifs sont disponibles dans certains systèmes de démonstration comme Coq [14, 4] ou Isabelle [27]. Notre expérience est effectuée avec Coq. Nous proposons de représenter les listes infinies polymorphes avec un type `stream` à un seul constructeur `Cons` et d'attacher une notation infixe à ce constructeur, ce qui se fait par les commandes suivantes :

```
CoInductive stream (A:Set) : Set :=
  Cons : A -> stream A -> stream A.

Implicit Arguments Cons.
Infix "::" : Cons : stream_scope.
Open Scope stream_scope.
```

Le type de chiffres que nous utilisons est un type énuméré :

```
Inductive idigit : Set := L | R | C.
```

La valeur 1 s'écrit aisément dans ce type :

```
Cofixpoint one : stream idigit := R::one.
```

La fonction de multiplication par 2 s'écrit alors de la façon suivante :

```
Cofixpoint mult2 (n:stream idigit) : stream idigit :=
  match n with L x => x | C x => R::mult2 x | R x => one end.
```

Pour vérifier la correction des fonctions que nous proposons, nous utilisons une propriété co-inductive pour mettre en correspondance les séquences infinies de chiffres et les nombres réels.

```
CoInductive represents: stream idigit -> Rdefinitions.R -> Prop:=
  reprL : forall s r, represents s r -> (0 <= r <= 1)%R ->
           represents (L::s) (r/2)
| reprR : forall s r, represents s r -> (0 <= r <= 1)%R ->
           represents (R::s) ((r+1)/2)
| reprC : forall s r, represents s r -> (0 <= r <= 1)%R ->
           represents (C::s) ((2*r+1)/4).
```

Une approche alternative pour décrire cette correspondance est d'associer à toute séquence de chiffres une suite d'intervalles correspondant aux préfixes finis de cette séquence. Ceci est fourni par une fonction `bounds` s n qui retourne un triplet (a, b, k) tel que le préfixe de longueur n de s représente l'intervalle $[a/2^k, b/2^k]$, par exemple `bounds (L::R::C::`s`) 2` vaut (1, 2, 2) parce que `LR` représente l'intervalle [1/4,1/2]. Puis nous considérons la suite des bornes inférieures de ces intervalles, nous montrons que cette suite est croissante et bornée, a une limite et nous obtenons une fonction `real_value` de type `stream idigit -> R`. Nous avons également modélisé cette approche. En particulier nous avons prouvé les théorèmes suivants :

```
Theorem represents_real_value : forall s, represents s (real_value s).

Theorem represents_equal : forall s r, represents s r -> real_value s = r.
```

L'énoncé qui exprime la correction de la multiplication par 2 prend la forme suivante :

```
Theorem mult2_correct :
  forall x u, (0 <= u <= 1/2)%R -> represents x u -> represents (mult2 x)(2*u)%R.
```

L'énoncé fait bien apparaitre la condition que le nombre multiplié par deux soit assez petit pour établir la propriété voulue sur le résultat. La démonstration repose sur une preuve par co-récurrence, et utilise les techniques proposées dans [4].

Pour l'addition, nous ne décrivons pas directement l'algorithme de cette fonction, mais nous faisons construire la représentation de cet algorithme par des fonctions de recherche de preuve. Le pricipe est le suivant : si l'on sait que l'algorithme doit effectuer un traitement par cas sur les données en entrée et que l'un de ces cas concerne l'addition d'une séquence de la forme `C::L::`x avec une séquence de la forme `L::`y, le système peut effectuer lui-même le calcul des bornes supérieurs des intervalles correspondant à `C::L` et `L` et leur demi-somme pour s'apercevoir que le résultat est nécessairement inférieur à 1/2, ce qui permet de déterminer que l'algorithme peut reposer sur une expression de la forme `L::(half_sum` a b`)`. Les arguments a b peuvent eux-mêmes être recherchés parmi les différentes combinaisons sur x et y, éventuellement préfixés par un chiffre, pour trouver l'une de combinaisons qui permettent de prouver l'égalité recherchée. Bien que l'algorithme soit obtenu par une technique de preuve, il ne contient aucun objet preuve. Cette propriété est importante car elle permet l'utilisation de cet algorithme dans des fonctions qui sont réductibles dans le système de preuve lui-même, ce qui est intéressant pour les démonstration par réflexion [8]. Faute de place, nous ne décrivons pas plus en détail cette partie de notre travail, mais le lecteur intrigué pourra se reporter au développement disponible sur internet[1].

La démonstration que la fonction de demi-somme est correcte donne le théorème suivant :

```
Theorem half_sum_correct :
  forall x y u v, represents x u -> represents y v ->
  represents (half_sum x y) ((u + v)/2).
```

[1] Voir `http ://www-sop.inria.fr/marelle/Yves.Bertot/proofs.html`.

Cette démonstration couvre les 25 cas de l'algorithme, les démonstrations se ramènent aisément à des vérifications arithmétiques élémentaires de formules fractionnaires affines[2] où seulent les constantes 2 et 4 apparaissent en dénominateur. Certaines de vérifications sont des égalités, que nous pouvons résoutre aisément avec une tactique comme `field` [11], les autres sont des inégalités que nous pouvons résoudre avec la tactique `fourier`.

4. Formules affines

Nous nous intéressons maintenant à une fonction plus générale que l'addition, qui combine deux valeurs réelles dans une formule affine. Il s'agit de calculer la valeur de

$$\frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'}$$

Lorsque les nombres a, b, c sont des nombres positifs et les nombres a', b', c' sont des nombres positifs non-nuls, et les nombres x et y sont représentés par des séquences infinies de chiffres. Cet algorithme est intéressant car il nous amène à mettre en œuvre une technique particulière pour combiner la co-récursion et la récursion bien fondée.

4.1. Principes de l'algorithme

La détermination des chiffres du résultat se base sur les remarques suivantes :

1. si l'on ne sait rien de plus sur x et y, on sait quand même que ces deux nombres sont compris entre 0 et 1, les valeur extrémales du résultat sont alors

$$\frac{c}{c'} \qquad \text{et} \qquad \frac{ab'c' + a'bc' + abc'}{a'b'c'}.$$

2. dès que la différence entre ces deux valeurs est strictement inférieure à $1/2$, il est possible de déterminer que le résultat est bien compris dans l'un des intervalles déterminés par `L`, `R`, et `C`, on peut alors produire le bon chiffre et effectuer un appel récursif pour une nouvelle formule affine,
3. si les valeurs des coefficients ne permettent de pas conclure, on peut observer le premier chiffre de x et y, ceci amène à diviser la distance entre valeur maximale et minimale par 2 : on peut donc observer les premiers chiffres de x et y, puis faire un appel récursif sur les sous-séquences x' et y', encore avec une nouvelle formule affine.

L'appel récursif décrit à l'étape 2 correspond à une utilisation de la co-récursion car l'appel récursif est bien gardé par un constructeur. En revanche, l'appel récursif décrit à l'étape 3 n'est pas inclus dans la production d'un premier chiffre. Cette récursion n'est acceptable que parce que nous pouvons exhiber un argument de récursion bien fondée : comme la longueur de l'intervalle des valeurs possibles est divisée par 2 à chaque appel récursive, cette longueur doit forcément finir par être plus petite que $1/2$.

Donnons deux exemples. Pour le premier exemple, supposons que $c/c' \geq 1/2$ alors nous savons que le résultat est supérieur à $1/2$ et nous pouvons faire le calcul suivant :

$$\begin{aligned} \frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'} &= \mathtt{R}(2 \times (\frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'}) - 1) \\ &= R(\frac{2a}{a'}x + \frac{2b}{b'}y + \frac{2 * c - c'}{c'}) \end{aligned}$$

[2] le produit de deux variables n'apparait jamais.

On voit bien que la production du chiffre `R` s'accompagne d'un appel récursif avec une nouvelle formule affine, dont tous les coefficients sont positifs dès que la condition $c/c' \geq 1/2$ est bien satisfaite.

Pour le deuxième exemple, supposons les égalités $x = \mathtt{L}x'$ et $y = \mathtt{R}y'$, alors on a le calcul suivant :

$$\begin{aligned} \frac{a}{a'}x + \frac{b}{b'}y + \frac{c}{c'} &= (\frac{a}{a'}\frac{x'}{2} + \frac{b}{b'}\frac{y'+1}{2} + \frac{c}{c'} \\ &= \frac{a}{2a'}x' + \frac{b}{2b'}y' + \frac{bc' + 2b'c}{2b'c'} \end{aligned}$$

Ici encore, on retrouve une nouvelle formule affine, avec des arguments différents, dont les coefficients sont positifs si les arguments initiaux étaient positifs. La distance entre valeurs extrémales de la formule affine initiale est $a/a' + b/b'$ et la distance entre valeurs extrémales de la nouvelle formule affine est $a/2a' + b/2b'$, soit la moitié de la distance précédente.

4.2. Modélisation et vérification formelle

Nous définissons une fonction qui prend en entrée une structure de donnée représentant la formule affine, il y a 6 nombres entiers relatifs et 2 séquences infinies de chiffres. Ces arguments sont regroupés dans une structure de donnée que nous nommons `affine_data`.

```
Record affine_data : Set :=
 {m_a : Z; m_a' : Z; m_b : Z; m_b' : Z; m_c : Z; m_c' : Z;
  m_x : stream idigit; m_y : stream idigit}.
```

En utilisant, la fonction `real_value` décrite précédemment, nous pouvons associer à cette structure de donnée la valeur entière qu'elle est censée représenter. La fonction `af_real_value` remplit ce rôle.

Nous définissons un prédicat `positive_coefficients` sur cette structure de donnée qui exprime que les entiers relatifs sont tous positifs et que les nombres représentant des dénominateurs sont non-nuls. Nous appelons `axbyc` la fonction qui calcule la formule affine, elle a le type suivant :

```
axbyc : forall x: affine_data, positive_coefficients x -> stream idigit.
```

La fonction `axbyc` contient une fonction auxiliaire, que nous avons nommée `axbyc_rec`. Cette fonction se charge de répéter les opérations décrite en phase 3 dans la section précédente. Il s'agit d'une fonction récursive bien fondée, elle retourne une valeur dans un nouveau type de données. Ce nouveau type de données combine une structure `affine_data`, une preuve que cette structure a tous ses coefficients positifs, une preuve que ces coefficients permettent l'émission de l'un des chiffres, et une preuve que la structure retournée a la même valeur par la fonction `af_real_value` que l'argument initial. Nous appelons cette nouvelle structure de données `decision_data`. Elle décrite par un type inductif à trois constructeurs nommés `caseR`, `caseL`, `caseC`. Le type de la fonction `axbyc_rec` est le suivant :

```
axbyc_rec : forall x, positive_coefficients x -> decision_data x
```

Avec cette fonction, il est possible d'écrire la fonction principale en quelques lignes :

```
CoFixpoint axbyc (x:affine_data)
   (h:positive_coefficients x):stream idigit :=
  match axbyc_rec x h with
    caseR y Hpos Hc _ =>
     R::(axbyc (prod_R y) (A.prod_R_pos y Hpos Hc))
  | caseL y Hpos _ _ =>
     L::(axbyc (prod_L y) (A.prod_L_pos y Hpos))
  | caseC y Hpos H1 H2 _ =>
     C::(axbyc (prod_C y) (A.prod_C_pos y Hpos H2))
  end.
```

Les fonctions `prod_R`, `prod_L`, `prod_C` effectuent la transformation de la formule affine qui correspond au chiffre produit. Les théorèmes `..._pos` fournissent les preuves que les appels récursifs se font bien avec des structures contenant des coefficients positifs.

La correction de la fonction `axbyc` s'exprime par l'énoncé suivant :

```
axbyc_correct:
  forall x, forall H :positive_coefficients x,
   (0 <= af_real_value x <= 1)%R -> real_value (axbyc x H) = af_real_value x.
```

Bien que nous ayons exprimé ce théorème par une égalité entre des valeurs réelles, nous avons effectué une preuve par co-récursion reposant sur la propriété `represents` et utilisé le théorème `represents_equal` obtenir l'égalité.

5. Calcul de séries entières

Nous avons ensuite abordé le problème de programmer le calcul de séries entières. Ce travail nous permet de fournir une représentation de e (le nombre d'Euler) et de réimplémenter la multiplication de façon similaire à la multiplication de [9].

5.1. Approche générale

Nous décrivons ici une technique pour calculer les séries entières dont la valeur est comprise entre 0 et 1. Les séries entières sont les nombres définis par les sommes $\sum_{i=0}^{\infty} a_i$, lorsque ces sommes sont convergentes. Leur étude est très proche de l'étude des suites convergentes, car il suffit de considérer la suite $u_n = \sum_{i=0}^{n} a_i$. Néanmoins, les séries présentent un avantage pratique certain car elle permettent d'utiliser une somme partielle (jusqu'à un n donné) pour donner une approximation du nombre cherché.

Notre approche repose sur le fait qu'une séquence infinie de chiffres correspond à une approximation de précision croissante du nombre cherché. Pour calculer la valeur d'une somme infinie à une précision faible, il peut être suffisant de n'observer que les premiers termes de la somme. Lorsque la précision augmente, il est nécessaire de prendre de plus en plus termes de la somme en compte.

Pour une série donnée $\sum_{i=0}^{\infty} a_i$, nous définissons une fonction f qui doit satisfaire la spécification informelle suivante :

$$f(x, y, n) = x + y \times \sum_{i=n}^{\infty} a_i.$$

Nous supposons généralement l'existence d'une fonction μ, telle

$$|\forall m.n \leq m \Rightarrow \sum_{i=m}^{\infty} a_i| < \mu(n).$$

Le calcul procède ensuite de la manière suivante :

1. On calcule une valeur $\phi(y, n)$, supérieure ou égale à n, telle que $y \times (\phi(y, n))$ soit majoré par $\frac{1}{16}$,
2. On calcule ensuite le nombre $v = x + y \times \sum_{i=n}^{\phi(y,n)-1} a_i$ par additions successives, et on effectue un traitement par cas sur ce nombre :
3. si v est de la forme $\mathtt{RR}v'$, $\mathtt{RC}v'$, $\mathtt{RLC}v''$, ou $\mathtt{RLR}v''$, alors on est certain que $v + \sum_{i=\phi(y,n)}^{\infty} a_i$ est supérieur à $1/2$ et l'on sait que le résultat peut avoir la forme $\mathtt{R}(f(\mathtt{R}v', 2y, \phi(y, n)))$, $\mathtt{R}(f(\mathtt{C}v', 2y, \phi(y, n)))$, $\mathtt{R}(f(\mathtt{LC}v', 2y, \phi(y, n)))$, $\mathtt{R}(f(\mathtt{LR}v', 2y, \phi(y, n)))$, respectivement.
4. si v est de la forme $\mathtt{CC}v'$, $\mathtt{CLC}v''$, $\mathtt{CLR}v''$, $\mathtt{LL}v'$, $\mathtt{LC}v'$, $\mathtt{LRL}v''$, $\mathtt{LRC}v''$ alors ce cas peut être traité de façon similaire au cas précédent, en produisant un résultat dont le premier chiffre est le même que le premier chiffre de v et en effectuant un appel récursif à la fonction f avec v privé de son premier chiffre, $2y$, et $\phi(y, n)$ comme arguments.

5. Si v est de la forme RLLv'', alors on peut observer que v est aussi représentable par la séquence CRLv'' et cette séquence est déjà traitée dans les cas précédents. On effectue un traitement similaire pour les cas LRRv'', CLLv'' et CRRv'', qui sont équivalents aux cas CLRv'', LRLv'', et RLRv'' respectivement.

Le calcul de $\phi(y, n)$ et de la somme $\sum_{i=n}^{\phi(n)-1}$ devra généralement se faire par une fonction récursive, probablement une fonction récursive bien-fondée, dont la terminaison est assurée par la convergence de la série. De plus, à chaque appel récursif, les arguments y et n satisfont l'invariant que $y \times \mu(n) < 1/8$. Cet invariant peut permettre d'éviter que le calcul de $\phi(y, n)$ soit complexe. En outre l'argument y est seulement multiplié par 2 à chaque appel récursif.

5.2. Séries positives

Lorsque l'on sait que les éléments a_i de la somme infinie sont tous positifs, il n'est pas nécessaire d'utiliser $\frac{1}{16}$ comme majorant, $\frac{1}{8}$ suffit. La technique de calcul repose alors sur les étapes suivantes :

1. On calcule une valeur $\phi(y, n)$, supérieure ou égale à n, telle que $y \times (\phi(y, n))$ soit majoré par $\frac{1}{8}$,
2. On calcule le nombre $v = x + y \times \sum_{i=n}^{\phi(y,n)-1} a_i$ et on effectue le traitement par cas suivant :
3. si v est de la forme Rv', alors on est certain que le résultat est supérieur à $1/2$, le résultat est R$(f(v', 2y, \phi(y, n)))$,
4. si v est de la forme Cv', mais pas de la forme CRv', alors le résultat est C$(f(v', 2y, \phi(y, n)))$,
5. si v est de la forme Lv', mais pas de la forme LRv', alors le résultat est L$(f(v', 2y, \phi(y, n)))$,
6. si v est de la forme CR ou LR, on peut utiliser les équivalences entre CR et RL d'une part et LR et CL d'autre part pour se ramener à un cas déjà traité.

Le majorant $\frac{1}{8}$ est suffisant parce que c'est la distance entre la borne supérieure de l'intervalle correspondant à CC et l'intervalle correspondant à C. Le traitement correspondant aux cas 3 à 6 ci-dessus est systématique et ne dépend pas de la série considérée. Nous l'avons programmé dans une fonction d'ordre supérieur qui peut être réutilisée d'une série à l'autre.

```
Definition series_body (A:Set)
  (f : stream idigit -> A -> stream idigit)
  (x : stream idigit) (a:A) : stream idigit :=
 let (dig,x'') := x in
 match dig with
   R =>  R::f x'' a
 | L => match x'' with R::x3 => C::f (L::x3) a | _ => L::f x'' a end
 | C => match x'' with R::x3 => R::f (L::x3) a | _ => C::f x'' a end
  end.
```

L'argument a doit contenir les information suffisantes pour retrouver les valeurs y et $\phi(y, n)$.

5.3. Application au calcul de e

Nous considérons que le nombre e est défini par la formule

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

Bien sûr ce nombre est supérieur à 2, mais nous voulons seulement obtenir sa partie fractionnaire, de telle sorte que nous nous contentons de calculer $\sum_{k=2}^{\infty} \frac{1}{k!}$. Un calcul rapide permet d'établir les

inégalités suivantes, pour n'importe quel n.

$$0 < \sum_{k=n}^{\infty} \frac{1}{k!} < \frac{1}{(n-1)!(n-1)}$$

En effet, pour $n \leq k$, nous avons $n!n^{k-n} < k!$, donc

$$\sum_{k=n}^{\infty} \frac{1}{k!} < \frac{1}{n!} \sum_{i=0}^{\infty} \frac{1}{n^i} = \frac{1}{n!} \frac{n}{n-1}$$

Appelons $\nu(n)$ le membre droit de cette inégalité. Dès que $2 < n$, nous savons que $\nu(n+1) < \frac{\mu(n)}{2}$. Ceci implique la propriété suivante :

$$\forall n, y. 0 < y \wedge 2 < n \wedge \phi(y, n) < \frac{1}{4} \Rightarrow \phi(y, n+1) < \frac{1}{8}.$$

Il n'est donc jamais nécessaire d'absorber plus d'un terme de la somme infinie dans l'argument x à chaque appel co-récursif.

Au lieu de définir une fonction à trois arguments, nous définissons une fonction à 4 arguments. Le quatrième argument est simplement un pré-calcul de la factorielle de $n-1$. La recherche de $\phi(y, n)$ se ramène à un simple calcul pour comparer $\frac{1}{(n-1)!(n-1)}$ avec $\frac{1}{8}$, une comparaison que nous ramenons à une comparaison entre deux entiers. Si la comparaison indique qu'il faut absorber un élément de plus dans l'accumulateur x, nous faisons cette opération en additionnant $\frac{y}{n!}$ à x en utilisant l'addition directe que nous avons définie précédemment et en utilisant une fonction `rat_to_stream` qui prend deux arguments entiers et fabrique la représentation du rapport entre ces deux nombres.

La fonction co-récursive a la forme suivante :

```
CoFixpoint e_series (v:stream idigit)(s :Z*Z*Z) :stream idigit :=
  let (aux, fact_nm1) := s in let (y, n) := aux in
  let (v', n', fact_nm1') :=
     if Z_le_gt_dec (8*y) (fact_nm1*(n-1))%Z then
        mk_triple v n fact_nm1
     else
        mk_triple (v + (rat_to_stream y (fact_nm1 * n)))
          (n+1) (fact_nm1*n) in
     series_body  _ e_series v' (2*y, n', fact_nm1')%Z.
```

Nous avons démontré que cette fonction co-récursive fournissant une bonne représentation de la somme infinie en démontrant le théorème suivant :

```
Theorem e_correct1 :
 forall v vr r n p fact_pm1,
   fact_pm1 = (Z_of_nat (fact (Zabs_nat (p-1)))) ->
   4 * n <= fact_pm1 * (p-1) -> 2 <= p -> 1 <= n ->
   represents v vr ->
   infinit_sum (fun i => (1/INR(fact (i+Zabs_nat p)))%R) r ->
   (vr + (IZR n)*r <= 1)%R ->
   represents (e_series v n p fact_pm1) (vr+(IZR n)*r)%R.
```

Dans cet énoncé, la formule

```
   infinit_sum (fun i => (1/INR(fact (i+Zabs_nat p)))%R) r
```

signifie "`r` est la valeur la somme infinie $\sum_{i=0}^{\infty} \frac{1}{(i+p)!}$". Nous pouvons maintenant définir les valeurs suivantes

```
Definition head := fast_add (rat_to_stream 1 2)(rat_to_stream 1 6).

Definition number_e_minus2 : stream idigit :=
  e_series head 1 4 6.
```

Nous pouvons ensuire déduire du théorème de correction l'énoncé suivant :

```
Theorem e_correct :
  forall r,
  infinit_sum (fun i => (1/INR(fact(i+Zabs_nat 2)))) r ->
  represents number_e_minus2 r.
```

Ce qui exprime que `number_e_minus2` est bien une représentation de $\sum_{i=2}^{\infty} \frac{1}{i!}$.

5.4. Multiplication

L'idée intuitive de la multiplication est que si u est la suite de chiffres $d_0::d_1::\ldots$ et si α est la fonction telle que $\alpha(\texttt{L}) = 0$, $\alpha(\texttt{C}) = \frac{1}{4}$ et $\alpha(\texttt{R}) = \frac{1}{2}$, alors uu' est la série entière

$$uu' = \sum_{i=0}^{\infty} \frac{\alpha(d_i)u'}{2^i}.$$

Il s'agit donc de faire la somme infinie de termes a_i définis par :

$$a_i = \frac{\alpha(d_i)u'}{2^i}.$$

Deux simplifications apparaissent vis-à-vis de l'approche générale. D'une part y est multiplié par 2 à chaque appel récursif, tandis que a_i contient un diviseur qui est également multiplié par 2 à chaque appel récursif. D'autre part, il est raisonnable de simplement consommer un élément de la somme à chaque appel récursif, sans tenir compte de la valeur de u'. Si cette approche est suivie, il n'est plus nécessaire de passer l'argument y de la présentation générale qui ne sert à rien et seules la séquences des d_i à partir d'un certain rang et le nombre u' sont nécessaires. Nous proposons donc de réutiliser la fonction `series_body` de la façon suivante :

```
CoFixpoint mult_a (x:stream idigit) (p:stream idigit*stream idigit)
  : stream idigit :=
  let (u,v) := p in
  match u with
    L::u' => series_body _ mult_a x (u',v)
  | C::u' => series_body _ mult_a (x+(L::L::v)) (u',v)
  | R::u' => series_body _ mult_a (x+(L::v)) (u',v)
  end.
```

La fonction `mult_a` x (u, u') calcule donc la valeur $x + uu'$ dès que $uu' < \frac{1}{4}$. Pour définir une fonction qui calcule correctement le produit de u et u', nous commençons par diviser le deuxième facteur par 4, puis nous multiplions le résultat par 4. L'implémentation naive a donc la forme suivante :

```
Definition mult (x y:stream idigit) : stream idigit :=
  mult2(mult2(mult_a zero (x,L::L::y))).

Infix "*" := mult : stream_scope.
```

Nous avons également démontré la correction de cette fonction, qui s'exprime par le théorème suivant :

```
mult_correct
     : forall (x y : stream idigit) (vx vy : Rdefinitions.R),
       represents x vx -> represents y vy -> represents (x * y) (vx * vy)
```

6. Conclusion

Notre expérience souffre d'un défaut de naissance. Nous avons formalisé les nombres réels en nous réduisant aux nombres compris entre 0 et 1, mais pour considérer une extension raisonnable, il faut pouvoir considérer l'ensemble complet des nombres réels. Il est bien sûr possible de passer à la droite réelle en multipliant un nombre compris entre 0 et 1 par un nombre entier, mais il faut bien faire attention de garder la propriété de redondance sur toute la ligne : multiplier par un nombre entier ne suffit pas car alors on ne dispose pas de moyen simple de représenter un intervalle autour de zéro. Il faudrait donc considérer les nombres de la forme $a + bx$, où a et b sont des nombres entiers et x une séquence infinie de chiffres.

L'approche usuelle est plutôt de partir de la représentation binaire, comme nous l'avons fait, mais d'ajouter un troisième chiffre qui s'interprète comme un le chiffre -1. Dans ce cas, on fournit une représentation pour tous les nombres compris entre -1 et 1, et on peut obtenir l'ensemble des nombres réels en multipliant par une puissance de 2. Nous pensons que cela devrait être un travail aisé de reprendre la formalisation décrite ici pour l'adapter à cet autre ensemble de chiffres. On peut d'ailleurs également envisager d'utiliser d'autres systèmes de avec un plus grand nombre de chiffres, par exemple en prenant tous les nombres représentables comme entiers signés dans les langages de programmation usuels. Toutefois, il est certain qu'une telle généralisation introduira une complexité supplémentaire dans la démonstration de correction de toutes les opérations.

Maintenant que nous disposons d'une multiplication entre nombres réels, il est naturel d'envisager de programmer les fonctions analytiques, la division, les fonctions exponentielles et trigonométriques, et la recherche de racines de fonctions continues dérivables par la méthode de Newton.

Il est également possible de généraliser le travail sur les transformations affines en considérant également les transformations de Möbius de la forme suivante :

$$\frac{axy + bx + cy + d}{exy + fx + gy + h}.$$

Le traitement de ces transformations contient encore des phases de production de chiffres et des phases d'observation des chiffres provenant des arguments.

Le travail sur les tranformations affines et son extension sur les transformations de Möbius ne mène pas à l'implémentation d'une librairie de calcul exact que l'on peut utiliser directement à l'intérieur du système de démonstration. En revanche, l'addition directe, et les calculs de séries peuvent souvent s'exécuter dans le système de démonstration lui-même. Le bénéfice de ce travail est donc de permettre l'utilisation de calculs exacts sur les nombres réels pour développer de nouvelles procédure de décision basées sur le principe des preuves à deux étages [1] ou preuves par réflexion [8]. On doit toutefois éviter un enthousiasme excessif : les calculs effectués par ces algorithmes dans le système de démonstration sont effroyablement inefficaces. Il serait quand même intéressant de savoir si notre travail peut contribuer au développement de tactiques de comparaison pour des formules mathématiques arbitraires, comme dans les travaux de R. Zumkeller sur les systèmes d'inéquations.

Références

[1] Gilles Barthe, Mark Ruys, and Henk Barendregt. A two-level approach towards lean proof-checking. In *TYPES '95 : Selected papers from the International Workshop on Types for Proofs and Programs*, pages 16–35, London, UK, 1996. Springer-Verlag.

[2] A. Bauer, M.H. Escardó, and A. Simpson. Comparing functional paradigms for exact real-number computation. In *Automata, languages and programming*, volume 2380 of *Lecture Notes in Comput. Sci.*, pages 489–500. Springer, 2002.

[3] Yves Bertot. Filters on coinductive streams, an application to eratosthenes' sieve. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications, TLCA 2005*, pages 102–115. Springer-Verlag, 2005.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art :the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[5] Hans-Juergen Boehm, Robert Cartwright, Mark Riggle, and Michael J. O'Donnell. Exact real arithmetic : A case study in higher order programming. In *LISP and Functional Programming*, pages 162–173, 1986.

[6] Sylvie Boldo. *Preuves formelles en arithmétiques àvirgule flottante*. PhD thesis, École Normale Sup'erieure de Lyon, November 2004.

[7] Sylvie Boldo, Marc Daumas, Claire Moreau-Finot, and Laurent Théry. Computer validated proofs of a toolset for adaptable arithmetic. *Journal of the ACM*, 2002. Submitted.

[8] S. Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281. LNCS, Springer-Verlag, 1997.

[9] Alberto Ciaffaglione and Pietro Di Gianantonio. A coinductive approach to real numbers. In Th. Coquand, P. Dybjer, B. Nordström, and J. Smith, editors, *Types 1999 Workshop, Lökeberg, Sweden*, number 1956 in LNCS, pages 114–130. Springer-Verlag, 2000.

[10] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics : 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 169–184. Springer-Verlag, September 2001.

[11] David Delahaye and Micaela Mayero. Field : une procédure de décision pour les nombres réels en coq. In *Proceedings of JFLA'2001*. INRIA, 2001.

[12] Pietro di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*, pages 148–161. Springer Verlag, 2003.

[13] Pietro di Gianantonio and Marino Miculan. Unifying recursive and co-recursive definitions in sheaf categories. In Igora Walukiewicz, editor, *Foundations of Software Science and Computation Structures (FOSSACS'04)*, volume 2987 of *LNCS*. Springer Verlag, 2004.

[14] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. *The Coq Proof Assistant User's Guide*. INRIA, May 1993. Version 5.8.

[15] Abbas Edalat and Peter John Potts. A new representation for exact real numbers. In Stephen Brookes and Michael Mislove, editors, *Electronic Notes in Theoretical Computer Science*, volume 6. Elsevier Science Publishers, 1998.

[16] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for proofs and Programs*, volume 996 of *LNCS*, pages 39–59. Springer Verlag, 1994.

[17] Ralph W. Gosper. HAKMEM, Item 101 B. `http://www.inwap.com/pdp10/hbaker/hakmem/cf.html#item101b`, feb. 1972. MIT AI Laboratory Memo No.239.

[18] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.

[19] Tanja Grubba, Peter Hertling, Hideki Tsuiki, and Klaus Weihrauch, editors. *CCA 2005 - Second International Conference on Computability and Complexity in Analysis, August 25-29, 2005, Kyoto, Japan*, volume 326-7/2005 of *Informatik Berichte*. FernUniversität Hagen, Germany, 2005.

[20] Patrick Pélissier Guillaume Hanrot, Vincent Lefèvre and Paul Zimmermann. The mpfr library. available at `http ://www.mpfr.org`.

[21] John Harrison. Formal verification of IA-64 division algorithms. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics : 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 2000.

[22] Branimir Lambov. Reallib : an efficient implementation of exact real arithmetic. In Grubba et al. [19], pages 169–175.

[23] Valérie Ménissier-Morain. *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire*. Thèse, Université Paris 7, dec 1994.

[24] Valérie Ménissier-Morain. Conception et algorithmique d'une représentation d'arithmétique réelle en précision arbitraire. In *Proceedings of the first conference on real numbers and computers*, 1995.

[25] Jean-Michel Muller. *Elementary Functions, Algorithms and implementation*. Birkhauser, 1997.

[26] Norbert Th. Müller. Implementing exact real numbers efficiently. In Grubba et al. [19], page 378.

[27] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[28] Milad Niqui. *Formalising Exact Arithmetic : Representations, Algorithms and Proofs*. PhD thesis, Radboud University, Nijmegen, September 2004.

[29] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7 :175–204, March 1997.

[30] David M. Russinoff. A Mechanically Checked Proof of IEEE Compliance of the AMD K5 Floating-Point Square Root Microcode. *Formal Methods in System Design*, 14(1) :75–125, January 1999.

[31] Jean E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transactions on Computers*, 39(8) :1087–1105, aug 1990.