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 deux extensions du travail de Ciaffaglione et Di Gianantonio [9] sur l'utilisation de types co-inductifs pour modéliser le calcul exact sur les nombres réels. La première extension décrit une méthode pour combiner récursion bien fondée et co-récursion pour calculer des formules affines. La deuxième extension permet de calculer les séries formelles convergentes. Ce travail s'applique dans le cadre de la démonstration sur ordinateurs avec types inductifs et co-inductifs, comme c'est possible avec le système Coq [14, 4, 16].

1. introduction

Les systèmes de démonstration comme ACL2, Coq, HOL, Isabelle, ou PVS fournissent un type de donnée pour les nombres réels. Dans le système Coq, par exemple, ce type est accompagné des opérations de base et des théorèmes qui montrent que l'on est en présence d'une structure de corps ordonné et complet. Néanmoins, ces descriptions ne permettent pas de faire des calculs rapidement.

Dans une première partie, nous décrivons le principe de la représentation des nombres réels à l'aide de suites infinies de chiffres redondants. Nous montrons comment cette représentation permet de calculer paresseusement sur ces suites infinies. Nous reprenons l'exemple déjà connu de l'addition et nous montrons au passage comment cette fonction peut être engendrée à l'aide des outils de recherche de preuve fournis par Coq. Cette contribution n'est pas originale par son résultat, puisque l'addition est déjà traitée dans [9], mais la technique de construction est particulière.

De manière générale, la programmation des opérations sur les suites infinies de chiffres est resteinte par la discipline de la programmation co-récursive, issue de l'approche catégorique et des notions de co-algèbres finales. Dans cette approche, les suites infinies sont engendrées par des procédés de calcul récursif, avec la contrainte que les appels récursifs ne sont autorisés que si des bits du résultats sont produits à chaque appel récursif. De prime abord, cette contrainte est très forte et interdit la programmation de nombreuses fonctions récursives, celles qui peuvent nécessiter plusieurs appels récursif pour chaque chiffre du résultat. Nous montrons qu'il est possible de décomposer le calcul récursif des séquences infinies en deux parties, une partie "co-récursive", qui vérifie bien la discipline que chaque appel récursif produit au moins un élément de la séquence infinie, mais n'a pas à vérifier de propriété de terminaison, et une partie récursive, qui doit correspondre à une fonction qui termine toujours, mais ne doit pas nécessairement produire d'éléments de la séquence infinie que l'on est en train de construire. Pour illustrer cette décomposition, nous étudions une opération de calcul de formule affine à deux arguments et à coefficients rationnels. Notre approche est réminiscente des techniques proposées dans [12], mais nous espérons que notre approche est plus facile à comprendre. C'est la deuxième contribution de notre article.

Notre travail aborde ensuite un objectif plus ambitieux. Il s'agit de disposer des fonctions mathématiques usuelles, comme par exemple les fonctions exponentielles ou trigonométriques. Une approche naturelle pour implémenter le calcul de ces fonctions est de passer par le calcul de séries formelles de la forme $\sum_{i=0}^{\infty} a_i x^i$. Ce type de calcul semble poser un problème insoluble : comment peut-on calculer une somme infinie avec un programme qui doit terminer en un temps fini ? C'est ce problème que nous voulons résoudre dans cet article. Nous donnons en particulier une approche pour calculer une somme de la forme $\sum_{i=0}^{\infty} a_i$ dans le cas particulier où la valeur de cette somme est comprise entre 0 et 1. Nous argumenterons en conclusion pourquoi résoudre ce cas particulier est une étape importante dans la résolution de notre objectif principal. A titre d'exemple, 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 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 en utilisant seulement le mécanisme de réduction présent dans le système de preuve (une centaine de décimales en moins d'une minute, même sans faire appel au mécanisme de compilation proposé récemment dans [18]). Ce travail est la troisième contribution principale de cet article.

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. Ici encore, le résultat n'est pas original car l'algorithme que nous obtenons a fondamentalement la même forme que celui de [9], mais c'est la progression qui est amusante : on définit le calcul de séries entières avant de définir la multiplication.

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 chiffres est fini. Les nombres représentables directement en machine sont en nombre fini et 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'aide d'outils de preuve sur ordinateur [10, 21, 7, 28, 6].

Une approche alternative est de calculer avec des représentation des nombres qui évitent les arrondis et les erreurs associées. Il s'agit alors de représenter un nombre réel comme un objet infini, dont une partie finie est connue exactement et représente une approximation du nombre, et dont la seconde partie est représentée par une procédure qui permet d'améliorer l'approximation si le besoin s'en fait sentir. Parmi les représentations étudiées, on peut citer les représentations à base de fractions continues [17, 29] 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. Dans tous les cas, 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 Ciaffaglione et di Gianantonio [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 [27] 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.

La technique pour combiner co-récursion et récursion bien fondée que nous avons mise en œuvre dans cet article est comparable aux techniques décrites par Di Gianantonio et Miculan dans [12, 13] et à la technique que nous avons utilisée dans [3], mais dans un cas plus complexe puisque l'on y considère des fonctions partielles.

3. Représentation redondante des nombres réels

Nous avons tous appris à nous servir de nombres à virgule pour représenter les nombres réels. Par exemple, nous avons l'habitude d'écrire tout nombre à virgule compris entre 0 et 1 par un texte de la forme $0.13483\cdots$, et nous savons que la séquence de chiffres doit être infinie pour certains nombres : tous ceux qui ne sont pas de la forme $\frac{a}{10^b}$ où a et b sont des entiers positifs. Il est un peu moins naturel, mais facile à comprendre, que tous les nombres de l'intervalle $[0,1]$ sont représentables par des suites infinies de chiffres : si un nombre a une représentation finie, il suffit de lui ajouter une suite infinie de 0 et le nombre 1 lui-même est représentable par la suite $0.999\ldots$.

Lorsque l'on connait un préfixe de l'une de ces suites infinies, on connait le nombre qu'elle représente avec une certaine précision. En fait, si l'on dispose de n chiffres, on sait que le résultat est compris dans un intervalle de longueur $\frac{1}{10^n}$. Nous avons l'habitude de réfléchir avec ces "préfixes" de séquences infinies et nous attendons de nos outils de calculs qu'ils nous donnent un préfixe correct pour le résultat d'une opération lorsque nous leur avons fourni des préfixes corrects pour les entrées.

Dans cette représentation le nombre 10 joue un rôle particulier : c'est la base. On peut changer de base et n'utiliser alors que des chiffres compris entre 0 et cette base. Par exemple, on peut utiliser la base 2. Les chiffres sont alors uniquement 0 et 1. Le nombre $\frac{1}{2}$ est représentable par la séquence $0.1000\cdots$ et le nombre 1 est représentable par la séquence $0.1111\cdots$. Lorsque s est une séquence infinie de chiffre on a les égalités suivantes :

$$\begin{aligned} 0.0s &= \frac{0.s}{2} \\ 0.1s &= \frac{0.s+1}{2} \end{aligned}$$

Lorsque l'on dispose d'un préfixe de longueur n d'une séquence infinie en base 2, on connait alors le nombre représenté avec une précision de $\frac{1}{2^n}$.

Pour le calcul de préfixes corrects de résultats d'opérations, cette représentation est inadaptée. Voici un exemple, utilisant la base 2, les nombres $\frac{1}{3}$ et $\frac{1}{6}$ et une simple addition. Nous savons que $\frac{1}{3}+\frac{1}{6}=\frac{1}{2}$ et nous savons également que la représentation en base 2 de ces nombres est la suivante :

$$\begin{aligned} \frac{1}{3} &= 0.01010101\cdots \\ \frac{1}{6} &= 0.00101010\cdots \\ \frac{1}{2} &= 0.10000000\cdots = 0.01111111\cdots \end{aligned}$$

Pour justifier la première égalité, nous pouvons tenir le raisonnement suivant :

$$0.01010101\cdots = \sum_{i=1}^{\infty} \frac{1}{2^{2i}} = \frac{1}{1-\frac{1}{4}} - 1 = \frac{1}{3}$$

Les autres égalités se justifient par des raisonnements similaires.

Un préfixe fini de la suite qui représente $\frac{1}{3}$ peut servir de préfixe pour tous les nombres compris dans un intervalle qui encadre $\frac{1}{3}$. Certains des éléments de cet intervalle sont plus petits que $\frac{1}{3}$, d'autres sont plus grands. Il en va de même pour n'importe quel préfixe fini de $\frac{1}{6}$. Donc, l'addition de $\frac{1}{3}$ et de $\frac{1}{6}$ à partir de préfixes finis doit retourner un préfixe fini de la représentation de $\frac{1}{2}$ qui puisse aussi être préfixe de nombres qui sont plus grands ou plus petits que $\frac{1}{2}$. Ce n'est pas possible. Le nombre $\frac{1}{2}$ admet deux représentations, mais aucune de ces représentation ne peut servir de préfixe pour des nombres qui sont à la fois plus grand et plus petit que $\frac{1}{2}$. Le préfixe 0.1 ne peut être utilisé que pour des nombres qui sont plus grands ou égaux à $\frac{1}{2}$. Le préfixe 0.0 ne peut être utilisé que pour des nombres qui sont plus petits ou égaux à $\frac{1}{2}$. Même si l'on connait les entrées avec une très grande précision, il est impossible de fixer le résultat avec une précision meilleure que 1.

La solution de ce problème est d'ajouter un nouveau chiffre qui sert à disposer d'un encadrement strict autour de $\frac{1}{2}$. Nous utilisons une représentation redondante basée sur trois "chiffres" dont deux ont une signification intuitive proche des chiffres 0 et 1.

- le chiffre `L` est utilisé comme le chiffre 0. Si x est une suite infinie de chiffres représentant le nombre v, $\mathtt{L}x$ représente le nombre $v/2$.
- le chiffre `R` est utilisé comme le chiffre 1. Si x est une suite infinies de chiffre représentant le nombre v, $\mathtt{R}x$ 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 $\mathtt{C}x$ représente le nombre $v/2 + 1/4$.

Nous n'écrivons plus les deux caractères "0." au début des séquences, ainsi le nombre $\frac{1}{2}$ sera écrit $\mathtt{LRRR}\cdots$, $\mathtt{RLLLL}\cdots$, ou $\mathtt{CCCCC}\cdots$. 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 $x/2$.

Les nombres $\mathtt{L}\cdots$, $\mathtt{R}\cdots$, $\mathtt{C}\cdots$, sont éléments des intervalles $[0, \frac{1}{2}]$, $[\frac{1}{4}, \frac{3}{4}]$, $[\frac{1}{2}, 1]$. Dans la suite ces intervalles seront appelés les intervalles de base. La nouvelle notation conserve une propriété importante de l'ancienne : lorsque l'on connait un préfixe à n chiffres d'une séquence infinie de chiffres, on connait une approximation du nombre représenté avec une précision de $\frac{1}{2^n}$.

Enfin, citons une propriété importante de redondance, qui sera réutilisée plusieurs fois dans cet article. Avec les nouveaux chiffres, un nombre qui s'écrit $\mathtt{CL}x$ s'écrit également $\mathtt{LR}x$, et un nombre qui s'écrit $\mathtt{CR}x$ s'écrit également $\mathtt{RL}x$.

3.1. Addition directe

Dans cette section nous décrivons un algorithme pour effectuer l'addition de deux nombres dans notre représentation.

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 considère deux nombres $\mathtt{L}x'$ et $\mathtt{L}y'$, on est sûr que la somme est dans [0,1] et que la demi-somme est dans [0,1/2]. Le résultat peut être de la forme $\mathtt{L}(\frac{x'+y'}{2})$. L'algorithme peut donc produire le chiffre `L` et spécifier que le reste de la séquence infinie représentant le résultat sera fourni par un appel récursif pour 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 additionne un nombre $\mathtt{L}x'$ avec un nombre $\mathtt{R}y'$, alors le résultat est $\mathtt{C}(\frac{x'+y'}{2})$
- Dans tous les autres cas, il faut observer le deuxième chiffre d'au moins l'un des deux arguments, il suffit souvent d'appliquer les équivalences $\mathtt{LR} \sim \mathtt{CL}$ et $\mathtt{RL} \sim \mathtt{CR}$ pour se ramener à un cas déjà étudié.

- Si l'un des arguments débute par `CC` et l'autre débute par `LL` ou `RR` alors on peut emprunter à l'un des arguments pour rendre à l'autre pour retrouver un cas déjà traité. Par exemple, considérons la demi-somme de $\mathtt{CC}x''$ et $\mathtt{LL}y''$. Dans ce cas, on peut remarquer que $\mathtt{CC}x''$ représente le même nombre que $\mathtt{LC}x''$ plus $1/4$ et que $\mathtt{LL}y''$ représente le même nombre que $\mathtt{LR}y''$ moins $1/4$. Le résultat est donc le même que la demi-somme de $\mathtt{LC}x''$ et $\mathtt{LR}y''$, ce cas est traité dans le premier item.

Le principe de l'algorithme de demi-somme est ainsi exprimable en quelque lignes, en pratique il existe 25 cas différents.

Pour la multiplication par 2, c'est plus direct. 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 multiplication par 2 est une fonction partielle par nature, nous en avons fait une fonction totale. Ici, il est nécessaire de retourner la valeur 1 lorsque le premier chiffre est `R`, parce que le nombre représenté peut être $1/2$, et nous voulons que le résultat soit correct pour ce cas.

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.

3.2. Modélisation et vérification

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 la demi-somme, 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. Nous définissons une fonction qui prend en entrée deux schémas de séquences infinies et qui retourne l'intervalle qui contient les valeurs possibles pour la demi-somme de toutes les valeurs représentables par ces schémas. Si cet intervalle est inclus dans un intervalle, le chiffre correspondant est produit, suivi par un appel récursif sur les deux schémas de séquence privés de leurs premiers chiffres auxquels est apportée une correction lorsque c'est nécessaire.

Par exemple, si l'on considère les schémas `C::L::`x, et `L::`y, on sait que les valeurs représentées par le premier schéma sont dans $[1/4, 1/2]$ et les valeurs représentées par le second schéma sont dans $[0, 1/2]$, la demi-somme est donc dans $[1/8, 1/2]$. On peut donc produire le chiffre `L`. En revanche, les propriétés suivantes peuvent être établies

$$\frac{\mathtt{CL}x + \mathtt{L}y}{2} = \mathtt{L}(\frac{\mathtt{L}x + y}{2}) + \frac{1}{8} = \mathtt{L}(\frac{\mathtt{R}x + y}{2}).$$

L'appel récursif doit être fait sur `R`x et y.

Si l'intervalle des valeurs possibles n'est pas inclus dans un intervalle de base, alors on introduit un traitement par cas sur l'un des arguments dans l'algorithme. Par conséquent, l'un des schémas est raffiné en trois nouveaux schémas plus précis et la recherche recommence à l'étape précédente. Par exemple, si les schémas considérés sont `C`x et `L`y, l'intervalle des valeurs possibles pour le résultat est $[1/8, 5/8]$, qui n'est inclus dans aucun des intervalles de base. On introduit dans la fonction d'addition un traitement par cas sur x, ce qui amène à considérer les trois cas `CL`x' et `L`y, `CC`x' et `L`y et `CR`x' et `L`y.

L'algorithme ainsi obtenu est correct par construction, mais il est quand même nécessaire de construire séparément une preuve de correction. Nous avons vérifié formellement 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).
```

Pour démontrer ce théorème il faut couvrir les 25 cas de l'algorithme. Dans chaque cas, il suffit de vérifier des égalités entre formules affines à coefficients rationnels, où seulent les constantes 2 et 4 apparaissent en dénominateur. Certaines de vérifications sont des égalités, que nous pouvons résoudre aisément avec une tactique comme `field` [11], les autres sont des inégalités que nous pouvons résoudre avec la tactique `fourier`. Ce développement est disponible sur internet [1].

Des expérience préliminaires nous permettent d'affirmer que la même technique s'appliquera sans problème pour calculer la soustraction de deux valeurs réelles.

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

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

où $\frac{a}{a'}$, $\frac{b}{b'}$ et $\frac{c}{c'}$ sont des nombres rationnels positifs, et les nombres x et y sont des séquences infinies de chiffres. Pour modéliser cet algorithme, il faut 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

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

2. dès que ces deux valeurs sont incluses dans l'un des intervalles de base, on peut produire le bon chiffre et effectuer un appel récursif pour une nouvelle formule affine,
3. si les valeurs extrémales sont mal placées, on peut observer le premier chiffre de x et y, ceci amène à diviser la distance entre les valeurs extrémales par 2. Si l'on répète ce procédé, on arrive nécessairement en un temps fini à une formule affine pour laquelle les valeurs extrémales sont inclues dans un intervalle de base (par exemple, on sait que si la différence entre les valeurs extrémales est inférieure à 1/4, les deux valeurs extrémales sont forcément dans l'un des intervalles de base).

L'appel récursif de 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 associé à la production d'aucun 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 1/4.

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

4.2. Modélisation et vérification formelle

Nous définissons une structure de donnée pour représenter la formule affine. Elle contient 6 nombres entiers relatifs et 2 séquences infinies.

```
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 définir une fonction `af_real_value` qui associe à un élément de cette structure de donnée la valeur réelle qu'il est censé représenter.

Nous définissons un prédicat `positive_coefficients` sur la 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` et qui se charge de répéter les opérations décrites 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 donnée. Ce nouveau type 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 est 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` n'est assurée que si la valeur de la formule affine est dans $[0,1]$, ce qui est bien exprimé par le théorème de correction que nous avons vérifié formellement.

```
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 l'avons démontré en utilisant une preuve par co-récursion reposant sur la propriété `represents`. Le théorème `represents_equal` permet d'obtenir l'égalité pour conclure.

5. Calcul de séries entières

5.1. Approche générale

Les séries entières sont les nombres définis par les sommes $\sum_{i=0}^{\infty} a_i$, lorsque ces sommes sont convergentes. Bien que nous n'ayons pas formellement décrit la soustraction dans cet article, nous supposerons qu'une telle soustraction existe et nous nous autorisons à considérer des séries dont certains termes sont négatifs.

Pour calculer la valeur d'une série, il est bien sûr impossible d'additionner une infinité de termes. 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.

Notre approche ne fonctionne que pour certaines séries. Premièrement, nous ne savons représenter que les valeurs entre 0 et 1, il faut donc que la valeur de la série soit dans cet intervalle. En pratique, on peut toujours se ramener dans l'intervalle, soit en divisant tous les termes de la somme, et donc la somme elle-même, par une puissance de 2 (en ajoutant un préfixe de chiffres `L`), soit en ajoutant ou en soustrayant une valeur entière aux premiers termes de la série, soit en appliquant les deux méthodes simultanément.

Deuxièmement, nous devons disposer d'un critère calculable qui permet d'affirmer que la série converge. Ceci peut être représenté par une fonction μ qui satisfait la propriété suivante :

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

La fonction μ va nous servir à déterminer combien de termes de la somme doivent être pris en compte pour calculer la limite à une précision donnée. Par exemple, si nous voulons calculer la limite à ε près, il suffit de trouver n tel que $\mu(n) < \frac{\varepsilon}{2}$, puis de calculer la valeur de $\sum_{i=1}^{n} a_i$ à $\frac{\varepsilon}{2}$ près.

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

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

Dans cette fonction, y joue en fait le rôle inverse du nombre ε du paragraphe précédent. Nous proposons de procéder par étapes :

1. On calcule une valeur $\phi(y, n)$, supérieure ou égale à n, telle que $|y \times (\phi(y, n))|$ soit majoré par $\frac{1}{16}$. Cette valeur $\phi(y, n)$ sert à déterminer le nombre de termes de la somme qui seront effectivement pris en compte pour le calcul du prochain chiffre.
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 $\mathtt{RLL}v''$, alors on peut observer que v est aussi représentable par la séquence $\mathtt{CRL}v''$ 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.

La valeur $\frac{1}{16}$ qui intervient dans la première étape est justifiée de la façon suivante : si v est de la forme CLCv'', on sait que v est dans l'intervalle $[5/16, 7/16]$, mais la valeur que l'on cherche à calculer est $v + y\sum_{i=\phi(y,n)}^{\infty} a_i$. Nous ne pouvons choisir le premier chiffre que si nous pouvons être sûr que cette valeur est soit dans $[\frac{1}{4}, \frac{3}{4}]$ soit dans l'intervalle $[0, \frac{1}{2}]$. La propriété $|y \times \sum_{i=\phi(y,n)}^{\infty} a_i| < \frac{1}{16}$ suffit à assurer l'une de ces deux propriétés.

A 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.

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 étapes 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 et les deux premiers termes de la somme sont égaux à 1, mais nous pouvons nous ramener dans l'intervalle $[0, 1]$ en enlevant ces deux termes. La fonction μ pour cette série est la fonction

$$\mu(n) = \frac{1}{(n-1)!(n-1)}$$

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

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

Dès que $2 < n$, nous savons que $\mu(n+1) < \frac{\mu(n)}{2}$. 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 ici 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 $\frac{1}{(n-1)!(n-1)} > \frac{1}{8}$, nous additionnons $\frac{y}{n!}$ à x en utilisant l'addition directe que nous avons définie précédemment et une fonction `rat_to_stream` qui prend deux arguments entiers et fabrique la séquence infinie de chiffres pour la fraction de ces deux entiers.

```
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.
```

Cette fonction réutilise donc la fonction `series_body` que nous avons décrite dans la section précédente.

Nous avons démontré que cette fonction calcule bien la partie fractionnaire de e avec 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, les fonctions `INR`, `Z_of_nat`, `Zabs_nat` et `IZR` sont des fonctions de conversion d'un type de nombres à l'autre. 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 calculons ensuite manuellement la somme des premiers termes, avant de lancer la fonction co-récursive pour calculer $\sum_{i=4}^{\infty} \frac{1}{i!}$, sachant que $(4-1)! = 6$.

```
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 $\sum_{i=0}^{\infty} \frac{1}{(i+2)!} = \sum_{i=2}^{\infty} \frac{1}{i!}$.

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 $\alpha(\mathtt{L}) = 0$, $\alpha(\mathtt{C}) = \frac{1}{4}$ et $\alpha(\mathtt{R}) = \frac{1}{2}$, alors uu' est la série entière

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

Cette série entière est naturellement convergente et sa limite est naturellement entre 0 et 1, puisqu'il s'agit du produit de deux nombres réels compris entre 0 et 1.

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 le dénominateur de a_i est également multiplié par 2 dans a_{i+1}. 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, l'argument y de la présentation générale ne sert plus à rien et seuls la séquence 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 la valeur $x + uu'$ dès que $uu' < \frac{1}{4}$ et x est assez petit. 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 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 bases 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] Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, 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] Milad Niqui. *Formalising Exact Arithmetic : Representations, Algorithms and Proofs*. PhD thesis, Radboud University, Nijmegen, September 2004.

[28] 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.

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