HAL open science

Comment dompter un troupeau de flottants sauvages ?

Arthur Correnson

To cite this version:

Arthur Correnson. Comment dompter un troupeau de flottants sauvages ?. JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs, Jan 2023, Praz-sur-Arly, France. pp.265-273. hal-03936872

HAL Id: hal-03936872

https://inria.hal.science/hal-03936872v1

Submitted on 12 Jan 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Comment dompter un troupeau de flottants sauvages ?

Arthur Correnson

CISPA Helmholtz Center for Information Security
arthur.correnson@cispa.de
École Normale Supérieure de Rennes
arthur.correnson@ens-rennes.fr

Résumé

Les nombres flottants tels que définis dans le standard IEEE-754 sont la solution la plus répandue pour approximer les nombres réels en machine. Toutefois, en pratique, de nombreux environnements logiciels et matériels présentent des subtilités non conformes avec ce standard. Cela complique la tâche de raisonner formellement sur les programmes effectuant des calculs numériques sans connaître *a priori* leur environnement d'exécution. Dans cet article, nous montrons l'impact de ce problème sur le compilateur CompCert, un compilateur formellement vérifié pour le langage C et proposant un support pour les flottants IEEE-754. Nous présentons les résultats d'un travail d'investigation visant à mesurer les difficultés liées à l'intégration de nouveaux types de flottants dans CompCert. Cette étude est motivée par la volonté d'étendre le compilateur à des cibles non conformes IEEE-754 utilisées, par exemple, en informatique embarquée.

Introduction

Contexte Représenter de manière exacte les nombres réels est impossible dans la mémoire finie d'un ordinateur. Les nombres flottants répondent à ce problème et propose une méthode pour approximer les nombres réels de manière finie au prix d'une perte de précision. Le standard IEEE-754 [2] normalise la notion de nombres flottants et définit un encodage binaire de ces nombres ainsi que le comportement attendu des opérations arithmétiques associées.

Malgré l'existence d'un standard pour l'implémentation des nombres flottants, de nombreux environnements logiciels et matériels présentent des spécificités non conformes IEEE-754. Ces particularités résultent souvent de contraintes techniques comme par exemple le coût de fabrication d'un processeur flottant ou la rapidité des calculs. Cette grande diversité des nombres flottants pose un véritable problème de compréhension des résultats produits par les programmes effectuant des calculs numériques. En effet, sans connaître au préalable l'environnement dans lequel va être exécuté un programme, anticiper le résultat des opérations flottantes devient pratiquement impossible. Ce problème est d'autant plus flagrant dans le contexte de la programmation certifiée dont le but est de prouver formellement que le comportement d'un programme est préservé par la compilation. Dans ce contexte, il est nécessaire de s'entendre sur l'interprétation des nombres flottants dans les langages de programmation mais également dans les environnements d'exécution ciblés par la compilation.

Le compilateur CompCert [10, 11, 9] est un compilateur certifié pour le langage C entièrement prouvé grâce à l'assistant de preuve Coq [1]. CompCert fait le choix de fixer l'usage du standard IEEE-754 [5], limitant ainsi la compilation à des cibles dont le traitement des flottants est conforme au standard. Toute la chaîne de compilation de CompCert dépendant fortement de ce choix, l'intégration d'un support pour de nouveaux types de nombres flottants pose plusieurs défis en matière d'ingénierie logicielle. Une première problématique est de rendre interchangeable le modèle de calcul flottant du compilateur sans invalider la chaîne de compilation et sa preuve de correction. Au delà de cette question, la tâche de formaliser en Coq de nouveaux

modèles de calcul flottant est difficile en elle-même et impose en particulier de s'assurer que les spécifications en Coq sont bien conformes aux documentations de référence.

Contributions Dans cet article nous présentons les résultats d'un travail d'investigation visant à mesurer les difficultés liées à l'intégration de nouveaux types de flottants dans CompCert. Cette étude effectuée au sein de l'entreprise AbsInt est motivée par la volonté d'étendre le compilateur à des cibles non conformes IEEE-754 utilisées notamment en informatique embarquée.

Plan Nous commençons par introduire le problème de la diversité des nombres flottants en présentant des exemples concrets (1). Nous exposons ensuite la notion de compilation certifiée et en particulier la compilation des nombres flottants dans le compilateur CompCert (2). Enfin, nous discutons des difficultés engendré par l'intégration de nouveaux types de flottants dans CompCert (3) et nous proposons plusieurs pistes de réflexion (4).

1 Les nombres flottants et le standard IEEE-754

Les nombres flottants IEEE Le standard IEEE-754 définit une représentation binaire des nombres flottants sur 16, 32 et 64 bits. Pour le cas des flottants 32 bits par exemple, l'encodage comporte 1 bit de signe (`s`), 8 bits d'exposant (`exp`) et les 23 bits restants pour représenter une partie fractionnaire (`frac`). En fonction des valeurs exactes de chaque champs, les nombres flottants sont interprétés au choix parmi 5 types de valeurs :

Valeurs	Interprétation	condition
NaN	valeur indéterminée	$\texttt{exp} = 255$, $\texttt{frac} > 0$
$+\infty, -\infty$	infinis signés en fonction du bit `s`	$\texttt{exp} = 255$, $\texttt{frac} = 0$
$+0, -0$	zéros signés en fonction du bit `s`	$\texttt{exp} = 0$, $\texttt{frac} = 0$
Nombres normalisés	$(-1)^{\texttt{s}} \times 2^{\texttt{exp}-127} \times (1.\texttt{frac})_2$	$1 \leq \texttt{exp} < 255$
Nombres dénormalisés	$(-1)^{\texttt{s}} \times 2^{\texttt{exp}-127} \times (0.\texttt{frac})_2$	$\texttt{exp} = 0$, $\texttt{frac} > 0$

Le standard introduit également plusieurs notion d'arrondi associant à tout réel x une représentation flottante $\texttt{round}_m(x)$ (m est un mode d'arrondi). Les opérations arithmétiques sont ensuite définies comme devant être l'arrondi de leur pendant réel. Par exemple pour l'addition flottante en mode d'arrondi m (notée $\oplus_m$), on a $x \oplus_m y := \texttt{round}_m(x + y)$. Notons que la présence d'un arrondi dans la définition suffit à invalider certaines propriétés arithmétiques comme l'associativité de l'addition.

Le standard et le mode d'arrondi par défaut Parmi les 5 modes d'arrondi définis dans le standard IEEE-754, le mode "au plus proche pair" arrondit tout réel au plus proche flottant dont la représentation binaire a un bit de poids faible pair. Ce mode spécifique est utilisé par défaut dans de très nombreux environnements, si bien que l'on précise rarement le mode d'arrondi explicitement dans les langages de programmation. Cependant, il est parfois préférable d'utiliser d'autres modes pour effectuer certains calculs. Ce choix d'un mode d'arrondi est donc un facteur qui contribue aux différences observables entre environnements de calcul flottants, y-compris entre environnements conformes au standard IEEE-754.

Des imprécisions au niveau logiciel Les langages de programmation ne sont pas toujours précis en ce qui concerne l'arithmétique flottante. La norme ISO du langage C [8] par exemple

laisse une grande liberté dans le traitement des opérations flottantes (voir figure 1). En particulier, l'usage du standard IEEE-754 n'est pas requis mais plutôt recommandé. Il en résulte qu'un même programme C peut produire des résultats flottants différents selon le compilateur utilisé, la machine qui exécute le programme ou même selon la machine qui exécute le compilateur. Ces imprécisions du langage C ainsi que d'autres langages sont davantage détaillées par Sylvie Boldo et ses contributeurs [6, 4].

> *The accuracy of the floating-point operations (`+`, `-`, `*`, `/`) and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results is implementation-defined. The implementation may state that the accuracy is unknown*

FIGURE 1 – Extrait de *ISO/IEC 9899, section 5.2.4.2.2 "characteristics of floating types"*

Des divergences au niveau matériel La plupart des processeurs modernes embarquent une unité de calcul flottant. Les processeurs grand-public que l'on trouve dans les ordinateurs personnels implémentent généralement le standard IEEE-754. Toutefois, certaines familles de processeurs utilisés dans des domaines spécifiques (informatique embarquée, traitement du signal, etc) prennent des libertés par rapport au standard. Par exemple, le standard SPE (Signal Processing Engine) et un document normatif décrivant une famille d'architectures matérielles dédiées au traitement du signal. Cette norme diffère explicitement du standard IEEE en autorisant que les opérations usuelles ne produisent pas de valeurs spéciales (infinis, NaN) ou de nombres dénormalisés (voir figure 2). Des règles spécifiques sont définies en remplacement.

> *Embedded floating-point operations do not produce +Inf, –Inf, NaN, or a denormalized number. [...] Thus, whenever an input operand of the embedded floating-point instruction has data values that are +infinity, -infinity, alized, NaN, or when the result of an operation produces an overflow or an underflow, an embedded floating-point data interrupt may be taken and the interrupt handler is responsible for delivering IEEE 754–compliant behavior if desired*

FIGURE 2 – Extrait de *Signal Processing Engine (SPE) Programming Environments Manual, section 3.3.1.4 about IEEE Std 754 Compliance*

2 Vérifier la compilation des nombres flottants

2.1 Préserver la sémantique

La compilation certifiée a pour but de s'assurer que le sens des programmes est conservé lorsqu'ils sont traduits en exécutables. Cela permet de garantir que l'intention du programmeur n'est pas détériorée par le compilateur, par exemple en effectuant des optimisations hasardeuses. Le caractère parfois contre-intuitif de l'arithmétique flottante est typiquement source

d'optimisations incorrectes. Par exemple, en nombres flottants, $x \oplus 0.0$ ne peut pas toujours être optimisé en x. En effet, dans le cas où $x = -0.0$, le standard IEEE impose que le résultat de $-0.0 + 0.0 = +0.0$ et $+0.0 \neq -0.0$. Certifier un compilateur permet de contrôler ce type d'erreurs.

Pour garantir qu'un compilateur préserve le sens des programmes, il est nécessaire de définir au préalable la sémantique des langages sources et cibles considérés pour la compilation. Pour un langage donné $\mathcal{L}$, un programme $P \in \mathcal{L}$ et un comportement observable C, on définit la relation $P \Downarrow C$ qui se lit comme "C est un comportement observable lors de l'exécution du programme P". Dans le cadre de la compilation d'un langage $\mathcal{L}_{source}$ vers $\mathcal{L}_{cible}$, un compilateur peut être vu comme une fonction partielle `compile` : $\mathcal{L}_{source} \rightharpoonup \mathcal{L}_{cible}$. Étant données deux sémantiques $\Downarrow_{source}$ et $\Downarrow_{cible}$ décrivant l'exécution des programmes sources et cibles, certifier le compilateur `compile` revient alors à démontrer le théorème suivant [10] :

$$\boxed{\forall P \in \mathcal{L}_{source}, \forall P' \in \mathcal{L}_{cible}, \forall C, \texttt{compile}(P) = P' \Rightarrow (P \Downarrow_{source} C \iff P' \Downarrow_{cible} C)}$$

Cette équivalence assure que les programmes compilés se comportent exactement comme dicté par leur code source. Notons que cette équivalence est souvent trop restrictive, en particulier dans le cas où le langage source est non-déterministe. D'autres variations de l'énoncé de correction d'un compilateur sont discutées dans les travaux de Xavier Leroy [10]. Le logiciel CompCert suit cette approche et propose d'implémenter un compilateur C dans l'assistant à la démonstration Coq. Cela permet de réaliser la preuve formelle de la correction du compilateur.

2.2 Modéliser la sémantique des nombres flottants

La définition d'une sémantique pour le langage C (ou tout autre langage proposant un support les nombres flottants) doit notamment modéliser l'évaluation des expressions manipulant des valeurs numériques. On donne pour cela un modèle mathématique de l'arithmétique flottante composé d'un ensemble support $\mathbb{F}$ représentant les nombres flottants ainsi qu'un ensemble de fonctions $\oplus, \ominus, \otimes$ etc pour interpréter les opérateurs. Deux questions se posent alors. D'une part quelle modélisation choisir parmi les différentes arithmétiques flottantes existantes ? D'autre part, une fois le modèle choisi, comment formaliser ce modèle dans le langage d'un assistant à la démonstration comme Coq ? Plusieurs réponses à ces problèmes existent dans la littérature [6, 7, 5]. Nous décrivons brièvement ici celles déployées dans CompCert.

Axiomatiser les flottants Une première solution pour formaliser les nombres flottants et de les axiomatiser : on postule leur existence ainsi que celle des opérations arithmétiques utiles. On peut également axiomatiser quelques propriétés nécessaires pour prouver la correction du compilateur. Cette approche a été initialement utilisée dans les premières versions du compilateur et pose deux principaux problèmes. Tout d'abord, elle nécessite de s'entendre sur les propriétés que l'on peut raisonnablement admettre. Ces propriétés peuvent varier d'une arithmétique à l'autre. Par ailleurs, en choisissant d'axiomatiser les nombres flottants, on s'interdit la possibilité d'effectuer des calculs lors de la compilation pour le besoin de certaines optimisations par exemple. Une solution pour pouvoir tout de même calculer lors de la compilation est d'utiliser les flottants du processeur de la machine qui exécute le compilateur. Toutefois, cette démarche fait l'hypothèse que le processeur employé traite les flottants de manière fidèle à l'axiomatisation choisie. Cette hypothèse n'est pas toujours réaliste et l'article [6] relève déjà cette limitation.

Imposer le standard IEEE-754 Une autre méthode pour formaliser le comportement des nombres flottants est d'imposer l'usage du standard IEEE-754. Ce choix est compatible avec la norme ISO C et permet de fixer une implémentation logicielle et formellement prouvée du standard IEEE-754. Cette approche est celle actuellement mise en œuvre dans le compilateur CompCert [5]. Les flottants sont représentés grâce à la bibliothèque Flocq [6] qui propose une formalisation du standard IEEE-754 en Coq ainsi qu'une implémentation prouvée des différentes opérations flottantes. Si cette solution résout le problème du calcul, elle ne permet cependant pas de décrire la sémantique des cibles de compilation qui ne supportent pas le standard IEEE-754.

3 Le défis de l'extension de CompCert à des cibles non conformes IEEE-754

L'approche par axiomatisation ainsi que celle consistant à fixer l'emploi du standard IEEE-754 limitent les possibilités. La première offre une certaine souplesse mais rend impossible le calcul. La deuxième permet de faire des calculs mais restreint la compilation à des cibles compatibles avec le standard. Nous explorons une troisième approche dans laquelle plusieurs standards sont modélisés. Le modèle de calcul flottant utilisé par CompCert est alors déterminé au moment du choix de la cible de compilation. Cette approche révèle de nombreux défis d'ingénierie logicielle que nous présentons ici.

Contrôler les conséquences sur la chaîne de compilation L'architecture du compilateur CompCert comprend une partie commune à toutes les cibles de compilation (analyse syntaxique, typage, optimisations de haut niveau, etc) ainsi que des parties spécifiques à chaque cible de compilation (jeux d'instructions, génération de code, optimisations de bas niveau, etc). Ces deux parties dépendent d'une seule et même modélisation des nombres flottants qui est utilisée aussi bien pour décrire la sémantique du langage C que celle de toutes les cibles de compilation. En rendant interchangeable le modèle de calcul flottant, on risque donc d'invalider toute la chaîne de compilation et sa preuve de correction. En particulier, il faut identifier quels sont les théorèmes d'arithmétique flottante qui peuvent être utilisés dans la partie commune du code si le modèle de calcul flottant est fixé par la cible de compilation. Un travail d'isolation des propriétés communes et spécifiques des différents modèles de calculs numériques est donc nécessaire. En assurant que seule des propriétés universelles sont utilisées dans la partie commune du compilateur, on garanti la possibilité de changer le modèle de calcul sans invalider les preuves déjà effectuées.

Valider les spécifications Pour chaque architecture présentant une unité de calcul flottante non conforme, il est nécessaire de modéliser son arithmétique associée en Coq. En multipliant le nombre de modèles différents, nous prenons davantage de risque d'introduire des erreurs dans leurs formalisations. De plus, les documents normatifs préfèrent souvent la prose aux équations et l'écart de formalisme entre le langage naturel et une définition Coq rend difficile la relecture et la validation *a posteriori* des spécifications. Pour illustrer cette difficulté, nous prenons l'exemple de l'addition flottante et sa spécification dans le standard IEEE-754 (voir figure 3).

Factoriser pour ne pas re-coder et re-prouver La tâche de formalisation d'un modèle de calcul flottant dans un assistant à la démonstration peut se décomposer en trois temps. Un première étape est de modéliser les nombres flottants en choisissant une structure de données adaptée. On peut dans un second temps implémenter les opérations arithmétiques sous forme de

> *Each of the computational operations that return a numeric result specified by this standard shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format (see 4 and 7). Clause 6 augments the following specifications to cover* ± 0, $\pm\infty$, *and NaN.*

FIGURE 3 – Spécification des opérations flottantes dans le standard IEEE

fonctions Coq opérant sur la représentation préalablement choisie. Enfin, on développe la preuve formelle des propriétés arithmétiques vérifiées par les opérateurs. Ces propriétés seront utilisées pour justifier la correction du compilateur. Cette suite d'étapes est très consommatrice de temps, en particulier le développement des preuves. Nous constatons toutefois que les différents modèles de calcul flottants étudiés partage un socle commun avec le standard IEEE-754, en particulier en matière de choix de représentation des nombres flottants et de vocabulaire employé dans les documents normatifs. Il est donc possible (voire nécessaire) de factoriser les étapes de modélisation, par exemple en utilisant les structures de données et les définitions fournies par la bibliothèque Flocq. En revanche, des différences notables subsistent dans le comportement des opérations et donc des propriétés arithmétiques. Les preuves doivent donc être traitées au cas par cas. Notons toutefois qu'utiliser le même vocabulaire (le même catalogue de définitions Coq) pour exprimer tous les modèles de calcul rend possible le partage de lemmes intermédiaires entre les différentes preuves.

4 Les solutions explorées

Au regard des points mentionnés dans la section précédente, nous proposons différentes approches pour intégrer dans CompCert de nouveaux types de flottants.

4.1 Des flottants paramétrables ?

Fonctoriser le module d'arithmétique flottante Une première solution explorée pour modéliser de nouveaux nombres flottants est de rendre paramétrable l'implémentation actuelle reposant sur la bibliothèque Flocq. Cette solution permet par exemple de rendre modifiable le mode d'arrondi utilisé (fixé au plus proche pair dans CompCert). La bibliothèque Flocq étant générique sur les modes d'arrondi, les changements impliqués en matière d'implémentation sont presque immédiats : il suffit d'étendre le module `Floats` modélisant l'arithmétique flottante en un foncteur `Floats(M : MODE)` dépendant d'un mode d'arrondi fixé par la cible de compilation. Notons toutefois qu'en rendant générique le mode d'arrondi utilisé par les opérations, il faut également rendre générique les théorèmes d'arithmétiques associés. Nous avons observé que les théorèmes utilisés dans CompCert sont indépendants du mode d'arrondi utilisé. La plupart des preuves peuvent donc conservées. Certaines preuves doivent être adaptées mais les énnoncés restent valides.

Limitations Cette approche permet de modéliser des subtiles nuances compatibles avec le standard IEEE-754 comme les différents modes d'arrondi. Toutefois, pour la modélisation de

flottants non conformes, cette approche n'est pas suffisante. En effet, si les écarts avec le standard sont trop importants, il est difficile d'identifier les paramètres à faire varier et de controler leur impact sur la validité des théorèmes.

4.2 Spécification formelle de flottants non conformes

Du pseudo-code pour réduire les écarts de formalisme Comme mentionné dans la section 3, un des principaux obstacles à la traduction de documents normatifs en code Coq est l'écart entre niveaux de vocabulaire utilisés. Il est donc utile de se doter d'outils facilitant la traduction des documentations à dispositions (standards, manuels des processeurs, etc) en code Coq. On peut par exemple passer par l'intermédiaire de pseudo-code. Cela permet d'éviter les transitions abruptes du langage naturel au langage formel tout en faisant apparaitre explicitement les objets mathématiques nécessaires à l'expression des spécifications en Coq. Nous prenons pour example la spécification de l'addition flottante du processeur PowerPC e200z4 (voir figure 4).

The low element of RA is added to the low element of RB and the result is stored in the low element of RD. If RA is NaN or infinity, the result is either pmax (sa = 0), or nmax (sa = 1). If RB is NaN or infinity, the result is either pmax (sb = 0), or nmax (sb = 1). If an overflow occurs, pmax or nmax (as appropriate) is stored in RD. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or −0 (for rounding mode RM) is stored in RD.

```
efsadd (a, b) {
  if (isnan(a) || isinf(a)) {
    max-val(a)
  } else if (isnan(b) || isinf(b)) {
    max-val(b)
  } else if (overflow(a + b)) {
    max-val(a + b)
  } else if (underflow-after-rounding(a + b)) {
      if (rounding = nearest || rounding = zero || rounding = +INF)
      { +0.0 } else { -0.0 }
  } else {
      round(a + b)
  }
}
```

FIGURE 4 – Spécification de l'addition flottante dans le manuel du processeur e200z4 et sa traduction sous forme de pseudo-code

Conversion du pseudo-code en Coq Une fois le pseudo-code écrit et les définitions mathématiques utiles identifiées, nous pouvons le traduire manuellement en Coq. Cette étape consiste essentiellement à choisir les types appropriés pour représenter les différents objets. Nous identifions également une collection de définitions primitives systématiquement utilisées dans les

documents normatifs pour l'arithmétique flottante (par exemple `round`, `overflow`, `underflow` apparaissant en rouge dans la figure 4). Nous implémentons ces primitives en Coq à l'aide de Flocq ce qui permet de ré-exploiter les types et les théorèmes déjà établies dans la bibliothèque.

Validation Passer par l'intermédiaire du pseudo-code permet de répartir l'effort nécessaire à la validation des spécifications. Une première relecture du pseudo-code est faite pour s'assurer de son adéquation par rapport au standard de référence. Notons que cette relecture ne nécessite aucune connaissance du langage Coq. Une deuxième relecture permet ensuite de vérifier la correction du code Coq vis à vis du pseudo-code. L'usage de pseudo-code comme interface rend la relecture plus efficace et plus fiable en séparant explicitement deux champs de compétences : la connaissance du standard considéré et celle du langage Coq.

Limitations La traduction de standards en Coq en passant par l'intérmedaire de pseudo-code facilite la tâche de validation des spécifications. Toutefois, la traduction doit être faite manuellement et les specifications Coq obtenues ne sont pas nécessairement exécutables. En particulier, elles utilisent des opérations sur les nombres réels qui ne peuvent pas être calculées en Coq. En revanche, ces spécifications formelles peuvent servir de référence pour l'implémentation et la vérification d'opérations exécutables.

5 Remerciements

Merci à François Bobot pour m'avoir fait découvrir (et sombrer dans) l'univers des nombres flottants. Merci également à Bernhard Schommer et Christoph Mallon pour les discussions interminables que nous avons eu sur l'interprétation du standard IEEE et de la norme ISO C. Enfin, merci au commité des JFLA pour les commentaires très constructifs qui ont guidés la rédaction de la version finale de cet article.

6 Conclusion

Le compilateur CompCert propose un support pour les flottants IEEE-754 et restreint la compilation à des cibles conformes au standard. Nous avons étudié la possibilité d'intégrer dans CompCert un support pour d'autres arithmétiques flottantes et concluons cette étude par deux constats. D'une part, une telle extension nécessite un travail initial d'isolation de certaines composantes logicielles. Cela permettrait de rendre interchangeable le modèle de calcul flottant tout en limitant les conséquences sur la chaîne de compilation et en particulier sa preuve de correction. D'autre part, l'écart de formalisme entre les documents normatifs qui standardisent l'arithmétique flottante et le langage logique de Coq rend difficile la formalisation de nouveaux modèles de calculs. Passer par l'intermédiaire du pseudo-code peut faciliter cette tâche. Des questions restent cependant ouvertes, notamment concernant la possibilité d'exécuter les spécifications Coq. Développer un langage de spécification formelle dédié à l'arithmétique flottante et pouvant être compilé vers des spécifications Coq executables pourrait être une piste de recherche intéressante. Des travaux similaires ont déjà été menés pour la spécification d'architectures matérielles [12, 3].

Références

[1] The coq proof assistant. https://coq.inria.fr.

[2] Iso/iec/ieee international standard - floating-point arithmetic. *ISO/IEC 60559 :2020(E) IEEE Std 754-2019*, pages 1–86, 2020.

[3] Alasdair Armstrong, Thomas Bauereiß, Brian Campbell, Shaked Flur, Kathryn E. Gray, Prashanth Mundkur, Robert M. Norton, Christopher Pulte, Alastair David Reid, Peter Sewell, Ian David Bede Stark, and Mark Wassell. Detailed models of instruction set architectures : From pseudocode to formal semantics. 2018.

[4] Sylvie Boldo. L'arithmétique des ordinateurs et sa formalisation. https://www.college-de-france.fr/agenda/seminaire/semantiques-mecanisees-quand-la-machine-raisonne-sur-ses-langages/arithmetique-des-ordinateurs-et-sa-formalisation, 2019.

[5] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified Compilation of Floating-Point Computations. *Journal of Automated Reasoning*, 54(2) :135–163, February 2015.

[6] Sylvie Boldo and Guillaume Melquiond. Flocq : A Unified Library for Proving Floating-point Algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, July 2011.

[7] John Harrison. Floating-point verification using theorem proving. In Marco Bernardo and Alessandro Cimatti, editors, *Formal Methods for Hardware Verification*, pages 211–242, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[8] ISO. *ISO/IEC 9899 :2011 Information technology — Programming languages — C*. International Organization for Standardization, 2011.

[9] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the gap – the formally verified optimizing compiler CompCert. In *SSS'17 : Developments in System Safety Engineering : Proceedings of the Twenty-fifth Safety-critical Systems Symposium*, pages 163–180. CreateSpace, 2017.

[10] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, July 2009.

[11] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert – a formally verified optimizing compiler. In *ERTS 2016 : Embedded Real Time Software and Systems*. SEE, 2016.

[12] Mark Wassell. Minisail - a kernel language for the isa specification language sail. *Archive of Formal Proofs*, June 2021. https://isa-afp.org/entries/MiniSail.html, Formal proof development.