Vérification de bout en bout d'une fonction de bibliothèque mathématique

Paul Geneau de Lamarlière

Vérification de bout en bout d'une fonction de bibliothèque mathématique

Paul Geneau de Lamarlière[1, 2]

[1]Mitsubishi Electric R&D Centre Europe, 35700 Rennes, France
[2]Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

Il est difficile de trouver des approximations flottantes de fonctions mathématiques qui soient à la fois efficaces et précises. Le code d'une telle fonction est souvent complexe et sujet à l'erreur, ce qui incite à l'utilisation de méthodes formelles, en particulier la preuve formelle, pour augmenter le niveau de confiance. Malheureusement, le manque d'automatisation ou le manque d'articulation entre l'automatisation et la partie manuelle de la preuve font que ce procédé est trop coûteux en pratique. Dans cet article, qui résume deux publications antérieures, nous revisitons ce problème en proposant une méthodologie et une automatisation dédiée, que nous appliquons à la preuve d'une approximation de l'exponentielle sur les flottants en double précision. La particularité de cet exemple est qu'il ne s'agit pas d'une simple modélisation d'un code externe, mais d'une véritable fonction flottante, définie dans la logique de Coq, que nous pouvons utiliser dans les preuves une fois que sa correction a été formellement vérifiée. Cette fonction présente toutes les caractéristiques d'une fonction de l'état de l'art : manipulation de bits, tableaux de constantes, exploitation des subtilités de l'arithmétique flottante, valeurs exceptionnelles, etc. Cette fonction a été intégrée aux stratégies de preuve de la bibliothèque CoqInterval et est vingt fois plus rapide que la précédente version.

1 Introduction

Beaucoup de domaines pratiques, comme l'ingénierie ou le calcul scientifique, utilisent des bibliothèques mathématiques, qui fournissent des approximations de fonctions mathématiques (*e.g.* `math.h` en C). Ces domaines d'application requièrent souvent que les approximations soient suffisamment précises et que leur calcul soit rapide. Pour cela, le code d'une approximation de fonction mathématique est en général complexe et subtil et, par conséquent, sa correction est loin d'être triviale [Mul16, §11-12]. Le calcul de fonctions mathématiques en machine est un sujet de recherche popularisé par Cody et Waite en 1980 [CW80], mais encore très actif aujourd'hui, avec par exemple la bibliothèque CORE-MATH qui constitue l'état de l'art [SZG22].

Une bibliothèque mathématique peut également être utilisée dans des preuves calculatoires. Un exemple est la bibliothèque CoqInterval, qui fournit des stratégies d'automatisation de la preuve de propriétés numériques pour l'assistant de preuve Coq [MMSP19]. Cette dernière est capable, entre autres, de prouver l'encadrement suivant de l'intégrale de Rump en quelques secondes :

$$\int_0^8 \sin(x + \exp x)dx = 0.3474 \pm 10^{-6}.$$

Cette intégrale est connue pour être particulièrement difficile à calculer du fait du nombre important d'oscillations dans l'intégrande. La méthode utilisée par CoqInterval est de diviser le domaine $[0; 8]$ en de nombreux sous-intervalles et de calculer une approximation de l'intégrande sur chacun d'eux. Il est donc important que le calcul de l'encadrement des fonctions mathématiques figurant dans l'intégrande (l'exponentielle et la fonction sinus) soit non seulement correct, mais aussi rapide.

Les approximations de fonctions de CoqInterval calculent avec des nombres flottants, ces derniers permettant de calculer rapidement des valeurs réelles avec une relative précision et couvrant de nombreux ordres de grandeur. À l'origine, les flottants de CoqInterval étaient émulés avec des paires d'entiers arbitrairement grands, la paire (m, e) représentant le réel $m \cdot 2^e$. Cette émulation des flottants a le mérite d'être formellement vérifiée et de permettre des calculs arbitrairement précis, mais pour les calculs qui ne nécessitent pas une précision plus élevée que 53 bits, les flottants du processeur constituent une alternative encore plus rapide. C'est en partie pour cette raison qu'un support pour les flottants primitifs a récemment été ajouté à Coq [MDMR23].

Même en modifiant la bibliothèque CoqInterval pour qu'elle utilise les flottants primitifs, les performances sont encore loin de ce que l'on pourrait espérer de l'utilisation de l'unité flottante du processeur. L'une des principales raisons est que les algorithmes utilisés par CoqInterval, conçus pour la précision arbitraire, sont particulièrement inadaptés aux flottants primitifs et leur précision fixe de 53 bits. Par exemple, l'algorithme utilisé par CoqInterval pour calculer une approximation de l'exponentielle d'un flottant x consiste à réduire l'argument x à l'intervalle $[-2^{-8}; 0]$ grâce aux identités

$$\exp(2x) = (\exp x)^2 \text{ et } \exp(-x) = \frac{1}{\exp x}$$

puis calculer un encadrement de l'exponentielle sur $[-2^{-8}, 0]$ en évaluant, en arithmétique d'intervalle, la série alternée

$$\exp x = 1 + x + \frac{x^2}{2} + \cdots + \frac{x^k}{k} + \cdots$$

à un ordre suffisamment grand.

Un avantage de cette réduction d'argument est qu'elle est applicable à des valeurs de $|x|$ arbitrairement grandes. Cependant, cela n'est utile que si le résultat de l'approximation peut lui-même être arbitrairement grand ou petit. Or, en double précision, une valeur de $\exp x$ supérieure à 2^{1024} entraîne un dépassement de capacité (ou *overflow*), tandis qu'une valeur de $\exp x$ inférieure à 2^{-1075} donne un résultat trivial (l'exponentielle renvoie 0). Ainsi, nous n'avons besoin de calculer une réduction d'argument que sur un domaine initial borné, auquel cas la méthode ci-dessus est beaucoup trop lente. En outre, pour des valeurs de x proches des limites du domaine, *i.e.*, $|x| \simeq 700$, cette réduction d'argument devient très imprécise et donne lieu à des intervalles d'approximation très pessimistes, constituant une perte de précision pouvant aller jusqu'à 20 bits.

Pour cette raison, nous aimerions donc remplacer le code de l'exponentielle de CoqInterval sur les flottants primitifs par une version plus rapide et précise, et utilisant les méthodes de l'état de l'art. Malheureusement, la preuve de correction d'une fonction flottante de l'état de l'art se heurte à de multiples difficultés. D'abord, les subtilités de l'arithmétique flottante impliquent que les preuves sont intrinsèquement longues et verbeuses. Un exemple représentatif, tiré de la bibliothèque CORE-MATH, est une approximation flottante de la fonction puissance $(x, y) \mapsto x^y$, dont la preuve de correction représente une dizaine de pages de calcul [HJZ+24]. Vérifier à la main qu'une telle preuve ne contient pas d'erreurs n'est donc pas raisonnable, ce qui incite à se tourner vers la preuve formelle. La preuve formelle reste

néanmoins pénible, puisqu'elle demande de prendre en compte tous les détails, y compris ceux considérés comme insignifiants et qui sont en général négligés dans les preuves sur papier (comme par exemple la gestion des dépassements de capacité). Une preuve partielle de cette même fonction puissance représente déjà plusieurs milliers de lignes de Coq [1].

Cet article propose un algorithme calculant un encadrement de l'exponentielle avec les flottants en double précision, ainsi que la méthodologie qui a été utilisée pour faciliter la vérification formelle de son implantation en Coq. Cette méthodologie, présentée en section 2, a vocation à être utilisée plus généralement dans le cadre de la vérification de n'importe quelle fonction d'une bibliothèque mathématique. Dans la section 3, nous présentons la nouvelle version de l'exponentielle sur les flottants primitifs pour CoqInterval et nous explicitons quelques détails importants de sa vérification, que nous illustrons avec des exemples. Le travail présenté dans cet article résume celui de deux publications antérieures, l'une présentée au *30th IEEE International Symposium on Computer Arithmetic* [GMF23] et l'autre à la *15th International Conference on Interactive Theorem Proving* [FGM24].

2 Proposition d'une méthodologie pour la vérification de fonctions flottantes

L'arithmétique flottante est subtile, ce qui rend le raisonnement difficile. Premièrement, un calcul flottant peut produire une valeur exceptionnelle (infini ou NaN). Ensuite, les résultats des opérations flottantes sont souvent inexacts à cause des erreurs d'arrondi. L'accumulation des erreurs peut notamment conduire à une dégradation de la précision dans les calculs qui comportent beaucoup d'opérations flottantes.

Supposons par exemple que nous voulons montrer que $x \oplus x = 2 \cdot x$ pour tout flottant x fini [2], où $\oplus$ désigne l'addition flottante, potentiellement inexacte, et $\cdot$ désigne la multiplication réelle, exacte. Premièrement, cette affirmation n'est pas vraie si $x \oplus x$ entraîne un dépassement de capacité puisque dans ce cas, l'addition flottante renvoie $+\infty$. Il faut donc une hypothèse $|x| \leqslant \Omega/2$, avec $\Omega = (2^{53} - 1) \cdot 2^{971}$ le plus grand flottant fini positif, et montrer que sous cette hypothèse l'addition $x \oplus x$ ne produit jamais de dépassement de capacité. Ensuite, comme les opérations flottantes sont arrondies, nous ne pouvons pas déduire immédiatement $x \oplus x = 2 \cdot x$. Il faut d'abord prouver que le résultat de l'addition est exact, ce qui est vrai dans le cas présent puisque $2 \cdot x$ est représentable par un flottant en base 2.

De manière générale, l'énoncé de correction d'une expression flottante e approchant une certaine expression réelle E est une propriété de la forme

$$e \text{ est finie et } \left|\frac{e}{E} - 1\right| \leq \varepsilon.$$

Pour des expressions e comportant beaucoup d'opérations, prouver ce type de propriété à la main est très pénible. Dans cette section, nous proposons donc une méthodologie pour faciliter la preuve de propriétés numériques sur les expressions flottantes. Cette méthodologie prend en compte tous les aspects de la preuve, y compris l'absence de comportements exceptionnels et l'encadrement des erreurs d'arrondi. Pour cela, nous définissons d'abord un langage d'expressions ainsi que les interprétations pertinentes pour notre méthodologie (section 2.1), ensuite nous explicitons le lien entre les interprétations et son intérêt pour les preuves (sections 2.2 et 2.3), enfin nous présentons les outils en Coq qui nous permettront d'alléger les preuves (sections 2.4 et 2.5).

1. Cette preuve a été écrite par Laurence Rideau et Laurent Théry et est disponible au lien suivant : `https://github.com/thery/ExpFloat/blob/master/algoP1.v`

2. Pour un flottant fini x, nous désignons également par x la valeur réelle correspondante.

2.1 Expressions arithmétiques

Les expressions arithmétiques sont représentées avec des arbres de syntaxe abstraite. Les nœuds internes de l'arbre correspondent à des opérations arithmétiques flottantes, incluant donc les opérations élémentaires, mais aussi des fonctions comme la racine carrée ou la partie entière. Les expressions peuvent également contenir des accès dans un tableau de littéraux. Une expression abstraite e peut être interprétée de plusieurs façons, dont deux en particulier qui nous intéressent ici : d'une part, on peut considérer le flottant calculé par e selon la norme IEEE-754 (noté $[\![e]\!]_{\mathsf{flt}}$), et d'autre part, on peut considérer la valeur réelle obtenue en effectuant les opérations de e sur les réels et en arrondissant les résultats (noté $[\![e]\!]_{\mathsf{rnd}}$). Par exemple, pour une addition, les deux interprétations sont définies comme suit :

$$\begin{array}{lcl} [\![u+v]\!]_{\mathsf{flt}} & := & [\![u]\!]_{\mathsf{flt}} \oplus [\![v]\!]_{\mathsf{flt}} \\ [\![u+v]\!]_{\mathsf{rnd}} & := & \circ([\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}}) \end{array}$$

où $\oplus$ est l'addition flottante et $\circ$ est l'arrondi au flottant le plus proche en double précision [3]. L'interprétation $[\![e]\!]_{\mathsf{flt}}$ correspond à la valeur effectivement calculée en machine par e, et par conséquent la valeur sur laquelle porte la propriété de correction que l'on souhaite prouver. L'interprétation $[\![e]\!]_{\mathsf{rnd}}$ est, quant à elle, celle sur laquelle nous aimerions raisonner, puisqu'il s'agit d'une valeur réelle.

L'interprétation $[\![e]\!]_{\mathsf{flt}}$ est définie sur les flottants de la bibliothèque `PrimFloat` [MDMR23]. Cette dernière fournit une spécification axiomatisée des flottants primitifs qui respecte la norme IEEE-754 et qui est prouvée formellement avec Flocq [BM17]. À l'écriture de cet article, certaines opérations arithmétiques ne sont pas disponibles en Coq pour les flottants primitifs, comme par exemple le FMA (pour *fused multiply-add*), l'opération ternaire $a, b, c \mapsto a \cdot b + c$ pour les flottants.

2.2 Relation entre les interprétations

Les interprétations $[\![e]\!]_{\mathsf{flt}}$ et $[\![e]\!]_{\mathsf{rnd}}$ ne sont pas égales a priori : la première représente un calcul flottant qui peut entre autres produire une valeur exceptionnelle, alors que la seconde est une expression réelle. Or, la correction de l'interprétation flottante $[\![e]\!]_{\mathsf{flt}}$ repose bien souvent sur le fait qu'elle soit égale à l'interprétation réelle $[\![e]\!]_{\mathsf{rnd}}$. Il nous faut donc expliciter le lien entre ces deux interprétations, surtout si l'on souhaite pouvoir passer d'un raisonnement sur $[\![e]\!]_{\mathsf{flt}}$ à un raisonnement sur $[\![e]\!]_{\mathsf{rnd}}$, plus simple.

Pour expliciter le lien entre ces deux interprétations, nous avons défini un prédicat WB (pour *well-behaved*) qui caractérise les expressions qui se comportent bien. Une expression flottante se comporte bien si sa valeur est finie et égale à la valeur de l'expression réelle correspondante. Par exemple, pour qu'une division se comporte bien, il faut que les deux opérandes se comportent bien, que le diviseur soit différent de zéro et que le résultat de la division n'entraîne pas de dépassement de capacité :

$$\mathrm{WB}(u/v) \triangleq \mathrm{WB}(u) \wedge \mathrm{WB}(v) \wedge [\![v]\!]_{\mathsf{rnd}} \neq 0 \wedge |\circ([\![u]\!]_{\mathsf{rnd}}/[\![v]\!]_{\mathsf{rnd}})| \leq \Omega \tag{1}$$

Nous avons alors prouvé que toute expression e qui satisfait $\mathrm{WB}(e)$ est bien définie, autrement dit, le théorème suivant :

Théorème 1 (Correspondance)**.** *Si e est une expression abstraite, alors*

$$\mathrm{WB}(e) \implies [\![e]\!]_{\mathsf{flt}} \textit{ est fini et } [\![e]\!]_{\mathsf{rnd}} = [\![e]\!]_{\mathsf{flt}}.$$

3. Plus précisément, $\circ$ est la fonction qui à un réel x associe le réel f le plus proche de x qui vérifie $f = m \cdot 2^e$ pour un certain entier $m \in [\![-2^{53}+1;\, 2^{53}-1]\!]$ et un certain entier $e \geq -1074$. En particulier, $\circ(x)$ peut être arbitrairement grand.

2.3 Suppléments au langage

Un avantage de notre approche est qu'il est assez facile d'enrichir les expressions abstraites avec des éléments d'information utiles dans les preuves. En particulier, il est possible de décrire des éléments couramment rencontrés dans les approximations de l'état de l'art. Un exemple est la présence d'opérations dont il est indispensable qu'elles ne produisent pas d'erreur d'arrondi (ce qu'il faudra prouver lors de la correction de l'algorithme). Nous avons donc décidé d'ajouter ces opérations exactes à notre langage d'expressions. Par exemple, pour l'addition exacte, les interprétations sont définies comme suit :

$$\begin{array}{lcl} [\![u +_{\mathsf{exact}} v]\!]_{\mathsf{flt}} & \triangleq & [\![u]\!]_{\mathsf{flt}} \oplus [\![v]\!]_{\mathsf{flt}} \\ [\![u +_{\mathsf{exact}} v]\!]_{\mathsf{rnd}} & \triangleq & [\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}} \end{array}$$

Pour que le théorème 1 reste valable, il faut également définir WB comme suit [4] :

$$\mathrm{WB}(u +_{\mathsf{exact}} v) \triangleq \begin{cases} \mathrm{WB}(u) \wedge \mathrm{WB}(v) \wedge \\ \circ([\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}}) = [\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}} \wedge \\ |([\![u]\!]_{\mathsf{rnd}} + [\![v]\!]_{\mathsf{rnd}})| \leq \Omega \end{cases}$$

Beaucoup de bibliothèques mathématiques exploitent également les subtilités de l'arithmétique flottante pour calculer rapidement certains résultats utiles. Par exemple, un algorithme simple pour calculer l'entier le plus proche d'un flottant en double précision est de lui ajouter $3 \cdot 2^{51}$, puis de soustraire $3 \cdot 2^{51}$ au résultat. Nous aimerions pouvoir utiliser cette astuce dans nos fonctions sans avoir à la prouver à chaque fois. Pour cela, nous avons ajouté une macro-opération `FastNearbyint` dans notre langage d'expressions, dont l'interprétation flottante est l'algorithme précédent, et dont l'interprétation réelle est l'entier le plus proche (que nous notons ici $\lceil \cdot \rfloor$) :

$$\begin{array}{lcl} [\![\texttt{FastNearbyint}\ e]\!]_{\mathsf{flt}} & \triangleq & [\![e]\!]_{\mathsf{flt}} \oplus \texttt{0x1.8p52} \ominus \texttt{0x1.8p52} \\ [\![\texttt{FastNearbyint}\ e]\!]_{\mathsf{rnd}} & \triangleq & \lceil [\![e]\!]_{\mathsf{rnd}} \rfloor \end{array}$$

Notons que l'algorithme ne fonctionne pas pour tous les flottants. Par exemple, si le flottant de départ est $2^{51}+1$ (un entier), ajouter puis soustraire $3 \cdot 2^{51}$ donnera la valeur 2^{51}, alors que la partie entière reste égale à $2^{51}+1$. Nous devons donc définir WB en conséquence :

$$\mathrm{WB}(\texttt{FastNearbyint}\ e) \triangleq \mathrm{WB}(e) \wedge |[\![e]\!]_{\mathsf{rnd}}| \leq 2^{51}$$

La correction de ce court algorithme est alors prouvée une fois pour toutes dans le théorème 1, et nous pouvons ainsi l'utiliser dans la vérification d'autres fonctions.

2.4 Automatisation de la preuve de $\mathrm{WB}(e)$

La bibliothèque CoqInterval permet de prouver formellement des encadrements d'expressions réelles. Elle fonctionne en calculant une approximation polynomiale de l'expression ainsi qu'un encadrement de la distance entre ce polynôme et l'expression initiale. On peut donc utiliser cette méthode pour prouver automatiquement une grande partie des conditions de $\mathrm{WB}(e)$. Mais pour cela, nous devons apporter une modification majeure à la bibliothèque CoqInterval puisque cette dernière ne reconnaît pas les fonctions d'arrondi, alors que la plupart des expressions présentes dans $\mathrm{WB}(e)$ en contiennent. Nous avons donc ajouté à CoqInterval l'encadrement suivant, étant donné un encadrement $[\underline{u}, \overline{u}]$ de u :

$$\circ(u) - u \in [-\tfrac{\varepsilon}{2}; \tfrac{\varepsilon}{2}] \quad \text{avec } \varepsilon = \mathrm{ulp}(\max(-\underline{u}, \overline{u})).$$

où ulp (pour *unit in the last place*) est la fonction qui à un flottant x associe la distance entre $|x|$ et son successeur.

4. Notons que l'exactitude de l'opération permet de supprimer l'arrondi dans l'inégalité par rapport à l'équation (1)

Pour prouver WB(e) automatiquement, nous pourrions simplement appeler manuellement CoqInterval sur chacune de ses clauses, mais cela ne serait pas efficace. En effet, les expressions présentes dans les différentes clauses de WB(e) sont fortement corrélées et calculer indépendamment un encadrement de chacune d'elles conduirait à beaucoup de calculs redondants. Nous avons donc apporté une autre modification à CoqInterval pour lui permettre de calculer efficacement un encadrement d'une liste d'expressions. Nous avons ensuite défini une tactique `simplify_wb` qui prouve rapidement un maximum de clauses de WB(e).

2.5 Fluidification du processus de preuve

Le code des approximations flottantes étant long et complexe, la preuve de leur correction consiste souvent à étudier le code progressivement : on commence par prouver une propriété intermédiaire sur une partie du code, puis cette propriété nous permet de déduire une propriété sur la suite du code, et ainsi de suite. Concrètement, avec notre méthodologie, cela signifie que la preuve de correction d'une fonction flottante consiste à répéter ces quelques étapes :

— énoncer une assertion intermédiaire de la forme "$[\![e]\!]_{\mathsf{flt}}$ est fini et sa valeur réelle satisfait un certain prédicat P" sur une sous-expression e ;
— appliquer le théorème 1 pour obtenir deux sous-buts, $P([\![e]\!]_{\mathsf{rnd}})$ et WB(e) ;
— prouver WB(e) automatiquement avec `simplify_wb`.

Nous avons créé une tactique `assert_float` qui rassemble ces trois étapes. Par exemple, dans la preuve de la nouvelle version de l'exponentielle que nous présentons à la section suivante, nous avons besoin de montrer que l'expression `k` dans le Listing 1 (lignes 4 et 5) se situe entre -68736 et 65536, ce qui se traduit par le script Coq suivant :

```
1 set (k0 := FastNearbyint (Op MUL (Var 0) InvLog2_64)).
2 change (_ - 0x1.8p52)%float with (evalPrim k0 [:x:]).
3 assert_float (fun (k : R) => -68736 <= k <= 65536).
4 { rewrite round_FIX_IZR. interval. }
```

Dans ce script, nous explicitons d'abord l'expression abstraite k_0 correspondant à l'expression flottante `k` (ligne 1 du script). Comme l'expression utilise l'algorithme rapide présenté en section 2.3, nous l'explicitons par l'emploi du constructeur `FastNearbyint`. Ensuite, nous remplaçons `k` par $[\![k_0]\!]_{\mathsf{flt}}$ dans le but (ligne 2). Nous utilisons la tactique `assert_float` pour énoncer notre propriété (ligne 3). Enfin, nous prouvons cette propriété sur $[\![k_0]\!]_{\mathsf{rnd}}$ (ligne 4). Quant à l'obligation de preuve WB(k_0) due à l'application du théorème 1, elle est immédiatement déchargée par `simplify_wb`.

3 Application : une meilleure approximation de l'exponentielle pour CoqInterval

Le listing 1 montre le code de la nouvelle version de l'exponentielle pour CoqInterval, que nous appelons `iexp`. La correction de la fonction `iexp` peut être énoncée comme suit :

Théorème 2. *Soit x un flottant fini et $(\underline{y}, \overline{y})$ la paire de flottants renvoyée par `iexp` x. Alors*

$$\underline{y} \leq \exp x \leq \overline{y}.$$

La preuve formelle de ce théorème étant longue et subtile, nous allons appliquer notre méthodologie pour la simplifier. Nous présentons d'abord les subtilités du code (section 3.1), puis nous montrons des exemples où notre méthodologie permet de réduire l'effort de preuve (section 3.2) et enfin nous dressons un bilan des performances de l'exponentielle (section 3.3).

```
1  Definition iexp x :=
2    if x < -0x1.74385446d71c4p9 then (0, 0x1.p-1074) else
3    if x > 0x1.62e42fefa39efp9  then (0x1.fffffffffffp1023, infinity) else
4    let k' :=  x * invLog2_64 + 0x1.8p52 in
5    let k  :=  k' - 0x1.8p52 in
6    let t  := (x - k * log2_64h) - k * log2_64l in
7    let y  :=  t * (p1 + t * (p2 + t * (p3 + t * (p4 + t * p5)))) in
8    let ki := (mantissa k' - 0x18000000000000)%uint63 in
9    let p0 :=  cst.(ki land 63) in
10   let d  :=  0x1.25p-57 in
11   let lb :=  p0 + (p0 * y - d) in let ub :=  p0 + (p0 * y + d) in
12   next_down (ldexp lb (ki asr 6)), next_up (ldexp ub (ki asr 6))
```

Listing 1. Code de la nouvelle version de l'exponentielle pour CoqInterval. Les symboles `invLog2_64`, `log2_64h`, `log2_64l`, `cst`, `p1`, `p2`, etc. sont des constantes prédéfinies.

3.1 Explication du code

Notre fonction prend en entrée un flottant x et renvoie une paire de flottants qui encadrent $\exp x$. Si $x < \ln\omega$ avec $\omega = 2^{-1074}$ le plus petit flottant strictement positif, on renvoie immédiatement la paire $(0,\ \omega)$ (ligne 2). À l'opposé, si $x > \ln\Omega$ avec Ω le plus grand flottant positif, on renvoie immédiatement la paire $(\Omega,\ +\infty)$ (ligne 3). Dans les autres cas, on effectue d'abord une réduction d'argument à partir de l'identité suivante (lignes 4 à 6) :

$$\exp(x) = \exp\left(x - k\cdot\frac{\ln 2}{64}\right)\cdot 2^{k/64} \quad \text{avec} \quad k = \left\lceil x\cdot\frac{64}{\ln 2}\right\rfloor. \tag{2}$$

Le calcul de k utilise l'algorithme de la section 2.3 pour calculer l'entier le plus proche d'un flottant. Pour le calcul de $t := x - k\cdot\ln 2/64$, la constante $\ln 2/64$ est approchée par une paire de flottants (c_1, c_2) telle que $c_1 \simeq \ln 2/64$ et $c_2 \simeq \ln 2/64 - c_1$. La constante c_1 est choisie de sorte que la multiplication flottante $k \otimes c_1$ ne produit pas d'erreur d'arrondi. Cette réduction d'argument s'inspire grandement de celle de Cody et Waite [CW80] et présente donc les mêmes subtilités.

L'argument réduit t est donc compris dans l'intervalle $[-\ln 2/128; \ln 2/128]$ [5]. On calcule ensuite une approximation y de $\exp t - 1$ avec un polynôme de degré 5 (ligne 7). Les coefficients p_1, p_2, p_3, p_4 et p_5 sont des flottants. Pour compenser l'erreur de cette approximation, on utilise un terme correcteur $d \sim 1.1\cdot 2^{-57}$ que l'on soustrait pour obtenir la borne inférieure et que l'on ajoute pour obtenir la borne supérieure (lignes 10 et 11).

Enfin, pour reconstruire l'exponentielle de x, il faut multiplier par $2^{k/64}$. Pour cela, on calcule la division euclidienne de k par 64 pour obtenir k_q et k_r (resp. `ki asr 6` et `ki land 63` dans le code) tels que $k = 64\cdot k_q + k_r$ et $0 \le k_r \le 63$. On peut alors décomposer la multiplication par $2^{k/64}$ en une multiplication par $2^{k_r/64}$ et une multiplication par 2^{k_q}. Cette dernière est triviale pour les flottants binaires. Quant à la multiplication par $2^{k_r/64}$, nous avons au préalable calculé le flottant le plus proche de $2^{i/64}$ pour chaque valeur de i comprise entre 0 et 63, et stocké le résultat dans un tableau de taille 64 (`cst` dans le code). Notons que pour minimiser l'erreur, le calcul de $2^{k_r/64}\cdot P(t) \pm d$ (avec P le polynôme d'approximation) est effectué comme $p_0 + p_0\cdot y \pm d$ et non $p_0\cdot(1+y)\pm d$ (ligne 11).

En dernier lieu, après avoir multiplié par 2^{k_q} (avec la primitive `ldexp`) la borne inférieure `lb` et la borne supérieure `ub`, pour compenser l'erreur sur p_0 et l'addition finale, on ne renvoie pas directement `lb` et `ub` mais respectivement le prédécesseur et le successeur de ces valeurs, c'est-à-dire respectivement le plus grand flottant strictement inférieur et le plus petit flottant strictement supérieur (ligne 12).

5. La valeur de $|t|$ peut être légèrement supérieure à $\ln 2/128$ à cause des erreurs dans le calcul de k, mais cela n'a pas d'incidence sur la correction de l'algorithme.

3.2 Illustration de la méthodologie de preuve

La première partie subtile de l'algorithme est la réduction d'argument. Nous devons montrer, entre autres, que le calcul de `t` n'entraîne pas de comportement exceptionnel et que la valeure réelle t de `t` vérifie la propriété suivante :

$$|t| \leq \frac{355}{65536} \wedge |t - k \cdot \ln 2| \leq 65537 \cdot 2^{-77}$$

Pour cela, nous explicitons l'expression abstraite t dont l'interprétation dans les flottants primitifs est `t`. Comme certaines opérations de `t` ne produisent jamais d'erreur d'arrondi et que cela est crucial dans la preuve, nous l'explicitons en utilisant le constructeur `OpExact`. Nous utilisons ensuite la tactique `assert_float` pour énoncer la propriété précédente et la prouver directement sur $[\![t]\!]_{\mathsf{rnd}}$:

```
set (t' := Op SUB (OpExact SUB (Var 1) ...) ...).
change (x - _ - _) with (evalPrim t' [:k, x:]).
assert_float (fun t => abs t <= 355 / 65536
                    /\ abs (t - (x - k * ln 2) <= 65537 * pow2 (-77))).
```

Après l'exécution de la tactique `assert_float`, nous avons trois nouveaux sous-buts : la propriété sur $[\![t]\!]_{\mathsf{rnd}}$, mais aussi les clauses correspondant aux opérations exactes dans $\mathrm{WB}(t)$, que CoqInterval n'est pas en mesure de prouver automatiquement. Nous devons donc prouver ces clauses à la main, ce qui constitue la principale difficulté de cette partie de la preuve.

Une fois que nous avons prouvé notre propriété intermédiaire sur l'argument réduit, la prochaine étape est de borner l'erreur sur l'approximation polynomiale. Les approximations polynomiales étant fréquemment utilisées dans les bibliothèques mathématiques, cette partie de la preuve est commune à beaucoup de fonctions mathématiques et constitue souvent l'objectif principal. Malheureusement, il s'agit aussi de la partie la plus pénible du fait de son côté purement calculatoire. Notre méthodologie permet cependant de la résumer à seulement quelques lignes de Coq :

```
change (Papprox t') with (evalPrim g0 [:t':]).
assert_float (fun y => abs y <= 0.0055
                    /\ abs (1 + y - exp t) <= 11 * pow2 (-62)).
{ split.
  - interval.
  - interval with (i_taylor t, i_bisect t, i_prec 80). }
```

Ce script montre que l'évaluation polynomiale n'entraîne pas de dépassement de capacité, que le résultat de l'évaluation est un flottant inférieur à 0.0055 en valeur absolue et que l'erreur totale (approximation et arrondis) est inférieure à $11 \cdot 2^{-62}$.

Nous utilisons la tactique `assert_float` 8 fois en tout dans la preuve. Un autre exemple où nous l'utilisons est pour énoncer une propriété sur le coefficient p_0, que l'on récupère en regardant l'entrée d'indice k_r dans le tableau de constantes `cst`, et qui est le flottant le plus proche de $2^{k_r/64}$:

```
change (consts.[kr]) with (evalPrim (ArrayAcc consts (Var 0)) [:kr:]).
assert_float (fun C => 0.984375 <= C <= 1.984375 /\
 (to_Z kr = 0%Z -> C = 1) /\
  abs (C - exp (IZR (to_Z kr) * (ln 2 / 64))) <= pow2 (-53)).
```

Enfin, il peut arriver dans la preuve que nous ayions besoin d'isoler une sous-expression flottante uniquement pour montrer qu'elle se comporte bien. Nous avons pour cela une version dégénérée de la tactique `assert_float`. Par exemple, pour montrer que $p_0 + y \cdot p_0$ ne produit pas de dépassement de capacité, nous faisons simplement :

```
change (C' + y'')%float with (evalPrim (Op ADD (Var 0) (Var 1)) [:C', y'':]).
assert_float.
```

3.3 Performances

La nouvelle version de l'exponentielle sur les flottants primitifs permet environ 840 000 exécutions par seconde et est à peu près vingt fois plus rapide que la précédente, pour un intervalle de sortie de 2 ulps dans 98% des cas et 3 ulps sinon, ce qui équivaut à une perte de précision maximale de 2 bits (contre 20 bits pour la version précédente).

Pour une fonction Coq, nous ne pouvons pas faire beaucoup mieux. D'une part, l'absence de FMA pour les flottants primitifs en Coq nous oblige à utiliser une multiplication et une addition pour évaluer des expressions de la forme $a \cdot b + c$, ce qui prend plus de temps et cause deux erreurs d'arrondi au lieu d'une seule. D'autre part, calculer des expressions dans la logique de Coq est intrinsèquement lent.

Pour avoir une meilleure idée des performances de notre approximation de l'exponentielle, nous avons donc codé notre fonction en C en utilisant l'opération FMA [6] et nous l'avons comparée à la bibliothèque GLIBC. Nous avons trouvé que notre fonction était environ 1.5 fois moins rapide, pour une erreur environ deux fois plus grande en moyenne. Pour améliorer les performances de notre algorithme, nous pourrions utiliser un tableau de constantes plus grand pour diminuer davantage le degré du polynôme. Nous pourrions aussi améliorer la précision de l'algorithme en stockant la valeur de p_0 sur deux flottants au lieu d'un seul, et en choisissant judicieusement la manière de calculer $p_0 \cdot (1 + y)$.

4 Conclusion

Au total, la preuve de l'exponentielle comporte environ 450 lignes de Coq (dont environ 200 lignes pour la reconstruction et 100 lignes pour la réduction d'argument), ce qui constitue une réduction considérable de la taille par rapport aux précédentes preuves formelles de fonctions flottantes. Il existait déjà une preuve formelle de l'exponentielle de Cody et Waite, mais cette dernière négligeait les comportements exceptionnels et représentait les flottants avec des nombres réels [BM17, §6.2.3]. Grâce à notre approche, la gestion des comportements exceptionnels est quasiment gratuite. L'exponentielle de Cody et Waite est également moins subtile que notre version, ne comprenant pas d'accès dans un tableau de constantes et présupposant l'existence d'une primitive calculant l'entier le plus proche d'un flottant.

À l'écriture de cet article, le langage d'expressions est en passe d'être remplacé par une version plus générique (avec un constructeur pour les constantes, un constructeur pour les opérations unaires, etc.). Ce changement a pour but de faciliter l'ajout de nouvelles opérations et macro-opérations couramment utilisées dans les fonctions flottantes. Un autre exemple de macro-opération est FastTwoSum, qui prend en entrée deux flottants a et b tels que $|a| > |b|$ et renvoie une paire de flottants (s, e) avec s le résultat de l'addition flottante de a et b, et $e = a + b - s$ l'erreur d'arrondi de l'addition. Un autre exemple de macro-opération que nous pourrions ajouter est tout simplement la réduction d'argument de Cody et Waite, dont la preuve constituait une partie significative de la vérification formelle de notre exponentielle. Pour aller plus loin, nous pourrions créer un outil en Coq pour générer directement les approximations polynomiales (par exemple en utilisant la méthode de Remez ou du mini-max) ainsi qu'une borne d'erreur et une preuve à l'aide de la bibliothèque CoqInterval.

Références

[BM17] Sylvie Boldo et Guillaume Melquiond : *Computer Arithmetic and Formal Proofs*. ISTE Press – Elsevier, 2017.

6. La version C de notre exponentielle ne calcule plus un encadrement mais un simple flottant, ainsi le code de la fonction se termine par `ldexp (fma (p0, y, p0), ki asr 6)`. En particulier, le terme correcteur d n'est plus présent.

[CW80] William J. Cody, Jr. et William Waite : *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[FGM24] Florian Faissole, Paul Geneau de Lamarlière et Guillaume Melquiond : End-to-end formal verification of a fast and accurate floating-point approximation. *In* Yves Bertot, Temur Kutsia et Michael Norrish, éditeurs : *Proceedings of the 15th International Conference on Interactive Theorem Proving*, volume 309 de *Leibniz International Proceedings in Informatics*, pages 14:1–14:18, Tbilisi, Georgia, septembre 2024.

[GMF23] Paul Geneau de Lamarlière, Guillaume Melquiond et Florian Faissole : Slimmer formal proofs for mathematical libraries. *In* Theo Drane et Anastasia Volkova, éditeurs : *30th IEEE International Symposium on Computer Arithmetic*, Portland, OR, USA, septembre 2023.

[HJZ+24] Tom Hubrecht, Claude-Pierre Jeannerod, Paul Zimmermann, Laurence Rideau et Laurent Théry : Towards a correctly-rounded and fast power function in binary64 arithmetic. 2024.

[MDMR23] Érik Martin-Dorel, Guillaume Melquiond et Pierre Roux : Enabling floating-point arithmetic in the Coq proof assistant. *Journal of Automated Reasoning*, 67, 2023.

[MMSP19] Assia Mahboubi, Guillaume Melquiond et Thomas Sibut-Pinote : Formally verified approximations of definite integrals. *Journal of Automated Reasoning*, 62(2) :281–300, 2019.

[Mul16] Jean-Michel Muller : *Elementary Functions, Algorithms and Implementation*. Birkhäuser, Boston, MA, 3rd édition, 2016.

[SZG22] Alexei Sibidanov, Paul Zimmermann et Stéphane Glondu : The CORE-MATH project. *In 29th IEEE Symposium on Computer Arithmetic*, septembre 2022.