Arithmétique réelle exacte certifiée, co-induction et base arbitraire

Nicolas Julien

INRIA Sophia Antipolis
Nicolas.Julien@sophia.inria.fr

Résumé

Nous décrivons dans cet article des algorithmes certifiés pour l'arithmétique réelle exacte basée sur la co-récursion. Nos travaux s'inspirent d'expériences précédentes utilisant des chiffres signés en base 2 [1, 2] mais généralisent ces opérations à l'utilisation de bases entières arbitraires. L'objectif est d'utiliser les operations rapides de l'arithmétique entière des micro-processeurs.

1. Introduction

Ce travail fait suite à celui de Bertot [1] qui décrit une bibliothèque certifiée de calculs en arithmétique réelle exacte. Il représente les nombres réels de $[0, 1]$ par des suites infinies de trois chiffres et décrit ses calculs à l'aide de la co-récursion. Il fournit des opérations de base et une méthode de calcul des séries entières convergentes dont il se sert pour calculer la constante d'Euler et l'opération de multiplication.

Nous présentons tout d'abord la représentation utilisée qui permet désormais de décrire l'ensemble des nombres réels et d'utiliser n'importe quelle base entière supérieure à 3. La formalisation de la base a pour ambition de rendre possible l'utilisation d'entiers bornés pour les chiffres de la base et ainsi bénéficier d'opérations rapides fournies par le microprocesseur. Nous décrivons ensuite comment le paramétrage par une base influe sur la complexité des opérations et nous fournissons des solutions pour adapter les algorithmes à ce nouveau cadre de travail. Puis nous décrivons une amélioration de l'approche pour calculer les séries. Enfin après avoir vu quelques exemples d'implantation, nous illustrerons à travers quelques tests les apports au niveau des performances de l'utilisation de grandes bases.

2. Contexte

Les outils usuels de calcul décrivent les nombres réels par une représentation finie : les nombres à virgule flottante. Les nombres représentés ne sont en réalité qu'un sous ensemble fini des nombres rationnels. Il est bien connu que calculer sur cette représentation implique des arrondis qui peuvent entraîner d'importantes erreurs de calcul lorsqu'ils ne sont pas maîtrisés [3].

L'arithmétique réelle exacte fournit des calculs que l'on peut effectuer avec une précision arbitraire ; le résultat est un intervalle aussi fin que l'on veut, dont on a la garantie qu'il contient le résultat escompté. Représenter de manière exacte les nombres réels nécessite un moyen de représenter des objets infinis. Nous utilisons des séquence inifinies de chiffres. Dans leurs travaux, Edalat et Heckmann [4] utilisent la même représentation en base arbitraire et décrivent leurs calculs par des transformations fractionnelles linéaires. Ciaffaglione et Di Gianantonio [2] utilisent cette représentation en base 2 et fournissent une certification de leurs résultats grâce à l'assistant de preuves Coq [5, 6]. Bertot [1] utilise différemment la base 2 et formalise le calcul de séries pour décrire des fonctions complexes. Il utilise aussi Coq pour certifier ces calculs.

Coq rend possible une telle représentation grâce à l'utilisation de la co-induction [7]. On peut définir des types co-inductifs qui permettent de représenter des types d'objets infinis et des fonctions co-récursives qui permettent de les construire. Comme ces fonctions construisent des objets infinis, leur évaluation est retardée étape par étape au fur-et-à-mesure que les objets sont explorés, de manière paresseuse. Coq ne permet de construire que des fonctions dont on est sûr qu'elles se terminent, pour assurer que les objets ont toujours un sens. Ainsi chaque étape d'évaluation d'un objet co-inductif doit se terminer. C'est pourquoi en Coq, les appels co-récursifs de fonctions ne sont autorisés qu'après avoir produit une partie finie non nulle de l'objet. Ainsi on a la garantie que l'on pourra toujours évaluer une partie finie d'un objet co-inductif en un temps fini. On retrouve la même contrainte sous une forme différente, lorsque l'on veut démontrer un prédicat co-inductif en Coq. En effet ceci revient à construire un terme de preuve qui sera un objet co-inductif. La tactique `cofix` permet d'aider à la construction de cette preuve. Elle permet d'obtenir une hypothèse de récurrence qui est identique au but à démontrer. L'utilisation de cette hypothèse correspond à effeectuer un appel co-récursif et n'est donc valable qu'après avoir démontrer une partie finie non vide de la preuve.

3. Représentation des réels

L'arithmétique réelle exacte nécessite l'utilisation de structures de données infinies. Une possiblité est de représenter un nombre réel de $[0,1]$ par la suite infinie de ses décimales. Par exemple $\frac{1}{3}$ est représenté par 333333333333... Connaître un préfixe fini d'une telle séquence permet de connaître le nombre avec une certaine précision. En effet, si la séquence représentant un nombre de $[0,1]$ commence par 1415 alors ce nombre est compris dans $[\frac{1415}{10000}, \frac{1416}{10000}]$.

Cette notation peut être étendue à une base quelconque, la base 2 étant la plus couramment utilisée en informatique. De plus on peut élargir l'intervalle des nombres représentés à $[-1,1]$ grâce à l'utilisation de chiffres signés. Ainsi, en base β, une suite infinie de chiffre d_n représente le nombre :

$$\sum_{i=1}^{\infty} \frac{d_i}{\beta^i} \qquad \text{tel que} - \beta < d_i < \beta.$$

La notation $k::x$ signifiera la séquence infinie commençant par le chiffre k et continuant par la séquence infinie x. Pour simplifier, nous noterons indifféremment une représentation et le nombre qu'elle dénote. On peut remarquer que la valeur de $k::x$ est $\frac{k+x}{\beta}$ car $\sum_{i=1}^{\infty} \frac{d_i}{\beta^i} = \frac{d_1 + \sum_{i=1}^{\infty} \frac{d_{i+1}}{\beta^i}}{\beta}$. Ainsi on peut déduire qu'une séquence $k::x$ représentera un nombre de $[\frac{k-1}{\beta}, \frac{k+1}{\beta}]$. Inversément on pourra aussi déduire qu'un nombre compris dans un tel intervalle aura au moins une représentation qui commence par le chiffre k quand k est bien un chiffre de la base.

Cette représentation est redondante car un même nombre peut admettre une infinité de représentations. Par exemple si x est une séquence infinie de chiffres signés de la base β, $0:: -1::x$ et $-1::\beta - 1::x$ représentent le même nombre :

$$\frac{0}{\beta} + \frac{-1}{\beta^2} + \frac{x}{\beta^2} = \frac{-1}{\beta} + \frac{\beta - 1}{\beta^2} + \frac{x}{\beta^2} = \frac{-1}{\beta^2} + \frac{x}{\beta^2}.$$

Cette représentation redondante est inspirée en informatique des travaux d'Avizienis [8] qui l'utilise pour améliorer le calcul de l'addition en évitant la propagation des retenues. Elle va rendre les calculs représentables dans le cadre co-inductif.

Bertot utilise dans ses travaux des nombres représentés en base 2 par des chiffres non pas signés mais fractionnaires ce qui lui permet d'avoir une telle représentation redondante sur l'intervalle $[0,1]$.

Dans notre travail, nous utiliserons des séquences infinies de chiffres signés dans une base arbitraire supérieure à 3 que nous munirons d'un entier relatif pour obtenir un couple (mantisse, exposant). Ainsi l'ajout de cet exposant nous permettra de sortir du cadre de [-1, 1] pour représenter l'ensemble des nombres réels.

4. Méthode de calcul

Comme nous utilisons une représentation des nombres réels par un couple (mantisse, exposant), lorsque nous voulons implanter un algorithme qui calcule une fonction mathématique, nous le définissons tout d'abord sur les mantisses, ce qui représente la partie compliquée, puis nous l'étendons sur l'ensemble des nombres.

La technique pour calculer sur les mantisses est la suivante : on essaie d'obtenir un encadrement du résultat suffisamment précis pour déterminer un préfixe fini non nul de sa représentation, en général un chiffre, puis on charge un appel récursif de calculer la suite. Comme ces fonctions produiront des mantisses, leur résultat sera donc dans [-1, 1]. Lorsque la fonction mathématique représentée a un intervalle d'arrivée plus grand, il faut donc la transformer en une fonction dont l'intervalle d'arrivée est satisfaisant.

Comme on a vu, dans notre représentation redondante, un nombre $k::x$ est dans l'intervalle $[\frac{k-1}{\beta}, \frac{k+1}{\beta}]$. Les chevauchements de deux de ces intervalles consécutifs sont de taille $\frac{1}{\beta}$. Ainsi connaître un intervalle contenant le résultat et de taille inférieure à $\frac{1}{\beta}$ est toujours suffisant pour en déterminer un premier chiffre possible.

4.1. Calcul de l'addition

L'image de l'addition sur les mantisses est l'intervalle $[-2, 2]$ qui contient des nombres non représentables par une mantisse. Dans son travail, Bertot propose donc de définir une opération stable, la demi-somme ($x, y \mapsto \frac{x+y}{2}$). Puis de la composer avec une fonction proche de la multiplication par 2 qui est capable de gérer les débordements : $x \mapsto \begin{cases} 2x & \texttt{si} \quad x \leq \frac{1}{2} \\ 1 & \texttt{sinon} \end{cases}$.

Comme nous travaillons dans une base arbitraire, nous avons considéré que l'adaptation de ce calcul de l'addition devait être basé sur une division puis une multiplication non plus par 2, mais par la base. Nous définissons donc en comparaison avec la demi-somme une opération de somme de deux mantisses et d'une retenue entière, le tout divisé par la base.

$$\text{sum\_div\_base} \begin{cases} \mathbb{Z} \times [\text{-}1,\, 1] \times [\text{-}1,\, 1] \times \mathbb{Z} & \mapsto & [\text{-}1,\, 1] \\ \beta, x, y, r & \mapsto & \frac{x+y+r}{\beta} \end{cases}$$

Cette opération est bien stable sur [-1, 1] si l'on a la garantie que la retenue r est toujours dans l'intervalle $[-\beta+2, \beta-2]$. L'algorithme que nous proposons pour cette fonction est le suivant :

- On lit le premier chiffre de x et y ce qui donne k_1, k_2, x' et y' tels que $x = k_1::x', y = k_2::y'$. On a donc :
$$\texttt{sum\_div\_base}(\beta, x, y, r) = \frac{r + \frac{x'+y'+k_1+k_2}{\beta}}{\beta}.$$
Comme k_1 et k_2 sont des chiffres de la base, on peut déduire $-2\beta+2 \leq k_1+k_2 \leq 2\beta-2$. Donc r semble être à peu près un bon candidat pour le premier chiffre du résultat et k_1+k_2 pour la prochaine retenue.
- Si $\beta - 1 \leq k_1 + k_2$ alors $k_1 + k_2$ est trop grand pour être une retenue valable, mais $k_1 + k_2 - \beta$ est acceptable. Ainsi le prochain chiffre peut être $r + 1$ qui est forcément un chiffre valable car,

comme r est la retenue précédente, $-\beta+2 \leq r \leq \beta-2$.

$$\begin{aligned}
\texttt{sum_div_base}(\beta, x, y, r) &= \frac{r+1+\frac{x'+y'+k_1+k_2-\beta}{\beta}}{\beta} \\
&= (r+1)\texttt{::sum_div_base}(\beta, x', y', k_1+k_2-\beta).
\end{aligned}$$

- Si $k_1+k_2 \leq -\beta+1$ alors k_1+k_2 est trop petit pour être une retenue valable, mais $k_1+k_2+\beta$ est acceptable. Ainsi le prochain chiffre peut être $r-1$ qui est forcément un chiffre valable car, comme r est la retenue précédente, $-\beta+2 \leq r \leq \beta-2$.

$$\begin{aligned}
\texttt{sum_div_base}(\beta, x, y, r) &= \frac{r-1+\frac{x'+y'+k_1+k_2+\beta}{\beta}}{\beta} \\
&= (r-1)\texttt{::sum_div_base}(\beta, x', y', k_1+k_2+\beta).
\end{aligned}$$

- Sinon k_1+k_2 est une valeur de retenue possible et r peut être le prochain chiffre.

$$\texttt{sum\_div\_base}(\beta, x, y, r) = \frac{r+\frac{x'+y'+k_1+k_2}{\beta}}{\beta} = r\texttt{::sum\_div\_base}(\beta, x', y', k_1+k_2)$$

À l'image de la multiplication par 2, nous devons ensuite définir un algorithme de multiplication d'une mantisse par la base qui prend en compte les dépassements de [-1, 1] :

$$\texttt{mult\_base}: \begin{cases} \mathbb{Z} \times [\text{-}1,\ 1] & \mapsto \quad [\text{-}1,\ 1] \\ \beta, x & \mapsto \quad \begin{cases} -1 & \text{si } x \leq \frac{-1}{\beta} \\ 1 & \text{si } x \geq \frac{1}{\beta} \\ x \times \beta & \text{sinon} \end{cases} \end{cases}.$$

Il nous faut tout d'abord définir les constantes représentant 1 et -1. Ce sont les bornes de l'intervalle de valeurs représentables par les mantisses. On les définit donc commes les séquences infinies du plus grand chiffre $\beta-1$ pour 1, et du plus petit chiffre $-(\beta-1)$ pour -1.

$$\sum_{i=1}^{\infty} \frac{\beta-1}{\beta^i} = \frac{\beta-1}{\beta} \sum_{i=0}^{\infty} \frac{1}{\beta^i} = \frac{\beta-1}{\beta} \frac{1}{1-\frac{1}{\beta}} = 1$$

On peut alors décrire le calcul de la fonction `mult_base` :

- On regarde le premier chiffre de x : $x = k_1\texttt{::}x'$
- Si $k_1 = 0$ alors le résultat est x' car ajouter un zéro en tête d'une séquence équivaut à diviser par la base le réel représenté.
- Si $k_1 \leq -2$, alors on peut déduire que $x \leq \frac{-1}{\beta}$ donc le résultat est la constante -1.
- Si $k_1 \geq 2$, alors on peut déduire que $x \geq \frac{1}{\beta}$ donc le résultat est la constante 1.
- Si $k_1 = 1$ il faut regarder un chiffre supplémentaire de x : $x = 1\texttt{::}k_2\texttt{::}x''$
  - Si $k_2 < 0$, alors on utilise la redondance de la notation pour se ramener à un cas que l'on sait faire $x = 1\texttt{::}k_2\texttt{::}x'' = 0\texttt{::}(k_2+\beta)\texttt{::}x''$, le résultat est donc $(k_2+\beta)\texttt{::}x''$.
  - Si $k_2 > 0$, on se ramène aussi à un cas que l'on sait faire $x = 1\texttt{::}k_2\texttt{::}x'' = 2\texttt{::}(k_2-\beta)\texttt{::}x''$, le résultat est donc 1.
  - Si $k_2 = 0$ alors on doit effectuer un appel récursif :

$$\texttt{mult\_base}(\beta, 1\texttt{::}0\texttt{::}x'') = \beta \times \frac{1+\frac{0+x''}{\beta}}{\beta} = \frac{\beta-1+\beta\times\frac{1+x''}{\beta}}{\beta} = \beta{-}1\texttt{::mult\_base}(\beta, 1\texttt{::}x'').$$

Dans ce cas, on ne peut pas décider si l'argument est supérieur ou égal à $\frac{1}{\beta}$. En revanche, si il était inférieur, on saurait qu'il en est tout de même suffisamment proche pour que le résultat

puisse commencer par $\beta - 1$. Si il était supérieur, alors le résultat serait la constante 1 qui commence aussi par $\beta - 1$. Le chiffre rendu $\beta - 1$ est donc bon dans tous les cas et l'appel récursif se chargera donc de continuer le calcul ou de construire la constante 1.
- Si $k_1 = -1$ on suit un raisonnement symétrique au cas où $k_1 = 1$.

En composant nos deux fonctions, on obtient une fonction sur les mantisses qui calcule la somme quand celle-ci est dans [-1, 1].

$$\texttt{add}(\beta, x, y) = \texttt{mult\_base}(\beta, \texttt{sum\_div\_base}(\beta, x, y, 0))$$

$$\texttt{add} : \left\{ \begin{array}{lcl} \mathbb{Z} \times [-1, 1] \times [-1, 1] & \mapsto & [-1, 1] \\ \beta, x, y & \mapsto & \left\{ \begin{array}{ll} -1 & \text{si } x + y \leq -1 \\ 1 & \text{si } x + y \geq 1 \\ x + y & \text{sinon} \end{array} \right. \end{array} \right.$$

4.2. La fonction make_digit

Nous venons de décrire un algorithme d'addition sur les mantisses qui à partir d'un préfixe de chaque argument calcule le premier chiffre de la somme. Une autre méthode est possible quand on a la certitude qu'un des deux nombres est suffisamment proche de 0. Dans ce cas, calculer un préfixe de l'autre nombre suffit à décider du premier chiffre du résultat.

En effet, supposons que l'on veuille faire la somme de x et y et que l'on sache que $|y| \leq \frac{\beta-2}{2\beta^2}$. Si on calcule les 2 premiers chiffres de $x = d_1 \texttt{::} d_2 \texttt{::} x''$ alors on connait un encadrement de x de taille $\frac{2}{\beta^2}$. Donc connait un encadrement de la somme de taille $\frac{1}{\beta}$ ce qui suffit à déterminer le premier chiffre de la somme.

Nous proposons donc de définir une fonction `make_digit` qui à partir de la représentation x d'un nombre rend une nouvelle représentation $k \texttt{::} x'$ de ce nombre dont le premier chiffre k peut toujours être produit en cas de somme avec un nombre proche de 0.

Soit un y “suffisamment proche de 0” :

$$x + \frac{-\beta + 2}{2\beta^2} \leq \texttt{make\_digit}(\beta, x) + y \leq x + \frac{\beta - 2}{2\beta^2}.$$

Le calcul de `make_digit` se fait de la façon suivante :
- On regarde les deux premiers chiffres de $x = k_1 \texttt{::} k_2 \texttt{::} x''$. On a donc

$$\frac{k_1 + \frac{k_2 + x''}{\beta}}{\beta} + \frac{-\beta + 2}{2\beta^2} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{k_1 + \frac{k_2 + x''}{\beta}}{\beta} + \frac{\beta - 2}{2\beta^2}$$

$$\frac{k_1 + \frac{2k_2 + 2x'' - \beta + 2}{2\beta}}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{k_1 + \frac{2k_2 + 2x'' + \beta - 2}{2\beta}}{\beta}$$

  Comme $x'' \in [-1, 1]$ on peut ajouter,

$$\frac{k_1 + \frac{2k_2 - \beta}{2\beta}}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{k_1 + \frac{2k_2 + \beta}{2\beta}}{\beta}$$

- Si $-\beta \leq 2k_2 \leq \beta$, alors on a

$$\frac{k_1 - 1}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{k_1 + 1}{\beta}$$

  Donc le résultat peut commencer par le chiffre k_1 et le reste est $k_2 \texttt{::} x''$:

$$\texttt{make\_digit}(\beta, x) = k_1 \texttt{::} k_2 \texttt{::} x'' = x.$$

- Sinon, si $\beta < 2k_2$
  - Si $k_1 \neq \beta - 1$, alors on utilise la redondance de la représentation $k_1{::}k_2{::}x'' = k_1 + 1{::}k_2 - \beta{::}x''$, on montre alors que
$$\frac{(k_1+1)-1}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq \frac{(k_1+1)+1}{\beta}$$
    Le résultat peut donc être :
$$\texttt{make\_digit}(\beta, x) = k_1 + 1{::}k_2 - \beta{::}x''.$$
  - Sinon $k_1 + 1 = \beta$, donc ce n'est pas un chiffre possible, mais comme on suppose le résultat dans [-1, 1], on a
$$\frac{(\beta-1)-1}{\beta} \leq \texttt{make\_digit}(\beta, x) + y \leq 1 = \frac{(\beta-1)+1}{\beta}$$
    Le résultat peut donc être :
$$\texttt{make\_digit}(\beta, x) + y = \beta - 1{::}k_2{::}x''.$$
- Sinon $2k_2 < -\beta$ et on adopte un raisonnement similaire.

4.3. Adaptation du calcul des séries

Bertot propose une description du calcul de séries entières convergentes utilisant la technique décrite par `make_digit`. En effet dans le cas d'une serie convergente, on pourra toujours séparer la série en une partie finie qui servira à décider du chiffre du résultat et d'une partie infinie aussi proche de 0 que souhaitée. Nous avons donc adapté la technique de Bertot à une base quelconque en isolant dans `make_digit` une partie du calcul. Dans notre algorithme de calcul de séries convergentes, nous nous plaçons encore dans un cadre où la base est supérieure à 3.

Pour pouvoir calculer la représentation en base β d'une série $\sum_{i=0}^{\infty} a_i$, convergeant dans l'intervalle [-1, 1], on commence par définir le calcul d'une fonction
$$f(\beta, y, n, r) = y \times \sum_{i=n}^{\infty} a_i + r.$$
Il suffira ensuite d'appeler $f(\beta, 1, 0, 0) = \sum_{i=0}^{\infty} a_i$ pour obtenir le calcul de la série.

Pour définir cette fonction, on calcule d'abord $p \geq n$, tel que $|y \times \sum_{i=p}^{\infty} a_i| \leq \frac{\beta-2}{2\beta^2}$. Comme la série est convergente et y et β sont fixés, on sait qu'un tel p existe. Ensuite on regarde les deux premiers chiffres de $x = y \times \sum_{i=n}^{p-1} a_i + r$, ce qui nous permet de connaître un encadrement de x de longueur $\frac{2}{\beta^2}$. Ces 2 encadrements nous permettent alors de déduire un encadrement de $f(\beta, y, n, r) = \sum_{i=p}^{\infty} a_i + x$ de $\frac{1}{\beta}$ ce qui est suffisant pour décider de son premier chiffre k. On calcule la séquence s des autres chiffres par un appel récursif de la même manière que décrit précédemment.

$$f(\beta, y, n, r) \quad = \quad \frac{k+s}{\beta} = y \times \sum_{i=n}^{p-1} a_i + y \times \sum_{i=p}^{\infty} a_i + r = y \times \sum_{i=p}^{\infty} a_i + x$$

$$\begin{aligned} \text{En conséquence} \qquad s &= \beta \times (y \times \sum_{i=p}^{\infty} a_i + x) - k \\ &= \beta \times y \times \sum_{i=p}^{\infty} a_i + \beta \times x - k \\ &= f(\beta, \beta \times y, p, \beta \times x - k) \end{aligned}$$

L'argument r de la fonction f représente donc le reste entre le nombre calculé et l'approximation donnée par le chiffre produit. La partie du calcul qui se charge de produire le chiffre et de calculer le reste est indépendante de la série calculée et peut être formalisée une fois pour toute.

Ainsi l'algorithme de fonction f qui calcule pour une série suivra le schéma suivant

- Trouver $p \geq n$ tel que $y \times \sum_{i=p}^{\infty} a_i$ soit suffisamment proche de 0
- Calculer $k::r' = \texttt{make\_digit}(\beta, y \times \sum_{i=n}^{p-1} a_i + r)$
- Produire l'appel récursif $k::f(\beta, \beta \times y, p, r')$

Souvent la fonction f n'aura pas exactement ces paramètres là. En effet, dans certains cas on peut établir formellement que des paramètres peuvent être simplifiés. On peut parfois aussi ajouter d'autres paramètres permettant d'améliorer le calcul, comme par exemple éviter de recalculer $n!$ depuis le début à chaque appel récursif.

Dans notre adaptation du calcul des séries entières, nous avons complètement isolé dans la fonction `make_digit` le calcul du prochain chiffre à produire et du reste. Dans le travail de Bertot, ce calcul était déjà formalisé quelque soit la série mais à travers une fonction `series_body` qui prenait en paramètre la fonction f et se chargeait en plus de lancer l'appel récursif de f.

Le fait que les appels récursifs de f ne soient pas décrits dans f mais dans `series_body` obligeait aussi à passer en paramètre les arguments de f à travers un argument de type générique. Cela pouvait rendre l'écriture de séries un peu obscure. Un autre point plus important est que nous n'avions pas réussi à montrer un théorème décrivant le travail de `series_body` sur la production du chiffre résultat. Ainsi lorsque l'on voulait montrer la correction du calcul d'une série, il nous fallait toujours procéder en deux étapes. D'abord montrer que l'utilisation de `series_body` dans ce calcul était correct ce qui obligeait à remontrer sans cesse que la production du premier chiffre l'était. Puis montrer que le calcul de la série lui-même était correct aussi.

Désormais il est possible de démontrer une fois pour toute le bon comportement de `make_digit`. L'isolation du travail décrit dans `make_digit` nous a permis de rendre l'écriture et la certification des séries plus aisées.

4.4. Calcul de la multiplication

Comme le propose Bertot, la multiplication sur les mantisses peut être traduite comme une série et l'on peut utiliser la technique expliquée ci-dessus pour la calculer. En effet si l'on considère qu'une mantisse u est représentée par la séquence $d_1::d_2::d_3::\ldots$, alors son produit avec une autre mantisse v sera : $u \times v = \sum_{i=1}^{\infty} \frac{d_i \times v}{\beta^i}$.

De plus, Bertot montre que dans le cas de la multiplication, le paramètre y de la méthode pour les séries est inutile car à chaque appel récursif il est multiplié par 2 puis divisé par 2 aussi. Cette propriété se retrouve dans notre cadre où β apparaît à la place de 2. Comme sa valeur initiale est 1, on peut donc s'en passer. On peut se passer aussi de l'argument n car $\sum_{i=n}^{\infty} \frac{d_i}{\beta^i}$ correspond à la mantisse u à laquelle on a enlevé les $n-1$ premiers chiffres. On se ramène alors à calculer la fonction $f(\beta, u, v, r) = u \times v + r$.

Pour cela, si on calcule le premier chiffre de $u = k::u'$, alors

$$f(\beta, u, v, r) = \frac{k + u'}{\beta} \times v + r = \frac{k \times v}{\beta} + r + \frac{u' \times v}{\beta}$$

Si $d::r' = \texttt{make\_digit}(\frac{k \times v}{\beta} + r)$ et $|\frac{u' \times v}{\beta}| \leq \frac{\beta - 2}{2\beta^2}$ alors

$$f(\beta, u, v, r) = d::f(\beta, u', v, r')$$

Comme u' ne peut représenter qu'un réel de [-1, 1], l'inégalité est assurée lorsque $|v| \leq \frac{1}{\beta}$. On appelle donc initialement f avec $0::v = \frac{v}{\beta}$ et on multiplie le résultat de f par β pour définir la multiplication. $u \times v = \beta \times u \times \frac{v}{\beta} = \beta \times f(\beta, u, 0::v, 0)$

Le problème ici est que l'on doit calculer $\frac{k \times v}{\beta} + r$. Or pour que ce calcul soit valide, il faut que le résultat soit dans [-1, 1]. Pour le garantir il nous faut faire la preuve d'un invariant que nous n'avons pas été capable de montrer en Coq en raison des contraintes de la co-induction. Pourtant la preuve nous semblait abordable. Nous avons donc dû modifier notre algorithme pour en rendre la certification possible.

- Nous regardons le premier chiffre de $u = k_1::u'$. Nous avons comme invariant que $-1 \leq u \times v + r \leq 1$ car le résultat doit être exprimable par une mantisse, et que $-1 \leq r \leq 1$ car r est une mantisse.
- Si $k_1 = 0$ alors la preuve ne posera pas de problème. On peut donc suivre le schéma précédent.
- Sinon il faut regarder un deuxième chiffre : $u = k_1::k_2::u''$.
  - Si le premier chiffre k est strictement positif,
    - Si le deuxième chiffre est strictement positif aussi alors on peut utiliser l'algorithme précédent car la preuve sera facile et on utilise encore le schéma précédent. En effet supposons que v soit positif, alors $-1 \leq r \leq \frac{k \times v}{\beta} + r \leq u \times v + r \leq 1$. Sinon on montre l'inégalité par un raisonnement symétrique.
    - Si le deuxième chiffre est strictement négatif, on peut utiliser la redondance de la notation pour se placer dans un cas où la preuve sera faisable : $k_1::k_2::u'' = k_1 - 1::k_2 + b::u''$. Ainsi soit on se retrouve dans le cas où le premier chiffre est nul soit dans le cas ou les deux chiffres sont strictement positifs.
    - Si le deuxième chiffre est nul, alors ce cas là est un peu plus compliqué. On remarque tout d'abord que $\frac{k_1 + \frac{0+u''}{\beta}}{\beta} \times v = \frac{\frac{\beta \times k_1 - 1}{\beta} + \frac{1+u''}{\beta}}{\beta}$. Comme 1 et $\beta \times k_1 - 1$ sont strictement positifs, on arrivera facilement à faire notre preuve en suivant un raisonnement analogue au cas où les deux chiffres sont strictement positifs.
  - Sinon, le premier chiffre est strictement négatif et on adopte un raisonnement symétrique au cas positif.

De plus, le fait d'utiliser une base quelconque pose un autre problème. En effet en base 2 le calcul de $\frac{d_i \times y}{\beta}$ est direct car les valeurs possibles pour les d_i sont -1, 0 et 1. Nous allons donc devoir d'abord écrire une fonction qui calcule $y \mapsto \frac{d_i \times y}{\beta}$. On relâche le problème en définissant une fonction plus générale qui calcule le produit d'une mantisse par un rationnel de [-1, 1] :

$$mult\_rat(\beta, x, a, b) \mapsto x \times \frac{a}{b}, -1 \leq \frac{a}{b} \leq 1$$

Nous avons défini cette fonction par une série qui suit le même schéma.

4.5. Calcul sur l'ensemble des réels

Une fois que l'on a défini nos calculs sur les mantisses, on définit leur extension sur $\mathbb{R}$ de manière classique pour une représentation (mantisse, exposant).

On définit d'abord des fonctions inductives `add_zero` et `del_zero` qui permettent d'ajouter ou d'enlever un nombre donné de zéros en tête d'une mantisse. Ceci correspond respectivement à diviser et multiplier autant de fois cette mantisse par la base. De plus, `del_zero` indique aussi la quantité de zéros qu'elle a pu enlever. Ces opérations nous permettront donc d'ajuster au besoin les exposants en utilisant la redondance de la représentation (mantisse, exposant) $(m, e) = m \times \beta^e = \frac{m}{\beta} \times \beta^{e+1} = (0::m, e + 1)$.

On peut alors définir le calcul de la somme sur deux nombres réels représentés par les couples (e_1, x) et (e_2, y) de la manière suivante.

– Si $e_1 \geq e_2$, alors

$$\beta^{e_1} \times x + \beta^{e_2} \times y = \beta^{e_1+1} \times \frac{x + \frac{y}{\beta^{e_1-e_2}}}{\beta} = (e1+1, \texttt{sum\_div\_base}(x, \texttt{add\_zero}(y, e1-e2), 0))$$

– Sinon, on effectue un calcul symétrique.

On peut remarquer qu'on peut s'affranchir d'appliquer la fonction `mult_base` en ajoutant 1 à l'exposant.

L'extension de la multiplication se fait de la manière suivante :

$$\begin{aligned}(m_1, e_1) \times (m_2, e_2) &= m_1 \times \beta^{e_1} \times m_2 \times \beta^{e_2} \\ &= m_1 \times \tfrac{m_2}{\beta} \times \beta^{e_1+e_2+1} \\ &= (m_1 \times 0\texttt{::}m_2, e_1 + e_2 + 1)\end{aligned}$$

Ici aussi on peut économiser un appel à `mult_base` en ajoutant 1 à l'exposant.

5. Formalisation en Coq

5.1. Définitions

Les types co-inductifs que l'on trouve en Coq rend possible notre représentation et la certification de nos algorithmes. On définit d'abord un type de séquence infinie pour représenter les mantisses.

```
CoInductive stream (A:Set): Set :=
  | Cons : A → stream A → stream A.
```

Les mantisses étant des séquences infinies d'entiers relatifs, elles auront donc pour type `stream` $\mathbb{Z}$ et les réels seront des couples de type $\mathbb{Z}\times$`stream` $\mathbb{Z}$. On va pouvoir définir nos mantisses au moyen de fonctions co-récursives.

Tout d'abord on définit des représentations de constantes usuelles. Il est évident que l'on peut représenter la constante 0 par la suite infinie de chiffres 0 dans n'importe quelle base. Comme on l'a vu précédemment, la constante 1 est représentée par la suite infinie du chiffre $\beta - 1$ si l'on est dans la base β.

```
CoFixpoint zero : stream ℤ := 0::zero.

CoFixpoint one (β:ℤ) : stream ℤ := β−1::one β.
```

On peut écrire ensuite l'algorithme de calcul de fonction `sum_div_base` décrit précédemment.

```
CoFixpoint sum_div_base (β : ℤ) (x y: stream ℤ) (k : ℤ) : stream ℤ :=
  match (x, y) with
   | (k1::x', k2::y') =>
      let s := k1 + k2 in
        if  s ≥ β − 1
          then k + 1::sum_div_base β x' y' s − β
          else if s ≤ −β + 1
            then k − 1::sum_div_base β x' y' s + β
            else k::sum_div_base β x' y' s
  end.
```

Puis on pourra alors définir l'addition sur les réels :

```
Definition add (β : ℤ) (r1 r2 : ℤ × streamℤ) : ℤ × streamℤ :=
  let (e₁,x₁):= r₁ in let (e₂,x₂):= r₂ in
    if (e₁ ≤ e₂)
      then (e₂+1, sum_div_base β (add_zero (e₂ − e₁) x₁) x₂)
      else (e₁+1, sum_div_base β x₁ (add_zero (e₁ − e₂) x₂)).
```

5.2. Élaboration des preuves

Une fois que l'on a défini nos algorithmes, on veut montrer qu'ils sont corrects. Notre façon de procéder est de supposer l'existence des nombres réels, ici leur définition standard en Coq, et montrer que notre représentation et nos opérations sont conformes à cette définition. De cette façon, on disposera des propriétés, théorèmes et tactiques existants sur les nombres réels dans les bibliothèques de Coq.

On définit d'abord un prédicat co-inductif qui va nous permettre d'affirmer qu'une mantisse représente bien un certain réel de [-1, 1]. Ce prédicat va utiliser le fait que si $x = k::x'$ alors $x = \frac{k+x'}{\beta}$. De plus, on va se servir de ce prédicat pour assurer qu'une mantisse n'est constituée que de chiffres de la base β.

```
CoInductive represents (β : ℤ): stream ℤ →ℝ →Prop :=
  | rep : ∀ s r k,
    represents β s r →
    −1 ≤ r ≤ 1 →
    −β < k < β →
      represents β k::s (k+r)/β.
```

On peut ensuite définir le prédicat qui va nous permettre de mettre en relation un couple (mantisse, exposant) avec un nombre réel.

```
Inductive represents_R (β : ℤ): ( ℤ * stream ℤ) →ℝ →Prop :=
  | rep_R : ∀ e s r,
    represents β s r →
      represents_R β (e,s) (r × β^e).
```

Ensuite, pour montrer qu'un algorithme sur les mantisses calcule bien une fonction mathématique sur [-1, 1], on va montrer que le prédicat `represents` est une sorte de morphisme entre l'algorithme et la fonction.

On pourra énoncer, par exemple, le théorème de la correction de la fonction `sum_div_base` de la façon suivante :

```
Theorem sum_div_base_correct :
  ∀ β r x y v_x v_y,
    represents β x v_x →
    represents β y v_y →
    −β+2 ≤ r ≤ β−2 →
      represents β (sum_div_base β x y r) ((v_x+v_y+r)/β)
```

On peut alors certifier le calcul de l'addition sur les réels en montrant l'énoncé suivant :

```
Theorem add_correct :
  ∀ β x y v_x v_y,
    represents_R β x v_x →
    represents_R β y v_y →
    represents_R β (add β x y) (v_x + v_y)
```

La partie compliquée des définitions des calculs concerne les algorithmes sur les mantisses. De même les preuves de corrections des calculs sur les mantisses représentent le travail difficile de la certification.

Une première raison est qu'il faut effectuer la preuve d'un prédicat co-inductif, ce qui demande une grande vigilance dans l'application de l'hypothèse de co-récurrence. En effet dans une preuve par co-récurrence, on ne peut appliquer l'hypothèse de co-récurrence que lorsque l'on a montré une partie finie non nulle du théorème. Si elle est mal appliquée ou si la vérification de sa bonne application est trop difficile, la preuve sera rejetée.

De plus, le prédicat `represents` comporte deux doubles inégalités. La première correspond à garantir que le chiffre produit est bien un chiffre de la base. La seconde garantit que le nombre que doit calculer l'appel récursif est bien dans [-1, 1] donc représentable. Ces inégalités représentent une grande partie des preuves. Malheureusement, la plupart ne peuvent pas être prouvées automatiquement par les tactiques *omega* qui résoud des inégalités dans l'arithmétique de Presburger, et *fourier* qui résoud des inégalités linéaires sur les réels. Il a donc fallu constituer une base de nombreux théorèmes pour faciliter ces preuves.

5.3. Fonctions certifiées disponibles

Les constantes et fonctions actuellement décrites et certifiées, c'est à dire dont on a fait la preuve d'un théorème basé sur `represents` ou `represents_R`, sont :

- Les constantes -1, 0, 1 et $e-2$ (la constante d'Euler moins 2) : `minus_one`, `zero`, `one` et `e_minus_2`.
- L'addition, la soustraction, l'opposé, la multiplication : `add`, `sub`, `opp` et `mult`.
- La transformation d'une fraction inférieure à 1 en valeur absolue en une mantisse : `rat_to_stream`
- La fonction `make_digit` certifiée par les deux théorèmes suivants :

```
Theorem make_digit_eq :
  ∀ β, 0 < β →
    ∀ x vx,
      represents β x vx →
      represents β (make_digit β x) vx.

Theorem make_digit_add :
  ∀ β, 0 < β →
    ∀  x k x' vx vy,
      make_digit β x = k::x' →
      represents β x vx →
      −1 ≤ vx + vy ≤ 1  →
      −β − 2 ≤ vy × (2β²) <= β − 2  →
      −1 ≤ (vx + vy) ∗ β − k ≤ 1.
```

- Ainsi que des fonctions annexes nécessaires à la description de ces fonctions : `sum_div_base`, `mult_rat`, `mult_base` ...

6. Analyses des résultats

Le but initial de la formalisation de la base est de pouvoir utiliser des opérations rapides sur les chiffres. Pour l'instant nous ne pouvons utiliser que les opérations standardes sur les entiers de Coq. Néanmoins, nous avons tout de même comparé quelques temps de calculs pour vérifier si déjà dans ces conditions l'utilisation de grandes bases apporte une amélioration. Nous avons donc effectué le même calcul pour différentes bases. Pour que les résultats des calculs soient comparables, nous avons ajusté, pour chaque base, le nombre de chiffres calculés pour obtenir un résultat avec la même précision. Nous avons aussi comparé avec la base LCR de Bertot, correspondant à la base 2. Ces calculs ont été effectués sur une machine équipée de deux processeurs P4 de 3.40GHz et 1Go de mémoire RAM.

– Calcul de $\frac{1}{3} + \frac{1}{7}$

Base utilisée	LCR	2^2	2^8	2^{16}	2^{32}	2^{64}
Nombre de chiffres calculés	6400	3200	800	400	200	100
Temps de calcul (s)	31	6.184	0.440	0.284	0.308	0.412

On voit ici nettement que l'utilisation de grandes bases améliore le calcul de l'addition. Nous pensons que le gain ne vient pas de la complexité mais de la diminution du mécanisme d'appels co-récursif. Cependant on remarque aussi un léger déclin de ce gain pour des bases très grandes. Une explication possible est que les chiffres de la base sont très grands et l'utilisation des entiers standards de Coq n'est pas le type le mieux adapté pour ces chiffres. Ainsi le gain apporté par la diminution d'appels co-récursifs serait finalement rattrapé par la complexité des opérations sur les entiers de Coq.

– Calcul de $\frac{1}{3} \times \frac{1}{7}$

Base utilisée	LCR	2^2	2^8	2^{16}	2^{32}	2^{64}
Nombre de chiffres calculés	128	64	16	8	4	2
Temps de calcul (s)	0.028	16.96	1.412	1.224	1.768	3.696

Encore une fois on peut voir que l'augmentation de la base diminue le temps de calcul. En revanche on se rend compte de la grosse perte de notre multiplication par rapport à celle en LCR. Ceci est dû à la fonction `mult_rat` implantée comme une série et appelée à chaque appel récursif de `mult`. De plus les arguments de `mult_rat` sont des entiers non bornés qui augmentent à chaque appel récursif. En base LCR, le calcul réalisé par cette fonction était direct ce qui explique l'écart des résultats.

– Calcul de $e - 2$

Base utilisée	LCR	10	32	64	100	2^{10}
Nombre de chiffres calculés	300	90	60	50	45	30
Temps de calcul (s)	2.3	19.88	13.60	1.516	3.420	604

Le calcul de $e - 2$ est décrit dans par la série $\sum_{i=2}^{\infty} \frac{1}{i!}$. On peut constater encore une fois que le calcul à tendance à s'améliorer à mesure que la base augmente. On voit aussi que que ce gain se détériore très vite. La raison est simple. Nous avons adapté le calcul de $e - 2$ de façon naïve. Un long calcul initial trop précis est réalisé pour simplifier les appels récursifs. Nous devrions corriger ce problème sans difficulté en ne calculant que ce qui est nécessaire à chaque étape.

7. Conclusion

La formalisation de la base dans la bibliothèque nous a permis d'obtenir un gain de performances constrasté en comparaison du travail de Bertot. En augmentant la base, le temps de calcul diminue car il y a moins d'appels récursifs pour une même précision. Mais nous constatons que ce gain s'essouffle avec de très grandes bases. Nous pensons que la raison du problème est que les calculs que nous effectuons sur les chiffres de très grandes bases ne sont plus adaptés à la représentation des entiers de Coq.

Tous les bénéfices que l'on peut tirer d'un travail en base arbitraire n'ont pas encore été exploités. En effet on pourrait espérer un gain de performances important dans nos calculs de chiffres de la base si cette base permet d'utiliser l'arithmétique du processeur [9]. On pourrait imaginer aussi utiliser une bibliothèque certifiée de calculs efficaces sur les grands nombres [10] pour nos calculs intermédiaires. Enfin nos évaluations ont été réalisées à l'intérieur du système Coq et pourraient être encore améliorées en extrayant le code Ocaml. On pourrait alors aussi envisager d'utiliser une bibliothèque de calcul exacte sur les grands nombres de Ocaml [11]. Ceci permettrait de gagner encore en efficacité mais baisserait le niveau de certification des calculs. C'est pourquoi nous travaillons actuellement à formaliser le type des chiffres et celui des nombres servant aux calculs intermédiaires au moyen de modules. De cette manière nous pourrons greffer facilement des bibliothèques de calcul mieux adaptées directement en Coq ou dans le module Ocaml extrait.

Malheureusement notre adaptation souffre d'un défaut majeur. La multiplication a vu sa complexité augmenter considérablement en raison de l'ajout du calcul d'une série à chaque étape de production d'un chiffre. Contrairement aux autres opérations implantées, la multiplication a vu ses performances baisser en comparaison de la base *LCR*. Ce problème est vraiment critique et nécessite une solution car notre objectif futur est d'utiliser la multiplication pour décrire le calcul de séries formelles. Ainsi nous envisageons de décrire la fonction inverse sous la forme d'une série $\frac{1}{x} = \frac{1}{1-(1-x)} = \sum_{i=0}^{\infty}(1-x)^i$. Nous imaginons aussi décrire des fonctions analytiques en utilisant leur développement de taylor. La description et la vérification de ces séries bénéficieront des avantages de notre modification du calcul des séries par l'isolation de la fonction `make_digit`. En revanche, il est évident que le temps de calcul de ces séries souffrira de la complexité actuelle de la multiplication que nous devons améliorer au plus vite.

Bibliographie

[1] Yves Bertot. Calcul de formules affines et de séries entières en arithmétique exacte avec types co-inductifs. In Thérèse Hardin et Luc Moreau, editor, *Journées francophones des langages applicatifs*. INRIA, Janvier 2006.

[2] 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.

[3] Muller, J.-M. (1997). *Elementary Functions, Algorithms and implementation.* Birkhauser.

[4] Abbas Edalat and Reinhold Heckmann. Computing with real numbers. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 193–267. Springer, 2000.

[5] Coq development team. *The Coq Proof Assistant Reference Manual, version 8.0*, 2004.

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

[7] 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.

[8] Algirdas A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10 :389–400, 1961. URL : `http ://cr.yp.to/bib/entries.html#1961/avizienis`.

[9] Arnaud Spiwack. Ajouter des entiers machine à coq. url http ://arnaud.spiwack.free.fr/papers/nativint.pdf, 2006.

[10] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. A computational approach to pocklington certificates in type theory. In *FLOPS*, volume 3945 of *LNCS*, pages 97–113. Springer, 2006.

[11] Valérie Menissier-Morain and Pierre Weis. An exact arithmetic package for ml.