Improving the Numerical Accuracy of Floating-Point Programs with Automatic Code Transformation Methods

Nasrine Damouche

▶ To cite this version:

Nasrine Damouche. Improving the Numerical Accuracy of Floating-Point Programs with Automatic Code Transformation Methods. Computer Arithmetic. Université de Perpignan, 2016. English. NNT : 2016PERP0032 . tel-01455727

HAL Id: tel-01455727
https://theses.hal.science/tel-01455727v1

Submitted on 3 Feb 2017

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.

THÈSE
Pour obtenir le grade de
Docteur

Délivré par
UNIVERSITE DE PERPIGNAN VIA DOMITIA

Préparée au sein de l'école doctorale**Énergie et environnement**
ED 305
Et de l'unité de recherche **LAMPS**

Spécialité **Informatique**:

Présentée par

Nasrine DAMOUCHE

TITRE DE LA THESE

Improving the Numerical Accuracy of Floating-Point Programs with Automatic Code Transformation Methods

Soutenue le 12/12/2016 devant le jury composé de

M. **Marc POUZET**, Professeur, Université Pierre et Marie Curie — Rapporteur
M. **Michel RUEHER**, Professeur, Université Nice Sophia Antipolis-CNRS — Rapporteur
M. **Pierre-Loïc GAROCHE**, Ingénieur de recherche – HDR, ONERA Toulouse — Examinateur
Mme. **Laure GONNORD**, Maître de conférences, Université Lyon1 — Examinatrice
Mme. **Yassamine SELADJI**, Maître de conférences, Université de Tlemcen, Algérie — Examinatrice
Mme. **Samira EL YACOUBI**, Professeur, Université de Perpignan Via Domitia — Examinatrice
M. **Matthieu MARTEL**, Professeur, Université de Perpignan, Via Domitia — Directeur de thèse
M. **Alexandre CHAPOUTOT**, Maître-assistant, ENSTA-Paristech, Palaiseau — Co-Directeur de thèse

Le code source est comme une belle femme, plus on le regarde, plus on trouve des défauts.

-Crayon

ACKNOWLEDGEMENTS

Yes! I did it. I finally completed what I started three years ago. Through the followings lines, I would like to thank all the people that supported me over the years and who made this possible. Of course, the following list is not exhaustive, and I will try to keep it short.

∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼

First of all, I would like to thank so much the honorably member of the jury. I am grateful to my reeding committee: Prof. Marc POUZET, Prof. Michel RUEHER, Prof. Pierre-Loïc GAROCHE, Prof. Laure GONNORD, Prof. Yassamine SELADJI and Prof. Samira EL YAKOUBI. Thank you for your useful comments, which contributed to improve the quality of this manuscript.

∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼

Second, a thesis it is not only a subject, deadlines, posters, or trips over the world, it is mostly a relationship with your supervisors. I have been lucky enough to do this thesis with Matthieu and Alexandre. They have always been there in the background assuring as well the tracking of my work to improve correctly and in the best direction. Their different personalities contribute to the success of this thesis. Thank you both for everything.

∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼

I also want to thank the director of LAMPS, Prof. Mircea SOFONEA for having welcomed me into the laboratory. An institute without a good secretariat is nothing. Luckily for me, Sylvia was always available to solve my issues and not only :). I thank also the second secretary Joëlle and all the members of LAMPS laboratory for the reception and the privileged working conditions that were offered to me.

∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼

I am also grateful to all my friends that I have met during these three years, without citing their names. I am really enjoyed our friendship.

∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼

To the best parents in the whole wide world! My mother and father, who are always present for me. The completion of this document is due to their love, their kind and their sens of humor. To my siblings that without them I am lost. Love you honestly my family!

∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼

For all those times you stood by me, for all the truth that you made me see, for all the joy you brought to my life, for all the wrong that you made right, for every dream you made come true, for all the love I found in you, I'll be forever thankful to you, Abdeslam.

∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼

I dedicate this thesis to my little Aksil who has filled our lives with happiness and love. I hope you will be proud of your mom. Hemlaγkun atas atas!

∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼ ∼

Nasrine

CONTENTS

1 AMÉLIORATION DE LA PRÉCISION NUMÉRIQUE DE PROGRAMMES xiii

1.1 Introduction xiv

1.2 Analyse et Transformation des Expressions xv

1.2.1 Analyse Statique pour la Précision Numérique xv

1.2.2 Transformation des Expressions Arithmétiques xix

1.3 Transformation des Commandes xxi

1.4 Preuve de Correction xxiv

1.5 Résultats Expérimentaux xxv

1.5.1 Amélioration de la Précision Numérique xxv

1.5.2 Accélération de la Vitesse de Convergence xxvi

1.5.3 Formats de Variables xxviii

1.6 Conclusion xxviii

2 Introduction 1

2.1 Context 1

2.2 Motivating example 4

2.2.1 PID Controller . . . 4
2.2.2 Experimental Results . . . 9
2.2.3 Conclusion . . . 11
2.3 Contribution . . . 12
2.3.1 Formal Transformations Rules for Programs . . . 12
2.3.2 A Theorem for Correctness . . . 12
2.3.3 Experimental Results . . . 13
2.4 Organization of This Work . . . 15
2.5 Publications . . . 16

3 Background and Related Work **21**
3.1 Introduction . . . 22
3.2 Computer Arithmetic . . . 22
3.2.1 Floating-Point Arithmetic . . . 22
3.2.2 Interval Arithmetic . . . 26
3.3 Languages . . . 27
3.3.1 SSA Form . . . 27
3.3.2 Formal Semantics . . . 28
3.4 Static Analysis . . . 31
3.4.1 Domain Theory . . . 31
3.4.2 Abstract Interpretation . . . 34
3.4.3 Errors Computation . . . 36
3.5 Tools . . . 39
3.6 Transformation of Expressions . . . 41
3.7 Conclusion . . . 44

4 Intraprocedural Transformation **49**
4.1 Introduction . . . 49
4.2 Transformation of Commands . . . 50
4.3 Example of Transformation . . . 58
4.3.1 Odometry. . . . 59
4.4 Proof of Correctness . . . 64
4.5 Conclusion . . . 75

5 Interprocedural Transformation **79**
5.1 Introduction . . . 79
5.2 Transformation of Functions . . . 80
5.2.1 Inlining Functions . . . 81
5.2.2 Specialization of Functions . . . 81
5.2.3 Passing the Parameters of Functions . . . 83
5.3 Choice of the Kind of Function Transformation . . . 85
5.4 Conclusion . . . 85

6 Experiments : Numerical Accuracy **89**
6.1 Introduction . . . 90
6.2 Overview of *SALSA* . . . 90
6.3 Improving the Accuracy of Intraprocedural Programs . . . 92
6.3.1 Odometry . . . 93
6.3.2 PID Controller . . . 93
6.3.3 Lead-Lag System . . . 95
6.3.4 Runge-Kutta Methods . . . 96
6.3.5 The Trapezoidal Rule . . . 99
6.3.6 Rocket Trajectory Simulation . . . 99

6.3.7 Experimental Results . . . 103
6.4 Improving the Accuracy of Interprocedural Programs . . . 104
6.4.1 Odometry . . . 105
6.4.2 Newton-Raphson's Method . . . 105
6.4.3 Runge-Kutta Method . . . 107
6.4.4 Simpson's Method . . . 109
6.4.5 Experimental Results . . . 109
6.5 Optimizing the Data-Type Formats of Variables . . . 110
6.5.1 Numerical Integration Method . . . 112
6.5.2 Experimental Results . . . 112
6.6 Conclusion . . . 117

7 Experiments : Convergence Acceleration 121
7.1 Introduction . . . 121
7.2 Accelerating the Convergence of Iterative Methods . . . 122
7.2.1 Linear Systems of Equations . . . 122
7.2.2 Zero Finding . . . 124
7.2.3 Eigenvalue Computation . . . 127
7.2.4 Iterative Gram-Schmidt Method . . . 129
7.3 Performance Analysis . . . 131
7.4 Conclusion . . . 133

8 Conclusion and Future Work 137
8.1 Conclusion . . . 137
8.2 Perspectives . . . 138

LIST OF FIGURES

1-1 *APEG correspondant à l'expression $e = ((a+a)+c) \times c$.* xxi

1-2 *Règles de transformation pour améliorer la précision de programmes.* xxiii

1-3 *Pourcentage de l'amélioration de la précision numérique de programmes.* xxvi

1-4 *Résultats de la simulation de la méthode de Simpson avec simple, double précision et le programme optimisé utilisant Salsa. Les valeurs des axes x et y correspondent respectivement aux valeurs de n et s de l'Equation 1.14.* xxx

2-1 *Description of a PID Controller.* 5

2-2 *Value of the measure m in the three* `PID` *algorithms.* 9

2-3 *Zoom on the measure m in the three* `PID` *algorithms on the interval [0,80] [0,5.5].* 10

2-4 *Difference between the values of the measure m in $\mathtt{PID}_1$ and $\mathtt{PID}_3$.* 10

2-5 *Comparison between results r and r' and between the corresponding measures m and m'.* 11

3-1 *Binary Representation of Floating-Point Numbers.* 24

3-2 *Small-step operational semantics of programs.* 30

3-3 *Complete lattice presented in the example 3.4.1.* 33

3-4 *APEG for the expression $e = ((a+a)+b) \times c$.* 42

3-5 *Some rules for APEG construction by pattern matching.* . . . 43

4-1 *Transformation rules used to improve the accuracy of programs.* . . . 52

4-2 *Parameters of the two-wheeled robot.* . . . 59

5-1 *Transformation rules used to deal with functions.* . . . 81

6-1 *Architecture of* SALSA. . . . 91

6-2 SALSA *files.* . . . 92

6-3 *Computed trajectories by the original and the transformed odometry programs.* . . 94

6-4 *The lead-lag system.* . . . 95

6-5 *Difference between the original and the transformed trajectories of the rocket.* . . 103

6-6 *Simulation results of the Simpson's method with single, double precision and optimized program using our tool. The values of x and y axes correspond respectively to the value of n and s in Equation 6.7.* . . . 116

7-1 Number of iterations of the Newton-Raphson's Method before and after optimization for initial values ranging from 0 to 3 (30 runs with a step of 0.1). . . . 126

7-2 Difference between numbers of iterations of original and optimized Iterated Power Method (tests done for $\mathtt{d} \in [175, 200]$ with a step of 1). . . . 129

7-3 Iterations number of original and optimized iterative Gram-Schmidt Method for the family $(Q_n)_n$ of vectors, $1 \leq n \leq 10$. . . . 132

LIST OF TABLES

1.1 *Format de base de la norme IEEE754.* xvi
1.2 *Valeurs spéciales de la norme IEEE754 en simple précision.* xvi
1.3 *Temps d'exécution des méthodes numériques itératives.* xxvii
1.4 *Nombre d'opérations flottantes nécessaire aux programmes pour converger.* . . . xxvii

3.1 *Basic `IEEE754` floating-point formats.* 24
3.2 *Special values of the `IEEE754` floating-point formats in simple precision.* 25

6.1 *Values of x before and after transformation of Odometry program at the first iterations.* 93
6.2 *Initial and new errors on the examples programs of Section 6.3.* 104
6.3 *Execution time measurements of programs of Section 6.3.* 104
6.4 *Comparison between the accuracy results of programs before and after optimization.* 110

7.1 Number of iterations of Jacobi's method before and after optimization to compute x_i, $1 \leq i \leq 4$. 124
7.2 Execution time measurements of programs of Section 7.2. 133
7.3 Floating-point operations needed by programs of Section 7.2 to converge. 133

CHAPITRE 1

AMÉLIORATION DE LA PRÉCISION NUMÉRIQUE DE PROGRAMMES

Contents

1.1 Introduction . . . . **xiv**
1.2 Analyse et Transformation des Expressions . . . . **xv**
1.2.1 Analyse Statique pour la Précision Numérique xv
1.2.2 Transformation des Expressions Arithmétiques xix
1.3 Transformation des Commandes . . . . **xxi**
1.4 Preuve de Correction . . . . **xxiv**
1.5 Résultats Expérimentaux . . . . **xxv**
1.5.1 Amélioration de la Précision Numérique xxv
1.5.2 Accélération de la Vitesse de Convergence xxvi
1.5.3 Formats de Variables xxviii
1.6 Conclusion . . . . **xxviii**

1.1 Introduction

Suite à des progrès rapides et incessants, l'informatique a pris une place 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 dans la fiabilité des systèmes embarqués, l'industrie encourage les chercheurs à valider [16, 37, 39, 48, 51, 82] et améliorer [56, 78] leurs logiciels afin d'éviter des failles et éventuellement des catastrophes comme l'échec du missile Patriot en 1991 et l'explosion de la fusée Ariane 5 en 1996.

Ce chapitre traite de la transformation automatique de programmes dans le but d'améliorer leur précision numérique [26, 29, 31]. De nombreuses techniques ont été proposées pour transformer automatiquement des expressions arithmétiques. Dans ses travaux de thèse [56], 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 [56, 67] 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 [29]. Appliquées dans un ordre déterministe, ces règles permettent d'obtenir un programme plus précis parmi tout ceux considérés. Les ré-

sultats obtenus montrent que la précision numérique des codes 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 [23] pour l'arithmétique des nombres flottants [29, 63].

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 comparant le programme transformé avec celui d'origine.

Ce chapitre est organisé comme suit. Nous détaillons à la section 1.2 les bases de l'arithmétique flottante et nous donnons par la suite un bref aperçu de la transformation des expressions arithmétiques. La section 1.3 concerne les différentes règles de transformation qui nous permettent d'obtenir les programmes optimisés automatiquement. Nous donnerons en section 1.4 le théorème de correction de notre transformation. En dernier lieu, dans la section 1.5, nous décrivons les différents résultats expérimentaux obtenus avec notre outil. Nous concluons à la section 1.6 qui résume nos travaux et ouvre sur de nombreuses perspectives.

1.2 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 [56, 67]. 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 décrivons la transformation automatique des expressions arithmétiques.

1.2.1 Analyse Statique pour la Précision Numérique

La norme IEEE754 est le standard scientifique permettant de spécifier l'arithmétique à virgule flottante [10, 76]. 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 généralement peu intuitive. La représentation d'un nombre x en virgule flottante, en base b, est défini avec :

$$x = s \cdot (x_0.x_1.x_2 \ldots x_{p-1}) \cdot b^e = s \cdot m \cdot b^{e-p+1} \quad , \tag{1.1}$$

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}, la norme IEEE754 définit plusieurs formats pour les nombres flottants (voir Table 1.1).

Format	Name	p	e bits	e_{min}	e_{max}
Binary16	Half precision	11	5	−14	+15
Binary32	Single precision	24	8	−126	+127
Binary64	Double precision	53	11	−1122	+1223
Binary128	Quadratic precision	113	15	−16382	+16383

TABLE 1.1 – *Format de base de la norme IEEE754.*

En fonction de la mantisse et de l'exposent, la norme IEEE754 dispose aussi de quatre valeurs spéciales comme le montre la Table 1.2. 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 ne 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 :

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

TABLE 1.2 – *Valeurs spéciales de la norme IEEE754 en simple précision.*

- 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 la norme IEEE754 pour les quatre modes d'arrondi $r \in \{-\infty, +\infty, 0, \sim\}$ cités précédemment pour $\uparrow_r : \mathbb{R} \rightarrow \mathbb{F}$, est donnée par :

$$x \circledast_r y = \uparrow_r (x * y) \quad , \tag{1.2}$$

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 (opération sur les réels). Clairement, les résultats des calculs à base de nombres flottants ne sont pas exacts et ceci est dû aux erreurs d'arrondi. Aussi utilise-t-on la fonction $\downarrow_r : \mathbb{R} \rightarrow \mathbb{R}$ permettant de renvoyer l'erreur d'arrondi du nombre en question. Cette fonction est définie par :

$$\downarrow_r (x) = x - \uparrow_r (x) \quad . \tag{1.3}$$

Il est à noter que les techniques de transformation présentées dans la Section 1.3 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 ce chapitre, ce qui revient à écrire $\uparrow$ et $\downarrow$ au lieu de $\uparrow_r$ et $\downarrow_r$.

Pour calculer les erreurs survenant 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 (1.3). Par exemple, prenons le nombre réel $1/3$ qui est 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 [65].

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 $\{(x,\mu) : x \in x^\sharp \text{ et } \mu \in \mu^\sharp\}$. 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 (1.4) qu'on note $\uparrow^\sharp(x^\sharp)$.

$$\uparrow^\sharp([\underline{x},\overline{x}]) = [\uparrow(\underline{x}), \uparrow(\overline{x})] \ . \tag{1.4}$$

La fonction abstraite $\downarrow^\sharp$ 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 (1.5).

$$\downarrow^\sharp([\underline{x},\overline{x}]) = [-y, y] \quad \text{avec} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \ . \tag{1.5}$$

En pratique, l'`ulp` 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 (1.6) à (1.8) donnent la sémantique des opérations élémentaires.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp(x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp(x_1^\sharp + x_2^\sharp)) \ , \tag{1.6}$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp(x_1^\sharp - x_2^\sharp), \mu_1^\sharp - \mu_2^\sharp + \downarrow^\sharp(x_1^\sharp - x_2^\sharp)) \ , \tag{1.7}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp(x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp(x_1^\sharp \times x_2^\sharp)) \ . \tag{1.8}$$

Notons qu'il existe d'autres domaines abstraits plus efficaces (relationnels), à titre d'exemple [16, 48, 51], et aussi des techniques complémentaires comme [14, 15, 37, 82]. 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 [44]. On citera également [16, 42, 69, 71, 82] qui s'intéressent à améliorer les intervalles de variation des variables à virgule flottante.

1.2.2 Transformation des Expressions Arithmétiques

Nous présentons ici brièvement les travaux de thèse de A. Ioualalen qui portent sur la transformation [26] des expressions arithmétiques en utilisant les APEGs [56, 67, 86]. 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 en pointillés qui contiennent des opérations, et des boites (voir Figure 1-1). 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 1.2.1.

La syntaxe des expressions arithmétiques et booléennes est donnée par l'équation (1.9).

$$\begin{array}{rcll} \text{Expr} \ni & e & ::= & id \mid cst \mid e+e \mid e-e \mid e\times e \mid e\div e \ . \\ \text{BExpr} \ni & b & ::= & \texttt{true} \mid \texttt{false} \mid b\wedge b \mid b\vee b \mid \neg b \mid e=e \mid e<e \mid e>e \ . \end{array} \quad (1.9)$$

Notons que dans l'équation (1.9), *id* est un identificateur et *cst* est une constante.

Les APEGs sont une extension de *Equivalence Program Expressions Graphs* (EPEGs) introduite par R. Tate et al. [85, 86]. Un APEG est définit comme suit :

1. Une constante *cst* ou un identificateur *id* est un APEG.
2. Une expression $p_1 * p_2$ est un APEG avec p_1 et p_2 sont des APEGs et $*$ est une opération binaire appartenant à $\{+,-,\times,\div\}$.
3. Une boite $\boxed{*(p_1,\ldots,p_n)}$ est un APEG, avec $* \in \{+,\times\}$ est une opération commutative et associative et $p_{i,1\leq i\leq n}$ sont des APEGs.
4. Un ensemble d'APEGs non vide $\{p_1,\ldots,p_n\}$ est un APEG où $p_{i,1\leq i\leq n}$ lui même n'est pas un ensemble d'APEGS. L'ensemble $\{p_1,\ldots,p_n\}$ est une classe d'équivalence.

Exemple 1.2.0

Afin de bien éclairer cette notion, un exemple de construction d'APEG est donné dans la Figure 1-1. Notons qu'une classe d'équivalence, représentée par les ellipses en pointillées (voir Figure 1-1), contient plusieurs APEGs $p_1,\ldots,p_n$. Pour former une expression valide, il faut sélectionner une seule expression $p_{i,1\leq i\leq n}$ dans chaque classe d'équivalence. Une boite $\boxed{*(p_1,\ldots,p_n)}$ est constituée d'une opération $*\in\{+,\times\}$ et de n expressions p_i, autrement dit, elle représente les différentes façons de combiner l'expression $p_1*\cdots*p_n$. D'un point de vue théorique, lorsque plusieurs expressions équivalentes contiennent une sous-expression commune, cette dernière est utilisée une seule fois dans l'APEG. Ce qui résulte que l'APEG, en compactant la représentation d'un ensemble d'expressions équivalentes, permet de de représenter dans une seule structure plusieurs expressions équivalentes. Dans la figure 1-1, pour des raisons de clarté, les feuilles a, b et c quand à elles, ont été dupliquées alors que réellement elles sont définies une seule fois dans la structure.

L'ensemble des expressions $\mathscr{A}(p)$ d'un APEG p est défini comme suit :

1. Si p est une constante *cst* ou un identificateur *id* alors $\mathscr{A}(p)$ = {v} ou $\mathscr{A}(p)$ = {x}.
2. Si p est une expression p_1*p_2 alors
$$\mathscr{A}(p)=\bigcup_{e_1\in\mathscr{A}(p_1),\ e_2\in\mathscr{A}(p_2)} e_1*e_2\ .$$
3. Si p est une boite $\boxed{*(p_1,\ldots,p_n)}$ alors $\mathscr{A}(p)$ contient toutes les combinaisons possibles de $e_1*\cdots*e_n$ pour chaque $e_1\in\mathscr{A}(p_1),\ldots,e_n\in\mathscr{A}(p_n)$.
4. Si p est une classe d'équivalence alors $\mathscr{A}(p)=\bigcup_{1\leq i\leq n}\mathscr{A}(p_i)$.

L'Équation (1.10) représente toutes les expressions équivalentes correspondant à l'APEG de l'expression $e=((a+a)+b)\times c)$ de la Figure 1-1.

$$\mathscr{A}(p)=\left\{\begin{array}{c}\big((a+a)+b\big)\times c,\ \big((a+b)+a\big)\times c,\ \big((b+a)+a\big)\times c,\\ \big((2\times a)+b\big)\times c,\ c\times\big((a+a)+b\big),\ c\times\big((a+b)+a\big),\\ c\times\big((b+a)+a\big),\ c\times\big((2\times a)+b\big),\ (a+a)\times c+b\times c,\\ (2\times a)\times c+b\times c,\ b\times c+(a+a)\times c,\ b\times c+(2\times a)\times c\end{array}\right\}\ . \tag{1.10}$$

■

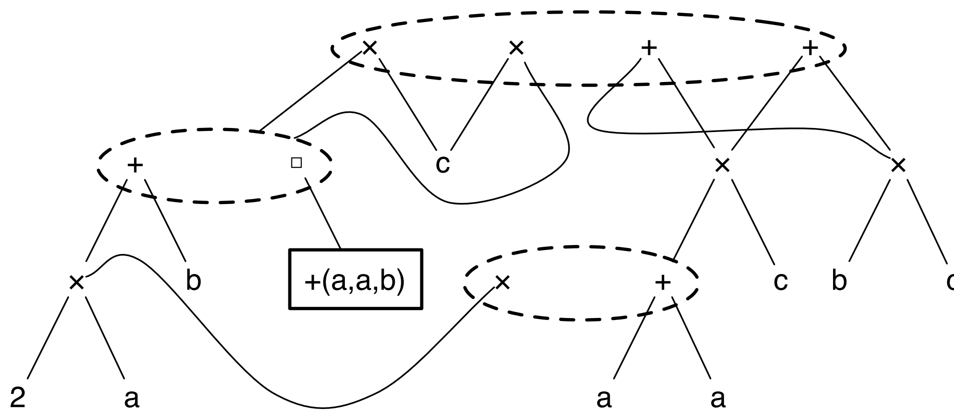


FIGURE 1-1 – *APEG correspondant à l'expression* $e = \big((a+a)+c\big) \times c$.

1.3 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 et non seulement les expressions arithmétiques. Notre transformation concerne les expressions arithmétiques comme l'addition, la multiplication, les fonctions trigonométriques, etc., mais aussi les morceaux de code tels que les affectations, conditionnelles et boucles. La syntaxe correspondant aux commandes est la suivante :

$$\text{Com} \ni c ::= id = e \mid c_1; c_2 \mid \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}_\Phi\ e\ \mathsf{do}\ c \mid \mathtt{nop}\ . \tag{1.11}$$

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 confusions causés par la lecture et l'écriture d'une même variables. Les différentes règles de transformation, données à la figure 1-2, 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, 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 :

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

Nous détaillons maintenant l'ensemble des règles de transformation intraprocédurale 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, $\delta : \mathcal{V} \rightarrow Expr$. 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 :

1. Les variables de l'expression e n'appartiennent pas à l'environnement δ.
2. La variable de référence ϑ que nous souhaitons optimiser n'est pas dans la liste noire β.
3. 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, re-parenthéser 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 lorsque l'on a connaissance statique de sa valeur. Dans le cas où la condition est `vraie` nous transformons la partie `then` et quand elle vaut `faux`, nous transformons la branche `else`. 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 sinon on rencontre des erreurs à l'exécution car des variables sont non initialisées $(C4)$. Nous mettons ces variables dans la liste noire β pour ne pas les supprimer dans une future utilisation.

Quant à la boucle while_Φ e 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

$$\frac{\delta' = \delta[id \mapsto e] \quad id \notin \beta}{\langle id = e, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \texttt{nop}, \delta', C, \beta\rangle} \quad (A1)$$

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

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

$$\frac{\begin{array}{c} C' = C\big[[];c_2\big] \quad \langle c_1, \delta, C', \beta\rangle \Rightarrow_\vartheta^* \langle c_1', \delta', C', \beta'\rangle \quad C'' = C[c_1';[]] \\ \langle c_2, \delta', C'', \beta'\rangle \Rightarrow_\vartheta \langle c_2', \delta'', C'', \beta''\rangle \end{array}}{\langle c_1\ ;\ c_2\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c_1'\ ;\ c_2', \delta'', C, \beta''\rangle} \quad (S3)$$

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

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

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

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

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

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

FIGURE 1-2 – *Règles de transformation pour améliorer la précision de programmes.*

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.

1.4 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 la 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 :

1. La variable de référence ϑ correspond à la même expression mathématique dans les deux programmes.
2. 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 1.4.1 (Comparaison des expressions arithmétiques)**.** Soit $v_1 = (x_1, \mu_1) \in \mathbb{E}$ 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 1.4.2 (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

$$\langle c_t, \delta_t, C, \beta \rangle \prec_\vartheta \langle c_o, \delta_o, C, \beta \rangle \tag{1.12}$$

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

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

- $\sigma_t(\vartheta) \sqsubseteq \sigma_o(\vartheta)$.
- Pour chaque $\text{id} \in \text{Dom}(\sigma_o) \setminus \text{Dom}(\sigma_t)$, $\delta_t(\text{id}) = e$ et $\langle e, \sigma \rangle \rightarrow_e^* \sigma_o(\text{id})$.

■

La Définition 1.4.2 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 formel δ_t, alors l'expression formelle correspondante est enregistrée dans δ_t.

Théorème 1.4.1 (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*

$$\big(\langle c_o, \delta_o, C, \beta \rangle \quad \Rightarrow_\vartheta \quad \langle c_t, \delta_t, C, \beta \rangle\big) \quad \Longrightarrow \quad \big(\langle c_t, \delta_t, C, \beta \rangle \quad \prec_\vartheta \quad \langle c_o, \delta_o, C, \beta \rangle\big) \ . \quad (1.13)$$

■

1.5 Résultats Expérimentaux

Nous avons développé un prototype qui transforme automatiquement des bouts de codes contenant des affectations, des conditionnelles, des boucles, 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.

1.5.1 Amélioration de la Précision Numérique

Les exemples de la Figure 1-3 illustrent le gain, en pourcentage, de précision numérique sur une suite de programmes provenant des systèmes embarqués et des méthodes d'analyse numérique [29]. 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 par 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 [43], nous remarquons que la précision numérique pour ce programme est améliorée de 19%.

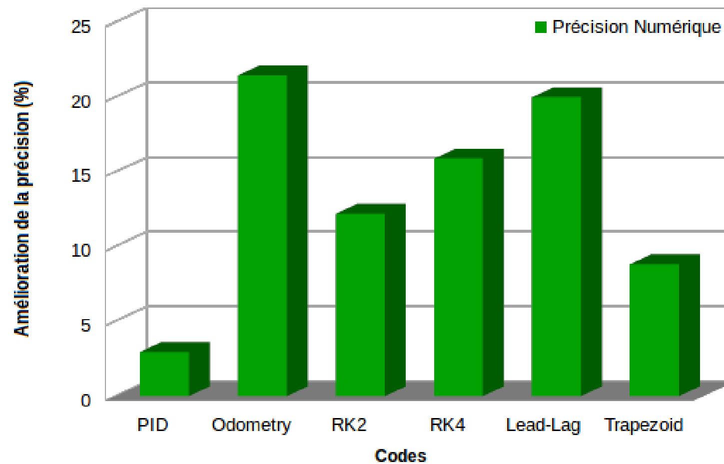


FIGURE 1-3 – *Pourcentage de l'amélioration de la précision numérique de programmes.*

1.5.2 Accélération de la Vitesse de Convergence

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 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`).

Par ailleurs, si on s'intéresse à analyser les performances de notre transformation en termes de temps d'exécution nécessaire pour chacune des méthodes numériques itératives précédemment mentionnées pour converger, la Table 1.3 donne un aperçu sur les expérimentations faites. À titre d'exemple, prenant la méthode de Jacobi. On remarque que l'accélération de la convergence du

	Original Code Execution Time in s	Optimized Code Execution Time in s	Percentage Improvement	Mean on n Runs
Jacobi	$1.49 \cdot 10^{-4}$	$0.38 \cdot 10^{-4}$	74.5%	10^4
Newton	$1.34 \cdot 10^{-3}$	$0.02 \cdot 10^{-3}$	98.4%	10^4
Eigenvalue	$4.50 \cdot 10^{-2}$	$3.07 \cdot 10^{-2}$	31.6%	10^3
Gram-Schmidt	$1.99 \cdot 10^{-1}$	$1.70 \cdot 10^{-1}$	14.5%	10^2

TABLE 1.3 – *Temps d'exécution des méthodes numériques itératives.*

programme a réduit le temps d'exécution requis de 74.5% par rapport au programme de départ. Globalement, nous accélérons la vitesse de convergence pour ces méthodes avec une moyenne de 20%.

Un autre point très important est de s'assurer que la transformation n'impacte pas le nombre d'opérations par itération, autrement dit, en accélérant la vitesse de convergence de ces méthodes, on va réduire le nombre d'itérations nécessaire à converger mais en même temps, on ne doit pas rajouter beaucoup d'opérations au programme initial. Pour se faire, on a compté le nombre d'opérations flottantes (flops) requis pour chaque itération que ça soit pour le programme de départ ou celui après optimisation, ainsi le nombre total des opérations pour toutes les itérations. La Table 1.4 détaille les résultats obtenus.

Method	$\sharp$ of $\pm$ per it Original Code	$\sharp$ of $\pm$ per it Optimized Code	Total $\sharp$ of $\pm$ Original Code	Total $\sharp$ of $\pm$ opt Optimized Code	Percentage of Improvement
Jacobi	13	15	25389	24420	3.81
Newton-Raphson	11	11	3465	132	96.19
Eigenvalue	15	15	694080	685995	1.16
Gram-Schmidt	21	19	791364	715996	9.52
Method	$\sharp$ of $\times$ per it Original Code	$\sharp$ of $\times$ per it Optimized Code	Total $\sharp$ of $\times$ Original Code	Total $\sharp$ of $\times$ opt Optimized Code	Percentage of Improvement
Jacobi	28	14	54684	22792	58.32
Newton-Raphson	27	26	8505	312	96.33
Eigenvalue	19	19	879168	868927	1.16
Gram-Schmidt	22	20	712316	647560	9.09

TABLE 1.4 – *Nombre d'opérations flottantes nécessaire aux programmes pour converger.*

1.5.3 Formats de Variables

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 $[a,b]=[1.9,2.1]$ où le polynôme s'évalue très mal dans l'arithmétique des nombres flottants.

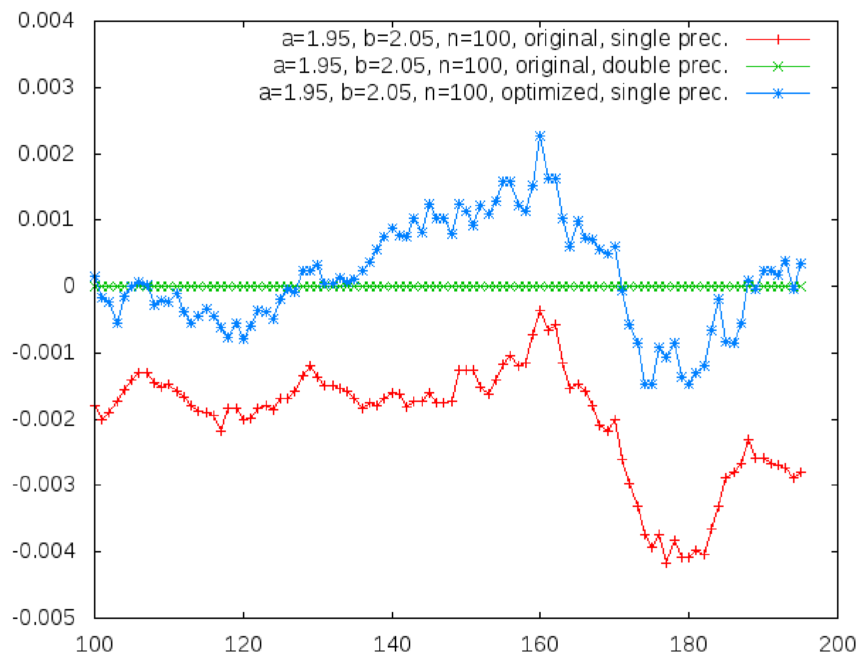
$$s=\int_{a}^{a+nh} f(x)\,\mathrm{d}x, \quad 0 \le n < \frac{b-a}{h}, \tag{1.14}$$

avec n est le nombre de sous-intervalles de [a,b] et h est la largeur des sous-intervalles.

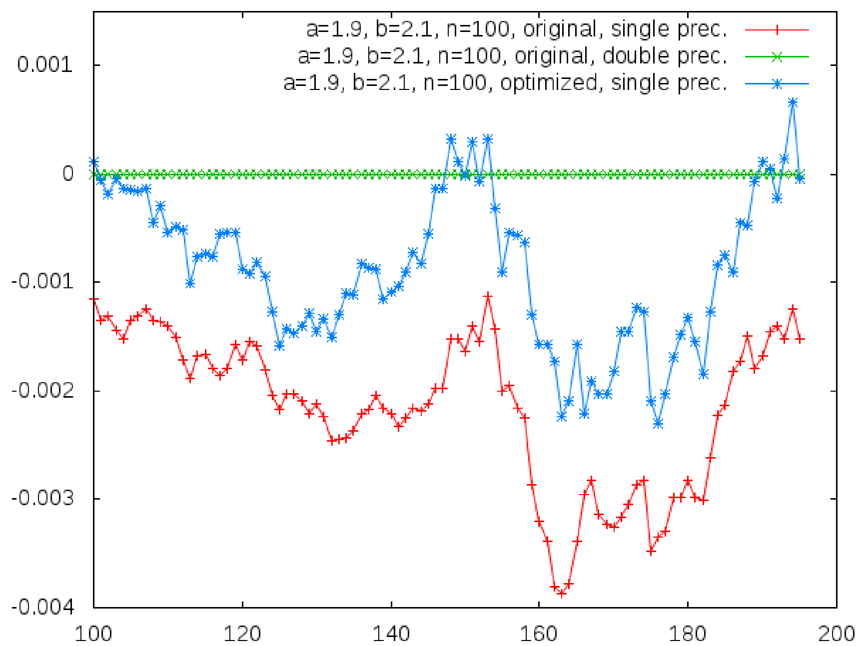
Nous avons comparé trois programmes, le premier code source écrit en simple précision (32 bits), un deuxième code source en double précision et le troisième programme qui est celui transformé en simple précision (32 bits). La Figure 1-4 illustre les résultats obtenus. Nous voyons que le programme transformé en 32 Bits est très proche de celui de départ en 64 Bits. L'intérêt 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.

1.6 Conclusion

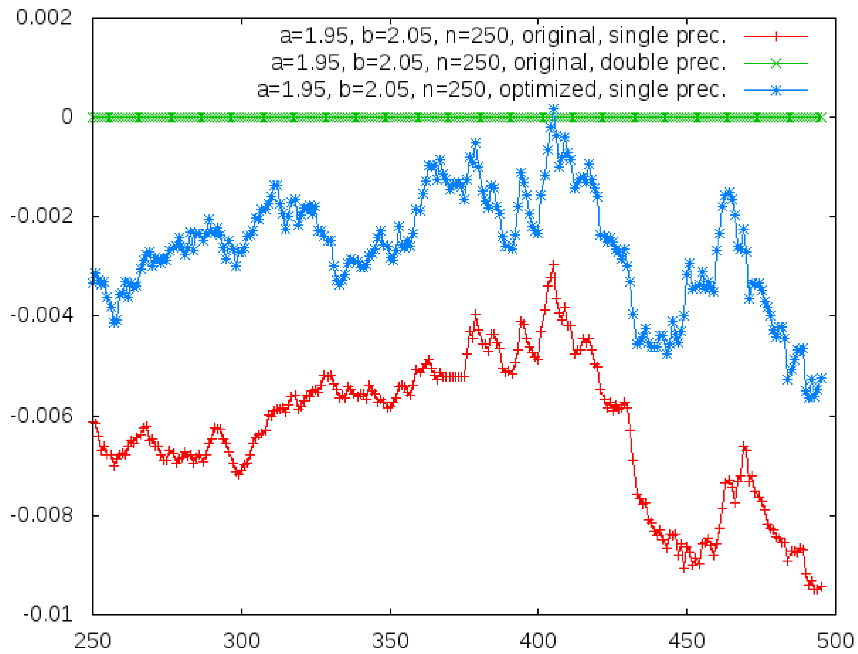
L'objectif principal de notre travail est d'améliorer la précision numérique des calculs par transformation automatique de programmes. Nous avons défini un ensemble de règles de transformation intra-procédurale qui ont été implémentées dans un outil appelé `Salsa`. Notre outil, basé sur les méthodes 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



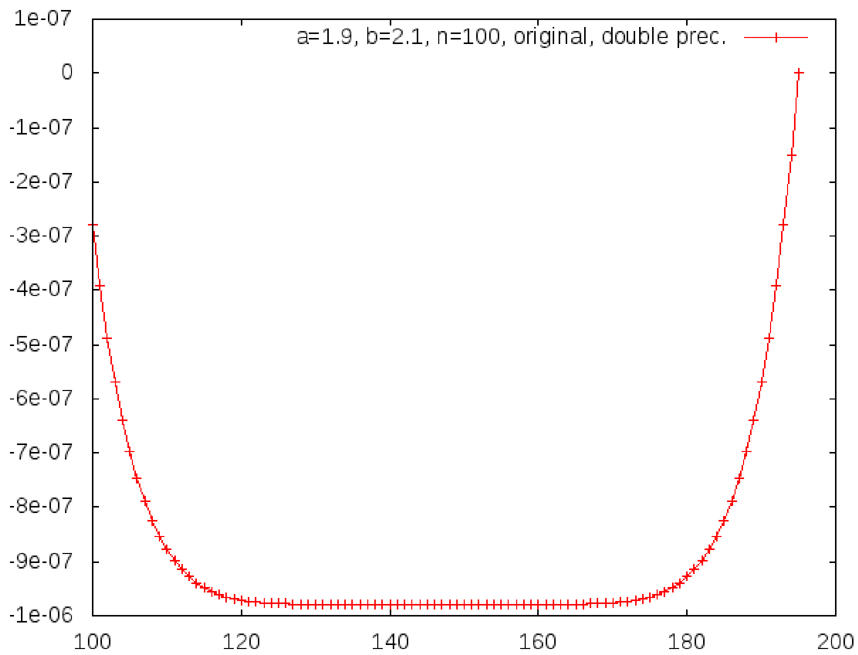
(a)



(b)



(c)



(d)

FIGURE 1-4 – *Résultats de la simulation de la méthode de Simpson avec simple, double précision et le programme optimisé utilisant Salsa. Les valeurs des axes x et y correspondent respectivement aux valeurs de n et s de l'Equation 1.14.*

présentés dans la Section 1.5 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.

RÉSUMÉ CHAPITRE 2

Les ordinateurs utilisent les nombres à virgule flottante pour approximer des valeurs de type réel. Les nombres flottants ne sont pas exacts vu qu'ils sont représentés sur un nombre fini de bits. Cela engendre une perte d'information causée par les erreurs d'arrondi. Il est à noter que les opérations élémentaires comme l'addition et la multiplication perdent leurs propriétés mathématiques telles que l'associativité et la distributivité. De plus, l'arithmétique flottante n'est pas intuitive, ce qui rend difficile de comparer deux expressions mathématiques équivalentes qui sont écrites différemment.

Les systèmes critiques utilisant l'arithmétique flottante nécessitent de vérifier et de valider leurs traitements afin d'augmenter la confiance en leur sûreté et leur fiabilité. Malheureusement, les techniques existantes fournissent souvent une surestimation des erreurs de calculs. Citons Ariane 5 et le missile Patriot comme fameux exemples de désastres causés par les erreurs de calculs. Ces dernières années, plusieurs techniques concernant la transformation d'expressions arithmétiques pour améliorer la précision numérique ont été proposées [56].

Notre travail consiste à transformer des programmes, basés sur l'arithmétique de nombres flottants, contenant des affectations, des structures de contrôle, des boucles, des séquences de commandes et des fonctions. Nous définissons un ensemble de règles de transformation permettant la réécriture automatique, en un temps polynomial. Ces règles permettent de créer des expressions plus larges en appliquant des calculs formels limités. Notre approche se base sur les techniques d'analyse statique par interprétation abstraite pour sur-approximer les erreurs d'arrondi dans les programmes.

Dans ce chapitre, nous introduisons l'objectif de notre travail à travers un exemple couram-

ment utilisé en avionique, l'algorithme du PID, qui permet de maintenir une mesure à une certaine valeur. Nous donnons différentes implémentations de cet algorithme ainsi que les résultats obtenus. Nous présentons de façon informelle également la principale contribution de ce travail qui est une transformation source à source de programmes pour améliorer la précision numérique. Par la suite, nous présentons l'organisation de ce manuscrit. Nous clôturons ce chapitre par la liste de différentes publications faites le long de ces trois années.

CHAPITRE 2

INTRODUCTION

Contents

2.1 Context **1**
2.2 Motivating example **4**
2.2.1 PID Controller 4
2.2.2 Experimental Results 9
2.2.3 Conclusion 11
2.3 Contribution **12**
2.3.1 Formal Transformations Rules for Programs 12
2.3.2 A Theorem for Correctness 12
2.3.3 Experimental Results 13
2.4 Organization of This Work **15**
2.5 Publications **16**

2.1 Context

Recent advances in Computer Science resulted in the development of sophisticated devices and complex software. However, this has also brought additional challenges to ensure properties of

performance and safety especially when these systems are used in aerospace and avionics. In particular, the accuracy of numerical processings may have an important impact on the validity of programs, since a small error in computations due to the round-off errors, may generate into dramatical consequences. As well-known examples of disasters, we may cite : In 1998, the USS Yorktown ship stopped in the middle of the sea for several hours because of a division by zero [77]. Another famous bug happened in 1996 that consists in the failure of the first Ariane 5 rocket flight [6]. The main reason of this failure is the overflow of a variable. More precisely, the engineers had taken the same flight software designed for Ariane 4 and forgot to update it to deal with greater values of the variable related to the horizontal speed of Ariane 5. This is why it exploded 39 seconds after launch. Another bug concerns the Patriot rocket which failed to intercept the Iraqi Scud rockets [5] during the Gulf War, in 1991. The reason is an error of computation. More precisely, the rocket used a counter which adds $1/10$ for every tenth of second. The problem is that in the value $1/10$ is not representable in binary. This value when encoded on 24 bits generates an error of approximatively 9.5×10^{-8} which gives an error of 0.34 seconds after one hundred hours of execution. Note that the speed of the rocket is $1676ms^{-1}$, by consequence, the system had an error of position of more than 500 meters on the distance between the two rockets. The consequences of this error were heavy, 28 soldiers died and 100 others were wounded. This last computer bug, is due to a loss of accuracy in the computations. In our case, we are interesting in such problems due to the numerical accuracy.

Computers may often use floating-point arithmetic to approximate real numbers. The floating-point arithmetic is defined by the `IEEE754` standard [10, 76]. A floating-point value is represented on a finite number of bits. Floating-point numbers may be inaccurate since information is lost because of the round-off errors. Therefore, the basic operations like addition and multiplication lose their mathematical properties such as associativity, distributivity, etc [46, 71]. If we consider three real numbers a, b and c, the three following sums

$$(a+b)+c,$$
$$a+(b+c),$$
$$b+(a+c)$$

return the same result regardless of the bracketing order. But, if the same expressions are evaluated in floating-point arithmetic, they generally give different results. In a critical environment such as the flight control system of an aircraft [39, 87], a bad implementation of the computations may cause a significant loss of accuracy and then invalidate the system. Similarly,

when performing many computations based on floating-point numbers on parallel computers to simulate complex physical phenomena, the floating-point arithmetic may make the program non-reproductible [21, 40, 41] or make the stabilization speed of numerical convergent methods slow [28]. By carefully rewriting the floating-point computations, the problems can be solved. But the way to rewrite is not easy and usually only done by numerical experts. Our research focuses on the development of automatic program transformations to improve the accuracy of the numerical computations.

The floating-point arithmetic is not intuitive. It is very difficult to compare two mathematically equivalent expressions written differently. To illustrate at best this notion, let us consider the example 1.1.1 given hereafter.

Example 2.1.1

Let us consider in this example two functions f and g which are mathematically equivalents. Note that f is the developed form of g. We have

$$f(x) = x^2 - 2.0 \times x + 1.0$$

and

$$g(x) = (x - 1.0) \times (x - 1.0) \ .$$

The main idea of this example is to check if the results returned by these two equivalents functions are the same when evaluating them on a computer using the floating-point arithmetics. These two functions are implemented in the `Ocaml` 4.01.0 language.

- If we compute $f(0.999)$, a value near the root of the polynomial, we obtain :

```
# let f x = x *. x -. 2.0 *. x +. 1.0 ;;
  val f : float -> float = <fun>
# f 0.999;;
  - : float = 1.00000000002875566 e-06
```

- If we compute $g(0.999)$, the result of the computation is :

```
# let g x = (x -. 1.0) *. (x -. 1.0) ;;
  val g : float -> float = <fun>
# g 0.999 ;;
  - : float = 1.00000000000000186 e-06
```

As we can observe, on small computations, we have already obtained different results because of the round-off errors arising during the evaluation. In other words, the rewritings of arithmetic expressions may have a significant impact on the accuracy of the computations.

2.2 Motivating example

2.2.1 PID Controller

To illustrate the objective of our work, we consider a `PID` algorithm [12, 13]. We apply on our `PID` implementation, in an imperative language, a set of appropriate transformations in order to improve the numerical accuracy of its computations [31]. More precisely, we take the original listing of a `PID` implementation and we apply on it several processings in order to generate two other `PID` implementations which are mathematically equivalent to the original one but more accurate. The first transformation only rewrites the assignments while, in the second transformation, the loop is also unfolded. While these transformations are made by hand, they are applied systematically, in the way which we have automatized later (see Chapter 4). The experimental data obtained by comparing the executions of these three codes show that the rewritings have a significant impact on the accuracy of the computations.

Hereafter, we only focus on the first results obtained when rewriting programs by hand. We can summarize it as follows

- We provide a description illustrating how a `PID` controller works,
- We give the original listing of a `PID` controller, and then we explain the process adopted to rewrite it in two ways,
- We illustrate the main results observed by the different experiments obtained.

Description of the PID Controller

The PID [12, 13, 31] is a widely used algorithm in embedded and critical systems, like aeronautic and avionic systems. It keeps a physical parameter (m) at a specific value known as the *setpoint* (c). In other words, it tries to correct a measure by maintaining it at a defined value. The original `PID` program is given in Listing 2.1. The error being the difference between the *setpoint* and the measure, the controller computes a correction based on the integral i and derivative d of the

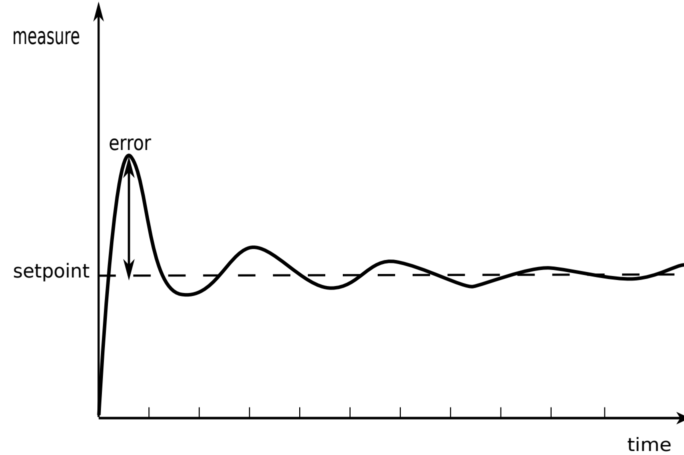


FIGURE 2-1 – *Description of a PID Controller.*

error and also from a proportional error term p. The weighted sum of these terms contributes to improve the reactivity, the robustness and the speed of the `PID` algorithm. More precisely the three terms are :

i) The proportional term p : the error e is multiplied by a proportional factor k_p,

$$p = k_p \times e \quad .$$

ii) The integral term i : the error e is integrated and multiplied by an integral factor k_i,

$$i = i + (k_i \times e \times dt) \quad .$$

iii) The derivative term d : The error e is differentiated with respect to time and is then multiplied by the derivative factor k_d. Let e_{old} denotes the value of the error at the previous iteration, we have :

$$d = k_d \times (e - e_{\text{old}}) \times \frac{1}{dt} \quad .$$

In practice, there exists many other ways to compute the terms d and i. In our implementation they are computed by Euler's method and the rectangle method respectively. The transformations described in the next sections are independent of these specific algorithms.

Listing 2.1 – Listing of the original code of $\texttt{PID}_1$.

```
kp = 9.4514; ki = 0.69006; kd = 2.8454; invdt = 5.0; dt = 0.2;
m = 8.0; c = 5.0; eold = 0.0;
while true {
      e = c - m;
      p = kp * e;
      i = i + (ki * dt * e);
      d = kd * invdt * (e - eold);
      r = p + i + d;
      eold = e;          // updating memory
      m = m + 0.01 * r;  // computing measure: the plant
}
```

How to Get a More Accurate PID ?

We detail here the different steps needed to transform the original PID program, named $\texttt{PID}_1$, into a new equivalent program named $\texttt{PID}_2$. The main idea consists of developing and simplifying the expressions of $\texttt{PID}_1$ and inlining them within the loop, in order to extract the constant expressions and to reduce the number of operations. At the nth iteration of the loop, we have

$$\begin{aligned} p_n &= k_p \times e_n \ , \\ i_n &= i_{n-1} + (k_i \times e_n \times dt) \ , \\ d_n &= k_d \times (e_n - e_{n-1}) \times \frac{1}{dt} \ . \end{aligned}$$

If we inline the expressions of p_n, i_n and d_n in the formula of the result expression r_n and after extracting the common factors, we get

$$r_n = e_n \times \left(k_p + k_d \times \frac{1}{dt}\right) + i_0 + k_i \times dt \times \sum_{i=1}^{n} e_i - k_d \times e_{n-1} \times \frac{1}{dt} \ . \tag{2.1}$$

Then we remark that there exists some constant sub-expressions (i.e. independent of the iteration number) in Equation (2.1). So, we compute them once before entering into the loop. We have

$$c_1 = k_p + k_d \times \frac{1}{dt} \ , \qquad c_2 = k_i \times dt \ , \qquad c_3 = k_d \times \frac{1}{dt} \ .$$

Next, we record in a variable s the sum $s = \sum_{i=0}^{n-1} e_i$ which adds the different errors from e_0 to e_{n-1}. Finally, we have

$$r_n = R + i_n \qquad \text{with} \qquad R = (c_1 \times e_n) - (c_3 \times e_{n-1}) \ .$$

Our $\mathtt{PID}_2$ algorithm is given in Listing 2.2.

Listing 2.2 – Listing of the optimized code of $\mathtt{PID}_2$.

```
kp = 9.4514; ki = 0.69006; kd = 2.8454; invdt = 5.0; dt = 0.2;
m = 8.0; c = 5.0; eold = 0.0; R = 0.0; s = 0.0;
c1 = kp + kd * invdt; c2 = kd * invdt; c3 = ki * dt;
while true {
    e = c - m;
    i = i + c2 * e;
    R = (c1 * e) - (c3 * eold);
    r = R + i;
    eold = e;
    m = m + 0.01 * r;
}
```

How to Get an Even More Accurate PID ?

The program $\mathtt{PID}_1$ can be transformed even more drastically by unfolding the loop. In our case, we arbitrarily choose to unfold it four times in order to keep for each execution the sum of the last four errors. Then, we change the order of the operations, either by summing the terms pairwise, or in increasing or decreasing order. The process applied to get the third PID program, named $\mathtt{PID}_3$, is given in the following. Let us start by unfolding four times the integral term, as usually done by static analyzers or compilers :

$$\begin{aligned} i_{n-1} &= i_{n-2} + k_i \times dt \times e_{n-1} \ , \\ i_{n-2} &= i_{n-3} + k_i \times dt \times e_{n-2} \ , \\ i_{n-3} &= i_{n-4} + k_i \times dt \times e_{n-3} \ , \\ i_{n-4} &= i_{n-5} + k_i \times dt \times e_{n-4} \ . \end{aligned}$$

We inline the previous expressions in i_n, we obtain

$$\begin{aligned} i_n = i_{n-5} + (k_i \times dt \times e_{n-4}) + (k_i \times dt \times e_{n-3}) + (k_i \times dt \times e_{n-2}) \\ + (k_i \times dt \times e_{n-1}) + (k_i \times dt \times e_n) \ . \end{aligned} \tag{2.2}$$

Listing 2.3 – Listing of the optimized code of $\mathtt{PID}_3$.

```
kp = 9.4514;  ki = 0.69006;  kd = 2.8454;  invdt =  5.0;  dt = 0.2; R = 0.0;  S = 0.0;
s = 0.0;  m = 8.0;  c = 5.0;  eold= 0.0; e1 = e2 = e3 = e4 = 0.0; k1 = kp + kd * invdt;
k2 = ki * dt; k3 = kd * invdt;
while true {
      e = c - m;
      i = i + k2 * e;
      R = (k1 * e) - (k3 * eold);
      S = s + (e4 + (e3 + (e2 + e1)));
      r = R + i + (k2 * S);
      eold= e;  e4 = e3;  e3 = e2;  e2 = e1;  e1 = e;
      m = m + 0.01 * r;
}
```

with $i_{n-5} = i_0 + k_i \times dt \times \sum_{i=1}^{n-5} e_i$. Equation (2.2) can be even more simplified, we have :

$$i_n = i_0 + k_i \times dt \times \sum_{i=1}^{n-5} e_i + (k_i \times dt \times ((((e_{n-4} + e_{n-3}) + e_{n-2}) + e_{n-1}) + e_n)) \quad .$$

Now, if we come back to the result expression after having done some manipulations, like developing the derivative and factorizing, we obtain as a final expression

$$r_n = e_n \times \left(k_p + k_d \times \frac{1}{dt}\right) + i_0 + k_i \times dt \times \sum_{i=1}^{n-5} e_i - k_d \times \frac{1}{dt} \times e_{n-1}$$
$$+ k_i \times dt \times ((((e_{n-4} + e_{n-3}) + e_{n-2}) + e_{n-1}) + e_n) \quad .$$

Denoting by

$$s = \big(((e_{n-4} + e_{n-3}) + e_{n-2}) + e_{n-1}\big),$$
$$k_1 = k_p + \left(k_d \times \frac{1}{dt}\right),$$
$$k_2 = k_i \times dt,$$

and

$$k_3 = k_d \times \frac{1}{dt},$$

the final expression of r_n is

$$r_n = R + i + k_2 \times \left(s + \sum_{i=1}^{n-5} e_i\right) \qquad \text{with} \qquad R = (e_n \times k_1) - (k_3 \times e_{n-1}) \ .$$

The complete code of $\texttt{PID}_3$ is given in Listing 2.3. Remark that, the transformation proposed in this section leads to a larger program requiring more memory for its execution. While it allows more transformations it may be irrelevant in some contexts where memory is critical.

2.2.2 Experimental Results

Let us focus now on the execution of the three `PID` programs introduced in Section 2.2.1. In our Python implementation [2], version `2.7.9`, using the *GMPY2* library for multiple precision computations [1, 3], the results obtained show that there is a significant difference between $\texttt{PID}_1$, $\texttt{PID}_2$ and $\texttt{PID}_3$ from the second or third digit of the decimal values of the result. To better visualize these results, the curves corresponding to the three `PID` programs are given in Figure 2-2. We can observe a significant difference between the curves corresponding to the three `PID`, mainly between the values 0 and 150 of the x-axis.

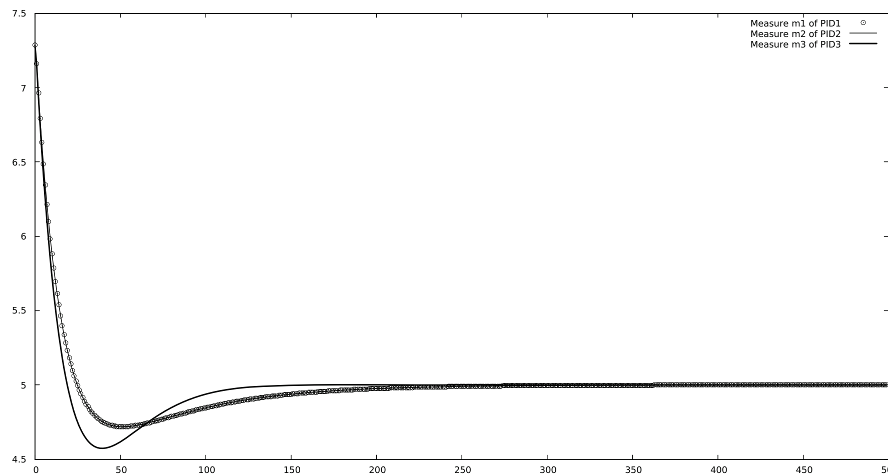


FIGURE 2-2 – *Value of the measure m in the three `PID` algorithms.*

We give in Figure 2-3 a zoom of Figure 2-2 in order to more illustrate the difference between the three curves corresponding to the three `PID` programs, mainly in the range 0 to 80 of the x-axis and 0 to 5.5 of the y-axis.

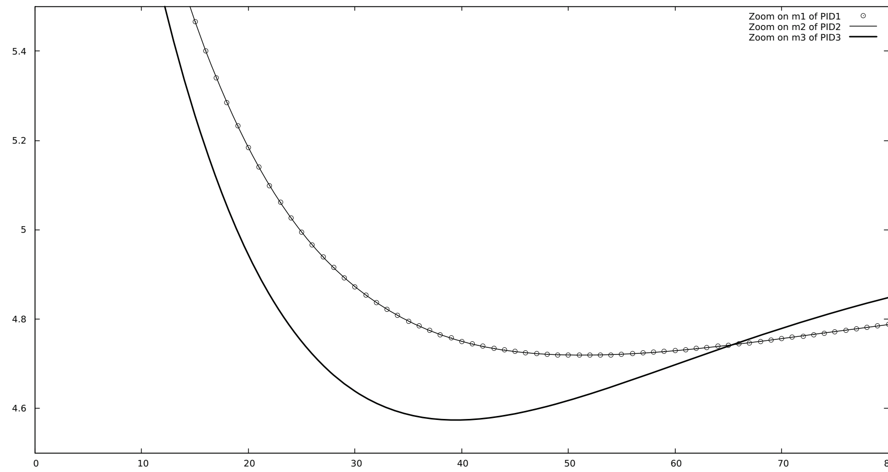


FIGURE 2-3 – *Zoom on the measure m in the three PID algorithms on the interval [0,80] [0,5.5].*

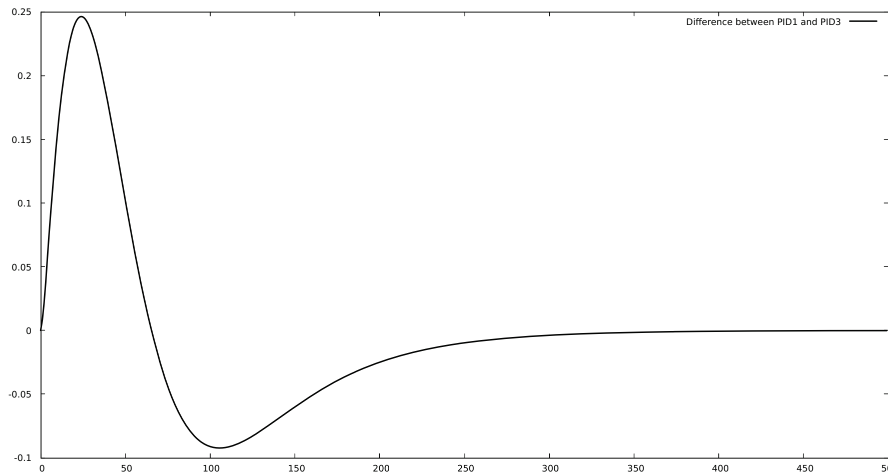


FIGURE 2-4 – *Difference between the values of the measure m in PID_1 and PID_3.*

Figure 2-4 shows the difference between $\mathtt{PID}_1$ and $\mathtt{PID}_3$. This difference, which is significantly important, is computed with many precisions. So, the same behavior was observed by using 24, 53 and 50000 bits of the mantissa. The error between $\mathtt{PID}_1$ and $\mathtt{PID}_3$ oscillates between -0.1 and 0.25 while the value ranges between 5 and 8.

We also observe that the differences between $\mathtt{PID}_1$ and $\mathtt{PID}_2$ are negligible. Concerning $\mathtt{PID}_1$ and $\mathtt{PID}_3$, we can remark that a small syntactic change in the code indeed yields an important difference in term of accuracy. For example, let us take the following expression r of $\mathtt{PID}_1$ and let us just inline the three terms p, i and d and factorize e in it. Initially, $r = p + i + d$ and we

obtain after factorizing :

$$r' = e \times \left(k_p + k_i \times dt + k_d \times \frac{1}{dt}\right) + i_0 - \left(k_d \times \frac{1}{dt} \times eold\right) \quad .$$

With this simple modification, the difference in accuracy is already important, as shown in Figure 2-5 which gives the difference between r and r' and between m and m' for the first iterations of the loop.

It	r	r'	m	m'
1	−71.449234	−71.449242	7.285508	7.285508
2	−12.165627	−12.993700	7.163851	7.155571
3	−19.748718	−21.010429	6.966364	6.945466
4	−17.074722	−18.747597	6.795617	6.757990
5	−16.089169	−18.077232	6.634725	6.577218
6	−14.934349	−16.752943	6.485382	6.409688
7	−13.892138	−15.672437	6.346460	6.252964
8	−12.913241	−14.609431	6.217328	6.106869
9	−12.000034	−13.609982	6.097328	5.970769
10	−11.147227	−12.662717	5.985856	5.844142

FIGURE 2-5 – *Comparison between results r and r' and between the corresponding measures m and m'.*

2.2.3 Conclusion

Our main attention focused here on the transformation of a standard `PID` algorithm. We will show in Chapter 4 how to automatize transformations to produce $\texttt{PID}_2$ and $\texttt{PID}_3$ from $\texttt{PID}_1$ (see Section 2.2.1). These transformations use systematic and general rules independent of the sample program used in this case study. These rules include the inlining of expressions, partial evaluation and loop unfolding. The results obtained when running the three codes show that these transformations impact significantly the accuracy of the results by up to several percents.

The developed software, based on the rules mentioned in the former paragraph as well as rewriting rules for the expressions (associativity, commutativity, etc.), aims at taking as input an original program, like $\texttt{PID}_1$, and at generating automatically other `PID` programs which are equivalents mathematically and more accurate.

2.3 Contribution

2.3.1 Formal Transformations Rules for Programs

In this thesis, we propose an automatic source to source transformation which ensures that the optimized codes are mathematically equivalent to the source code and more accurate in the floating-point arithmetic. In other words, our main contribution is to improve the numerical accuracy of computations by automatically rewriting programs. Note that the generated programs do not necessarily have the same semantics as the original program but are mathematically equivalent. Martel in [66] introduced a semantic-based programs transformation [26] for floating-point arithmetic expressions. This method enables one to automatically rewrite a formula into another one which is mathematically equivalent and numerically more accurate. The step further, which is the main objective of this thesis, is to transform automatically pieces of code, *i.e.*, assignments, conditionals, loops, functions, etc.

These transformation rules have been implemented in a tool named `SALSA` [29]. It takes as input an imperative program and intervals for its input data and generates as output a more accurate program. Generally, the original program is totally different from the optimized program, because, they are some variables which have disappeared from the code and in the same time new ones appeared. More clearly, both of these programs may not have necessary the same semantics but mathematically they compute the same outputs.

The tool proposed behaves like a compiler at the difference that it uses the results of a static analysis by abstract interpretation [23, 24] which provides intervals for each variable at every control point. It has been developed with the objective to be easy to use and efficient to correct the computations of programs coming from different areas like embedded systems and physics. The rewriting is done by a set of transformation rules that we have defined and implemented. Applied deterministically, these rules allow one to get a more accurate program among all these considered. In addition, our tool takes as input ranges of variables which are or introduced by the user or coming from embedded sensors that transform physical measurements into digital data values.

2.3.2 A Theorem for Correctness

To ensure the correctness of our transformation, we have introduced a theorem which compares two programs and shows that the more accurate program may be used instead of the less accurate

one [34]. This proof uses a classical operational semantics for both arithmetic expressions and commands. To compare two programs, we need to specify the *target variable* that we aim at improving the accuracy. More precisely, the transformed program p_t is more accurate than the original program p_o if and only if the following conditions are satisfied

1. The *target variable* ϑ consists in the same mathematic expression in p_o and p_t.
2. The variable ϑ is more accurate in the optimized program p_t than in p_o.

Currently, we optimize only one *target variable* at once from the ranges of the intervals associated to the values of the program at the beginning and the bounds of the errors computed by abstract interpretation [64, 65].

2.3.3 Experimental Results

Another contribution of this thesis is a series of experiments and, in particular, the impact of improving the accuracy of programs on :

- The acceleration of the convergence speed of numerical iterative methods.
- The data-types of the variables which may be optimized to obtain the same accuracy with less bits.

A very interesting advantage of our method is its efficiency to accelerate the convergence speed of some well known numerical iterative methods like Jacobi's method, Newton-Raphson's method, iterative Gram-Schmidt method and a method to compute the eigenvalues of a matrix by optimizing their accuracy. More particularly, we have succeeded to reduce their required number of iterations to converge.

A step further, we are interested in observing the number of floating-point operations (flops) used in both original and optimized program of the numerical iterative methods, because we do not want to reduce the number of iterations by increasing the number of computations per iteration into the transformed program. The results show that the total number of floating-point operations (flops) in the transformed program is less than the total number of flops in the sources code of these methods. We have also computed the execution time needed by each of these methods (before and after optimization). The experimental results show that the gain is always positive.

Another criterion consists in studying the effect of the accuracy improvement of programs on their data-type formats of variables, in single and double precision. More precisely, we compare

two original programs written respectively in single (32 bits) and double precision (64 bits) with the transformed program in single precision. The experimentations show that the optimized program (32 bits) is far close to the exact result of the original program (64 bits). This study is very interesting because it allows one to the user to degrade in the precision without loss much of information, otherwise, he saves the memory storage space, the bandwidth and the necessary execution time. Along of this thesis, our programs are written in the `C` language and compiled by the `GCC` compiler version 4.9.2 with the `-o0` optimization level.

2.4 Organization of This Work

This thesis is organized as follows.

Chapter 3 **Background and Related Work.** This chapter discusses the arithmetic models, *i.e.*, the floating-point and interval arithmetic [10, 73, 74, 76]. It describes static analysis by abstract interpretation which is a central technique used in our tool. Next, it presents the related work of A. Ioualalen [56] that concerns the transformation of arithmetic expressions made of additions, subtractions, multiplications and divisions using Abstract Program Equivalence Graphs (APEGs).

Chapter 4 **Intraprocedural Transformation.** This chapter details the main contribution of this work. It presents how transforming automatically more than arithmetic expressions, in other words, pieces of code. More precisely, it consists in the intraprocedural program transformation such as assignments, conditionals, loops, sequences of commands. The intraprocedural transformation rules used by our tool are detailed in this chapter. We give also a full example of transformation with the different steps required to achieve the optimized program by using our tool. Finally, we end with a proof of correctness of the transformation relying on an operational semantics.

Chapter 5 **Interprocedural Transformation.** This chapter concerns the interprocedural program transformation. We are interesting in transforming functions. The interprocedural transformation rules are given in details in this chapter as well as a set of examples illustrating how the interprocedural rules behave.

Chapter 6 **Numerical Accuracy.** This chapter describes two applications done with our approach. It describes the various examples taken from many applicative domains and gives the listing of programs before and after being transformed. `SALSA` handles the different examples taken to study the improvement of the numerical accuracy of programs and the impact of optimizing the accuracy on the formats of the variables considered. It shows by how much the accuracy of programs is improved, and that their execution time is reduced. In addition, it gives the results obtained when studying the impact of the accuracy on the data-type formats of variables such as single and double precision.

Chapter 7 **Convergence Acceleration.** This chapter studies the relation between improving the numerical accuracy and the convergence speed of some representative numerical iterative methods. It illustrates by how many the number of iterations of the considered methods needed to converge is reduced.

Chapter 8 **Conclusion and Future Work.** This last chapter concludes and outlines our perspectives and future work direction.

2.5 Publications

This thesis has given rise to several publications in conference proceedings and journals described hereafter.

1. **International Journal on Software Tools for Technology Transfer (STTT), 2016** [34]
   This article focuses on the transformation of intraprocedural pieces of code to automatically improve their numerical accuracy. The transformation consists in optimizing a target variable with respect to some given ranges for the input variables of the program. This work is in an extension of the article [29]. It presents with more details the transformation done by our tool. In addition, it introduces the correctness proof of the transformation rules. In order to ensure the soundness of our transformation, the theorem makes a comparison between two programs and states that the more accurate program may be used in place of the less accurate one.

2. **International Workshop on Numerical Software Verification (NSV), 2016** [36]
   We introduce in this article FPBench, a standard benchmark format for validation and optimization of numerical accuracy in floating-point computations. FPBench is a first step toward addressing an increasing need in our community for comparisons and combinations of tools from different application domains. To this end, FPBench provides a basic floating-point benchmark format and accuracy measures for comparing different tools. The FPBench format and measures allow comparing and composing different floating-point tools. We describe the FPBench format and measures and show that FPBench expresses benchmarks from recent papers in the literature, by building an initial benchmark suite drawn from these papers. We intend for FPBench to grow into a standard benchmark suite

for the members of the floating-point tools research community. This work has been done in collaboration with the University of Washington (Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern and Zachary Tatlock).

3. **Groupement de Recherche Génie de la Programmation et du Logiciel (GDR-GPL), 2016** [32]
This article, written in french, summarizes the intraprocedural transformation of programs published in [29]. We give the different applications of our techniques [28, 33] such as the effect of improving the accuracy of programs on the acceleration speed of numerical iterative methods and on the data-types of variables (single or double precision). We detail then the experimental results obtained by our tool.

4. **IEEE International Conference on Control, Decision and Information Technologies (CODIT), 2016** [33]
In this work, we examine the impact of improving the numerical accuracy of programs on their data-types formats, single or double precision. To do this, we have compared two source codes written respectively in single and double precision with the transformed program obtained with our tool in single precision. The results show that the optimized program in single precision is very close to the original program executed with a double precision. These experiments emphasize the efficient of our transformation and allow the user to be more accurate by degrading in the precision and working with smaller data-type format (single precision) while obtaining results close to the exact values. The benefit of this work is to provide to the user the opportunity of reducing the use of the memory space, the bandwidth and the required execution time without losing much of information.

5. **International Symposium on Code Generation and Optimization (CGO), 2016** [35]
It consists in a poster and 2 pages article which is made of a summary of all our results mainly the improvement of the numerical accuracy of programs, the acceleration of their convergence speed, the reduction of the data-type format of programs. We illustrate the gain of these applications by graphics made of most of our examples with the percentage of improvement.

6. **International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR), 2015** [28]
   We have explored in this research several numerical iterative methods to detect the impact of improving their numerical accuracy on their convergence speed. We have emphasize our experiments using the Jacobi's method, Newton-Raphson's method, an iterative Gram-Schmidt method and finally a method that computes the eigenvalues of a matrix. We have demonstrated that we reduce the total number of iterations required by these methods to converge by about 20% when their accuracy is optimized.

7. **ACM International Conference on Computing Frontiers (CF), 2015** [30]
   The main goal of this paper is to apply our tool on a larger application. This application concerns the simulation of a rocket trajectory in the space, it is about one hundred lines of code. We outline the significant results obtained by using our tool in terms of accuracy. The difference between the two trajectories corresponding to the original and the transformed program is very important.

8. **International Workshop on Formal Methods for Industrial Critical Systems (FMICS), 2015** [29]
   The main contribution of this publication is the intraprocedural transformation of programs. It focuses mainly on a set of transformation rules that allow us to improve the numerical accuracy of programs made of assignments, conditionals, while loop and sequences of commands. A set of transformation rules has been defined and implemented in our tool. The tool takes as input an imperative original program and generates in the output another program more accurate.

9. **International Workshop on Numerical Software Verification (NSV), 2014** [31]
   It consists in the first work in the direction of improving the numerical accuracy of programs where we transform by hand programs. For our case study, we have taken as example a PID controller. From the original source code, we rewrite two other programs more accurate than the original one and which are completely different from each other. The results obtained show by how we improve the accuracy of each of them.

RÉSUMÉ CHAPITRE 3

Dans ce chapitre, nous rappelons quelques concepts de base utiles à la compréhension de ce manuscrit comme l'arithmétique des ordinateurs et l'interprétation abstraite et nous présentons ensuite des travaux connexes aux nôtres concernant la transformation de programmes. Nous commençons par rappeler les notions de base sur l'arithmétique des ordinateurs tel que l'arithmétique des intervalles et l'arithmétique à virgule flottante et nous détaillons le standard IEEE754. Nous décrivons aussi notre langage basé sur la forme SSA (Static Single Assignment). Cette dernière permet d'éviter qu'une même variable soit écrite en différents points de programme. Un point important consiste à définir la sémantique opérationnelle de nos programmes dont on aura besoin plus loin dans ce manuscrit pour prouver la correction de notre transformation de programmes.

Ensuite, nous présentons la théorie des domaines, les notions d'ordre partial et complet, les treillis, les fonctions monotones, les fonctions continues ainsi que les théorèmes de Kleene et de Tarski, les méthodes d'analyse statique par interprétation abstraite et nous expliquons comment les erreurs d'arrondi se propagent durant l'évaluation des expressions arithmétiques. Par la suite, nous présentons très brièvement quelques outils les plus en rapport avec nos travaux comme Fluctuat, Astrée, FPTaylor, ROSA et Herbie dont le point commun est d'estimer ou d'améliorer la précision numérique des calculs des codes qu'ils analysent.

Dans ses travaux de recherche, Arnaut Ioualalen [56] s'est intéressé à transformer automatiquement les expressions arithmétiques, plus précisément, les opérations élémentaires telles que l'addition et la multiplication. En introduisant une nouvelle représentation intermédiaire nommée APEG (Abstract Program Equivalent Graph), A. Ioualalen représente dans une structure polynomiale un nombre exponentiel d'expressions arithmétiques équivalentes. Un APEG est décrit par

des classes d'équivalences contenant les opérations de l'expression arithmétique à transformer, et aussi des boites contenant une opération arithmétique avec plusieurs expressions signifiant ainsi les différentes façons de combiner ces expressions avec l'opération en question. Pour construire un APEG, son approche définit deux algorithmes : de propagation et d'expansion.

L'objectif de cette thèse est d'aller une étape plus loin que les expressions arithmétique en transformant automatiquement des morceaux de code. Cette transformation concerne les affectations, les boucles, les structures de contrôle et les fonctions.

CHAPITRE 3

BACKGROUND AND RELATED WORK

Contents

3.1 Introduction **22**

3.2 Computer Arithmetic **22**

3.2.1 Floating-Point Arithmetic 22

3.2.2 Interval Arithmetic 26

3.3 Languages **27**

3.3.1 SSA Form 27

3.3.2 Formal Semantics 28

3.4 Static Analysis **31**

3.4.1 Domain Theory 31

3.4.2 Abstract Interpretation 34

3.4.3 Errors Computation 36

3.5 Tools **39**

3.6 Transformation of Expressions **41**

3.7 Conclusion **44**

3.1 Introduction

As mentioned in the general introduction, we are interesting in improving the numerical accuracy of programs by automatic program transformation. Our transformation is based on a static analysis technique, abstract interpretation. Basically, our source programs perform floating-point computations. Therefore, in this chapter, we describe how to transform the arithmetic expressions by using the Abstract Program Expression Graph introduced in the PhD thesis of A. Ioualalen [56]. We start by recalling, in Section 3.2, the floating-point and interval arithmetic. Then, in Section 3.3, we introduce some elements of programing language theory : the SSA form and an operational semantics. Section 3.4 briefly describes the abstract interpretation framework and how to compute the round-off errors in programs. Section 3.5 describes related tools and section 3.6 discusses former work of concerning the transformation of arithmetic expressions. We conclude in Section 3.7.

3.2 Computer Arithmetic

In this section, we present the computer arithmetic used in the next chapters of this thesis. We start with the arithmetic of floating-point numbers used by computers to approximate real numbers. The popularity of this arithmetic is due to its ability to represent in the same format small and large values. For example, numbers from 10^{-45} to 10^{38} are representable in single precision. Next, in Section 3.2.2, we introduce the interval arithmetic.

3.2.1 Floating-Point Arithmetic

The floating-point arithmetic [10, 46, 76, 75] is the most used arithmetic to approximate real numbers. However, with this arithmetic, many information is lost because of the round-off errors which introduce in most cases significant differences compared to the exact results.

A real number x is represented in the floating-point arithmetic in base $b \geq 2$ by :

- A sign $s \in \{0,1\}$ with 0 : positive and 1 : negative,
- A mantissa m, with $0 \leq m_i < b$, written on p digits,
- An exponent $e \in [e_{min}, e_{max}]$.

We have :

$$x = (-1)^s \cdot m_x \cdot b^e \tag{3.1}$$

with $m_x = m_0.m_1.m_2 \ldots m_{p-1}$ and $m_i \in \{0, \ldots, b-1\}$ for $0 \le i \le p-1$ with p the precision, $p \ge 1$.

The representation of numbers in the floating-point arithmetic is not unique for example, if we want to represent the number 3.215 in base $b = 10$ with a precision $p = 4$, we may have :

$$0.3215 \cdot 10^1 \quad or \quad 3215.0 \cdot 10^{-3} \ .$$

Until the $80's$, every constructor disposed of its own floating-point arithmetic implementation according to the base, the exponents or the size of the mantissa, etc. Then for the same program, we could have different results on different architectures. To deal with this difficulty, the floating-point arithmetic implementations had a strong need to be more homogeneous. To do this, one has to

- define the format of data and their implementation on machine ;
- define the behavior and the accuracy of each elementary operation ;
- define the special values, the round-off modes, and the management of the exceptions.

All these points contributed to the definition of the `IEEE754` Standard [10].

IEEE754 standard

The `IEEE754` standard [10] consists in the scientific notation adapted to represent numbers in machines, as in Equation (3.1), and the semantics of the elementary operations for floating-point arithmetic [10, 76]. It allows one to represent either very small or very large numbers while maintaining desirable properties of arithmetic operations like execution speed. The floating-point arithmetic is implemented in most of modern general-purpose processors. The standard defines a floating-point number by a triple made of of sign, mantissa and exponent. Note that it is possible to define a floating-point number in several binary formats for example with sizes of 32, 64 and 128 bits by varying only the mantissa and the exponent.

The `IEEE754` Standard consists in representing any number by a mantissa m, a precision p and an exponent e in a given base b. As an example, if we take the decimal number 0.1, in base $b = 2$ with precision $p = 24$, it cannot be represented exactly, that is why we approximate it by $1.10011001100110011001101 \times 2^{-4}$.

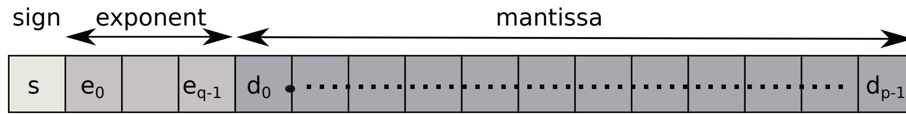


FIGURE 3-1 – *Binary Representation of Floating-Point Numbers.*

In general, the format of a floating-point number x in base b is denoted as in Equation (3.1). This is illustrated in Figure 3-1.

A float-point number x is normalized if and only if $d_0 \neq 0$. Thanks to normalization, each floating-point number has only one representation. The `IEEE754` Standard specifies several formats for floating-point numbers by providing specific values for p, b, e_{min} and e_{max} as described in Table 3.1. Depending on special values of the mantissa and the exponent, the standard defines the denormalized numbers, infinites and the NaN (Not a Number). These special values are defined in Table 3.2.

The `IEEE754` Standard also defines some rounding modes, towards $+\infty$, $-\infty$, 0 and to the nearest. Let $\mathbb{F}$ be the set of floating-point numbers used by the program for the current operation (*e.g.* Binary32 or Binary64 a.k.a single or double precision) and let us write $\uparrow_{+\infty}$, $\uparrow_{-\infty}$, $\uparrow_0$ and $\uparrow_\sim$ for the rounding functions, the `IEEE754` Standard defines the semantics of the elementary operations, with $\uparrow_r$: $\mathbb{R} \rightarrow \mathbb{F}$, by

$$x \circledast_r y = \uparrow_r (x * y) \quad , \tag{3.2}$$

where $\circledast_r$ denotes a floating-point operation $+$, $-$, $\times$ or $\div$ computed using the rounding mode r and $*$ denotes an exact operation. Because of the round-off errors, the results of the computations are not exact. However as explained in Section 3.4.3, the error on the result of the elementary operations is bounded thanks to the `IEEE754` Standard which guaranties the roundoff errors. We also use the function $\downarrow_r$: $\mathbb{R} \rightarrow \mathbb{R}$ which returns the round-off errors. We have

$$\downarrow_r (x) = x - \uparrow_r (x) \quad . \tag{3.3}$$

Format	Name	p	e	e_{min}	e_{max}
Binary16	Half precision	11	5	-14	$+15$
Binary32	Single precision	24	8	-126	$+127$
Binary64	Double precision	53	11	-1122	$+1223$
Binary128	Quadratic precision	113	15	-16382	$+16383$

TABLE 3.1 – *Basic `IEEE754` floating-point formats.*

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

TABLE 3.2 – *Special values of the IEEE754 floating-point formats in simple precision.*

Our transformation techniques are independent of the selected rounding mode and, in this article, for the sake of simplicity, we assume that all the floating-point computations are done by using the rounding mode to the nearest. We write $\uparrow$ and $\downarrow$ instead of $\uparrow_r$ and $\downarrow_r$ whenever the rounding mode r does not matter.

Round-off Errors

In order to represent a real number using a floating-point format, we approximate and represent it with a finite number of bits. Consequently, if we choose to store the result of some computation in a fixed number of bits, we have some quantities which cannot be represented exactly. This is why the result of a floating-point computation must often be rounded to fit into this finite representation.

Example 3.2.1

For example, if we want to approximate the number $1/3$ by keeping only 5 digits of precision, we write 0.33333in base 10. The round-off error of this approximation is equal to $0.0000033\ldots33$, obtained by subtracting 0.33333 from $1/3$.

■

In some context and depending on the values of these round-off errors, the result may be completely false. The consequences are generally small but it can be catastrophic in special cases, as described in Section 2.1.

3.2.2 Interval Arithmetic

Ramon Moore introduced the interval arithmetic for the first time in 1962 in [72]. As its name indicates, the interval arithmetic [73, 74, 79] uses intervals including the desired values instead of real numbers. For example, when taking a value x with an error ε on it, we represent it by the interval $[x-\varepsilon, x+\varepsilon]$. If we are interested in computing with intervals, the elementary operations for two intervals $\mathbf{x} = [\underline{x}, \overline{x}]$ and $\mathbf{y} = [\underline{y}, \overline{y}]$, with $\underline{x}$ (resp. $\underline{y}$) the lower bound of the interval $\mathbf{x}$ (resp. $\mathbf{y}$) and $\overline{x}$ (resp. $\overline{y}$) the upper bound of the interval $\mathbf{x}$ (resp. $\mathbf{y}$), where $\underline{x}$, $\overline{x}$, $\underline{y}$ and $\overline{y}$ are floating-point numbers, are :

$$
\begin{aligned}
&\mathbf{x}+\mathbf{y} = [\uparrow_{-\infty}(\underline{x}+\underline{y}), \uparrow_{+\infty}(\overline{x}+\overline{y})],\\
&\mathbf{x}-\mathbf{y} = [\uparrow_{-\infty}(\underline{x}-\overline{y}), \uparrow_{+\infty}(\overline{x}-\underline{y})],\\
&\mathbf{x}\times\mathbf{y} = [min\{\uparrow_{-\infty}(\underline{x}\underline{y}), \uparrow_{-\infty}(\underline{x}\overline{y}), \uparrow_{-\infty}(\overline{x}\underline{y}), \uparrow_{-\infty}(\overline{x}\overline{y})\}, max\{\uparrow_{+\infty}(\underline{x}\underline{y}), \uparrow_{+\infty}(\underline{x}\overline{y}), \uparrow_{+\infty}(\overline{x}\underline{y}), \uparrow_{+\infty}(\overline{x}\overline{y})\}],\\
&\mathbf{x}^2 = [min(\uparrow_{-\infty}(\underline{x}^2), \uparrow_{-\infty}(\overline{x}^2)), max(\uparrow_{+\infty}(\underline{x}^2), \uparrow_{+\infty}(\overline{x}^2))]\ if\ 0 \notin [\underline{x}, \overline{x}],\ [0, max(\uparrow_{+\infty}(\underline{x}^2), \uparrow_{+\infty}(\overline{x}^2))]\ else,\\
&1/[\underline{y}, \overline{y}] = [min(\uparrow_{-\infty}(1/\underline{y}), \uparrow_{-\infty}(1/\overline{y})), max(\uparrow_{+\infty}(1/\underline{y}), \uparrow_{+\infty}(1/\overline{y}))] \quad if\ 0 \notin [\underline{y}, \overline{y}],\\
&\mathbf{x}/\mathbf{y} = [\mathbf{x}\times 1/\mathbf{y}] \quad if\ 0 \notin [\underline{y}, \overline{y}].
\end{aligned}
\tag{3.4}
$$

Example 3.2.2

To make clear how the interval arithmetic works, we illustrate it with the computation of following polynomial expression $(2\times\mathbf{x}^2)+\mathbf{x}-3$, with $\mathbf{x} = [-2, 1]$.

$$
\begin{aligned}
2\times x^2 + x - 3 &= 2\times[-2,1]^2 + [-2,1] - 3\\
&= 2\times[0,4] + [-2,1] - 3\\
&= [0,8] + [-2,1] - 3\\
&= [-5,6]
\end{aligned}
\tag{3.5}
$$

3.3 Languages

3.3.1 SSA Form

The SSA (Static Single Assignment) form [27] allows only one static assignment of each variable in the program. It avoids conflicts arising when reading and writing the same variable many times. A problem arises when the value assigned to a variable may have been computed in several places depending on the control flow. This is why the SSA form introduces Φ-nodes. We attach Φ nodes to conditional and while statements to denote their sets of Φ-nodes. A Φ-node $\Phi(id, id_1, id_2)$ is understood as an assignment of the form $id = \Phi(id_1, id_2)$ where $\Phi(id_1, id_2) = id_1$ or $\Phi(id_1, id_2) = id_2$ depending on the control flow. The construction of Φ-nodes is classical and is left to the reader [11, 27]. We only illustrate it by means of the Example 3.3.1.

Example 3.3.1

Initially, we give the original program in Listing 3.6.

$$\begin{array}{l} x = 2\ ; \\ \textsf{if}\ (x > 1)\ \textsf{then} \\ \quad x = x \times 2\ ; \\ \textsf{else} \\ \quad x = x \div 2\ ; \\ z = x \end{array} \tag{3.6}$$

The SSA form of the program given in Listing 3.6 is

$$\begin{array}{l} x_1 = 2\ ; \\ \textsf{if}\ (x_1 > 1)\ \textsf{then} \\ \quad x_2 = x_1 \times 2\ ; \\ \textsf{else} \\ \quad x_3 = x_1 \div 2\ ; \\ \Phi(x_4, x_2, x_3)\ ; \\ z = x_4 \end{array} \tag{3.7}$$

In the new program, given in Listing 3.7, the variables are assigned only once. Then x has been split into x_1, x_2, x_3 and x_4. The Φ-node $\Phi(x_4, x_2, x_3)$ states that x_4 is assigned to x_2 or x_3 depending on the branch taken by the control flow.

3.3.2 Formal Semantics

The formal semantics of programs allows one to describe mathematically the different behaviors of a program. It makes it possible to predict the result of a program assuring that the compiler respects the semantics. Among the various formal semantics that make it possible to describe the behavior of a program, we can cite :

- The operational semantics : The execution of a program is defined by a sequence of computational steps,
- The denotational semantics : The meaning of a program is translated into a function between the program states.
- The axiomatic semantics : The program instructions are described as logical statements or predicates assertions among the variables of the program,

Basically, these different semantics are equivalents (we can refer the reader to [89] for more details).

Operational Semantics

In the following, we introduce an usual SOS operational semantics for our `while` language. This semantics is used in Chapter 4 to prove the correctness of our transformation. This semantics is standard but we need to explicit it in order to achieve the proof of correctness of our transformation in Section 4.4.3.

The language considered in our study is given by Equations (3.8) and (3.9). Let Expr and BExpr be the set of arithmetic and boolean expressions respectively. We have

$$\begin{array}{rcl} \mathsf{Expr} \ni & e & ::= \; id \mid cst \mid e+e \mid e-e \mid e\times e \mid e\div e \;. \\ \mathsf{BExpr} \ni & b & ::= \; \mathtt{true} \mid \mathtt{false} \mid b\wedge b \mid b\vee b \mid \neg b \mid e=e \mid e<e \mid e>e \;. \end{array} \tag{3.8}$$

Where id is an identifier in a finite set $\mathscr{V}$ and cst is a constant in val, the domain of values. Depending on the context, val is the set of real numbers $\mathbb{R}$, the set of floating-point numbers $\mathbb{F}$ of a given precision (see Section 3.2.1) or the set of floating-point numbers with errors $\mathbb{E}$ introduced in Section 3.4.3. Apart the domain of values, the semantics is independent of $\mathbb{R}$, $\mathbb{F}$ or $\mathbb{E}$.

The syntax of commands is given in Equation (3.9). It corresponds to the core of an impera-

tive language.

$$\mathsf{Cmd} \ni c ::= id = e \mid c_1; c_2 \mid \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}_\Phi\ e\ \mathsf{do}\ c \mid \mathtt{nop}\ . \tag{3.9}$$

Our command language admits assignments $id = e$ where e is defined by Equation (3.8), sequences of instructions, a conditional $\mathsf{if}_\Phi\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$, a loop statement $\mathsf{while}_\Phi\ b\ \mathsf{do}\ c$ where b is defined by Equation (3.8) and the void operation $\mathtt{nop}$. We write $\equiv$ the syntactic equivalence, in other words, $x \equiv c$ means that x is syntactically the command c. We assume that our programs are written in SSA form [27] as explained in Section 3.3.1.

Let $\mathscr{V}$ be a finite set of identifiers and let $\sigma \in \mathsf{Mem}$ be an environment that map variables of $\mathscr{V}$ to values

$$\sigma : \mathscr{V} \to \mathsf{val}\ , \tag{3.10}$$

In Figure 3-2, we define

$$\to_e : \mathsf{Expr} \to \mathsf{val}\ , \tag{3.11}$$

and

$$\to_b : \mathsf{BExpr} \to \{\mathtt{true}, \mathtt{false}\}\ , \tag{3.12}$$

the transition functions for the evaluation of arithmetic (Expr) and boolean (BExpr) expressions.

The operational semantics uses states which are defined by a pair of command (Cmd) and memory (Mem). We assume that

$$\mathsf{State} = \{\langle c, \sigma\rangle : c \in \mathsf{Cmd}, \sigma \in \mathsf{Mem}\}\ . \tag{3.13}$$

The operational semantics maps State to State :

$$\to : \left| \begin{array}{l} \mathsf{State} \to \mathsf{State} \\ \langle c, \sigma\rangle \mapsto \langle c', \sigma'\rangle \end{array} \right. . \tag{3.14}$$

The standard semantics of programs is given in Figure 3-2 (rules $(R8)$ to $(R15)$). The rules for assignments rely on the semantics $\to_e$ of expressions also given in Figure 3-2 (rules $(R1)$ to $(R4)$). Recall that, as explained in Section 3.2.1, our technique is independent of the rounding mode used in the floating-point arithmetic and, in Figure 3-2, we omit to mention it.

For a sequence of commands (rules $(R10)$ and $(R11)$), we execute the first command in the initial environment, and then we execute the second in the resulting environment. The next

$$\frac{\sigma(x) = cst}{\langle x, \sigma\rangle \rightarrow_e \langle cst, \sigma\rangle} \quad (R1)$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow_e \langle e'_1, \sigma\rangle}{\langle e_1 \odot e_2, \sigma\rangle \rightarrow_e \langle e'_1 \odot e_2, \sigma\rangle} \quad with \quad \odot \in \{+,-,\times,\div\} \quad (R2)$$

$$\frac{\langle e_2, \sigma\rangle \rightarrow_e \langle e'_2, \sigma\rangle}{\langle cst_1 \odot e_2, \sigma\rangle \rightarrow_e \langle cst_1 \odot e'_2, \sigma\rangle} \quad with \quad \odot \in \{+,-,\times,\div\} \quad (R3)$$

$$\frac{cst = cst_1 \odot cst_2}{\langle cst_1 \odot cst_2, \sigma\rangle \rightarrow_e \langle cst, \sigma\rangle} \quad with \quad \odot \in \{+,-,\times,\div\} \quad (R4)$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow_e \langle e'_1, \sigma\rangle}{\langle e_1 \mathscr{R}\, e_2, \sigma\rangle \rightarrow_b \langle e'_1 \mathscr{R}\, e_2, \sigma\rangle} \quad with \quad \mathscr{R} \in \{=,<,>,\leq,\geq\} \quad (R5)$$

$$\frac{\langle e_2, \sigma\rangle \rightarrow_b \langle e'_2, \sigma\rangle}{\langle cst_1 \mathscr{R}\, e_2, \sigma\rangle \rightarrow_b \langle cst_1 \mathscr{R}\, e'_2, \sigma\rangle} \quad with \quad \mathscr{R} \in \{=,<,>,\leq,\geq\} \quad (R6)$$

$$\frac{cst = cst_1 \mathscr{R}\, cst_2}{\langle cst_1 \mathscr{R}\, cst_2, \sigma\rangle \rightarrow_b \langle cst, \sigma\rangle} \quad with \quad \mathscr{R} \in \{=,<,>,\leq,\geq\} \quad (R7)$$

$$\frac{\langle e, \sigma\rangle \rightarrow \langle e', \sigma\rangle}{\langle id = e, \sigma\rangle \rightarrow \langle id = e', \sigma\rangle} \quad (R8)$$

$$\frac{\sigma' = \sigma[id \mapsto cst]}{\langle id = cst, \sigma\rangle \rightarrow \langle nop, \sigma'\rangle} \quad (R9)$$

$$\frac{\langle c_1, \sigma\rangle \rightarrow \langle c'_1, \sigma'\rangle}{\langle c_1; c_2, \sigma\rangle \rightarrow \langle c'_1; c_2, \sigma'\rangle} \quad (R10)$$

$$\frac{}{\langle \mathsf{nop}; c, \sigma\rangle \rightarrow \langle c, \sigma\rangle} \quad (R11)$$

$$\frac{}{\langle \mathsf{if}_\Phi\ \mathsf{true\ then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \rightarrow \langle c_1, \sigma\rangle} \quad (R12)$$

$$\frac{}{\langle \mathsf{if}_\Phi\ \mathsf{false\ then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \rightarrow \langle c_2, \sigma\rangle} \quad (R13)$$

$$\frac{\langle b, \sigma\rangle \rightarrow_b \langle b', \sigma\rangle}{\langle \mathsf{if}_\Phi\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \rightarrow \langle \mathsf{if}_\Phi\ b'\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle} \quad (R14)$$

$$\frac{}{\langle \mathsf{while}_\Phi\ b\ \mathsf{do}\ c, \sigma\rangle \rightarrow \langle \mathsf{if}_\Phi\ b\ \mathsf{then}\ c;\ \mathsf{while}_\Phi\ b\ \mathsf{do}\ c\ \mathsf{else\ nop}, \sigma\rangle} \quad (R15)$$

FIGURE 3-2 – *Small-step operational semantics of programs.*

kind of rules is for conditionals and uses the semantics $\rightarrow_b$ also given in Figure 3-2 (rules $(R5)$ to $(R7)$). If the condition is evaluated to true (Rule $(R12)$), we take the semantics of the then branch, otherwise, we take the semantics of the else branch (Rule $(R13)$). The rules for while loops execute the body c if the condition is true and then ran again the loop. They return the environment unchanged if the condition is false (Rule $(R15)$).

3.4 Static Analysis

Static analysis allows one to validate a program for a large number of input values, for example intervals inputs. Several static analyzers like ASTREE [25] use the interval arithmetic and other (relational) domains. Generally, the variability of the intervals used to define the inputs of a program to be validated by static analysis is very large, and the results obtained are over-approximated in order to validate the accuracy of computations. To reduce the inaccuracy of computations resulting from the static analysis, relational domains are used [42, 51, 68, 69, 70]. Most relational domains compute the ranges of the variables with more or less precision. We may cite the domain of polyhedra [81], octagons [69, 70], digital filters [42], etc. In our case, we are interested in domains which compute the numerical accuracy. A non-relational domain is introduced in Section 3.4.3. Relational domains are described in [51, 65].

3.4.1 Domain Theory

Abstract interpretation is a widely used framework for static analysis. To introduce this notion, let us start by giving an overview concerning the domain theory. In this section, we introduce the notions of partially ordered sets and fixed point theorems.

Definition 3.4.1 (Partial Order)**.** A partial order consists in a set P equipped with a binary relation, denoted $\sqsubseteq$, which is :

- Reflexive, $\forall x \in P, x \sqsubseteq x$;
- Transitive, $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$;
- Antisymmetric, $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$;

The partial order is denoted by $(P, \sqsubseteq)$.

■

Example 3.4.1

The set of intervals with real bounds is a partial order for the relation $\subseteq$ for inclusion.

Definition 3.4.2 (Upper Bound)**.** Let us consider that $(P, \sqsubseteq)$ is a partial order and that $Q \subseteq P$. We say that an element $p \in P$ is an upper bound of Q if and only if :

$$\forall x \in Q,\ x \sqsubseteq p \tag{3.15}$$

p is the least upper bound of Q, denoted by $p = \sqcup Q$, if any upper bound q of Q satisfies $q \sqsubseteq p$.

■

Definition 3.4.3 (Lower Bound)**.** Let us consider that $(P, \sqsubseteq)$ is a partial order and that $Q \subseteq P$. We say that an element $p \in P$ is a lower bound of Q if and only if :

$$\forall x \in Q,\ p \sqsubseteq x \tag{3.16}$$

p is the greatest lower bound of Q, denoted by $p = \sqcap Q$, if any lower bound q of Q satisfies $p \sqsubseteq q$.

■

Example 3.4.2

The intervals of $\mathbb{R}$ have a lower bound $\emptyset$ but no upper bound.

Definition 3.4.4 (Increasing Chain)**.** Let us consider that $(P, \sqsubseteq)$ is a partial order. An increasing chain is a subset Q of P equipped with a lower bound and such that :

$$\forall x, y \in Q, x \sqsubseteq y \ \vee \ y \sqsubseteq x \tag{3.17}$$

■

Definition 3.4.5 (Complete Partial Order)**.** A partial order $(P, \sqsubseteq)$ is a complete partial order, (CPO), if and only if any non empty increasing chain Q has an upper bound.

■

Definition 3.4.6 (Lattice)**.** A lattice is a partial order $(P, \sqsubseteq)$ such that any finite set $Q \in P$ has an upper bound and a lower bound.

■

Definition 3.4.7 (Complete Lattice)**.** A complete lattice is a partial order $(P, \sqsubseteq)$ such that any set $Q \in P$ (finite or not) has an upper bound and a lower bound.

■

Example 3.4.3

To illustrate the definition of a complete lattice, let us take the following example. Let us consider a partial order $(Z, \sqsubseteq)$ with $Z = \{\top, \bot, 0, \mathbb{Z}^{-}, \mathbb{Z}^{+}, \mathbb{Z}^{*}, \mathbb{Z}^{*-}, \mathbb{Z}^{*+}\}$. The set Z is ordered as illustrated in Figure 3-3. In Figure 3-3, the dotted line linking $\mathbb{Z}^{-}$ and $\mathbb{Z}^{*-}$ is interpreted as follows : $\mathbb{Z}^{*-} \sqsubseteq \mathbb{Z}^{-}$.

■

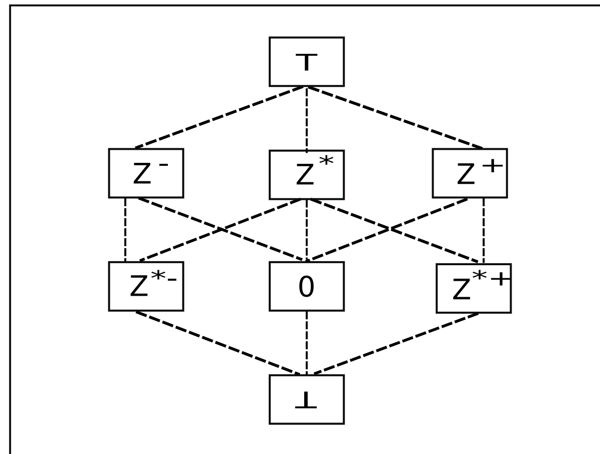


FIGURE 3-3 – *Complete lattice presented in the example 3.4.1.*

Definition 3.4.8 (Monotonic Function)**.** Let us consider two CPO $(P, \sqsubseteq_P)$ and $(Q, \sqsubseteq_Q)$. We say that a function $f : P \to Q$ is monotone if and only if :

$$\forall x, y \in P, x \sqsubseteq_P y \Rightarrow f(x) \sqsubseteq_Q f(y). \tag{3.18}$$

■

Definition 3.4.9 (Continuous Function)**.** Let us consider two CPO $(P, \sqsubseteq_P)$ and $(Q, \sqsubseteq_Q)$. We say that a function $f : P \to Q$ is continuous if and only if it is monotone and if for any increasing chain X of P :

$$\sqcup_Q \{f(x) : x \in X\} = f(\sqcup_P X). \tag{3.19}$$

In other words, a continuous function consists in a monotone function that preserves the upper bounds of the increasing chains.

■

Definition 3.4.10 (Fixed Point). Let us consider a partial order $(P, \sqsubseteq)$ and $f : P \to P$ a function on P. We say that x is a fixed point of f if and only if $f(x) = x$. We say that x is the smallest fixed point of f (when existing) if and only if x is a fixed point of f and if $\forall y \in P, f(y) = y \Rightarrow x \sqsubseteq y$.

■

Remark. We denote by lfp(f) the smallest fixed point of f and by gfp(f) the greatest fixed point of f, when they exist.

Theorem 3.4.1 (Tarski's Theorem, [84]). *Assume that P be a complete lattice and let $f : P \to P$ be a monotonic function. The function f admits a smallest lfp(f) and a greatest fixed point gfp(f).*

■

Theorem 3.4.2 (Kleene's Theorem, [17]). *Assume that $f : P \to P$ is a continuous function on a complete lattice P such that the smallest (resp. greatest) element is $\perp$ (resp. $\top$). Then we have :*

$$lfp(f) = \sqcup\{f^i(\perp) \ : \ i \in \mathbb{N}\} \ , \qquad gfp(f) = \sqcap\{f^i(\top) \ : \ i \in \mathbb{N}\} \ .$$

■

3.4.2 Abstract Interpretation

The main goal of abstract interpretation [23, 24] is to approximate in a sound manner the semantics of a programming language. It can be used to prove properties of programs without executing them. Abstract interpretation allows us to compute an over-approximation of the concrete semantic by the abstract semantic. It is introduced to cope with the problems representing an infinite set of cases and also the computation of the fixed-point when computing the semantic of loops that possibly requires an infinite number of iterations of Kleene's theorem.

The abstract semantics uses an abstract domain which is a set of values $\varepsilon^\sharp$ representable on machine. A concretization function $\gamma : \varepsilon^\sharp \to \varepsilon$ associates for every element $\sigma^\sharp \in \varepsilon^\sharp$ a set of concrete values, possibly not finite and an abstraction function $\alpha : \varepsilon \to \varepsilon^\sharp$ does the converse operation.

Example 3.4.4

As an example, let us consider $Sign = \{\bot, -, +, 0, \dot{-}, \dot{+}, \varnothing, \top\}$, with :

The abstract domain of signs with a concretization function $\gamma : Sign \to \mathrm{P}(\mathbb{Z})$ is defined by :

$$\gamma(\bot) = \varnothing, \quad \gamma(-) = \{n \in \mathbb{Z} : n < 0\}, \quad \gamma(+) = \{n \in \mathbb{Z} : n > 0\}, \quad \gamma(0) = \{0\},$$

$$\gamma(\dot{-}) = \{n \in \mathbb{Z} : n \leq 0\}, \quad \gamma(\dot{+}) = \{n \in \mathbb{Z} : n \geq 0\}, \quad \gamma(\varnothing) = \{n \in \mathbb{Z} : n \neq 0\}, \quad \gamma(\top) = \mathbb{Z}$$

In this example, we have that the abstract element $\dot{+}$ represents all the integers greater than or equal to zero while $+$ consists in only the positive integers. We have that $+$ is more precise than $\dot{+}$ because it give us match more informations on the concrete values really represented.

Among the properties of the abstract interpretation, we find :

- The abstraction function α and the concretization function γ are increasing functions.
- The abstraction semantics $t = \alpha(T)$ is an approximation of the concrete one.
- The abstraction semantics $t = \alpha(T)$ includes the initial concrete semantics T,

$$T \subseteq (\gamma(\alpha(T)) \ .$$

- The concretization semantics does not loose any information when abstracting it.

Many tools based-on the abstract interpretation are used by industries like ASTREE [25], Fluctuat [38, 48], AiT [7], etc (see Section 3.5).

We introduce now the functions $[\![.]\!]$ and $[\![.]\!]^\sharp$ which take concrete (resp. abstract) environment in order to compute the concrete (resp. abstract) semantics. For any command c defined in Equation (3.9) and any environment δ, the concrete semantics is defined by

$$[\![c]\!]\sigma = \sigma' \quad \Longleftrightarrow \quad \langle c, \sigma \rangle \to^* \langle c, \sigma' \rangle$$

where $\to$ is the operational semantics introduced in Figure 3-2 and $\to^*$ the transitive closure of $\to$. The abstract semantics $[\![.]\!]^\sharp$ must over-approximate the concrete semantics $[\![.]\!]$ as given in Equation (3.20).

$$\forall \sigma^\sharp \in \varepsilon^\sharp,\ [\![c]\!]\big(\gamma(\sigma^\sharp)\big) \ \subseteq \ \gamma\big([\![c]\!]^\sharp(\sigma^\sharp)\big) \ . \tag{3.20}$$

For efficiency reasons, the computation of the abstract semantics must be less precise than the

concrete one, in other words, if we apply the abstract function and then the concretion, we obtain a larger set than if we do the concretization first and then compute the concrete semantics.

Example 3.4.5

In this example, we highlight the computation of the abstract semantics of the conditional statement `if` that is defined as follow :

$$[\![\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]^\sharp(\sigma^\sharp) = [\![c_1]\!]^\sharp(\sigma) \cup^\sharp [\![c_2]\!]^\sharp(\sigma)\ \ .$$

The computation of the abstract semantics of `if` consists in computing separately the abstract semantics of both branches `then` and `else` and then join the two results. The property of the Equation (3.20) are satisfied. Nevertheless we introduce an over-approximation since in the concrete only one branch of the conditional is ran.

■

3.4.3 Errors Computation

We present now the computation of errors during the evaluation of arithmetic expressions [65]. First, we introduce the concrete semantics and then the abstract semantics. In the concrete semantics, we use values which are pairs $(x,\mu) \in \mathbb{F} \times \mathbb{R} = \mathbb{E}$ where x denotes the floating point number used by the machine and μ denotes the exact error attached to $\mathbb{F}$, *i.e.*, the exact difference between the real and floating-point numbers as defined in Equation (3.3). For example, the real number $\frac{1}{3}$ is represented by the value $v = (\uparrow_\sim \left(\frac{1}{3}\right), \downarrow_\sim \left(\frac{1}{3}\right)) = (0.333333, (\frac{1}{3} - 0.333333))$. The semantics of the elementary operations on $\mathbb{E}$ is defined in [65].

Our transformation technique, introduced in Chapter 4, is independent of the selected rounding mode and for the sake of simplicity, we assume that all the floating-point computations are done by using the rounding mode to the nearest. We write $\uparrow$ and $\downarrow$ instead of $\uparrow_r$ and $\downarrow_r$ whenever the rounding mode r does not matter.

We do not wish to optimize our programs for a given value of the inputs but for all the possible values of the inputs. Consequently, our tool uses an abstract semantics [23] based on $\mathbb{E}$. The abstract values are represented by a pair of intervals. The first interval contains the range of the floating-point values of the program and the second one contains the range of the errors obtained by subtracting the floating-point values from the exact ones. In the abstract value denoted by $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, we have $x^\sharp$ the interval corresponding to the range of the values and $\mu^\sharp$ the interval

of errors on $x^\sharp$. This value abstracts a set of concrete values $\{(x,\mu)\ :\ x \in x^\sharp \text{ and } \mu \in \mu^\sharp\}$ by intervals in a component-wise way. We introduce now the semantics of arithmetic expressions on $\mathbb{E}^\sharp$. We approximate an interval $x^\sharp$ with real bounds by an interval based on floating-point bounds, denoted by $\uparrow^\sharp (x^\sharp)$. Here bounds are rounded to the nearest (see Equation (3.21)).

$$\uparrow^\sharp ([\underline{x},\overline{x}]) = [\uparrow(\underline{x}), \uparrow(\overline{x})] \ . \tag{3.21}$$

In the other direction, we have the function $\downarrow^\sharp$ that abstracts the concrete function $\downarrow$. It over-approximates the set of exact values of the error $\downarrow(x) = x - \uparrow(x)$. Every error associated to $x \in [\underline{x},\overline{x}]$ is included in $\downarrow^\sharp ([\underline{x},\overline{x}])$. We also have for a rounding mode to the nearest

$$\downarrow^\sharp ([\underline{x},\overline{x}]) = [-y, y] \quad \text{with} \quad y = \frac{1}{2}\text{ulp}\big(\max(|\underline{x}|,|\overline{x}|)\big) \ . \tag{3.22}$$

Formally in Equation (3.22), the *unit in the last place*, denoted by ulp(x), is the weight of the least significant digit of the floating-point number x. Equations (3.23) to (3.25) give the semantics of the addition, subtraction and multiplication over $\mathbb{E}^\sharp$, for other operations see [65]. If we sum two floating-point numbers, we must add the errors on the operands to the error produced by the round-off of the result. If we subtract two floating-point numbers, we must subtract errors on the operands and add the error produced by the round-off of the result. When multiplying two floating-point numbers, the semantics is given by the development of $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$.

$$(x_1^\sharp,\mu_1^\sharp) + (x_2^\sharp,\mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp + x_2^\sharp),\ \ \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp)\big) \ , \tag{3.23}$$

$$(x_1^\sharp,\mu_1^\sharp) - (x_2^\sharp,\mu_2^\sharp) = \big(\uparrow^\sharp (x_1^\sharp - x_2^\sharp),\ \ \mu_1^\sharp - \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp - x_2^\sharp)\big) \ , \tag{3.24}$$

$$(x_1^\sharp,\mu_1^\sharp) \times (x_2^\sharp,\mu_2{,}^\sharp) = \big(\uparrow^\sharp (x_1^\sharp \times x_2^\sharp),\ \ x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp)\big) \ . \tag{3.25}$$

Example 3.4.6

In order to illustrate the semantics of the elementary operations given in equations (3.23) to (3.25), we consider the product of two floating-point numbers x_1 and x_2 associated to an initial errors respectively μ_1 and μ_2. In our example, we have chosen

$$([\underline{x_1},\overline{x_1}],[\underline{\mu_1},\overline{\mu_1}]) = \big([3.141, 3.142],[0.00059, 0.000592]\big)$$

and

$$([\underline{x_2},\overline{x_2}],[\underline{\mu_2},\overline{\mu_2}]) = \big([99.98, 99.99],[0.09, 0.1]\big) \ .$$

The computation is processed as following :

- First, we multiply x_1 by x_2. The product gives :

$$[3.141, 3.142] \times [99.98, 99.99] = [314.03718, 314.16858] \quad .$$

  For the sake of simplicity, we use in this example a simplified system of floating-point numbers whose mantissas are made of four decimal digits. So, we keep only four digits, the result is $\uparrow^\sharp (x_1^\sharp \times x_2^\sharp)$ = $[314.0, 314.2]$,

- Next, the global error $[0.3731513, 0.41063328]$ is generated by the product and obtained by the sum of :
  - The first floating-point interval x_1 multiplied by the error μ_2 gives $[0.28269, 0.3142]$,
  - The second interval x_2 multiplied by the error μ_1 results $[0.0589882, 0.05919408]$,
  - The error μ_1 multiplied by the error μ_2 is equals to $[0.0000531, 0.0000592]$,
  - The error generated by multiplying x_1 and x_2 is $[0.03142, 0.03718]$.

  We assume that $y = \max([|314.0|, |314.2|]) = 314.2$. Let us consider that $u = \text{ulp(y)}$. We have $\text{ulp(y)} = 2^{\text{ufp(y)}-p}$, where ufp() and p respectively are the *unit in the first place* and the precision of numbers. We have that $314.2 = 2^8 + 2^5 + 2^4 + 2^3 + 2^1 + \ldots$. This results that $\text{ufp}(314.2) = 8$. Note that, by taking $p = 4$, we find that $\text{ulp(y)} = 2^{8-4} = 2^4$. In our case, we have chosen to use the rounding mode to the nearest, ($\downarrow (x) = \frac{1}{2}\text{ulp(x)}$), which gives us at the last the following interval :

$$\downarrow^\sharp ([314.0, 314.2]) = [-2^3, 2^3] \quad .$$

Note that more efficient abstract domains exist, *e.g.*, [20, 16, 48, 51] as well as complementary techniques [14, 15, 37, 82]. Let us also mention that other methods exist to transform, synthesize or repair arithmetic expressions in the integer or fixed arithmetic [44]. Let us cite also [16, 42, 69, 71, 82] which are interesting to improve the ranges of the floating-point variables. These techniques could be combined with our approach in a future work.

3.5 Tools

To deal with numerical issues in computations, it is very important that programmers have sound tools which ensure and improve the numerical accuracy of their code. In these last few years, researchers have introduced a set of tools and we briefly describe the most representative hereafter.

- ASTREE [18, 25] is a static analyzer for critical software. It proves that real time and embedded programs, written in the C language, can be executed without error. This tool is able to detect specific errors of the C language like the division by zero, the overflow of an array, errors relatively to the most large representative value or errors due to non-compliance of properties defines by the user. ASTREE computes precise ranges for floating-point variables, this is why it is applied in several important industries.

- Fluctuat [49, 50] measures the error due to the use of floating-point numbers instead of reals in programs in the C language. It also, helps the user to debug its code by detecting the commands introducing the largest errors. Based on static analysis for numerical accuracy of floating-point computations and performing abstract interpretation [23, 24], Fluctuat is described in [38, 48, 63, 65]. This approach is used by several industries. It allows one, even with large intervals in inputs, to bound safety all the errors arising during a computation. In addition, it highlights the lines introducing the source of errors and facilitates the programmer task to detect the responsible operation of the significant precision loss. Then it helps the programmer to improve the accuracy by hand.

- PolySpace [4] is a commercial static analyzer tool integrated to Matlab/Simulink. PolySpace performs static analysis to automatically detect run-time errors. Using abstract interpretation and is used in many industries. For example, NASA software used this tool in the Mars Exploration Rover.

- Code Contracts [62] is a tool integrated to Microsoft Visual Studio Suite. It also uses abstract interpretation to validate the predicates of the pre and post-conditions of contracts asserted by the programmer.

- FPTaylor, a new tool used to estimate the round-off errors of floating-point computations presented in [82]. It uses Taylor series expansions and interval arithmetic to compute sound error bounds. This new approach consists in

1. Considering the round-off errors as noise,
2. Computing the Taylor expansions in a symbolic form,
3. Representing the difficult functional expressions as symbolic coefficients,
4. Applying a rigorous global maximization method.

- ROSA, a new approach introduced in [37] intends to combine an exact SMT solver based on reals with an approximate and sound affine and interval arithmetic computation. Its use requires to set a desired postconditions. The tool takes care of the uncertainties as well as the desired target precision. Next, the compiler verifies that the desired precision could be soundly obtained in finite-precision implementation when all the uncertainties and their propagation are included. This method gives an interesting results on many numerical codes, such as dynamical systems, transcendental functions, and controller implementation.

- Herbie [78] is a tool based on a heuristic search to improve the numerical accuracy of an arithmetic expression. It estimates and localizes the round-off errors of an expression using sampled points, applies a set of rules in order to improve the accuracy of the expression, takes series expansions and combines these improvements for the different inputs. More precisely and in order to evaluate the floating-point expression errors, Herbie :
  1. Samples the input points of programs,
  2. Compares the results obtained with floating-point arithmetic with the results obtained with an arbitrary precision arithmetic,
  3. Determines the operations that introduce the most error by comparing results from the two computations,
  4. Uses the automatic rewriting rules and makes series expansion in order to generate alternatives for the determined operations,
  5. Combines the different alternatives that improve the accuracy of the arithmetic expressions for the different input regions,
  6. Returns a new program which improves the numerical accuracy of expressions among all regions.

  Herbie has been evaluated through a set of examples taken from the classic numerical methods. The results obtained show that Herbie was able to improve the accuracy of arithmetic

expressions on each example considered. This gain is by up to 60 bits, while imposing a median performance overhead of 40%.

3.6 Transformation of Expressions

In this section, we briefly present former work by Arnault Ioualalen and Matthieu Martel to semantically transform [26] arithmetic expressions using Abstract Program Equivalent Graphs (APEG) [56]. This intermediary representation (IR) of programs is an extension of another IR called Equivalent Program Expression Graphs [85, 86]. EPEGs have been introduced to address the phase of ordering problems in compilers [9, 22, 61, 88], that is the problem of determining in which order to apply the optimizations of compilers to obtain the best result. For example, this makes it possible to search for the maximal shared sub-expressions [59, 83].
Contrarily to EPEGs, APEGs make it possible to remain in polynomial size while dealing with an exponential number of equivalent expressions. To prevent any combinatorial problem, APEGs hold in abstraction boxes many equivalent expressions up to associativity and commutativity. A box containing n operands can represent up to $1 \times 3 \times 5 \ldots \times (2n-3)$ possible formulæ. In order to build large APEGs, two algorithms are used : *propagation* and *expansion* algorithm. The first one searches recursively in the APEG where a symmetric binary operator is repeated and introduces abstraction boxes. Then, the second algorithm finds a homogeneous part and inserts a polynomial number of boxes. In order to add new shapes of expressions in an APEG, one propagates recursively subtractions and divisions into the concerned operands, propagate products, and factorizing common factors. Finally, an accurate formula is searched among all the equivalent formulæ represented in an APEG using the abstract semantics of Section 3.4.
Recall that in this section, the syntax used is defined by Equation (3.8) previously seen in Section 3.3.2.
The APEGs are an extension of the Equivalence Program Expression Graphs (EPEGs) introduced by R. Tate *et al.* [86, 85]. An APEG is defined inductively as follows :

1. A constant *cst* or an identifier *id* is an APEG,
2. An expression $p_1 * p_2$ is an APEG, where p_1 and p_2 are APEGs and $*$ is a binary operator among $\{+, -, \times, \div\}$,

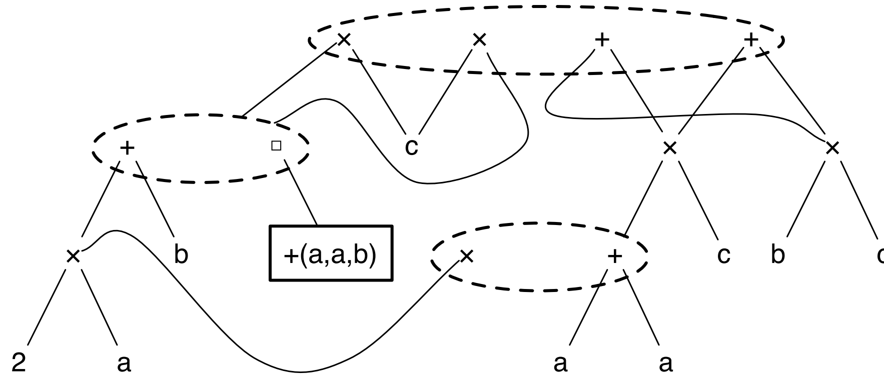


FIGURE 3-4 – *APEG for the expression* $e = \big((a+a)+b\big) \times c$.

3. A box $\boxed{*(p_1,\ldots,p_n)}$ is an APEG, where $* \in \{+,\times\}$ is a commutative and associative operator and the $p_{i,1\leq i\leq n}$, are APEGs,
4. A non-empty set $\{p_1,\ldots,p_n\}$ of APEGs consists in an APEG where $p_{i,1\leq i\leq n}$, is not a set of APEGs itself. We call the set $\{p_1,\ldots,p_n\}$ the equivalence class.

An example of APEG is given in Figure 3-4. When an equivalence class (denoted by a dotted ellipse in Figure 3-4) contains many APEGs $p_1,\ldots,p_n$ then one of the p_i, $1 \leq i \leq n$ may be selected in order to build an expression. A box $\boxed{*(p_1,\ldots,p_n)}$ represents any parsing of the expression $p_1 * \ldots * p_n$. From an implementation point of view, when several equivalent expressions share a common sub-expression, the latter is represented only once in the APEG. Then APEGs provide a compact representation of a set of equivalent expressions and make it possible to represent in an unique structure many equivalent expressions of very different shapes. For readability reasons, in Figure 3-4, the leafs corresponding to the variables a, b and c are duplicated while, in practice, they are defined only once in the structure.
The set $\mathscr{A}(p)$ of expressions contained inside an APEG p is defined inductively as follows :

1. If p is a constant *cst* or an identifier *id* then $\mathscr{A}(p) = \{cst\}$ or $\mathscr{A}(p) = \{x\}$,
2. If p is an expression $p_1 * p_2$, with $* \in \{+,\times\}$, then

$$\mathscr{A}(p) = \bigcup_{e_1\in\mathscr{A}(p_1),\ e_2\in\mathscr{A}(p_2)} e_1 * e_2 \ ,$$

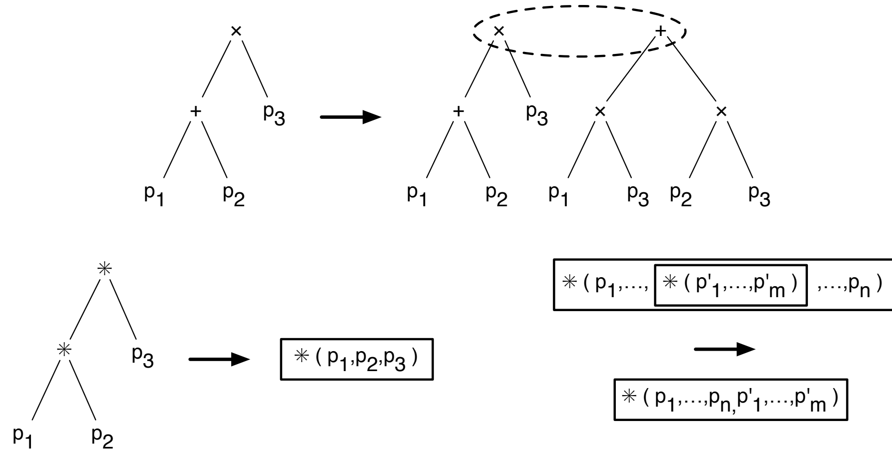


FIGURE 3-5 – *Some rules for APEG construction by pattern matching.*

3. If p is a box $\boxed{*(p_1,\dots,p_n)}$, with $* \in \{+,\times\}$, then $\mathscr{A}(p)$ contains all the parsings of $e_1 * \dots * e_n$ for all $e_1 \in \mathscr{A}(p_1),\dots,e_n \in \mathscr{A}(p_n)$,
4. If p is an equivalence class then $\mathscr{A}(p) = \bigcup_{1\leq i\leq n} \mathscr{A}(p_i)$.

For instance, the APEG p of Figure 3-4 represents all the following expressions :

$$\mathscr{A}(p) = \left\{ \begin{array}{c} \big((a+a)+b\big)\times c,\ \big((a+b)+a\big)\times c, \\ \big((b+a)+a\big)\times c,\ \big((2\times a)+b\big)\times c, \\ c\times\big((a+a)+b\big),\ c\times\big((a+b)+a\big), \\ c\times\big((b+a)+a\big),\ c\times\big((2\times a)+b\big), \\ (a+a)\times c+b\times c,\ (2\times a)\times c+b\times c, \\ b\times c+(a+a)\times c,\ b\times c+(2\times a)\times c \end{array} \right\} . \tag{3.26}$$

In their article on EPEGs, R. Tate *et al.* use rewriting rules to extend the structure up to saturation [86, 85]. In our context, such rules would consist of performing some pattern matching in an existing APEG p and then adding new nodes in p, once a pattern has been recognized.

Example 3.6.1

The Figure 3-5 gives the rules corresponding to distributivity and box construction for APEGs by pattern matching. In order to understand how do they work, let us explain the construction of each of these rules. For example, if we apply the distributive rule for APEGs on the expression $(p_1 + p_2) \times p_3$, we distribute the multiplication on the addition,

then we obtain several expressions, among them :

$$(p_1 \times p_3) + (p_2 \times p_3), \quad (p_3 \times p_2) + (p_1 \times p3), \quad (p_3 \times p_1) + (p_2 \times p_3), \quad \dots$$

The second rule represented in Figure 3-5 describes how to construct boxes. If the expressions are of the form $p_1 * p_2 * \cdots * p_n$, with $* \in \{+, \times\}$, boxes may be constructed. The box corresponding to the expression $p_1 * p_2 * \cdots * p_n$ is represented by $\boxed{*(p_1, p_2, \dots, p_n)}$ which represents the different manners to combine the $p_{i_{,i \in \{1,2,\dots,n\}}}$ with the operation $*$. In addition, if we are interesting to represent an expression with different operations, we can represent it as a box within another box. To illustrate that, let us take the following expression $(p_1 + p_2) \times p_3$. Its representation with boxes is given by $\boxed{*(\boxed{+(p_1, p_2)}, p_3)}$. It means the different combinations of the sum of p_1 and p_2 with the multiplication by p_3. So, we can rewrite it as

$$p_3 \times (p_1 + p_2), \quad (p_2 + p_1) \times p_3, \quad (p_1 \times p_3) + (p_2 \times p_3), \quad \dots$$

An alternative technique for APEG construction is to use dedicated algorithms. Such algorithms, working in polynomial time, have been proposed in [56].

3.7 Conclusion

In this chapter, we have seen briefly the computer arithmetic, more precisely the floating-point arithmetic which is specified by the IEEE754 standard. We have introduced the framework of static analysis by abstract interpretation which is a main technique in our approach and which is used to evaluate our arithmetic expressions and programs and compute the errors in our transformation. Finally, we have introduced an intermediary representation, APEG, defined by [56] and used to transform the arithmetic expressions.
In our contribution, we aim at going further than transforming arithmetic expressions. We are interesting in transforming pieces of code such as intraprocedural or interprocedural programs. We find these transformations with more details in the next chapter. A suite of transformation rules have been defined. They have been implemented in our tool SALSA.
It is well-known that the accuracy of floating-point computations is critical since subtle rounding errors can generate an important difference between the floating-point results and the exact results expected. In addition, the floating-point arithmetic is unintuitive and

very sensitive to roundoff errors arising during computations. To cope with all these problems, several numerical methods have been used in order to manually analyze and rewrite floating-point programs to guaranty their numerical accuracy. A limitation of these manual techniques is that they are difficult to apply and typically do not check certificates to guaranty the program accuracy. This is why the research community has developed many automated techniques guaranteeing bounds on the accuracy of floating-point computations and automatically improving their accuracy. Comparing the other techniques presented in Section 3.5, the approach of `SALSA` differs dramatically from them. `SALSA` improves the worst-case accuracy of programs, including arithmetic expressions and commands, using abstract interpretation to bound maximum absolute error, producing a sound over-approximation of the maximum error.

RÉSUMÉ CHAPITRE 4

Dans ce chapitre, nous nous focalisons sur la transformation intraprocédurale de programmes pour améliorer leur précision numérique. En d'autres termes, nous présentons comment transformer automatiquement des codes contenant les affectations, les structures de contrôle, les séquences de commandes ainsi la boucle `while`. Cette transformation, basée sue les méthodes d'analyse statique par interprétation abstraite, est définie par un ensemble de règles de transformation implémentées dans un outil appelé *SALSA*.
Nous expliquons par la suite le fonctionnement de chacune de ces règles de transformation. Les deux premières règles de transformation concernent les affectations. La première consiste à supprimer, sous certaines conditions, les affectations du code après les avoir sauvegardées dans la mémoire pour une future utilisation. La deuxième règle permet de substituer les variables mémorisées à l'aide de la première règle dans l'expression en question. Cette règle de transformation nous permet de générer des expressions arithmétique plus grandes que nous réécrivons de manière à trouver l'expression la plus précise. En ce qui concerne les structures de contrôles, nous avons quatre règles. Si le test de la conditionnelle est statiquement connu, autrement dit, soit le test est toujours évalué à *vrai* soit toujours à *faux*, nous transformons uniquement la bonne branche. Dans le cas contraire, nous transformons les deux branches de la conditionnelle. Une dernière règle de la conditionnelle consiste à ré-insérer avant une conditionnelle les variables qui ont été supprimées et qui ne devaient pas être enlevées. Pour les séquences de commandes, nous disposons de trois règles. Si l'un des membres de la séquence est un élément vide, nous nous intéressons à transformer simplement l'élément non nul. Sinon, nous transformons les deux membres de la séquence. Le dernier type des règles de transformation est dédié à la boucle `while`.

Une première règle permet de transformer le corps de la boucle tandis que la deuxième permet de ré-injecter des variables qui ont été supprimées dans le programme et dont on aura besoin pour notre transformation.
Dans le but de clarifier le fonctionnement de notre outil, nous détaillons via un exemple venant du domaine de la robotique comment transformer les programmes en utilisant ces règles de transformation. En partant du programme initial, nous décrivons le processus complet suivi par notre implémentation pour atteindre le programme optimisé.
À la fin de ce chapitre, nous donnons une preuve de correction de notre transformation de programmes. Notons que les programmes initiaux et transformés sont syntaxiquement très différents mais ils sont mathématiquement équivalents. Cette preuve, basée sur une sémantique opérationnelle, montre que notre approche retourne un programme plus précis parmi tout ceux générés.

CHAPITRE 4

INTRAPROCEDURAL TRANSFORMATION

Contents

4.1 Introduction **49**

4.2 Transformation of Commands **50**

4.3 Example of Transformation **58**

4.3.1 Odometry. 59

4.4 Proof of Correctness **64**

4.5 Conclusion **75**

4.1 Introduction

The main contribution of this chapter is our intraprocedural program transformation [26, 31]. The central goal is to generate a more accurate and efficient program using a fully automatic method of transformation. In general, the generated program may be semantically different from the program given in the input but both programs must be mathematically equivalent. For automatic transformation of single arithmetic expressions, several techniques have already been proposed. In particularly, we use the APEG techniques introduced in Section 3.6.

In this chapter, we start by describing in details how transforming intraprocedural programs. We define and explain the different rules used to optimize the accuracy of the

computations of programs. We introduce also a soundness theorem for our transformation. This proof relies on the operational semantics of Section 3.3.2 and is applied to the intraprocedural program transformation. Our main motivation is to prove that our transformation is efficient, in other words, by transforming programs, the programs generated are more accurate than the original one.
To achieve this transformation, our approach makes some assumptions.

1. The first hypothesis consists in writing the program in SSA form (Static Single Assignment) manner to avoid conflicts between variables read and written several times in programs.
2. The second hypothesis is that the transformation rules are applied deterministically on the programs. This means that rules are employed one by one in a given order and for each new transformed program, we reapply these rules until the final program does not change.
3. Another point consists in improving the numerical accuracy of a given variable which has been chosen by the user. We call it the *target variable* and we denote it by ϑ.
4. The final point relying on the choice of the best program. The most accurate program is selected from all the different equivalent programs generated, which are written in the same programming language, according to the value of the error on Variable ϑ.

4.2 Transformation of Commands

In this section, we describe the formal rules used to transform intraprocedural pieces of code. The syntax of commands is given in Equation (3.9).
The transformation rules $\Rightarrow_\vartheta$ that allow us to transform programs, in order to optimize a user-defined variable ϑ *i.e.*, the *target variable*, are presented in Figure 4-1. We denote by $\Rightarrow_\vartheta^*$ the transitive closure of $\Rightarrow_\vartheta$. In our approach, we use states of the form $\langle c, \delta, C, \beta \rangle$ where :

- c is a command, as defined in Equation (3.9),
- δ is an environment which maps variables to expressions,

$$\delta : \mathscr{V} \to \text{Expr} \ ,$$

with $\mathscr{V}$ denotes the finite set of identifiers. Intuitively, this environment, fed by Rule $(A1)$,

records the expressions assigned to variables in order to inline them later on in larger expressions thanks to Rule $(A2)$. We write $\delta[x \mapsto e]$, the environment which is equal to $\delta(x)$ for all id $\in \mathscr{V}$ such that id $\neq x$, otherwise, it returns the value e.

– $C \in$ Ctx is a single hole context [53] defined in Equation (4.1). It records the program enclosing the current expression to be transformed and which is intended to fit in the hole denoted by [ ].

$$\text{Ctx} \ni C ::= [\,] \mid id = e \mid C_1 \,; C_2 \mid \mathsf{if}_\Phi \; e \; \mathsf{then} \; C_1 \; \mathsf{else} \; C_2 \mid \mathsf{while}_\Phi \; e \; \mathsf{do} \; C \mid \mathtt{nop} \quad , \quad (4.1)$$

with e is defined by Equation (3.8).

– let $\vartheta \in \mathscr{V}$ denote the *target variable* that we aim at optimizing.

– let $\beta \subseteq \mathscr{V}$ be a list of assigned variables that should not be removed from the source program. Initially, $\beta = \{\vartheta\}$, *i.e.*, the target variable ϑ may not be removed. The set β is modified by rules $(C1), (C2), (C4)$ and $(W2)$.

Before describing the rules of Figure 4-1, let us start with introducing the notations used in our transformation.

– Assigned(c) denotes the set of identifiers assigned in the command c,

– Dom(δ) denotes the list of variables memorized in δ and

– Var(e) denotes the set of variables of the expression e.

Rule $(A1)$ allows one to discard an assignment $id = e$ by memorizing in δ the formal expression e in order to inline it later, in a larger expression. When using Rule $(A1)$, to get a semantically equivalent program, we must respect a restriction which is that the transformation is done only if Identifier *id* does not belong to the set β of variables which must not be removed. Note that we always have $\vartheta \in \beta$ so that the identifier *id* of the assignment cannot be the *target variable* when using $(A1)$.
Rule $(A2)$ offers an alternative way of processing assignments, when the condition of Rule $(A1)$ is not fulfilled. The action of substituting the variables of e by their definitions in δ is denoted by $\delta(e)$. We also use the function $Size(e)$ which computes the size of the expression e (*i.e.*, the number of nodes of its syntactic tree). Rule $(A2)$ transforms the expression $e' = \delta(e)$ into an expression e'' by a call $\langle e', \sigma^\sharp \rangle \rightsquigarrow^* e''$ to the tool based on APEGs and which transforms expressions, as described in Section 3.6 in Chapter 3. The abstract environment $\sigma^\sharp : \mathscr{V} \rightarrow \mathbb{E}^\sharp$ used for this transformation results from a static

$$\frac{\delta' = \delta[id \mapsto e] \quad id \notin \beta}{\langle id = e, \delta, C, \beta \rangle \Rightarrow_\vartheta \langle \texttt{nop}, \delta', C, \beta \rangle} \quad (A1)$$

$$\frac{e' = \delta(e) \quad \sigma^\sharp = [\![C[c]]\!]^\sharp \iota^\sharp \quad \langle e', \sigma^\sharp \rangle \leadsto^* e''}{\langle id = e, \delta, C, \beta \rangle \Rightarrow_\vartheta \langle id = e'', \delta, C, \beta \rangle} \quad (A2)$$

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

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

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

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

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

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

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

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

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

FIGURE 4-1 – *Transformation rules used to improve the accuracy of programs.*

analysis using the domain $\mathbb{E}^\sharp$ also introduced in Section 3.4 in Chapter 3. As mentioned earlier, in Rule $(A2)$, $\iota^\sharp$ denotes the user-defined initial environment which binds the free variables of the program to intervals. For example, in Section 4.3.1, the variable `s1` is set to `[0.52,0.53]` in $\iota^\sharp$. The program given to the static analyzer is $C[c]$, *i.e.*, the program obtained by inserting the command c into the context C. According to these notations, if the size of the inlined expression $e' = \delta(e)$ is less than a user-defined maximal size S_{Max}, then the expression e' is transformed into an expression e'' by $\langle e', \sigma^\sharp \rangle \rightsquigarrow^* e''$ which transforms the source expression into a more accurate one for the environment $\sigma^\sharp$. In our implementation, this corresponds to a call to the APEG tool [56, 67]. The returned expression e'' is inserted in the new assignment $id = e''$.

Remark that by inlining expressions in variables when transforming programs, we create large formulæ. In our implementation, in order to facilitate their manipulation, we slice these formulæ at a defined level of the syntactic tree on several sub-expressions and we assign them to intermediary variables. Finally, we inject these new assignments into the main program, see `TMP` in Listing 4.5 in Section 4.3 of this chapter.

Rules $(S1)$ to $(S3)$ deal with sequences. Rules $(S1)$ and $(S2)$ are special cases enabling the system to discard the `nop` statements while the general rule for sequences is $(S3)$. The first command c_1 is transformed into c_1' in the current environment δ, in the context $C\big[[];c_2\big]$ and a new context C'' is built which inserts c_1' inside C. Then c_2 is transformed into c_2' using the context $C[c_1';[]]$, the formal environments δ' and the list β' resulting from the transformation of c_1. Finally, we return the state $\langle c_1' \ ; \ c_2', \delta'', C, \beta'' \rangle$.

Example 4.2.1

To explain the use of rules $(A1)$, $(A2)$, $(S1)$ to $(S3)$, let us consider the example of Equation (4.2) in which three variables x, y and z are assigned. In this example, ϑ consists of the variable z that we aim at optimizing and $a = 0.1$, $b = 0.01$, $c = 0.001$ and $d = 0.0001$

are constants.

$$
\begin{aligned}
& \langle \mathtt{x} = \mathtt{a}+\mathtt{b};\, \mathtt{y} = \mathtt{c}+\mathtt{d};\, \mathtt{z} = \mathtt{x}+\mathtt{y},\ \delta,\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(A1)}{\Longrightarrow_\vartheta}\ & \langle \mathtt{nop};\, \mathtt{y} = \mathtt{c}+\mathtt{d};\, \mathtt{z} = \mathtt{x}+\mathtt{y},\ \delta' = \delta[\mathtt{x} \mapsto \mathtt{a}+\mathtt{b}],\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(S1)}{\Longrightarrow_\vartheta}\ & \langle \mathtt{y} = \mathtt{c}+\mathtt{d};\, \mathtt{z} = \mathtt{x}+\mathtt{y},\ \delta' = \delta[\mathtt{x} \mapsto \mathtt{a}+\mathtt{b}],\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(A1)}{\Longrightarrow_\vartheta}\ & \langle \mathtt{nop};\, \mathtt{z} = \mathtt{x}+\mathtt{y},\ \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c}+\mathtt{d}],\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(S1)}{\Longrightarrow_\vartheta}\ & \langle \mathtt{z} = \mathtt{x}+\mathtt{y},\ \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c}+\mathtt{d}],\ [\,],\ \{\mathtt{z}\}\rangle \\
\underset{(A2)}{\Longrightarrow_\vartheta}\ & \langle \mathtt{z} = ((\mathtt{d}+\mathtt{c})+\mathtt{b})+\mathtt{a}, \delta'',\ [\,], \{\mathtt{z}\}\rangle
\end{aligned}
\tag{4.2}
$$

In Equation (4.2), initially, the environment δ is empty. If we apply the first rule $(A1)$, we may remove the variable x and memorize it in δ. So, the line corresponding to the variable discarded is replaced by `nop` and the new environment is $\delta = [x \mapsto a+b]$. We then repeat the same process by using $(A1)$ on the variable y and Rule $(S1)$ which discards the `nop` statement. For the last step, we must not apply $(A1)$ to z because the condition is not satisfied ($z = \vartheta$). Then we use $(A2)$, we substitute x and y by their value in δ and we transform the expression.
The rules $(S1)$ and $(S3)$ are needed to achieve the transformation : First, $(S3)$ is used with $c_1 = \{\mathtt{x = a + b}\}$ and $c_2 = \{\mathtt{y = c + d ; z = x + y}\}$. For c_1, $(A1)$ is called from $(S3)$ and, concerning c_2, $(S3)$ is recursively called, as well as $(A1)$ and $(S1)$.

Rules $(C1)$ to $(C4)$ concern conditionals. The first two rules correspond to a partial evaluation of the program [57], when the test evaluates to `true` or `false` in the environment $\sigma^\sharp$ which is computed by static analysis

$$\sigma^\sharp = [\![C[\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]]\!]^\sharp \iota^\sharp \ .$$

The conditional, in rules $(C1)$ and $(C2)$, is replaced by either the Branch c_1 or c_2. Since the conditional is removed, we have to take care of the Φ-nodes. We use the function $\Psi(\Phi, c)$ which replaces in the command c any assignment $x = e$ by $y = e$ if $\Phi(y,u,v) \in \Phi$ with $u = x$ or $v = x$. Similarly, $\Psi(\Phi, \delta)$ replaces, in the formal environment δ, $x \mapsto e$ by $y \mapsto e$ if $\Phi(y,u,v) \in \Phi$ with $u = x$ or $v = x$. Doing so, we propagate the effect of the Φ-nodes when a conditional is removed. In this case, the *target variable* ϑ does not appear necessarily in

c_1 or c_2 but the variables assigned in these branches are used in the Φ nodes. Consequently, they may not be removed from c_1 or c_2 and we have to transform the command with $\beta' = \beta \cup Assigned(c_i)$, for $i = 1$ or 2. Here, $Assigned(c)$ denotes the set of identifiers assigned in the command c. Finally, some assignments have to be inserted before the command of the transformed branch. They correspond to the variables read in c'_1 or c'_2 and whose definitions have been stored in δ. Note that, in some cases and when it is necessary, we re-inject variables that have been discarded from the main program by means of the function $AddDefs$.

Example 4.2.2

Let us consider the program p, in SSA form.

$$p \quad \equiv \quad \begin{array}{l} x_1 = 0; \\ \text{if}_{\Phi(x_4,x_2,x_3)}\ \mathit{cond}\ \text{then} \\ \quad x_2 = a+b \\ \text{else} \\ \quad x_3 = c+d; \\ \vartheta = x_4 \end{array} \tag{4.3}$$

In the rest of this example, we assume that $c_1 \equiv a+b$ and $c_2 \equiv c+d$. Depending on the value of condition, we transform this program into

$$\begin{cases} x_1 = 0; \\ \vartheta = a+b & \text{if } \mathit{cond} \ , \\ \vartheta = c+d & \text{if } \neg\mathit{cond} \ . \end{cases} \tag{4.4}$$

For example, if the condition is true, the steps followed by the transformation are

$$\begin{array}{rcl} \langle p,\delta,C,\beta\rangle & \underset{(C1)}{\Longrightarrow_\vartheta} & \langle \Psi(\Phi,c_1),\Psi(\Phi,\delta),C,\beta\rangle \\ & \equiv & \langle \Psi(\Phi,x_2 = a+b;\vartheta = x_2),\Psi(\Phi,\delta),C,\beta\rangle \\ & \equiv & \langle x_4 = a+b;\vartheta = x_4),\delta,C,\beta\rangle \\ & \underset{(A1)}{\Longrightarrow_\vartheta} & \langle \mathtt{nop};\vartheta = x_4,\delta'[x_4 \mapsto a+b],C,\beta\rangle \\ & \underset{(S1)}{\Longrightarrow_\vartheta} & \langle \vartheta = x_4,\delta'[x_4 \mapsto a+b],C,\beta\rangle \\ & \underset{(A2)}{\Longrightarrow_\vartheta} & \langle \vartheta = a+b,\delta',C,\beta\rangle \end{array}$$

■

Rule $(C3)$ is the general rule for conditionals. The then and else branches are transformed, assuming that the variables of the condition do not appear the domain of δ. Here, $\mathit{Assigned}(c)$ denotes the set of identifiers assigned in the command c. The variables assigned in the branches have to be added to β and the environment δ' resulting from the union of δ'_1 and δ'_2 generated respectively by the transformation of c_1 and c_2. Thanks to the SSA form, the variables assigned in both branches are distinct. The function $\mathit{Var}(e)$ re-

turns the set of variables occurring in the expression e while $\mathrm{Dom}(\delta)$ denotes the domain of definition of δ. Finally, Rule $(C4)$ is used when the conditions for Rule $(C3)$ do not hold. In this case, $Var(e) \cap Dom(\delta) \neq \emptyset$ and we need to reinsert the common variables into the source code. Let $Var(e)$ be the list of variables occurring in the expression e. Firstly, a new command c' corresponding to sequences of assignments of the form $id = \delta(id)$ is built from $AddDefs(Var(e), \delta)$ such that for any set of variables V

$$AddDefs(V,\delta) \equiv \mathrm{id}_1 = \delta(\mathrm{id}_1)\ \ ;\ \ \ldots\ \ ;\ \ \mathrm{id}_n = \delta(\mathrm{id}_n)\ \ \text{with}\ \ \ V = \{\mathrm{id}_1, \ldots, \mathrm{id}_n\}\ \ . \tag{4.5}$$

Secondly, the variables of $Var(e)$ are removed from the domain of δ, yielding δ'. The resulting command is the command c'' obtained by transforming $c'; \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$ with δ' and $\beta \cup Var(e)$.

Example 4.2.3

Let us take another example to explain the rules $(C3)$ and $(C4)$. Initially, we have $\langle q, \delta, C, \beta = \{\vartheta\}\rangle$ and

$$q \quad \equiv \quad \begin{array}{l} x_1 = a; \\ \mathsf{if}_{\Phi(y_3,y_1,y_2)}\ (x_1 > 1)\ \mathsf{then} \\ \quad y_1 = x_1 + 2; \\ \mathsf{else} \\ \quad y_2 = x_1 - 1; \\ \vartheta = y_3 \end{array} \tag{4.6}$$

By rule $(A1)$, x_1 is stored in δ. We write $\langle q, \delta, C, \beta = \{\vartheta\}\rangle \underset{(A1)}{\Longrightarrow}_{\vartheta} \langle q', \delta'[x_1 \mapsto a], C, \beta = \{\vartheta\}\rangle$ Then, we transform recursively the new program

$$q' \quad \equiv \quad \begin{array}{l} \mathsf{if}_{\Phi(y_3,y_1,y_2)}\ (x_1 > 1)\ \mathsf{then} \\ \quad y_1 = x_1 + 2; \\ \mathsf{else} \\ \quad y_2 = x_1 - 1; \\ \vartheta = y_3 \end{array} \tag{4.7}$$

This program is semantically incorrect since the test in q' is undefined. However,

$$Var(e) \cap Dom(\delta) \neq \emptyset$$

and we cannot apply Rule $(C3)$. Instead, Rule $(C4)$ is used to re-inject the statements $x_1 = a$ in the program and to add x_1 to the blacklist β in order to avoid an infinite loop in the transformation. The new blacklist is $\beta' = \{\vartheta, x_1\}$. Note that since our programs are in SSA form, we do not need to take care of variables redefinitions or scope.

The last two rules $(W1)$ and $(W2)$ are for the while statements. They follow the same spirit than the rules $(C3)$ and $(C4)$ for the conditional statement. Rule $(W1)$ makes it possible to transform the body c of the loop assuming that the variables of the condition e have not been stored in δ. In this case, c is optimized in the context $C[\texttt{while}_\Phi\ e\ \texttt{do}\ []]$ where C is the context of the loop itself. Rule $(W2)$ first builds the list $V = Var(e) \cup Var(\Phi)$ where $Var(\Phi)$ is the list of variables read and written in the Φ nodes of the loop. The set V is used to achieve two tasks : firstly, it is used to build a new command c' corresponding to the sequence of assignments $id = \delta(id)$, for all $id \in V$, as for Rule $(C4)$. Secondly, the variables of V are removed from the domain of δ and added to β. The resulting command is the command c'' obtained by transforming $c'; \texttt{while}_\Phi\ e\ \texttt{do}\ c$ with δ' and $\beta \cup V$.
We end this section with complexity considerations. At each step of the transformation of a program p, only one rule of Figure 4-1 may be selected. Consequently, the transformation would be linear in the size n, *i.e.*, the number of statements, of p if we would not re-inject assignments. However, a given assignment cannot be removed twice, so the transformation is quadratic. Finally, the entire transformation of a program p is repeated until nothing changes, that is at most n times. Hence, the global complexity for a program transformation of size n is $\mathcal{O}(n^3)$.

4.3 Example of Transformation

In this section, we give a detailed explanation of how we transform programs using the transformation rules presented in Section 4.2. We give the different steps needed to achieve the optimized program from the original one. In the rest of this thesis and for a sake of simplicity and length, all the examples are simply stated, *i.e.*, we give just their code before and after being transformed.

4.3.1 Odometry.

To illustrate our transformation, we have chosen an example taken from Robotics and whose code is given in Listing 4.1. It concerns the computation of the position (x,y) of a two wheeled robot by odometry. Given the instantaneous rotation speeds s_l and s_r of the left and right wheels, it aims at computing the position of the robot in a Cartesian space. Let C be the circumference of the wheels of the robot, L the length of its axle and θ the robot moving angle (see Figure 4-2). We assume that s_l and s_r, are coming from sensors,

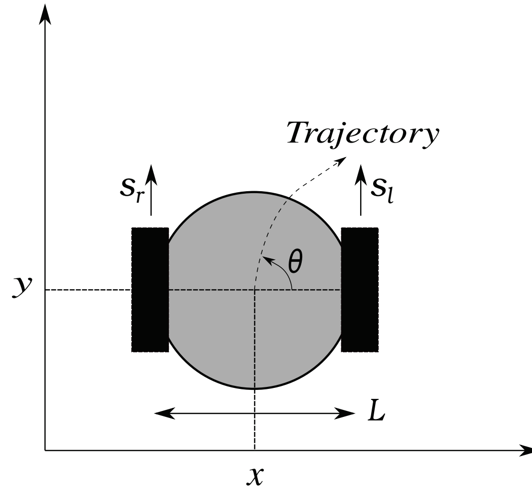


FIGURE 4-2 – *Parameters of the two-wheeled robot.*

are updated by the system, by side-effect. The computation of the position of the robot is given by

$$x(t+1) = x(t) + \Delta d(t+1) \times \cos\left(\theta(t) + \frac{\Delta\theta(t+1)}{2}\right) \quad , \tag{4.8}$$

$$y(t+1) = y(t) + \Delta d(t+1) \times \sin\left(\theta(t) + \frac{\Delta\theta(t+1)}{2}\right) \quad , \tag{4.9}$$

with

$$\theta(t+1) = \theta(t) + \Delta\theta(t) \quad , \tag{4.10}$$

$$\Delta d(t) = \big(\Delta d_r(t) + \Delta d_l(t)\big) \times 0.5 \quad , \tag{4.11}$$

$$\Delta d_l(t) = s_l(t) \times C \quad , \tag{4.12}$$

$$\Delta d_r(t) = s_r(t) \times C \quad , \tag{4.13}$$

$$\Delta\theta(t) = \big(\Delta d_r(t) - \Delta d_l(t)\big) \times \frac{1}{L} \quad . \tag{4.14}$$

In equations (4.8) to (4.14), $\theta(t)$ is the direction of the robot, $d(t)$ is the elementary movement of the robot at time t and $d_l(t)$ and $d_r(t)$ are the elementary movements of the left and right wheels. We assume that `cos` and `sin`, not computed by a library, are obtained by a Taylor series expansion as shown in Equations (4.15) and (4.16).

$$\cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} \quad , \tag{4.15}$$

$$\sin(y) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} \quad . \tag{4.16}$$

Listing 4.1 – Listing of the original Odometry program.

```
sl = [0.52,0.53];
sr = 0.785398163397; theta = 0.0; t = 0.0; x = 0.0; y = 0.0; inv_l = 0.1; c = 12.34;
while (t < 100.0) do {
  delta_dl = (c * sl);
  delta_dr = (c * sr);
  delta_d = ((delta_dl + delta_dr) * 0.5);
  delta_theta = ((delta_dr - delta_dl) * inv_l);
  arg = (theta + (delta_theta * 0.5));
  cos = (1.0 - ((arg * arg) * 0.5)) + ((((arg * arg)* arg)* arg) / 24.0);
  x = (x + (delta_d * cos));
  sin = (arg - (((arg * arg)* arg)/6.0))+(((((arg * arg)* arg)* arg)* arg)/120.0);
  y = (y + (delta_d * sin));
  theta = (theta + delta_theta);
  t = (t + 0.1);
}
```

Before starting the transformation, we have to design a *target variable* that we aim at optimizing, ϑ. In our case, we would like to improve the accuracy of the variable x corresponding to the position of the robot on the axis of abscissa, *i.e.*, $\vartheta = \{x\}$. Assume that the black list contains the variable at improving in order to avoid such elimination from the program while transforming, $\beta = \{x\}$. Initially, the environment δ and the context C are empty. The program is given at Listing 4.1.

As mentioned previously in Section 2, our transformation rules are applied in a deterministic order, that means, our tool uses these rules one after the other. As remark, the assignment $sl = [0.52, 0.53]$ belongs to the variable $\iota^\sharp$ discussed in Section 4.2, that we have not to remove from the program along of the transformation because it is an input. Firstly, if all the restrictions presented in Rule $(A1)$ of Figure 4-1 are satisfied, in other words,

– The identifier *sr* of the assignment is different from the *target variable* ;
– The identifier *sr* of the assignment does not belong to the black list ;
– The identifier *sr* of the assignment is not in the environment δ,

then, the first rule of assignment $(A1)$ is applied. So, the first assignment of our Listing 4.1 $sr = 0.785398163397$ will be removed from the program and memorized in the environment δ for a future use. The new δ then is $\delta' = [sr \mapsto 0.785398163397]$. Note that when we remove the assignment, in the program, we replace it by the statement `nop` that means no operation. Our tool is charged to simplify this line by deleting this `nop` using the sequences rules defined in Figure 4-1. The concerned rule is $(S1)$, it allows one to remove the first member of a sequence from the code. Second, our tool position on the next line of program, that means the assignment $\theta = 0.0$, it verifies if all conditions are alright, then, this assignment will be removed from the code and saved in the memory δ. The updated memory is $\delta'' = [sr \mapsto 0.785398163397;\ \theta \mapsto 0.0]$. This process of applying $(A1)$ and then $(S1)$ is repeated several times on all the assignments found before the `while` loop. The final δ in this case is $\delta = [sr \mapsto 0.785398163397;\ \theta \mapsto 0.0,\ t \mapsto 0.0;\ y \mapsto 0.0;\ inv_l \mapsto 0.1;\ c \mapsto 12.34]$. Note that, for this moment, the context is always [] until now.
The new program after all these transformations is the following, given by Figure 4.2.

Listing 4.2 – Listing of the Odometry program during transformation.

```
sl = [0.52,0.53];
while (t < 100.0) do {
  delta_dl = (c * sl);
  delta_dr = (c * sr);
  delta_d = ((delta_dl + delta_dr) * 0.5);
  delta_theta = ((delta_dr - delta_dl) * inv_l);
  arg = (theta + (delta_theta * 0.5));
  cos = (1.0 - ((arg * arg) * 0.5)) + ((((arg * arg)* arg)* arg) / 24.0);
  x = (x + (delta_d * cos));
  sin = (arg - (((arg * arg)* arg)/6.0))+(((((arg * arg)* arg)* arg)* arg)/120.0);
  y = (y + (delta_d * sin));
  theta = (theta + delta_theta);
  t = (t + 0.1);
}
```

with $\delta = [sr \mapsto 0.785398163397;\ \theta \mapsto 0.0,\ t \mapsto 0.0;\ y \mapsto 0.0;\ inv_l \mapsto 0.1;\ c \mapsto 12.34]$, C = [], $\vartheta = \{x\}$, $\beta = \{x\}$.
Now, our tool is positioned on the line corresponding to the `while` loop. If the test is true, we apply the rule $(W1)$ as mentioned on Figure 4-1 to transform the body of the loop. Our tool begins by analyzing the body of the loop, in each case it found that some variables

are undefined in the body, the program is likely incorrect. For this reason, we introduce the second rule $(W2)$ that allows one to re-insert the variables which have been omitted from the code and been saved into δ. These variables are y, θ and t. In order to avoid such scenario in the future, these variables must be added into the blacklist and removed from the environment δ. Then $\beta = \{x, y, \theta, t\}$ and $\delta = [sr \mapsto 0.785398163397;\ inv_l \mapsto 0.1;\ c \mapsto 12.34]$. At this level, the context C is [] again.
Our tool enters into the body of the loop and starts the transformation. Here, the context is changed and it contains all the program enclosing the commands at optimizing, in other words, the context is :

$$
\begin{array}{l}
C = [\\
\qquad sl = [0.52, 0.53];\ theta = 0.0;\ t = 0.0;\ x = 0.0;\ y = 0.0; \\
\qquad while\ (t < 100.0)\ do\{ \\
\qquad \ldots \\
\qquad \} \\
\quad]
\end{array}
$$

The following step consists in using the assignment rules on the first line of the loop body as previously done. If all conditions are satisfied, we apply $(A1)$ on the first assignment $delta_{dl}$. So, we omit this assignment from the program and we kept it within the environment δ. The same steps are applied repeatedly on the rest of the assignment presented in the loop body whose are $delta_{dr}$, $delta_d$, $delta_{theta}$, arg, cos and sin. The intermediate transformed program is described in Listing 4.3.

Listing 4.3 – Listing of the Odometry program during transformation.

```
sl = [0.52,0.53]; theta = 0.0; t = 0.0; x = 0.0; y = 0.0;
while (t < 100.0) do {
  x = (x + (delta_d * cos));
  y = (y + (delta_d * sin));
  theta = (theta + delta_theta);
  t = (t + 0.1);
}
```

We observe that our transformation to be correct, we need both assignments `sin` and `cos`. These assignments must be re-inserted into the body of the loop. Then, we apply on them the transformation rule $(A2)$ which consists in inlining the expressions saved in the environment δ within the corresponding assignment, *i.e.*, within `sin` and `cos` and then within x and y, and next, we rewrite the obtained expression by calling the APEG's tool to return a more accurate expressions. By this substitution, a larger expressions have been created.

For example, the assignment of x, after inlining, is given by Equation (4.17) :

$$\begin{aligned} x = \quad & (x+((0.5\times((1.0-((theta+(((9.69181333632-(sl\times 12.34))\times 0.1)\times 0.5))\times(0.5\times((0.5\\ & \times(0.1\times(9.69181333632-(sl\times 12.34))))+theta))))+(((((theta+(((9.69181333632\\ & -(sl\times 12.34))\times 0.1)\times 0.5))\times(theta+(((9.69181333632-(sl\times 12.34))\times 0.1)\times 0.5)))\\ & \times(theta+(((9.69181333632-(sl\times 12.34))\times 0.1)\times 0.5)))\times(theta+(((9.69181333632\\ & -(sl\times 12.34))\times 0.1)\times 0.5)))/24.0)))\times(9.69181333632+(sl\times 12.34)))); \end{aligned} \tag{4.17}$$

The new context at this point of transformation is :

$$\begin{aligned} C = [& \\ & sl=[0.52,0.53];\ theta=0.0;\ t=0.0;\ x=0.0;\ y=0.0;\\ & while\ (t<100.0)\ do\\ & \{\\ & \vdots\\ & x=(x+(delta_d\times cos));\\ & y=(y+(delta_d\times sin));\\ & theta=(theta+delta_{theta});\\ & t=(t+0.1)\\ & \}\\ ] & \end{aligned}$$

In Listing 4.4, we give the intermediate odometry program including the substitutions done.

Listing 4.4 – Listing of the Odometry program during transformation.

```
sl = [0.52,0.53]; theta = 0.0; t = 0.0; x = 0.0; y = 0.0;
while (t < 100.0) do {
  x = (x + ((0.5 * ((1.0 - ((theta + (((9.69181333632 - (sl * 12.34)) * 0.1) * 0.5))
    * (0.5 * ((0.5 * (0.1 * (9.69181333632 - (sl * 12.34))))+ theta)))) + (((((theta
    + (((9.69181333632 - (sl * 12.34)) * 0.1) * 0.5)) * (theta + (((9.69181333632
    - (sl * 12.34)) * 0.1) * 0.5))) * (theta + (((9.69181333632 - (sl * 12.34)) * 0.1)
    * 0.5)))* (theta + (((9.69181333632 - (sl * 12.34)) * 0.1) * 0.5))) / 24.0)))
    * (9.69181333632 + (sl * 12.34)))) ;
  y = (((9.691813336318980 + (12.34 * sl)) * [...] ;
  theta = (theta + (0.1 * (9.69181333632 - (sl * 12.34)))) ;
  t = (t + 0.1);
}
```

When dealing with very large expressions, as those for x and y, we proceed at slicing them

at a given level of the syntactic tree, and then, associated them to intermediary variables namely TMP. This is the last step of our transformation. The final program optimized is shown in Listing 4.5.

Listing 4.5 – Listing of the transformed Odometry program.

```
sl = [0.52,0.53]; theta = 0.0; t = 0.0; x = 0.0; y = 0.0;
while (t < 100.0) do {
  TMP_6 = (0.1 * (0.5 * (9.691813336318980 - (12.34 * sl))));
  TMP_23 = ((theta + (((9.691813336318980 - (sl * 12.34)) * 0.1) * 0.5)) * (theta
         + (((9.691813336318980 - (sl * 12.34)) * 0.1) * 0.5)));
  TMP_25 = ((theta + TMP_6) * (theta + TMP_6)) * (theta + (((9.691813336318980
         - (sl * 12.34)) * 0.1) * 0.5));
  TMP_26 = (theta + TMP_6) ;
  x = ((0.5 * (((1.0 - (TMP_23 * 0.5)) + ((TMP_25 * TMP_26) / 24.0)) * ((12.34 * sl)
    + 9.691813336318980))) + x);
  TMP_27 = ((TMP_26 * TMP_26) * (theta + (((9.691813336318980 - (sl * 12.34))
         * 0.1) * 0.5)));
  TMP_29 = (((TMP_26 * TMP_26) * TMP_26) * (theta + (((9.691813336318980 - (sl * 12.34))
         * 0.1) * 0.5)));
  y = (((9.691813336318980 + (12.34 * sl)) * (((TMP_26 - (TMP_27 / 6.0)) + ((TMP_29
    * TMP_26) / 120.0)) * 0.5)) + y);
  theta = (theta + (0.1 * (9.691813336318980 - (12.34 * sl))));
  t = t + 0.1;
}
```

This transformation offers several advantages :

- it creates large enough formulæ well-suited to be efficiently transformed by existing techniques for the transformation of arithmetic expressions, based on the use of Abstract Program Expression Graphs [56, 67].
- it may create static formulæ made of constant terms which can be evaluated statically in an extended precision arithmetic. This may also reduce the number of operations in the target program and then optimizes its execution time, see [28].

4.4 Proof of Correctness

In an effort to assert the correctness of our transformation for numerical accuracy, we now prove that our approach really generates a more accurate and correct program among the many equivalent programs. This proof relies on the operational semantics of both arithmetic expressions and commands. It is crucial because the transformed programs are syntactically different from the original ones. Indeed, if we operate on critical embedded systems,

one necessarily needs to be sure that the transformed programs really behave like the original ones.
Key to our approach is the use of Theorem 4.4.3 introduced later on in this section and which compares two programs and states that the most accurate one may be used in place of the less accurate one. This comparison needs to specify a *target variable* ϑ defined by the user. More precisely, a transformed program p_t is more accurate than an original program p_o if and only if the two conditions given hereafter are verified :

- The first condition consists in ensuring that the *target variable* ϑ of both programs corresponds to the same mathematical expression.
- The second condition requires that the *target variable* ϑ is more accurate in the transformed program p_t than in the original program p_o.

The main difficulty of the proof comes from the fact that the transformation discards some assignments which have to be re-injected later on to build larger expressions. This makes our transformation unusual and forms the originality of the proof presented below.
In the rest of this section, we are going to use the rules, introduced in Section 4.2, of the form $\langle c,\delta,C,\beta\rangle \Rightarrow_{\vartheta}^{*} \langle c',\delta',C,\beta'\rangle$. However, to avoid overloaded notations, we will omit the context C and the black list β whenever they are not necessary to the proof and we will write simply $\langle c,\delta\rangle \Rightarrow_{\vartheta}^{*} \langle c',\delta'\rangle$. We denote by $\Rightarrow_{\vartheta}^{*}$ the reflexive transitive closure of $\Rightarrow_{\vartheta}$.
In order to define the correctness of the transformation, we introduce two order relations which make it possible to compare commands based on their accuracy.
First, we use the order relation $\sqsubseteq \subseteq \mathbb{E}\times\mathbb{E}$, with $\mathbb{E}=\mathbb{F}\times\mathbb{R}$, which states that a value (x,μ) is more accurate than a value (x',μ') if they correspond to the same real value and if the error μ is less than μ'.
Recall that the operational semantics used is defined by

$$\rightarrow : \left| \begin{array}{l} \text{State} \rightarrow \text{State} \ , \\ \langle c,\sigma\rangle \mapsto \langle c',\sigma'\rangle \end{array} \right. .$$

with State is defined by a pair of command (Cmd) and memory (Mem),

$$\{\langle c,\sigma\rangle : c\in \text{Cmd}, \sigma\in \text{Mem}\}.$$

The transition functions for the evaluation of arithmetic (Expr) and boolean (BExpr) expressions are defined as following

$$\rightarrow_e : \ \text{Expr} \rightarrow \mathbb{E} \ ,$$

$$\rightarrow_b : \text{BExpr} \rightarrow \{\texttt{true}, \texttt{false}\} \ .$$

Definition 4.4.1 (Comparison of Expressions)**.** Let us consider $v_1 = (x_1, \mu_1) \in \mathbb{E}$ and $v_2 = (x_2, \mu_2) \in \mathbb{E}$. We say that v_1 is more accurate than v_2, denoted by $v_1 \sqsubseteq v_2$ iff $x_1 + \mu_1 = x_2 + \mu_2$ and $|\mu_1| \leq |\mu_2|$. ■

Second, Definition 4.4.2 says that :

- Two commands compute the same value of ϑ in any environment in the exact arithmetic,
- The transformed command is more accurate,
- If a variable is not defined in the concrete environment σ_t after execution then the corresponding formal expression has been stored in δ_t.

Let $\sigma : \mathscr{V} \rightarrow \mathbb{E}$ be an environment for values and $\delta : \mathscr{V} \rightarrow \text{Expr}$ be a formal environment. In the following, $\delta(e)$ is the expression obtained by substituting in the expression e any identifier *id* of δ by its value $\delta(\text{id})$.

Definition 4.4.2 (Comparison of Commands)**.** Let c_o and c_t be two commands, let δ_o and δ_t be two formal environments. Finally, let ϑ be the *target variable*. We say that

$$\langle c_t, \delta_t \rangle \prec_\vartheta \langle c_o, \delta_o \rangle$$

if and only if for all $\sigma \in \text{Mem}$,

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

such that $\sigma_t(\vartheta) \sqsubseteq \sigma_o(\vartheta)$ and such that for all $\text{id} \in \text{Dom}(\sigma_o) \setminus \text{Dom}(\sigma_t)$, $\delta_t(\text{id}) = e$ and $\langle e, \sigma \rangle \rightarrow_e^* \sigma_o(\text{id})$. ■

First of all, we introduce a lemma which states that we may substitute an expression e to a variable *id* inside a larger expression as long as e evaluates to $\sigma(id)$.

Lemma 4.4.1 (Sub-expression substitution)**.** *Let e be an expression containing the identifier id. Assume a formal environment δ such that $\delta(id) = e'$ with e' another expression. Moreover, assume that $\forall \sigma \in Mem$, $\exists cst \in \mathbb{E}$,*

$$\langle e', \sigma \rangle \rightarrow_e^* cst \ .$$

If, $\forall \sigma \in Mem$*,*

$$\exists r \in \mathbb{E}, \quad \langle e, \sigma[id \mapsto cst] \rangle \rightarrow_e^* r \ ,$$
$$\exists r' \in \mathbb{E}, \quad \langle \delta(e), \sigma \rangle \rightarrow_e^* r' \ ,$$

then $r = r'$*.* ■

Proof 1. *Given Lemma 4.4.1, the proof is done by induction on the structure of the expressions introduced in Equation* (3.8)*. We have to consider the following cases :*

- *If the expression is a constant, in other words,* $e \equiv cst$*, then we know that* $\delta(cst) = cst$*. Obviously, there is no change. In addition, we have* $\langle cst, \sigma \rangle \rightarrow_e cst$ *and* $\langle \delta(e), \sigma \rangle \rightarrow_e cst$*. Consequently, when we deal with constants the lemma is always correct.*
- *If the expression is a variable then* $e \equiv id$ *and we know by hypothesis that* $\delta(id) = e'$*. In this case, we have that* $\langle id, \sigma \rangle \rightarrow_e v$ *and, by hypothesis, we know that if* $\langle \delta(id), \sigma \rangle \rightarrow_e^* v'$ *then* $v = v'$*. So the lemma is also correct in this case.*
- *Finally, if we have an expression of the form* $e \equiv e_1 \odot e_2$ *where* $\odot \in \{+, -, \times, \div\}$ *then we apply the induction hypothesis on expressions* e_1 *and* e_2*. So, we have* $\delta(e_1) = e_1'$*,* $\langle e_1, \sigma[id \mapsto v_1] \rangle \rightarrow_e v_1'$ *and* $\langle \delta(e_1), \sigma \rangle \rightarrow_e v_1''$*. The same process is applied to* e_2 *and we know by induction hypothesis that* $v_1' = v_1''$ *and* $v_2' = v_2''$*. We conclude that* $v_1' \odot v_1'' = v_2' \odot v_2''$*. The lemma is correct for arithmetic operations.*

We introduce now a second lemma concerning the soundness of the transformation of arithmetic expressions. As presented in Section 4.2, we assume that a function $\rightsquigarrow$ transforming arithmetic expressions is given, see [56]. This function transforms an expression e_o into a more accurate expression e_t in the environment σ. Again, $\rightsquigarrow^*$ denotes the reflexive transitive closure of $\rightsquigarrow$.

Lemma 4.4.2 (Soundness of expressions transformation)**.** *Let* e_o *be an original arithmetic expression,* e_t *be the transformed expression. For all* $\sigma \in Mem$*, assume that* $\langle e_o, \sigma \rangle \rightsquigarrow^* e_t$*, i.e.,* e_o *is transformed into a mathematically equivalent expression* e_t*, such that*

$$\exists v_o \in \mathbb{E}, \quad \langle e_o, \sigma \rangle \rightarrow_e v_o \ ,$$
$$\exists v_t \in \mathbb{E}, \quad \langle e_t, \sigma \rangle \rightarrow_e v_t \ ,$$

then $v_t \sqsubseteq v_o$*.* ■

Proof 2. *Such a transformation is proved correct in [56].*

The following theorem relates the original commands and the transformed commands by using the relation $\prec_\vartheta$ introduced in Definition 4.4.2.

Theorem 4.4.3 (Soundness of command transformation)**.** *Let c_o be the original code, c_t be the transformed code, δ_o be the initial environment of the transformation, δ_t be the final environment of the transformation, then we have*

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

■

Proof 3. *The proof is by structural induction on commands. Hence, we consider each kind of expression and each rule of transformation of programs presented in Figure 3-2 which applies to the current kind of expression.*

Assignments *The first case is when the command c is an assignment, i.e., $c \equiv id = e$. In this case, we have two transformation rules, $(A1)$ and $(A2)$. Rule $(A1)$ consists in discarding an assignment when some conditions are satisfied while Rule $(A2)$ is used to substitute expressions within the assignment using the information already stored in the environment δ.*

- *Let us start with $(A1)$ which produces*

$$\langle id = e, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \mathsf{nop}, \delta_t, C, \beta\rangle \ .$$

We consider two cases :

1. *$id \equiv \vartheta$ is impossible because the variable id is in the black list β. In Rule $(A1)$, all the variables of the black list may not be discarded from the source code.*
2. *$id \not\equiv \vartheta$ then the new program is* nop *and the assignment is memorized in δ_t. That is to say, if i) $id \not\equiv \vartheta$ and ii) $id \notin \beta$ then $c_t \equiv \mathsf{nop}$ and $\delta_t = \delta[id \mapsto e]$.*
   In one hand, we have by composition of Rule $(R9)$ and $(R10)$ of the operational semantics, for all $\sigma \in Mem$

$$\exists \sigma_o \in Mem, \exists v \in \mathbb{E} \quad \langle id = e, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_o\rangle \quad with \quad \sigma_o = \sigma[id \mapsto v]$$

On the other hand, by discarding the previous assignment $id = e$, we have $c_t \equiv$ nop. The execution ends with $\langle \mathsf{nop}, \sigma_t\rangle$ where $\sigma = \sigma_t$. Then, since $\vartheta \neq id$, we have that $\sigma_o(\vartheta) = \sigma_t(\vartheta)$. In addition, we have $\langle id, \sigma\rangle \to_e v = \sigma_o(id)$. This proves that the theorem holds in this case, i.e., $\langle c_o, \delta_o\rangle \prec_\vartheta \langle c_t, \delta_t\rangle$.

– *The second part of the proof in the case of assignments is dedicated to demonstrate* $(A2)$. *First, if*
$$\langle id = e, \delta, C, \beta \rangle \Rightarrow_{\vartheta} \langle id = e'', \delta, C, \beta \rangle \ ,$$
then the original program c_o *is the assignment* $id = e$ *and the transformed program* c_t *is* $id = e''$, *such that*
$$e' = \delta(e), \quad \langle e', \sigma \rangle \rightsquigarrow^* e''.$$
In other words, the new expression e'' *is obtained by inlining in the expression* e *the variables stored in the environment* δ, *yielding an expression* e'. *Next,* e'' *is obtained from* e' *using the transformation of the expressions* $\rightsquigarrow^*$.
Let us assume that $\forall \sigma \in Mem$, *we have*
$$\exists v \in \mathbb{E}, \quad \langle id = e, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma[id \mapsto v] \rangle \quad \textit{with} \quad \langle e, \sigma \rangle \rightarrow_e^* v \ .$$
For the transformed program, assuming that $\forall \sigma \in Mem$, *we have*
$$\exists v' \in \mathbb{E}, \quad \langle id = e'', \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma[id \mapsto v'] \rangle \quad \textit{with} \quad \langle e'', \sigma \rangle \rightarrow_e^* v' \ .$$
According to Lemma 4.4.1, we kown that $\forall \sigma \in Mem$
$$\exists v \in \mathbb{E}, \quad \langle e, \sigma' \rangle \rightarrow_e^* v \ ,$$
$$\exists v' \in \mathbb{E}, \quad \langle e', \sigma \rangle \rightarrow_e^* v' \ ,$$
such that $v = v'$, *with* $\sigma' = \sigma[id \mapsto v]$ *for all* $id \in Dom(\delta)$ *and* $\langle \delta(id), \sigma \rangle \rightarrow_e^* v$.
According to Lemma 4.4.2, we know that if $\forall \sigma \in Mem$
$$\langle e', \sigma \rangle \rightsquigarrow^* e''$$
we have
$$\exists v' \in \mathbb{E}, \quad \langle e', \sigma \rangle \rightarrow_e^* v' \ ,$$
$$\exists v'' \in \mathbb{E}, \quad \langle e'', \sigma \rangle \rightarrow_e^* v'' \ ,$$
and
$$v'' \sqsubseteq v' = v \ .$$

In addition, we know that $\delta_o = \delta_t$. This demonstrates that $\langle c_t, \delta \rangle \prec_\vartheta \langle c_o, \delta \rangle$ for $(A2)$.

Sequences *For a sequence of commands, if one member of sequence of commands is* nop*, we have the following situations.*

- *According to Rule $(S1)$, if $c \equiv \mathsf{nop}; c_2$, then we have*

$$\left(\langle \mathsf{nop}; c_2, \delta, C, \beta \rangle \Rightarrow_\vartheta \langle c_2, \delta', C, \beta \rangle\right)$$

and $\delta = \delta'$. So, for all $\sigma \in Mem$, we have

$$\langle \mathsf{nop}; c_2, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma' \rangle \ ,$$
$$\langle c_2, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma'' \rangle \ ,$$

and $\sigma' = \sigma''$. Consequently,

$$\langle c_2, \delta, C, \beta \rangle \prec_\vartheta \langle \mathsf{nop}; c_2, \delta', C, \beta \rangle \ .$$

- *The case for Rule $(S2)$ is similar to the former one.*
- *For the general case $c \equiv c_1; c_2$, we know by induction hypothesis that :*

$$\left(\langle c_1, \delta, C\big[[\,]; c_2\big], \beta \rangle \Rightarrow_\vartheta \langle c_1', \delta', C\big[[\,]; c_2\big], \beta \rangle\right) \Longrightarrow \left(\langle c_1', \delta', C\big[[\,]; c_2\big], \beta \rangle \prec_\vartheta \langle c_1, \delta, C\big[[\,]; c_2\big], \beta \rangle\right) \ ,$$

$$\left(\langle c_2, \delta, C\big[c_1'; [\,]\big], \beta \rangle \Rightarrow_\vartheta \langle c_2', \delta', C\big[c_1'; [\,]\big], \beta \rangle\right) \Longrightarrow \left(\langle c_2', \delta', C\big[c_1'; [\,]\big], \beta \rangle \prec_\vartheta \langle c_2, \delta, C\big[c_1'; [\,]\big], \beta \rangle\right) \ ,$$

We know that $\forall \sigma \in Mem$, we have $\langle c_1, \sigma \rangle \rightarrow \langle c_1', \sigma \rangle$ and $\langle c_2, \sigma \rangle \rightarrow \langle c_2', \sigma \rangle$. According to Rule $(S3)$, we have

$$\langle c_1; c_2, \delta, C, \beta \rangle \Rightarrow_\vartheta \langle c_1'; c_2', \delta'', C, \beta \rangle \text{ with } \delta = \delta''.$$

Then for all $\sigma \in Mem$, we have that

$$\langle c_1; c_2, \sigma \rangle \rightarrow^* \sigma_2, \ \langle c_1'; c_2', \sigma \rangle \rightarrow^* \sigma_2' \text{ and } \sigma_2'(\vartheta) \sqsubseteq \sigma_2(\vartheta).$$

Consequently, $\langle c'_1; c'_2, \delta'_2, C, \beta\rangle \prec_\vartheta \langle c_1; c_2, \delta_1, C, \beta\rangle$.

Conditionals *The next case concerns the conditional statement* $c \equiv \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$. *We have to demonstrate the correctness of the transformation for the four cases corresponding to rules* $(C1)$ *to* $(C4)$ *introduced in Figure 4-1. If the condition is statically known then :*

- *If the condition e is always evaluated to* true*, then by induction hypothesis we write*

$$\langle c_1, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'_1, \delta', C, \beta\rangle \ ,$$

  and, for all $\sigma \in Mem$ *we have*

$$\langle c_1, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_1\rangle \ ,$$
$$\langle c'_1, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma'_1\rangle \ ,$$

  and $\sigma'_1(\vartheta) \sqsubseteq \sigma_1(\vartheta)$.
  By using the Rule $(C1)$*, we have that*

$$\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'_1, \delta', C, \beta\rangle$$

  and $\delta = \delta'$*. Consequently, we deduce that for all* σ*, that*

$$\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma_1\rangle \ ,$$
$$\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c'_1\ \mathsf{else}\ c'_2, \sigma\rangle \to^* \langle \mathsf{nop}, \sigma'_1\rangle \ ,$$

  and $\sigma'_1(\vartheta) \sqsubseteq \sigma_1(\vartheta)$*. Thus it results that :*

$$\langle c'_1, \delta', C, \beta\rangle \prec_\vartheta \langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \ .$$

- *If the condition e is always* false*, then we execute only the else branch. This case is similar to the previous one.*

Otherwise :

- *if Var(e)* $\cap$ *Dom*(δ) $\neq \emptyset$ *then we use* $(C3)$ *and by induction hypothesis, we have :*

$$\langle c_1, \delta, C, \beta \rangle \Rightarrow_\vartheta \langle c'_1, \delta', C, \beta \rangle \ ,$$
$$\langle c_2, \delta, C, \beta \rangle \Rightarrow_\vartheta \langle c'_2, \delta', C, \beta \rangle \ ,$$

And, for all $\sigma \in Mem$ *we have*

$$\langle c_1, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma_1 \rangle \ , \langle c'_1, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma'_1 \rangle \ ,$$
$$\langle c_2, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma_2 \rangle \ , \langle c'_2, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma'_2 \rangle \ ,$$

and $\sigma'_1(\vartheta) \sqsubseteq \sigma_1(\vartheta)$ *and* $\sigma'_2(\vartheta) \sqsubseteq \sigma_2(\vartheta)$. *In the program transformation, we transform* c_1 *and* c_2 *as indicated above and, at the end,* $\delta' = \delta'_1 \cup \delta'_2$. *In the operational semantics, we execute either* $\langle c'_1, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma'_1 \rangle$ *instead of* $\langle c_1, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma_1 \rangle$ *or* $\langle c'_2, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma'_2 \rangle$ *instead of* $\langle c_2, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma_2 \rangle$. *In any case,*

$$\sigma'_1(\vartheta) \sqsubseteq \sigma_1(\vartheta) \text{ and } \sigma'_2(\vartheta) \sqsubseteq \sigma_2(\vartheta) \ .$$

Consequently, since $\delta' = \delta'_1 \cup \delta'_2$,

$$\langle \mathit{if}_\Phi \quad e \ \mathsf{then} \ c'_1 \ \mathsf{else} \ c'_2, \delta', C, \beta \rangle \ \prec_\vartheta \ \langle \mathit{if}_\Phi \quad e \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_2, \delta, C, \beta \rangle \ . \quad (4.18)$$

- *In the last case, let* $c \equiv \mathit{if}_\Phi \ e \ \mathsf{then} \ c_1 \ \mathsf{else} \ c_2$. *If*

$$\langle c', \delta', C, \beta \rangle \prec_\vartheta \langle c, \delta, C, \beta \rangle \ ,$$

and if we add the same assignments within a command c_a *at the beginning of* c *and* c' *then necessarily,* $\langle c_a; c', \delta' \rangle \prec_\vartheta \langle c_a; c, \delta \rangle$.

While Loop *The last kind of transformation rules concerns the while loop* $c \equiv \mathsf{while}_\Phi \ e \ \mathsf{do} \ c$. *The rules* $(W1)$ *and* $(W2)$ *are similar to the transformation rules* $(C3)$ *and* $(C4)$ *for conditionals. The most important rule is* $(W1)$.

- *if Var(e)* $\cap$ *Dom*(δ) $\neq \emptyset$ *then by induction hypothesis, we have :*

$$\langle c, \delta, C, \beta \rangle \Rightarrow_\vartheta \langle c', \delta', C, \beta \rangle \ ,$$

and, for all $\sigma \in Mem$ we have

$$\langle c, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma' \rangle \ ,$$
$$\langle c', \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma'' \rangle \ ,$$

and $\sigma'(\vartheta) \sqsubseteq \sigma(\vartheta)$. We transform c as in the case of conditionals. In the operational semantics, we execute repeatedly $\langle c', \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma'' \rangle$ instead of $\langle c, \sigma \rangle \rightarrow^* \langle \mathsf{nop}, \sigma' \rangle$ as long as the condition e is true. Then at each iteration σ, and $\sigma''(\vartheta) \sqsubseteq \sigma'(\vartheta)$. Consequently, by transitivity of $\sqsubseteq$ we obtain that

$$\langle while_\Phi \ e \ do \ c', \delta', C, \beta \rangle \prec_\vartheta \langle while_\Phi \ e \ do \ c, \delta, C, \beta \rangle \ .$$

We give hereafter a general case of Theorem 4.4.3.

Theorem 4.4.4 (Multi-steps transformation)**.** *We assume that c_o is the original program, c_t is the transformed program and ϑ is the* target variable *to be improved. Assume also that $\langle c_o, \delta_o, C, \beta \rangle \Rightarrow^*_\vartheta \langle c_t, \delta_t, C, \beta \rangle$, then*

$$\langle c_t, \delta_t, C, \beta \rangle \prec_\vartheta \langle c_o, \delta_o, C, \beta \rangle \ .$$

■

Proof 4. *By induction on the length of the transformation (number of application of $\Rightarrow_\vartheta$).*

In the following, we extend our definitions and theorems to the abstract semantics introduced in Section 3.4.

Definition 4.4.3 (Abstract Comparison of Expressions)**.** Let $v_1^\sharp = (x_1^\sharp, \mu_1^\sharp) \in \mathbb{E}^\sharp$ and $v_2^\sharp = (x_2^\sharp, \mu_2^\sharp) \in \mathbb{E}^\sharp$. We say that $v_1^\sharp$ is more accurate than $v_2^\sharp$, denoted by $v_1^\sharp \sqsubseteq^\sharp v_2^\sharp$ iff $x_1^\sharp +^\sharp \mu_1^\sharp \subseteq x_2^\sharp +^\sharp \mu_2^\sharp$ and

$$\max(|\underline{\mu_1}|, |\overline{\mu_1}|) \leq \max(|\underline{\mu_2}|, |\overline{\mu_2}|) \ ,$$

where $\mu_1^\sharp = [\underline{\mu_1}, \overline{\mu_1}]$ and $\mu_2^\sharp = [\underline{\mu_2}, \overline{\mu_2}]$. ■

Abstract environments of values are defined, which map identifiers to abstract values. Let $\mathscr{V}$ be a set of identifiers and let $\sigma^\sharp \in \mathsf{Mem}^\sharp$ be an environment that map variables of $\mathscr{V}$ to abstract values

$$\sigma^\sharp : \mathscr{V} \rightarrow \mathbb{E}^\sharp \ . \tag{4.19}$$

We assume given an abstract semantics of commands which uses states defined by a pair

of command and abstract memory :

$$\text{State}^\sharp = \{\langle c, \sigma^\sharp \rangle : c \in \text{Cmd}, \sigma^\sharp \in \text{Mem}^\sharp\} \quad . \tag{4.20}$$

The operational semantics maps abstract states to abstract states

$$\begin{array}{rcrcl} \rightarrow^\sharp & : & \text{State}^\sharp & \rightarrow & \text{State}^\sharp \\ & & \langle c, \sigma^\sharp \rangle & \mapsto & \langle c', \sigma'^\sharp \rangle \quad . \end{array} \tag{4.21}$$

Such operational semantics correspond to standard abstract interpretations used in static analyses [23, 48, 51, 65]. Definition 4.4.4 extends the comparison of commands to abstract environments.

Definition 4.4.4 (Abstract Comparison of Commands)**.** Let c_o and c_t be two commands, let δ_o and δ_t be two formal environments. Finally, let ϑ be the *target variable*. We say that

$$\langle c_t, \delta_t \rangle \prec^\sharp_\vartheta \langle c_o, \delta_o \rangle$$

if and only if for all $\sigma^\sharp \in \text{Mem}$,

$$\begin{array}{l} \exists \sigma^\sharp_o \in \text{Mem}, \langle c_o, \sigma^\sharp \rangle \rightarrow^{\sharp *} \langle \text{nop}, \sigma^\sharp_o \rangle \quad , \\ \exists \sigma^\sharp_t \in \text{Mem}, \langle c_t, \sigma^\sharp \rangle \rightarrow^{\sharp *} \langle \text{nop}, \sigma^\sharp_t \rangle \quad , \end{array}$$

such that :

- $\sigma^\sharp_t(\vartheta) \sqsubseteq^\sharp \sigma^\sharp_o(\vartheta)$,
- $\forall \text{id} \in \text{Dom}(\sigma^\sharp_o) \setminus \text{Dom}(\sigma^\sharp_t)$, $\delta_t(\text{id}) = e$ and

$$\langle e, \sigma^\sharp \rangle \rightarrow^{\sharp *}_e \sigma^\sharp_o(\text{id}) \quad .$$

■

Theorem 4.4.5 (Static Soundness of Commands Transformation)**.** *Let c_o be the original code, c_t be the transformed code, δ_o be the initial environment of the transformation, δ_t be the final environment of the transformation, then we have*

$$\left(\langle c_o, \delta_o, C, \beta \rangle \quad \Rightarrow_\vartheta \quad \langle c_t, \delta_t, C, \beta \rangle\right) \quad \Longrightarrow \quad \left(\langle c_t, \delta_t, C, \beta \rangle \quad \prec^\sharp_\vartheta \quad \langle c_o, \delta_o, C, \beta \rangle\right) \quad .$$

■

Obviously, Theorem 4.4.4 can be easily extended to the abstract case in the same way than Theorem 4.4.5. Beside this formal proof of correctness, to go further with this question, complementary work is carried out in this direction in order to certify and validate that the programs transformed by our tool are equivalent to the original programs by using the Coq proof assistant [55]. More precisely, we aim at generating certificates, written as formal proofs, stating that the generated programs are mathematically equivalent and more accurate than the source programs.

4.5 Conclusion

This chapter outlined the main contribution of our research work. It detailed how transforming pieces of code such as intraprocedural programs to improve the accuracy of their numerical computations. This transformation is based on a set of transformation rules which are defined in this part and have been implemented in a tool named Salsa. These rules have been detailed with the use of examples that illustrate their applications. we have presented the principles of our tool that automatically rewrites programs in order to improve their numerical accuracy. More precisely, we have shown how to perform intraprocedural rewritings of commands.

We have also detailed on an example of programs, the odometry, the process followed to automatically transform an original program and we have given the different steps needed to reach the optimized program from the program taken in the input.

In addition, we have introduced a proof that emphasizes the soundness of our intraprocedural transformation. In the rules of Figure 4-1, correctness conditions have been defined to guarantee that the dependencies are respected and to ensure the correctness of the rewritings in conditions and loops. A complete set of experimentations is described in chapters 6 and 7.

RÉSUMÉ CHAPITRE 5

Ce chapitre est destiné à la transformation interprocédurale de programmes pour améliorer leurs précision numérique. Pour avoir un outil plus complet, nous généralisons notre technique de manière à traiter les programmes contenant des fonctions. Dans un premier temps, nous donnons la syntaxe de nos fonctions et ensuite nous présentons les différentes règles de transformation nécessaires pour l'amélioration de la précision des programmes.
Dans cette partie, nous avons défini trois règles de transformation de fonctions. La première règle consiste à substituer l'appel de la fonction dans le programme principal par le corps de la fonction appelée. Par la suite, nous appliquons les règles de transformation intraprocédurale vues dans le chapitre précédent pour obtenir un programme plus précis. La deuxième règle de transformation des fonctions effectue une spécialisation des fonctions. Cette règle n'est utilisée que si nous avons un petit nombre d'appels à une grande fonction dans le programme original. L'idée est de passer les valeurs de la fonction quand la variabilité de l'intervalle est inférieure à 2^4. La variabilité est définie par la différence entre la borne supérieure et la borne inférieure de l'intervalle. Nous utilisons par la suite les règles de transformation intraprocédurale sur le programme transformé. La dernière règle de transformation de fonctions consiste à passer les paramètres d'une fonction en utilisant une expression. Cette règle peut être considérée comme une évaluation des paramètres de la fonction appelante. Lorsque nous appliquons cette règle, nous obtenons une nouvelle fonction dont ses paramètres sont les variables de l'expression de l'appel de la fonction. À la fin, nous appelons les règles de transformation intraprocédurale pour améliorer la précision numérique des calculs du programme optimisé. Ces trois règles de transformation sont illustrées via des exemples qui illustrent au mieux leurs utilisations.

Nous détaillons par la suite comment choisir entre ces trois règles de transformation. Pour ce faire, nous avons défini une fonction qui sélectionne la règle à appliquer sur le programme pour améliorer sa précision. Cette fonction contient une variable qui permet ou pas de substituer la fonction appelée dans la fonction appelante. Cette variable tient compte du nombre de lignes de la fonction appelée, du nombre d'appels de la fonction appelante ainsi la taille totale du programme. Si cette variable est inférieure à une constante donnée, nous appliquons la première règle de transformation de fonction. Sinon, nous appliquons la règle de spécialisation de fonction ou bien la règle d'évaluation des arguments de la fonction suivie de la règle de spécialisation de fonction.

CHAPITRE 5

INTERPROCEDURAL TRANSFORMATION

Contents

5.1 Introduction **79**

5.2 Transformation of Functions **80**

5.2.1 Inlining Functions 81

5.2.2 Specialization of Functions 81

5.2.3 Passing the Parameters of Functions 83

5.3 Choice of the Kind of Function Transformation **85**

5.4 Conclusion **85**

5.1 Introduction

The main contribution of this chapter is the interprocedural program transformation [80]. In this chapter, we aim at generalizing our techniques to cover interprocedural program transformations and, in particular, we focus on functions refactoring and specialization with respect to the values of arguments. We start by defining the syntax of our functions, and, then we describe the different rules needed to improve their accuracy in the program. We start by describing in details how to transform interprocedural programs. Then we define three different rules used to optimize the accuracy of the computations of programs, inlining of functions, function specialization and lazy evaluation of the arguments. In addi-

tion, we present an heuristic which helps to select the best transformation for each function. Experimental results are given.

5.2 Transformation of Functions

Since large codes necessarily contain several functions and procedures, compilers and transformation tools have to use interprocedural techniques [80] to adequately support such programs. Our interprocedural transformation rebuilds the program in order to make it more accurate. One weakness of the technique of Section 4.2 is that it cannot take a complete program, since it does not support functions. To cope with this, we have defined a set of transformation rules which are given in Figure 5-1.

Let us note that, in our transformation, we have no problem to operate with global or local variables because our programs are written in SSA form. It is also important to mention that, for our function transformation, we start by applying the interprocedural transformation rules described in Figure 5-1 and then the intraprocedural transformation rules presented in Figure 4-1 (see Chapter 4).

The general syntax of a programs is

$$\mathsf{Prog} \ni p ::= \mathsf{f} \mid p\,\mathsf{f}$$

$$\mathsf{f} ::= f(u) \quad \{ \quad c; \quad return\ v \quad \} \ . \tag{5.1}$$

In Equation (5.1), $f(u)$ $\{$ c; $return\ v$ $\}$ defines a function whose argument is u, whose body is the command c and which returns v.

Recall that a command c is defined by Equation (3.9) and detailled in Chapter 4. For the sake of simplicity, in our formal definitions, we only consider functions with one single argument. However, in our implementation we support function with many arguments. The generalization is straightforward.

For a sake of simplicity, in Equation (5.1) and Figure 5-1, we write $f(u)$ $\{c;$ $return\ v\}$ insteed of Type $f(\text{Type } u)$ $\{$ c; $return\ v\}$ with Type $\in \{Float, Double, \ldots\}$.

We assume that any program p has a function named *main* which is the first function called when executing p, and the returned variable v consists in the *target variable* at optimizing $\vartheta = \{v\}$ with $\vartheta \in \mathscr{V}$. Recall that $\mathscr{V}$ denotes the set of identifiers.

$$\frac{f(u)\{c; return\ v\}}{\langle z = f(e), \delta, C, \beta\rangle \rightarrow_{\vartheta} \langle u = e; c; z = v, \delta, C, \beta\rangle} \quad (F1)(Inline)$$

$$\frac{\gamma = [\![e]\!]^{\sharp}\sigma^{\sharp} \quad \sigma^{\sharp} = [\![C[c]]\!]^{\sharp}\iota^{\sharp} \quad g(a)\{c'; return\ v\} \quad f(u)\{c; return\ v\} \quad \langle c, \delta[u \mapsto \gamma], [], \{v\}\rangle \rightarrow_{v} \langle c', \delta, C, \beta\rangle}{\langle z = f(e), \delta, C, \beta\rangle \rightarrow_{\vartheta} \langle z = g(), \delta, C, \beta\rangle} \quad (F2)(Value)$$

$$\frac{f(u)\{c; return\ v\}\langle c, \delta[u \mapsto e], [], \{v\}\rangle \rightarrow_{z} \langle c', \delta, [], \{v\}\rangle \quad g(u)\{c'; return\ v\}}{\langle z = f(e), \delta, C, \beta\rangle \rightarrow_{\vartheta} \langle z = g(Var(e)), \delta, C, \beta\rangle} \quad (F3)(Formal)$$

FIGURE 5-1 – *Transformation rules used to deal with functions.*

5.2.1 Inlining Functions

The first rule $(F1)$ consists in inlining the body of the function into the calling function. This makes it possible to create larger expressions in the caller. Then the new program can be more optimized by calling the intraprocedural transformation rules previously seen in Section 4.2.

Example 5.2.1

Let us consider the following example to explain how Rule $(F1)$ for function transformation is applied. The original program contains a call to a function `callee()`. The principle here is to inline the body of the function `callee()` within the main function in the source code, in `caller()`. In the code of Listing 5.1, we present the original program, the new program after applying the interprocedural transformation and then by using the intraprocedural optimization, we obtain the final improved program.

5.2.2 Specialization of Functions

The second transformation rule $(F2)$ is mainly used when we deal with a small number of calls to a large function in the original program. The idea is to pass the values of the function when the variability of the interval is small (for example whenever it is less than 2^4). By variability, we mean that the distance between the lower bound and the higher bound

Listing 5.1 – Example of program transformed by our tool. Top left : The original program. Top right : The optimized program using the interprocedural transformation rule (F1). Bottom Right : The optimized program obtained by using the intraprocedural transformation rules.

```
assert x = [100.0, 200.0]
double caller(){
  y = (x * x) + 15.0 ;
  z = callee(y) ;
  return z ;
}

double callee(double u){
  v = (55.123 * u * u * u) + (12.453
    * u * u) + (239.078 * u) + 0.3 ;
  return v ;
}
```

$\xrightarrow[(1)]{Inter}$

```
assert x = [100.0, 200.0]
double caller(){
  y = (x * x) + 15.0 ;
  u = y ;
  v = (55.123 * u * u * u) + (12.453
    * u * u) + (239.078 * u) + 0.3 ;
  z = v ;
  return z ;
}
```

$\xrightarrow[(2)]{Intra}$

```
assert x = [100.0, 200.0]
double caller(){
  v = ((((239.078 * (15.0 + (x * x)))
    + 0.3) + ((12.452 * (15.0 + (x
    * x))) * (15.0 + (x * x))))
    + (((55.123 * (15.0 + (x * x)))
    * (15.0 + (x * x))) * (15.0
    + (x * x)))) ;
  z = v ;
  return z ;
}
```

of an interval is small. If the variability of the interval is smaller than a parameter ω then, we apply $(F2)$, we substitute the variable u of the function `f` by the value γ and we return $\texttt{callee}_y$ as mentioned in Figure 5-1. In practice, we choose $\omega = 2^4$. Next, the intraprocedural transformation rules of program seen in Section 4.2 are applied to the transformed program. The *target variable* to optimize is the variable z returned by the function. The new black list β' contains the variable z in addition to the original *target variable* ϑ.

Example 5.2.2

Let us illustrate the application of the second interprocedural program transformation rule $(F2)$ of functions. By applying $(F2)$, we substitute the parameters of the function `callee()` by the parameters of the call's function in the `caller()`. That means that we substitute the value of u of the function `callee()` by the value of y of the call function, so the new function is named $\texttt{callee}_y()$ without parameters. We say that we have passed the function `callee()` by values. Next, we apply on the transformed program the intraprocedural transformation rules of Section 4.2 to improve its accuracy as shown in Listing 5.2.

Listing 5.2 – Example of program transformed by our tool. Top left : The original program. Top right : The optimized program using the interprocedural transformation rule (F2). Bottom Right : The optimized program obtained by using the intraprocedural transformation rules.

```
assert a = [10.0, 20.0]
double caller(){
   x = 2.0 ;
   y = 3.0 * x + 9.0  ;
   z = callee(y) ;
   return z ;
}

double callee(double h){
   v = (a * a * h * h * h) + (a * h
      * h) + ((a * 0.5) * h) + 0.3 ;
   return v ;
}
```

$\xrightarrow[(1)]{Inter}$

```
assert a = [10.0, 20.0]
double caller(){
   x = 2.0 ;
   y = 15.0 ;
   z = callee_y() ;
   return z ;
}

double callee_y(){
   v = (a * a * 15.0 * 15.0 * 15.0)
      + (a * 15.0 * 15.0) + ((a * 0.5)
      * 15.0) + 0.3 ;
   return v ;
}
```

$\xrightarrow[(2)]{Intra}$

```
assert a = [10.0, 20.0]
double caller(){
   x = 2.0 ;
   y = 15.0 ;
   z = callee_y() ;
   return z ;
}

double callee_y(){
   v = ((0.5 * (15.0 * a)) + ((0.3 + ((a
      * 15.0) * 15.0)) + ((((a * a) * 15.0)
      * 15.0) * 15.0))) ;
   return v ;
}
```

Remark. In our case, the functions are specialized according to their call site, and they are left separated. In other words, several calls to Rule $(F2)$ for the same function `callee()` yield several specialized versions of `callee()`. In a future work, we aim at grouping them.

5.2.3 Passing the Parameters of Functions

The last rule $(F3)$ consists in passing the parameters of a given function `callee()` using an expression. It can be seen as a lazy evaluation of the parameters in the caller. By applying this rule, we obtain the new function $\texttt{callee}_y\texttt{()}$ whose parameters are the variables of the expressions of the call to the function. Then we rewrite the new function $\texttt{callee}_y\texttt{()}$ by using the intraprocedural transformation rules to optimize the numerical accuracy of the

computations.

Example 5.2.3

To understand the use of the rule $(F3)$, we give the following example. We apply to it the third interprocedural transformation rule presented in Figure 5-1. As we observe, we have replaced the parameter u of the called function `callee()` with the expression corresponding to y in the main function. We create a new function named $\texttt{callee}_y\texttt{()}$ with a new parameter x corresponding to the variable z used in the expression assigned to y. Next, we call the intraprocedural transformation tool to optimize the intermediary program, in other words, the body of the new function $\texttt{callee}_y\texttt{()}$, as shown in the code of Listing 5.3.

Listing 5.3 – Example of program transformed by our tool. Top left: The original program. Top right: The optimized program using the interprocedural transformation rule (F3). Bottom Right: The optimized program obtained by using the intraprocedural transformation rules.

```
assert a = [10.0, 20.0]
double caller(){
  x = 2.0 ;
  y = 15.0 * x - 1.0 ;
  z = callee(y) ;
  return z ;
}

double callee(double h){
  v = (a * a * h * h * h) + (a * h
    * h) + ((a * 0.5) * h) + 0.3 ;
  return v ;
}
```

$\xrightarrow[(1)]{Inter}$

```
assert a = [10.0, 20.0]
double caller(){
  x = 2.0 ;
  y = 15.0 * x - 1.0 ;
  z = callee_y(y) ;
  return z ;
}

double callee_y(double u){
  v = (a * a * u * u * u) + (a * u * u)
    + ((a * 0.5) * u) + 0.3 ;
  return v ;
}
```

$\xrightarrow[(2)]{Intra}$

```
assert a = [10., 20.]
double caller(){
  TMP_1 = 29. ;
  z = callee_y() ;
  return z ;
}

double callee_y(){
  TMP_1 = 29. ;
  v = (((a * (TMP_1 * TMP_1)) + (a * (a
    * ((TMP_1 * TMP_1) * TMP_1)))) + (0.3
    + ((a * 0.5) * TMP_1))) ;
  return v ;
}
```

5.3 Choice of the Kind of Function Transformation

We introduce here how to choose the interprocedural transformation rules that will be applied on the program in order to improve its numerical accuracy. First, we have defined a function named `RuleSelector` which selects the appropriate rule to be used. This function includes a variable named `inlineAllowed` that allows or not the inlining of the caller function within the callee function. In other words, it allows or not the use of the first interprocedural transformation rule $(F1)$. The `inlineAllowed` makes it possible to inline the function by computing the expression $(((sizeF * sizeCall)/totalSize)$, where

- $sizeF$ is the number of lines of the callee function,
- $sizeCall$ is the calls number of the caller function,
- $totalSize$ is the total lines number of the program.

If this expression is less than a defined coefficient `inlineFactor` then we inline the function, else the inligning is not allowed. In our case, the value of the coefficient `inlineFactor` is initialized at 5. Second, if the first interprocedural tranformation rule $(F1)$ can not be applied, we use either the function specialization rule $(F2)$ or the lazy evaluation of the function arguments rule $(F3)$ followed by the rule of specialization of function $(F2)$.

5.4 Conclusion

This chapter detailed how to transform interprocedural programs to improve the accuracy of their numerical computations. This transformation is based on a set of transformation rules which are defined in Figure 5-1 and which have been implemented in our tool *SALSA*. A strategy to choose between the rules $(F1)$ to $(F3)$ are also been described in Section 5.3. In the next chapters, in order to validate our tool, we have taken a set of representative programs taken from various fields of science and engineering. Also, we give the different experiment results obtained on Salsa to show the efficiency of our approach on the numerical accuracy improvement.

RÉSUMÉ CHAPITRE 6

Dans les chapitres précédents, nous avons montré comment transformer automatiquement des programmes basés sur l'arithmétique des nombres flottants. Les règles de transformation présentées dans le Chapitre 4 et le Chapitre 5 ont été implémentées dans un outil appelé *SALSA*. À son tour, *SALSA* appelle un autre outil, *SARDANA*, qui améliore la précision numérique des expressions arithmétiques. Ce chapitre est dédié à montrer l'efficacité de notre outil, *SALSA*, sur un ensemble d'exemples venant de différents domaines tel que les systèmes embarqués et les méthodes d'analyse numérique.
Une première application consiste à améliorer la précision numérique de programmes. Nous avons expérimenté notre implémentation sur un programme qui calcule la position d'un robot à deux roues (Odometry), l'algorithme PID, qui permet de maintenir une mesure à une certaine consigne, un système masse-ressort qui consiste à changer la position initiale d'une masse vers une position désirée, les méthodes de Runge-Kutta de deuxième et de quatrième ordre, la méthode des trapèzes ainsi la méthode de Newton. La précision numérique de tout ces programmes a été améliorée. Prenant par exemple le programme d'Odométrie, sa précision est améliorée de 21%. Notons que pour chaque programme considéré, nous donnons son code avant et après transformation.
Une autre application de notre outil consiste à étudier l'impact de l'amélioration de la précision numérique des programmes sur le format des variables utilisées (simple ou double précision). Par cette étude, nous offrons 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 de départ exécuté en double précision. Pour ce faire, nous avons choisi de calculer l'intégrale du polynôme $(x-2)^7$ en utilisant la méthode

de Simpson. Il est à noter que notre étude est restreinte à observer le comportement du polynôme autour de la racine, en d'autre termes, sur l'intervalle $[1.9, 2.1]$ où le polynôme s'évalue très mal dans l'arithmétique des nombres flottants. Dans notre cas, nous avons comparé trois programmes, le premier code source écrit en simple précision (32 bits), un deuxième code source en double précision (64 bits) et le troisième programme qui est celui transformé en simple précision (32 bits). Nous avons observé que le programme transformé en simple précision est très proche de celui de départ en double précision. L'avantage de cette comparaison est de permettre à l'utilisateur de travailler avec un format plus compact sans perdre beaucoup d'informations en lui permettant ainsi d'économiser de l'espace mémoire, de la bande passante et du temps de calcul.

CHAPITRE 6 EXPERIMENTS : NUMERICAL ACCURACY

Contents

6.1 Introduction 90

6.2 Overview of *SALSA* 90

6.3 Improving the Accuracy of Intraprocedural Programs 92

6.3.1 Odometry 93

6.3.2 PID Controller 93

6.3.3 Lead-Lag System 95

6.3.4 Runge-Kutta Methods 96

6.3.5 The Trapezoidal Rule 99

6.3.6 Rocket Trajectory Simulation 99

6.3.7 Experimental Results 103

6.4 Improving the Accuracy of Interprocedural Programs 104

6.4.1 Odometry 105

6.4.2 Newton-Raphson's Method 105

6.4.3 Runge-Kutta Method 107

6.4.4 Simpson's Method 109

6.4.5 Experimental Results 109

6.5 Optimizing the Data-Type Formats of Variables 110

6.5.1 Numerical Integration Method 112

6.5.2 Experimental Results 112
6.6 Conclusion 117

6.1 Introduction

We have shown in Chapter 4 and Chapter 5 how to optimize automatically intraprocedural and interprocedural programs [29] based on floating-point arithmetic. A tool named *SALSA* has been developed which implements the rules of Figure 4-1 and Figure 5-1. *SALSA* calls another tool, *SARDANA*, to improve the numerical accuracy of arithmetic expression. *SARDANA* uses the APEG introduced in Section 3.6 (see Chapter 3). We have experimented *SALSA* to improve the numerical accuracy of small control command programs (*e.g.* PID and lead-lag controllers) and numerical procedures (trapeze rule and Runge-Kutta methods [29]). We have also demonstrated the efficiency of our tool to optimize slightly larger codes like the simulation of a rocket trajectory in space. This program contains about 100 lines of code [30].
Note that all the example programs are optimized. This is not surprising since the original codes are direct implementations of the mathematical formulæ and have not been written for the floating-point arithmetic. However, generally speaking, our tool never returns a code whose accuracy in the worst case is worst than the original program. In the worst case, it returns the original program without optimization.
This chapter starts by introducing our tool *SALSA*. It details its architecture and describes the process followed by *SALSA* to improve the accuracy of programs given in inputs. After this short presentation of *SALSA*, we give the first application of our programs transformation. It shows the efficiency of our tool to improve the numerical accuracy of computations on a suite of examples coming from different domains.

6.2 Overview of *SALSA*

- *SALSA* [29, 31] is the tool developed during this PhD to implements the ideas of sections 4.2 and 5.2. *SALSA* aims at improving the numerical accuracy of programs written in an imperative language. It is based on static analysis by abstract interpretation in order to bound maximum absolute error, producing a sound over-approximation of the maximum error. Written in `Caml` language, our tool contains a parser and a static analyzer to infer

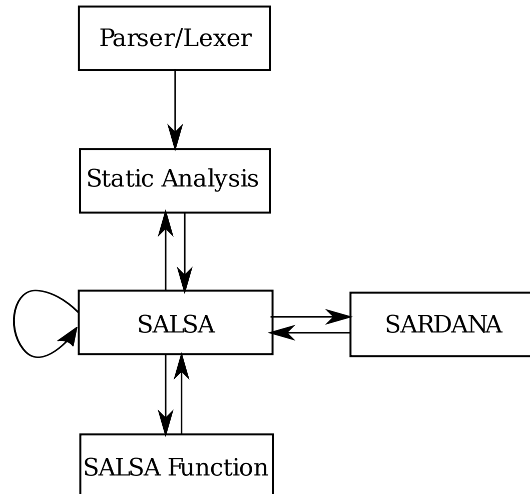


FIGURE 6-1 – *Architecture of* SALSA.

safe ranges and error bounds for each variable at each control point of the program.
SALSA is implemented with respect to two main points :

- First, to transform pieces of codes, we have implemented the set of intraprocedural transformation rules introduced in Section 4.2. The interprocedural transformation rules have been implemented separately in another module which calls the intraprocedural module when needed as described in Figure 6-1.
- Second, to transform arithmetic expressions, we have integrated the *SARDANA* tool described in [56]. Again, the intraprocedural tool calls *SARDANA* when needed.

In practice, *SALSA* works as follows.

- *SALSA* receives as input a file corresponding to the program that we aim at improving.
- The program must follows the syntax accepted by *SALSA*. It must specify the *target variable*, ϑ, that will be optimized. The syntax of our programs is given in Figure 6-2.
- *SALSA* analyses the input program.
- *SALSA* transforms this program by applying the intraprocedural and/or interprocedural transformation rules. It builds large expressions.
- *SALSA* calls *SARDANA* to rewrite the arithmetic expressions. *SARDANA* re-parses the given arithmetic expression in different ways in order to return a more accurate expression.

– When transforming programs, we may have larger expressions that *SALSA* slices at a defined level of the syntactic tree and we assign them to intermediary variables named *TMP*.
– *SALSA* re-analyses the transformed programs recursively until the accuracy is no longer improved (see Figure 6-1).
– *SALSA* returns as output a file containing the transformed program with best accuracy.

SALSA has been evaluated on a suite of control algorithms and numerical methods to improve their numerical accuracy of computations. The experiment results are discussing hereafter.

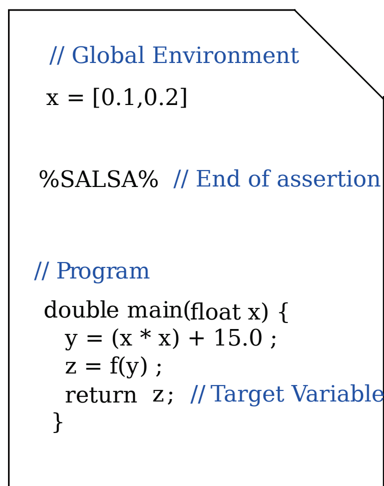


FIGURE 6-2 – SALSA *files.*

6.3 Improving the Accuracy of Intraprocedural Programs

As a step toward ensuring the efficiency of our tool and in order to perform experiments with it, we consider here a set of examples in the domains of robotics, avionics, numerical analysis, etc. First, we briefly describe each of these programs and we give their listing before and after transformation. Their accuracy is then discussed.

6.3.1 Odometry

We aim at transforming the odometry program introduced in Section 4.3.1 (see Listing 4.1) into a better program which improves the numerical accuracy of the computed position of the robot. The speed of the left wheel is assumed to belong to an interval of $[0.52, 0.53]$ radians per second ($\frac{\pi}{6} \approx 0.523598$) so that the program is optimized for a range of values of s_l and not only for a single value. Our prototype develops and simplifies the expressions Δ_d, `cos` and `sin` and then inline them within the loop, in x and y. In addition, it creates new intermediary variables, called `TMP`, in order to avoid to have too large expressions.

It	x_o (Original code)	x_t (Transformed code)
1	8.681698	8.444116
2	17.038230	16.589474
3	24.756744	24.147995
4	31.549016	30.852965
5	37.163761	36.469708
6	41.398951	40.806275
7	44.114126	43.724118
8	45.242707	45.148775

TABLE 6.1 – *Values of x before and after transformation of Odometry program at the first iterations.*

Using our tool, we obtain the final program given in Listing 4.5. If we compare the resulting values x_o and x_t of the original and transformed Odometry programs, we observe that the transformation leads to a significant difference in the accuracy of the computations, as shown in Table 6.1. The results show an important difference on the third or even on the second digit of the decimal values of the result. The difference in the computed trajectory of the robot is shown in Figure 6-3. In addition, the execution time required by this program to be transformed is $0.038s$.

6.3.2 PID Controller

The second example consists in the PID Controller previously introduced in Section 2.2.1. In this case, the transformation is done automatically using our tool not by hand as discussed in Chapter 2. We assume that the measure m belongs to the interval [4.5,9.0] and it corresponds

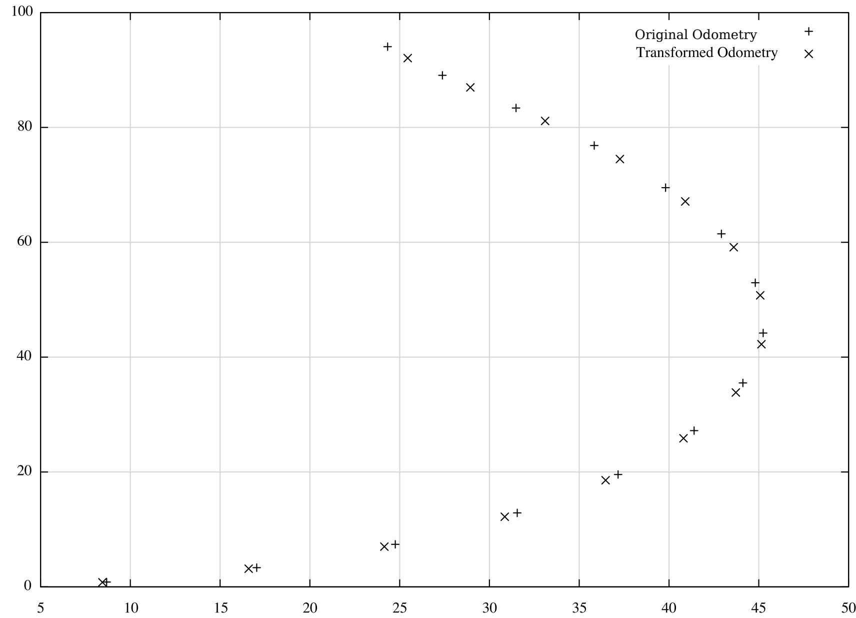


FIGURE 6-3 – *Computed trajectories by the original and the transformed odometry programs.*

to the *target variable* that we aim at improving, $\vartheta = \{m\}$. The program corresponding to the implementation of the PID Controller is given in Listing 6.1.

Listing 6.1 – Listing of the original PID program.

```
m = [4.5,9.0]; kp = 9.4514; ki = 0.69006; kd = 2.8454; t = 0.0; i = 0.0; eold = 0.0;
c = 5.0; dt = 0.2; invdt = 5.0;
while true do {
  e = c − m;
  p = kp ∗ e;
  i =  i + ((ki ∗ dt) ∗ e);
  d = ((kd ∗ invdt) ∗ (e − eold));
  r = ((p + i) + d);
  m = m + (0.01 ∗ r);
  eold = e;
  t = t + dt;
}
```

We optimize the original PID program shown in Listing 6.1, in order to improve its numerical accuracy, by applying the transformation rules presented previously on Figure 4-1 to it. In this case, our tool has simplified and developed expressions and then inlined them into other expressions. Listing 6.2 shows the optimized program. Note that the execution time needed by the PID program to be improved using our tool is $0.075s$.

Listing 6.2 – Listing of the transformed PID program.

```
m = [4.5,9.0];   t = 0.0;  i = 0.0;
while true do {
   i = (i + (0.138012 * (5.0 - m)));
   eold = (5.0 - m);
   m = (m + (0.01 * ((((5.0 - m) * 9.4514) + i) + (((5.0 - m) - eold) * 14.227))));
   t = t + 0.2;
}
```

6.3.3 Lead-Lag System

Our next example is a dynamical system. This system includes a single mass and a single spring and is governed by an automatically synthesized controller [43] which tries to move the mass from the initial position y to the desired one y_d as illustrated in Figure 6-4. The main variables in this algorithm are :

- x_c is of the discrete-time controller state,
- y_c is the bounded output tracking error or the saturation mechanism,
- u is the mechanical system input.

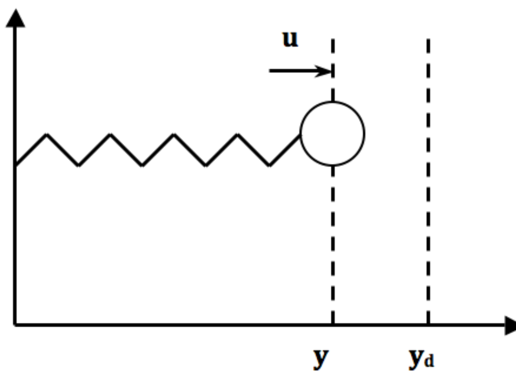


FIGURE 6-4 – *The lead-lag system.*

We assume that the position y of the mass m belongs to the interval [2.1,17.9]. The equations modeling the system, as described in [43], are given in Equation (6.1).

$$
\begin{aligned}
yc &= \mathsf{max}(\mathsf{min}(y - yd, 1), -1) \;\; ; \\
u &= Cc \times xc + Dc \times yc \;\; ; \\
xc &= Ac \times xc + Bc \times yc \;\; .
\end{aligned}
\tag{6.1}
$$

Listing 6.3 gives the code listing of the original Lead-Lag program. In this program, we are interesting in improving the *target variable* $xc1$. By optimizing the numerical accuracy of this program, we obtain the transformed program given in Listing 6.4 in a time of $0.155s$.

Listing 6.3 – Listing of the original Lead-Lag program.

```
y = [2.1,17.9]; xc0 = 0.0; xc1 = 0.0; yd = 5.0; Ac11 = 1.0; Bc0 = 1.0;
Bc1 = 0.0; Cc0 = 564.48; Ac00 = 0.499;  Ac01 = -0.05; Ac10 = 0.01;
Cc1 = 0.0; Dc = -1280.0; t = 0.0;
while(t < 5.0) do {
  yc = (y - yd);
  if(yc < -1.0) { yc = -1.0; }
  if(1.0 < yc) { yc = 1.0; }
  xc0 = (Ac00 * xc0) + (Ac01 * xc1) + (Bc0 * yc);
  xc1 = (Ac10 * xc0) + (Ac11 * xc1) + (Bc1 * yc);
  u = (Cc0 * xc0) + (Cc1 * xc1) + (Dc * yc);
  t = (t + 0.1);
}
```

Listing 6.4 – Listing of the transformed Lead-Lag program.

```
y = [2.1,17.9]; t = 0.0; xc1 = 0.0; xc0 = 0.0;
while(t < 5.0) do {
  yc = (-5.0+y);
  if(yc < -1.0) { yc = -1.0; }
  if(1.0 < yc)  { yc = 1.0; }
  u = (((564.48 * xc0)+(0.0 * xc1))+(-1280.0 * yc));
  xc0 = (((-0.05 * xc1)+(1.0 * yc))+(0.499 * xc0));
  xc1 = (((0.01 * xc0)+(0.0 * yc))+(1.0 * xc1));
  t = (t + 0.1);
}
```

6.3.4 Runge-Kutta Methods

This example concerns two implementations of Runge-Kutta methods [58]. We consider an order 2 and an order 4 method.

Second Order Runge-Kutta Method

The order 2 method integrates a differential equation whose solution is $y(x)$. The second order method uses the derivative at point y in order to find the intermediary point. Then, it uses this intermediary point to have the next value of the function. The derivative of $y(x)$ at the points x_i and $x_i + \frac{h}{2}$ are :

$$k_1 = (\frac{dy}{dx}) = h \times f(x_i, y_i) \ ,$$

$$k_2 = (\frac{dy}{dx}) = h \times f(x_i + \frac{h}{2}, y_i + \frac{h}{2}) \ . \tag{6.2}$$

Finally, we have $y_{n+1} = y_n + k_2 + O(h^3)$. Let us consider the function $f(y) = (100.1 - y)^2$ that we compute its integral $\int_{-10.0}^{10.} f(y)dy$ and a step $h = 0.1$. Initially, we assume that $y_0 \in [-10.1, 10.1]$ and the variable to be improved is $\vartheta = \{yn+1\}$. The implementations of the original and transformed second order Runge-Kutta method are given respectively in Listing 6.5 and Listing 6.6. The execution time corresponding to the transformation of second order Runge-Kutta program is $0.049s$.

Listing 6.5 – Listing of the original second order Runge-Kutta program.

```
yn = [-10.1,10.1]; t = 0.0; k = 1.2; c = 100.1; h = 0.1;
while(t < 1.0) do {
  k1 = k * (c - yn) * (c - yn);
  k2 = k * (c - (yn + (0.5 * h * k1))) * (c - (yn + (0.5 * h * k1)));
  yn+1 = yn + h * k2;
  yn = yn+1;
  t = t + h;
 }
```

Listing 6.6 – Listing of the transformed second order Runge-Kutta program.

```
yn = [-10.1,10.1]; t = 0.0;
while(t < 1.0) do {
  yn+1 = (yn + (( 1.2 * (10.1 - ((((1.2 * (10.1 - yn)) * (10.1 - yn))
        * 0.005) + yn))) * (10.1 - ((((1.2 * (10.1 - yn)) * (10.1 - yn))
        * 0.005) + yn))));
  t = t +  0.01;
  yn = yn+1;
}
```

Fourth Order Runge-Kutta Method

For the order 4 method, we obtain as final formula :

$$y_{i+1} = y_1 + \tfrac{1}{6}\left[k_1 + 2 \times k_2 + 2 \times k_3 + k_4\right] \times h \ . \tag{6.3}$$

Listing 6.7 – Listing of the original fourth order Runge-Kutta program.

```
yn = [-10.1,10.1]; t = 0.0; k = 1.2; c = 100.1;
h = 0.1;
while(t < 1.0) do {
  k1 = (k * (c - yn)) * (c - yn);
  k2 = (k * (c - (yn + ((0.5 * h) * k1)))) * (c - (yn + ((0.5 * h) * k1)));
  k3 = (k * (c - (yn + ((0.5 * h) * k2)))) * (c - (yn + ((0.5 * h) * k2)));
  k4 = (k * (c - (yn + (h * k3)))) * (c - (yn + (h * k3)));
  yn+1 = yn + ((1/6 * h) * (((k1 + (2.0 * k2)) + (2.0 * k3)) + k4));
  t = (t + h);
 }
```

with

$$k_1 = h \times f(x_i, y_i) \ ,$$

$$k_2 = h \times f(x_i + \frac{h}{2}, y_i + \frac{h}{2}) \ ,$$

$$k_3 = h \times f(x_i + \frac{h}{2}, y_i + \frac{h}{2}) \ ,$$

$$k_4 = h \times f(x_i + \frac{h}{2}, y_i + \frac{h}{2}) \ .$$

The function f needs to be evaluated four times to improve its accuracy. The listing of the

Listing 6.8 – Listing of the transformed fourth order Runge-Kutta program.

```
yn = [-10.1,10.1]; t = 0.0;
while(t < 1.0) do {
  TMP_7 = (1.2 * (100.099 - yn));
  TMP_8 = (100.099 - yn);
  TMP_13 = (1.2 * (100.099 - (yn + (0.05 * ((1.2 * (100.099 - (yn + (0.05 * (TMP_7
          * TMP_8))))) * (100.099 - (yn + (0.05 * ((1.2 * TMP_8) * (100.099
          - yn))))))))));
  TMP_14 = (100.099 - (yn + (0.05 * ((1.2 * (100.099 - (yn + (0.05 *(TMP_7
          * TMP_8))))) * (100.099 - (yn + (0.05 * ((1.2 * TMP_8) * (100.099 - yn));
  TMP_18 = (yn + (0.05 * ((1.2 * (100.099 - (yn + (0.05 *(TMP_7 * TMP_8)))))
          * (100.099 - (yn + (0.05 * ((1.2 * TMP_8) * (100.099 -yn))))))));
  TMP_28 = ((1.2 * (100.099 - (yn + (0.05 * (TMP_7 * TMP_8))))) *(100.099 - (yn
          + (0.05 * ((1.2 * TMP_8) * (100.099 - yn))))));
  TMP_38 = ((TMP_14 * TMP_13) * 0.1) + yn;
  TMP_40 = 0.1 * ((1.2 * TMP_14) * (100.099 - TMP_18));
  yn+1 = (yn + (0.016666667 * ((((TMP_7 * TMP_8) + (2.0 * TMP_28)) + (2.0
       * (TMP_13 * TMP_14))) + ((1.2 * (100.099 - TMP_38)) * (100.099
       - (yn + TMP_40))))));
  t = (t + 0.1);
}
```

original program is presented in Listing 6.7.

By applying a sequence of transformation rules to the original Runge-Kutta code, we obtain a

program far more accurate than the original one. In the optimized program, given in Listing 6.8, the transformation improves the variable $\vartheta = \{y_{n+1}\}$ in a time of $0.073s$.

6.3.5 The Trapezoidal Rule

This example concerns an algorithm for the trapezoidal rule [58], well known in numerical analysis to approximate the definite integral $\int_a^b f(x)\,dx$. This trapezoidal rule works by approximating the region between x and $x+h$ under the graph of the function f as a trapezoid and calculates its area, for all points $x = a + i \cdot h$ with $0 \leq i < \frac{b-a}{h}$. Here, we compute the integral $\int_{0.25}^{5000} g(x)dx$ of some function :

$$g(x) = \frac{u}{0.7x^3 - 0.6x^2 + 0.9x - 0.2} \quad . \tag{6.4}$$

We assume that u is a user defined parameter in the range $[1.11, 2.22]$. In addition, we have unfold the body of the loop twice to obtain better results with our tool. The program performing the function $g(x)$ corresponding to Equation (6.4) is described in Listing 6.9.

Listing 6.9 – Listing of the original trapeze program.

```
u = [1.11, 2.22]; a = 0.25; b = 5000.0; n = 25.0;
r = 0.0; xa = 0.25; h = ((b - a) / n);
while (xa <  5000.0) do {
  xb = (xa + h) ;
  if (xb > 5000.0) {
    xb = 5000.0 ;
    gxa = (u / ((((((0.7 * xa) * xa) * xa) - ((0.6 * xa) * xa)) + (0.9 * xa)) - 0.2));
    gxb = (u / ((((((0.7 * xb) * xb) * xb) - ((0.6 * xb) * xb)) + (0.9 * xb)) - 0.2));
    r = (r + (((gxb + gxa) * 0.5) * h));
    xa = (xa + h);
    gxa = gxb ;
  }
}
```

Once the trapezoidal rule program is given to our tool, many transformation rules would be applied on it to improve its accuracy. Note that the variable at optimizing is $\vartheta = \{r\}$. Listing 6.10 corresponds to the program generated after optimization. This program is transformed in a time of $0.440s$.

6.3.6 Rocket Trajectory Simulation

Our tool demonstrated its efficiency on small programs by improving their numerical accuracy. Among other examples, we have taken a larger example that contains more than one hundred

Listing 6.10 – Listing of the transformed trapeze program.

```
u = [1.11, 2.22]; xa = 0.25; r = 0.0;
while (xa < 5000.0) do {
  TMP_1 = (0.7 * (xa + 199.99));
  TMP_2 = (xa + 199.99);
  TMP_9 = ((((0.7 * xa) * xa) * xa) - ((0.6 * xa) * xa)) + (0.9 * xa);
  TMP_11= (((199.99 + xa) * (TMP_2 * TMP_1)) - ((199.99 + xa) * (TMP_2 * 0.6)))
        + (0.9 * TMP_2);
  r = (r + ((((u / (TMP_11 - 0.2)) + (u / (TMP_9 - 0.2))) * 0.5) * 199.99));
  xa = (xa + 199.99);
}
```

lines of code [30]. The example computes the positions of a rocket and a satellite in space. It consists of simulating their trajectories around the Earth using the Cartesian and polar systems, in order to project the gravitational forces in the system composed of the Earth, the rocket and the satellite. Note that the coordinates of the satellite u_i and of the rocket w_i, $1 \leq i \leq 4$ are computed by Euler's method.

The program corresponding to this example is given in Listing 6.11. The `else` branch is similar to the `then` branch at the difference that w'_2 and w'_4 are computed without the expression $A \cdot w_i/(M_f - A \cdot t) \cdot dt$. At the end of the loop, variables are updated, for example $u_1 = u'_1$, etc.

Constants are : the radius of the Earth $R = 6.4 \cdot 10^6\ m$, the gravity $G = 6.67428 \cdot 10^{-11}\ m^3 \cdot kg^{-1} \cdot s^{-2}$, the mass of the Earth $M_t = 5.9736 \cdot 10^{24}\ kg$, the mass of the rocket $M_f = 150000\ kg$ and the gas mass ejected by second $A = 140\ kg \cdot s^{-1}$. The release rate of the rocket v_l is $0.7 \cdot \sqrt{((G \cdot M_t)(D)}$ with $D = R + 4.0 \cdot 10^5\ m$ the distance between the rocket and the Earth. Other variables are set to 0.

Listing 6.11 illustrates the original rocket trajectory simulation program. Listing 6.12 shows the program that corresponds to the optimized rocket trajectory simulation. The execution time required by this transformation is $0.088s$ and the *target variable* is the position of the rocket in the space, $\vartheta = \{(x, y)\}$. The simulations of the trajectories of the rocket before and after optimization are given in Figure 6-5. Lastly, if we are interested to observe the behavior of both trajectories before and after being optimized, we can remark a significant difference between the two curves. This difference shows on how match we improve the numerical accuracy of this large program. Note that Figure 6-5 is obtained after 2.25 days of simulated time.

Remark. Note that in our program transformations, we choose to unfold the body of the loop when we deal with a smaller expressions. The loop unfolding can be applied on the program several times, until the expressions become larger, in other words, when the size of the expressions

Listing 6.11 – Listing of the original rocket trajectory simulation program.

```
u = [1.11, 2.22]; xa = 0.25; r = 0.0; Mf = 150000; R = 6.4 * 10e6; A = 140;
G = 6.67428 * 10e-11;
M_t = 5.9736 * 10e24;
D = R + 4.0 * 10e5;
v_l = 0.7 * sqrt((G * M_t)(D));
while (i < nbsteps) do {
 if (m_f > 0.0) {
    u'1 = u2 * dt + u1;
    u'3 = u4 * dt + u3;
    w'1 = w2 * dt + w1;
    w'3 = w4 * dt + w3;
    u'2 = -G * Mt / (u1 * u1) * dt + u1 * u4 * u4 * dt + u2;
    u'4 = -2.0 * u2 * u4 / u1 * dt + u4;
    w'2 = -G * Mt / (w1 * w1) * dt + w1 * w4 * w4 * dt + (A * w2)
         / (Mf - A * t) * dt + w2;
    w'4 = -2.0 * w2 * w4 / w1 * dt + A * w4 / (Mf - A * t) * dt + w4;
    m'f = mf - A * t ;
     t = t + dt;
 }
 else {
    u'1 = u2 * dt + u1;
    u'3 = u4 * dt + u3;
    w'1 = w2 * dt + w1;
    w'3 = w4 * dt + w3;
    u'2 = -G * Mt / (u1 * u1) * dt + u1 * u4 * u4 * dt + u2;
    u'4 = -2.0 * u2 * u4 / u1 * dt + u4;
    w'2 = -G * Mt / (w1 * w1) * dt + w1 * w4 * w4 * dt + w2;
    w'4 = -2.0 * w2 * w4 / w1 * dt + w4
    m'f = mf
     t = t + dt
 }
c=1.0-(w'3*u'3*0.5);
s=u'3-(u'3*u'3*u'3) / 0.166666667;
x=w'1*c;
y=w'1*s;
i=i+1.0;

u1=u'1;
u2=u'2;
u3=u'3;
u4=u'4;
w1=w'1;
w2=w'2;
w3=w'3;
w4=w'4;
mf=m'f;
}
```

Listing 6.12 – Listing of the transformed rocket trajectory simulation program.

```
u = [1.11, 2.22]; xa = 0.25; r = 0.0;
while (i < nbsteps) do {
  if (m_f > 0.0) {
    TMP_2 = (u1 * u1);
    TMP_4 = (59735.99e20 / (w1 * w1));
    TMP_10 = (140.0 * t);
    m'f = (mf + (t * (-140.0)));
    u'1 = (u1 + (u2 * 0.1));
    u'3 = (u3 + (u4 * 0.1));
    w'1 = (w1 + (w2 * 0.1));
    w'3 = (w3 + (w4 * 0.1));
    u'2 = ((-((0.66743e-10 * (59735.99e20 / TMP_2))* 0.1) + (((u1 * u4)* u4)* 0.1))+ u2);
    u'4 = (((-2.0 * (u2 * (u4 / u1))) * 0.1) + u4);
    w'2 = (((-((0.66743e(-10) * TMP_4) * 0.1)+ (((w1 * w4) * w4) *  0.1)) + (((140.0
        * w2) / (150000.0 - (140.0 * t))) * 0.1)) + w2);
    w'4 = (((-2.0 *  (w2 * (w4 / w1))) * 0.1) + ((140.0 * ((w4 / (150000.0
        - TMP_10)) * 0.1)) + w4));
    t = t + 0.1;
  }
  else {
    TMP_2 = (u1 * u1);
    TMP_14 = (w1 * w1);
    u'1 = (u1 + (u2 * 0.1));
    u'3 = (u3 + (u4 * 0.1));
    w'1 = (w1 + (w2 * 0.1));
    w'3 = (w3 + (w4 * 0.1));
    u'2 = ((-((0.66743e-10 * (59735.99e20 / TMP_2))* 0.1) + (((u1 * u4)* u4)* 0.1))+ u2);
    u'4 = (((-2.0 * (u2 * (u4 / u1))) * 0.1) + u4);
    w2_i = ((-((0.66743e-10 * (59735.99e20 / TMP_14)) * 0.1) + (((w1 * w4)* w4) * 0.1)) + w2);
    w4_i = (((-2.0 * (w2 * (w4 / w1))) * 0.1) + w4);
    t = t + 0.1;
  }
  c = (1.0 - ((u3 + (u4 * 0.1)) * (0.5 * (u3 + (u4 * 0.1)))));
  s = ((u3 + (u4 * 0.1)) - (((u3 + (u4 * 0.1)) * ((u3 + (u4 * 0.1)) * (u3 + (u4 * 0.1))))
    / 0.166666667));
  x = (c * (u1 + (u2 * 0.1)));
  y = (s * (u1 + (u2 * 0.1)));
  i = i + 1.0;
u1=u'1;
u2=u'2;
u3=u'3;
u4=u'4;
w1=w'1;
w2=w'2;
w3=w'3;
w4=w'4;
mf=m'f;
}
```

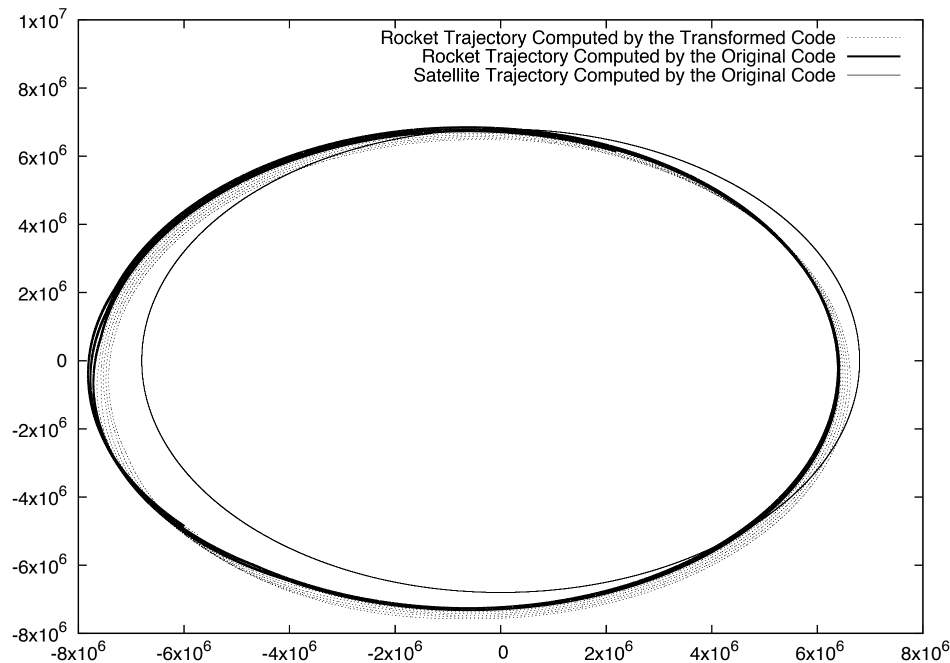


FIGURE 6-5 – *Difference between the original and the transformed trajectories of the rocket.*

is more than 10 operations for example. In a future work, the choice to unfold the body of the loop will be automatized and defined by a transformation rules.

6.3.7 Experimental Results

We show hereafter how our tool improves the accuracy of programs. Table 6.2 compares the initial and the new error of each of the programs presented in this section and shows as well how much our tool improves the numerical accuracy of these programs. For example, if we take the case of `odometry`, we observe that we optimize it by 21.43%. If we compare the implementation of Runge-Kutta method, we remark that the order four methods is improved of 15.87%. The Lead-Lag system is optimized by 19.96%. The improvement of the error is given in Table 6.2, where s is the slice size, *i.e.*, the parameter defining at which height of the syntactic tree we cut the expressions.

In addition to improving the accuracy of the programs described above, we evaluated our tool

Code	Initial Error	New Error	s	%
Odometry	$0.106578865995068 \times 10^{-10}$	$0.837389354639250 \times 10^{-11}$	5	21.43
PID Controller	$0.453945103062736 \times 10^{-14}$	$0.440585745442590 \times 10^{-14}$	5	2.94
Lead-Lag System	$0.294150262243136 \times 10^{-11}$	$0.235435212105148 \times 10^{-11}$	10	19.96
Runge-Kutta 2	$0.750448486755706 \times 10^{-7}$	$0.658915054553695 \times 10^{-7}$	5	12.19
Runge-Kutta 4	$0.201827996912328 \times 10^{-1}$	$0.169791306481639 \times 10^{-1}$	5	15.87
Trapezoidal Rule	$0.536291684923368 \times 10^{-9}$	$0.488971110442931 \times 10^{-9}$	20	8.82

TABLE 6.2 – *Initial and new errors on the examples programs of Section 6.3.*

Code	Original Code Execution Time (s)	Optimized Code Execution Time (s)	Percentage Improvement	Mean on n runs
Odometry	$3.2 \cdot 10^{-2}$	$2.0 \cdot 10^{-2}$	37.5%	10^4
PID Controller	$2.0 \cdot 10^{-2}$	$0.8 \cdot 10^{-2}$	60.0%	10^2
Lead-Lag System	$8.0 \cdot 10^{-2}$	$6.0 \cdot 10^{-2}$	25.0%	10^3
Runge-Kutta 4	$1.6 \cdot 10^{-2}$	$1.2 \cdot 10^{-2}$	25.0%	10^3
Trapezoidal Rule	$2.0 \cdot 10^{-2}$	$0.8 \cdot 10^{-2}$	60.0%	10^4
Rocket	$0.82 \cdot 10^{2}$	$0.61 \cdot 10^{2}$	25.6%	10^2

TABLE 6.3 – *Execution time measurements of programs of Section 6.3.*

on their execution time. Our evaluation shows the effect of improving the numerical accuracy of programs through the examples chosen, on their execution time. In Figure 6.3, we compare the execution time needed by each program before and after optimization. These programs are implemented in the `C` programming language and compiled with `GCC 4.2.1` using the default optimization level of compilers `-o0`. For example, if we take the case of `odometry`, we observe that we optimize its execution time by 37.5%. More generally, the percentage of execution time improvement of all programs presented before, is between 25% and 60% which is significantly important.

6.4 Improving the Accuracy of Interprocedural Programs

In this section we discuss some practical experiences we made with our tool on the interprocedural program transformation. The interprocedural transformation rules seen in Section 5.2 have been integrated to our tool. To perform our analysis, we take a list of various existing examples which are introduced in this chapter or presented in Chapter 7.

Listing 6.13 – Listing of the original Odometry program with functions.

```
double main(){
 x = 0.0; y = 0.0; arg  = 0.0; delta_d = 0.0; delta_dl = 0.0; delta_dr = 0.0;
 delta_theta = 0.0; t = 0.0; x = 0.0; y = 0.0; inv_l = 0.1; c = 12.34;
 sr = 0.785398163397; sl = 0.525398163397; theta = 0.0;
 while (t < 10.0) {
   delta_dl = c * sl;
   delta_dr = c * sr;
   delta_d = (delta_dl + delta_dr) * 0.5;
   delta_theta = (delta_dr - delta_dl) * inv_l;
   arg = theta + (delta_theta * 0.5);
   z = cosi(arg);
   x = x + (delta_d * z);
   q = sini(arg);
   y = y + (delta_d * q);
   theta = theta + delta_theta;
   t = t + 1.0;
 }
 return x;
}
double cosi(double a){
  b = 1.0 - (a * a * 0.5) + ((a * a * a * a) * 0.0416666666);
  return b;
}
double sini(double u){
  w = u - ((u * u * u) * 0.1666666666) + ((u * u * u * u * u) * 0.008333333);
  return w;
}
```

6.4.1 Odometry

The first example consists in the Odometry program previously introduced in Section 6.3.1. At the difference, the program given in Listing 6.13 contains two functions cosi and sini that compute respectively the cosine and the sine using Taylor series expansion. Listing 6.14 gives the transformed program obtained by applying our interprocedural transformation on the program given in Listing 6.13. The accuracy of the transformed Odometry program is improved by 55.59%.

6.4.2 Newton-Raphson's Method

We take the example of the Newton-Raphson method presented in Section 7.2.2 of Chapter 7 at the difference, here that, it contains two functions that compute respectively the polynomial $(x-2)^5$ and its derivative. The original program corresponding to this method is given in Listing 6.15. By applying the interprocedural transformation, the transformed program obtained is given in Listing 6.16. When experimenting, the numerical accuracy of the transformed program of Newton-Raphson method is improved by 99.97%. This significant improvement is due es-

Listing 6.14 – Listing of the transformed Odometry program with functions.

```
double main() {
  t = 0.0; theta = 0.0; y = 0.0; x = 0.0; arg = 0.0; delta_theta = 0.0;
  delta_d = 0.0; delta_dr = 0.0; delta_dl = 0.0;
  while (t < 10.0) {
    delta_dl = 6.483413336318978;
    delta_dr = 9.691813336318978;
    delta_d = (0.5 * (delta_dl + delta_dr));
    delta_theta = (0.1 * (delta_dr - delta_dl));
    arg = (theta + (delta_theta * 0.5));
    z = ((0.0416666666 * (arg * (arg * (arg * arg)))) + (1.0 - (0.5 * (arg * arg))));
    x = (x + (delta_d * z));
    q = ((0.008333333 * (arg * (arg * (arg * (arg * arg))))) + (arg - (0.1666666666 * (arg
      * (arg * arg)))));
    y = (y + (delta_d * q));
    theta = (theta + delta_theta);
    t = (t + 1.0);
  };
  return x;
}
```

sentially to the floating-point errors arising during the computations of the developed form of polynomial which evaluate very poorly close to the root.

Listing 6.15 – Listing of the original Newton-Raphson's method program.

```
double main(){
  a  = 0.0; a0  = 0.0;
  while (a0 <= 3.0) {
    u = f(a);
    w = ff(a);
    a_n = a - (u / w);
    a = a_n;
    a0 = a0 + 0.1;
  };
  return a_n;
}
double f(double x){
  y = (x * x * x * x * x) - (10.0 * x * x * x * x) + (40.0 * x * x * x) - (80.0 * x * x)
    + (80.0 * x) - (32.0);
  return y;
}
double ff(double v){
  z = (5.0 * v * v * v * v) - (40.0 *v * v * v) + (120.0 * v * v) - (160.0 * v) + (80.0);
  return z;
}
```

Listing 6.16 – Listing of the transformed Newton-Raphson's method program.

```
double main() {
  a0 = 0.0;
  a = 0.0;
  while (a0 <= 3.0) {
    u = (-32.0 + ((a * 80.0) + (((a * (a * (a * 40.0))) + ((a * (a * (a * (a * a)))) - (a
      * (a * (10.0 * (a * a)))))) - (a * (a * 80.0)))));
    w = (80.0 + (((a * (a * 120.0)) + ((5.0 * ((a * a) * (a * a))) - (a * (40.0 * (a * a)))))
      - (a * 160.0)));
    a_n = ((u / -w) + a);
    a = a_n;
    a0 = (a0 + 0.1);
  };
  return a_n;
}
```

6.4.3 Runge-Kutta Method

In this section, we aim at transforming the runge-kutta method previously described in Section 6.3.4 in order to improve its accuracy using our interprocedural rules presented in Section 5.2 of Chapter 5.

Second order Runge-Kutta Method

Listing 6.17 is similar to Listing 6.5 at the difference, it includes functions. Listing 6.18 presents the transformed program obtained using our tool. The program obtained is improved by 5.24%.

Listing 6.17 – Listing of the original second order Runge-Kutta Method.

```
double main(){
  x = 1.0; y = 1.0; z = 0.0; h = 0.1; t = 0.0; yn = 0.0; sixieme = 0.16666667;
  while (t < 1.0)  {
    p = f(x,y);
    k1 = h * p;
    p1 = x + h/2.0;
    p2 = y + k1/2.0;
    p3 = f(p1,p2);
    k2 = h * p3;
    yn = y + k2;
    t = t + 0.1;
    y = yn;
  }
  return yn;
}
double f(double u, double v){
  w = 2.0 * u + 3.0 * v;
  return w;
}
```

Listing 6.18 – Listing of the transformed second order Runge-Kutta Method.

```
double main() {
  y = 1.0; t = 0.0; yn = 0.0;
  while (t < 1.0) {
    p = (2.0 + (3.0 * y));
    p2 = (((5.0 * 0.1) / 2.0) + y);
    u = (1.0 + 0.05);
    w = (((0.05 + 1.0) * 2.0) + (3.0 * p2));
    p3 = (2.0 + (3.0 * y));
    yn = (y + (0.1 * p3));
    t = (t + 0.1);
    y = yn;
  };
  return yn;
}
```

Fourth order Runge-Kutta Method

Let us consider in this section the Runge-Kutta method of fourth order presented in Section 6.3.4. We give in Listing 6.19 the original program corresponding to this method with functions. After transformation, our tool returns the program given in Listing 6.20 which accuracy improvement is 0.31%.

Listing 6.19 – Listing of the original fourth order Runge-Kutta Method.

```
double main(){
   x = 1.0; y = 1.0; z = 0.0; h = 0.1; t = 0.0; yn = 0.0; sixieme = 0.16666667;
   while (t < 1.0)  {
     ff = f(x,y);
     k1 = h * ff;
     p1 = x + h/2.0;
     p2 = y + k1/2.0;
     p3 = f(p1,p2);
     k2 = h * p3;
     q1 = y + k2/2.0;
     q2 = f(p1,q1);
     k3 = h * q2;
     s1 = x + h;
     s2 = y + k3;
     s3 = f(s1,s2);
     k4 = h * s3;
     yn = y + (sixieme * h * (k1 + (2.0 * k2) + (2.0 * k3) + k4));
     t = t + 0.1;
     y = yn;
   }
   return yn;
}
double f(double u, double v){
  w = 2.0 * u + 3.0 * v;
  return w;
}
```

Listing 6.20 – Listing of the transformed fourth order Runge-Kutta Method.

```
double main() {
  y = 1.0; t = 0.0; yn = 0.0;
  while (t < 1.0) {
    ff = (2.0 + (3.0 * y));
    p2 = (((5.0 * 0.1) / 2.0) + y);
    u = (1.0 + 0.05);
    w = (((0.05 + 1.0) * 2.0) + (3.0 * 1.25));
    p3 = (2.0 + (3.0 * y));
    q1 = (((5.0 * 0.1) / 2.0) + y);
    u = (1.0 + 0.05);
    w = (((0.05 + 1.0) * 2.0) + (3.0 * 1.25));
    q2 = (2.0 + (3.0 * y));
    s2 = (y + (0.1 * 5.0));
    u = 1.1;
    w = ((1.1 * 2.0) + (3.0 * 1.5));
    s3 = (2.0 + (3.0 * y));
    yn = ((((s3 * 0.1) + ((0.1 * (2.0 * q2)) + ((0.1 * (2.0 * p3)) + (ff * 0.1)))) * 0.016666667) + y);
    t = (t + 0.1);
    y = yn;
  };
  return yn;
}
```

6.4.4 Simpson's Method

The last example is in the Simpson's method presented in Section 6.5.1. We apply our interprocedural transformation on the original program presented in Listing 6.21 to automatically iomprove its accuracy. The transformed program is given in Listing 6.22.

6.4.5 Experimental Results

To demonstrate the efficiency of our tool, we show on the Table 6.4 the different results obtained when running on it the selected list of examples. Our experimentations concerns the Odometry, the Newton-Raphson method, the Runge-Kutta method of second and fourth order and the method of Simpson. If we take for example the Odometry program, our technique improves its numerical accuracy by about 55.59%.

Listing 6.21 – Listing of the original Simpson method.

```
double main(){
  a = 1.95; b = 2.05; n = 250.0; i = 1.0; h = (b - a) / n; p = f(a); q = f(b); s = p +  q;
  while (i < 250.0) {
    u = a + (i * h);
    w = f(u);
    s = s + 4.0 * w;
    i = i + 1.0;
  };
    i = 2.0;
  while (i < 249.0) {
    v = a + (i * h);
    r = f(v);
    s = s + 2.0 * r;
    i = i + 1.0;
  };
  s = s * (h / 3.0);
  return s;
}
double f(double x){
  y = ((x * x * x * x * x * x * x) - 14.0 * (x * x * x * x * x * x) + 84.0 * (x * x * x * x
    * x) - 280.0  * (x * x * x * x) + 560.0 * (x * x * x)  - 672.0 * (x * x) + 448.0
    * x - 128.0);
  return y;
}
```

Code	Error		
	old	new	%
Odometry	80.1387828935	35.5874162935	55.59
Newton-Raphson	1361.78356143	0.4	99.97
Runge-Kutta 2	1.583	1.5	5.24
Runge-Kutta 4	1.05941375119	1.05609166779	0.31
Simpson Method	137.523389936	137.523389936	0.0

TABLE 6.4 – *Comparison between the accuracy results of programs before and after optimization.*

6.5 Optimizing the Data-Type Formats of Variables

In this section, we are interesting in studying another criterion of our tool. We emphasize the efficiency of our implementation in terms of improving the data-types format of variables used by the programs. More precisely, we show that by using our tool, we approximate the results to be ever accurate and close to the results obtained under the double precision while using single precision.

In the light of these ideas, let us confirm our claims by means of a small examples such as Simpson's method. We start by briefly describing what our program computes, and then we give

Listing 6.22 – Listing of the transformed Simpson method.

```
double main() {
  TMP_1 = ((1.95 * 1.95) * 1.95);
  TMP_3 = ((1.95 * 1.95) * 1.95);
  p = ((((((((((TMP_1 * 1.95) * 1.95) * 1.95) * 1.95) - (14. * (((TMP_3 * 1.95) * 1.95)
    * 1.95))) + (84. * ((((1.95 * 1.95) * 1.95) * 1.95) * 1.95))) - (280. * (((1.95 * 1.95)
    * 1.95) * 1.95))) + (560. * ((1.95 * 1.95) * 1.95))) - (672. * (1.95 * 1.95)))
    + (448. * 1.95)) - 128.);
  TMP_7 = ((2.05 * 2.05) * 2.05);
  TMP_9 = ((2.05 * 2.05) * 2.05);
  q = ((((((((((TMP_7 * 2.05) * 2.05) * 2.05) * 2.05) - (14. * (((TMP_9 * 2.05) * 2.05)
    * 2.05))) + (84. * ((((2.05 * 2.05) * 2.05) * 2.05) * 2.05))) - (280. * (((2.05 * 2.05)
    * 2.05) * 2.05))) + (560. * ((2.05 * 2.05) * 2.05))) - (672. * (2.05 * 2.05)))
    + (448. * 2.05)) - 128.);
  i = 1.0;
  s = (-0.000000000780688 + 0.000000000780574);
  while (i < 250.) {
    u = (1.95 + (i * (0.1 / 250.0)));
    TMP_13 = ((u * u) * u);
    TMP_14 = u;
    TMP_15 = ((u * u) * u);
    TMP_16 = u;
    TMP_17 = u;
    TMP_18 = u;
    w = ((((((((((TMP_13 * TMP_14) * u) * u) * u) - (14.0 * (((TMP_15 * TMP_16) * u) * u)))
      + (84.0 * ((((TMP_17 * TMP_18) * u) * u) * u))) - (280.0 * (((u * u) * u) * u)))
      + (560.0 * ((u * u) * u))) - (672.0 * (u * u))) + (448.0 * u)) - 128.0);
    s = (s + (4.0 * w));
    i = (i + 1.0);
  };
  i = 2.0;
  while (i < 249.0) {
    v = (1.95 + (i * (0.1 / 250.0)));
    TMP_19 = ((v * v) * v);
    TMP_20 = v;
    TMP_21 = ((v * v) * v);
    TMP_22 = v;
    TMP_23 = v;
    TMP_24 = v;
    r = ((((((((((TMP_19 * TMP_20) * v) * v) * v) - (14.0. * (((TMP_21 * TMP_22) * v) * v)))
      + (84.0 * ((((TMP_23 * TMP_24) * v) * v) * v))) - (280.0 * (((v * v) * v) * v)))
      + (560.0 * ((v * v) * v))) - (672.0 * (v * v))) + (448.0 * v)) - 128.0);
    s = (s + (2.0 * r));
    i = (i + 1.0);
  };
  s = (((0.1 / 250.0) / 3.0) * s);
  return s;
}
```

their listing before and after being optimized with our tool. Their accuracy is then discussed.

6.5.1 Numerical Integration Method

Simpson's method is an improvement of the trapeze rule for numerical integration to approximate the computation of $\int_a^b f(x)\,\mathrm{d}x$. It uses a second order approximation of the function f by a quadratic polynomial P that takes three abscissa points a, b and m with $m = (a+b)/2$. When integrating the polynomial, we approximate the integral of f on the interval $[x, x+h]$ ($h \in \mathbb{R}$ small) with the integral of P on the same interval. Formally, the smaller the interval is, the better the integral approximation is. Consequently, we divide the interval $[a,b]$ into subintervals $[a, a+h]$, $[a+h, a+2h]$, ... and then we sum the obtained values for each interval. We write :

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\Big[f(x_0) + 2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{\frac{n}{2}} f(x_{2j-1}) + f(x_n)\Big] \quad , \tag{6.5}$$

where

- n is the number of subintervals of [a,b] where n is even,
- $h = (b-a)/n$ is the length of the subintervals,
- $x_i = a + i \times h$ for $i = 0, 1, \dots, n-1, n$.

In our case, we have chosen the polynomial given in Equation (6.6). It is well-known by the specialists of floating-point arithmetic that the developed form of the polynomial evaluates very poorly close to a multiple root. This motivates our choice of the function f below for our experiments.

$$\begin{array}{rcl} f(x) & = & (x-2.)^7 \\ & = & x^7 - 14.\times x^6 + 84.\times x^5 - 280.\times x^4 + 560.\times x^3 - 672.\times x^2 + 448.\times x - 128. \end{array} \tag{6.6}$$

The listing corresponding of the implementation of the Simpson's method is described in Listing 6.23.

6.5.2 Experimental Results

In this section, the experimental results, described hereafter, compare the numerical accuracy of programs using single and double precision with a program transformed with our tool and running with single precision only. Note that the codes taken for this study are written in `C`, compiled with the `GCC` compiler version `4.2.1` and executed by an `Intel Core i7` processor

Listing 6.23 – Listing of the original Simpson's method.

```
int main() {
  a = 1.9; b = 2.1; n = 100.0; i = 1.0; x = a; h = (b - a)/n;
  f = ((x*x*x*x*x*x*x*x) - 14.0 * (x*x*x*x*x*x*x) + 84.0 * (x*x*x*x*x*x) - 280.0
    * (x*x*x*x*x) + 560.0 * (x*x*x*x) - 672.0 * (x*x) + 448.0 * x - 128.0);
  x = b ;
  g = ((x*x*x*x*x*x*x*x) - 14.0 * (x*x*x*x*x*x*x) + 84.0 * (x*x*x*x*x*x) - 280.0
    * (x*x*x*x*x) + 560.0 * (x*x*x*x) - 672.0 * (x*x) + 448.0 * x - 128.0);
  s = f + g;
  while (i < n) {
    x = a + (i * h);
    f = ((x*x*x*x*x*x*x*x) - 14.0 * (x*x*x*x*x*x*x) + 84.0 * (x*x*x*x*x*x) - 280.0
      * (x*x*x*x*x) + 560.0 * (x*x*x*x) - 672.0 * (x*x) + 448.0 * x - 128.0);
    s = s + 4.0 * f;
    i = i + 1.0;
  };
  i = 2.0 ;
  while (i < n-1) {
    x = a + (i * h) ;
    f = ((x*x*x*x*x*x*x*x) - 14.0 * (x*x*x*x*x*x*x) + 84.0 * (x*x*x*x*x*x) - 280.0
      * (x*x*x*x*x) + 560.0 * (x*x*x*x) - 672.0 * (x*x) + 448.0 * x - 128.0);
    s = s + 2.0 * f;
    i = i + 1.0;
  };
  s = s * (h / 3.0);
}
```

under `Ubuntu 15.04`. In addition, programs are compiled with the optimization level $-o0$ to avoid any optimization done by the compiler and additionally, we enforce the copy in the memory at each execution step by declaring all the variables as `volatile` (this avoids that values are kept in registers using more than 64 bits).

The results obtained and presented in Figure 6-6 when executing the original program in single and double precision and the transformed program in single precision, demonstrate that our approach succeeds well to improve the accuracy. If we are interested in the result of computations around the multiple root 2.0, we can see on sub-figures (a), (b) and (c) of Figure 6-6 that the behavior of the optimized code is far closer to the original program executed with a double precision than the single precision original program. This shows that single precision may suffices in many contexts.

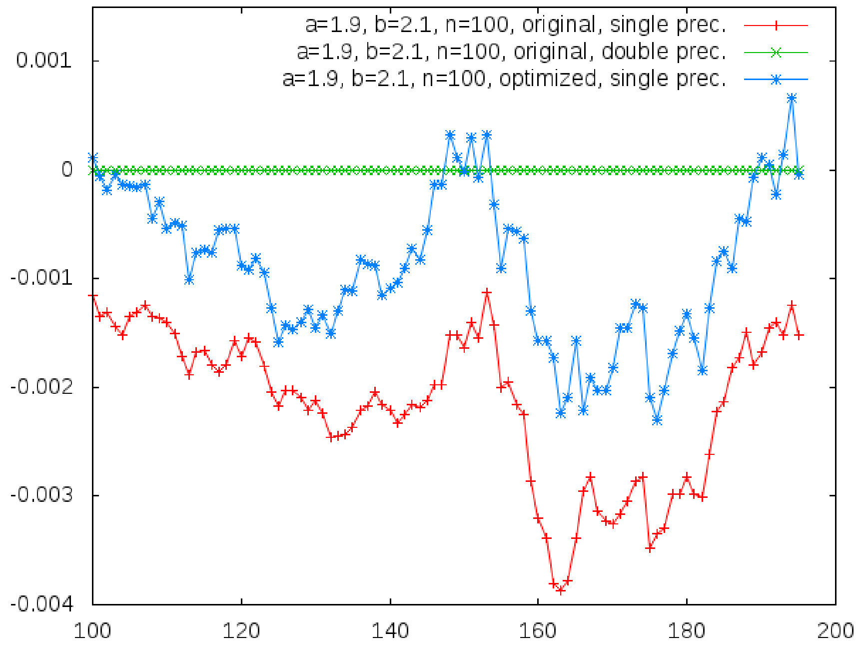
In Figure 6-6, one can see the difference, for different values of the step $h > 0$, in the computation of

$$s = \int_a^{a+nh} f(x)\,\mathrm{d}x, \quad 0 \le n < \frac{b-a}{h} \tag{6.7}$$

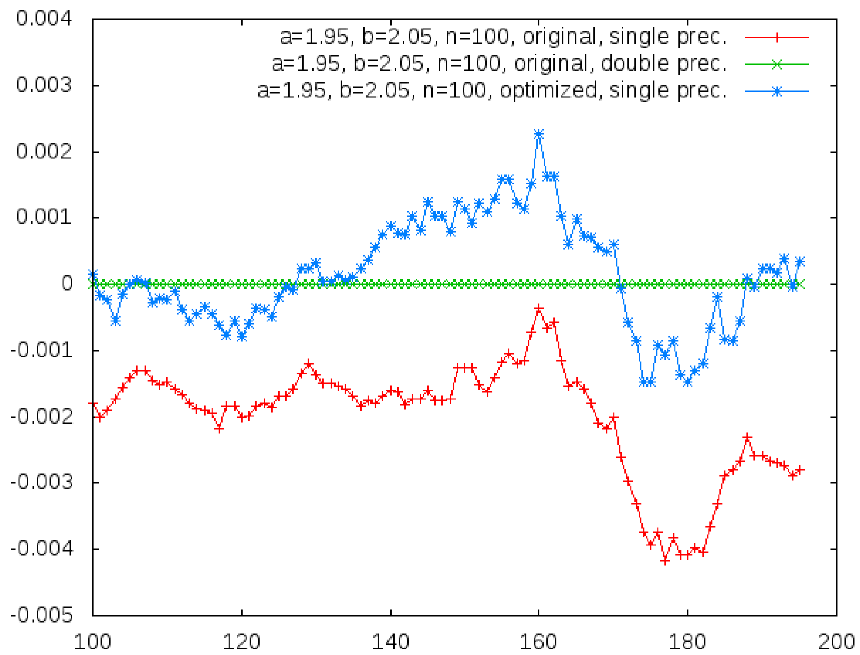
Listing 6.24 – Listing of the transformed Simpson method.

```
int main() {
  TMP_3 = 6.859 ;
  TMP_1 = ((((((TMP_3 * 1.9) * 1.9) * 1.9) * 1.9) - (14. * (((TMP_3 * 1.9) * 1.9)
        * 1.9))) + (84. * ((6.859 * 1.9) * 1.9)));
  TMP_2 = (1.9 * (280. * TMP_3));
  TMP_15 = 9.26100000000001;
  TMP_13 = ((((((TMP_15 * 2.1) * 2.1) * 2.1) * 2.1) - (14. * (((TMP_15 * 2.1)
         * 2.1) * 2.1))) + (84. * ((9.26100000000001 * 2.1) * 2.1)));
  TMP_14 = (2.1 * (280. * TMP_15));
  TMP_27 = 3.61;
  TMP_25 = ((((((TMP_27 * 1.9) * 1.9) * 1.9) * 1.9) * 1.9)
         - (14. * ((((TMP_27 * 1.9) * 1.9) * 1.9) * 1.9)));
  TMP_26 = (1.9 * (159.599999999999994 * TMP_3));
  TMP_32 = (TMP_3 * 1.9);
  i = 1.;
  s = ((((((TMP_1 - TMP_2) + (560. * TMP_3)) - 2425.920000000000073)
    + 851.199999999999932) - 128.) + (((((TMP_13 - TMP_14) + (560. * TMP_15))
    - 2963.520000000000437) + 940.800000000000068) - 128.));
  f = ((((((TMP_25 + TMP_26) - (280. * TMP_32)) + (560. * (TMP_27 * 1.9)))
    - (672. * TMP_27)) + 851.199999999999932) - 128.);
  x = 2.1;
  while (i < 100.) {
    x = (1.9 + (i * 0.002));
    TMP_37 = (x * x);
    TMP_35 = ((((((TMP_37 * x) * x) * x) * x) * x) - (14. * ((((TMP_37 * x) * x)
           * x) * x)));
    TMP_36 = (84. * (((TMP_37 * x) * x) * x));
    TMP_42 = (x * (TMP_37 * x));
    f = ((((((TMP_35 + TMP_36) - (280. * TMP_42)) + (560. * (TMP_37 * x)))-(672.
      * TMP_37)) + (448. * x)) - 128.);
    s = (s + (4. * f));
    i = (i + 1.);
  } ;
  i = 2. ;
  while (i < 99.) {
    x = (1.9 + (i * 0.002));
    TMP_47 = (x * x);
    TMP_45 = ((((((TMP_47 * x) * x) * x) * x) * x) - (14. * ((((TMP_47 * x) * x)
           * x) * x)));
    TMP_46 = (84. * (((TMP_47 * x) * x) * x));
    TMP_52 = (x * (TMP_47 * x));
    f = ((((((TMP_45 + TMP_46) - (280. * TMP_52)) + (560. * (TMP_47 * x))) - (672.
      * TMP_47)) + (448. * x)) - 128.);
    s = (s + (2. * f));
    i = (i + 1.);
  } ;
 s = (0.000666666666667 * s) ;
}
```

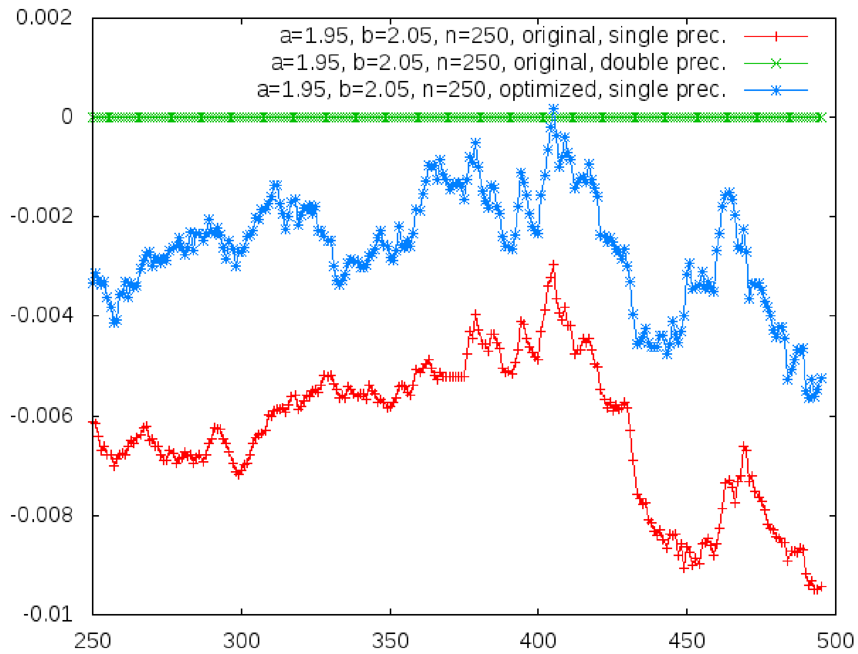
between the original program with both single and double precision and the transformed program (in single precision) in terms of numerical accuracy of computations. Obviously, the accuracy of the computations of the polynomial $f(x)$ depends on the values of x. The more the value of x is close to the multiple root, the worst the result is. For this reason, we make the interval $[a,b]$ vary by choosing $[a,b] \in \{[1.9,2.1],[1.95,2.05],[1.95,2.05]\}$ and by choosing to split them in $n =$



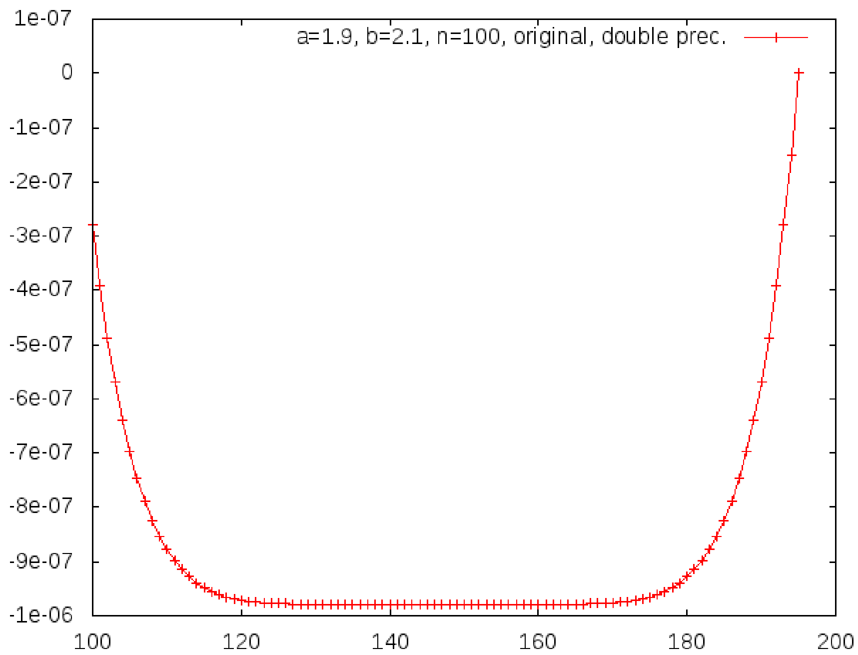
(a)



(b)



(c)



(d)

FIGURE 6-6 – *Simulation results of the Simpson's method with single, double precision and optimized program using our tool. The values of x and y axes correspond respectively to the value of n and s in Equation 6.7.*

100, $n = 100$ and $n = 250$ slices respectively for the application of Simpson's rule. Concerning the results obtained, our tool states that the percentage of the optimization computed by the abstract semantics of Section 3.4 is up to 99.39%. This means that the bound (obtained by the technique of Section 3.4) on the numerical error of the computed values of the polynomial at any iteration is reduced by 99.39%.

Curve (d) of Figure 6-6 displays the function $f(x)$ at points $n = 1.9 + 0.02 \times i$, $100 \leq i \leq 200$. Next, if we take for example Curve (b) of Figure 6-6, we observe that our implementation is as accurate as the original program in double precision for many values of n since it gives results very close to the double precision while working in single precision. Note that, for the $x - axis$ of Figure 6-6, we have chosen to observe only the interesting interval of n which is around the multiple root 2.0.

6.6 Conclusion

This chapter focuses on an important point which is the results obtained by our tool. We have shown that our tool improves the numerical accuracy of a set of representative programs taken from various fields of sciences and engineering. We have automatically tuned them and analyzed their accuracy before and after the transformation. The experiment results show the efficiency of our approach on the numerical accuracy of computations.

We have also extended our experiments to study the relation between the numerical accuracy of programs with the data-types formats of variables of programs. We have demonstrated that by comparing three programs :

1. The original program executed in single precision (32 bits),
2. The original program running in double precision (64 bits),
3. The optimized program in single precision (32 bits),

allow us to say that the experiment results obtained, on the method of Simpson, shown that the program optimized in single precision give us results close to an original program in double precision, and this without losing much information.

RÉSUMÉ CHAPITRE 7

Ce chapitre est consacré à étudier l'effet de l'optimisation de la précision numérique de programmes sur le temps d'exécution et la vitesse de convergence d'algorithmes itératifs. Pour effectuer cette étude, nous avons choisi quatre méthodes numériques itératives. Nous citons, la méthode de Jacobi, la méthode de Newton, une méthode de puissances itérées qui calcule la plus grande valeur propre d'une matrice ainsi un algorithme itératif d'orthogonalisation qui est basé sur la méthode de Gram-Schmidt. Nous définissons brièvement chaque méthode et nous donnons par la suite leur code avant et après transformation.

L'efficacité de notre outil en terme d'accélération de la vitesse de convergence de ces méthodes est mesurée par le nombre d'itérations nécessaires avant et après la transformation. Notons que le programme initial et le programme optimisé des méthodes numériques itératives sont implémentés dans le langage de programmation *C*, compilés avec le compilateur *GCC 4.2.1*, et exécutés sur le processeur *Intel Core i5* avec *4 Go* de mémoire en *IEEE* simple précision. Ces programmes sont compilés en utilisant le niveau d'optimisation du compilateur par défaut *-o0*.

Cette expérimentation nous a montré que l'amélioration de leurs précisions numériques permet de réduire le temps nécessaire par ces méthodes numériques itératives pour converger. À titre d'exemple, le nombre d'itérations essentiel par la méthode de Jacobi pour converger est réduit de 15% en moyenne.

Une étude complémentaire en terme de temps d'exécution de ces quatre méthodes itératives ainsi le nombre d'opérations flottantes (flops) a été faite. L'objectif est de garantir que lorsqu'on gagne en nombre d'itérations, on ne perd pas en temps d'exécution dans une seule itération de

boucle. Nous avons calculé le temps d'exécution nécessaire par chacune des méthodes considérées avant et après la transformation et les résultats obtenus montrent que le pourcentage de réduction est très important. Cette étude permet de montrer l'efficacité de notre outil et sa capacité d'améliorer la précision numérique ainsi le temps d'exécution en même temps. Prenant comme exemple la méthode de Jacobi, son temps d'exécution est réduit de 74.5%.

De plus, nous nous sommes intéressés à calculer le nombre d'opérations flottantes dans le programme de départ et celui transformé. Pour chaque méthode, nous comptons le nombre d'additions, de soustractions, de multiplications et de division pour chaque itération de boucle et aussi le nombre d'itération total nécessaire pour converger. Les résultats obtenus montrent que le pourcentage d'amélioration est toujours positif.

CHAPITRE 7

EXPERIMENTS : CONVERGENCE ACCELERATION

Contents

7.1 Introduction 121

7.2 Accelerating the Convergence of Iterative Methods 122

7.2.1 Linear Systems of Equations 122

7.2.2 Zero Finding 124

7.2.3 Eigenvalue Computation 127

7.2.4 Iterative Gram-Schmidt Method 129

7.3 Performance Analysis 131

7.4 Conclusion 133

7.1 Introduction

The main contribution of this chapter is to show that the execution time of numerical iterative methods is improved by increasing their numerical accuracy. In order to demonstrate the impact of the accuracy on the convergence time, we have chosen a set of four representative iterative methods which are Jacobi's Method, Newton-Raphson's Method, an Iterated Power Method used to compute the largest eigenvalue of a matrix and an iterative orthogonalization algorithm based

on Gram-Schmidt Method. The efficiency of our techniques in accelerating the convergence of these algorithms is measured by the number of iterations before and after optimization. Significant speedups are obtained in terms of execution time and number of floating-point operations (flops) needed to achieve the computation.

The original and optimized numerical iterative methods have been implemented in the `C` programming language, compiled with `GCC 4.2.1`, and executed on an `Intel Core i5` with `4 Go` memory in IEEE754 single precision in order to emphasize the effect of the finite precision. Programs are compiled with the default optimization level of the compiler `-o0`. We have tried other levels of optimization without observing significant changes in our results.

7.2 Accelerating the Convergence of Iterative Methods

In this section, we describe the different numerical iterative methods taken in our study. We give the listings of these methods before and after the transformation.

7.2.1 Linear Systems of Equations

We start with a first case study concerning Jacobi's method [58] which consists of an iterative computation that solves linear systems of the form $\mathbf{Ax} = \mathbf{b}$. From this equation, we build a sequence of vectors ($\mathbf{x}^{(0)}$, $\mathbf{x}^{(1)}$, ... , $\mathbf{x}^{(k)}$, $\mathbf{x}^{(k+1)}$, ...) that converges towards the solution $\mathbf{x}^{(k)}$ of the system of linear equations. To compute $\mathbf{x}^{(k+1)}$, we use :

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)}}{a_{ii}} \qquad \text{where } \mathbf{x}^{(k)} \text{ is known.} \tag{7.1}$$

The method iterates until $|x_i^{(k+1)} - x_i^{(k)}| < \varepsilon$ for the desired x_i, $1 \leq i \leq n$. A sufficient condition for the stability of Jacobi's method is that

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}|. \tag{7.2}$$

Let us now examine how we can improve the convergence of Jacobi's method on the example given in Equation (7.3). This system is stable with respect to the sufficient condition of Equa-

tion (7.2) but it is close to be unstable in the sense that $\forall i, 1 \leq i \leq 4, |a_{ii}| \approx \sum_{j=1, j \neq i}^{j=4} |a_{ij}|$.

$$\begin{pmatrix} 0.62 & 0.1 & 0.2 & -0.3 \\ 0.3 & 0.602 & -0.1 & 0.2 \\ 0.2 & -0.3 & 0.6006 & 0.1 \\ -0.1 & 0.2 & 0.3 & 0.601 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 1.0/2.0 \\ 1.0/3.0 \\ 1.0/4.0 \\ 1.0/5.0 \end{pmatrix}. \quad (7.3)$$

To solve Equation (7.3) by Jacobi's method, we use the algorithm presented in Listing 7.1. This program is transformed with our tool by using the set of transformation rules described in Section 4. Note that, in the version of this program given to our tool, we have unfolded the body of the while loop twice. This makes it possible to rewrite more drastically the code by mixing the computations of both iterations. In this example, without unfolding, we win very few iterations and, obviously, if we unfold the body of the loop more than twice, our tool improves even more the accuracy at the price of a larger code. Note that in the examples of the next sections, we do not perform such an unfolding because our tool already optimizes significantly the original codes (results would be even better with unfolding).

The program corresponding to Jacobi's method after optimization is shown in Listing 7.2. Note that this code is rather not intuitive and could be very difficult to write by hand. Concerning the accuracy of the variables, our tool states that the percentage of the optimization computed by the abstract semantics of Section 3.4 is up to 44.5%. This means that the bound on the numerical error of the computed values of x_i, $1 \leq i \leq 4$ at any iteration is reduced by 44.5%. The *target variable* at improving in this program is $\vartheta = \{x_i\}$, in other words, we improved the fourth variables (x_1, x_2, x_3, x_4) separately and we observed at each case the number of iterations required by the method to converge.

In Table 7.1, one can see the difference between the original and the transformed programs

Listing 7.1 – Listing of the original program of Jacobi's method.

```
eps = 10e-16; a11 = 0.61; a22 = 0.602; a33 = 0.6006; a44 = 0.601; b1 = 0.5; b2 = 1.0/3.0;
b3 = 0.25; b4 = 1.0/5.0;
while (e > eps) {
  x_n1 = (b1/a11) - (0.1/a11) * x2 - (0.2/a11) * x3 + (0.3/a11) * x4;
  x_n2 = (b2/a22) - (0.3/a22) * x1 + (0.1/a22) * x3 - (0.2/a22) * x4;
  x_n3 = (b3/a33) - (0.2/a33) * x1 + (0.3/a33) * x2 - (0.1/a33) * x4;
  x_n4 = (b4/a44) + (0.1/a44) * x1 - (0.2/a44) * x2 - (0.3/a44) * x3;
  e = x_n1 - x1; x1 = x_n1; x2 = x_n2; x3 = x_n3; x4 = x_n4;
}
```

Listing 7.2 – Listing of the optimized program of Jacobi's method.

```
eps = 10e-16 ;
while (e > eps) {
  TMP_1 = (0.553709856035437 - (x1 * 0.498338870431894)) ;
  TMP_2 = (0.166112956810631 * x3) ;
  TMP_6 = (0.333000333000333 * x1) ;
  x_n1 = (((0.819672131147541 - (0.163934426229508 * ((TMP_1 + TMP_2)
       - (0.332225913621263 * x4)))) - (0.327868852459016 * (((0.416250416250416
       - TMP_6) + (0.4995004995005 * x2)) - (0.166500166500167 * x4))))
       + (0.491803278688525 * (((0.332778702163062 + (0.166389351081531 * x1))
       - (0.332778702163062 * x2)) - (0.499168053244592 * x3)))) ;
  x_n2 = (((0.553709856035437 - (0.498338870431894 * x_n1)) + (0.166112956810631
       * (((0.416250416250416 - TMP_6) + (0.4995004995005 * x2)) - (0.166500166500167
       * x4)))) - (0.332225913621263 * (((0.332778702163062 + (0.166389351081531 * x1))
       - (0.332778702163062 * x2)) - (0.499168053244592 * x3)))) ;
  x_n3 = (((0.416250416250416 - (0.333000333000333 * x_n1)) + (0.4995004995005 * x_n2))
       - (0.166500166500167 * (((0.332778702163062 + (0.166389351081531 * x1))
       - (0.332778702163062 * x2)) - (0.499168053244592 * x3)))) ;
  x_n4 = (((0.332778702163062 + (0.166389351081531 * x_n1)) - (0.332778702163062 * x_n2))
       - (0.499168053244592 * x_n3)) ;
  e = (x_n4 - x4) ;  x1 = x_n1 ;  x2 = x_n2 ;  x3 = x_n3 ;  x4 = x_n4 ;
}
```

in term of the number of iterations needed to compute x_1, x_2, x_3 and x_4. Roughly speaking, about 15% less iterations are needed with the optimized code. Obviously, the fact that the body of the loop is unfolded twice, in the optimized code is taken into account in the computation of the number of iterations needed to converge.

x_i	Initial Number of iteration	Iterations Number after optimization	Difference	Percentage of Improvement
x_1	1891	1628	263	14.0
x_2	2068	1702	366	17.3
x_3	2019	1702	317	15.7
x_4	1953	1628	325	16.7

TABLE 7.1 – Number of iterations of Jacobi's method before and after optimization to compute x_i, $1 \leq i \leq 4$.

7.2.2 Zero Finding

Newton-Raphson's Method [58] is a numerical method used to compute the successive approximations of the zeros of a real-valued function. In order to understand how this method works, let us start with the derivative $f'(x)$ of the function f which may be used to find the slope, and thus

the equation of the tangent to the curve at a specified point. The method starts in an interval, for the equation $f(x) = 0$, in which there exists only one solution, the root a.

We choose a value u_0 close enough to a and then we build a sequence $(u_n)_{n\in\mathbb{N}}$ where u_{n+1} is obtained from u_n, as the abscissa of the meet point of the x-axis and the tangent at point $(u_n, f(u_n))$ to the function f. The final formula is given in Equation (7.4). Note that the computation stops when $|u_{n-1} - u_n| < \varepsilon$.

$$u_{n+1} = u_n - \frac{f(u_n)}{f'(u_n)}. \tag{7.4}$$

In general, Newton-Raphson's converges very quickly (quadratic convergence) but it may be slower if the computation of f or f' is inaccurate. For our case study, we have chosen functions which are difficult to evaluate in the IEEE754 floating-point arithmetic. Let us consider the function $f(x) = (x-2)^5$. The developed formula of f and its derivative f' are :

$$f(x) = x^5 - 10x^4 + 40x^3 - 80x^2 + 80x - 32, \tag{7.5}$$

$$f'(x) = 5x^4 - 40x^3 + 120x^2 - 160x + 80. \tag{7.6}$$

It is well-known from floating-point arithmetic experts that evaluating the developed form of a polynomial close to a multiple root may be quite inaccurate [60]. Consequently, this example presents some numerical difficulties for Newton-Raphson's method which converges slowly in this case.

The algorithm corresponding to Equation (7.4) is given in Listing 7.3. We recognize the computation of $f(x)$ and its derivative $f'(x)$ called `ff`. When optimizing this program with our tool,

Listing 7.3 – Listing of the original Newton-Raphson's program.

```
eps = 0.0005 ; e = 1.0 ; x = 0.0 ;
while (e > eps){
  f  = (x*x*x*x*x) - (10.0*x*x*x*x) + (40.0*x*x*x) - (80.0*x*x) + (80.0*x) - (32.0) ;
  ff = (5.0*x*x*x*x) - (40.0*x*x*x) + (120.0*x*x) - (160.0*x) + (80.0) ;
  x_n = x - (f / ff) ;
  e = (x - x_n) ;
  x = x_n ;
  if (e < 0.0) {
    e = (e * (-1.0)) ;
  }
  else {
    e = (e * 1.0) ;
  } ;
}
```

Listing 7.4 – Listing of the optimized Newton-Raphson's program.

```
eps = 0.0005 ; e   = 1.0 ; x   = 0.0 ; x_n = 1.0 ;
while (e > eps){
  TMP_1 = (((((x * x) * x) * x) * x) - ((((10.0 * x) * x) * x) * x)) ;
  TMP_2 = ((x * x) * (40.0 * x)) ;
  TMP_3 = (80.0 * x) ;
  TMP_5 = (((5.0 * x) * x) * (x * x)) ;
  TMP_6 = ((x * x) * (40.0 * x)) ;
  TMP_7 = (120.0 * x) ;
  x_n = (x - (((((TMP_1 + TMP_2) - (TMP_3 * x)) + TMP_3) - 32.0)
       / ((((TMP_5 - TMP_6) + (TMP_7 * x)) - (160.0 * x)) + 80.0))) ;
  e = (x - x_n) ;
  x = x_n ;
  if (e < 0.0) {
    e = (e * (-1.0)) ;
  }
  else {
    e = (e * 1.0) ;
  } ;
}
```

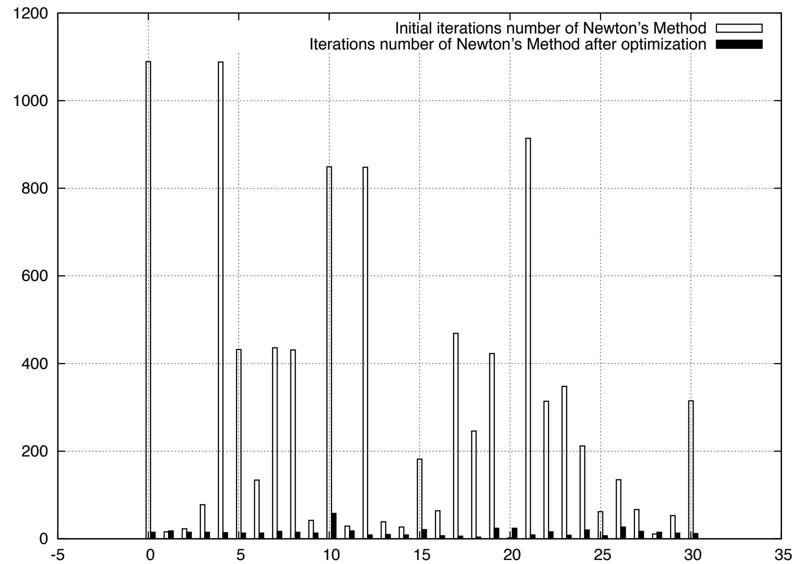


FIGURE 7-1 – Number of iterations of the Newton-Raphson's Method before and after optimization for initial values ranging from 0 to 3 (30 runs with a step of 0.1).

we get the program of Listing 7.4. The accuracy of the x_n's is improved up to 1.53% following the semantics of Section 3.4.

The results given in Figure 7-1 show how much our tool optimizes the number of iterations

needed to converge. Obviously, this number of iterations needed to converge to the solution with a given precision depends on the initial value x_0. We have experimented several initial values. We make x_0 go from 0 to 3 with a step of 0.1. The 30 results are presented in Figure 7-1. Due to the numerical inaccuracies, the number of iterations ranges from 10 to 1200, approximatively. It is always close to 10 with the transformed program.

7.2.3 Eigenvalue Computation

The Iterated Power Method is a method used to compute the largest eigenvalue of a matrix and the related eigenvector [47]. We start by setting an arbitrary initial vector $\mathbf{x}^{(0)}$ containing a single non-zero element. Next, we build an intermediate vector $\mathbf{y}^{(1)}$ such that $\mathbf{A}\mathbf{x}^{(0)} = \mathbf{y}^{(1)}$. Then, we build $\mathbf{x}^{(1)}$ by re-normalizing $\mathbf{y}^{(1)}$ so that the selected component is again equal to 1. This process is repeated up to convergence. Optionally, we may change the reference vector if it converges to 0. Note that the renormalization factor converges to the largest eigenvalue and $\mathbf{x}$ converges to the related eigenvector, under the conditions that the largest eigenvalue is unique and that all eigenvectors are independent. The convergence speed is proportional to the ratio between the two largest eigenvalues (in absolute value). For our experiments, let us take a square matrix $\mathbf{A}$ of dimension 4 with the eigenvector $(0.0 \;\; 0.0 \;\; 0.0 \;\; 1.0)^T$ given on the Equation (7.3) :

$$\mathbf{A} = \begin{pmatrix} \mathtt{d} & 0.01 & 0.01 & 0.01 \\ 0.01 & \mathtt{d} & 0.01 & 0.01 \\ 0.01 & 0.01 & \mathtt{d} & 0.01 \\ 0.01 & 0.01 & 0.01 & \mathtt{d} \end{pmatrix} \text{with } \mathtt{d} \in [175.0, 200.0]. \tag{7.7}$$

By applying the Iterated Power Method, the first intermediary vector is

$$\begin{array}{lcl} \mathbf{A}\mathbf{x}^0 & = & \mathbf{y}^1 \;, \\ \mathbf{A}y^1/y_4^1 & = & \mathbf{y}^2 \;, \\ \mathbf{A}y^2/y_4^2 & = & \mathbf{y}^3 \;, \\ & \dots & \end{array} \tag{7.8}$$

To re-normalize this intermediate vector, we divide it by the last value $\mathtt{d}$, in order to have $y_4^{(1)}$ equal to 1.0. The new vector is : $(0.01/\mathtt{d} \;\; 0.01/\mathtt{d} \;\; 0.01/\mathtt{d} \;\; 1.0)^T$. We keep iterating with the new intermediate vector. We have to repeat the former operation on this new intermediate vector in order to re-normalize it. By repeating this process several times, the series converge to the eigenvector $(1.0 \;\; 1.0 \;\; 1.0 \;\; 1.0)^T$.

Listing 7.5 – Listing of the original iterated power method.

```
eps = 0.0005 ; d = 175.0 ; v1 = 0.0 ; v2 = 0.0 ; v3 = 0.0 ; v4 = 1.0 ; a41 = 0.01 ;
a44 = d ; a11 = d ; a12 = 0.01 ; a13 = 0.01 ; a14 = 0.01 ; a21 = 0.01 ; a22 = d ;
a42 = 0.01 ; e = 1.0 ; a23 = 0.01 ; a24 = 0.01 ; a31 = 0.01 ; a32 =  0.01 ; a33 = d ;
a34 = 0.01 ; a43 = 0.01 ;
while (e  >  eps) {
   vx = a11 * v1 + a12 * v2 + a13 * v3 + a14 * v4 ;
   vy = a21 * v1 + a22 * v2 + a23 * v3 + a24 * v4 ;
   vz = a31 * v1 + a32 * v2 + a33 * v3 + a34 * v4 ;
   vw = a41 * v1 + a42 * v2 + a43 * v3 + a44 * v4 ;
   v1 = vx / vw ;
   v2 = vy / vw ;
   v3 = vz / vw ;
   v4 =  1.0 ;
   e = 1.0 - v1;
   if (v1 < 1.0) {
     e = 1.0 - v1 ;
   };
   else {
     e = v1 - 1.0;
   };
}
```

The original code of the iterated power method is given in Listing 7.5. Our tool has improved the error bounds computed by the semantics of Section 3.4 of up to 25.76%. The optimized code is given in Listing 7.6. The *target variable* at improving in this method is $\vartheta = \{v_1\}$.

Listing 7.6 – Listing of the optimized iterated power method.

```
eps = 0.0005 ; d = 175.0 ; v1  =  0.0 ;  v2 =   0.0 ;   v3 =  0.0 ;  v4 =  1.0 ;  e = 1.0 ;
while (e > eps) {
  vx = ((((0.01 * v4) + (0.01 * v2)) + (0.01 * v3)) + (d * v1)) ;
  vy = ((((0.01 * v1) + (0.01 * v4)) + (0.01 * v3)) + (d * v2)) ;
  vz = ((((0.01 * v4) + (0.01 * v2)) + (0.01 * v1)) + (d * v3)) ;
  vw = ((((0.01 * v2) + (0.01 * v1)) + (0.01 * v3)) + (d * v4)) ;
  v1 = (vx / vw) ;
  v2 = (vy / vw) ;
  v3 = (vz / vw) ;
  v4 = 1.0 ;
  e = (1.0 - v1) ;
  if (v1 < 1.0) {
    e = 1.0 - v1 ;}
  else { e = v1 - 1.0 ;}
}
```

When running this program, we observe significant improved results. In other words, the transformed implementation succeeds to reduce the numbers of iterations needed to converge and accelerates the convergence speed of the iterative power method. The experimental results are summarized in Figure 7-2. The number of iterations is reduced by at least 475 iterations.

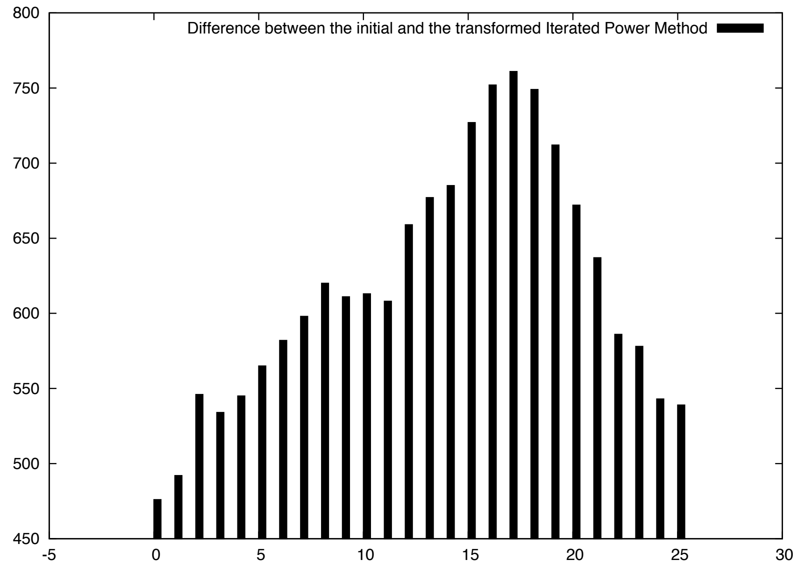


FIGURE 7-2 – Difference between numbers of iterations of original and optimized Iterated Power Method (tests done for $\mathtt{d} \in [175, 200]$ with a step of 1).

7.2.4 Iterative Gram-Schmidt Method

The Gram-Schmidt method is used to orthogonalize a set of non-zero vectors in a Euclidean or Hermitian space $\mathbb{R}^n$. This method takes as input a linear independent set of vectors $\mathbf{Q} = \{\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j\}$. The output is the orthogonal set of vectors $\mathbf{Q'} = \{\mathbf{q'}_1, \mathbf{q'}_2, \ldots, \mathbf{q'}_j\}$, with $1 \leq j \leq n$ [8, 47, 54]. The process followed by Gram-Schmidt method starts by defining the projection :

$$proj_{\mathbf{q'}}(\mathbf{q}) = \frac{\langle \mathbf{q}, \mathbf{q'} \rangle}{\langle \mathbf{q'}, \mathbf{q'} \rangle} \mathbf{q'}. \tag{7.9}$$

In Equation (7.9), $\langle \mathbf{q}, \mathbf{q'} \rangle$ is the dot product of the vectors $\mathbf{q}$ and $\mathbf{q'}$. It means that the vector $\mathbf{q}$ is projected orthogonally onto the line spanned by the vector $\mathbf{q'}$. The normalized vectors are $\mathbf{e}_j = \frac{\mathbf{q'}_j}{||\mathbf{q'}_j||}$ where $||\mathbf{q'}_j||$ consists of the norm of the vector $\mathbf{q'}_j$. Explicitly, Gram-Schmidt process

can be written as :

$$
\begin{aligned}
\mathbf{q'}_1 &= \mathbf{q}_1, \\
\mathbf{q'}_2 &= \mathbf{q}_2 - proj_{\mathbf{q'}_1}(\mathbf{q}_2), \\
&\vdots \\
\mathbf{q'}_j &= \mathbf{q}_j - \sum_{j=1}^{j-1} proj_{\mathbf{q'}_j}(\mathbf{q}_j).
\end{aligned}
$$

In general, Gram-Schmidt method is numerically stable and it is not necessary to use an iterative algorithm. However, important numerical errors may arise when the vectors become more and more linearly dependent. In this case iterative algorithms yield better results, as for example the algorithm of Listing 7.7 which repeats the orthogonalization step until the ratio $\frac{||\mathbf{q'}_j||_2}{||\mathbf{q}_j||_2}$ becomes large enough [54]. First, it starts by computing the orthogonal projection of $\text{span}(\{\mathbf{q_1}, \mathbf{q_2}, \mathbf{q_3}\})$. Then, it substracts this projection from the original vector and then normalizes the result to obtain $\mathbf{q}_3$, *i.e.*, $\text{span}(\{\mathbf{q_1}, \mathbf{q_2}, \mathbf{q_3}\}) = \text{span}(\{\mathbf{x_1}, \mathbf{x_2}, \mathbf{x_3}\})$ and $\mathbf{q}_3$ is orthogonal to $\mathbf{q}_1$, $\mathbf{q}_2$. We assume that $r_{jj} > 0$.

To understand how this algorithm works, let us take for example a set of vectors in $\mathbb{R}^3$ that we aim at orthogonalizing.

$$
Q_n = \left\{ \mathbf{q}_1 = \begin{pmatrix} 1/7\mathtt{n} \\ 0 \\ 0 \end{pmatrix}, \mathbf{q}_2 = \begin{pmatrix} 0 \\ 1/25\mathtt{n} \\ 0 \end{pmatrix}, \mathbf{q}_3 = \begin{pmatrix} 1/2592 \\ 1/2601 \\ 1/2583 \end{pmatrix} \right\}. \tag{7.10}
$$

For our experiments, we have chosen the values of $\mathtt{n}$ ranging from 1 to 10.

In Listing 7.8, we give the transformed iterative Gram-Schmidt algorithm generated by our tool. By applying our techniques to the iterative Gram-Schmidt algorithm presented previously in Listing 7.7, we show in Figure 7-3 that the transformed algorithm converges faster than the original one by up to 14.5%. Note that in this method, the variable at optimizing is $\vartheta = \{r\}$.

Listing 7.7 – Listing of the original iterative Gram-Schmidt program.

```
Q11 = 1.0 / 7n ; Q12 = 0.0 ; Q13 = 0.0 ; Q21 = 0.0 ; Q22 = 1.0 / 25n; Q23 = 0.0 ;
Q31 = 1.0 / 2592.0 ; Q32 = 1.0 / 2601.0 ;  Q33 = 1.0 / 2583.0; eps = 0.000005 ;
qj1 = Q31; qj2 = Q32;  qj3 = Q33;  r1 = 0.0; r2 = 0.0; r3 = 0.0; e = 10.0;
r = qj1 * qj1 + qj2 * qj2 + qj3 * qj3; rold = sqrt(r);
while( e >  eps) {
    h1 = Q11 * qj1 + Q21 * qj2 + Q31 * qj3 ;
    h2 = Q12 * qj1 + Q22 * qj2 + Q32 * qj3 ;
    h3 = Q13 * qj1 + Q23 * qj2 + Q33 * qj3 ;
    qj1 = qj1 - (Q11 * h1 + Q12 * h2 + Q13 * h3) ;
    qj2 = qj2 - (Q21 * h1 + Q22 * h2 + Q23 * h3) ;
    qj3 = qj3 - (Q31 * h1 + Q32 * h2 + Q33 * h3) ;
    r1 = r1 + h1 ;
    r2 = r2 + h2 ;
    r3 = r3 + h3 ;
    r = qj1 * qj1 + qj2 * qj2 + qj3 * qj3 ;
    rjj = sqrt(r);
    e =  1.0 - (rjj / rold) ;
    if (e < 0.0) {
      e = -e ;
    };
    rold = rjj ;
}
```

Listing 7.8 – Listing of the optimized iterative Gram-Schmidt program.

```
Q11 = 1.0 / 7n ; Q12 = 0.0  ; Q13 = 0.0  ; Q21 = 0.0  ; Q22 = 1.0 / 25n;
Q23 = 0.0; Q31 = 1.0 / 2592.0 ; Q32 = 1.0 / 2601.0 ;  Q33 = 1.0 / 2583.0;
eps = 0.000005 ; qj1 = Q31; qj2 = Q32;  qj3 = Q33;  r1 = 0.0; r2 = 0.0;
r3 = 0.0; e = 10.0; r = qj1 * qj1 + qj2 * qj2 + qj3 * qj3; rold = sqrt(r);
while ( e > eps) {
  TMP_6 = (qj1 * qj3) ;
  TMP_14 = (qj2 * qj3) ;
  qj1 = (qj1 - ((0.14285714285 *(((qj1 * qj3)) + (0.14285714285 * qj1))));
  qj2 = (qj2 - ((0.04 * (((0.0 * qj1) + (qj2 * qj3)) + (0.04 * qj2))))) ;
  qj3 = (qj3 - (((qj2 * ((TMP_14) + (0.04 * qj2))) + (qj3 + (qj3 * qj3))))
      + (qj1 * (((qj1 * qj3)) + (0.14285714285 * qj1))))) ;
  r1 = (r1 + ((TMP_6) + (0.14285714285 * qj1))) ;
  r2 = (r2 + ((TMP_14) + (0.04 * qj2))) ;
  r3 = (r3 + ((qj3 * qj3))) ;
  r = qj1 * qj1 + qj2 * qj2 + qj3 * qj3 ;
  rjj = sqrt(r);
  e =  1.0 - (rjj / rold) ;
  if (e < 0.0) {
    e = -e ;
  };
  rold = rjj ;
}
```

7.3 Performance Analysis

We have shown in the former sections that we optimize the number of iterations of our four iterative numerical algorithms. In this section, we provide complementary benchmarks concerning

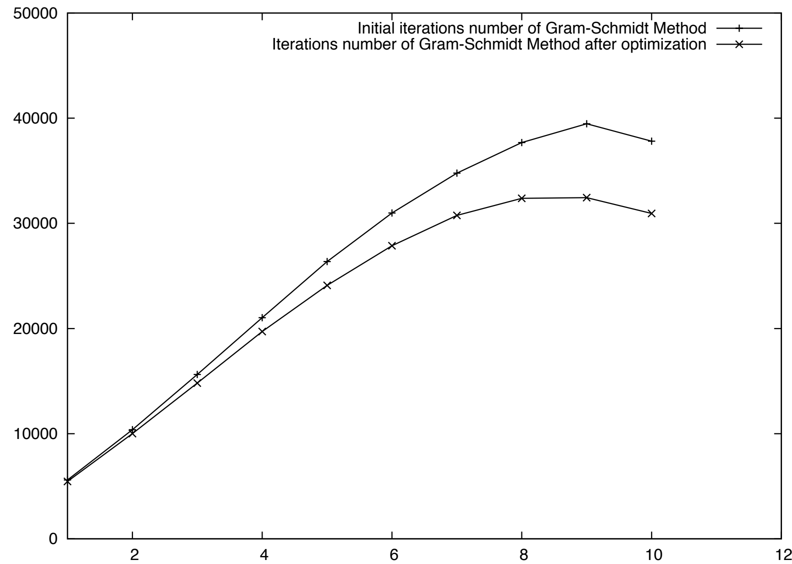


FIGURE 7-3 – Iterations number of original and optimized iterative Gram-Schmidt Method for the family $(Q_n)_n$ of vectors, $1 \leq n \leq 10$.

speedups and the number of floating-point operations. Our objective is to check that the gains in the number of iterations are not annealed by overheads in the execution time of a single iteration or by other side effects for example due to the compiler.

We have chosen to observe the speedups of x_4 for Jacobi's method, and $x_0 = 3$ for Newton-Raphson's method. We have taken $d = 200$ for the iterated power method and $\mathbf{q}_{11} = \frac{1}{63}$ and $\mathbf{q}_{22} = \frac{1}{225}$ for iterative Gram-Schmidt method.

If we focus on measuring the execution time of the four programs before and after optimization, we observe that the percentage of improvement is rather important. If we take for example Jacobi's method, we remark that we reduce its execution time by 74.5%. We give in Table 7.2 the speedups results obtained for the four methods. These results are very interesting to emphasize the usefulness of our tool and its ability to improve accuracy and execution time simultaneously. We have also counted the number of floating-point operations (flops) in the original and optimized codes. The numbers are given in Table 7.3. For each method, we count the number of additions and subtractions as well as the number of products and divisions for a single iteration and for the total number of iterations required in each case to converge. These results are coherent with the observed execution times.

Method	Original Code Execution Time (s)	Optimized Code Execution Time (s)	Percentage Improvement	Mean on n Runs
Jacobi	$1.49 \cdot 10^{-4}$	$0.38 \cdot 10^{-4}$	74.5%	10^4
Newton-Raphson	$1.34 \cdot 10^{-3}$	$0.02 \cdot 10^{-3}$	98.4%	10^4
Eigenvalue	$4.50 \cdot 10^{-2}$	$3.07 \cdot 10^{-2}$	31.6%	10^3
Gram-Schmidt	$1.99 \cdot 10^{-1}$	$1.70 \cdot 10^{-1}$	14.5%	10^2

TABLE 7.2 – Execution time measurements of programs of Section 7.2.

Method	$\sharp$ of $\pm$ per it Original Code	$\sharp$ of $\pm$ per it Optimized Code	Total $\sharp$ of $\pm$ Original Code	Total $\sharp$ of $\pm$ opt Optimized Code	Percentage of Improvement
Jacobi	13	15	25389	24420	3.81
Newton-Raphson	11	11	3465	132	96.19
Eigenvalue	15	15	694080	685995	1.16
Gram-Schmidt	21	19	791364	715996	9.52
Method	$\sharp$ of $\times$ per it Original Code	$\sharp$ of $\times$ per it Optimized Code	Total $\sharp$ of $\times$ Original Code	Total $\sharp$ of $\times$ opt Optimized Code	Percentage of Improvement
Jacobi	28	14	54684	22792	58.32
Newton-Raphson	27	26	8505	312	96.33
Eigenvalue	19	19	879168	868927	1.16
Gram-Schmidt	22	20	712316	647560	9.09

TABLE 7.3 – Floating-point operations needed by programs of Section 7.2 to converge.

7.4 Conclusion

This chapter focuses on the impact of automatic transformation of programs in order to improve their numerical accuracy on the convergence time of numerical iterative algorithms. Our experiments show the efficiency of our approach on the time required by numerical iterative methods to converge. We have experimented several representative numerical iterative methods by giving them to our tool and we have shown that the transformed programs converge more quickly than the original ones without loss of accuracy. We have extended this study with complementary results concerning the execution time and the total number of floating-point operations.

RÉSUMÉ CHAPITRE 8

Nous avons présenté dans ce manuscrit notre technique de transformation intraprocédurale et interprocédurale de programmes pour améliorer leur précision de calcul numérique. Elle se base sur une réécriture automatique des bouts de code tel que les boucles, les structures de contrôles, les fonctions, etc. Nous avons également défini un ensemble de règles de transformation implémentées dans un outil *SALSA*. Notre outil qui se base sur les méthodes d'analyse statique par interprétation abstraite, prend en entrée un programme initial et retourne en sortie un programme optimisé. Notre outil permet de comparer les erreurs de calculs du programme initial et celui transformé. Il est à noter que ces deux programmes n'ont pas forcément la même sémantique mais sont mathématiquement équivalents.

Nous avons ensuite montré la correction de notre transformation à travers une preuve mathématique qui compare la précision numérique du programme initial avec celle de programme transformé. Cette preuve par induction est appliquée sur les expressions arithmétiques et les commandes.

De plus, nous avons montré l'efficacité de notre approche via une série d'exemples provenant de divers domaines comme la robotique, l'avionique, les méthodes numériques, etc. Les résultats des expérimentations montrent de combien est améliorée la précision numérique. Nous avons aussi effectué une étude sur l'effet de l'amélioration de la précision sur la vitesse de convergence de quelques méthodes numériques itératives. Les résultats obtenus montrent que le nombre d'itérations nécessaires pour que ces méthodes itératives convergent est réduit en améliorant leur précision. Nous avons étudier également l'impact de l'amélioration de la précision sur le format des variables utilisées. Les résultats ont montré que le programme transformé en simple préci-

sion est plus proche de celui initial en double précision. Cette application permet à l'utilisateur de travailler en simple précision sans perdre beaucoup d'informations.

En première perspective, nous souhaitons étendre notre approche pour traiter les pointeurs et autres traits de langage. De plus, nous envisageons d'appliquer notre outil pour améliorer la précision numérique des systèmes distribués, et aussi pour étudier le compromis temps d'exécution, performance de calculs, précision numérique et vitesse de convergence. Un autre point très prometteur concernant la reproductibilité des résultats. Nous souhaitons étudier comment notre outil améliore la reproductibilité des calculs. D'autres travaux consistent à intégrer notre outil dans un compilateur LustreC développé à l'ONERA de Toulouse, mais aussi, dans le cadre d'une collaboration scientifique avec l'université de Washington, à comparer notre outil aux autres outils existant à l'aide de différents benchmarks venant de divers domaines d'application.

CHAPITRE 8

CONCLUSION AND FUTURE WORK

Contents

8.1 Conclusion 137
8.2 Perspectives 138

8.1 Conclusion

The work presented in this thesis describes our technique for intraprocedural and interprocedural program transformation to improve the numerical accuracy of their computations. It focuses on how automatically rewriting pieces of code such as loops, conditionals, functions, etc,. Then, we have defined a set of transformation rules which are implemented in our tool, *SALSA*. Based on static analysis by abstract interpretation, *SALSA* takes as input an original program and generates an optimized program. These two programs do not have the same semantics but they are mathematically equivalent for the considered outputs. This tool allows us to compare the original program with the transformed one in terms of error of computations. We have demonstrated the soundness of our approach by a mathematic proof that compares two programs, more precisely, it compares the numerical accuracy of the original code with that of the transformed code. This proof by structural induction is applied to the different language constructions supported by *SALSA*. To emphasize the efficiency of our tool, we have experimented it on several examples

coming from various domains like robotics, avionics, numerical methods, etc,. The results obtained are very relevant, we have succeeded to improve their numerical accuracy as discussed in previous chapter.

When experimenting, we have observed that :

- The numerical accuracy of programs bounded by worst errors static analysis are improved from 2.94% to 21.43%. In addition, if we compare the execution time needed by each program before and after optimization, we observe that we do not slow down the programs. Without compiler optimization `GCC -o0`, our method even accelerate the executions. This is due to the fact that our transformation performs constant propagation and partial evaluation of the static expressions.
- The improvement of the numerical accuracy of iterative methods accelerates their convergence speed. More precisely, the number of iterations required by Newton's method, Jacobi's method, etc, to converge to the solution is reduced. This improvement is more than 20%. In addition the execution time of all of the considered numerical iterative methods is optimized by up to 15%. This observation is an important result of our work.
- The improvement of the numerical accuracy of programs impacts the data-types formats of the variables used. That means that when optimizing the accuracy of computations, the program optimized executed with 32 bits is close to an original program running with 64 bits. These experiments give the user the advantage to degrade in the precision without loss of information and that allows one to economize the memory, the execution time, the bandwidth, etc.

8.2 Perspectives

In future research, it will be interesting to have a more complete tool. We aim at generalizing *SALSA* to other kinds of programming language patterns like pointers. This research direction is difficult to be implemented but it remains a very important point to be achieved. Note that, a work in progress consists in implementing arrays.

Another extension aims at extending our approach to optimize several reference variables simultaneously. A difficulty is that the optimization of one variable may decrease the accuracy of other variables. Compromises have to be done. Finally, our transformation relies on a static analysis of the source codes. Indeed, we select the optimized program by using the abstract semantics in Section 3.4, we compute certified error bounds which can be over-approximated.

We would like to improve it by using more accurate relational domains in order to obtain finer error bounds completed by statistical results on the actual accuracy gains on concrete executions.

The transformation introduced in this thesis computes an over-approximation of the round-off errors due to the floating-point arithmetic. However, the transformation itself is independent of the computation of the errors, it only depends on the order $\sqsubseteq$ introduced in Definition 4.4.1 and using another order would change the objective of the transformation. We could use it with another static analysis which would estimate differently which expression of an APEG is the best. For example, we could use the compromise between accuracy and execution time or different arithmetic such as the fixed-point arithmetic which is widely used in embedded systems. A way of coupling our transformation with fixed, integer or interval arithmetic is discussed in [67].

On-going work looks at how to optimize at compile-time the numerical accuracy of synchronous programs [52, 19]. This work is done in collaboration with the developers of the `LustreC` compiler (P.L. Garoche and X. Thirioux). It consists in integrating our program transformation techniques into a large compiler for Lustre language [45] which is developed at the ONERA and IRIT in Toulouse. In other words, we bring our ideas about generating more accurate imperative programs to the optimization of embedded codes initially written in a synchronous language. It introduces optimizations of the accuracy at compile-time for synchronous programs, like Lustre and Simulink, in the industrial development process. A new tool named LustreCSalsa, that takes as input a program written in Lustre which can be generated by Simulink models, optimizes the accuracy of the program by automatic transformation, is under development. Our approach consists in taking an original program which is written in synchronous language and generating another program more accurate than the first one using an imperative language as intermediary language. Another extension aims at generating a Lustre program from a program initially written in Simulink. As long term, our objective is to transform efficiently hight level programs, for example written in Simulink to low level code written in C or assembly. In this context, the preservation of the accuracy of computations is crucial.

On-going international work with Zachary Tatlock, professor at the University of Washington, USA, consists in comparing our tools on various benchmarks coming from different applications. An initial work on FPBench [36] provides a foundation for evaluating and comparing floating-point analysis and optimization tools. Already the FPBench format can serve as a common language, allowing tools like *SALSA*, Herbie, FPTaylor, Rosa, etc., to cooperatively analyze and improve floating-point code. In future work, we will add additional metrics for accuracy and performance to the set of evaluators provided by the FPBench tooling and begin developing a

standard set of benchmarks around the various measures. We will also expand the set of languages with direct support for compilation to and from the FPBench format. As more tools grow support for FPBench, we will provide automated comparisons of different floating-point tools. Longer term, we intend to support for mixed-precision benchmarks, fixed-point computations, and additional data structures such as vectors, records, and matrices.

A significant interest would be to extend the current work with a case study concerning a real size numerical application. The study described in [30] is a first step in this direction. Indeed, in collaboration with the physicist team of LAMPS Laboratory of the university of Perpignan, we are interesting to study problems from the non-smooth contact mechanics. It is well-known that floating-point computations may impact the numerical quality of the results of simulations as well as their execution times, especially when dealing with sensitive functions such as the ones that we find in non-smooth contact mechanics. In this direction, we aim at optimizing the accuracy of numerical simulations on the time required by their iterative methods to converge. Indeed, we will apply *SALSA* on selected examples coming from non-smooth and non-linear problems and on algorithms for linear systems. For our case study, we have taken an example of a linear contact between a deformable body and a foundation.

Another interesting perspective consists in extending our work to perform the high performance computing programs. In this direction, we aim at solving problems due to the numerical accuracy like the order of operation of a distributed system. We are interesting also in studying the compromise execution time, computation performances, numerical accuracy and the convergence acceleration of numerical methods. A key issue is to study the impact of accuracy optimization on the convergence time of distributed numerical algorithms like the ones used usually for high performance computing. In addition, still about distributed systems, an important issue concerns the reproducibility of the results : different runs of the same application yield different results due to the variations in the order of evaluation of the mathematical expression. We would like to study how our technique could improve reproducibility.

BIBLIOGRAPHIE

[1] https ://pypi.python.org/pypi/gmpy2.

[2] https ://www.python.org/.

[3] http ://www.mpfr.org/.

[4] http ://www.polyspcae.com.

[5] Patriot missile defense : Software problem led to system failure at dhahran, saudi arabi. Technical Report GAO-IMTEC-92-26, IMTEC-92-26, february 1992.

[6] Ariane 5 flight 501 failure. Technical report, Inquiry Board, Jully 1996.

[7] Ait tool, http ://www.absint.com, 2007.

[8] N. Abdelmalek. Roundoff error analysis for gram-schmidt method and solution of linear least squares problem. *BIT*, 11 :345–368, 1971.

[9] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In D. B. Whalley and R. Cytron, editors, *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, pages 231–239. ACM, 2004.

[10] ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.

[11] A-W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[12] K. J. Åström and T. Hagglund. *PID Controllers, 2nd ed*. Instrument Society of America, 1995.

[13] K. J. Åström, C. C. Hang, P. Persson, and W. Khuen Ho. Towards intelligent PID control. *Automatica*, 28(1) :1–9, 1992.

[14] 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.

[15] 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.

[16] 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.

[17] G. Birkhoff. *Lattice Theory (3rd ed).* Colloquium Publications, 1967.

[18] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In R. Cytron and R. Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003*, pages 196–207. ACM, 2003.

[19] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre : A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 178–188. ACM Press, 1987.

[20] A. Chapoutot. Interval slopes as a numerical abstract domain for floating-point variables. In R. Cousot and M. Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 184–200. Springer, 2010.

[21] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk. Numerical reproducibility for the parallel reduction on multi- and many-core architectures. Technical Report hal-00949355v4, Technical Report hal-00949355, INRIA, DALI–LIRMM, LIP6, 2014.

[22] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1) :7–22, 2002.

[23] 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.

[24] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4) :511–547, 1992.

[25] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In S. Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences*

on Theory and Practice of Software, ETAPS 2005, Proceedings, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[26] P. Cousout and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Principles of Programming Languages*, pages 178–190. ACM, 2002.

[27] R. Cytron and R. Gershbein. Efficient accomodation of may-alias information in SSA form. In *Programming Language Design and Implementation (PLDI)*, pages 36–45. ACM, 1993.

[28] N. Damouche, M. Martel, and A. Chapoutot. Impact of accuracy optimization on the convergence of numerical iterative methods. In M. Falaschi, editor, *LOPSTR 2015*, volume 9527 of *Lecture Notes in Computer Science*, pages 143–160. Springer, 2015.

[29] N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In M. Núñez and M. Güdemann, editors, *FMICS'15*, volume 9128 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2015.

[30] N. Damouche, M. Martel, and A. Chapoutot. Optimizing the accuracy of a rocket trajectory simulation by program transformation. In *CF'15*, pages 40 :1–40 :2. ACM, 2015.

[31] N. Damouche, M. Martel, and A. Chapoutot. Transformation of a PID controller for numerical accuracy. *Electronic Notes in Theoretical Computer Science*, 317 :47–54, 2015.

[32] N. Damouche, M. Martel, and A. Chapoutot. Amélioration à la compilation de la précision de programmes numériques. In P-E. Moreau F. Dadeau, editor, *8th Journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel, GDR-GPL'2016*, pages 27–34, 2016.

[33] N. Damouche, M. Martel, and A. Chapoutot. Data-types optimization for floating-point formats by program transformation. In *IEEE International Conference on Control, Decision and Information Technologies, CoDIT'16*. IEEE, 2016.

[34] N. Damouche, M. Martel, and A. Chapoutot. Improving the numerical accuracy of programs by automatic transformation. *International Journal on Software Tools for Technology Transfer*, 2016.

[35] N. Damouche, M. Martel, and A. Chapoutot. Numerically accurate code generation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO'16*, page xxiii. ACM New York, NY, USA, 2016.

[36] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock. Toward a standard benchmark format and suite for floating-point analysis. In P. Prabhakar S. Bogomolov, M. Martel, editor, *NSV'16*, Lecture Notes in Computer Science. Springer, 2016.

[37] E. Darulova and V. Kuncak. Sound compilation of reals. In S. Jagannathan and P. Sewell, editors, *POPL'14*, pages 235–248. ACM, 2014.

[38] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In M. Alpuente, B. Cook, and Ch. Joubert, editors, *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[39] 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.

[40] J. Demmel and H. D. Nguyen. Fast reproducible floating-point summation. In A. Nannarelli, P-M. Seidel, and P. Tak Peter Tang, editors, *IEEE Symposium on Computer Arithmetic, ARITH 2013*, pages 163–172. IEEE Computer Society, 2013.

[41] J. Demmel and H. D. Nguyen. Parallel reproducible summation. *IEEE Transactions on Computers*, 64(7) :2060–2070, 2015.

[42] J. Feret. Static analysis of digital filters. In D. 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.

[43] E. Feron. From control systems to control software. *IEEE Control Systems Magazine*, 30(6) :50–71, 2010.

[44] 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.

[45] P-L. Garoche, F. Howar, T. Kahsai, and X. Thirioux. Testing-based compiler validation for synchronous languages. In J. M. Badger and K. Yvonne Rozier, editors, *NASA Formal Methods - 6th International Symposium, NFM 2014, Proceedings*, volume 8430 of *Lecture Notes in Computer Science*, pages 246–251. Springer, 2014.

[46] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1) :5–48, 1991.

[47] G. H. Golub and C. F. van Loan. *Matrix computations (3. ed.)*. Johns Hopkins University Press, 1996.

[48] 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.

[49] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations : A simple abstract interpreter. In D. 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, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 209–212. Springer, 2002.

[50] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *European Congress on Embedded Real Time Software, ERTS 2006, Proceedings*, 2006.

[51] E. Goubault and S. Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*. Springer, 2011.

[52] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Springer-Verlag New York Inc., 2010.

[53] E. Hankin. *Lambda Calculi A Guide For Computer Scientists*. Clarendon Press, Oxford, 1994.

[54] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal. Orthogonalization routine in SLEPc Technical Report STR-1. In *Polytechnic University of Valencia*. STR1, 2007.

[55] G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The* Coq *Proof Assistant - A tutorial - Version 8.0*, April 2004.

[56] 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.

[57] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3) :480–503, 1996.

[58] Atkinson K. *An Introduction to Numerical Analysis*. J. Wiley & Sons, 1989.

[59] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion : Theory and practice. *ACM Transactions on Programming Languages and System*, 16(4) :1117–1155, 1994.

[60] Ph. Langlois and N. Louvet. How to ensure a faithful polynomial evaluation with the compensated horner algorithm. In *ARITH-18*, pages 141–149. IEEE Computer Society, 2007.

[61] S. Lerner, D. Grove, and C. Chambers. Composing dataflow analyses and transformations. In J. Launchbury and J. C. Mitchell, editors, *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 270–282. ACM, 2002.

[62] F. Logozzo. Practical verification for the working programmer with codecontracts and abstract interpretation - (invited talk). In R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 19–22. Springer, 2011.

[63] M. Martel. Propagation of roundoff errors in finite precision computations : A semantics approach. In D. 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.

[64] M. Martel. An overview of semantics for the validation of numerical programs. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005*, volume 3385 of *Lecture Notes in Computer Science*, pages 59–77. Springer, 2005.

[65] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1) :7–30, 2006.

[66] M. Martel. Semantics-based transformation of arithmetic expressions. In H. Riis Nielson and G. Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 298–314. Springer, 2007.

[67] M. Martel. Accurate evaluation of arithmetic expressions (invited talk). *Electronic Notes in Theoretical Computer Science*, 287 :3–16, 2012.

[68] A. Miné. The octagon abstract domain. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01*, page 310. IEEE Computer Society, 2001.

[69] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In D. 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*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004.

[70] A. Miné. *Domaines numérique abstraits faiblement relationnels*. PhD thesis, École Normale Supérieure, 2005.

[71] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[72] R. Moore. Interval arithmetic and automatic error analysis in digital computing. *PhD thesis, Applied Math Statistics Lab, Stanford*, 25, 1962.

[73] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[74] R.E. Moore. Interval analysis. *Prentics Hall*, 1966.

[75] J-M. Muller. On the definition of ulp(x). Technical Report 2005-09, Laboratoire d'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, 2005.

[76] 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.

[77] P. G. Neumann. USS Yorktown Dead in Water After Divide by Zero. Technical Report 13 :07 :45-0700, PGN Stark Abstracting, Jully 1998.

[78] 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.

[79] N. Revol. Arithmétique par intervalles. *Calculateurs Parallèles*, 13(387), 2001.

[80] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1&2) :131–170, 1996.

[81] S. Sankaranarayanan, M. Colón, H. B. Sipma, and Z. Manna. Efficient strongly relational polyhedral analysis. In E. A. Emerson and K. S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006*, volume 3855 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2006.

[82] 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.

[83] B. Steffen, J. Knoop, and O. Rüthing. The value flow graph : A program representation for optimal program transformations. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 1990.

[84] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. In *Pacific journal of mathematics*, pages 285–309. Bd. 5, 1955.

[85] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation : a new approach to optimization. In Z. Shao and B. C. Pierce, editors, *ACM SIGPLAN-SIGACT POPL'09*, pages 264–276. ACM, 2009.

[86] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation : A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

[87] A. Werey, D. Delmas, and M. Martel. Fine-tuning the accuracy of numerical computations in avionics automatic code generators. In *European Congress Embedded Real Time Software and Systems, ERTS 2016*, Lecture Notes in Computer Science, pages 180–187, 2016.

[88] D. Whitfield and M. Lou Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and System*, 19(6) :1053–1084, 1997.

[89] G. Winskel. *The formal semantics of programming languages : an introduction*. MIT Press, 1993.

ABSTRACT

Critical software based on floating-point arithmetic requires rigorous verification and validation process to improve our confidence in their reliability and their safety. Unfortunately, available techniques for this task often provide overestimates of the round-off errors. We can cite Arian 5, Patriot rocket as well-known examples of disasters. These last years, several techniques have been proposed concerning the transformation of arithmetic expressions in order to improve their numerical accuracy and, in this work, we go one step further by automatically transforming larger pieces of code containing assignments, control structures and functions. We define a set of transformation rules allowing the generation, under certain conditions and in polynomial time, of larger expressions by performing limited formal computations, possibly among several iterations of a loop. These larger expressions are better suited to improve, by re-parsing, the numerical accuracy of the program results. We use abstract interpretation based static analysis techniques to over-approximate the round-off errors in programs and during the transformation of expressions. A tool has been implemented and experimental results are presented concerning classical numerical algorithms and algorithms for embedded systems.

RÉSUMÉ

Les systèmes critiques basés sur l'arithmétique flottante exigent un processus rigoureux de vérification et de validation pour augmenter notre confiance en leur sureté et leur fiabilité. Malheureusement, les techniques existentes fournissent souvent une surestimation d'erreurs d'arrondi. Nous citons Arian 5 et le missile Patriot comme fameux exemples de désastres causés par les erreurs de calculs. Ces dernières années, plusieurs techniques concernant la transformation d'expressions arithmétiques pour améliorer la précision numérique ont été proposées. Dans ce travail, nous allons une étape plus loin en transformant automatiquement non seulement des expressions arithmétiques mais des programmes complets contenant des affectations, des structures de contrôle et des fonctions. Nous définissons un ensemble de règles de transformation permettant la génération, sous certaines conditions et en un temps polynômial, des expressions plus larges en appliquant des calculs formels limités, au sein de plusieurs itérations d'une boucle. Par la suite, ces larges expressions sont re-parenthésées pour trouver la meilleure expression améliorant ainsi la précision numérique des calculs de programmes. Notre approche se base sur les techniques d'analyse statique par interprétation abstraite pour sur-rapprocher les erreurs d'arrondi dans les programmes et au moment de la transformation des expressions. Cette approche est implémenté dans notre outil et des résultats expérimentaux sur des algorithmes numériques classiques et des programmes venant du monde d'embarqués sont présentés.