Amélioration à la Compilation de la Précision de Programmes Numériques

Nasrine Damouche[1], **Matthieu Martel**[1], **Alexandre Chapoutot**[2]

[1] Laboratoire de Mathématiques et de Physiques, LAMPS, Université de Perpignan Via Domitia, France.
e-mail: nasrine.damouche@univ-perp.fr
e-mail: matthieu.martel@univ-perp.fr

[2] U2IS, ENSTA ParisTech, Université de Paris-Saclay, 828 bd des Maréchaux, 91762 Palaiseau cedex France.
e-mail: alexandre.chapoutot@ensta-paristech.fr

Résumé Les calculs en nombres flottants sont intensivement utilisés dans divers domaines, notamment les systèmes embarqués critiques. En général, les résultats de ces calculs sont perturbés par les erreurs d'arrondi. Dans un scenario critique, ces erreurs peuvent être accumulées et propagées, générant ainsi des dommages plus ou moins graves sur le plan humain, matériel, financier, etc. Il est donc souhaitable d'obtenir les résultats les plus précis possible lorsque nous utilisons l'arithmétique flottante. Pour ce faire, nous avons développé un outil qui corrige partiellement ces erreurs d'arrondi, par une transformation automatique et source à source des programmes. Notre transformation repose sur une analyse statique par interprétation abstraite qui fournit des intervalles pour les variables présentées dans les codes sources. Nous transformons non seulement des expressions arithmétiques mais aussi des morceaux de code avec des affectations, des boucles, des conditionnelles, des fonctions, etc. Les résultats obtenus par notre outils sont très prometteurs. Nous avons montré que nous améliorions de manière significative la précision numérique des calculs en minimisant l'erreur par rapport à l'arithmétique exacte des réels. Un autre critère très intéressant est que notre technique permet d'accélérer la vitesse de convergence de méthodes numériques itératives par amélioration de leur précision comme les méthodes de Newton, Jacobi, Gram-Schmidt, etc. Nous avons réussi à réduire le nombre d'itérations nécessaire pour converger de plusieurs dizaines de pourcents. Nous avons aussi étudié l'impact de l'optimisation de la précision sur le format des variables (en simple ou double précision). Pour ce faire, nous avons comparé deux programmes sources écrits en simple et en double précision avec celui transformé en simple précision. Les résultats obtenus montrent que le programme transformé (32 Bits) est très proche du résultat exact de celui de départ (64 Bits). Cela permet à l'utilisateur de dégrader la précision sans perdre beaucoup d'informations. D'un point de vue théorique, nous avons prouvé que les programmes générés n'ont pas forcément la même sémantique que les programmes d'origine, mais que mathématiquement, ils sont équivalents.

Correspondence to: nasrine.damouche@univ-perp.fr

Mots Clés : Précision numérique, Arithmétique des nombres flottants, Transformation de programmes, Analyse statique, Preuve de correction.

1 Introduction

Suite à des progrès rapides et incessants, l'informatique a pris une ampleur prépondérante dans divers domaines d'application comme l'industrie spatiale, l'aéronautique, les équipements médicaux, le nucléaire, etc. Nous avons tendance à croire aveuglément aux différents calculs effectués par les ordinateurs mais un problème majeur se pose, lié à la fiabilité des traitements numériques, car les ordinateurs utilisent des nombres à virgule flottante qui n'ont qu'un nombre fini de chiffres. Autrement dit, l'arithmétique des ordinateurs basée sur les nombres flottants fait qu'une valeur ne peut être représentée exactement en mémoire, ce qui oblige à l'arrondir. En général, cette approximation est acceptable car la perte est tellement faible que les résultats obtenus sont très proches des résultats réels. Cependant dans un scénario critique, ces approximations engendrent des dégâts considérables sur le plan industriel, financier, humain et bien d'autres. La complexité des calculs en virgule flottante dans les systèmes embarqués ne cesse d'augmenter, rendant ainsi le sujet de la précision numérique de plus en plus sensible. Vu le rôle qu'elle joue sur la fiabilité des systèmes embarqués, l'industrie encourage les chercheurs pour valider [4,9,10,13,14,23] et améliorer [15, 22] leurs logiciels afin d'éviter des failles et éventuellement des catastrophes comme l'échec du missile Patriote en 1991 et l'explosion de la fusée Ariane 5 en 1996.

Cet article traite de la transformation automatique de programmes dans le but d'améliorer leur précision numérique [6, 7,8]. De nombreuses techniques ont été proposées pour transformer automatiquemnt des expressions arithmétiques. Dans ses travaux de thèse [15], A. Ioualalen a introduit une nouvelle représentation intermédiaire (IR) permettant de représenter dans une structure polynomiale, un nombre exponentiel d'expressions arithmétiques équivalentes. Cette représentation, nommée APEG [15,16] pour Abstract Program Expression Graph, a réussi à réduire la complexité de la transformation en un temps et une taille polynomiaux. Le but de notre travail est d'aller au delà des expressions arithmétiques, en s'intéressant à transformer automatiquement des bouts de code de taille plus ou moins grande. Notre transformation opère sur des séquences de commandes comprenant des affectations, des conditionnelles, des boucles, des fonctions,

etc., pour améliorer leur précision numérique. Nous avons défini un ensemble de règles de transformation pour les commandes [7]. Appliquées dans un ordre déterministe, ces règles permettent d'obtenir un programme plus précis parmi tout ceux considérés. Les résultats obtenus montrent que la précision numérique des programmes est significativement améliorée (en moyenne de 20%). Actuellement, nous nous intéressons à optimiser une seule variable de référence à partir des intervalles donnés aux valeurs d'entrées de programme et des bornes d'erreurs calculées en utilisant les techniques d'interprétation abstraite [5] pour l'arithmétique des nombres flottants [7,17].

Théoriquement, nous avons défini un ensemble de règles de transformation qui ont été implémentées dans un logiciel, Salsa. Cet outil se comporte comme un compilateur à la seule différence qu'il utilise les résultats d'une analyse statique fournissant des intervalles pour chaque variable à chaque point de contrôle. Notons que le programme généré ne possède pas forcément la même sémantique que celui de départ mais que les programmes sources et transformés sont mathématiquement équivalents pour les entrées (intervalles) considérées. De plus, le programme transformé est plus précis. La correction de notre approche repose sur une preuve mathématique par induction comparant le programme transformé avec celui d'origine.

Cet article est organisé comme suit. Nous détaillons à la section 2 les bases de l'arithmétique flottante et nous donnons par la suite un bref aperçu de la transformation des expressions arithmétiques. La section 3 concerne les différentes règles de transformation qui nous permettent d'obtenir les programmes optimisés automatiquement. Nous donnerons en section 4 le théorème de correction de notre transformation. En dernier lieu, dans la section 5, nous décrivons les différents résultats expérimentaux obtenus avec notre outil. Nous concluons à la section 6 qui résume nos travaux et ouvre sur de nombreuses perspectives.

2 Analyse et transformation des expressions

Dans cette section, nous présentons les méthodes utilisées pour borner et réduire les erreurs d'arrondi sur les expressions arithmétiques [15,16]. Dans un premier temps, nous présentons brièvement la norme IEEE754 et les méthodes d'analyse statique permettant de calculer les erreurs de calculs. Par la suite, nous évoquons la transformation automatique des expressions arithmétiques.

2.1 Analyse statique pour la précision numérique

La norme IEEE754 est le standard scientifique permettant de spécifier l'arithmétique à virgule flottante [1,21]. Les nombres réels ne peuvent être représentés exactement en mémoire sur machine. A cause des erreurs d'arrondi apparaissant lors des calculs, la précision des résultats numériques est

x	s	e	m
+0	0	00000000	00000000000000000000000
-0	1	00000000	00000000000000000000000
+∞	0	11111111	00000000000000000000000
-∞	1	11111111	00000000000000000000000
NaN	0	11111111	00001001110000011000001 (exemple)

FIGURE 1. Valeurs spéciales de la norme IEEE754.

généralement peu intuitive. La représentation d'un nombre x en virgule flottante, en base b, est défini avec :

$$x = s \cdot (x_0.x_1.x_2 \ldots x_{p-1}) \cdot b^e = s \cdot m \cdot b^{e-p+1} \quad , \qquad (1)$$

avec, $s \in \{0,1\}$ le signe, $m = x_0.x_1.x_2 \ldots x_{p-1}$ la mantisse tel que $0 \leq x_i < b$ et $0 \leq i \leq p-1$, p la précision et enfin l'exposant $e \in [e_{min}, e_{max}]$.

En donnant des valeurs spécifiques pour p, b, e_{min} et e_{max}, le standard IEEE754 définit plusieurs formats pour les nombres flottants. En fonction de la mantisse et de l'exposent, la norme IEEE754 dispose de quatre valeurs spéciales comme le montre la Figure 1. Les NAN (Not a Number) correspondent aux exceptions ou aux résultats d'une opération invalide comme $0 \div 0$, $\sqrt{-1}$ ou $0 \times \pm\infty$.

Par exemple, il est clair que le nombre $1/3$ contient une infinité de chiffres après la virgule en bases 10 et 2, ce qui fait qu'on peut pas le représenter avec une suite finie de chiffres sur ordinateur. Même si l'on prend deux nombres exacts qui sont représentables sur machine, le résultat d'une opération n'est généralement pas représentable. Ceci montre la nécessité d'arrondir. Le standard IEEE754 décrit quatre modes d'arrondi pour un nombre x à virgule flottante :

- L'arrondi vers $+\infty$, renvoyant le plus petit nombre machine supérieur ou égal au résultat exact x. On le note $\uparrow_{+\infty}(x)$.
- L'arrondi vers $-\infty$, renvoyant le plus grand nombre machine inférieur ou égal au résultat exact x. On le note $\uparrow_{-\infty}(x)$
- L'arrondi vers 0, renvoyant $\uparrow_{+\infty}(x)$ si x est négatif ou $\uparrow_{-\infty}(x)$ si x est un nombre positif. On le note $\uparrow_0(x)$.
- L'arrondi au plus près, renvoyant le nombre machine le plus proche du résultat exact x. On le note $\uparrow_\sim(x)$.

La sémantique des opérations élémentaires comme défini par le standard IEEE754 pour les quatre modes d'arrondi $r \in \{-\infty, +\infty, 0, \sim\}$ cités précédemment pour $\uparrow_r : \mathbb{R} \to \mathbb{F}$, est donnée par :

$$x \circledast_r y = \uparrow_r (x * y) \quad , \qquad (2)$$

avec, $\circledast_r \in \{+, -, \times, \div\}$ une des quatre opérations élémentaires utilisées pour le calcul des nombres flottants en utilisant le mode d'arrondi r et $* \in \{+, -, \times, \div\}$ l'opération exacte (opérations sur les réels). Clairement, les résultats des calculs à base des nombres flottants ne sont pas exacts et ceci est dû aux erreurs d'arrondi. Par ailleurs, on utilise la fonction $\downarrow_r : \mathbb{R} \to \mathbb{R}$ permettant de renvoyer l'erreur d'arrondi du nombre en question. Cette fonction est définie par :

$$\downarrow_r (x) = x - \uparrow_r (x) \ . \quad (3)$$

Il est à noter que nos techniques de transformation présentées dans la Section 3 ne dépendent pas d'un mode d'arrondi précis. Pour simplifier notre analyse, on suppose qu'on utilise le mode d'arrondi au plus près dans le reste de cet article, ce qui revient à écrire $\uparrow$ et $\downarrow$ au lieu de $\uparrow_r$ et $\downarrow_r$.

Pour calculer les erreurs se glissant durant l'évaluation des expressions arithmétiques, nous définissons des valeurs non standard faites d'une paire $(x,\mu) \in \mathbb{F} \times \mathbb{R} = \mathbb{E}$, où la valeur x représente un nombre flottant et μ l'erreur exacte liée à x. Plus précisément, μ est la différence exacte entre la valeur réelle et flottante de x comme définit par l'équation (3). A titre d'exemple, prenons le nombre réel $1/3$ qui sera représenté par la valeur suivante :

$$v = (\uparrow_\sim (1/3), \downarrow_\sim (1/3)) = (0.33333333, (1/3 - 0.33333333)).$$

La sémantique concrète des opérations élémentaires dans $\mathbb{E}$ est détaillée dans [18].

La sémantique abstraite associée à $\mathbb{E}$ utilise une paire d'intervalles $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, tel que le premier intervalle $x^\sharp$ contient les nombres flottants du programme, et le deuxième intervalle $\mu^\sharp$ contient les erreurs sur $x^\sharp$ obtenues en soustrayant le nombre flottant de la valeur exacte. Cette valeur abstrait un ensemble de valeurs concrètes $\{(x,\mu) : x \in x^\sharp$ et $\mu \in \mu^\sharp\}$. Revenons maintenant à la sémantique des expressions arithmétiques dont l'ensemble des valeurs abstraites est noté par $\mathbb{E}^\sharp$. Un intervalle $x^\sharp$ est approché avec un intervalle défini par l'équation (4) qu'on note $\uparrow^\sharp (x^\sharp)$.

$$\uparrow^\sharp ([\underline{x}, \overline{x}]) = [\uparrow (\underline{x}), \uparrow (\overline{x})] \ . \quad (4)$$

La fonction d'abstraction $\downarrow^\sharp$, quant à elle, abstrait la fonction concrète $\downarrow$, autrement dit, elle permet de sur-approcher l'ensemble des valeurs exactes d'erreur, $\downarrow (x) = x - \uparrow (x)$ de sorte que chaque erreur associée à l'intervalle $x \in [\underline{x}, \overline{x}]$ est incluse dans $\downarrow^\sharp ([\underline{x}, \overline{x}])$. Pour un mode d'arrondi au plus proche, la fonction d'abstraction est donnée par l'équation (5).

$$\downarrow^\sharp ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{avec} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \ . \quad (5)$$

En pratique, l'ulp(x) qui est une abréviation de *unit in the last place* représente la valeur du dernier chiffre significatif d'un nombre à virgule flottante x. Formellement, la somme de deux nombres à virgule flottante revient à additionner les erreurs générées par l'opérateur avec l'erreur causée par l'arrondi du résultat. Similairement pour la soustraction de deux nombres flottants, on soustrait les erreurs sur les opérateurs et on les ajoute aux erreurs apparues au moment de l'arrondi. Quant à la multiplication de deux nombres à virgule flottante, la nouvelle erreur est obtenue par développement de la formule $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$. Les équations (6) à (8) donnent la sémantique des opérations élémentaires.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp)) \ , \quad (6)$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp (x_1^\sharp - x_2^\sharp), \mu_1^\sharp - \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp - x_2^\sharp)) \ , \quad (7)$$

$$\begin{aligned}(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = (&\uparrow^\sharp (x_1^\sharp \times x_2^\sharp), \\ &x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp)) \ . \quad (8)\end{aligned}$$

Notons qu'il existe d'autres domaines abstraits plus efficaces, à titre d'exemple [4,13,14], et aussi des techniques complémentaires comme [2,3,9,23]. De plus, on peut faire référence à des méthodes qui transforment, synthétisent ou réparent les expressions arithmétiques basées sur des entiers ou sur la virgule fixe [12]. On citera également [4,11,19,20, 23] qui s'intéressent à améliorer les rangs des variables à virgule flottante.

2.2 Les expressions arithmétiques

Nous présentons ici rapidement les travaux de thèse de A. Ioualalen qui portent sur la transformation [6] des expressions arithmétiques en utilisant les APEGs [15,16,24]. Les APEGs, abréviation de *Abstract Program Equivalent Graph*, permettent de représenter en taille polynomiale un nombre exponentiel d'expressions mathématiques équivalentes. Un APEG se compose de classes d'équivalence représentées par des ellipses qui contiennent des opérations, et des boites. Pour former une expression valide, nous construisons l'APEG correspondant à l'expression arithmétique en question d'abord, ensuite, il faut choisir une opération dans chaque classe d'équivalence. Pour éviter le problème lié à l'explosion combinatoire, les APEGs regroupent plusieurs expressions arithmétiques équivalentes à base de commutativité, d'associativité et de distributivité dans des boites. Une boite avec n opérateurs peut représenter un très grand nombre de formules équivalentes allant jusqu'à $1 \times 3 \times 5 ... \times (2n-3)$ expressions. La construction d'un APEG nécessite l'usage de deux algorithmes. Le premier algorithme dit de *propagation* effectue une recherche récursive dans l'APEG afin d'y trouver les opérateurs binaires symétriques qui à leur tour, seront mis dans les boites abstraites. Le deuxième algorithme, d'*expansion*, cherche dans l'APEG une expression plus précise parmi toutes les expressions équivalentes. Enfin, nous recherchons l'expression arithmétique la plus précise selon la sémantique abstraite de la Section 2.1.

La syntaxe des expressions arithmétiques et booléennes est donnée par l'équation (9).

$$\begin{aligned}\text{Expr} \ni e &::= id \mid cst \mid e+e \mid e-e \mid e \times e \mid e \div e \\ \text{BExpr} \ni b &::= \texttt{true} \mid \texttt{false} \mid b \wedge b \mid b \vee b \mid \neg b \mid \\ &\quad e = e \mid e < e \mid e > e\end{aligned} \quad (9)$$

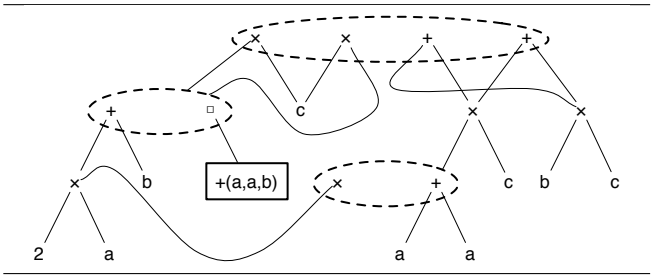

FIGURE 2. *APEG for the expression* $e = \big((a+a)+c\big) \times c$.

Exemple Afin de bien éclairer cette notion, nous donnons ci-dessous quelques expressions arithmétiques correspondant à l'APEG de l'expression $e = ((a+a)+b) \times c)$ de la Figure 2. Sur cette dernière, les ellipses en pointillé correspondent aux classes d'équivalences qui contiennent des APEGs et les rectangles sont les boites constituées d'une opération et n opérandes, autrement dit, c'est les différentes façons de combiner ces opérateurs avec les opérandes en question.

$$\mathscr{A}(p) = \left\{ \begin{array}{c} \big((a+a)+b\big) \times c,\ \big((a+b)+a\big) \times c, \\ \big((b+a)+a\big) \times c,\ \big((2\times a)+b\big) \times c, \\ c \times \big((a+a)+b\big),\ c \times \big((a+b)+a\big), \\ c \times \big((b+a)+a\big),\ c \times \big((2\times a)+b\big), \\ (a+a) \times c + b \times c,\ (2\times a) \times c + b \times c, \\ b \times c + (a+a) \times c,\ b \times c + (2\times a) \times c \end{array} \right\} . \quad (10)$$

■

3 Transformation des commandes

Afin d'améliorer au mieux la précision numérique des calculs, nous transformons automatiquement des programmes utilisant l'arithmétique des nombres à virgule flottante en nous appuyant sur des techniques d'interprétation abstraite. Cette transformation concerne les expressions arithmétique comme l'addition, la multiplication, les fonctions trigonométriques, etc., mais aussi les morceaux de code tel que les affectations, conditionnelles, boucles et fonctions. La syntaxe correspondant à nos commandes est la suivante :

$$\begin{aligned} \mathsf{Com} \quad \ni \quad c \ ::= \ & id = e \mid c_1; c_2 \mid \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\ & \mid \mathsf{while}_\Phi\ e\ \mathsf{do}\ c \mid \mathtt{nop}\ . \end{aligned} \quad (11)$$

Pour réaliser notre transformation, nous avons défini un ensemble de règles permettant de réécrire les programmes en des programmes plus précis. Ces règles de transformation ont été implémentées dans un outil appelé Salsa qui prend en entrée un programme initialement écrit dans un langage impératif, et retourne en sortie un programme écrit dans le même langage et numériquement plus précis. Les programmes sont écrits sous forme SSA pour *Static Single Assignment*, chaque variable est écrite uniquement une seule fois dans le code source, ce qui évite les conflits causés par la lecture et l'écriture d'une même variables. Les différentes règles de transformation, données à la figure 3, sont utilisées dans un ordre déterministe c'est-à-dire qu'elles sont appliquées l'une après l'autre. Notre transformation est répétée jusqu'à ce que le programme final ne change plus. Dans notre cas, on optimise une seule variable à la fois nommée variable de référence et notée ϑ. Un programme transformé, p_t, est plus précis que le programme original, p_o, si et seulement si :

- La variable de référence ϑ correspond mathématiquement à la même expression mathématique dans les deux programmes,
- Cette variable de référence est plus précise dans p_t que dans p_o.

Nous détaillons maintenant l'ensemble des règles de transformation intra-procédurale qui nous permettent de réécrire les différentes commandes. La transformation de nos programmes utilise un environnement formel δ qui relie des identificateurs à des expressions formelles, $\delta : \mathscr{V} \rightarrow Expr$. Tout d'abord, nous avons deux règles pour l'affectation dont la forme est $c \equiv id = e$. La règle $(A1)$ traduit une heuristique permettant de supprimer une affectation du programme source et de la sauvegarder dans la mémoire δ si les conditions suivantes sont bien vérifiées : i) les variables de l'expression e n'appartiennent pas à l'environnement δ, ii) la variable de référence ϑ que nous souhaitons optimiser n'est pas dans la liste noire β, iii) la variable v que l'on souhaite supprimer et mettre dans δ ne correspond pas à la variable de référence ϑ. La seconde règle $(A2)$ permet de substituer les variables déjà mémorisées dans δ dans l'expression e afin d'obtenir une expression plus grosse que nous allons, par la suite, reparenthéser pour en trouver une forme qui est numériquement plus précise. Pour ce qui concerne les séquences de commandes $c_1; c_2$, nous disposons de plusieurs règles de transformation selon la valeur des deux membres c_1 et c_2. Si un des deux membres est égal à `nop`, nous transformons uniquement l'autre membre (règles $(S1)$ et $(S2)$). Sinon, nous réécrivons les deux membres de la séquence (règle $(S3)$). Le troisième type de transformation de commandes concerne les conditionnelles. Les trois premières règles $(C1)$ à $(C3)$ consistent à évaluer la condition e tout en ayant connaissance statique de sa valeur. Dans le cas où la condition est `vraie` nous transformons la partie `if` et quand elle vaut `faux`, nous transformons la branche `then`. Par ailleurs, si on ne connais pas statiquement la valeur de la condition, nous transformons les deux branches de la conditionnelle. Une dernière règle stipule que les variables qui ont été supprimées alors qu'il ne fallait pas les enlever du programme doivent être ré-injecter dans le corps de la conditionnelle car sinon on rencontre des ennuis à l'exécution car des variables sont non initialisées $(C4)$. Par conséquent, nous mettons ces variables dans la liste noire β pour ne pas les supprimer dans une future utilisation. Quant à la boucle $\mathsf{while}_\Phi\ e\ \mathsf{do}\ c$, nous avons défini deux règles. Une première règle $(W1)$ réécrit le corps de la boucle tandis que l'autre règle intervient lorsque nous rencontrons des variables non définies dans le programme car elles ont été supprimées et sauvegardées dans δ. Comme pour les conditionnelles à ce stade là, nous ré-insérons les variables en question dans le corps de la boucle et nous les rajoutons à la liste noire.

$$\frac{\delta' = \delta[id \mapsto e] \quad id \notin \beta}{\langle id = e, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \mathtt{nop}, \delta', C, \beta\rangle} \quad (A1) \qquad \frac{e' = \delta(e) \quad \sigma^\sharp = [\![C[c]]\!]^\sharp \iota^\sharp \quad \langle e', \sigma^\sharp\rangle \rightsquigarrow^* e''}{\langle id = e, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle id = e'', \delta, C, \beta\rangle} \quad (A2)$$

$$\frac{}{\langle \mathtt{nop}\ ;\ c\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c, \delta, C, \beta\rangle} \quad (S1) \qquad \frac{}{\langle c\ ;\ \mathtt{nop}\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c, \delta, C, \beta\rangle} \quad (S2)$$

$$\frac{C' = C[[];c_2] \quad \langle c_1, \delta, C', \beta\rangle \Rightarrow^*_\vartheta \langle c_1', \delta', C', \beta'\rangle \quad C'' = C[c_1';[]] \quad \langle c_2, \delta', C'', \beta'\rangle \Rightarrow_\vartheta \langle c_2', \delta'', C'', \beta''\rangle}{\langle c_1\ ;\ c_2\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_1'\ ;\ c_2', \delta'', C, \beta''\rangle} \quad (S3)$$

$$\frac{\sigma^\sharp = [\![C[\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]]\!]^\sharp \iota^\sharp \quad [\![e]\!]^\sharp \sigma^\sharp = \mathtt{true} \quad \langle c_1, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_1', \delta', C, \beta\rangle}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \Psi(\Phi, c_1'), \Psi(\Phi, \delta'), C, \beta\rangle} \quad (C1)$$

$$\frac{\sigma^\sharp = [\![C[\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]]\!]^\sharp \iota^\sharp \quad [\![e]\!]^\sharp \sigma^\sharp = \mathtt{false} \quad \langle c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_2', \delta', C, \beta\rangle}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \Psi(\Phi, c_2'), \Psi(\Phi, \delta'), C, \beta\rangle} \quad (C2)$$

$$\frac{Var(e) \cap Dom(\delta) = \emptyset \quad \beta' = \beta \cup Assigned(c_1) \cup Assigned(c_2) \quad \langle c_1, \delta, C, \beta'\rangle \Rightarrow_\vartheta \langle c_1', \delta_1, C, \beta_1\rangle \quad \langle c_2, \delta, C, \beta'\rangle \Rightarrow_\vartheta \langle c_2', \delta_2, C, \beta_2\rangle \quad \delta' = \delta_1 \cup \delta_2}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1'\ \mathsf{else}\ c_2', \delta', C, \beta'\rangle} \quad (C3)$$

$$\frac{V = Var(e) \quad c' = AddDefs(V, \delta) \quad \delta' = \delta_{|Dom(\delta)\setminus V} \quad \langle c'; \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta', C, \beta \cup V\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle} \quad (C4)$$

$$\frac{Var(e) \cap Dom(\delta) = \emptyset \quad C' = C[\mathsf{while}_\Phi\ e\ \mathsf{do}\ []] \quad \langle c, \delta, C', \beta\rangle \Rightarrow_\vartheta \langle c', \delta', C', \beta'\rangle}{\langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c', \delta', C, \beta'\rangle} \quad (W1)$$

$$\frac{V = Var(e) \cup Var(\Phi) \quad c' = AddDefs(V, \delta) \quad \delta' = \delta_{|Dom(\delta)\setminus V} \quad \langle c'; \mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta', C, \beta \cup V\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle}{\langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle} \quad (W2)$$

FIGURE 3. *Règles de transformation pour améliorer la précision de programmes.*

4 Preuve de correction

Pour vérifier la correction de notre transformation, nous introduisons un théorème qui compare deux programmes et montre que le programme le plus précis peut être utilisé à la place de celui moins précis. Notre preuve est basée sur une sémantique opérationnelle classique pour les expressions arithmétiques et les commandes. La comparaison entre les deux programmes nécessite de spécifier une variable de référence ϑ définie par l'utilisateur. Plus précisément, un programme transformé p_t est plus précis qu'un programme d'origine p_o si et seulement si les deux conditions suivantes sont vérifiées :

— La variable de référence ϑ correspond à même expression mathématique dans les deux programmes,
— La variable de référence ϑ est plus précise dans le programme transformé p_t que dans celui d'origine p_o.

Nous utilisons la relation d'ordre $\sqsubseteq \subseteq \mathbb{E} \times \mathbb{E}$ disant qu'une valeur (x, μ) est plus précise qu'une valeur (x', μ') si elles correspondent à la même valeur réelle et si l'erreur μ est plus petite que μ'.

Définition 1 (Comparaison des expressions). Nous considérons $v_1 = (x_1, \mu_1) \in \mathbb{E}$ et $v_2 = (x_2, \mu_2) \in \mathbb{E}$. Nous disons que v_1 est plus précise que v_2, noté $v_1 \sqsubseteq v_2$, si et seulement si $x_1 + \mu_1 = x_2 + \mu_2$ et $|\mu_1| \leq |\mu_2|$. ■

Définition 2 (Comparaison des commandes). Soit c_o et c_t deux commandes, δ_o et δ_t deux environnements formels et ϑ la variable de référence. Nous disons que

$$\langle c_t, \delta_t, C, \beta\rangle \prec_\vartheta \langle c_o, \delta_o, C, \beta\rangle \tag{12}$$

si et seulement si pour chaque $\sigma \in \mathrm{Mem}$,

$$\exists \sigma_o \in \mathrm{Mem}, \langle c_o, \sigma\rangle \rightarrow^* \langle \mathtt{nop}, \sigma_o\rangle \ ,$$
$$\exists \sigma_t \in \mathrm{Mem}, \langle c_t, \sigma\rangle \rightarrow^* \langle \mathtt{nop}, \sigma_t\rangle \ ,$$

— soit $\sigma_t(\vartheta) \sqsubseteq \sigma_o(\vartheta)$,
— soit pour chaque $\mathrm{id} \in \mathrm{Dom}(\sigma_o) \setminus \mathrm{Dom}(\sigma_t)$, $\delta_t(\mathrm{id}) = e$ et $\langle e, \sigma\rangle \rightarrow^*_e \sigma_o(\mathrm{id})$.

■

La deuxième définition spécifie que :

— Les deux commandes c_o et c_t calculent la même valeur de référence ϑ dans les deux environnements δ_o et δ_t dans une arithmétique exacte,
— La commande transformée est plus précise,
— Si après exécution du programme, une variable est indéfinie dans l'environnement concret δ_t, alors l'expression formelle correspondante est déjà enregistrée dans δ_t.

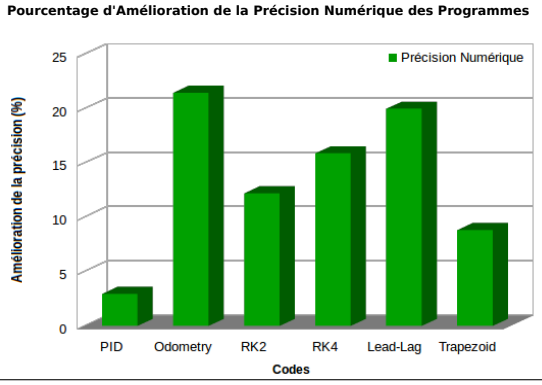

FIGURE 4. *Pourcentage de l'amélioration de la précision numérique des programmes.*

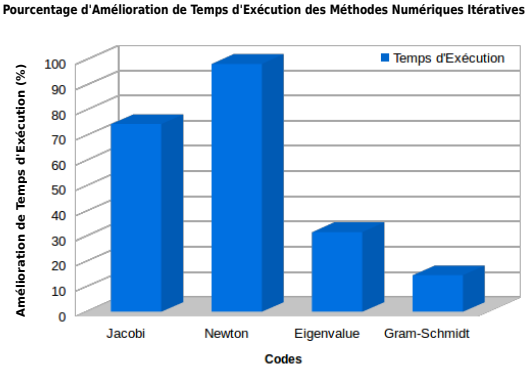

FIGURE 5. *Pourcentage d'amélioration de temps d'exécution des méthodes numériques itératives.*

Théorème 1 (Correction de la transformation des commandes). *Soit c_o le code d'origine, c_t le code transformé, δ_o l'environnement initial, δ_t l'environnement final de la transformation, nous écrivons ainsi*

$$\big(\langle c_o,\delta_o,C,\beta\rangle \Rightarrow_\vartheta \langle c_t,\delta_t,C,\beta\rangle\big) \Longrightarrow \big(\langle c_o,\delta_o,C,\beta\rangle \prec_\vartheta \langle c_t,\delta_t,C,\beta\rangle\big) \quad . \quad (13)$$

■

5 Résultats expérimentaux

Nous avons développé un prototype qui transforme automatiquement des bouts de codes contenant des affectations, des conditionnelles, des boucles, des fonctions, etc., écrit dans un langage impératif. De nombreux tests ont été menés afin d'évaluer l'efficacité de notre outil pour les entrées (intervalles) considérées. Les résultats obtenus sont concluants. Les exemples de la Figure 4 illustrent le gain, en pourcentage, de précision numérique sur une suite de programmes provenant des systèmes embarqués et des méthodes d'analyse numérique [7]. Les programmes transformés sont plus précis que ceux de départ, c'est-à-dire, que la nouvelle erreur après transformation est plus petite que l'erreur de départ. Prenant à titre d'exemple le programme qui calcule la position d'un robot à deux roues (`odometry`), la précision a été amélioré de 21%. Si nous observons aussi le système masse-ressort qui consiste à changer la position initiale y d'une masse vers une position désirée y_d, nous remarquons que la précision numérique pour ce programme est améliorée de 19%. Nous nous sommes aussi intéressés à étudier l'impact de l'optimisation de la précision numérique des calculs sur la vitesse de convergence des méthodes numériques itératives. Pour ce faire, nous avons pris une série d'exemples de méthodes telles que les méthodes de Jacobi, Newton, Gram-Schmidt ainsi qu'une méthode de calcul des valeurs propres d'une matrice. En utilisant notre outil, nous avons réussi à réduire le nombre d'itérations nécessaire à la convergence de toutes ces méthodes en améliorant la précision de leurs calculs. Les programmes ont été écrits dans le langage de programmation `C` et compilés avec `GCC` tout en désactivant toutes les optimisations du compilateur (`-o0`). La Figure 5 donne un aperçu sur les expérimentations faites sur les méthodes numériques précédemment citées. Nous accélérons la vitesse de convergence pour ces méthodes avec une moyenne de 20%. Dans le futur, nous souhaitons renouveler cette étude sur des codes réels issus de la physique.

Une autre évaluation de notre méthode de transformation concerne l'impact de l'optimisation de la précision numérique des programmes sur le format des variables utilisées, (simple ou double précision). Cette optimisation offre à l'utilisateur la possibilité de travailler sur des programmes en simple précision tout en étant sûr que les résultats obtenus seront proches des résultats obtenus avec le programme initial exécuté en double précision. Pour cela, nous avons choisi de calculer l'intégrale du polynôme $(x-2)^7$ avec la méthode de `Simpson` et nous nous sommes restreints à étudier le comportement autour de la racine, donc sur l'intervalle $[1.9, 2.1]$ où le polynôme s'évalue très mal dans l'arithmétique des nombres flottants. Nous avons comparé trois programmes, le premier code source écrit en simple précision (32 bits), un deuxième code source en double précision et le troisième programme qui est celui transformé en simple précision (32 bits). Sur la Figure 6 sont illustrés les résultats obtenus. Nous voyons que le programme transformé en 32 Bits est très proche de celui de départ en 64 Bits. L'atout principal de cette comparaison est de permettre à l'utilisateur d'utiliser un format plus compact sans perdre beaucoup d'informations, ce qui permet d'économiser de la mémoire, de la bande passante et du temps de calcul.

6 Conclusion

L'objectif principal de notre travail est d'améliorer la précision numérique des calculs par transformation automatique

FIGURE 6. *Résultats de la simulation de la méthode des Simpson avec simple, double précision et le programme optimisé utilisant notre outil.*

de programmes. Nous avons défini un ensemble de règles de transformation intra-procédurale et inter-procédurales qui ont été implémentées dans un outil appelé `Salsa`. Notre outil, basé sur les méthodes d'analyse statique par interprétation abstraite, prend en entrée un programme écrit dans un langage impératif et retourne en sortie un autre programme mathématiquement équivalent mais plus précis. Nous avons ensuite démontré la correction de cette approche en faisant une preuve mathématique qui compare les deux programmes, plus précisément, on compare la précision du code source avec celle du code transformé. Cette preuve par induction est appliquée aux différentes constructions du langage supporté par notre outil. Les résultats expérimentaux présentés dans la Section 5 montrent les différentes applications de notre outil.

Une perspective consiste à étendre notre outil pour traiter les codes massivement parallèles. Dans cette direction, nous nous intéressons à résoudre des problèmes spécifiques de la précision numérique comme l'ordre des opérations de calculs dans les programmes parallèles. Nous nous intéressons aussi à explorer le compromis temps d'exécution, performance de calculs, précision numérique ainsi que vitesse de convergence des méthodes numériques. Un point très ambitieux consiste à étudier l'impact de l'amélioration de la précision numérique sur le temps de convergence d'algorithmes de calculs distribués comme ceux utilisés pour le calcul haute performance. De plus, notre intérêt porte sur les problèmes de reproductibilité des résultats, plus précisément, plusieurs exécutions d'un même programme donne des résultats différents et ce à cause de la variabilité de l'ordre d'exécution des expressions mathématiques.

Références

1. ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.
2. E-T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *Symposium on Principles of Programming Languages, POPL '13, 2013*, pages 549–560. ACM, 2013.
3. F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *Programming Language Design and Implementation, PLDI '12, 2012*, pages 453–462. ACM, 2012.
4. J. Bertrane, P. Cousot, R. Cousot, F. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes*, 36(1) :1–8, 2011.
5. P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.
6. P. Cousout and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Principles of Programming Languages*, pages 178–190. ACM, 2002.
7. N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In Manuel Núñez and Matthias Güdemann, editors, *FMICS'15*, volume 9128 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2015.
8. N. Damouche, M. Martel, and A. Chapoutot. Transformation of a PID controller for numerical accuracy. *Electr. Notes Theor. Comput. Sci.*, 317 :47–54, 2015.
9. E. Darulova and V. Kuncak. Sound compilation of reals. In Suresh Jagannathan and Peter Sewell, editors, *POPL'14*, pages 235–248. ACM, 2014.
10. D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and V. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.
11. J. Feret. Static analysis of digital filters. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2004.
12. X. Gao, S. Bayliss, and G-A. Constantinides. SOAP : structural optimization of arithmetic expressions for high-level synthesis. In *Field-Programmable Technology, FPT*, pages 112–119. IEEE, 2013.
13. E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *Static Analysis Symposium, SAS*, volume 7935 of *Lecture Notes in Computer Science*, pages 1–3. Springer, 2013.
14. E. Goubault and S. Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*. Springer, 2011.
15. A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *SAS'12*, volume 7460 of *LNCS*, pages 75–93. Springer, 2012.
16. M. Martel. Accurate evaluation of arithmetic expressions (invited talk). *Electr. Notes Theor. Comput. Sci.*, 287 :3–16, 2012.
17. Matthieu Martel. Propagation of roundoff errors in finite precision computations : A semantics approach. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2002.
18. Matthieu Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1) :7–30, 2006.
19. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004.
20. David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.
21. J-M. Muller, N. Brisebarre, F. De Dinechin, C-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.
22. J.-R. Wilcox P. Panchekha, A. Sanchez-Stern and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI'15*, pages 1–11. ACM, 2015.
23. A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM'15*, volume 9109 of *LNCS*, pages 532–550. Springer, 2015.
24. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation : A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.