Analyses de terminaison des calculs flottants

Fonenantsoa Maurica Andrianampoizinimaro

THÈSE DE DOCTORAT

En vue de l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ DE LA RÉUNION
EN INFORMATIQUE

Présentée par **Fonenantsoa MAURICA**

Le 8 décembre 2017

Analyses de terminaison des calculs flottants

Termination Analysis of Floating-Point Computations

Devant le jury composé de :

Amir BEN-AMRAM, Professeur *Collège académique de Tel-Aviv*	**Examinateur**
Sylvie BOLDO, Directrice de recherche *INRIA*	**Rapporteur**
Christian DELHOMME, Professeur *Université de La Réunion*	**Examinateur**
Laure GONNORD, MCF HDR *Université Lyon 1*	**Examinateur**
Salvador LUCAS, Professeur *Université Polytechnique de Valence*	**Rapporteur**
Frédéric MESNARD, Professeur *Université de La Réunion*	**Directeur**
Marianne MORILLON, Professeur *Université de La Réunion*	**Examinateur**
Etienne PAYET, Professeur *Université de La Réunion*	**Co-directeur**

Laboratoire d'Informatique et de Mathématiques

Cette thèse a reçu le soutien financier de la Région Réunion et de l'Union Européenne (Fonds Européen de Développement Régional - FEDER) dans le cadre du Programme Opérationnel de Coopération Territoriale.

My deepest gratitude first goes to my advisors, Frédéric Mesnard and Étienne Payet. Their guidance, their commitment and most importantly their trust in me made these last three years the most scientifically and humanely rewarding experience I ever had.

I am also deeply thankful to the members of my thesis jury. I thank Sylvie Boldo, Salvador Lucas and Amir Ben-Amram for their thorough reading and for their interest, as judged by the long list of comments they sent me. Their efforts significantly improved the quality of this work. I thank Christian Delhommé for our several afternoons of discussion. His always-fresh viewpoints gave me new looks at many technical problems and many technical proofs. I thank Laure Gonnord who monitored the progression of this work over these last three years through my thesis follow-up committee. She helped define the directions to take. I thank Marianne Morillon who accepted to be an examiner despite her many other responsibilities.

I thank Franck Védrine for giving me an academic license for Fluctuat, as well as for the several email exchanges we had on it. I thank Matt Lewis for the access to Juggernaut and its test suite. I thank Fausto Spoto for our exchanges on Julia. I thank Peter Schrammel for our exchanges on 2LS, and for the wonderful interpretation of Liszt's Chapelle he sent me.

I thank Nimit Singhania and Tian Tan for our never-ending online discussions on how is life like as a PhD student in Software Verification in various places of the world.

I am very thankful to the laboratory which allowed me to conduct my research in comfortable conditions. LIM makes every effort to ensure that its PhD students are not penalized by the geographical remoteness of Reunion Island.

Last but certainly not least, I thank my family, especially my mother, who had to bear with my innumerable whims. Of course, by family I also refer to you guys, the "$AVP - P + P$".

I fear I cannot mention all the people who helped in only a single page. I also fear that the thanks I made to those I could mention are far too insufficient. One thing is for sure though: I received a lot of support, I am deeply thankful for that.

Contents

List of Abbreviations

ARF	**A**ffine **R**anking **F**unction
cWF	**c**o**W**ell-**F**ounded
ELRF	**E**ventual **L**inear **R**anking **F**unction
FMA	**F**used **M**ultiply-**A**dd
FP	**F**loating-**P**oint
FPU	**F**loating-**P**oint **U**nit
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
k**-PAA**	k-**P**iecewise **A**ffine **A**pproximation functions, $k \in \mathbb{N}^*$
LLRF	**L**exicographic **L**inear **R**anking **F**unction
LP	**L**inear **P**rogramming
LRF	**L**inear **R**anking **F**unction
MLC	**M**ulti-path **L**inear-**C**onstraint
PR	**P**odelski-**R**ybalchenko
PRRF	**P**odelski-**R**ybalchenko-type **R**anking **F**unction
SImpL	**S**imple **Imp**erative **L**anguage
SOF	**S**ingle **O**peration **F**orm
SRF	**S**pace of **R**anking **F**unctions
SL	**S**imple **L**oop
$\mathbf{SL}_{\mathbb{F}}$	**S**imple floating-point **L**oop
$\mathbf{SL}_{\mathbb{Q}}$	**S**imple rational **L**oop
$\mathbf{SL}_{\mathbb{Z}}$	**S**imple integer **L**oop
SC	**S**ize-**C**hange
TI	**T**ransition **I**nvariant
TPA	**T**ransition **P**redicate **A**bstraction
ULP	**U**nit in the **L**ast **P**lace
WF	**W**ell-**F**ounded

General Introduction

Mark spent three hours taking a selfie. Now he wants to post it on Facebook. In order to get as much Likes as possible, he decides to edit it. He chooses to use ImageMagick[1] which is a command-line tool for image processing. He tries to modify the hue of the image and then, disaster! ImageMagick stops responding. The culprit is a software bug discovered in 2017: the computation of the new floating-point value of the hue may fail to terminate[2].

To the fictional character Mark, that bug is actually not very harmful. His Facebook fans just need to wait a little bit longer until he manages to retouch his selfie. However, there were cases in history where termination bugs and incorrect floating-point computations led to dramatic events. On one hand, a non-expected infinite loop caused the Microsoft Azure Storage Service to be interrupted for more than ten hours in 2014[3]. On the other hand, a faulty implementation of the floating-point division operation ultimately cost $475 million to Intel in 1994[4].

This thesis was born from these issues: it is concerned with termination analysis of programs that perform floating-point computations. Though termination analysis is a topic that has already been intensively studied during the last decades, literature mainly focused on mathematical programs that perform exact calculations. Notably, most of the produced work consider programs that handle real numbers, rational numbers or integers. In comparison and to the author's knowledge, the case where floating-point numbers are used instead is only covered in a limited manner. That may be partly due to the hardness of analyzing floating-point computations. Indeed, floating-point numbers are *non*-exact representations of real numbers: tricky rounding errors are inherent to their manipulation.

This thesis has six chapters. Chapters 1 and 2 are introductions to termination analysis and floating-point numbers. These two first chapters also serve as literature

[1] https://www.imagemagick.org/
[2] https://www.imagemagick.org/discourse-server/viewtopic.php?f=3&t=31506
[3] https://azure.microsoft.com/fr-fr/blog/update-on-azure-storage-service-interruption/
[4] http://boole.stanford.edu/pub/anapent.pdf

survey. Chapters 3, 4 and 5 present the main contributions of this thesis. These three chapters discuss approximations to the rationals that eliminate the difficulties introduced by the rounding errors and that allow the use of some well-known termination proving techniques. Chapter 6 discusses experimental results. It recounts how the techniques developed in this thesis apply to real-world programs. A general conclusion embedding the future directions the author envision ends this work.

Introduction Générale

Marc a pris trois heures pour prendre un selfie. Maintenant, il veut le publier sur Facebook. Afin d'obtenir autant de Likes que possible, il décide de le retoucher un peu. Pour cela, il choisit d'utiliser ImageMagick qui est un outil en ligne de commande pour le traitement d'image. Il essaie de modifier la teinte de l'image et c'est alors le désastre ! ImageMagick ne répond plus. La cause provient d'un bug logiciel découvert en 2017 : le calcul de la nouvelle valeur flottante de la teinte peut ne pas terminer.

Pour le personnage fictif Marc, ce bug n'est pas particulièrement nuisible. Ses fans sur Facebook doivent juste attendre un peu, le temps qu'il parvienne à retoucher son selfie. Cependant, il existe des cas dans l'histoire où des bugs de terminaison et des calculs flottants incorrects ont eu des conséquences dramatiques. D'une part, une boucle infinie imprévue a provoqué l'interruption du service Microsoft Azure Storage pendant plus de dix heures en 2014. D'autre part, une implémentation défectueuse de la division flottante a coûté 475 millions de dollars à Intel en 1994.

Cette thèse est née de ces problèmes: elle s'intéresse à l'analyse de terminaison des programmes qui effectuent des calculs flottants. L'analyse de terminaison est un sujet qui a déjà été intensivement étudié au cours des dernières décennies. Cependant, la littérature a principalement étudié des programmes mathématiques qui effectuent des calculs exacts. Notamment, la plupart des travaux produits étudient des programmes qui manipulent des nombres réels, des nombres rationnels ou des nombres entiers. En comparaison de cela et à la connaissance de l'auteur, le cas où les nombres flottants sont utilisés à la place n'a été qu'assez peu traité. Cela peut être en partie dû à la difficulté d'analyser les calculs flottants. En effet, les nombres flottants sont des représentations inexactes des nombres réels: de subtiles erreurs d'arrondis sont inhérentes à leur manipulation.

Cette thèse comporte six chapitres. Les chapitres 1 et 2 sont des introductions à l'analyse de terminaison et aux nombres flottants. Ces deux premiers chapitres servent également d'état de l'art. Les chapitres 3, 4 et 5 présentent les principales contributions de la thèse. Ces trois chapitres présentent des approximations rationelles qui éliminent les difficultés introduites par les erreurs d'arrondis et qui permettent l'utilisation de techniques d'analyse de terminaison bien connues. Le chapitre 6 présente les résultats expérimentaux. Il montre comment les techniques développées dans cette thèse s'appliquent aux programmes du monde réel. Une conclusion générale contenant les orientations futures envisagées par l'auteur boucle ce travail.

Part I

Basics and Survey

Chapter 1

Proving Program Termination

Abstract. This chapter is a brief exposition to program termination analysis. The topic is now more than sixty years of existence during which a vast amount of work has been conducted. Exhaustivity is far beyond the goal of this chapter. Instead, it aims at establishing the notions and main results on termination analysis that are used in this thesis, notably in Chapters 3, 4 and 5. This first chapter also aims at situating this thesis in the literature by mentioning a few relevant related work.

Résumé. Ce chapitre parle brièvement d'analyse de terminaison de programmes. C'est un sujet qui existe depuis plus de soixante ans maintenant. Une vaste quantité de résultats a été produits pendant ce temps. Ce chapitre ne se veut pas être exhaustif. Il a plutôt pour objectif d'établir les principales notions en analyse de terminaison qui seront utilisées dans cette thèse, notamment dans les chapitres 3, 4 et 5. Ce premier chapitre a également pour objectif de situer cette thèse dans la littérature en mentionnant quelques travaux connexes.

1.1 Introduction

The program termination problem consists in determining whether a given program will always stop or may execute forever. In contrast to the simplicity of its formulation, the problem has instances that are notoriously hard to solve. A typical example is the $3x + 1$ problem presented Figure 1.1. Traditionally credited to Collatz, this apparently simple problem started circulating among the mathematical community by the early 1950s. In those days, all mathematicians at Yale and at the University of Chicago worked on it but to no avail. Since then the problem was jokingly said to be a Cold War invention of the Russians meant to slow the progress of mathematics in the West. Until this day, $3x + 1$ continues to completely baffle mathematicians.

```
x = randNat();
    // a random natural number
while (x > 1) {
   if (x%2 == 0) // if x is even
     x = x/2;
   else x = 3*x + 1;
}
```

FIGURE 1.1: The $3x + 1$ problem [Lag11]: does this program always terminate for any possible initial value of x?

Now why do scientists put so much effort in trying to prove termination? Is that property really important in practice? Apart from the Azure Storage Service bug presented in the General Introduction, there were other cases where non-expected infinite loops caused dysfunctions to the Microsoft products. For example, throughout the day of December 31, 2008, the former portable media player Microsoft Zune was down due to a termination bug[1]. For yet another example, the dreaded Blue Screen of Death that plagued Windows users in the 2000s was sometimes caused by non-terminating device drivers[2]. These dysfunctions cost a lot to Microsoft. And the damages could be even worse: a non-responding software in an aircraft could ultimately cause human losses. Also, the verification of some program properties can be reduced to termination checking. That is notably the case of *liveness properties* [Urb15, Part IV]. These properties ensure that something desirable happens at least once or infinitely often during the program execution. For example ensure that any allocated memory space is eventually freed. Also, checking termination of a program can provide ways of measuring its time complexity [Ali+10]. Last, termination provers can even be used to verify... biological models [Coo+11]. For example cancer occurs when some cells continue do divide infinitely because their mechanism of replication failed to terminate.

The rest of this chapter is organized as follows. Section 1.2 introduces some basic notions on termination analysis. It also describes the programs we are interested in. Section 1.3 studies their termination when their set of states is finite. That section notably raises the need for program approximation, which is the topic of Section 1.4. Section 1.5 talks about termination of our considered class of programs when their set of states if infinite. Section 1.6 concludes.

[1] https://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/
[2] http://www.zdnet.com/article/why-the-blue-screen-of-death-no-longer-plagues-windows-users/

1.2 Program termination basics

This section introduces some notions and notations related to termination analysis that are used throughout the thesis.

Definition 1.1 (The Simple Imperative Language (SImpL)). *Let SImpL be the imperative language defined by the following syntax:*

$$
\begin{aligned}
\mathit{stmt} ::= {} & X = \mathit{aexp}; && (X \in \mathcal{X}) \\
| \quad & \texttt{if}(\mathit{bexp})\{\mathit{stmt}\}\ \texttt{else}\{\mathit{stmt}\} \\
| \quad & \texttt{while}(\mathit{bexp})\{\mathit{stmt}\} \\
| \quad & \mathit{stmt}\ \ \mathit{stmt} \\
\mathit{prog} ::= {} & \mathit{stmt}
\end{aligned}
$$

where $\mathcal{X}$ is a set of symbols that represent the program variables. These variables range over numbers: $\mathbb{Z}$, $\mathbb{Q}, \mathbb{R}$ or subsets of them (which can be finite). Then aexp denotes an arithmetic expression that can use variables, numbers and the arithmetic operators $+, -, *$ and $/$. Then bexp denotes a boolean expression constructed with arithmetic expressions on the program variables, relations of strict and large inequalities over these expressions, boolean values and the logical operators $\neg, \wedge$ and $\vee$.

Also, SImpL possesses functions that return random booleans or random numbers.

Definition 1.2 (Programs in SImpL as transition systems). *We formalize a program $\mathcal{P}$ written in the SImpL language as a transition system $(\mathcal{S}, \mathcal{I}, \mathcal{R})$ where $\mathcal{S}$ is the set of all possible states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation from a state to its immediate possible successors.*

Let the set of program variables be $\mathcal{X} = \{x_1 \dots x_n\}$. We denote x the column vector $x = (x_1 \dots x_n)^T$. We use primed variables to denote the next value of their unprimed match after a transition.

Let $\mathcal{D}$ be the domain of the variables: $x_i \in \mathcal{D}$. A state of $\mathcal{P}$ is defined by the values of the program variables at a specific program point: $\mathcal{S} = \mathcal{L} \times \mathcal{D}^n$. The set $\mathcal{L}$ is the finite *set of all program points. We use the variable l to indicate the current program point: it can be viewed as the program counter.*

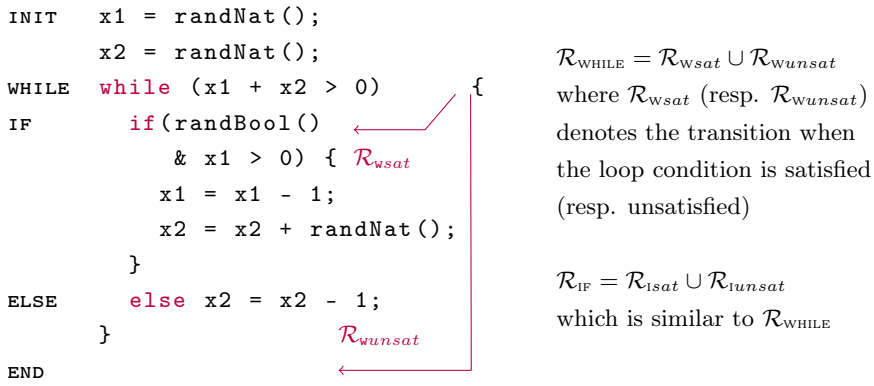


FIGURE 1.2: The program *balls*. Put x_1 red balls and x_2 white balls in a bag. Pick a ball randomly. If it is red then replace it by as many white balls as you want. Else if it is white then just remove it from the bag. Again, pick a ball randomly and repeat the process. Can we infinitely keep picking a ball from the bag?

Notation 1.1 (The [...] notation). *Let* $\mathcal{R} = \{((l,x),(l',x')) \in (\mathcal{L} \times \mathcal{D}^n)^2 | l = ? \wedge l' = ? \wedge c_1(x,x') \wedge \ldots \wedge c_m(x,x')\}$ *where* c_i *are constraints linking* x *with* x'*. For the sake of reading, we simply denote* $\mathcal{R} = [l = ?,\ l' = ?,\ c_1(x,x'),\ \ldots,\ c_m(x,x')]$*.*

Example 1.1 (The program *balls*). *See Figure 1.2. The program variables are* x_1, x_2 *and* x_3 *where* x_3 *serves as a temporary variable for the evaluation of the branching condition at program point* IF*. The program* balls *can be formalized as the transition system* $\mathcal{P}_b = (\mathcal{S}_b, \mathcal{I}_b, \mathcal{R}_b)$ *described in the following.*

The set of states is $\mathcal{S}_b = \mathcal{L}_b \times \mathcal{D}_b^3$*. We choose the set of program points* $\mathcal{L}_b = \{\text{INIT}, \text{WHILE}, \text{IF}, \text{ELSE}, \text{END}\}$ *as shown in Figure 1.2. The domain* $\mathcal{D}_b$ *of the program variables is* $\mathcal{D}_b = \mathbb{N}$*.*

The set of initial states is $\mathcal{I}_b = \{(l,(x_1,x_2,x_3)) \in \mathcal{L}_b \times \mathbb{N}^3 | l = \text{INIT}\}$*.*

The transition relation is $\mathcal{R}_b = \bigcup_{i \in \mathcal{L}_b} \mathcal{R}_i$ *where* $\mathcal{R}_i$ *denotes the transition at program point* i*. At points where the program may fork, for example for loops and conditional branchings, we have to consider the different possible paths as shown in Figure 1.2.*

Thus $\mathcal{R}_b = \mathcal{R}_{\text{INIT}} \cup \mathcal{R}_{\text{WHILE}} \cup \mathcal{R}_{\text{IF}} \cup \mathcal{R}_{\text{ELSE}} \cup \mathcal{R}_{\text{END}}$ *where:*

$$\mathcal{R}_{\text{INIT}} = \left\{ \Big((l, (x_1, x_2, x_3)), (l', (x'_1, x'_2, x'_3)) \Big) \in (\mathcal{L}_b \times \mathbb{N}^3)^2 | l = \text{INIT} \wedge l' = \text{WHILE} \right\}$$

shortened $\mathcal{R}_{\text{INIT}} = [l = \text{INIT}, l' = \text{WHILE}]$ *following Notation 1.1.*

$$\left.\begin{aligned} \mathcal{R}_{\text{WHILE}} = [l = \text{WHILE}, x_1 + x_2 > 0, l' = \text{IF}, x'_1 = x_1, x'_2 = x_2] \\ \cup [l = \text{WHILE}, \neg(x_1 + x_2 > 0), l' = \text{END}, x'_1 = x_1, x'_2 = x_2] \end{aligned}\right\} \mathcal{R}_{\text{WUNSAT}}$$

$$\left.\begin{aligned} \mathcal{R}_{\text{IF}} = [l = \text{IF}, x_3 = 1 \wedge x_1 > 0, l' = \text{WHILE}, x'_1 = x_1 - 1, x'_2 \geq x_2] \\ \cup [l = \text{IF}, \neg(x_3 = 1 \wedge x_1 > 0), l' = \text{ELSE}, x'_1 = x_1, x'_2 = x_2] \end{aligned}\right\} \mathcal{R}_{\text{IUNSAT}}$$

$$\mathcal{R}_{\text{ELSE}} = [l = \text{ELSE}, \neg(x_3 = 1 \wedge x_1 > 0), l' = \text{WHILE}, x'_1 = x_1, x'_2 = x_2 - 1]$$

The relation $\mathcal{R}_{\text{END}} = \varnothing$ *can be omitted.*

Hypothesis 1.1. *In this thesis, we consider that SImpL programs run on a machine with unbounded memory in the sense that there is always enough free space to store the values of the variables.*

We point out that the termination problem has been intensively studied for different formalisms. Notable mentions are termination analysis of logic programs [LSS97] [MR03], that of term and string rewrite programs [Gie+04][LM08] and that of imperative programs [CPR06][SMP10]. These formalisms all have the same expressive power. For example anything we can compute with the C programming language can be computed in Prolog and vice versa. Also, programs constructed under these different formalisms can all ultimately be viewed as transition systems.

Now what does "a program terminates" precisely mean? A common way to characterize termination is through the notion of well-foundedness.

Definition 1.3 (Well-founded relation, well-founded structure and co-well-foundedness). *A Well-Founded (WF) relation* $\prec \subseteq \mathcal{S} \times \mathcal{S}$ *is a binary relation with no infinite decreasing chains, that is with no infinite sequence of elements* S_i *such that* $S_1 \succ S_2 \succ \cdots \succ S_n \succ \cdots$ *We say that* $\mathcal{S}$ *equipped with* $\prec$*, that we note* $(\mathcal{S}, \prec)$*, is a WF structure. Dually,* $\prec$ *is coWell-Founded (cWF) if it has no infinite increasing chains.*

Example 1.2. *The structure* $(\mathbb{N}, <)$ *is WF. Indeed we cannot infinitely decrease in the natural numbers as we will eventually reach* 0*. For similar reasons the structure* $(Q_1, \dot{<})$ *where* $Q_1 = \{q \in \mathbb{Q} | q \geq 1\}$ *and* $q_1 \dot{<} q_2 \iff q_1 \leq \frac{q_2}{2}$ *is WF. However the*

structure $(\mathbb{Q}_+, \dot{<})$ *where where* $Q_+ = \{q \in \mathbb{Q} | q \geq 0\}$ *is* not *WF. Dually the structures* $(\mathbb{N}, >)$ *and* $(\mathbb{Q}, \dot{>})$ *are cWF whereas* $(\mathbb{Q}_+, \dot{>})$ *is not.*

Termination Characterization 1.1. *A program* $\mathcal{P} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ *terminates* if and only if *its transition relation restricted to the states that can be reached from the initial states has no infinite increasing chain. That is if and only if* $\mathcal{R} \cap \mathcal{R}_{acc}$*, where* $\mathcal{R}_{acc} = \mathcal{S}_{acc} \times \mathcal{S}_{acc}$ *in which* $\mathcal{S}_{acc}$ *denotes the set of states that are accessible from the initial states, is cWF.*

The restriction to $\mathcal{R}_{\text{acc}}$ is because some states may lead to non-termination but these states only matter if they can be reached from the initial states. Thus to prove termination of $\mathcal{P}$, we need to determine $\mathcal{R}_{\text{acc}}$ *and* prove the cWFness of $\mathcal{R} \cap \mathcal{R}_{\text{acc}}$. As the determination of $\mathcal{R}_{\text{acc}}$ is an additional challenge, we define a notion of termination that is *not* conditioned by the initial states.

Definition 1.4 ($\mathcal{S}$-universal termination). *We say that a program* $\mathcal{P} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ $\mathcal{S}$*-universally terminates if any possible execution of the program* $\mathcal{P}^{\#} = (\mathcal{S}, \mathcal{S}, \mathcal{R})$ *terminates. That is if* $\mathcal{P}$ *terminates when started from* any possible program point *with* any possible value *of the variables.*

The notion of $\mathcal{S}$-universal termination sligthly differs from the notion of universal termination encountered in the literature. A program $\mathcal{P}$ universally terminates if it terminates when started from any element of $\mathcal{I}$. Universal termination is also called unconditional termination or definite termination [Urb15, Definition 4.2.2] while $\mathcal{S}$-universal termination is also called mortality [Ben15, Definition 1].

The problem that is dual to universal termination is that of existential termination [SD94, Section 1.3.2] which is also called potential termination [Urb15, Definition 4.2.1]. It consists in finding at least one element of $\mathcal{I}$ that ensures termination. A similar problem is that of conditional termination. It consists in finding the weakest precondition for termination [BIK14]: we want to find the largest subset of $\mathcal{I}$ for which termination is guaranteed.

For all of these types of termination, as long as the set of initial states $\mathcal{I}$ matters that is $\mathcal{I} \subsetneq \mathcal{S}$ then we need to consider $\mathcal{R}_{\text{acc}}$. *In this thesis, we are mainly interested in* $\mathcal{S}$*-universal termination analysis.*

Termination Characterization 1.2 (For $\mathcal{S}$-universal termination). *The following statements are equivalent:*

(a) The program $\mathcal{P} = (\mathcal{S}, \mathcal{S}, \mathcal{R})$ *terminates. (Notice that* $\mathcal{I} = \mathcal{S}$*).*

(b) Its transition relation $\mathcal{R}$ is cWF.

(c) One of the k-th power $\mathcal{R}^k$ of $\mathcal{R}$ is cWF.

(d) Any k-th power $\mathcal{R}^k$ of $\mathcal{R}$ is cWF.

(e) The transitive closure $\mathcal{R}^+$ of $\mathcal{R}$ is cWF.

(f) The transitive closure $\mathcal{R}^+$ of $\mathcal{R}$ is disjunctively cWF, that is it can be written as a finite *disjunction of cWF relations.*

(g) There exists a function f from $(\mathcal{S}, \mathcal{R})$ to some WF structure $(\mathcal{W}, \prec)$[3] such that $\forall S, S' \in \mathcal{S} : S\mathcal{R}S' \implies f(S) \succ f(S')$. The function f is called a ranking function *for $\mathcal{P}$.*

(h) There exist a k-th power $\mathcal{R}^k$ of $\mathcal{R}$ and a function f from $(\mathcal{S}, \mathcal{R}^k)$ to some WF structure $(\mathcal{W}, \prec)$ such that $\forall S, S' \in \mathcal{S} : S\mathcal{R}^k S' \implies f(S) \succ f(S')$. The function f is called a slow ranking function *for $\mathcal{P}$.*

Notice that in Termination Characterization 1.2.g and 1.2.h, the function f *and* the WF structure $(\mathcal{W}, \prec)$ are both existentially quantified. Thus termination techniques based on these formulations have to find both $(\mathcal{W}, \prec)$ *and* f. Many techniques prealably define the WF structure they consider. For example a widely used WF structure is that of the natural numbers $(\mathbb{N}, <)$.

1.3 Termination of finite state programs

Consider the program *balls* of Example 1.1, Figure 1.2. The variables x_1 and x_2 are natural numbers: as such, they can take an infinite number of values: $0, 1, 2, 3 \ldots 10^{100} \ldots$ Let us investigate the case where the variables can only take a finite number of values. For example this applies when they are machine integers or, as we will see later, floating-point numbers.

Theorem 1.1 (Decidability of termination of finite state SImpL programs). *Consider a program written in the SImpL language of Definition 1.1. Let it be formalized by the transition system $\mathcal{P} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ as described in Definition 1.2. If $\mathcal{S}$ is finite then termination of $\mathcal{P}$ is decidable.*

Proof. *Consider the finite set of partial executions of length $|\mathcal{S}| + 1$ at most. Then $\mathcal{P}$ terminates if none of these executions goes through a same state twice.*

[3] Or equivalently to some cWF structure $(\mathcal{W}, \succ)$.

Techniques and tools have been developed from that result. For example [DKL15] proposes a technique implemented in a tool called JUGGERNAUT[4] that decides termination of any program written in restricted version of C. Instead of just checking the possible program executions one by one, JUGGERNAUT first expresses termination through the existence of ranking functions as in Termination Characterization 1.2.g. Then it calls a second-order SAT solver [KL14]. The solver implements various optimization techniques like Symbolic Bounded Model Checking or Genetic Programming.

Now, can we have a decision algorithm for Theorem 1.1 that is *efficient*? Indeed naive brute-force usually takes too much time to be useful in practice. For example if we simply want to enumerate all of the 2^{64} values that can be taken by a 64 bits signed integer, and even if we can enumerate $400,000$ million of values per second[5], it would still take us more than 17 months. In this thesis, we call an algorithm to be efficient when it answers in polynomial time at most. Can we devise algorithms that decide termination of finite state SImpL programs in polynomial time?

Hypothesis 1.2. *In this thesis, we suppose $P \subsetneq NP$.*

Theorem 1.2 (Following Theorem 1.1). *No algorithm can decide termination of $\mathcal{P}$ in polynomial time.*

Proof. *Consider the BOOLE programs of [Jon97, Definition 26.0.7]. (Claim 1) Deciding their termination is PSPACE-complete [Jon97, Corollary 28.1.4]. (Claim 2) The class of BOOLE programs are a subset of that of finite state SImpL programs. The theorem follows from (Claim 1), (Claim 2) and Hypothesis 1.2.*

Theorem 1.2 is a very strong theoretical limitation. It says that any sound *and* complete algorithm that tells whether a finite state SImpL program will terminate is doomed to answer in exponential time at least. If we want to have polynomial solutions then we have to sacrifice either soundness or completeness.

In this thesis, we ideally want to get always-sound answers in polynomial time. Thus our only option is to sacrifice completeness. Instead of just answering "YES" or "NO", we allow our algorithms to answer "I DON'T KNOW". Then the challenge consists in returning that third answer as rarely as possible. We keep soundness and sacrifice completeness through means of abstractions.

[4] https://github.com/blexim/synth/

[5] In 2016, an Intel Core i7 6950X processor can perform up to $317,900$ Million Instructions Per Second at 3.0 GHz.

1.4 Abstractions and termination

When facing a hard problem, we often try to transform it into an easier one. Sometimes the transformations preserve the problem: that is reformulation. Some other times, they may change it: that is abstraction. Here we refer to abstraction in its most general sense: *over*-approximation or *under*-approximation. In any cases, the abstraction must be done in a way that allows us to give sound answers for the original problem. First we start with over-approximations.

Definition 1.5 (Program over-approximation). *A program* $\mathcal{P}^{\#} = (\mathcal{S}^{\#}, \mathcal{I}^{\#}, \mathcal{R}^{\#})$ *is an over-approximation of a program* $\mathcal{P} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ *if any execution of* $\mathcal{P}$ *is also an execution of* $\mathcal{P}^{\#}$*. That is* $\mathcal{P}^{\#}$ *over-approximates* $\mathcal{P}$ *if* $\mathcal{S} \subseteq \mathcal{S}^{\#}, \mathcal{I} \subseteq \mathcal{I}^{\#}$ *and if* $\mathcal{R}^{\#}$ *is such that* $\forall S, S' \in \mathcal{S} : S \mathcal{R} S' \implies S \mathcal{R}^{\#} S'$*.*

Example 1.3. *See Figure 1.3.*

```
int x = 10;
while(x > 0) {
  x = x - 1;
}
```

```
x = randRat();
while(x >= 0) {
  x = randRatLeq(x - 1);
      // a random rational number
      // less or equal to x - 1
}
```

FIGURE 1.3: A simple Java program $\mathcal{P}_s$ (left) and an over-approximation of it to the rationals $\mathcal{P}_s^{\#}$ (right)

Suppose we want to analyze termination of some rather complex program. We can consider an over-approximation of it that is *easier or quicker to analyze* and then infer termination. Indeed a program and an over-approximation of it are related as follows in regard to their termination.

Proposition 1.1 (Program over-approximation and termination). *Given a program* $\mathcal{P}$ *and an over-approximation of it* $\mathcal{P}^{\#}$*, if* $\mathcal{P}^{\#}$ $\mathcal{S}$*-universally terminates then so does* $\mathcal{P}$*.*

Proof. *Derived from Definition 1.5.*

Notice that the use of program over-approximations may introduce incompleteness. Indeed if $\mathcal{P}^{\#}$ $\mathcal{S}$-universally terminates then we return "YES $\mathcal{P}$ terminates".

Otherwise we cannot conclude anything regarding termination of $\mathcal{P}$: we return "I DON'T KNOW whether $\mathcal{P}$ terminates".

In addition to over-approximating $\mathcal{P}$, we can also under-approximate its Space of Ranking Functions $SRF(\mathcal{P})$. Instead of considering all the possible ranking functions, we can only consider the ones that are *easy or quick to find*. That is, instead of considering $SRF(\mathcal{P})$ we only consider $SRF^{\#}(\mathcal{P})$ such that $SRF^{\#}(\mathcal{P}) \subseteq SRF(\mathcal{P})$. Again the use of ranking function under-approximations may introduce incompleteness. If we manage to exhibit ranking functions in $SRF^{\#}(\mathcal{P})$ then $\mathcal{P}$ terminates: we return "YES $\mathcal{P}$ terminates". Otherwise we return "I DON'T KNOW whether $\mathcal{P}$ terminates".

Program over-approximation and ranking function under-approximation are related as follows.

Proposition 1.2 (Program over-approximation and ranking functions). *Given a program $\mathcal{P} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ and a corresponding over-approximation $\mathcal{P}^{\#} = (\mathcal{S}^{\#}, \mathcal{I}^{\#}, \mathcal{R}^{\#})$, any ranking function for $\mathcal{P}^{\#}$ restricted to $\mathcal{S}$ is also a ranking function for $\mathcal{P}$.*

Another way to interpret Proposition 1.2 is to say that program over-approximation implies ranking functions under-approximation. With respect to the relation of inclusion $\subseteq$, the bigger the program over-approximation, the smaller the space of induced ranking functions. This is illustrated in Figure 1.4.

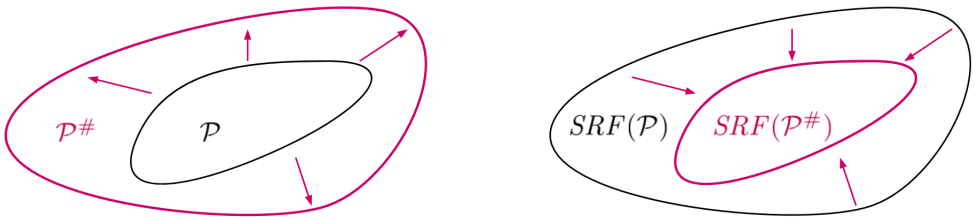


FIGURE 1.4: A program $\mathcal{P}$, an over-approximation $\mathcal{P}^{\#}$ of it and their spaces of ranking functions

Some termination proof techniques over-approximate the program to analyze with new ones and search the whole space of ranking functions in the over-approximations [LJB01]. Other techniques analyze the program as is but restrict the study to specific classes of ranking functions [CSZ13]. Yet other ones use both, for example by performing a ranking functions under-approximation after a program over-approximation

[PR04a]. The important thing is to remain sound throughout these processes. Notably we have to be careful with program *under*-approximations. Indeed in the context of $\mathcal{S}$-universal termination, they might lead to unsound analysis.

We hope we could show how central are abstractions to termination analysis. A very general approach for devising termination proof techniques is as follows. First we delimit the class of programs we want to analyze. Then we choose the Termination Characterization we will rely on. Then we smartly and soundly abstract. Last we analyze the obtained abstractions. That is the scheme proposed in [CC12] as a framework of Abstract Interpretation [CC77][CC10]. Then the framework was revised in [Urb15] with an emphasis on definite, universal termination. Notably, it characterizes termination as an existence of ranking functions from the program states to the ordinals less than ω^ω. The considered abstract domain is that of segmented ranking functions.

1.5 Termination of infinite state programs

We start by pointing out that the procedure that is sketched in the proof of Theorem 1.1 and that decides termination of *finite* state SImpL programs *cannot* be applied to *infinite* state SImpL programs. Indeed that procedure basically checks one after another each possible execution: if the set of state if infinite then the set of possible executions can be infinite. Question arises: is there another way to decide termination of infinite state SImpL programs?

Theorem 1.3 (Undecidability of termination of infinite state SImpL programs, Undecidability of the Halting problem). *Consider a program written in the SImpL language of Definition 1.1. Let it be formalized by the transition system $\mathcal{P} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ as described in Definition 1.2. If $\mathcal{S}$ is infinite then termination of $\mathcal{P}$ is undecidable.*

Proof. *SImpL programs can simulate Turing machines. Termination of Turing machines is undecidable in the general case [Chu36][Tur37][Str65]*[6].

We emphasize that Theorem 1.3 does not mean we are *always unable* to prove termination of infinite state SImpL programs. Instead, it means we are *unable to*

[6] Amusingly there is a poem version of the proof: http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html

always do so. Differently stated, any algorithm we devise for deciding termination will always fail for at least one infinite state SImpL program.

Despite Theorem 1.3, working with infinite state programs has its benefits. For example contrarily to fixed-width machine integers, we do not need to worry about overflows when handling integers from $\mathbb{Z}$. Also contrarily to floating-point numbers, we do not need to worry about rounding errors when handling numbers from $\mathbb{R}$. Thus a way to overcome the difficulties of computing with fixed-width machine numbers is to consider abstract computations within the infinite sets $\mathbb{Z}, \mathbb{Q}$ or $\mathbb{R}$.

Until recently, literature mainly focused on termination analysis of programs that compute within $\mathbb{Z}, \mathbb{Q}$ or $\mathbb{R}$. We survey some of the techniques that have been produced in the rest of this section. They can roughly be categorized into two approaches: the global approach and the local approach [CPR11]. On one hand, the global approach considers all the possible program executions in one big swipe. It attempts to find a *single* termination argument for all of them. On the other hand, the local approach considers the possible program executions separately. It attempts to find *a set* of termination arguments: one termination argument per considered group of program executions.

More precisely, techniques that use the global approach observe the evolution of the program after every *single transition*: they rely on Termination Characterization 1.2.b and 1.2.g[7]. Those that use the local approach observe the evolution of the program throughout *all the possible successive transitions*: they rely on Termination Characterization 1.2.d, 1.2.e and 1.2.f.

1.5.1 Global approach for proving termination

Given an infinite state SImpL program, we want to find global ranking functions for it. As their existence is undecidable in the general case due to Theorem 1.3, literature restricted the question to specific classes of ranking functions for specific classes of programs.

[7] Actually we can put in this category any techniques that observe the evolution of the program after every *fixed successive transitions*. Indeed they attempt to find *single* termination arguments expressed as *single* slow ranking functions that are valid for all the program executions. They rely on Termination Characterization 1.2.c and 1.2.h.

One of the most studied class of abstract programs in the literature is the class of Simple Loops. With only slight differences, Simple Loops are also called Single-path Linear-Constraint loops [BG14, Subsection 2.2], Linear Simple Loops [CFM15, Definition 1] or Linear Arithmetic Simple While loops in [PR04a, Definition 1].

Definition 1.6 (Simple rational (resp. integer) Loops, $\mathrm{SL}_{\mathbb{Q}}$ (resp. $\mathrm{SL}_{\mathbb{Z}}$)). *We call Simple rational (resp. integer) Loop a loop of the form* `while` $(Dx \leq d)$ `do` $V \begin{pmatrix} x \\ x' \end{pmatrix} \leq v$. *The column vector* $x = \begin{pmatrix} x_1 & \cdots & x_n \end{pmatrix}^T \in \mathbb{Q}^{n\times 1}$ *(resp.* $\mathbb{Z}^{n\times 1}$*) represents the variables at the beginning of an iteration. The primed equivalent* x' *represents the variables at the end of an iteration after they have been updated. The constants* D, d, V *and* v *are rational (resp. integer) matrices such that* $D \in E^{m\times n}, d \in E^{m\times 1}, V \in E^{t\times 2n}, v \in E^{t\times 1}, m \in \mathbb{N}, t \in \mathbb{N}, m > 0, t > 0$ *for* $E = \mathbb{Q}$ *(resp.* $E = \mathbb{Z}$*).*

Using our formalization of programs as transition systems as in Definition 1.2, a Simple Loop is a program that has a transition relation $\mathcal{R}$ defined by linear constraints between the program variables. Also it only has a single program point which is placed at the beginning of the `while` instruction: all the updates in the body of the loop are performed simultaneously. Thus the space of possible states $\mathcal{S} = \mathcal{L} \times \mathcal{D}^n$ can be simplified to $\mathcal{S} = \mathcal{D}$ since $|\mathcal{L}| = 1$. That is for Simple Loops, we can give no consideration to the program points and define a state only by the values of the program variables.

Example 1.4. *Consider the loop* $\mathcal{P}_l$ *presented in Figure 1.5. The variables range over the integers. A Simple Loop* $\mathcal{P}_l^{\#}$ *that over-approximates* $\mathcal{P}_l$ *can be obtained by conjuncting polyhedral invariants computed at program points* ♣ *and* ♠*. Polyhedral invariants can be obtained by using for example Abstract Interpretation [CC77].*

Unfortunately termination of $\mathrm{SL}_{\mathbb{Z}}$ was shown to be at least EXPSPACE-hard [BGM12, Theorem 6.4]. That is we cannot decide existence of ranking functions for $\mathrm{SL}_{\mathbb{Z}}$ in polynomial time in the general case. However we can do so for restricted classes of ranking functions. Then, termination of Simple rational Loops is *probably* undecidable. But again, we can decide existence of specific kind of ranking functions for $\mathrm{SL}_{\mathbb{Q}}$ in polynomial time.

Definition 1.7 (Linear Ranking Functions (LRF) for Simple Loops). *Let* $\mathcal{P}$ *be a Simple integer (resp. rational) Loop that has the column vector* $x \in \mathbb{Z}^{n\times 1}$ *(resp.* $x \in \mathbb{Q}^{n\times 1}$*) as variables and the binary relation* $\mathcal{R}$ *as transition relation. The linear*

```
x1 = randInt();                    x1 = randInt();
x2 = randInt();                    x2 = randInt();
while (x1 > 0 & x2 > 0) {          while (true) {
♣   if(randBool())                   ♣ x1 ≥ 0 & x2 ≥ 0
      x1 = x1 - 1;                   ♠ x1′ + x2′ ≤ x1 + x2 - 1
    else                           }
      x2 = x2 - 1;
♠ }
```

FIGURE 1.5: An integer loop $\mathcal{P}_l$ (left) and an abstraction of it to a Simple integer Loop $\mathcal{P}_l^{\#}$ (right).

function $f(x) = cx$*, where* $c \in \mathbb{Q}^{1\times n}$ *is a constant row vector, is a LRF for* $\mathcal{P}$ *if* $\forall x, x' : x\mathcal{R}x' \implies f(x') \geq 0 \wedge f(x) \geq f(x') + 1$[8].

Example 1.5 (Continuing Example 1.4). *The function* $f_l(x_1, x_2) = x_1 + x_2$ *is a LRF for the* $SL_{\mathbb{Z}}$ $\mathcal{P}_l^{\#}$*. Indeed it is of linear form and we easily verify that* $\forall x_1, x_2, x_1', x_2' \in \mathbb{Z} : (x_1, x_2)\mathcal{R}_l^{\#}(x_1', x_2') \implies f_l(x_1, x_2) \geq 0 \wedge f_l(x_1, x_2) \geq f_l(x_1', x_2') + 1$*. As* $\mathcal{P}_l^{\#}$ *is an over-approximation of* $\mathcal{P}_l$ *then* f_l *is also a LRF for* $\mathcal{P}_l$ *due to Proposition 1.2. Hence* $\mathcal{P}_l$ *terminates*[9]*.*

The definition of LRF actually varies across the literature depending on the considered WF structure. Compare for example the WF structure considered in [PR04a, Theorem 1] with that in [BM13, Definition 2.1] and [BG14, Section 2.3]. Thus LRFs are sometimes confused with *Affine* Ranking Functions.

Definition 1.8 (Affine Ranking Functions (ARF) for Simple Loops, following Definition 1.7). *The affine function* $f(x) = cx + d$*, where* $c \in \mathbb{Q}^{1\times n}$ *is a constant row vector and where* d *is a rational constant, is an ARF for* $\mathcal{P}$ *if* $\forall x, x' : x\mathcal{R}x' \implies f(x') \geq 0 \wedge f(x) \geq f(x') + 1$.

Existence of ARFs for $\mathrm{SL}_{\mathbb{Q}}$ *is* decidable and the decision problem is known to lay in P [PR04a, Theorem 1]. Then, existence of LRFs for $\mathrm{SL}_{\mathbb{Z}}$ *is* decidable and the decision problem is known to lay in coNP [BG14, Theorem 3.12].

In this thesis, we will mainly consider Simple Loops and ARFs for the global approach for proving termination. In the remaining paragraphs of this section, we

[8] It means f maps the program states to the WF structure $(\mathbb{Q}^+, \prec)$ where $\mathbb{Q}^+ = \{q > 0 | q \in \mathbb{Q}\}$ and $q' \prec q \iff q \geq q' + 1$.

[9] Under the condition that $\mathcal{P}_l$ always reaches ♠ from ♣. It is the case here but it may have not been so if there was for example an inner loop placed between these two program points.

will cite a few results for different classes of programs and different classes of ranking functions. Our goal is to situate Simple Loops and LRFs in the literature. Also we aim to show that they are central pieces to many termination analysis techniques as judged by the number of work that originated from them.

A recent extension of the class of LRFs is the class of Eventual LRFs (ELRF) [BM13]. An ELRF is a function that becomes a LRF after $k \in \mathbb{N}$ successive transitions during which some linear function increases. LRFs are clearly a strict subclass of ELRFs as a LRF is an ELRF with $k = 0$. Existence of ELRFs for $\mathrm{SL}_{\mathbb{Q}}$ *is* decidable and the decision problem is known to lay in P [BM13, Theorem 3.16]. Even more recently, the notion of ELRFs has been extended to the notion of l-depth ELRFs [LZF16]; additional references for this kind of functions are [LH15][BG17].

Another extension of the class of LRFs is the class of Lexicographic LRFs (LLRFs)[10]. Also an extension of the class of Simple Loops is the class of Multi-path Linear-Constraint loops (MLC) [BG14, Subsection 2.2]. Existence of LLRFs for rational MLCs *is* decidable and the decision problem is known to lay in P [BG14, Theorem 5.37]. Also, existence of LLRFs for integer MLCs *is* decidable and the decision problem is known to lay in coNP [BG14, Theorem 5.24]. Last, in the same way the class of LRFs can be extended to the class of ELRFs, we can extend the class of LLRFs to the class of Lexicographic ELRFs [ZLW16].

Another class of programs that generalizes Simple Loops is the class of polynomials [BMS05b][Luc05]. Another class of ranking functions that generalizes the class of *single* LLRFs is the class of multi-dimensional affine ranking functions [Ali+10].

1.5.2 Local approach for proving termination

Given an infinite state SImpL program, we want to shift the cWFness analysis of $\mathcal{R}$ to *separate* cWFness analyses of a *set* of simpler relations. To achieve that we use Termination Characterization 1.2.f and show that the transitive closure $\mathcal{R}^+$ of its transition relation is a finite disjunction of cWF relations. That characterization comes from the Infinite Ramsey Theorem, see Appendix D.

Notice that Termination Characterization 1.2.f can also be used to characterize $\mathcal{S}$-existential *non*-termination which is the problem dual to $\mathcal{S}$-universal termination.

[10] As it is the case for LRFs, the definition of LLRFs also varies across the literature depending on the considered WF structure. Compare for example the WF structure considered in [BG15, Definition 1] with that in [BMS05a, Definition 5] and [BG15, Definition 2].

Indeed if there is one disjunct of $\mathcal{R}^+$ that is not cWF then non-terminating execution exists [PM06][Pay08][PS09].

When using the local approach, two questions arise. First how do we compute the set of disjuncts that compose $\mathcal{R}^+$? Then how do we analyze their cWFness? For the first question, $\mathcal{R}^+$ may not be computable. For the second question, the cWFness of a relation is undecidable in the general case. However there are special cases for which we can provide workable answers to these two questions.

Definition 1.9 (Constraint $D^\prec(x,x')$ of $\prec$-decrease). *Consider the vector of program variables $x = (x_1 \dots x_n)^T, x_i \in E$ where E is a set of numbers, and its value $x' = (x'_1 \dots x'_n)^T$ after an application of the transition relation. Let $(E, \prec)$ be a WF structure. We say that the constraint $D^\prec(x,x')$ is a constraint of $\prec$-decrease if it is a (possibly void) conjunction of $x'_? \prec x_?$ and $x'_? \preceq x_?$ where $x_? \in \{x_1, \dots, x_n\}$ and $x'_? \in \{x'_1, \dots, x'_n\}$: $D^\prec(x,x') \equiv \left(\bigwedge x'_? \prec x_?\right) \wedge \left(\bigwedge x'_? \preceq x_?\right)$.*

Example 1.6. *The list of all possible constraints of $\prec$-decrease for 2 program variables are:* False, $x'_1 \prec x_1, x'_1 \preceq x_1, x'_1 \prec x_2 \dots$ $x'_1 \prec x_1 \wedge x'_1 \prec x_2, x'_1 \prec x_1 \wedge x'_1 \preceq x_2 \dots$ True. *The constraint* False *denotes an impossible constraint whereas* True *results from the void conjunction.*

Lemma 1.1. *Let $\mathcal{P}_\prec = (\mathcal{S}, \mathcal{S}, \mathcal{R}_\prec)$ be a program as specified in Definition 1.2. Let its transition relation $\mathcal{R}_\prec$ link the program variables through constraints of $\prec$-decrease. Then $\mathcal{R}^+_\prec$ is computable.*

Proof. *The relation $\prec$ is computable in the sense that we can always decide whether two elements are in relation through $\prec$. Also the set of all possible constraints of $\prec$-decrease finite and stable by composition. It remains so even when considering all the possible constraints on the program points.*

Theorem 1.4 (Following Lemma 1.1, [LJB01]). *Disjunctive cWFness of $\mathcal{R}^+_\prec$ is decidable. That is termination of $\mathcal{P}_\prec$ is decidable. Each disjunct of $\mathcal{R}^+_\prec$ is cWF* if and only if *those of them that are idempotent all have at least one constraint of strict self-decrease $x'_i \prec x_i$.*

Example 1.7 (Continuing Example 1.1). *We want to analyze the cWFness of $\mathcal{R}_b$. We first abstract it to the relation $\mathcal{R}^<_b$ that only uses decreasing constraints between the program variables. The considered WF structure is $(\mathbb{N}, <)$. We have*

$\mathcal{R}_b^< = \mathcal{R}_{\text{INIT}}^< \cup \mathcal{R}_{\text{WHILE}}^< \cup \mathcal{R}_{\text{IF}}^< \cup \mathcal{R}_{\text{ELSE}}^<$ *where:*

$$\begin{aligned}
\mathcal{R}_{\text{INIT}}^< =& [l = \text{INIT}, l' = \text{WHILE}] \\
\mathcal{R}_{\text{WHILE}}^< =& [l = \text{WHILE}, l' = \text{IF}, x_1' \leq x_1, x_2' \leq x_2] \quad \Big\} \mathcal{R}_{\text{Wsat}}^< \\
&\cup [l = \text{WHILE}, l' = \text{END}, x_1' \leq x_1, x_2' \leq x_2] \Big\} \mathcal{R}_{\text{Wunsat}}^< \\
\mathcal{R}_{\text{IF}}^< =& [l = \text{IF}, l' = \text{WHILE}, x_1' < x_1] \quad \Big\} \mathcal{R}_{\text{Isat}}^< \\
&\cup [l = \text{IF}, l' = \text{ELSE}, x_1' \leq x_1, x_2' \leq x_2] \Big\} \mathcal{R}_{\text{Iunsat}}^< \\
\mathcal{R}_{\text{ELSE}}^< =& [l = \text{ELSE}, l' = \text{WHILE}, x_1' \leq x_1, x_2' < x_2]
\end{aligned}$$

Now we compute the transitive closure $\mathcal{R}_b^{<^+}$ *of* $\mathcal{R}_b^<$*. We have* $\mathcal{R}_b^{<^+} = \bigcup_{i=1}^{33} \mathcal{T}_i$ *where:*

$$\mathcal{T}_1 = \mathcal{R}_{\text{INIT}}^< \quad \cdots \quad \mathcal{T}_{33} = [l = \text{ELSE}, l' = \text{ELSE}, x_1' < x_1]$$

Last we select among these $\mathcal{T}_i$ *the ones that are idempotent. There are six of them:*

$$\begin{aligned}
\mathcal{T}_{idm_1} =& [l = \text{WHILE}, l' = \text{WHILE}, x_1' \leq x_1, x_2' < x_2] \\
\mathcal{T}_{idm_2} =& [l = \text{WHILE}, l' = \text{WHILE}, x_1' < x_1] \\
\mathcal{T}_{idm_3} =& [l = \text{IF}, l' = \text{IF}, x_1' \leq x_1, x_2' < x_2] \\
\mathcal{T}_{idm_4} =& [l = \text{IF}, l' = \text{IF}, x_1' < x_1] \\
\mathcal{T}_{idm_5} =& [l = \text{ELSE}, l' = \text{ELSE}, x_1' \leq x, x_2' < x_2] \\
\mathcal{T}_{idm_6} =& [l = \text{ELSE}, l' = \text{ELSE}, x_1' < x_1]
\end{aligned}$$

As each of the $\mathcal{T}_{idm_j}$ *has a strict self-decrease then each of the* $\mathcal{T}_i$ *is cWF by Theorem 1.4:* $\mathcal{R}_b^{<^+}$ *is disjunctively cWF. Thus* $\mathcal{R}_b^<$ *is cWF by equivalence between Termination Characterization 1.2.b and 1.2.f. So is* $\mathcal{R}_b$ *by Proposition 1.1. The program* balls $\mathcal{S}$*-universally terminates: it terminates when launched from any program point with any values of the program variables.*

The idea of abstracting $\mathcal{R}$ to $\mathcal{R}_\prec$ and then analyzing $\mathcal{R}_\prec^+$ was first investigated by [LJB01] under the name of Size-Change (SC) principle. Then [CLS05] generalized the approach to monotonicity constraints. *In this thesis, we will mainly consider these two frameworks for the local approach for proving termination.*

Another line of research on the local approach originated from the concept of Transition Invariant (TI) [PR04b]. A TI is simply an over-approximation of $\mathcal{R}^+$.

It can be obtained through Transition Predicate Abstraction (TPA) [PR07][PR11]: we abstract in a iterative manner, and to a set Φ of predefined constraints. This is in contrast to SC which abstracts only once, and to the precise set of decreasing constraints.

Notice that $\mathcal{R}^+_\prec$ of SC *is* a TI. Indeed it is an over-approximation of $\mathcal{R}^+$. Actually when using TPA, we can fix Φ to a specific set of constraints [HJP10, Definition 25] so that the TI we get is a subset of the TI we would have got using SC [HJP10, Lemma 36]. That is we can fix Φ so that TPA can decide termination by SC, and potentially more [HJP10, Theorem 37]. However the great flexibility of TPA renders the approach hard to study. Notably the class of programs for which it can decide termination is broad but vague. Also complexity results are yet to be produced. That motivates us to stick to the simple but just as powerful framework of SC and its derivatives.

1.6 Conclusion

The art of proving program termination has come a long way since its beginnings. We are now able to automatically check termination of reasonably large and non-trivial programs thanks to refined techniques [SMP10][Coo+13]. They smartly work around various undecidability and complexity limitations mainly by means of abstractions. However, progresses have yet to be made for specific points. This is notably the case for termination analysis of programs that handles fixed-width machine numbers. Indeed most of the techniques found in the literature consider programs that work with infinite sets of numbers and that perform error-free computations.

In this thesis, we address termination analysis of programs that manipulate floating-point numbers. These programs have a finite space of states and perform computations that are affected by rounding errors. We develop new termination proof techniques or adapt the existing ones in order to analyze them. In Chapter 3, we develop a new technique that bridges termination analysis of floating-point programs with termination analysis of rational ones. Then we go deeper by investigating the two main approaches for proving termination as presented in this chapter. In Chapter 4, we study the global approach for termination analysis of floating-point programs. In Chapter 5, we study the local one.

Chapter 2

Floating-Point Numbers

Abstract. This chapter introduces Floating-Point (FP) numbers which are a widely used format for representing numbers on machines. One of the first modern machines known to have implemented FP arithmetic goes back to 1941[1]. Since then, a lot of work have been conducted regarding the analysis of FP computations. In this chapter we present the notions on FP numbers that we need throughout the thesis.

Résumé. Ce chapitre présente les nombres flottants. Ce sont des représentations en machine des nombres réels qui sont largement utilisées. L'une des premières machines modernes connues pour avoir implémenté l'arithmétique flottante remonte à 1941. Depuis lors, beaucoup de travaux ont été menés concernant la vérification de calculs flottants. Dans ce chapitre, nous présentons les notions relatives aux nombres flottants dont nous aurons besoin tout au long du manuscrit.

2.1 Introduction

FP numbers are *approximative* representations of real numbers. Indeed FP numbers are stored on a finite and fixed amount of memory: commonly on 32 or 64 bits. Thus real numbers have to be *rounded* to fit into the limited available memory.

Most of the time, these rounding errors are so small that many programmers manipulate FP numbers as if they were manipulating real numbers. For example not all programmers are aware that the Java constant `Math.PI` is *not* the real number π but only an approximation of it on 64 bits. The fact is these rounding errors can cause disasters when not handled correctly.

In 1991, an American Patriot missile failed to intercept an incoming Iraqi Scud missile due to FP rounding errors. That killed 28 soldiers, injured around 100 other

[1] The $Z3$ computer [Cer81].

people and caused millions of dollars of financial loss[2]. In 1992, a FP bug impacted the German Parliament makeup[3]. A party has seated in Parliament though it should not have if the results of the FP computation were correctly printed. Similarly in Microsoft Excel 2007, a few FP numbers were not correctly printed[4]. For example the formula `=850*77.1` printed 100000 instead of 65535. We may think that spreadsheet errors are not very harmful but they can actually cost millions of dollars[5].

The rest of this chapter is organized as follows. Section 2.2 introduces the basics. Section 2.3 presents IEEE 754 which is a widely used standard for the implementation of FP computations. Section 2.4 discusses the difficulties encountered when verifying FP computations. Section 2.5 gives a few properties of FP arithmetic. Section 2.6 concludes.

2.2 Floating-point numbers basics

Consider the two Java programs *pDec* and *pSqrt* presented in Figure 2.1. Do they always terminate for any possible value supplied by the user through the `input` function? First let us suppose that we do not use the Java type `float` but rational or real numbers. In that case, both programs always terminate. Indeed the variable x of *pDec* cannot infinitely be decreased by $\frac{1}{10}$ while remaining strictly positive. Similarly the difference $x_M - x_m$ in *pSqrt* cannot infinitely be divided by 2 while remaining strictly greater than some strictly positive quantity d.

Now let us use the Java type `float` in *pDec* and *pSqrt*: both programs do not always terminate for any possible input. Indeed *pDec* terminates if the supplied x is for example 10 but does not if it is 10^7. Similarly *pSqrt* terminates if the supplied d is for example 10^{-3} but does not if it is 10^{-9}. To explain these surprising changes of behaviors, let us give some basic notions on FP numbers.

Definition 2.1 ($\mathbb{F}_{\beta,p,e_{min},e_{max}}$). *A Floating-Point (FP) number $\hat{x}$ is a particular rational number defined as $\hat{x} = (-1)^s \hat{m} \beta^e$ where $s \in \{0,1\}, \beta \in \mathbb{N}, \beta \geq 2, e \in \mathbb{Z}, e \in [e_{min}, e_{max}]$ and $\hat{m} = m_0.m_1 m_2 \cdots m_{p-1}, m_i \in \mathbb{N}, 0 \leq m_i \leq \beta - 1$ such that $m_0 = 0$ only if $|\hat{x}| < \beta^{e_{min}}$ in which case e is set to e_{min}, otherwise $m_0 \neq 0$ in which case $e \in [e_{min}, e_{max}]$. The number β is called* radix. *The number p is called*

[2] http://www.gao.gov/products/IMTEC-92-26
[3] http://catless.ncl.ac.uk/Risks/13/37#subj4.1
[4] https://blogs.office.com/2007/09/25/calculation-issue-update/
[5] http://ww2.cfo.com/spreadsheets/2014/10/spreadsheet-error-costs-tibco-shareholders-100m/

```
float x = input();
while (x > 0) {
  x = x - 0.1;
}
```

(a) Simply decreasing x

```
float xm = 1, xM = 2;
float d = input(); // d > 0
do {
  float x = (xm + xM) / 2;
  float hx = x*x - 2;
  if (hx < 0)  xm = x;
  else xM = x;
} while (xM - xm > d);
```

(b) Computing an interval $[x_m, x_M]$ of length d approximating $\sqrt{2}$ using the dichotomy method

FIGURE 2.1: Two Java programs: *pDec* (left) and *pSqrt* (right). If the variables are rationals or reals then both programs always terminate. However they may not terminate when using the Java type `float` due to the rounding errors.

precision. *The number $\hat{m}$ is called* significand. *To β, p, e_{min} and e_{max} correspond the set $\mathbb{F}_{\beta,p,e_{min},e_{max}}$ of FP numbers.*

We point out that two zeros are defined, depending on the value of s. However for simplicity, we consider that they both refer to a single, unsigned zero.

Definition 2.2 (Rounding mode, rounding function, overflow). *Given a real number x and the two FP numbers $\hat{x}_1$ and $\hat{x}_2$ that are closest to x and that straddle it, $\hat{x}_1 \leq x \leq \hat{x}_2$, the rounding mode defines the choice to make between $\hat{x}_1$ and $\hat{x}_2$ when approximating x.*

In the rounding mode to-nearest-ties-to-even*, that we shorten* to-nearest*, we choose the one that is closest to x. In case of tie, we choose the one that has the last unit in the significand $\hat{m}$ even. We denote ζ_n the corresponding rounding function. In the rounding mode* to-zero *we choose the one that is closest to 0 and we denote ζ_0 the corresponding rounding function. In the rounding mode* to-positive-infinity *we choose the one that is closest to $+\infty$ and we denote $\zeta_{+\infty}$ the corresponding rounding function. In the rounding mode* to-negative-infinity *we choose the one that is closest to $-\infty$ and we denote $\zeta_{-\infty}$ the corresponding rounding function.*

We say that there is an overflow *when the real number x to round is of such a large magnitude that it cannot be represented. Notably for the rounding mode to-nearest, overflow occurs when $|x| > (\beta - \frac{\beta^{-p+1}}{2})\beta^{e_{max}}$.*

Example 2.1 (The toy FP type `myfloat`). *Consider the toy FP type* ***myfloat*** *presented in Figure 2.2 and the real number* $x = 9\sqrt{2} = 12.7\cdots = (-1)^0 1.27\cdots 10^1$. *Using* ***myfloat****,* x *is rounded into* $\hat{x} = (-1)^0 1.3 \cdot 10^1 = 13$ *if the rounding mode is to-nearest-(and in case of tie then go to even). It is rounded into* $\hat{x} = (-1)^0 1.2 \cdot 10^1 = 12$ *if the rounding mode is to-zero.*

Now suppose we use `myfloat` in the programs *pDec* and *pSqrt* and that the rounding mode is to-nearest. If we supply 20 as value of x then *pDec* does not terminate since $20 - 0.1 = 19.9$ is rounded to 20 itself. Also if we supply 10^{-3} as value of d then *pSqrt* does not terminate since the tightest interval approximating $\sqrt{2}$ that we can obtain with `myfloat` is $[1.4, 1.5]$ which is of length 10^{-1}. Similar phenomena occur when the Java type `float` is used.

Now we introduce some terminology.

Definition 2.3 (Correct rounding). *Given a rational or real function* $\star$*, its FP match* $\circledast$[6] *and* n *FP numbers* $\hat{x}_1 \dots \hat{x}_n$ *taken as arguments, we say that* $\circledast$ *correctly rounds* $\star$ *if* $\forall \hat{x}_1 \dots \hat{x}_n : \circledast(\hat{x}_1 \dots \hat{x}_n) = \zeta(\star(\hat{x}_1 \dots \hat{x}_n))$.

Example 2.2. *Using* ***myfloat*** *and with the rounding mode set to to-nearest, the FP division* $\oslash$ *correctly rounds the rational division for the arguments* 1 *and* 3 *in that order if* $1 \oslash 3 = \zeta_n(\frac{1}{3}) = \zeta_n(0.333\cdots) = 0.3$. *The FP exponential function* $\circledcirc$ *does not correctly round the real exponential function for the argument* 2 *if* $\circledcirc^2 = 7.2$ *since* $\zeta_n(e^2) = \zeta_n(7.389\cdots) = 7.4$.

The precedence of FP functions is the same as that of their rational or real match. Thus the FP arithmetic operations are performed in the following order: $\odot$, $\oslash$, $\oplus$ and $\ominus$.

Definition 2.4 (Unit in the Last Place).[7] *Consider* $x \in \mathbb{R}$ *that is rounded into* $\hat{x} \in \mathbb{F}, \hat{x} = (-1)^s \hat{m} \beta^e$. *We call* Unit in the Last Place *(ULP) of* x *the weight of the least significant information unit*[8] *of* $\hat{m}$*:* $ulp(x) = \beta^{-p+1}\beta^e$. *In particular we point out that* $ulp(x) = \beta^{-p+1}\beta^{e_{min}} = ulp(0)$ *if* $\hat{x}$ *is a subnormal.*

Example 2.3. *Using* ***myfloat****,* $ulp(112) = 10, ulp(10) = 1$ *and* $ulp(0.6) = 0.1 = ulp(0)$.

[6] For example we denote / the rational division and $\oslash$ the FP one.

[7] The definition of ULP actually varies across the literature [Mul05]. Kahan has his own definition [Mul05, Definition 1]. So has Harrison and Muller [Mul05, Definition 2]. Then Goldberg uses yet another definition [Mul05, Definition 3]. These definitions are mostly equivalent except for the cases when x is around a power of β.

[8] For example bit if $\beta = 2$, trit if $\beta = 3$, digit if $\beta = 10$.

Definition 2.5 (β-ade).[9] *In a radix-β FP system, a β-ade B is an interval of real numbers such that $B =]-\beta^{e+1}, -\beta^{e}]$ or $B =]-\beta^{e_{min}}, \beta^{e_{min}}[$ or $B = [\beta^{e}, \beta^{e+1}[$ with $e_{min} \leq e \leq e_{max}$. We define the ULP u of B as the ULP of the FP numbers in B: $u = \beta^{-p+1}\beta^{e}$.*

Example 2.4. *For* `myfloat` *we have seven β-ades:* $]-1000, -100],]-100, -10],]-10, -1],]-1, 1[, [1, 10[, [10, 100[$ *and* $[100, 1000[$.

Definition 2.6 (Normal numbers, subnormal numbers, $s_{min}, n_{min}, n_{max}$). *If a FP number $\hat{x}$ is such that $|\hat{x}| \geq \beta^{e_{min}}$ then it is called a* normal number. *Else if $\hat{x}$ is such that $|\hat{x}| < \beta^{e_{min}}$ and $\hat{x} \neq 0$ then it is called a* subnormal number. *Call s_{min} the smallest positive subnormal, n_{min} the smallest positive normal and n_{max} the biggest positive normal.*

Example 2.5. *For* `myfloat` $s_{min} = \beta^{-p+1}\beta^{e_{min}} = 0.1$, $n_{min} = \beta^{e_{min}} = 1$ *and* $n_{max} = (\beta - \beta^{-p+1})\beta^{e_{max}} = 990$.

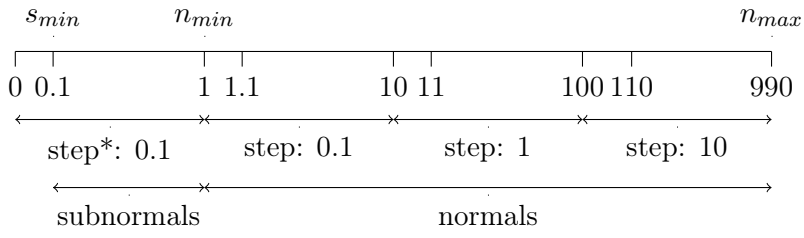


FIGURE 2.2: A toy FP type `myfloat` with $\beta = 10, p = 2, e_{min} = 0$ and $e_{max} = 2$. Symmetry to the origin for the negatives. Call $\mathbb{F}_t$ the corresponding set of FP numbers.

In a normal number, there are no leading zeros in the significand $\hat{m}$. Instead leading zeros in $\hat{m}$ are moved to the exponent. This guarantees the uniqueness of the representation. For example using `myfloat`, 7 would be written as $(-1)^0 7.0 \cdot 10^0$ instead of $(-1)^0 0.7 \cdot 10^1$. Subnormal numbers are numbers where the representation would result in an exponent that is below the minimum exponent e_{min}. Thus they are allowed to use leading zeros in $\hat{m}$. For example using `myfloat`, 0.7 would be written as $(-1)^0 7.0 \cdot 10^{-1}$ if leading zeros in $\hat{m}$ are not allowed. However since -1 is

[9] The notion is inspired from that of binade [Kah02][MNR03].

less than the minimum exponent $e_{min} = 0$ then 0.7 is written with the exponent set to e_{min} and with as much leading zeros in $\hat{m}$ as needed: $(-1)^0 0.7 \cdot 10^0$.

```
#include <fenv.h>
fesetenv(FE_DFL_DISABLE_SSE_DENORMS_ENV);
void subAndDiv(float a, float b) {
  if (a != b) {
    float c = 1 / (a - b);
  }
}
```

FIGURE 2.3: Since the support for subnormals is disabled, this C function can perform a division by zero. That is the case for example for $a = 2^{-125}$ and $b = 1.5 \cdot 2^{-126}$.

Subnormals provide the guarantee that the result of the FP subtraction of two different FP numbers is never rounded to zero. For example using `myfloat`, if subnormals are not implemented then the FP subtraction $10 \ominus 9.7$ would result in $\zeta(10 - 9.7) = \zeta(0.3) = 0$ since there is no FP number between 0 and 1. Such behavior can cause dangerous effects like division by zero as illustrated in Figure 2.3. When a FP computation results in a subnormal number, we say that the computation *underflows*.

At this point and before continuing, we hope the reader is now able to understand the following definition of, say, the rounding function to-nearest.

Definition 2.7 (ζ_n). *The rounding function to-nearest ζ_n is defined as follows:*

$$\zeta_n(x) = [x]_{ulp(x)} \tag{2.1}$$

where $$ulp(x) = \beta^{-p+1} \cdot \beta^{exp(x)} \tag{2.2}$$

and $$exp(x) = \begin{cases} \lfloor log_\beta(|x|) \rfloor \ \textit{if} \ |x| \geq \beta^{e_{min}} \\ e_{min} \ \textit{otherwise} \end{cases} \tag{2.3}$$

and $$\beta \geq 2, p \geq 2, \beta \in \mathbb{N}, p \in \mathbb{N}, e_{min} \in \mathbb{Z}$$

The notation $[a]_b$ denotes the multiple of b nearest to a while the notation $\lfloor a \rfloor$ denotes the greatest integer smaller or equal to a.

When to use floating-point numbers

We start by giving a few well-known recommendations regarding FP computations. They are taken from the CERT[10] and MISRA[11] coding standards.

CERT FLP30-C, NUM09-J. Do not use FP variables as loop counters.
CERT FLP02-C. Do not use FP numbers if accurate computation is required.
MISRA S1244. FP numbers should not be tested for equality.

These recommendations are very wise indeed, particularly for programmers who are not familiar with FP computations[12]. However we have to pay attention to not misinterpret them. Notably we must not be lead into thinking that FP computations cannot be trusted and have to be avoided as much as possible. There are cases for which FP numbers are perfectly suitable:

(Case1) Speed is needed. FP computations are extremely fast. If time constraints take over accuracy constraints then FP numbers might be the ideal solution. This is illustrated in Figure 2.4 in which we compare `Aprational`[13] and `double`.

(Case2) We do not have much memory at our disposal. FP numbers are stored on a fixed amount of memory. This is in contrast to libraries that manipulate arbitrarily precise numbers: the more the precision, the more the amount of memory required.

(Case3) Time, memory and accuracy constraints are equally important. This is the case for critical systems such as control programs in avionics. Expertise is required in order to produce results with guaranteed accuracy.

2.3 The Ieee 754 standard

Until the 1980s, each circuit manufacturer had his own implementation of FP arithmetic. Notably they each devised their own FP types: they chose their own values for the radix β, the precision p and the range $[e_{min}, e_{max}]$ of the exponent as illustrated in Figure 2.5(a). Moreover their implementations often did not guarantee correct rounding. That led to surprising behaviors. For example on some Cray machines, the FP multiplication by 1 sometimes resulted in an overflow. Also on the IBM 370

[10] https://www.securecoding.cert.org/
[11] https://www.misra.org.uk/
[12] "95% of folks out there are completely clueless about FP," James Gosling.
[13] A library for manipulating rational numbers, http://www.apfloat.org/.

```
Aprational t = new Aprational("1");
long start = time();
t = t.add(new Aprational("1/200"));
long duration = time() - start;
```

(a) The variable `t` is exactly computed. However for the few executions we try, the variable `duration` can be as big as 11: we cannot use `Aprational`.

```
double t = 1;
long start = time();
t = t + 0.005;
long duration = time() - start;
```

(b) The variable `t` is not exactly computed. However for the few executions we try, the variable `duration` remains equal to zero: the FP addition takes less than a millisecond.

FIGURE 2.4: Investigating ways to implement a Java clock that starts from 1s and that is to be refreshed every 5 ms. We compare `Aprational` and `double` for the first update of the clock: it must be done as quickly as possible and in strictly less than 5 ms.

machines, the FP square root of -4 was equal to 2 in Fortran. The rather messy situation of FP arithmetic back then is depicted in [Kah81].

Machine	FP type	β	p	$[e_{min}, e_{max}]$
Cray 1	`single`	2	48	$[-8192, 8191]$
	`double`	2	96	$[-8192, 8191]$
IBM 3090	`single`	16	6	$[-64, 63]$
	`double`	16	14	$[-64, 63]$
	`extended`	16	28	$[-64, 63]$

(a) FP types used by the supercomputers Cray 1 (1976) and IBM 3090 (1985)

FP type	β	p	$[e_{min}, e_{max}]$
`binary32`	2	24	$[-126, 127]$
`binary64`	2	53	$[-1022, 1023]$

(b) FP types of the 1985 version of IEEE 754

FIGURE 2.5: A few FP types across history

In 1985 and under the impulsion of Kahan, the IEEE 754 standard was published [Ieea]. It gave numerous directives regarding the implementation of FP arithmetic. The most important ones are given in the following:

(Dir1) *FP types.* The standard defined two basic FP types: the single precision type which is encoded on 32 bits, it is also called binary32; and the double precision type which is encoded on 64 bits, it is also called binary64. The corresponding values of β, p and $[e_{min}, e_{max}]$ are shown in Figure 2.5(b). Support for subnormals was required.

(Dir2) *FP functions.* The standard required the FP arithmetic operations $\oplus$, $\ominus$, $\odot$, $\oslash$ and the FP square root $\sqrt{\ }$ to be correctly rounded.

(Dir3) *Rounding modes.* The standard required support for the four rounding modes to-nearest, to-zero, to-positive-infinity and to-negative-infinity presented in Definition 2.2.

(Dir4a) *Exceptions.* The standard specifies five kinds of exceptions: INVALID OPERATION, DIVISION BY ZERO, OVERFLOW, UNDERFLOW and INEXACT. INVALID OPERATION is signaled when computing for example the square root of a negative number. DIVISION BY ZERO is signaled when attempting to divide by zero. We point out that the standard disinguishes the positive zero $+0$ from the negative zero -0. OVERFLOW (resp. UNDERFLOW) is signaled when a computation results in an overflow (resp. underflow). INEXACT is signaled when the result of a computation cannot be exactly represented.

(Dir4b) *Silent responses and special values.* When the aforementioned exceptions occur, we can define the response that should be returned by the involved computation. For example we can configure the INVALID OPERATION, DIVISION BY ZERO and OVERFLOW exceptions such that they cause the computation to terminate. In particular we can define silent responses that will allow the computation to smoothly continue. The silent response to INVALID OPERATION is to produce a special number called Not-a-Number or `NaN`. The silent response to DIVISION BY ZERO is to produce the special number $+$`Infinity` or $-$`Infinity` depending on the sign of the zero in denominator. The silent response to OVERFLOW is to also produce an infinity. The silent response to UNDERFLOW and INEXACT is to round the result.

Then the standard was revised in 2008 [Ieeb]. IEEE 754-2008 is also called IEEE 754-R and is the active version to date. It extends IEEE 754-1985 as follows:

(Dir1$^+$) *FP types.* The standard now defines decimal FP types which are FP types with $\beta = 10$. Also it defines the quadruple precision type which is encoded on 128 bits or digits.

(Dir2$^+$) *FP functions.* The standard now defines a new operator called Fused Multiply-Add (FMA): $\forall \hat{x}_1, \hat{x}_2, \hat{x}_3 : fma(\hat{x}_1, \hat{x}_2, \hat{x}_3) = \zeta(\hat{x}_1 \cdot \hat{x}_2 + \hat{x}_3)$. Notice that in the general case $fma(\hat{x}_1, \hat{x}_2, \hat{x}_3) \neq \hat{x}_1 \bigodot \hat{x}_2 \bigoplus \hat{x}_3$. Also the standard now recommends, yet does not require, correct rounding for a few functions like $e^x, ln(x)$ and $sin(x)$. The complete list of recommended functions can be found in [Ieeb, Table 9.1].

(Dir3$^+$) *Rounding modes.* The standard now defines the new rounding mode *to-nearest-ties-away-from-zero*. It is similar to the rounding mode *to-nearest-ties-even* with the difference that we choose the FP number that has the biggest absolute value in case of tie.

IEEE 754-2008 is due to expire next 2018: a revision is targeted soon.

2.4 Pitfalls of verifying floating-point computations

Now the reader might think that the verification of FP computations across different platforms can be done in a uniform way since the standardization of IEEE 754. Notably the reader might think that two platforms that claim to be IEEE 754-compliant will always produce the same results for the same FP computations. Unfortunately the behavior of FP computations depends on many things such as the programming language, the compiler and the physical architecture: a total IEEE 754-compliance has to take into account all these stages. Moreover there are subtle details on FP computations that are unknown to many programmers and that lead them into incorrect understanding of their own source code.

Decimal-to-binary conversion. One source of misunderstanding is that programmers usually think and code in decimal whereas machines compute in binary. Indeed though decimal FP types have been standardized in IEEE 754-2008, only the binary ones are currently supported in most systems. Unfortunately the conversion from radix $\beta = 10$ to $\beta = 2$ introduces errors. For example the decimal number $x = 0.2$ cannot be exactly represented in any binary FP type since its binary expansion $0.00110011\cdots$ is infinite: using the IEEE 754 binary32 type, the decimal number that is actually stored is $\hat{x} = 0.199999992549419403076171875$[14]. There are special cases for which the conversion from a decimal number to a binary FP number is exact [Mat68][Mul+10, Theorem 1]. However in the general case, the obtained binary conversion has to be rounded. *We must keep in mind that the numbers we write in our source code may* not *be the numbers used at runtime.*

Binary-to-decimal conversion. Notice that the conversion from an internal binary number to a decimal number is always exact. Indeed a *finite* binary representation can be always converted into a *finite* decimal representation. However many printing functions do not display the exact decimal conversion. Indeed they are often required to print a very few number of digits: if the decimal conversion has too many digits then it is rounded. For example in Java, the sequence of instructions `float x = 0.2f; System.out.print(x);` will print 0.2 instead of the lengthy $\hat{x} = 0.199\cdots75$

[14] That is why the Java equality `0.2f == 0.199999992549419403076171875f` is counter-intuitively evaluated to `true`.

that is actually stored. Thus when debugging FP computations, *we must pay attention as to whether the printed decimal number is indeed the exact conversion of the internal binary number*. Interested readers can refer to [JW04].

Compiler-dependent issues. Recall the Fused Multiply-Add operation that was standardized since IEEE 754-2008: $fma(\hat{x}_1, \hat{x}_2, \hat{x}_3) = \zeta(\hat{x}_1 \cdot \hat{x}_2 + \hat{x}_3)$. Some processors like IBM PowerPC or Intel/FP Itanium implement it. Now consider for example the C instruction `x = a*b + c*d;` in which all the variables are of the type `float`. How will that instruction be compiled if FMA is available? Depending on the compilation options, the compiler may choose not to use FMA and to simply evaluate the instruction as $x = a \odot b \oplus c \odot d$. But it may also choose to use FMA and to evaluate the instruction as either $x = fma(a, b, c \odot d)$ or as $x = fma(c, d, a \odot b)$. These three possible evaluations can result in three different values of x, all sound with respect to C's semantics. Thus *we must keep in mind that the same source code can be soundly compiled into different binary codes, each behaving differently.* Moreover there is the possibility that the source code is *not* soundly compiled. Notably the compiler may unsafely transform FP expressions during optimization processes. To this date and to our knowledge, the only existing compiler that is formally certified to be sound is CompCert [Ler09].

Architecture-dependent issues. Then there is another difficulty: *the same source code is compiled differently for different architectures, on which the obtained binaries can behave differently.* One of the most subtle illustration is that of the double rounding phenomenon. It happens on IA32 architectures such as the Intel x86 series or the Pentium series. IA32 processors feature a FP Unit (FPU) called x87. This unit has 80-bit registers that can be associated to the C type `long double`. Computation on x87 is as follows. First the intermediate operations are computed and rounded to 80 bits. Then the final result is rounded to the destination format, to 64 bits for example. Surprisingly, such double rounding can yield different results from direct rounding to the destination format. This is illustrated in Figure 2.6.

Language-dependent issues. Let us consider the case of the Java programming language. To prevent architecture-dependent issues such as double roundings and to ensure that the FP results will be bit-by-bit accurate across different Java Virtual Machines, Java introduced the `strictfp` modifier which can be used on classes, interfaces and non-abstract methods[15]. This modifier requires every intermediate result to

[15]Schildt, Herbert (2007). Java: A Beginner's Guide (4 ed.). McGraw-Hill Companies

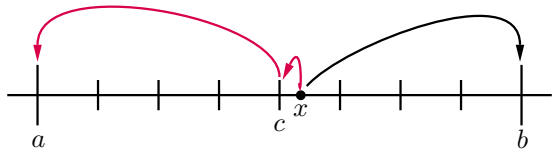


FIGURE 2.6: Consider the IEEE 754 double precision format: $a = 1$ and $b = 1 + 2^{-52}$ are two consecutive `double`. Consider the real number $x = 1 + 2^{-53} + 2^{-64}$. Its correct rounding into a `double` should be b. However if computations are first performed in `long` then x is first rounded to c which is the nearest `long`. Then the rounding of c into a `double` is a since the mantissa of a is even. More details can be found in [BN10, Section 2.2][Mon08, Section 3.1.2].

be truncated to 64 bits. Doing so should force the program to adopt strict IEEE-754 round-to-nearest semantics. Then there is another way to control FP computations at the source code level in Java: through the `StrictMath` class. Recall that IEEE 754-2008 recommends correct rounding for a few elementary functions like *sin* or *cos*. As it is only recommended but not required, many FPU give results that can be one or two FP numbers off the correct one. Thus when the `Java.Math` library call the FPU's implementations of elementary functions, it can get incorrectly rounded results. In contrast to that, the `StrictMath` class does not use FPU's implementations of elementary functions. Instead, it has its own implementations which guarantee correct rounding. The point is *we must pay attention to the actions made at the source code level that can affect FP computations*. Moreover we must be aware that the constraints we impose at source code level may just be ignored during compilation. For example compilers like GCJ ignore the `strictfp` modifier. Also the use of Just-In-Time compilation may add undesirable effects.

Other issues. There are still many subtle causes that affect FP computations. A lot of them can be found in [Mon08]. These causes are so subtle that [Mul+10, Section 7.2.6] refers to them as "horror stories". For example, even the simple act of printing data can be a trap. Indeed, common sense commands us to think that printing out a variable cannot change its value. But the fact is it can [Mul+10, C listing 7.3].

Hypotheses on the behaviors of the FP computations in this thesis

We hope the reader now understands that there are many things to take into account when verifying FP computations. A sound verification has to take into account many

things such as the programming language, the compiler, the architecture and so on. For example [BN11] develops techniques for verifying FP computations whether the compiler optimizes while [BN10] takes into account multiple architectures at once.

Hypothesis 2.1. *In this thesis, we assume that our FP system respects the following directives:*

(MyDir1) FP types. *Our FP types can be freely parameterized. That is β, p and $[e_{min}, e_{max}]$ can take any values. This allows our results to be applied not only to programs that use the* IEEE 754 *FP types, binary or decimal, but also to programs that use personalized ones like with the MPFR library [Fou+07]. Subnormals are supported.*

(MyDir2) FP functions. *We will only use the four FP arithmetic operations $\oplus, \ominus, \odot$ and $\oslash$. We assume they all guarantee correct rounding.*

(MyDir3) Rounding modes. *When not explicited, the considered rounding mode is to-nearest.*

(MyDir4a)* Exceptions and termination.** ***We consider that the* INVALID OPERATION, DIVISION BY ZERO *and* OVERFLOW *exceptions cause the computation to terminate [on an error].

(MyDir4b) Special values. *Assuming (MyDir4a), we do not need the special values* `NaN` *and* `±Infinity`. *We consider that we round the result when the* UNDERFLOW *and* INEXACT *exceptions occur. Also, we consider that there is only a single, unsigned zero.*

(MyDir5) Sound executions. *We assume that the semantics of the source code and that of the executable code are equivalent: when executed, the program behaves exactly as described by the source code. Notably we assume that the compiler does not perform unsound optimizations and that the architecture is not affected by double rounding issues.*[16]

[16] In the same way, we also assume that no exotic events, such as over-exposure to cosmic rays or a moth trapped in the machine, affect the computations. See https://science.slashdot.org/story/17/02/19/2330251/serious-computer-glitches-can-be-caused-by-cosmic-rays and http://modernnotion.com/the-first-computer-bug-was-an-actual-bug/.

2.5 A few properties of floating-point computations

Due to the rounding errors that affect the result of each operation, FP arithmetic is different from real arithmetic. We give a few of its properties in this section.

Property 2.1. *FP addition* $\oplus$ *and FP multiplication* $\odot$ *are not always associative.*

Proof. *Suppose we use* `myfloat` *with the rounding mode set to to-nearest. A counter-example to the associativity of* $\oplus$ *is that* $(100 \oplus 3) \oplus 3 = 100$ *while* $100 \oplus (3 \oplus 3) = 110$. *Similarly, a counter-example to the associativity of* $\odot$ *is that* $(100 \odot 0.3) \odot 0.1 = 3$ *while* $100 \odot (0.3 \odot 0.1) = 0$.

Property 2.2. *FP multiplication* $\odot$ *is not distributive over FP addition* $\oplus$.

Proof. *A counter-example using* `myfloat` *with the rounding mode set to to-nearest:* $3 \odot (100 \oplus 3) \neq 3 \odot 100 \oplus 3 \odot 3$.

Definition 2.8 (Absorption). *Let* $\hat{x}_1$ *and* $\hat{x}_2$ *be two FP numbers. If we add* $\hat{x}_1$ *to* $\hat{x}_2$ *and if* $\hat{x}_1$ *is very big compared to* $\hat{x}_2$ *then it may occur that the result is rounded to* $\hat{x}_1$ *itself:* $\hat{x}_1 \oplus \hat{x}_2 = \hat{x}_1$. *That phenomenon is called* absorption.

Example 2.6. *Using* `myfloat` *with the rounding mode set to to-nearest:* $100 \oplus 0.1 = \zeta_n(100.1) = 100$.

Definition 2.9 (Catastrophic cancellation). *Let* $\hat{x}_1$ *and* $\hat{x}_2$ *be two FP numbers. If we subtract* $\hat{x}_2$ *to* $\hat{x}_1$ *and if* $\hat{x}_1$ *is close to* $\hat{x}_2$ *then the most significant information units*[17] *of their respective significand match and cancel each other. Hence if* $\hat{x}_1$ *and* $\hat{x}_2$ *are affected by rounding errors then many accurate information units may disappear: the final result may importantly differ from that we would have obtained using real arithmetic. That phenomenon is called* catastrophic cancellation.

Example 2.7. *Consider the expression* $\Delta = b^2 - 4ac$ *taken from the quadratic formula. If* $a = 1.3, b = 3.4$ *and* $c = 2.2$ *then* $\Delta = 0.12$. *Now suppose we use* `myfloat` *with the rounding mode set to to-nearest:* $\hat{\Delta} = \hat{x}_1 \ominus \hat{x}_2$ *with* $\hat{x}_1 = 3.4 \odot 3.4 = 12$ *and* $\hat{x}_2 = 4 \odot 1.3 \odot 2.2 = (4 \odot 1.3) \odot 2.2 = 5.2 \odot 2.2 = 11$. *Thus* $\hat{\Delta} = 12 \ominus 11 = 1$ *which is almost ten times bigger than* Δ*: catastrophic*[18] *cancellation occurred.*

[17] Bits, trits, digits or whatever depending on the radix β.

[18] Cancellation is *not* always catastrophic: it is not always a bad thing. For example a case of beneficial cancellation can be found in http://introcs.cs.princeton.edu/java/91float/, Creative Exercises number 18.

See Figure 2.7 for illustrations of these properties with Java.

```
float x = 8388608,
      y = 0.5f,
      z = 3;
boolean
  a = (x + y) == x, // true
  b = (x + y + z) ==
      (x + (y + z)), // false
  c = (z * (x + y)) ==
      (z*x + z*y)); // false
```

(a) Absorption phenomenon, non-associativity of $\oplus$ and non-distributivity of $\odot$ over $\oplus$

```
float a = 100,
      b = 2048.0007f,
      c = 10485.7668f,
      d = b*b - 4*a*c;
        // gives 0.5 while
        // reals approximately
        // give 0.147
```

(b) Catastrophic cancellation phenomenon

FIGURE 2.7: Two Java programs that illustrate the peculiar properties of FP arithmetic

Now the reader might think that FP computations are *always* affected by rounding errors. Actually there are special cases for which they are not. That is there are special cases for which the result of a FP computation is equal to the result that would have been obtained if operations were performed on real numbers. A trivial example is the FP multiplication and FP division by (-1): in any FP system, $\hat{x} \odot (-1) = \hat{x} \oslash (-1) = -\hat{x}$ if $\hat{x} \neq$ `NaN`[19]. There are more sophisticated examples as shown in the following.

Property 2.3. *In a radix-β FP system with an exponent range of $[e_{min}, e_{max}]$, let $\hat{x}_1$ be a FP number that is an integer power of β: $\hat{x}_1 = (-1)^{s_1}\beta^{e_1}$. For any FP number $\hat{x}_2 = (-1)^{s_2}\hat{m}_2\beta^{e_2}$, we have $\hat{x}_2 \odot \hat{x}_1 = \hat{x}_2 \cdot \hat{x}_1$ (resp. $\hat{x}_2 \oslash \hat{x}_1 = \hat{x}_2 / \hat{x}_1$) if and only if $e_2 + e_1 \in [e_{min}, e_{max}]$ (resp. $e_2 - e_1 \in [e_{min}, e_{max}]$).*

Property 2.4 (Sterbenz lemma [Ste74][Mul+10, Lemma 2]). *In a radix-β FP system with subnormal numbers available, if two FP numbers $\hat{x}_1$ and $\hat{x}_2$ are such that $\frac{\hat{x}_1}{2} \leq \hat{x}_2 \leq \hat{x}_1$ then $\hat{x}_1 \ominus \hat{x}_2 = \hat{x}_1 - \hat{x}_2$. The result is valid for the four rounding modes to-nearest, to-zero, to-positive-infinity and to-negative-infinity.*

Property 2.5 (Hauser theorem [Hau96][Mul+10, Theorem 3]). *In a radix-β FP system with any rounding mode, for any FP numbers $\hat{x}_1$ and $\hat{x}_2$ such that $\hat{x}_1 \oplus \hat{x}_2$ is a subnormal number, we have $\hat{x}_1 \oplus \hat{x}_2 = \hat{x}_1 + \hat{x}_2$.*

[19] The case of `NaN` has to be excluded since `NaN == NaN` is FALSE

Property 2.6 ([Bol15]). *In a radix-2, radix-3 and radix-4 FP system with the rounding mode set to to-nearest-ties-even or to-nearest-ties-away-from-zero, the FP square root of the FP square of a FP number $\hat{x}$ is exactly $|\hat{x}|$. The result is* not *valid for any of the three rounding modes to-zero, to-positive-infinity and to-negative-infinity.*

Most of the time, results of FP computations are not exact. In such cases, we can provide bounds for the rounding errors. Notably we can provide bounds for the absolute and relative errors when rounding a real number.

Definition 2.10 (Machine epsilon ϵ). *In a FP system with the rounding mode set to to-nearest, we call machine epsilon ϵ the quantity $\epsilon = \frac{\beta^{-p+1}}{2}$. Machine epsilon is also called unit roundoff.*

Definition 2.11 (Absolute error). *The absolute error $\mathscr{A}(x)$ of the approximation of $x \in \mathbb{R}$ by $\hat{x} = \zeta(x)$ is defined as $\mathscr{A}(x) = |x - \hat{x}|$.*

Definition 2.12 (Relative error). *The relative error $\mathscr{R}(x)$ of the approximation of $x \in \mathbb{R}$ by $\hat{x} = \zeta(x)$ is defined as $\mathscr{R}(x) = \frac{|x-\hat{x}|}{|x|}$. The case when $x = 0$ is particular: $\mathscr{R}(0) = 0$ since 0 is exactly representable.*

Property 2.7 (Bound $\mathscr{A}_s$ for the absolute error of subnormals [Mul+10, Section 2.2.3]). *In a FP system with the rounding mode set to to-nearest, if $x \in \mathbb{R}$ is rounded into a subnormal number then $\mathscr{A}(x) \leq \mathscr{A}_s, \mathscr{A}_s = \epsilon\beta^{e_{min}}$.*

Example 2.8 (Continuing Example 2.5). *Any real number $x \in \mathbb{R}$ such that $|x| \leq n_{min}, n_{min} = 1$, is approximated by a FP number $\hat{x}$ which is at a distance of at most $\mathscr{A}_s = \frac{0.1}{2} = 0.05$ from x.*

Property 2.8 (Bound $\mathscr{R}_n$ for the relative error of normals [Mul+10, Section 2.2.3]). *In a FP system with the rounding mode set to to-nearest, if $x \in \mathbb{R}$ is rounded into a normal number then $\mathscr{R}(x) \leq \mathscr{R}_n$ such that $\mathscr{R}_n = \epsilon$.*

Example 2.9 (Continuing Example 2.5). *Any real number $x \in \mathbb{R}$ such that $|x| > n_{min}, n_{min} = 1$, is approximated by a FP number $\hat{x}$ which is at a distance of at most $100\mathscr{R}_n$ percent of $|x|$ from x with $\mathscr{R}_n = \epsilon = \frac{0.1}{2} = 0.05$.*

Property 2.9 (Bound $\mathscr{A}_{sn}$ for the absolute error of subnormals and normals). *In a FP system with the rounding mode set to to-nearest, if $x \in \mathbb{R}$ is rounded into a subnormal or a normal then $\mathscr{A}(x) \leq \mathscr{A}_{sn}$ such that $\mathscr{A}_{sn} = \epsilon\beta^{e_{max}}$.*

More details on FP numbers can be found in [Gol91] and [Mul+10]. At this point, we hope the reader understands the hardness of analyzing FP computations. Due to the rounding errors that may accumulate and propagate, even simple computations like that of x^n through iterated FP multiplications require efforts to analyze [GLM15]. It is in that sense that in 2014, Kahan called out for "desperately needed remedies for the undebuggability of large floating-point computations in science and engineering[20]": analysis of FP computations is a timely topic.

2.6 Conclusion

In this chapter, we have seen the specificities of Floating-Point (FP) arithmetic. FP arithmetic is different from the traditional real or rational arithmetic. This is due to the existence of rounding errors. They affect the computations in a way such that the basic arithmetic properties we are used to are not preserved anymore: FP addition is not associative, FP multiplication is not distributive over FP addition and so on.

We presented in Chapter 1 a few termination proof techniques. They have been presented with programs that use mathematical integers, rational numbers or real numbers. Due to the peculiarity of FP computations, manipulations have to be performed before we can apply these techniques to programs that work with FP numbers. Starting next chapter, we enter the core of this thesis: we present ways to analyze termination of FP computations.

[20] https://people.eecs.berkeley.edu/ wkahan/Boulder.pdf

Part II

Termination Analysis of Floating-Point Computations

Chapter 3

Piecewise Affine Approximations

Some of the results described in this chapter have been the subject of two publications in international conferences [MMP16b][MMP17]. They are described here in an harmonized and enhanced manner.

Abstract. In this chapter, we present a way to analyze termination of Floating-Point (FP) computations. It takes advantage of the already existing work on termination analysis of rational computations. We translate FP expressions into rational ones by means of sound approximations. This is done through the use of rational piecewise affine functions. The approximations we obtain can be ordered in increasing tightness, with respect to a specific sense that we define. Each of them is optimal with respect to a specific sense that we also define. Our approach consists in gradually increasing the tightness of the rational approximation until termination can be proved or disproved.

Résumé. Dans ce chapitre, nous présentons une technique qui permet d'analyser la terminaison de calculs flottants. Elle tire profit du travail déjà existant sur l'analyse de terminaison de calculs rationnels. Nous traduisons les expressions flottantes en expressions rationnelles grâce à des approximations sûres. Cela est fait par l'utilisation de fonctions affines par morceaux. Les approximations que nous obtenons peuvent être classées en précision croissante, selon un sens particulier que nous précisons. Chacune d'elles est optimale, selon un sens particulier que nous précisons également. Notre approche consiste à augmenter progressivement la précision de l'approximation rationnelle jusqu'à ce que la terminaison puisse être prouvée ou réfutée.

Prerequisites. This chapter relies on the following notions from previous chapters:

Chapter 1: Termination Analysis

Proposition 1.1: Over-approximation and termination

3.1 Introduction

To the best of our knowledge, there is only limited work on termination analysis of FP computations. One of the first work that addresses the problem is [SS05]. It analyses termination of logic programs that manipulate FP numbers.

Then techniques were developed to handle the more general case of programs that manipulate bit-vectors. These techniques are based on Model Checking [BK08]. For example [Coo+13] presents a few algorithms to generate ranking functions for relations over machine integers. Notably it presents a template-matching approach which consists in using SAT-solvers or QBF-solvers to check the existence of ranking functions that have predefined forms. Somewhat similarly, [DKL15] expresses termination of programs that manipulate bit-vectors in a decidable fragment of second-order logic. These methods are complete for the problem they consider but they are highly costly.

In contrast to termination analysis of FP computations, that of rational ones has already produced many techniques. To cite a few, there is the synthesis of Affine Ranking Functions [PR04a], that of Eventual Linear Ranking Functions [BM13], the multi-dimensional rankings approach [Ali+10] and the Size-Change principle [LJB01]. The reader can refer to Chapter 1 for further references. Unfortunately application of

these techniques to termination analysis of FP computations is not straightforward due to rounding errors.

In this chapter, we develop a new technique that helps addressing that lack of work. We approximate the rounding function in order to get rid of the difficulties introduced by the rounding errors. This is done through the use of rational piecewise affine functions. Thus we transpose termination analysis of FP computations into termination analysis of rational ones. The novelty of the approach lies in the nature of the approximations: we use k-Piecewise Affine Approximation functions (k-PAA), $k \in \mathbb{N}, k > 0$. The approximations we get are organized in a way such that if termination cannot be decided using some k-PAA then we can increase our chances to do so by increasing k. Also we show that each of our k-PAA is optimal with respect to some quality measure and some conditions that we define.

This chapter is organized as follows. Section 3.2 presents the notion of k-PAA. In particular it presents our way of comparing k-PAAs. Section 3.3 pursues with its application to termination analysis. Section 3.4 presents the afore-mentioned optimality result. Section 3.5 concludes.

3.2 Pairs of k-piecewise affine approximation functions

Subsection 3.2.1 starts by giving a few well-known approximations of FP functions and exhibits the notion of k-PAA. Then Subsection 3.2.2 provides ways to measure and compare their quality. Last Subsection 3.2.3 develops a specific way to increase the quality of a given k-PAA.

3.2.1 A few well-known approximations

Definition 3.1 (Affine function). *A (rational) affine function f of one variable x is a function such that:*

$$\begin{array}{rcccl} f & : & \mathbb{Q} & \to & \mathbb{Q} \\ & & x & \mapsto & f(x) = ax + b, a \in \mathbb{Q}, b \in \mathbb{Q} \end{array}$$

Property 3.1 (The (μ_3, ν_3) pair of approximation functions). *Consider a FP system with the rounding mode set to to-nearest. Consider the rational function $\star$ and its FP equivalent $\circledast$ that takes n FP numbers $\hat{x}_1 \dots \hat{x}_n$ as arguments. If $\circledast$*

correctly rounds $\star$ *then we have a pair of affine functions* (μ_3, ν_3) *such that* $\mu_3(x) \leq \circledast(\hat{x}_1 \dots \hat{x}_n) \leq \nu_3(x)$ *where* $x = \star(\hat{x}_1 \dots \hat{x}_n)$ *and:*

$$\forall x \in \mathbb{Q} : \mu_3(x) = \begin{cases} x \cdot (1 - \mathscr{R}_n) & \text{if } x \in]n_{min}, n_{max}] \\ x - \mathscr{A}_s & \text{if } x \in [-n_{min}, n_{min}] \\ x \cdot (1 + \mathscr{R}_n) & \text{if } x \in [-n_{max}, -n_{min}[\end{cases}$$

$$\nu_3(x) = \begin{cases} x \cdot (1 + \mathscr{R}_n) & \text{if } x \in]n_{min}, n_{max}] \\ x + \mathscr{A}_s & \text{if } x \in [-n_{min}, n_{min}] \\ x \cdot (1 - \mathscr{R}_n) & \text{if } x \in [-n_{max}, -n_{min}[\end{cases}$$

where $\mathscr{A}_s = \epsilon\beta^{e_{min}}$ *and* $\mathscr{R}_n = \epsilon$.

Proof. *Derived from Definition 2.3, Property 2.7 and Property 2.8.*

Property 3.2 (Following Property 3.1, the (μ_2, ν_2) pair of approximation functions). *We also have* $\mu_2(x) \leq \circledast(\hat{x}_1 \dots \hat{x}_n) \leq \nu_2(x)$ *such that:*

$$\mu_2(x) = \begin{cases} x \cdot (1 - \mathscr{R}_n) - \mathscr{A}_s & \text{if } x \in [0, n_{max}] \\ x \cdot (1 + \mathscr{R}_n) - \mathscr{A}_s & \text{if } x \in [-n_{max}, 0[\end{cases}$$

$$\nu_2(x) = \begin{cases} x \cdot (1 + \mathscr{R}_n) + \mathscr{A}_s & \text{if } x \in [0, n_{max}] \\ x \cdot (1 - \mathscr{R}_n) + \mathscr{A}_s & \text{if } x \in [-n_{max}, 0[\end{cases}$$

Proof. *From Property 3.2, merge the positive normals with the positive subnormals and the negative normals with the negative subnormals.*

Property 3.3 (Following Property 3.2, the (μ_1, ν_1) pair of approximation functions). *We also have* $\mu_1(x) \leq \circledast(\hat{x}_1 \dots \hat{x}_n) \leq \nu_1(x)$ *such that:*

$$\begin{aligned} \mu_1(x) &= x - \mathscr{A}_{sn} \quad \text{if } x \in [-n_{max}, n_{max}] \\ \nu_1(x) &= x + \mathscr{A}_{sn} \quad \text{if } x \in [-n_{max}, n_{max}] \end{aligned}$$

where $\mathscr{A}_{sn} = \epsilon\beta^{e_{max}}$.

Proof. *Derived from Definition 2.3 and Property 2.9.*

These approximations are for example used in [BMR12][BN11][Min07]. We approximated FP functions using affine functions defined in three, two and one piece(s). We generalize the idea to k pieces, $k \geq 1$.

Definition 3.2 (Pair of k-Piecewise Affine Approximation functions (k-PAA)). *Consider ζ_n on some interval I that is partitioned into k intervals $I_1 \dots I_k$. The pair (μ, ν) is a pair of k-PAA of ζ_n on I if:*

$$\begin{aligned} \forall x \in I: \quad & \mu(x) \leq \zeta_n(x) \leq \nu(x) \\ & \mu(x) = \left\{\mu_i(x) \text{ if } x \in I_i\right\}_{1 \leq i \leq k} \\ & \nu(x) = \left\{\nu_i(x) \text{ if } x \in I_i\right\}_{1 \leq i \leq k} \end{aligned}$$

where μ_i and ν_i are affine functions.

3.2.2 Comparing approximations

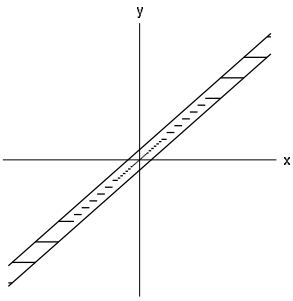
Consider the pairs of k-PAA (μ_1, ν_1), (μ_2, ν_2) and (μ_3, ν_3) presented in Subsection 3.2.1. Question arises: which of them approximates ζ_n the best? To answer that question, let us graphically visualize the question. Take for example the FP type `myfloat` for which ζ_n is exactly defined as follows:

$$\begin{aligned} \zeta_n \quad : \quad & \mathbb{R} \quad \rightarrow \quad \mathbb{F}_{\texttt{myfloat}} \\ x \quad \mapsto \quad & y = \hat{x} = \zeta_n(x) = \begin{cases} 990 & 985 < x < 995 \\ \dots & \\ 100 & 99.5 \leq x \leq 105 \\ 99 & 98.5 < x < 99.5 \\ 98 & 97.5 \leq x \leq 98.5 \\ \dots & \\ -990 & -995 < x < -985 \\ \infty & \text{otherwise (this case causes termination, see Hypothesis 2.1)} \end{cases} \end{aligned} \tag{3.1}$$

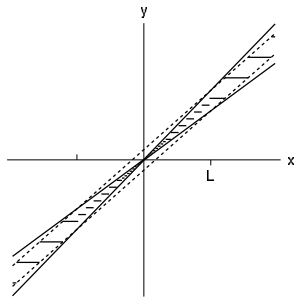
The stairs-like function in Figure 3.1 caricatures ζ_n. The pairs (μ_1, ν_1), (μ_2, ν_2) and (μ_3, ν_3) are respectively illustrated in Figure 3.1(a), 3.1(b) and 3.1(c). We propose the following metric to measure their quality.

Definition 3.3 (Quality measure: tightness). *Consider the pair of k-PAA (μ, ν) of ζ_n on some interval $I = [x_{min}, x_{max}]$. We measure the quality of (μ, ν) by the*

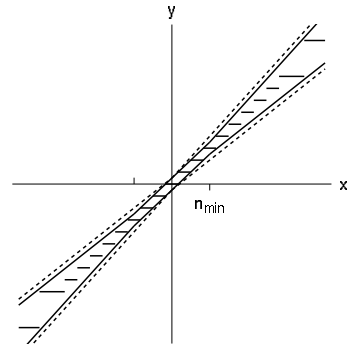
surface of the area enclosed between μ *and* ν *that is by* $S(\mu,\nu) = \int_I \big(\mu(x) - \nu(x)\big)dx = S_\mu + S_\nu$ *where* $S_\mu = \int_I \big(\mu(x) - \zeta_n(x)\big)dx$ *and* $S_\nu = \int_I \big(\zeta_n(x) - \nu(x)\big)dx$. *If there is a pair* (μ',ν') *such that* $S(\mu',\nu') < S(\mu,\nu)$ *then we say that* (μ',ν') *is tighter than* (μ,ν), *in respect to which we say that* (μ',ν') *is better than* (μ,ν).



(a) The pair (μ_1,ν_1) from Property 3.3



(b) The pair (μ_2,ν_2) from Property 3.2 in plain lines. The pair (μ_1,ν_1) from Property 3.3 in dashed line. See Appendix B for details on L. We point out that μ_2 and ν_2 do *not* cross at the origin. They only seem to do so because of the scale: see Figure 3.1(c) for a zoom in on the origin.



(c) The pair (μ_3,ν_3) from Property 3.1 in plain lines. The pair (μ_2,ν_2) from Property 3.2 in dashed line.

FIGURE 3.1: A few PAA of ζ_n. The stairs-like function represents ζ_n.

Theorem 3.1 (Following Property 3.1, 3.2 and 3.3). *We have* $S(\mu_3,\nu_3) < S(\mu_2,\nu_2)$ *that is* (μ_3,ν_3) *is a tighter approximation of* ζ_n *than* (μ_2,ν_2). *Then* $S(\mu_2,\nu_2) < S(\mu_1,\nu_1)$ *that is* (μ_2,ν_2) *is tighter than* (μ_1,ν_1) if and only if $e_{max} > e_{min} + p$.

Proof. *See Appendix B.*

For the IEEE-754 FP types, the range of the exponent is very big compared to the precision: $e_{max} - e_{min} > p$. That is for the IEEE-754 FP types, (μ_2,ν_2) is tighter than (μ_1,ν_1).

Definition 3.4 (Nested pairs). *We say that* (μ',ν') *is nested in* (μ,ν) *if* $\forall x : \mu(x) \leq \mu'(x) \wedge \nu'(x) \leq \nu(x)$ *and* $\exists x : \mu(x) < \mu'(x) \vee \nu'(x) < \nu(x)$.

Proposition 3.1. *If* (μ',ν') *is nested in* (μ,ν) *then* (μ',ν') *is tighter than* (μ,ν).

Proof. *Derived from Definition 3.3 and 3.4.*

Theorem 3.2 (Following Property 3.1, 3.2 and 3.3). *The pair (μ_3, ν_3) is nested in (μ_2, ν_2). The pair (μ_2, ν_2) is* not *nested in (μ_1, ν_1).*

Proof. *See Appendix B.*

3.2.3 Gradual increase of tightness

Now we propose pairs of k-PAA of which the tightness is controlled when approximating ζ_n on some interval I. We increase the tightness by increasing the number of pieces k. Conversely we decrease the tightness by decreasing k. Basically we attempt to start from (μ_3, ν_3) restricted on I. Then we isolate the β-ades in decreasing order of their ULPs.

Definition 3.5 (β-ade isolation, the $\blacktriangle^n$ operation). *Consider (μ, ν) from Definition 3.2. Let B be a β-ade of ULP u. We say that (μ, ν) isolates the FP numbers in B if $\exists I_i \forall x \in I_i : I_i = I \cap B$[1] $\wedge\, \mu_i(x) = x - \frac{u}{2} \wedge \nu_i(x) = x + \frac{u}{2}$. We also simply say that (μ, ν) isolates B.*

We denote $\blacktriangle^n$ the operation that makes (μ, ν) isolate the n β-ades in I that have the greatest ULPs among those that have not been isolated yet. If two β-ades have the same greatest ULPs, one in the positive numbers and one in the negatives, then $\blacktriangle^n$ first isolates the positive one.

Algorithm 3.1 (PAA).

INPUT *A FP type $\mathbb{F}$*
An interval $I = [\hat{x}_{min}, \hat{x}_{max}]$ where $\hat{x}_{min}, \hat{x}_{max} \in \mathbb{F}$
An integer $1 \leq k \leq m$ where m is the number of β-ades B such that $B \cap I \neq \varnothing$

OUTPUT *A pair of k-PAA (μ, ν) of ζ_n on I*

NOTATION *$(\mu, \nu)_I$ denotes (μ, ν) restricted on I*

BEGIN

1: IF $k \geq 3$ THEN RETURN $\blacktriangle^n \left[(\mu_3, \nu_3)_I \right]$
WHERE *n is such that the obtained result is defined in k pieces*

2: IF $k = 2$ THEN

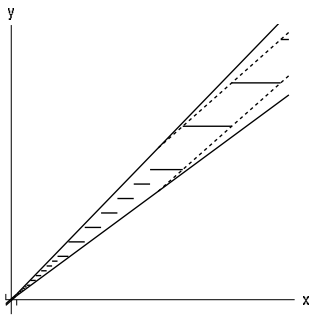
[1] We cannot simply write $I_i = B$. Indeed by doing so we would fail to isolate the FP numbers that belong to β-ades B such that $B \cap I \neq \varnothing$ but $B \not\subset I$, that is to β-ades B such that $B \cap I \neq B$.

IF $(\mu_3, \nu_3)_I$ *is defined in* 2 *pieces* THEN RETURN $(\mu_3, \nu_3)_I$
ELSE IF $(\mu_3, \nu_3)_I$ *is defined in* 1 *piece* THEN RETURN $\blacktriangle^1\Big[(\mu_3, \nu_3)_I\Big]$
ELSE IF $(\mu_3, \nu_3)_I$ *is defined in* 3 *pieces* THEN RETURN $(\mu_2, \nu_2)_I$

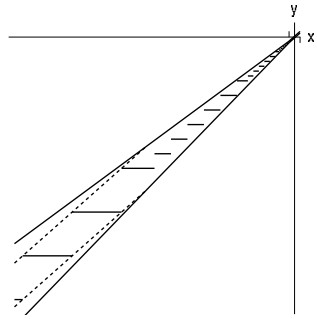
3: IF $k = 1$ THEN RETURN $\big(\text{OPT-}\mu(\mathbb{F}, I), \text{OPT-}\nu(\mathbb{F}, I)\big)$
WHERE OPT-μ *(resp.* OPT-ν*) returns*
the affine function that best lower (resp. upper) approximates ζ_n *on* I*,*
details will be given in Section 3.4.

END

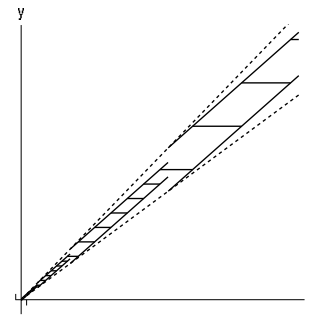
Example 3.1. *See Figure 3.2 and 3.3.*



(a) (μ_3, ν_3) in plain lines: it is symmetrical to the origin. (μ_4, ν_4) takes (μ_3, ν_3) and isolates the β-ade of greatest ULP. There are two candidates: we take the one with positive FP numbers. Then we approximate these numbers with the dashed lines.



(b) Following Figure 3.2(a). (μ_5, ν_5) takes (μ_3, ν_3) and isolates the two β-ades of greatest ULP. We take the two candidates. Then we approximate the FP numbers in them with the dashed lines.



(c) Some (μ_k, ν_k) such that every β-ades are isolated as illustrated by the plain lines. The dashed lines represents (μ_3, ν_3). Symmetry to the origin for the negatives. Notice how the plain lines are nested in the dashed lines. The $\blacktriangle^n$ operation is idempotent past this point.

FIGURE 3.2: Pairs of k-PAA $(\mu_k, \nu_k) = \text{PAA}\big(\mathbb{F}, [-n_{max}, n_{max}], k\big)$

The reason why we attempt to start from $(\mu_3, \nu_3)_I$ is motivated as follows.

Definition 3.6 ((μ_3, ν_3)-derived pair). *Let (μ, ν) be a k-PAA defined on some interval I. We say that (μ, ν) is a (μ_3, ν_3)-derived pair if it was obtained by taking (μ_3, ν_3) restricted on I and making it successively isolate the β-ades: $\exists n : (\mu, \nu) = \blacktriangle^n\Big[(\mu_3, \nu_3)_I\Big]$.*

	$\mu(x)$	$\nu(x)$	$x \in$
$k=1$	$x-5$	$x+5$	$[-990, 990]$
$k=2$	$\frac{20}{21}x - 0.05$	$\frac{22}{21}x + 0.05$	$[0, 990]$
	$\frac{22}{21}x - 0.05$	$\frac{20}{21}x + 0.05$	$[-990, 0[$
$k=3$	$\frac{20}{21}x$	$\frac{22}{21}x$	$]10, 990]$
	$x - 0.05$	$x + 0.05$	$[-10, 10]$
	$\frac{22}{21}x$	$\frac{20}{21}x$	$[-990, -10[$
$k=4$	$x-5$	$x+5$	$[100, 990]$
	$\frac{20}{21}x$	$\frac{22}{21}x$	$]10, 100]$
	$x - 0.05$	$x + 0.05$	$[-10, 10]$
	$\frac{22}{21}x$	$\frac{20}{21}x$	$[-990, -10[$

FIGURE 3.3: A few pairs of k-PAA returned by PAA for $I = [-n_{max}, n_{max}]$ when using `myfloat`

Lemma 3.1. *Let (μ, ν) $\big($resp. $(\mu', \nu')\big)$ be a k-PAA (resp. k'-PAA, $k' > k$) returned by Algorithm 3.1. If (μ, ν) is (μ_3, ν_3)-derived then is (μ', ν') is nested in (μ, ν).*

Proof. *See Appendix B.*

The interest of constructing a k'-PAA that is nested in another k-PAA is motivated in the following section for the case of termination analysis.

3.3 Application to termination analysis

First let us see how to concretely use pairs of k-PAA to abstract a program.

Definition 3.7 (Single Operation Form (SOF)). *We say that a program $\mathcal{P}$ is in Single Operation Form if each arithmetic expression in it has one operation at most. We also say that $\mathcal{P}$ is a SOF program. Any program $\mathcal{Q}$ can be transformed into a SOF program $\mathcal{Q}_{SOF}$ through the use of intermediate variables. We denote* SOF *the corresponding transformation function: $\mathcal{Q}_{SOF} = \text{SOF}(\mathcal{Q})$.*

Example 3.2 (The $\mathcal{P}_g$ program). *See Figure 3.4.*

A possible way of putting a program in SOF form is to put it in SSA form [Cyt+91] first, then to split the assignments that involve multiple operations into multiple assignments that only involve a single operation each.

Definition 3.8 ((μ, ν)-abstraction). *Let $\mathcal{P}$ be a SOF program that manipulates FP numbers. Let (μ, ν) be a pair of approximation functions of the rounding function ζ_n. We call (μ, ν)-abstraction of $\mathcal{P}$ the program $\mathcal{P}^{\#}$ in which the result of each FP*

```
myfloat x = randFP();
            // a random FP number
            // of myfloat type
        y = randFP();
while(x*y >= 0 & y > -60) {
  x = x - 5.5;
  y = y + 3.5 + 3.5;
  if(x < -100) x = 100;
}
```

```
myfloat x = randFP();
        y = randFP(),
        c0 = x*y,
        t0 = randFP();
while(c0 >= 0 & y > -60) {
  x = x - 5.5;
  t0 = y + 3.5;
  y = t0 + 3.5;
  if(x < -100) x = 100;
  c0 = x*y;
}
```

FIGURE 3.4: The program $\mathcal{P}_g$ (left) and SOF($\mathcal{P}_g$) (right)

computation $t = \circledast(\hat{x}_1 \dots \hat{x}_n)$ *is replaced by the rational constraints* $\mu(x) \leq t \leq \nu(x) \wedge x = \star(\hat{x}_1 \dots \hat{x}_n)$. *We suppose that the FP operation* $\circledast$ *correctly rounds the rational one* $\star$. *We denote* $\alpha_{(\mu,\nu)}$ *the corresponding abstraction function:* $\mathcal{P}^{\#} = \alpha_{(\mu,\nu)}(\mathcal{P})$.

Example 3.3 (Following Example 3.2). *The* (μ_1, ν_1)*-abstraction of* SOF($\mathcal{P}_g$) *is illustrated in Figure 3.5.*

```
rational x = randRatBetween(-990,990),
             // a random rational number r
             // -990 <= r <= 990
  y = randRatBetween(-990,990),
  c0 = randRatBetween(x*y - 5, x*y + 5);
  t0 = randRat();
while(c0 >= 0 & y > -60) {
  x = randRatBetween(x - 10.5, x - 0.5);
  t0 = randRatBetween(y - 1.5, y + 8.5);
  y = randRatBetween(t0 - 1.5, t0 + 8.5);
  if(x < -100) x = 100;
  c0 = randRatBetween(x*y - 5, x*y + 5);
}
```

FIGURE 3.5: (μ_1, ν_1)-abstraction of SOF($\mathcal{P}_g$). We point out that -60 and 100 are exactly represented with `myfloat`; otherwise they have to be rounded. We remind that overlows cause termination.

Now we propose a termination proof technique that gradually increases the tightness of the abstraction it uses. The tighter the abstraction, the bigger its set of invariants.

Definition 3.9 (Invariant). *An invariant of a program $\mathcal{P}$ is a property that holds for any possible execution of $\mathcal{P}$.*

Theorem 3.3. *Let $\mathcal{P}$ be a program. Let (μ, ν) be a k-PAA and (μ', ν') a k'-PAA such that (μ', ν') is nested in (μ, ν). The set of invariants of $\mathcal{P}^{\#} = \alpha_{(\mu,\nu)}(\mathcal{P})$ is a subset of that of $\mathcal{P}^{\#'} = \alpha_{(\mu',\nu')}(\mathcal{P})$.*

Proof. *Any execution of $\mathcal{P}^{\#'}$ is also one of $\mathcal{P}^{\#}$.*

Corollary 3.1. *Let $\mathcal{P}$ be a program. Let (μ, ν) $\big($resp. $(\mu', \nu')\big)$ be a k-PAA (resp. k'-PAA, $k' > k$) returned by the* Paa *algorithm. If (μ, ν) is (μ_3, ν_3)-derived then the set of invariants of $\mathcal{P}^{\#} = \alpha_{(\mu,\nu)}(\mathcal{P})$ is a subset of that of $\mathcal{P}^{\#'} = \alpha_{(\mu',\nu')}(\mathcal{P})$.*

Proof. *Derived from and Theorem 3.3 and Lemma 3.1.*

Algorithm 3.2 (PaaTerm).

Input *A program $\mathcal{P}$, a FP type $\mathbb{F}$, an integer $k_{max} \geq 1$*
An interval I in which range the results of all computations
$I = [-n_{min}, n_{max}]$ by default

Output "Yes" *which means that $\mathcal{P}$ terminates, can also answer* "I don't know"

Begin

1: $\mathcal{P}_{SOF} = \text{Sof}(\mathcal{P})$
2: $k = 1$
3: While *($k \leq k_{max}$) {*
4: $\quad (\mu, \nu) = \text{Paa}(\mathbb{F}, I, k)$
5: $\quad \mathcal{P}^{\#} = \alpha_{(\mu,\nu)}(\mathcal{P}_{SOF})$ *// Notice that $\mathcal{P}^{\#}$ only performs rational computations*
6: $\quad$ If *$\mathcal{P}^{\#}$ universally terminates* Then Return "Yes"
7: $\quad$ Else $k \leftarrow k + 1$
8: *}*
9: Return "I don't know"

End

Theorem 3.4. *Algorithm 3.2 is sound.*

Proof. *The transformation of $\mathcal{P}$ into $\mathcal{P}_{SOF}$ is just syntaxical: $\mathcal{P}$ terminates* if and only if *$\mathcal{P}_{SOF}$ does. If $\alpha_{(\mu,\nu)}(\mathcal{P}_{SOF})$ terminates then so does $\mathcal{P}_{SOF}$ by Proposition 1.1.*

Example 3.4 (Following Example 3.3). *$\mathcal{P}_g$ always terminates. Indeed for any initial value x_0 and y_0 of x and y:*

(Inv1) x strictly decreases at each iteration except when it is updated to 100 when reaching -100. That is x alternately becomes positive and negative.

(Inv2) y strictly increases at each iteration until reaching 100 if $-60 \leq y_0 < 100$ or y stays unchanged if $y_0 \geq 100$. That is y eventually becomes positive.

Following (Inv1) and (Inv2), x will be negative while y will be positive at some point: the loop condition $xy \geq 0$ will be eventually unsatisfied. The evolution of the capturing of (Inv1) and (Inv2) when analyzing $\mathcal{P}_g$ with PaaTerm *is shown in Figure 3.6.*

	$k=1$	$k=2$	$k=3$	$k=4$
(Inv1)	✓	✗	✗	✓
(Inv2)	✗	✓	✓	✓

FIGURE 3.6: Proving termination of $\mathcal{P}_g$ using PaaTerm. (Inv1) and (Inv2) need to be captured in order to prove terminatin of $\mathcal{P}_g$. We capture (Inv1) at $k = 1$ then we lose it at $k = 2$. This is because (μ_2, ν_2) is not nested in (μ_1, ν_1). Starting from $k = 2$, any invariant that is captured remains so due to Corollary 3.1 (and Theorem 3.3 coupled with Theorem 3.2 for the transition from $k = 2$ to $k = 3$).

We point out that the results of PaaTerm strongly depend on the interval I given as input. This is discussed later in Section 6.4.2 Q2. We also point out that our work is mainly focused on the piecewise-defined abstraction function $\alpha_{(\mu,\nu)}$. This is in contrast to the work of Urban [Urb15] who is interested in piecewise-defined ranking functions.

3.4 The Opt-ν problem

Now consider the problems raised in Algorithm 3.1: those of finding the affine functions that best upper and lower approximate ζ_n.

Definition 3.10 (Opt-μ, Opt-ν). Opt-ν *(resp.* Opt-μ*) is the problem of finding the affine segment $\nu(x)$ (resp. $\mu(x)$),* for all $x \in I, I = [\hat{x}_{min}, \hat{x}_{max}]$ *where $\hat{x}_{min}, \hat{x}_{max} \in \mathbb{F}$ that solves the following optimization problem:*

$$\begin{cases} minimize(S) & \\ S = \int_I \big(\nu(x) - \zeta_n(x)\big)dx & \big(resp.\ S = \int_I \big(\zeta_n(x) - \mu(x)\big)dx\big) \\ \zeta_n(x) \leq \nu(x) & \big(resp.\ \mu(x) \leq \zeta_n(x)\big) \\ \nu(x) = ax + b & \big(resp.\ \mu(x) = ax + b\big) \\ a \in \mathbb{Q}, b \in \mathbb{Q}, x \in \mathbb{R} & \end{cases} \tag{3.2}$$

In the following, we only give the solution for Opt-ν since that for Opt-μ can be retrieved in a similar manner.

3.4.1 A first solution to Opt-ν

As expressed in Definition 3.10 and considering Definition 2.7, the problem is daunting. A natural question arises: can we even solve it? To answer that question, notice that placing a segment above a set of segments can be simplified into placing it above the two endpoints, left and right, of each of them. Even better, for the particular case of the set of constant segments that define the rounding function ζ_n, we just have to consider the *left endpoints*. Indeed, the right endpoint of a constant segment is always below the left endpoint of the next constant segment as shown in Figure 3.7.

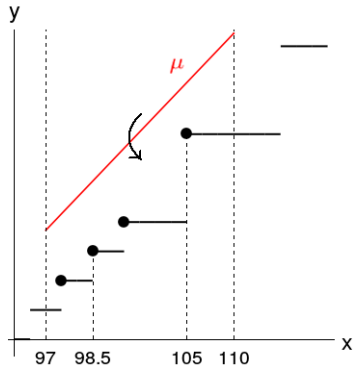


Figure 3.7: The Opt-μ problem for the FP type `myfloat` and the interval $I = [97, 110]$: we want to lower the affine segment $\nu(x)$ as much as possible while remaining above the four left endpoints.

This allows us to transform the constraints $\zeta_n(x) \leq \nu(x), \nu(x) = ax + b, x \in I$ of Equation 3.2 into a conjunction of linear inequalities. Suppose for example that we

want to solve OPT-ν for `myfloat` and for the interval $I = [97, 100]$. We transform the constraint $\zeta_n(x) \leq \nu(x), \nu(x) = ax + b, x \in [97, 110]$ as follows:

$$\begin{cases} 110 & \leq \nu(x) \text{ at } x = 105 \\ 100 & \leq \nu(x) \text{ at } x = 99.5 \\ 99 & \leq \nu(x) \text{ at } x = 98.5 \\ 98 & \leq \nu(x) \text{ at } x = 97.5 \\ \nu(x) & = ax + b \end{cases} \iff \begin{cases} 110 & \leq 105a + b \\ 100 & \leq 99.5a + b \\ 99 & \leq 98.5a + b \\ 98 & \leq 97.5a + b \end{cases}$$

It remains to notice that the objective function S to minimize is also a linear expression of a and b:

$$S = \frac{1}{2}(x_{max}^2 - x_{min}^2)a + (x_{max} - x_{min})b \tag{3.3}$$

Thus we managed to completely transform the OPT-ν problem into a Linear Programming (LP) problem. Continuing our example, OPT-ν is reduced to:

$$\begin{cases} minimize(S) \\ S = 1345.5a + 13b \\ 110 \leq 105a + b \\ 100 \leq 99.5a + b \\ 99 \leq 98.5a + b \\ 98 \leq 97.5a + b \\ a, b \in \mathbb{Q} \end{cases} \tag{3.4}$$

which we can solve by using for example the Simplex algorithm: $a = \frac{8}{5}$ and $b = -58$, that is the optimal ν is $\nu(x) = \frac{8}{5}x - 58$.

Theorem 3.5. OPT-ν *can be reduced into a linear programming problem.*

Proof. *The linearization process is shown in the preceding three paragraphs.*

We know that LP problems can be solved in polynomial time [Kha80]. Does that mean that the solution we just proposed is efficient? No. Indeed the size of the LP we obtain can become astronomical. In our illustrative example, for the FP type `myfloat` and the interval $I = [97, 110]$, we obtained a LP problem that has an objective function subject to four constraints. If we had $I = [97, \mathit{130}]$ instead,

we would have to consider *six* constraints as there are six left endpoints in that interval. Actually we need to consider as many constraints as the number of the FP numbers in I. Thus for the Ieee-754 binary64 format, we may have to consider up to approximately 2^{64} constraints. Clearly that naive transformation into a LP problem is not useful in practice.

3.4.2 A second solution to Opt-ν

Now we present an algorithm that solves Opt-ν very efficiently: in constant time regarding the considered FP type and the interval I. Our solution relies on the following intermediate result.

Lemma 3.2 (Endpoints lemma). *Let g be a real function of $x \in \mathbb{R}$ defined on the interval $I = [x_{min}, x_{max}]$. Let ν be an affine upper approximation of g on I: $g(x) \leq \nu(x), x \in I$. If $\nu(x_{min}) = g(x_{min})$ and $\nu(x_{max}) = g(x_{max})$ then ν is optimal: $\int_I \big(\nu(x) - g(x)\big)dx$ is minimal.*

Proof. *We reason by contradiction. If ν is placed lower then it will be under g at least at one point: ν will not be an upper approximation of g. If it is placed higher then the surface between ν and g will increase: ν will not be optimal.*

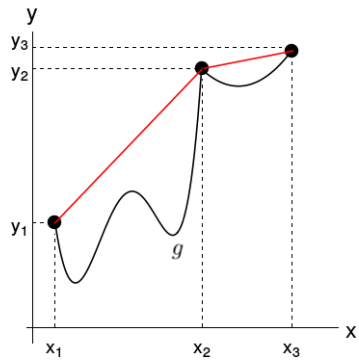


Figure 3.8: The endpoints lemma. Placing a segment above the function $g(x)$ on the interval $[x_1, x_3]$ is equivalent to simply placing it above the three points $P_i = (x_i, y_i), i \in \{1, 2, 3\}$.

Simply put, the endpoints lemma says that if an affine upper approximation function ν intersects the function g it approximates on two points x_1 and x_2 then ν is optimal on the interval $[x_1, x_2]$. The use of that lemma is as shown in Figure 3.8. If we can find a set of such intersecting points for the considered function g then we

just need to place ν above these points. Using that reasoning we can abstract the rounding function to-nearest ζ_n to a set of four points at most.

In the following, consider the euclidian plane on which is drawn the graph of ζ_n.A line drawn from a point P_1 of the plane to another one P_2 forms the segment $\overline{P_1P_2}$.

Algorithm 3.3 (Efficient solution for OPT-ν).

INPUT $I = [\hat{x}_{min}, \hat{x}_{max}]$ *and* $\hat{x}_{min}, \hat{x}_{max} \in \mathbb{F}$

OUTPUT ν *solving* OPT-ν

DEFINITION *We call ULP of an endpoint the ULP of its abscissa*

We call INTSEG *the operation that takes in parameter a segment and that returns the interval on which that segment is defined*

We call AFFSEG *the operation that takes in parameter a segment and that returns the affine function corresponding to it*

BEGIN

1: Determine the four points P_{min}, P_i, P_j *and* P_{max}

P_{min}, P_{max}*: consider the two left endpoints [of* ζ_n*] at the edges* I

P_{min} *is the one [of abscissa] closest to the origin*[2]

P_{max} *is the other one*

P_i*: left endpoint in* I *closest to the origin and having an ULP strictly greater than that of* P_{min}

IF *such point does not exist* THEN $P_i = P_{min}$

P_j*: left endpoint in* I *closest to the origin and of same ULP as* P_{max}

LET $p_{min}, p_i, p_j, p_{max}$ *respectively be the abscissas of* $P_{min}, P_i, P_j, P_{max}$

IF $p_{min}p_i \leq 0$ THEN $P_i \leftarrow P_j$

2: Choose the optimal ν

LET p_{mid} *be the midpoint of* I*:* $p_{mid} = \frac{\hat{x}_{min}+\hat{x}_{max}}{2}$

IF $p_{mid} \in$ INTSEG$(\overline{P_{min}P_i})$ THEN RETURN $\nu =$ AFFSEG$(\overline{P_{min}P_i})$

ELSE IF $p_{mid} \in$ INTSEG$(\overline{P_iP_j})$ THEN RETURN $\nu =$ AFFSEG$(\overline{P_iP_j})$

ELSE IF $p_{mid} \in$ INTSEG$(\overline{P_jP_{max}})$ THEN RETURN $\nu =$ AFFSEG$(\overline{P_jP_{max}})$

END

Theorem 3.6. *Algorithm 3.3 solves* OPT-μ *in constant time regarding the considered FP type* $\mathbb{F}$ *and the interval* I*.*

[2] That is the one with the smallest abscissa in absolute value.

Proof. *For the sake of conciseness, we only give the main arguments of the proof in the next four paragraphs* (§)*. More details are given in Appendix E.*

(§1)*. Let us call* ULP-increasing (left) endpoints *the left endpoints where the ULP increases when going from the origin to the infinities. For example in Figure 3.9(c), the left endpoint at abscissa* 105 *is an ULP-increasing endpoint.*

(§2)*. Given two left endpoints P_1 and P_2, the segment $\overline{P_1P_2}$ remains above ζ_n for any of the three following cases: (a) P_1 and P_2 are of same ULP, (b) P_1 and P_2 are both ULP-increasing endpoints, (c) P_2 is the first ULP-increasing endpoint after P_1 when leaving the origin.*

(§3)*. The left endpoints P_{min} and P_i satisfy case (c). The left endpoints P_i and P_j satisfy case (b). The left endpoints P_j and P_{max} satisfy case (a). Thus by the endpoints lemma, the segments $\overline{P_{min}P_i}$, $\overline{P_iP_j}$ and $\overline{P_jP_{max}}$ are optimally placed on their respective intervals.*

(§4)*. The fact that P_i is merged with P_j if $p_{min}p_i \leq 0$ is because in that case the segment $\overline{P_{min}P_j}$ remains above ζ_n, thus "short-circuiting" $\overline{P_{min}P_i}$ as illustrated on Figure 3.9(d).*

(§5)*. Now it remains to place ν above these three segments. (Claim 1) The solution is (the line that corresponds to) one of them. (Claim 2) The segment to choose is the one that is defined at the midpoint.*

To avoid making the text cumbersome, we do not give the algebraic values of the coordinates of P_{min}, P_i, P_j and P_{max}. Instead we will graphically illustrate the determination of these points with a few examples. We just point out that the computation of these coordinates is similar to the computation of the FP numbers preceding and following a given FP number. Intuitively we can express the coordinates of the left (resp. right) endpoint corresponding to a given FP number with the value of the FP number that precedes (resp. follows) it. Thus in the same way we know how to compute these straddling FP numbers very easily, see for example the functions `mpfr_nextbelow` and `mpfr_nextabove` of the MPFR library [Fou+07], we know how to determine P_{min}, P_i, P_j and P_{max} very efficiently.

Now the illustrative examples. In the following, we consider the FP type `myfloat` and we want to find the optimal ν for a given interval I. For the intuition, we will just say that the ULP of an endpoint is the length of its corresponding constant

segment. We emphasize though that that definition does *not* correspond exactly to the definition used in Algorithm 3.3: it just serves for the intuition.

Example 3.5 ($\mathbf{I} = [97, 110]$, see Figure 3.9(a)).

1: Determining P_{min}, P_{max} and P_i

- *P_{min}, P_{max}: the two left endpoints at the edges of I are $(97.5, 98)$ and $(105, 110)$. The first one is P_{min}, the second one is P_{max}.*
- *P_i (intuitive but approximative definition): left endpoint in I closest to the origin whose corresponding segment is longer than that of P_{min}:* $(105, 110)$
- *P_j: left endpoint in I closest to the origin and such that its corresponding segment is of same length as that of P_{max}:* $(105, 110)$

In this case, P_i, P_j and P_{max} are all the same point.

2: Choosing the optimal ν
We have $p_{mid} = \frac{97+110}{2} = 103.5$ which is such that $p_{mid} \in \textsc{IntSeg}(\overline{P_{min}P_i})$ since $\textsc{IntSeg}(\overline{P_{min}P_i}) = [p_{min}, p_i] = [97.5, 105]$. Thus the line $(P_{min}P_i)$ optimally approximates ζ_n on I: $\nu(x) = \frac{8}{5}x - 58$. That is indeed the solution we obtained after solving the linear programing problem of Equation 3.4.

We point out that the optimal ν we obtain does not correspond to any of the well-known approximations presented in Section 3.2.1.

Example 3.6 ($\mathbf{I} = [5, 120]$, see Figure 3.9(b)).

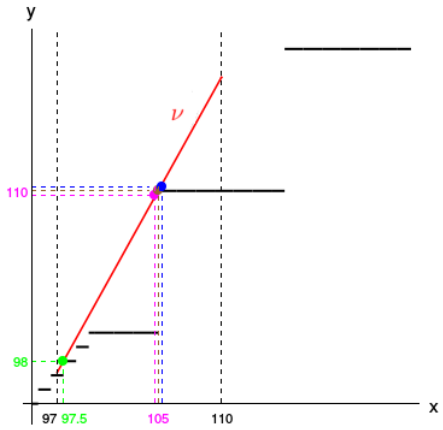
1: Determining P_{min}, P_i, P_j and P_{max}. See figure.

2: Choosing the optimal ν
We have $p_{mid} = \frac{5+120}{2} = 62.5$ which is such that $p_{mid} \in \textsc{IntSeg}(\overline{P_iP_j})$. Thus the line (P_iP_j) optimally approximates ζ_n on I: $\nu(x) = \frac{22}{21}x$.

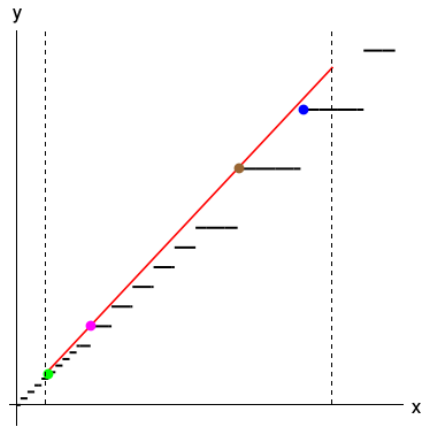
We point out that the FP numbers in I are normals and the optimal ν we obtain is the approximation by relative error as often preferred in the literature for approximating normals. That is $\nu(x) = \frac{22}{21}x = \left(1 + \mathcal{R}_n^o\right) \cdot x$ where $\mathcal{R}_n^o = \frac{\epsilon}{1+\epsilon}$ is the optimal bound of the relative errors for normals. This coincides with the results of [JR16].

Example 3.7 ($\mathbf{I} = [97, 130]$, see Figure 3.9(c)).

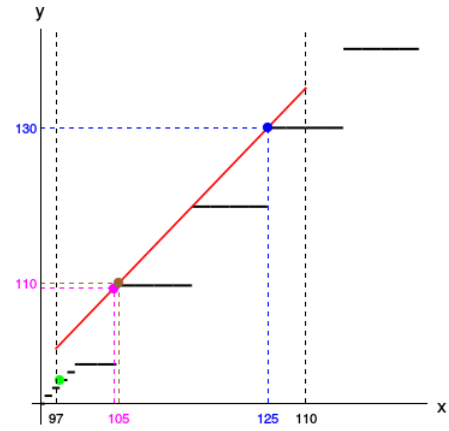
1: Determining P_{min}, P_i, P_j and P_{max}. See figure: in this case, P_i and P_j are the same point.



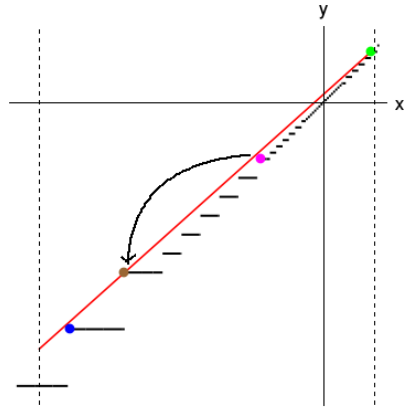
(a) $I = [97, 110]$



(b) $I = [5, 120]$, $P_{min} = (5.05, 5.1)$, $P_i = (10.5, 11)$, $P_j = (105, 110)$, $P_{max} = (115, 120)$. Approximative figure.



(c) $I = [97, 130]$, $P_{min} = (97.5, 98)$



(d) $I = [-130, 15]$, $P_{min} = (14.5, 15)$, $P_i = P_j = (-105, -100)$, $P_{max} = (-125, -120)$. The segment $\overline{P_{min}Pi}$ is "short-circuited" by the segment $\overline{P_{min}Pj}$. Approximative figure.

Figure 3.9: ν for `myfloat` and a given I

2: Choosing the optimal ν

We have $p_{mid} = \frac{97+130}{2} = 113.5$ which is such that $p_{mid} \in \text{IntSeg}(\overline{P_j P_{max}})$. Thus the line $(P_j P_{max})$ optimally approximates ζ_n on I: $\nu(x) = x + 5$.

We point out that though the FP numbers in I are normals, our algorithm says that the best approximation is not *the approximation by the relative error as in the case*

of Example 3.6 and as often preferred in the literature. Indeed ν is such that $\nu(x) = x + \mathcal{A}_I^o$ where $\mathcal{A}_I^o = \frac{ulp(x_{max})}{2} = 5$ is the optimal bound for the absolute error for the normals in I.

Example 3.8 ($\mathbf{I = [-130, 15]}$, see Figure 3.9(d)).
1: Determining P_{min}, P_i, P_j and P_{max}. See figure: in this case, we have $p_{min}p_i \leq 0$. Thus P_i is merged with P_j.

2: Choosing the optimal ν
We have $p_{mid} = \frac{15-130}{2} = -57.5$ which is such that $p_{mid} \in \textsc{IntSeg}(\overline{P_{min}P_i})$ since $\textsc{IntSeg}(\overline{P_{min}P_i}) = [p_i, p_{min}] = [-105, 14.5]$. Thus the line $(P_{min}P_i)$ optimally approximates ζ_n on I: $\nu(x) = \frac{230}{239}x + \frac{250}{239}$.

We point out that there are both normals and subnormals in I. This is to show that our algorithm handles seamlessly intervals containing normals only, subnormals only or a mix of both.

3.5 Conclusion

We have presented a new technique for analyzing termination of FP computations. Our technique consists in approximating the FP expressions by upper and lower piecewise affine functions. The novelty lies in the quality of the approximations which can be as tight as needed by increasing or decreasing the number of pieces. Also they are optimal for the chosen partitioning. In practice, we have to limit the number of pieces to reasonable values. This renders our approach incomplete: experiments need to be conducted in order to evaluate its efficiency. This is done in Chapter 6.

In this chapter we focused on developing rational approximations of FP computations. Then we just left the rest of the job to some termination analyzer for rational computations. Hence the efficiency of our approach strongly depends on the termination proof techniques that are implemented: we need to investigate them closely. In the next two chapters, we will see how to adapt a few well-known termination proof techniques so that they can handle FP numbers.

Chapter 4

On the Affine Ranking Problem for Simple Floating-Point Loops

Some of the results described in this chapter have been the subject of a publication in an international conference [MMP16a]. They are described here in an harmonized and enhanced manner.

Abstract. In this chapter, we are interested in detecting Affine Ranking Functions (ARFs) for Simple floating-point Loops ($SL_{\mathbb{F}}$). We already know that the existence of these functions can be decided in polynomial time for Simple rational Loops ($SL_{\mathbb{Q}}$). Here we show that the problem is coNP-hard for $SL_{\mathbb{F}}$. In order to work around that theoretical limitation, we present an algorithm that remains polynomial by sacrificing completeness. The algorithm is based on the Podelski-Rybalchenko algorithm and can synthesize in polynomial time the ARFs it detects. To our knowledge, our work is the first adaptation of this well-known algorithm to the FP numbers.

Résumé. Dans ce chapitre, nous nous intéressons à la détection de Fonctions de Rang Affines pour les Boucles Simples flottantes. Nous savons déjà que l'existence de ces fonctions peut être décidée en temps polynomial pour les Boucles Simples rationnelles. Ici, nous montrons que le problème est coNP-dur pour les Boucles Simples flottantes. Afin de contourner cette limitation théorique, nous présentons un algorithme qui reste polynomial en sacrifiant la complétude. L'algorithme est basé sur celui de Podelski-Rybalchenko et peut synthétiser en temps polynomial les Fonctions de Rang Affines qu'il détecte. À notre connaissance, notre travail est la première adaptation aux nombres flottants de cet algorithme bien connu.

Prerequisites. This chapter relies on the following notions from previous chapters:
Chapter 1: Termination Analysis
- Theorem 1.1: Decidability of termination of finite state programs
- Hypothesis 1.2: $P \subsetneq NP$
- Proposition 1.2: Program over-approximation and ranking functions
- Definition 1.6: $\mathrm{SL}_{\mathbb{Q}}$ and $\mathrm{SL}_{\mathbb{Z}}$
- Definition 1.7: LRFs for Simple Loops
- Definition 1.8: ARFs for Simple Loops

Chapter 2: FP Numbers
- Example 2.1: The toy FP type `myfloat`
- Property 2.9: Bound $\mathscr{A}_{sn}$ for the absolute error of subnormals and normals
- **Hypothesis 2.1: Behaviors of the FP computations**

Chapter 3: Piecewise Affine Approximations (PAA)
- Property 3.2: The (μ_1, ν_1) pair of approximation functions
- Definition 3.2: Pair of k-PAA
- Definition 3.8: (μ, ν)-abstraction

4.1 Introduction

This work is a continuation of a series of connected results concerning Simple Loops. [Tiw04] first showed that termination of loops of the form `while` $(Ux \leq u)$ `do` $x' = Vx + v$ `done` where the column vector of n variables x ranges over $\mathbb{R}^{n\times 1}$ is decidable. Then [Bra06] showed that the problem is also decidable when $x \in \mathbb{Q}^{n\times 1}$. It remains so even when $x \in \mathbb{Z}^{n\times 1}$ under the condition that $u = 0$. Then [BGM12] investigated the more general case of non-deterministic loops of the form `while` $(Ux \leq u)$ `do` $V \begin{pmatrix} x \\ x' \end{pmatrix} \leq v$ `done`. Termination of such loops was proved to be EXPSPACE-hard when $x, x' \in \mathbb{Z}^{n\times 1}$ and with the coefficients being rationals. The existence of *Linear* Ranking Functions (LRFs) for such loops is however known to be coNP-complete [Ben13][BG14]. The result applies even when the variables range over a finite set $E \subset \mathbb{Z}$ with $|E| \geq 2$, like the case of machine integers. Now we want to detect *Affine* Ranking Functions when variables are of FP type.

This chapter is organized as follows. Section 4.2 studies AffRF($\mathbb{F}$) which is the problem of finding ARFs for $\mathrm{SL}_{\mathbb{F}}$. It notably shows that the problem is coNP-hard. Section 4.3.2 presents our adaptation of the Podelski-Rybalchenko algorithm to the

FP numbers. It notably provides a sufficient but not necessary condition that ensures the existence of ARFs. The condition is checkable in polynomial time. The adaptation is achieved through the use of affine approximations. Section 4.4 concludes.

4.2 The AffRF(𝔽) problem

Definition 4.1 (Simple floating-point Loops ($SL_{\mathbb{F}}$)). *Similarly to $SL_{\mathbb{Q}}$ and $SL_{\mathbb{Z}}$, $SL_{\mathbb{F}}$ are loops that are described by the set of inequalities $A'' \odot \begin{pmatrix} x \\ x' \end{pmatrix} \leq b$ in which x and x' are column vectors of FP variables. The FP matrix multiplication $\odot$ is similar to the rational matrix multiplication with the difference that the operations are done within the set of the FP numbers:*

$$\begin{cases} a_{11} \odot x_1 \oplus a_{12} \odot x_2 \oplus \ldots \oplus a'_{1n} \odot x'_n \leq b_1 \\ a_{21} \odot x_1 \oplus a_{22} \odot x_2 \oplus \ldots \oplus a'_{2n} \odot x'_n \leq b_2 \\ \ldots \\ a_{m1} \odot x_1 \oplus a_{m2} \odot x_2 \oplus \ldots \oplus a'_{mn} \odot x'_n \leq b_m \end{cases}$$

where a_{ij}, x_i, x'_i and b_i are respectively elements of A'', x, x' and b. FP multiplications are performed before FP additions. Since there are no parentheses that indicate the order in which the FP additions are performed, we consider that they are done from left to right.

Definition 4.2 (AffRF(𝔽)). *AffRF(𝔽) is the problem of finding ARFs for $SL_{\mathbb{F}}$.*

Theorem 4.1. *AffRF(𝔽) is decidable.*

Proof. *Recall the proof of Theorem 1.1. In addition to checking non-repeating states in each partial execution, we also check the existence of ARFs that are valid for all of them. In the current case, a partial execution $\rho = X_1 X_2 \ldots X_m$ is a sequence of states X_i that satisfy the considered $SL_{\mathbb{F}}$ as in Definition 4.1. Now we say that $f(X) = CX + d, C \in \mathbb{Q}^{1\times n}, d \in \mathbb{Q}$, is a valid ARF for ρ if the following system of*

constraints Φ_ρ is satisfiable:

$$\begin{cases} CX_1 + d \geq 0 \\ CX_1 \geq 1 + CX_2 \\ \dots \\ CX_{m-1} + d \geq 0 \\ CX_{m-1} \geq 1 + CX_m \end{cases} \tag{4.1}$$

$\Phi = \bigwedge_{all\ \rho} \Phi_\rho$ is the system of constraints of validity of f for all possible partial executions. If Φ is satisfiable then the space of all the valid ARFs is described by $f(X) = CX + d$. Otherwise there is no valid ARF.

The decision algorithm we proposed is highly costly. Indeed suppose that there are n variables in the program and that there are N FP numbers that can be taken as values. Then in the worst case we need to build the system of constraints of validity from the N^{2n} possible transitions.

Question arises: can we have a more efficient algorithm that solves AffRF($\mathbb{F}$)? Notably we would like to know if a polynomial one exists. For that purpose, we rely on a recent result regarding the complexity of a subset of the problem.

Definition 4.3 (LinRF). *LinRF($\mathbb{Z}$) is the problem of finding LRFs for $SL_\mathbb{Z}$. When variables only range over a* finite *subset E of $\mathbb{Z}$, we denote LinRF(E) the corresponding problem.*

Lemma 4.1 ([BG14, Theorem 3.1]). *LinRF($\mathbb{Z}$) is coNP-hard. Even when variables range only over a finite subset of $\mathbb{Z}$, the problem remains coNP-hard.*

Theorem 4.2. *AffRF($\mathbb{F}$) is coNP-hard.*

Proof. *We show that there is a subset of AffRF($\mathbb{F}$) that is coNP-hard. To this end, we show that there are FP types for which Lemma 4.1 apply.*

Consider the finite set $Z_M \subset \mathbb{Z}$ defined as $Z_M = \{z \in \mathbb{Z} | -M \leq z \leq M\}$. For all $M \in \mathbb{N}, M > 0$, we can construct the FP type $\mathbb{F}_M$ defined by the parameters $\beta = M$, $p = 1$, $e_{min} = 0$ and $e_{max} = 1$ for which $Z_M = \mathbb{F}_M$. Both Z_M and $\mathbb{F}_M$ have the same elements. Notice that the operations used over the elements of $\mathbb{F}_M$, which are the FP addition and the FP multiplication as shown in Definition 4.1, are exact when assuming Hypothesis 2.1: $\forall \hat{x}_1, \hat{x}_2 \in \mathbb{F}_M : \hat{x}_1 \oplus \hat{x}_2 = \hat{x}_1 + \hat{x}_2$ and $\hat{x}_1 \odot \hat{x}_2 = \hat{x}_1 \cdot \hat{x}_2$.

Hence $\forall M \in \mathbb{N}, M > 0, LinRF(\mathbb{F}_M) = LinRF(\mathbb{Z}_M)$. *Since* $LinRF(\mathbb{Z}_M)$ *is coNP-hard by Lemma 4.1, so is* $LinRF(\mathbb{F}_M)$. *The theorem follows from the fact that* $LinRF(\mathbb{F}_M)$ *is a subset of AffRF(*$\mathbb{F}$*) as the class of LRFs is a subset of that of ARFs.*

Corollary 4.1. *There is no polynomial algorithm for deciding AffRF(*$\mathbb{F}$*).*

Proof. *We assume Hypothesis 1.2. As we already know [Sip97], the coNP class contains problems that are at least as difficult as the NP class.*

Although that theoretical limitation may be discouraging, it is important to point out that it applies to the problem of finding *one* general algorithm for *all* the possible instances of AffRF($\mathbb{F}$). There may be special cases for which polynomial decision algorithms exist. We could also have correct algorithms that are polynomial but not complete. That is we could have algorithms that detect in polynomial time only some parts of the space of the existing ARFs.

4.3 A sufficient condition for inferring Affine Ranking Functions in polynomial time

4.3.1 The Podelski-Rybalchenko algorithm

Our technique is based on the Podelski-Rybalchenko algorithm [PR04a].

Definition 4.4 (Podelski-Rybalchenko-type Ranking Function (PRRF) for $\mathbf{SL}_{\mathbb{Q}}$). *Let* $\mathcal{P}$ *be a* $SL_{\mathbb{Q}}$ *that has the column vector* $x \in \mathbb{Q}^{n\times 1}$ *as variables and the binary relation* $\mathcal{R}$ *as transition relation. The linear function* $f(x) = cx$, *where* $c \in \mathbb{Q}^{1\times n}$ *is a constant row vector, is a PRRF for* $\mathcal{P}$ *if* $\exists \delta, \delta_0 \forall x, x' : x\mathcal{R}x' \implies f(x') \geq \delta_0 \wedge f(x) \geq f(x') + \delta$ *where* $\delta, \delta_0 \in \mathbb{Q}$ *and* $\delta > 0$.

Theorem 4.3 (Podelski-Rybalchenko (PR) algorithm [PR04a]). *Given* $\mathcal{L}_{\mathbb{Q}}$, *a* $SL_{\mathbb{Q}}$ *described by* $A'' \begin{pmatrix} x \\ x' \end{pmatrix} \leq b$ *such that* $A'' \in \mathbb{Q}^{m\times 2n}$, $b \in \mathbb{Q}^{m\times 1}$ *and* $x, x' \in \mathbb{Q}^{n\times 1}$, *let* $A'' = (A\ A')$ *where* $A, A' \in \mathbb{Q}^{m\times n}$. *A PRRF exists for* $\mathcal{L}_{\mathbb{Q}}$ if and only if *there*

exist $\lambda_1, \lambda_2 \in \mathbb{Q}^{1\times m}$ *such that:*

$$\begin{cases} \lambda_1, \lambda_2 & \geq 0 \\ \lambda_1 A' & = 0 \\ (\lambda_1 - \lambda_2)A & = 0 \\ \lambda_2(A + A') & = 0 \\ \lambda_2 b & < 0 \end{cases}$$

in which case the space of PRRF is completely described by functions f *such that:*

$$f(x) = \theta x \text{ where } \theta = \lambda_2 A' \text{ and}$$
$$\forall x, x' : \begin{cases} f(x) \geq \delta_0 \text{ where } \delta_0 = -\lambda_1 b \\ f(x) \geq f(x') + \delta \text{ where } \delta = -\lambda_2 b, \delta > 0 \end{cases}$$

Proof. *Omitted as there are already various works discussing it in various ways. Interested readers can find in-depth study of the PR algorithm in [Bag+12]. Notably it points out the use of Farkas' Lemma in [Bag+12, Section 5.2].*

Example 4.1. *Consider the program* $\mathcal{P}_{ilog37q}$ *that has* $n = 2$ *variables presented in Figure 4.1. The loop of* $\mathcal{P}_{ilog37q}$ *can be expressed in the matrix form* $\begin{pmatrix} A_q & A'_q \end{pmatrix} \begin{pmatrix} x \\ x' \end{pmatrix} \leq b$ *by letting*

$$A_q = \begin{pmatrix} -1 & 0 \\ 0 & -1 \\ -1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix}, A'_q = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 37 & 0 \\ -37 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix}, b = \begin{pmatrix} -37 \\ -1 \\ 0 \\ 0 \\ -1 \\ 1 \end{pmatrix} \tag{4.2}$$

The corresponding linear system given by PR is satisfiable and the row vectors $\lambda_1 = \begin{pmatrix} \lambda_1^1 & \lambda_1^2 & \lambda_1^3 & \lambda_1^4 & \lambda_1^5 & \lambda_1^6 \end{pmatrix}$ *and* $\lambda_2 = \begin{pmatrix} \lambda_2^1 & \lambda_2^2 & \lambda_2^3 & \lambda_2^4 & \lambda_2^5 & \lambda_2^6 \end{pmatrix}$ *are such that:*

$$\begin{cases} \lambda_1^1 = -\lambda_1^2 + \lambda_1^4 + \lambda_2^1 + \lambda_2^3 - \lambda_2^4 \\ \lambda_1^2 = -\lambda_2^5 + \lambda_2^6 \\ \lambda_1^3 = \lambda_1^4 \\ \lambda_1^4 \geq 0 \\ \lambda_1^5 \geq 0 \\ \lambda_1^6 = \lambda_1^5 \end{cases} \quad \textit{and} \quad \begin{cases} \lambda_2^1 = 36\lambda_2^3 - 36\lambda_2^4 \\ \lambda_2^2 = 0 \\ \lambda_2^3 > 0 \\ 0 \leq \lambda_2^4 < \lambda_2^3 \\ \lambda_2^5 \geq 0 \\ \lambda_2^5 \leq \lambda_2^6 < 1332\lambda_2^3 - 1332\lambda_2^4 \end{cases} \tag{4.3}$$

Thus the space of ranking function for $\mathcal{P}_{ilog37q}$ that is detected by PR is described by $f(x_1, x_2) = \theta_1 x_1 + \theta_2 x_2$ where $\theta_1 > 0$ and $0 \leq \theta_2 < 36\theta_1$. The function f is such that $\exists \delta_0 > 0, \delta > 0 \forall x_1, x_2, x_1', x_2' : f(x_1', x_2') \geq \delta_0 \wedge f(x_1', x_2') + \delta \leq f(x_1, x_2)$.

```
rational x1 = input(), x2 = 1;
while(x1  >= 37 & x2 >=1) {
  x1 = x1 / 37;
  x2 = x2 + 1;
}
```

FIGURE 4.1: A program that computes and stores in `x2` the integer base-37 logarithm of `x1`, $\mathcal{P}_{ilog37q}$. A similar program with variables ranging over the integers is studied in [Bag+12].

Theorem 4.4 (PRRF and ARF). *PRRF and ARF are equivalent in the sense that any PRRF can be transformed into an ARF and any ARF can be transformed into a PRRF.*

Proof. *Let $(\mathbb{Q}^u, \leq^v)$ denote the WF structure such that $\mathbb{Q}^u = \{q \in \mathbb{Q} | q > u\}$ and $q' \leq^\delta q \iff q' + v \leq q$. The PRRF $f_{pr} = cx$ defined on the WF structure $(\mathbb{Q}^{\delta_0}, \leq^\delta)$ can be transformed into the ARF $f_{pra} = \frac{c}{\delta}x - \frac{-\delta_0}{\delta}$ defined on the WF structure $(\mathbb{Q}^0, \leq^1)$. Then the ARF $f_a = \phi x + \psi$ defined on the WF structure $(\mathbb{Q}^0, \leq^1)$ can be transformed into the PRRF $f_{apr} = \phi x$ defined on the WF structure $(\mathbb{Q}^{-\psi}, \leq^1)$.*

Example 4.2 (Continuing Example 4.1). *The space of ARF for $\mathcal{P}_{ilog37q}$ is completely described by $f(x_1, x_2) = \phi_1 x_1 + \phi_2 x_2 + \psi$ where $\phi_1 = \frac{1+\phi_2}{36}, \phi_2 \geq 0$ and $\psi = -1 - \phi_1 - 2\phi_2$.*

Thus *every time an ARF exists for a given $SL_{\mathbb{Q}}$, PR does find it in polynomial time.* Moreover PR can synthesize the ARF it detects. Unfortunately PR cannot be

applied to $\mathrm{SL}_{\mathbb{F}}$. Indeed the peculiarities of FP computations render invalid most of the mathematical results on which PR relies.

4.3.2 A FP version of Podelski-Rybalchenko

To work around Corollary 4.1, we sacrifice completeness in the following FP adaptation of PR.

Theorem 4.5 (General FP version of Podelski-Rybalchenko). *Consider the FP type $\mathbb{F}$ which has such parameters and which uses such rounding mode that $\mathcal{A}_{sn}$ is the maximal absolute error. Consider also $\mathcal{L}_{\mathbb{F}}$, the $SL_{\mathbb{F}}$ described by $A'' \odot \begin{pmatrix} x \\ x' \end{pmatrix} \leq b$ such that $A'' \in \mathbb{F}^{m\times 2n}$, $b \in \mathbb{F}^{m\times 1}$ and $x, x' \in \mathbb{F}^{n\times 1}$. By letting $A'' = (A\ A')$ where $A, A' \in \mathbb{F}^{m\times n}$ and if no overflow occurs, an ARF exists for $\mathcal{L}_{\mathbb{F}}$* if *there exist $\lambda_1, \lambda_2 \in \mathbb{Q}^{1\times m}$ such that:*

$$\begin{cases} \lambda_1, \lambda_2 & \geq 0 \\ \lambda_1 A' & = 0 \\ (\lambda_1 - \lambda_2)A & = 0 \\ \lambda_2(A + A') & = 0 \\ \lambda_2 \mathbf{c} & < 0 \end{cases}$$

where $c \in \mathbb{Q}^{m\times 1}$ and $c = b + colvect^m((4n-1)\mathcal{A}_{sn})$. Here, $colvect^m(e)$ denotes the column vector of m elements, all elements equaling e.

The detected ARF are of the form $f(x) = \phi x + \psi$ where $\phi = \frac{1}{\delta}\lambda_2 A'$ and $\psi = \frac{1}{\delta}\lambda_1 c$ where $\delta = -\lambda_2 c$.

Proof. *Let us call l_k the first member of the k-th inequality in $\mathcal{L}_{\mathbb{F}}$, that is:*

$$l_k = a_{k1} \odot x_1 \oplus a_{k2} \odot x_2 \oplus \ldots \oplus a'_{kn} \odot x'_n \tag{4.4}$$

$$l_k \leq b_k \tag{4.5}$$

By approximating by below each operation using the μ_1 lower approximation function from Property 2.9, we get an affine lower bound for l_k:

$$\begin{array}{ll}
(a_{k1}x_1 - \mathcal{A}_{sn}) \oplus a_{k2}\odot x_2 & \oplus \ldots \oplus a'_{kn}\odot x'_n \leq l_k \\
a_{k1}x_1 \oplus (a_{k2}x_2 - \mathcal{A}_{sn}) & \oplus \ldots \oplus a'_{kn}\odot x'_n - \mathcal{A}_{sn} \leq l_k \\
(a_{k1}x_1 + a_{k2}x_2 - \mathcal{A}_{sn}) & \oplus \ldots \oplus a'_{kn}\odot x'_n - 2\mathcal{A}_{sn} \leq l_k \\
 & \ldots \\
a_{k1}x_1 + a_{k2}x_2 & + \cdots + a'_{kn}x'_n - (4n-1)\mathcal{A}_{sn} \leq l_k
\end{array} \tag{4.6}$$

Combining Equation 4.5 with 4.6 we get:

$$\begin{array}{ll}
a_{k1}x_1 + a_{k2}x_2 + \ldots + a'_{kn}x'_n - (4n-1)\mathcal{A}_{sn} & \leq l_k \leq b_k \\
a_{k1}x_1 + a_{k2}x_2 + \ldots + a'_{kn}x'_n & \leq b_k + (4n-1)\mathcal{A}_{sn}
\end{array} \tag{4.7}$$

Hence the FP loop $\mathcal{L}_{\mathbb{F}}$ described by $A''\odot \begin{pmatrix} x \\ x' \end{pmatrix} \leq b$ is approximated by the rational loop $\mathcal{L}_{\mathbb{Q}}$ described by $A'' \begin{pmatrix} x \\ x' \end{pmatrix} \leq c$ where $c = b + colvect^m((4n-1)\mathcal{A}_{sn})$.

It remains to apply Theorem 4.3 and Theorem 4.4 on $\mathcal{L}_{\mathbb{Q}}$. If an ARF f is found for $\mathcal{L}_{\mathbb{Q}}$ then f is also an ARF for $\mathcal{L}_{\mathbb{F}}$ by Proposition 1.2.

Example 4.3. *Consider the program $\mathcal{P}_{ilog37q}$ from Example 4.1. We are interested in its FP version $\mathcal{P}_{ilog37f}$ in which variables are of* `myfloat` *type. The rounding mode is to-nearest.*

We have $\mathcal{A}_{sn} = 5$. Thus $\mathcal{P}_{ilog37f}$ is approximated by the rational loop $\mathcal{P}^{\#}_{ilog37f}$ described by $\begin{pmatrix} A_f & A'_f \end{pmatrix} \begin{pmatrix} x \\ x' \end{pmatrix} \leq c$ such that:

$$A_f = A_q, A'_f = A'_q, c = b + colvect^6(35) = c_2 = \begin{pmatrix} -2 & 34 & 35 & 35 & 34 & 36 \end{pmatrix}^T \tag{4.8}$$

By applying Theorem 4.3 and Theorem 4.4, ARF for $\mathcal{P}_{ilog37f}$ exist and are of the form $f(x_1, x_2) = \phi_1 x_1 + \phi_2 x_2 + \psi$ such that $\phi_1 \geq 1 + 36\phi_2, \phi_2 \geq 0$ and $\psi \geq -2\phi_1 + 34\phi_2$.

We can have a better approximation of $\mathcal{P}_{ilog37f}$ by noticing that there are less than $(4n-1)$ operations per line. Moreover by only approximating the operations that are not known to be exactly computed, we get the approximation $\mathcal{P}^{\#}_{ilog37f2}$ described by $\begin{pmatrix} A_{f2} & A'_{f2} \end{pmatrix} \begin{pmatrix} x \\ x' \end{pmatrix} \leq c_2$ such that:

$$A_{f2} = A_q, A'_{f2} = A'_q, c_2 = \begin{pmatrix} -37 & -1 & 10 & 10 & 4 & 6 \end{pmatrix}^T \tag{4.9}$$

By applying Theorem 4.3 and Theorem 4.4, ARF for $\mathcal{P}_{ilog37f2}$ exist and are of the form $f(x_1, x_2) = \phi_1 x_1 + \phi_2 x_2 + \psi$ such that $\phi_1 \geq \frac{37+222\phi_2}{1322}$, $\phi_2 \geq 0$ and $\psi \geq -37\phi_1 - \phi_2$. We can verify that the space of ARF for $\mathcal{P}^{\#}_{ilog37f}$ is a strict subset of the space of ARF for $\mathcal{P}^{\#}_{ilog37f2}$. That is we detect a wider range of ARFs with $\mathcal{P}^{\#}_{ilog37f2}$.

To get even more better approximations, we could also use pairs of k-PAA functions instead of simply affine ones. However it would introduce efficiency issues: the adaptation would not remain polynomial.

Theorem 4.6 (From a $\mathrm{SL}_{\mathbb{F}}$ to an exponential number of $\mathrm{SL}_{\mathbb{Q}}$). *Consider the FP type $\mathbb{F}$. Consider also the $SL_{\mathbb{F}}$ $\mathcal{L}_{\mathbb{F}}$ described by $A'' \odot \begin{pmatrix} x \\ x' \end{pmatrix} \leq b$ such that $A'' \in \mathbb{F}^{m \times 2n}$, $b \in \mathbb{F}^{m \times 1}$ and $x, x' \in \mathbb{F}^{n \times 1}$. Consider the pair (μ, ν) of k-PAA. The (μ, ν)-abstraction of $\mathcal{L}_{\mathbb{F}}$ is composed of a disjunction of $k^{(4n-1)m}$ $SL_{\mathbb{Q}}$.*

Proof. *The proof is similar to that for the general FP version of the PR algorithm (Theorem 4.5). The difference is that when approximating by below one operation, there are k cases to take into account. Since there are $(4n-1)m$ operations, the theorem follows.*

Last we focused on the specific case of $\mathrm{SL}_{\mathbb{F}}$. Our idea can be extended to the more general case of single-`while` programs that only use what we call $\mathbb{F}$-linear expressions.

Definition 4.5 ($\mathbb{F}$-linear expressions). *Let $\hat{E}$ be a FP expression. Replace all the FP operations in $\hat{E}$ by rational operations. If the obtained expression is linear then we say that $\hat{E}$ is $\mathbb{F}$-linear.*

We can show that if we approximate all FP operations in a $\mathbb{F}$-linear expression using some affine approximation function then we will get linear expressions.

4.4 Conclusion

We have studied the hardness of termination proofs for Simple Loops when the variables are of FP type. We have focused on termination inferred from the existence of ARFs. The problem of deciding the existence of LRFs for $\mathrm{SL}_{\mathbb{Z}}$ was studied in depth very recently and was shown to be coNP-complete. To the best of our knowledge, our work is the first attempt at providing a similar result for FP numbers: the problem of deciding the existence of ARFs for $\mathrm{SL}_{\mathbb{F}}$ is coNP-hard. This is a very valuable information as it dissuades us from looking for a decision algorithm that is both polynomial and complete.

To design a polynomial algorithm, we have traded completeness for efficiency. We have proposed the first adaptation of the Podelski-Rybalchenko algorithm for Simple FP Loops. This is achieved by means of affine approximations. As we have provided a sufficient but not necessary condition for inferring termination, experimentations need to be conducted in order to get a practical evaluation of the approach. This is left for future work.

Chapter 5

On the Local Approach for Floating-Point Computations

Abstract. In this chapter, we are interested in the local approach for analyzing termination of Floating-Point (FP) computations. Contrarily to the previous chapter which focused on Affine Ranking Functions (ARFs) for Simple FP Loops, this chapter considers a wider class of termination arguments for a wider class of programs. We rely on the Size-Change (SC) principle as well as on its generalization to monotonicity constraints. To our knowledge, our work is the first one that shows in details how to use these two well-known approaches when handling FP numbers.

Résumé. Dans ce chapitre, nous nous intéressons à l'approche locale pour analyser la terminaison des calculs flottants. Contrairement au chapitre précédent qui s'est concentré sur les Fonctions de Rang Affines pour les Boucles Flottantes Simples, ce chapitre considère une plus large classe d'arguments de terminaison pour une plus large classe de programmes. Nous nous basons sur le principe du Size-Change ainsi que sur sa généralisation aux contraintes de monotonicité. À notre connaissance, notre travail est le premier qui montre en détail comment utiliser ces deux approches bien connues lorsque les variables manipulées sont de types flottants.

Prerequisites. This chapter relies on the following notions from previous chapters:

Section 1.5.2: Local approach for proving termination
Definition 1.9: Constraint $D^{\prec}(x, x')$ of $\prec$-decrease
Definition 1.7: LRFs for SLs
Definition 1.8: ARFs for SLs
Section 2.2: FP numbers basics
Hypothesis 2.1: Hypotheses on the behaviors of the FP computations

Section 2.5: A few properties of FP computations
Definition 4.5: $\mathbb{F}$-linear expressions
Definition 3.8: (μ, ν)-abstraction
Notation 1.1: The $[\dots]$ notation

5.1 Introduction

We start by reminding the general algorithm used for analyzing termination of programs using the SC approach.

Algorithm 5.1 (SCGen).

INPUT *A program* $\mathcal{P} = (\mathcal{S}, \mathcal{S}, \mathcal{R})$
A WF structure $(E, \prec)$
An abstraction function $\alpha^{\prec}$*:*
$\mathcal{R}^{\prec} = \alpha^{\prec}(\mathcal{R})$ *is the abstraction of* $\mathcal{R}$ *that links the program variables through constraints of* $\prec$*-decrease*
OUTPUT *"YES" if* $\mathcal{R}^{\prec}$ *is cWF which implies that* $\mathcal{P}$ *terminates*
"NO" otherwise
BEGIN

1: $\mathcal{R}^{\prec} = \alpha^{\prec}(\mathcal{R})$
2: $\mathcal{R}^{\prec+} = \mathit{Closure}(\mathcal{R}^{\prec})$
$= \bigcup \underbrace{\left\{ l, x_1 \dots x_n, l', x'_1 \dots x'_n \middle| l =? \wedge l' =? \wedge D_i^{\prec}(x, x') \right\}}_{\mathcal{T}_i = \left[l=?, l'=?, D_i^{\prec}(x,x') \right] \text{ using the } [\dots] \text{ notation}}$
3: IF *there is at least one* $x'_j \prec x_j$ *in each idempotent* $\mathcal{T}_i$ THEN RETURN "YES"
OTHERWISE RETURN "NO"
END

This chapter is organized as follows. Section 5.2 shows to use SCGen for the analysis of FP computations. Section 5.3 extends the idea to monotonicity constraints. Section 5.4 concludes.

5.2 Size-change termination of floating-point computations

The WF structure $(E, \prec)$ and the abstraction function $\alpha^{\prec}$ need to be defined when using SCGen for analyzing FP computations. We respectively do so in Subsection 5.2.1 and 5.2.2. In Subsection 5.2.3, we present a full example.

5.2.1 Building the well-founded structure $(E, \prec)$

SC is often presented with the WF structure $(\mathbb{N}, <)$. We investigate the case of $(\mathbb{F}, <)$.

Proposition 5.1. *The structure $(\mathbb{F}, <)$ is WF.*

Proof. *$\mathbb{F}$ is finite and totally ordered by $<$, under the condition of Section 2.4, (MyDir3).*

Notation 5.1 ($\mathbb{Q}^{q_{min}}, \overline{\mathbb{Q}}^{q_{max}}$). *We denote $\mathbb{Q}^{q_{min}} = \{q \in \mathbb{Q} | q \geq q_{min}\}$ where $q_{min} \in \mathbb{Q}$. We denote $\overline{\mathbb{Q}}^{q_{max}} = \{q \in \mathbb{Q} | q \leq q_{max}\}$ where $q_{max} \in \mathbb{Q}$.*

Notation 5.2 ($\prec^{\delta}$). *We denote $\prec^{\delta}, \delta \in \mathbb{Q}$, the binary relation over $\mathbb{Q}$ such that $x_1 \prec^{\delta} x_2 \iff x_1 \leq x_2 + \delta$.*

Proposition 5.2. *The WF structure $(\mathbb{Q}^{-n_{max}}, \prec^{s_{min}})$ is a sound abstraction of $(\mathbb{F}, <)$: the set of all possible descents in $(\mathbb{F}, <)$ is a subset of those in $(\mathbb{Q}^{-n_{max}}, \prec^{s_{min}})$.*

Proof. *The smallest distance between two consecutive FP numbers is s_{min} and the smallest FP number is $-n_{max}$. This applies whether or not subnormals are supported.*

That is we can use $(\mathbb{Q}^{-n_{max}}, \prec^{s_{min}})$ instead of $(\mathbb{F}, <)$ in SCGen in a sound way: if SCGen answers "Yes" when called with $(\mathbb{Q}^{-n_{max}}, \prec^{s_{min}})$ as WF structure then so does it when called with $(\mathbb{F}, <)$.

Now we investigate the cases in which we know the ranges of the computations. Notably we want to know whether a range analysis might be of any help in our building of $(E, \prec)$.

Proposition 5.3. *Suppose that results of all FP computations lay in some range $I = [\hat{x}_{min}, \hat{x}_{max}]$. Let u be the smallest ULP of the numbers in I. Then the WF structure $(\mathbb{Q}^{\hat{x}_{min}}, \prec^{u})$ is a sound abstraction of $(\mathbb{F} \cap I, <)$.*

Proof. *Similar to that of Proposition 5.2.*

Now we investigate the cases in which we use multiple FP types. Indeed SCGen requires a *single* WF structure to perform.

Proposition 5.4. *Let $\mathbb{F}_1$ and $\mathbb{F}_2$ be two FP types. Suppose that $(\mathbb{Q}^{q_1}, \prec^{\delta_1})$ and $(\mathbb{Q}^{q_2}, \prec^{\delta_2})$ respectively are sound abstractions of $(\mathbb{F}_1, <)$ and $(\mathbb{F}_2, <)$. Then $(\mathbb{F}_1 \cup \mathbb{F}_2, <)$ is soundly abstracted by the WF structure $(\mathbb{Q}^{min(-q_1,-q_2)}, \prec^{min(\delta_1,\delta_2)})$.*

Proof. *Similar to that of Proposition 5.2.*

Proposition 5.4 can be extended to more that two FP types. This is useful for analyzing programs that manipulate single precision FP numbers, as well as double and quadruple precision ones for example. Also machine integers can be viewed as particular FP types, see proof of Theorem 4.2.

5.2.2 Defining the abstraction function $\alpha^{\prec}$

Definition 5.1 ($\prec$-abstraction). *Let E be a constraint that links $x = (x_1 \dots x_n)^T$ with $x' = (x'_1 \dots x'_n)^T$. Let Φ be some constraint of $\prec$-decrease that also links x with x'. We say that Φ is a $\prec$-abstraction of E if $\forall x, x' : E \implies \Phi$.*

Example 5.1. *See Figure 5.1.*

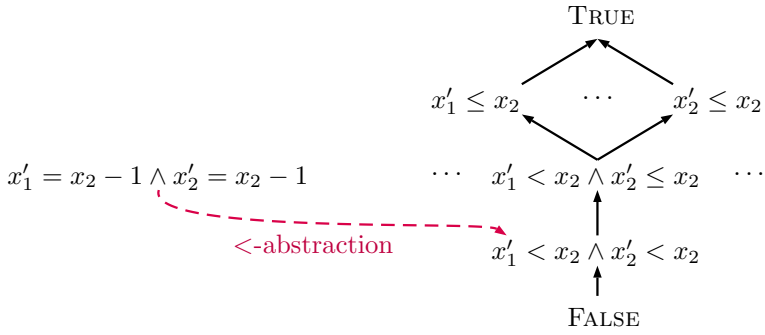


Figure 5.1: On the right, the lattice $L_2^<$ of all constraints of $<$-decrease organized by $\implies$ when $x = (x_1\ x_2)^T$ and x is rational. The most precise $<$-abstraction of a given constraint linking x with x' is the least element of $L_2^<$ that satisfies Definition 5.1.

Proposition 5.5. *The most precise $<$-abstraction, where $<$ is the usual ordering, of any FP arithmetic expression is computable.*

Proof. *We can enumerate all the possibles $<$-abstraction. We can also enumerate all the possible values of the variables. Then we can simply check the validity of each $<$-abstraction for each possible value of the variables.*

However finding the most precise $<$-abstraction of a general FP expression through mere brute enumeration is of no use in practice: it would take too much memory and time. In the following we restrict our study to $\mathbb{F}$-linear expressions.

Algorithm 5.2 ($\alpha^{<}_{\mathbb{F}\textbf{Lin}}$).

INPUT *An $\mathbb{F}$-linear expression $\hat{E}$ that links $\hat{x} = (\hat{x}_1 \dots \hat{x}_n)^T$ with $\hat{x}' = (\hat{x}'_1 \dots \hat{x}'_n)^T$*

OUTPUT *A $<$-abstraction $\hat{E}^{<}$ of $\hat{E}$*

BEGIN

1: LET *$\hat{E}^{\mathbb{Q}}$ be a (μ, ν)-abstraction of $\hat{E}$ obtained through the use of some affine approximation functions*

2: RETURN *$\alpha^{\prec}_{\mathbb{Q}\text{Lin}}(\hat{E}^{\mathbb{Q}})$ in which $\alpha^{\prec}_{\mathbb{Q}\text{Lin}}$ is defined in Algorithm 5.3*

END

Algorithm 5.3 ($\alpha^{\prec}_{\mathbb{Q}\textbf{Lin}}$).

INPUT *A linear expression $E \equiv c_1x_1 + \dots + c_nx_n + c'_1x'_1 + \dots + c'_nx'_n \leq b$*
A WF structure $(E, \prec)$ where E refers to $\mathbb{Q}^{q_{min}}$ for some q_{min} and $\prec$ is a shorthand for $\prec^{\delta}$ for some δ

OUTPUT *The most precise $\prec$-abstraction $E^{\prec}$ of E*

BEGIN

1: INIT $E^{\prec} \leftarrow$ TRUE

2: FOR *each possible $\Phi \equiv x'_i \mathrel{\overline{\prec}} x_j$ where $\mathrel{\overline{\prec}}$ refers to either $\prec$ or $\preceq$*
IF $\forall x, x' : E \implies \Phi$ THEN $E^{\prec} \leftarrow E^{\prec} \wedge \Phi$

3: RETURN $E^{\prec}$

END

Proposition 5.6. *Algorithm $\alpha^{<}_{\mathbb{F}\text{Lin}}$ only requires to check the satisfiability of a polynomial number of linear constraints in order to perform.*

Proof. *There are $2n^2$ possible Φ in $\alpha^{\prec}_{\mathbb{Q}\text{Lin}}$.*

We point out that satisfiability of the formula $\forall x, x' : E \implies \Phi$ in $\alpha^{\prec}_{\mathbb{Q}\text{Lin}}$ is decidable since theory of linear arithmetic over rationals is decidable. Also, the use of (μ, ν)-abstraction in $\alpha^{<}_{\mathbb{F}\text{Lin}}$ introduces incompleteness: we might have $\hat{E} \implies \Phi$ without having $\hat{E}^{<} \implies \Phi$.

5.2.3 Illustrative example

Let us analyze the program *pscf* presented in Figure 5.2 using the SC approach.

```
float x1 = randFp();
double x2 = 10;
while (x1 > 1 & x2 > 1) {
  if(randBool()) {
    x1 = x1 / 2;
    x2 = randFp();
  }
  else x2 = x2 / 3;
}
```

FIGURE 5.2: The program *pscf*, inspired from that of Figure 1.2.

For simplicity, let us admit that termination of *pscf* can be decided from the cWFness of the following transition relation: $\mathcal{R}_{pscf} = \underbrace{[x_1 > 1, x_2 > 1, x_1' = x_1 \oslash^f 2]}_{\hat{\Psi}_1} \cup \underbrace{[x_1 > 1, x_2 > 1, x_1' = x_1, x_2' = x_2 \oslash^d 3]}_{\hat{\Psi}_2}$ in which $\oslash^f$ (resp. $\oslash^d$) refers to the FP division performed using the IEEE-754 binary32 (resp. binary64) precision type.

Building the WF structure $(E, \prec)$. Let $\mathbb{F}_f$ and $\mathbb{F}_d$ respectively represent the IEEE-754 binary32 and binary64 precision type. The smallest subnormal in $\mathbb{F}_f$ is $s_f = 2^{-149}$ whereas that of $\mathbb{F}_d$ is $s_d = 2^{-1074}$. The biggest normal in $\mathbb{F}_f$ is $n_f = (2 - 2^{-23})2^{127}$ whereas that of $\mathbb{F}_d$ is $n_d = (2 - 2^{-52})2^{1023}$. By Proposition 5.2, we can respectively abstract $(\mathbb{F}_f, <)$ and $(\mathbb{F}_d, <)$ by $(\mathbb{Q}^{-n_f}, \prec^{s_f})$ and $(\mathbb{Q}^{-n_d}, \prec^{s_d})$. By Proposition 5.4, we can abstract $(\mathbb{F}_f \cup \mathbb{F}_d, <)$ by $(\mathbb{Q}^{min(-n_f,-n_d)}, \prec^{min(s_f,s_d)}) = (\mathbb{Q}^{-n_d}, \prec^{s_d})$.

Abstracting into constraints of $\prec$-decrease. First we respectively abstract the $\mathbb{F}$-linear expressions $\hat{\Psi}_1$ and $\hat{\Psi}_2$ by the linear ones $\hat{\Psi}_1^l$ and $\hat{\Psi}_2^l$:

$$\hat{\Psi}_1^l \equiv x_1 > 1 \wedge x_2 > 1 \wedge x_1' = \frac{x_1}{2}$$
$$\hat{\Psi}_2^l \equiv x_1 > 1 \wedge x_2 > 1 \wedge x_1' = x_1 \wedge$$
$$\wedge (1 - \mathcal{R}_n^d)\frac{x_2}{3} \leq x_2' \leq (1 + \mathcal{R}_n^d)\frac{x_2}{3} \wedge \mathcal{R}_n^d = 2^{-53}$$

where $\hat{\Psi}_1^l$ is obtained using Property 2.3 while $\hat{\Psi}_2^l$ is obtained using Property 3.1 combined with a range analysis that says that x_2 is always a normal number at the beginning of each iteration. Now we call $\alpha^{\prec}_{\mathbb{Q}^{\text{LIN}}}$ on these linear expressions and for the WF structure $(\mathbb{Q}^{-n_d}, \prec^{s_d})$. For example all the $x_i' \preceq^{s_d} x_j$ constraints that abstract

$\hat{\Psi}_2^l$ are listed as follows:

$$\begin{aligned}
\hat{\Psi}_2^l &\implies x_1' \preceq^{s_d} x_1\\
\hat{\Psi}_2^l &\implies x_2' \preceq^{s_d} x_2\\
\hat{\Psi}_2^l &\implies x_2' \prec^{s_d} x_2\\
\hat{\Psi}_2^l &\implies \textsc{True}
\end{aligned}$$

Thus the most precise constraint of $\prec^{s_d}$-decrease that abstracts $\hat{\Psi}_2^l$ is $\Phi_2 \equiv x_1' \preceq^{s_d} x_1 \wedge x_2' \preceq^{s_d} x_2 \wedge x_2' \prec^{s_d} x_2 \wedge \textsc{True} \equiv x_1' \preceq^{s_d} x_1 \wedge x_2' \prec^{s_d} x_2$. Similarly the most precise constraint of $\prec^{s_d}$-decrease that abstracts $\hat{\Psi}_1^l$ is $\Phi_1 \equiv x_1' \prec^{s_d} x_1$.

SC analysis. It remains to analyze the cWFness of the relation $\mathcal{R}_{pscf}^{\prec} = \mathcal{T} = [x_1' \prec^{s_d} x_1] \vee [x_1' \preceq^{s_d} x_1, x_2' \prec^{s_d} x_2]$. Its transitive closure is $\mathcal{T}^+ = [x_1' \prec^{s_d} x_1] \vee [x_1' \preceq^{s_d} x_1, x_2' \prec^{s_d} x_2] \vee [x_1' \prec^{s_d} x_1, x_2' \prec^{s_d} x_2]$. Since $\mathcal{T}^+$ is a disjunction of relations that all have a strict $\prec^{s_d}$-decrease then $\mathcal{R}_{pscf}$ is cWF.

5.3 Monotonicity constraints over floating-point numbers

We could also abstract into constraints of the form $x_i \overline{\prec} x_j'$ instead of $x_i' \overline{\prec} x_j$ in $\alpha^{\prec}_{\mathbb{Q}\text{Lin}}$, Algorithm 5.3.

Proposition 5.7. *The structure $(\mathbb{F}, >)$ is WF. The WF structure $(\mathbb{Q}^{n_{max}}, \succ^{s_{min}})$ where $\succ^{s_{min}}$ is the converse of $\prec^{s_{min}}$ is a sound abstraction of $(\mathbb{F}, >)$.*

Proof. *Similar to that of Proposition 5.2 with noticing that the biggest FP is n_{max}.*

Example 5.2 (Following Section 5.2.3). *Suppose we had $\mathcal{R}_{pscf} = [x_1 < 1000, x_2 < 1000, x_1' = x_1 \odot^f 2] \cup [x_1 < 1000, x_2 < 1000, x_1' = x_1, x_2' = x_2 \odot^d 3.5]$ in which $\odot^f$ (resp. $\odot^d$) refers to the FP multiplication performed using the* IEEE-754 *binary32 (resp. binary64) precision type.*
First the WF structure to consider is $(\mathbb{Q}^{n_d}, \succ^{s_d})$. Then the abstraction into constraints of $\succ^{s_d}$-decrease is $\mathcal{R}_{pscf}^{\prec} = \mathcal{T} = [x_1' \succ^{s_d} x_1] \vee [x_1' \succeq^{s_d} x_1, x_2' \succ^{s_d} x_2]$. Its transitive closure is $\mathcal{T}^+ = [x_1' \succ^{s_d} x_1] \vee [x_1' \succeq^{s_d} x_1, x_2' \succ^{s_d} x_2] \vee [x_1' \succ^{s_d} x_1, x_2' \succ^{s_d} x_2]$. Since $\mathcal{T}^+$ is a disjunction of relations that all have a strict $\succ^{s_d}$-decrease then $\mathcal{R}_{pscf}$ is cWF.

Then we can mix constraints of $\prec$-decrease with those of $\succ$-decrease. We can enrich the abstractions even more by considering constraints of the form $x_i \overline{\prec} x_j$ as

well as $x'_i \overline{\prec} x'_j$. We get monotonicity constraints, which generalize constraints of decrease.

Definition 5.2 ($(\mathbb{N},<)$-monotonicity constraint, [CLS05, Definition 1]). *Let $x = (x_1 \dots x_n)^T$ be a vector of variables, $x_i \in \mathbb{N}$. The constraint $M^<(x,x')$ is a constraint of $(\mathbb{N},<)$-monotonicity if it is a (possibly void) conjunction of either $\overline{x} < \overline{y}$ or $\overline{x} \leq \overline{y}$ where $\overline{x},\overline{y} \in \{x_1 \dots x_n, x'_1 \dots x'_n\}$.*

Definition 5.3 (Balanced $(\mathbb{N},<)$-monotonicity constraint, [CLS05, Definition 8]). *A $(\mathbb{N},<)$-monotonicity constraint $M^<(x,x')$ is balanced if $\big(M^<(x,x') \implies x_i < x_j\big) \iff \big(M^<(x,x') \implies x'_i < x'_j\big)$.*

Lemma 5.1 ([CLS05, Theorem 5]). *Let $M^<(x,x')$ be a* balanced *$(\mathbb{N},<)$-monotonicity constraint that is* not necessarily idempotent*. If there is a ranking function for $M^<(x,x')$ then there is a LRF for the $SL_{\mathbb{Z}}$ described by $M^<(x,x')$.*

Now we investigate $(\mathbb{F},<)$-monotonicity constraints.

Definition 5.4 ($(\mathbb{Q}^{q_{min}},\prec^\delta)$-monotonicity constraint $M^\prec(x,x')$, $(\mathbb{N},<)$-reduction). *Similar to Definition 5.2 except that we use $\mathbb{Q}^{q_{min}}$ for some q_{min} instead of $\mathbb{N}$ and $\prec^\delta$ for some δ instead of $<$. The $(\mathbb{N},<)$-reduction of $M^\prec(x,x')$ is the $(\mathbb{N},<)$-monotonicity constraint obtained simply by making the variables live in $\mathbb{N}$ and using $<$ instead of $\prec^\delta$.*

Proposition 5.8. *Let $M^\prec(x,x')$ be a $(\mathbb{Q}^{q_{min}},\prec^\delta)$-monotonicity constraint. If there is a ranking function for $M^\prec(x,x')$ then there is a LRF for its balanced $(\mathbb{N},<)$-reduction.*

Proof. *$(Q^{q_{min}},\prec^\delta)$ has a sub-order isomorphic to the naturals, namely the set $\{q_{min} + i\delta | i \geq 0\}$. Thus from the assumption that $M^\prec$ has a ranking function, it follows that the $(\mathbb{N},<)$-reduction of M also has a ranking function by application of the isomorphism. Now the conclusion about the balanced constraint is obtained through Lemma 5.1.*

Algorithm 5.4 (MonF).

INPUT *A program $\mathcal{P} = (\mathcal{S},\mathcal{S},\mathcal{R})$*
A relation $\prec^\delta$ that we shorten $\prec$ and
that is WF on some $\mathbb{Q}^{q_1}$ and *cWF on some $\overline{\mathbb{Q}}^{q_2}$*
(see Notation 5.1)
An abstraction function $\alpha^\Diamond$:

$\mathcal{R}^{\Diamond} = \alpha^{\Diamond}(\mathcal{R})$ *is the abstraction of* $\mathcal{R}$ *that links the program variables through constraints of* $\prec$*-monotonicity*

OUTPUT *"*YES*"* $\mathcal{P}$ *terminates*
*"*I DON'T KNOW*" otherwise*

BEGIN

1: $\mathcal{R}^{\Diamond} = \alpha^{\Diamond}(\mathcal{R})$

2: $\mathcal{R}^{\Diamond^+} = Closure(\mathcal{R}^{\Diamond})$
$= \bigcup \mathcal{T}_i$ *where* $\mathcal{T}_i = \left[l =?, l' =?, M_i^{\prec}(x, x')\right]$

3: FOR EACH $M_i^{\prec}(x, x')$ *of the* $\mathcal{T}_i$ *that have* $l = l'$
LET $M_i^{<}(x, x')$ *be the balanced* $(\mathbb{N}, <)$*-reduction of* $M_i^{\prec}(x, x')$
IF $M_i^{<}(x, x')$ *does not admit ARF*
THEN RETURN "I DON'T KNOW"

4: RETURN "YES"

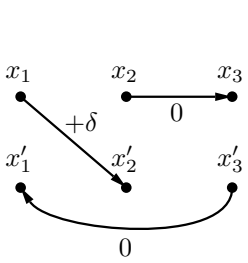
END

Theorem 5.1. *Algorithm 5.4 is sound.*

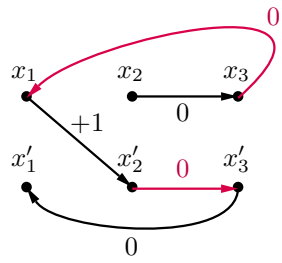
Proof. *The* $l = l'$ *test is because if* $l \neq l'$ *then* $\mathcal{T}_i$ *is trivially WF/cWF. If each* $M_i^{\prec}(x, x')$ *admits ARF then* $\mathcal{R}^{\Diamond}$ *is cWF by Termination Characterization 1.2.f:* $\mathcal{P}$ *terminates.*

Example 5.3. *Suppose that we could abstract* $<$ *on* $\mathbb{F}$ *by* $\prec^{\delta}$ *for some* δ *that is WF on* $\mathbb{Q}^{q_{min}}$ *for some* q_{min} *and cWF on* $\overline{\mathbb{Q}}^{q_{max}}$ *for some* q_{max}*. Consider the monotonicity constraint* $M^{\prec} \equiv x_1 \prec^{\delta} x_2' \wedge x_2 \preceq^{\delta} x_3 \wedge x_3' \preceq^{\delta} x_1'$ *in which the variables are such that* $q_{min} \leq x_i \leq q_{max}$*. The balanced* $(\mathbb{N}, <)$*-reduction of* $M^{\prec}$ *is* $M^{<} \equiv x_1 < x_2' \wedge x_2 \leq x_3 \wedge x_3' < x_1' \wedge x_2' \leq x_3' \wedge x_3 \leq x_1$ *in which the variables are such that* $n_{min} \leq x_i \leq n_{max}$ *for some* n_{min} *and* n_{max} *in* $\mathbb{N}$*. See Figure 5.3 for illustration. We can check with PR that* $M^{<}$ *admits ARFs, for example* $f(x_1, x_2, x_3) = -x_1 + n_{max}$ *since it is such that* $\forall x_i, x_i' : M^{<} \implies f(x_1, x_2, x_3) \geq f(x_1', x_2', x_3') + 1 \wedge f(x_1, x_2, x_3) \geq 0$*. Thus* $M^{\prec}$ *has a ranking function.*

We point out that since the set of FP numbers is finite, we can always construct an abtraction $\prec$ on $\mathbb{Q}$ of $<$ on $\mathbb{F}$ that is both WF and cWF. Now we have the intuition that the test made at step 3 of Algorithm 5.4 is complete for $\mathcal{R}^{\Diamond}$. That is, $\mathcal{R}^{\Diamond}$ is cWF *if and only if* each $M_i^{\prec}(x, x')$ admits ARF. We believe so because the test of Proposition 5.8 considers LRFs for a relation $\prec$ which is WF but *not* cWF. To make it complete for the case where $\prec$ is WF *and* cWF, we also need to take into account



(a) Monotonicity graph of $M^{\prec}$



(b) Monotonicity graph of the balanced $(\mathbb{N}, <)$-reduction $M^{<}$ of $M^{\prec}$. The variable x_1 is increased by one through $M^{<}$.

FIGURE 5.3: Illustration of Example 5.3

the infinite increasing chains: we believe the extension to ARFs does so. This is yet to be proven and is left for future work.

5.4 Conclusion

We have shown how to use the SC and the monotonicity constraints frameworks for the analysis of FP computations. Our work can be seen as part of a series of connected recent results on these two approaches. Notably [Ben09] considered the case of constraints over any well-founded domain which was afterwards studied in depth for the particular case of the integer domain in [Ben11].

About complexity concerns, deciding termination by SC is a PSPACE-complete when considering $(\mathbb{N}, <)$. However a polynomial restriction of SC has already been developped in [BL07] and has produced very satisfactory practical results. The FP adaptations of the SC and monotonocity constraints frameworks that we developed need to be put to the test by practical experiments. This is left for future work.

Part III

Practical Experiments

Chapter 6

Practical Experiments

Abstract. In this final chapter, we experiment some of the techniques we developped in this thesis. Our objective is twofold. First we evaluate the capabilities of current state-of-the-art termination analyzers regarding Floating-Point (FP) computations. Then we show that our techniques give good results in practice and that they can hold their ground against those implemented in currently existing tools. The evaluation is done on a test suite that we built from source codes of widely used programs. To our knowledge, our test suite is the first one that is designed for termination analysis of FP computations.

Résumé. Dans ce dernier chapitre, nous expérimentons certaines des techniques que nous avons développées dans cette thèse. Notre objectif est double. Tout d'abord, nous évaluons les performances des analyseurs de terminaison actuels par rapport aux calculs flottants. Ensuite, nous montrons que nos techniques donnent de bons résultats dans la pratique et qu'elles tiennent tête à celles mises en œuvre dans les outils existants. L'évaluation est faite sur une suite de tests que nous avons élaborée à partir de codes sources de programmes largement utilisés. À notre connaissance, notre suite de tests est la première dédiée à l'analyse de terminaison de calculs flottants.

Prerequisites. This chapter relies on the following notions from previous chapters:

Definition 1.7: Linear Ranking Functions (LRFs)
Section 2.4: Hypotheses on the behaviors of the FP computations
Algorithm 3.2: PAATERM
Property 3.2: The (μ_2, ν_2) pair of approximation functions
Definition 4.5: $\mathbb{F}$-linear expressions
Section 4.3.2: A FP version of Podelski-Rybalchenko

6.1 Introduction

All the experimentations are conducted on a regular machine and with a timeout as shown in Figure 6.1.

Machine's processor	Intel Core i5-5200U 2.20GHz
Machine's RAM	8GB
Time out	300s

FIGURE 6.1: Parameters of the experimentations

This chapter is organized as follows. Section 6.2 presents the test suite we designed for termination analysis of FP computations. Section 6.3 shows existing tools handle it. Section 6.4 presents a research prototype we developped. It implements the PAATERM algorithm presented in Chapter 3. Section 6.5 studies termination of a specific FP computation. The proof is semi-manual. This last section illustrates the gap that fully automatic termination provers still need to fill in order to analyze relatively complex FP computations. Section 6.6 concludes.

6.2 The Floops test suite

We want to evaluate the capabilities of the current state-of-the-art termination analyzers regarding FP computations. We build a test suite for that purpose. Indeed the author knows of no test suite dedicated to termination analysis of FP computations. For example TermComp does not consider FP computations yet.

TermComp [Gie+15]. TermComp is an international competition which aims to evaluate tools for fully-automated termination analysis. It has been organized annually since 2004. Different programming paradigms are considered: term rewriting systems, imperative programming, logic programming and functional programming. The goal is to demonstrate the power of the leading tools in each of these areas. The Termination Problem Data Base[1] (TPDB) is the collection of all the examples used in the competition. *For the time being, there is no entry dedicated to FP computations in the TPDB.*

The Floops test suite[2]. We want our test suite to be built from already existing programs. Our aim is to show that termination of FP computations is a practical

[1] http://termination-portal.org/wiki/TPDB
[2] https://bitbucket.org/fmaurica/floops/

problem and concerns a wide variety of software. Also we want our test suite to be built from publicly available source codes. This leaves anyone free to use it for future experimentations. For these reasons, we decided to skim the C source codes of the Ubuntu distribution and we selected the programs presented in Figure 6.2: we call Floops the obtained test suite. In the process we discovered buggy programs which we reported to the developers[3] [4].

	Package	Description
ubuntu1.c	bcache-tools	Marsaglia polar method
ubuntu2.c	gauche	Random number generator
ubuntu3.c	ghostscript	Round a binary64 number
ubuntu4.c	glibc	Excerpt of Bessel's function
ubuntu5.c	libpano	Rotate equirectangular image
ubuntu6.c	maxlib	Modulo of a binary64 number
ubuntu7.c	maxlib	Large-Kolen adaptation model
ubuntu8.c	postfix	Handle destination delivery failure
ubuntu9.c	postfix	Handle destination delivery failure
ubuntu10.c	bash	Integer log10
ubuntu11.c	git	Newton's method with binary32 numbers
ubuntu12.c	open-vm-tools	Newton's method with binary64 numbers

FIGURE 6.2: The Floops test suite: a test suite for termination analysis of FP computations built from the source code of the Ubuntu distribution.

6.3 State-of-the-art termination analyzers

Now how do current state-of-the-art termination analyzers handle Floops? We investigated Julia, Juggernaut, 2LS and AProVE. Results are shown in Figure 6.3.

Julia[5] **[Spo16]**. Julia is a commercial tool for static analysis of Java code and Java bytecode. Julia won the first prize in the Java Bytecode Recursive Contest at TermComp in 2009 and 2010. It uses Abstract Interpretation techniques. Unfortunately

[3] In the ImageMagick package, http://www.imagemagick.org/discourse-server/viewtopic.php?t=31506. The promptness with which the developers fixed the bug is noteworthy. ImageMagick is used on many web servers including those of Facebook: a bug in it can cause dramatic events, http://www.securityweek.com/facebook-awards-40000-bounty-imagetragick-hack.

[4] In the GSL library, https://savannah.gnu.org/bugs/?50459.

[5] https://www.juliasoft.com/

	Julia	Juggernaut		2LS		AProVE
ubuntu1.c	?	✗	3s	?	8s	?
ubuntu2.c	?	✗	43s	?	1s	?
ubuntu3.c	?	?	TO	?	5s	?
ubuntu4.c	?	?	TO	?	TO	?
ubuntu5.c	?	✓	4s	?	6s	?
ubuntu6.c	?	?	TO	?	TO	?
ubuntu7.c	?	?	TO	?	2s	?
ubuntu8.c	?	?	TO	?	TO	?
ubuntu9.c	?	?	TO	?	4s	?
ubuntu10.c	?	✓	6s	✓	2s	?
ubuntu11.c	?	?	TO	?	TO	?
ubuntu12.c	?	?	TO	?	TO	?

FIGURE 6.3: ✓: always terminates. ✗: there is at least one possible value of the variables that leads to non-termination. ?: can decide neither ✓nor ✗. TO: time out after 300s.
Initial ranges of the variables are fixed, to $[0, 10^3]$ for most of all.

it cannot prove nor disprove termination of any of (Java translations of) Floops' programs. According to one of Julia's scientific consultant, this is because the tool does not support FP computations yet: for the time being they are simply abstracted into the greatest element $\top$ of the abstract lattice that is taken into consideration.

Juggernaut[6] **[DKL15]**. Juggernaut is a termination prover for C programs. FP numbers are supported. Juggernaut generates a termination specification from the source code. Then it calls a second-order SAT solver. The tool is complete: it can *always* decide termination of the considered program. The only question is whether it can do so within the alloted time. Of the 12 programs of Floops, Juggernaut says that 2 universally terminates and 2 existentially non-terminates.

2LS[7] **[SK16]**. 2LS is a tool for analyzing C programs. It supports termination analysis of FP computations. 2LS uses a template-based approach. Hence the problem is reduced to quantifier elimination in first order logic. Of the 12 programs of Floops, 2LS can only decide termination of one. However 2LS times out less often compared to Juggernaut.

[6] https://github.com/blexim/synth/
[7] http://svn.cprover.org/wiki/doku.php?id=2ls_for_program_analysis

AProVE[8] **[Gie+17]**. AProVE is a system for automated termination and complexity proofs of Term Rewrite Systems (TRS) and derivatives. AProVE also handles several other formalisms such as imperative programs (Java Bytecode and C / LLVM), functional programs (Haskell 98) and logic programs (Prolog). The power of AProVE is demonstrated at TermComp where it regularly wins first places in several categories. Unfortunately AProVE does not provide support for FP numbers for the time being.

Non-determinism. The first three programs ubuntu1.c, ubuntu2.c and ubuntu3.c use non-deterministic assignements. They should be treated carefully. Consider for example ubuntu1.c which is shown in Figure 6.4. The program may or may not always terminate depending on the semantics of the `nondet()` function:

(ubuntu1-a) Suppose `nondet()` to be uniform. Notably suppose that it returns all possible binary64 numbers when called infinitely many times. Then ubuntu1.c always terminates. Indeed we will eventually have, say, $x = 0$ and $y = 0$ which will unsatisfy the loop condition.

(ubuntu1-b) Suppose `nondet()` to be not uniform. In particular, suppose that it always returns some binary64 numbers while never returning other ones when called infinitely many times. Then ubuntu1.c can fail to terminate. Indeed we can keep getting, say, $x = 1$ and $y = 1$ which always satisfies the loop condition.

```
double x, y, s;
do {
  x = nondet(); y = nondet();
  s = x * x + y * y;
} while (s >= 1);
```

FIGURE 6.4: ubuntu1.c as passed to Juggernaut. The `nondet()` function is a function specific to Juggernaut for indicating non-deterministic values. Juggernaut says that ubuntu1.c can fail to terminate: Juggernaut considers `nondet()` to be not uniform, see Point (ubuntu1-b).

[8] http://aprove.informatik.rwth-aachen.de/

6.4 The FTerm analyzer

Now we present a tool we developped for analyzing termination of FP computations. It uses the termination by gradual tightness approach. First in Subsection 6.4.1, we present a few relevant implementation details. Then in Subsection 6.4.2, we evaluate the practical efficiency of our tool.

6.4.1 Implementing termination by gradual tightness

FTerm[9]**.** We have implemented the PaaTerm algorithm in Java. We call FTerm our tool. It is as follows:

Input: a single-while program $\mathcal{P}$ written in a C-like syntax, see Figure 6.5.
Output: "Yes" $\mathcal{P}$ terminates for all possible input or "I don't know".
Supported FP types: any FP type of parameters $\beta, p, e_{min}, e_{max}$.
Supported rounding mode: to-nearest.

```
myfloat(2,53,-1022,1023) time, x_expected, x_period;
while(x_expected + x_period <= time) {
  x_expected = x_expected + x_period;
}
grange(0 <= time <= E15, 0 <= xexpected <= E15,
       1 <= xperiod <= E15);
```

Figure 6.5: The program `ubuntu7.c` from the Floops test suite written in FTerm's input syntax. The considered FP type is defined through the instruction `myfloat`$(\beta, p, e_{min}, e_{max})$. The instruction `grange`$(x_1 \leq a_1 \leq b_1, \ \ldots, \ x_n \leq a_n \leq b_n)$ indicates the global range of each variable: $x_i \in [a_i, b_i]$.

Internal operations (Binterm, Aprational). The two following points are of interest regarding the internal operations performed by FTerm:

(Int1) We use Binterm[10] [SMP10] to analyze termination of the obtained rational approximations. Binterm is the analyzer that was used by Julia to win first prizes at TermComp. It implements the following techniques: synthesis of LRFs, ELRFs, LLRFs and the monotonicity constraints framework.

[9] https://bitbucket.org/fmaurica/fterm/
[10] https://bitbucket.org/fmaurica/binterm/

(Int2) We use the Aprational library[11] for most of the computations. Indeed FTerm only manipulates rational numbers in order to avoid rounding error problems and overflow problems.

Exceptional conditions handling. We consider that exceptional conditions such as overflows or division by zero do not occur. This is an information we can use to enrich the constraints that define the program to study. For example it gives the following constraints from the operation $t = x \textcircled{/} y$: $t, x, y \in [-n_{max}, n_{max}]$ and $y \neq 0$. Another way to interpret this is to say that if an exceptional condition occurs then the program terminates, since the corresponding constraint has been violated.

Automation level. FTerm is still experimental. Much of PAATERM is implemented but there are a few things that remain manual for the time being:

(Man1) We manually provide the global ranges of the variables. That is the ranges within which the variables lay at any program point at any time. The keyword `grange` serves this purpose, see Figure 6.5.

(Man2) We manually indicate known FP properties such as the exactness of the FP multiplication by a power of β.

These manipulations can be automated as follows for future versions. First (Man1) can be automated by making FTerm call tools such as Fluctuat to get the ranges. Then (Man2) can be automated by providing the list of FP properties that we know and that we want to consider.

6.4.2 Practical evaluation

How does FTerm handle Floops? Of the 12 programs in the benchmark, 8 are single-while ones that can be analyzed by FTerm. Results are shown in Figure 6.6. We comment them in a Q&A fashion.

Q1. How efficient is FTerm?

R1. We answer in two parts. First we consider the number of programs of which termination is decided. Then we consider the overall time the analysis takes to perform.

(R1-1) FTerm decides termination of 5 of the 8 programs. These 5 programs only manipulate $\mathbb{F}$-linear expressions: their rational approximations only manipulate linear expressions. This is not the case of the 3 remaining programs which

[11] A library for manipulating rational numbers, http://www.apfloat.org/.

	Without range analysis	With range analysis			Number of variables
	Verdict	Verdict	k	Binterm time	
u3.c	?	?	5	TO	3
u4.c	?	?	5	TO	5
u5.c	?	✓	4	157s	2
u6.c	?	?	5	TO	4
u7.c	?	✓	2	131s	3
u8.c	?	✓	1	63s	4
u9.c	?	✓	1	65s	4
u10.c	?	✓	1	1s	2

FIGURE 6.6: FTerm results on Floops. See Figure 6.3 for the notation. The range analysis is done manually for now: see Section 6.4.1, Automation level (Man1).

manipulate *non*-$\mathbb{F}$-linear expressions: their rational approximations manipulate *non*-linear expressions. Nevertheless FTerm decides termination of more programs of Floops than current state-of-the-art tools.

(R1-2) FTerm is slow: it generally answers within minutes or times out. This is partly due to the fact that the size of the rational approximation exponentially grows with k. However there is another cause that significantly affect FTerm's performance: manipulating big rational numbers can require much space and thus much time. For example using the binary64, the absolute error for the subnormals is bounded by $\mathscr{A}_s = \epsilon\beta^{e_{min}} = \frac{2^{-52}}{2}2^{-1022} =$

1/40480450661462123670499069343783461409911329952828423671380271605486067913599069378392076740287424899037415572863362382277961747477158695373402679988147701984303484855313272272893381548418643268247953535694549013712401496684938539723620671129831911268162011302471753910466682923046100506437265501729201252661541548218698956 8.

Q2. How important is the range analysis?

R2. Range analysis is very important: without it FTerm cannot decide anything. The question that remains and that we leave for future work is: how precise the range analysis should be so that we get satisfactory results? Indeed on one hand, if it is

not precise enough then we might not be able to prove termination. On the other hand, computing very precise ranges might take too much time. See Appendix C for a discussion on range analysis for FP computations.

6.5 Semi-manual proofs

In this section, we investigate a few of the Floops programs of which termination could not be decided. We use Mathematica to assist us. First in subsection 6.5.1, we give a few generalities concerning Mathematica. Then in subsection 6.5.2, we illustrate its use through concrete examples.

6.5.1 The Mathematica software suite

Mathematica is a proprietary Computer Algebra System developed by Wolfram Research. We interact with Mathematica's kernel through the Wolfram Language. Mathematica has a lot of features. Those that are of interest to us and that we used while writing this thesis are as follows:

(Mat1) Support for symbolic computation.
(Mat2) Solvers for systems of inequalities over integers, real numbers and complex numbers. Solvers for recurrence relations. Solvers for optimization problems.
(Mat3) Support for arbitrary precision arithmetic and interval arithmetic.
(Mat4) Tools for data and function vizualization.
(Mat5) A language that is easy to learn, an intuitive user interface, a concise documentation.

The Reduce function. Mathematica's solvers are very handy for theorem proving. In particular, Mathematica provides the Reduce function which can solve polynomial systems, univariate transcendental equations, Diophantine systems and many others. Reduce can also perform existential and universal quantifier elimination. Reduce considers variables to be existentially quantified when not explicited. Reduce and related functions use about 350 pages of Wolfram Language code and 1400 pages of C code[12]: this is a suggestion of their richness.

To illustrate the use of Reduce, suppose we want to decide existence of LRFs for the simple relation $\mathcal{R}_s = \{(x, x') \in \mathbb{R}^2 | x' = x - 1 \wedge x > 0\}$. We have: $\exists a \forall x, x' :$

[12] http://reference.wolfram.com/language/tutorial/SomeNotesOnInternalImplementation.html

$\underbrace{x\mathcal{R}_s x' \implies ax \geq ax' + 1 \wedge ax \geq 0}_{\Psi(x,x',a)}$, or equivalently: $\exists a \neg\exists x, x' : \neg\Psi(x, x', a)$. We check it with Mathematica as follows:

```
? xRx1 := x1 == x - 1 && x > 0
  psi := xRx1 => a*x >= a*x1 + 1 && a*x >= 0
  Reduce[Exists[{x, x1}, !psi], Reals]
> a < 1
```

That is $\exists x, x' : \neg\Psi(x, x', a) \iff a < 1$ or $\forall x, x' : \Psi(x, x', a) \iff a \geq 1$. Thus LRFs exist for $\mathcal{R}_s$ and they are *completely* described by $f(x) = ax$ such that $a \geq 1$.

6.5.2 Floating-point implementations of Newton's method

Let us prove termination of ubuntu11.c and ubuntu12.c which are FP implementations of Newton's method for computing an approximation of the square root of a number. They are shown in Figure 6.7.

```
float d, x = val;
if (val == 0) return 0;
do {
  float y = (x + (float)val/x)/2;
  d = (y > x) ? y - x : x - y;
  x = y;
} while (d >= 0.5);
```

(a) ubuntu11.c

```
double xn, xn1 = x;
if (x == 0.0) return 0.0;
do {
  xn = xn1;
  xn1 = (xn + x/xn) / 2.0;
} while (fabs(xn1 - xn)>1E-10);
```

(b) ubuntu12.c

FIGURE 6.7: Computing an approximation of the square root of a number using FP versions of Newton's method.

We recall that Newton showed that $\sqrt{t} = lim_{n\to\infty} u_n$ where (u_n) is the sequence over real numbers defined as: $u_0 = t \wedge u_{n+1} = \frac{1}{2}\left(u_n + \frac{t}{u_n}\right)$. In practice we end the computation when the desired accuracy is reached. That is when $|u_{n+1} - u_n| < \theta$ for some $\theta > 0$. What ubuntu11.c and ubuntu12.c do is to iteratively compute u_n using FP computations. Call (v_n) the corresponding sequence over FP numbers: $v_0 = t \wedge v_{n+1} = \left(v_n \oplus t \oslash v_n\right) \oslash 2$. Computation terminates when $|v_{n+1} \ominus v_n| < \theta$. In the following, we first show how to prove termination of the iterative computation of (u_n). Then we do so for (v_n).

Termination proof for (u_n). We are interested in proving the cWF-ness of the relation $\mathcal{R}_u = \left\{(u, u') \in \mathbb{R}^2 | u' = \frac{1}{2}\left(u + \frac{t}{u}\right) \wedge |u' - u| \geq \theta\right\}$ for $t > 0$ and $\theta > 0$

given. To that end, we show that $\mathcal{R}_u$ cannot be infinitely applied while satisfying $|u' - u| \geq \theta$. Indeed $|u' - u|$ strictly decreases over successive applications of $\mathcal{R}_u$: $\exists\delta\forall u, u', u'', t, \theta : \underbrace{u\mathcal{R}_u u' \wedge u'\mathcal{R}_u u'' \implies |u' - u| \geq |u'' - u'| + \delta}_{\Psi(u,u',u'',\delta,t,\theta)}, \delta > 0$, or equivalently: $\exists\delta\neg\exists u, u', u'', t, \theta : \neg\Psi(u, u', u'', \delta, t, \theta)$. We check it with Mathematica for $\theta = \frac{1}{2}$ for example, as it is the case for ubuntu11.c:

```
? theta := 1/2
  uRu1 := u1 == (u + t/u)/2 && Abs[u1 - u] >= theta
  u1Ru2 := u2 == (u1 + t/u1)/2 && Abs[u2 - u1] >= theta
  psi := uRu1 && u1Ru2 => Abs[u1 - u] >= Abs[u2 - u1] + delta
  Reduce[Exists[{u, u1, u2, t, theta},
         !psi && t > 0 && delta > 0], Reals]
> delta > 1/2
```

Mathematica says that $\exists u, u', u'', t, \theta : \neg\Psi(u, u', u'', \delta, t, \theta) \iff \delta > \frac{1}{2}$ or $\forall u, u', u'', t, \theta : \Psi(u, u', u'', \delta, t, \theta) \iff \delta \leq \frac{1}{2}$ with $\delta > 0$. That is $|u' - u|$ is decreased of at least $\frac{1}{2}$ after each application of $\mathcal{R}_u$ for any given $t > 0$ and for $\theta = \frac{1}{2}$: $\mathcal{R}_u$ is cWF.

Termination proof for (v_n). Checking termination of the iterative computations of (v_n) is not as straightforward as that of (u_n) due to rounding errors. This is illustrated in Figure 6.8(a), 6.8(b) and 6.8(c). Let us find sufficient conditions for termination. Notably we are interested in finding conditions Φ for which the guard condition is eventually unsatisfied: $\Phi \implies \exists m : \Delta v_m < \theta$ where (Δv_n) is the sequence defined as $\Delta v_n = |v_{n+1} \ominus v_n|$. To that end, let us constrain θ to be such that $L < \theta$ where L is an *eventual upper bound* of Δv_n:

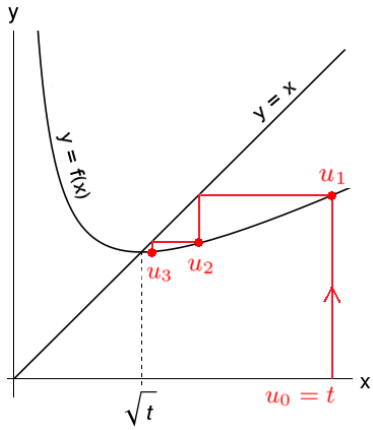
$$\exists m\forall m' : m \leq m' \implies \Delta v_{m'} \leq L \tag{6.1}$$

Indeed if $L < \theta$ then eventually $\Delta v_n < \theta$ since eventually $\Delta v_{m'} \leq L$.

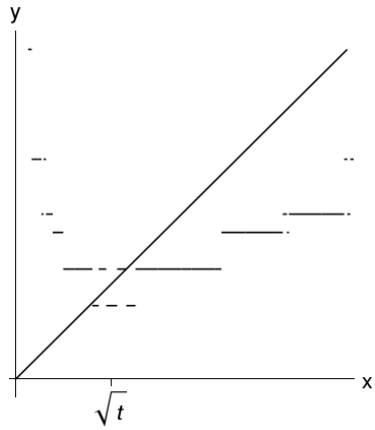
How to determine L? We resort to abstractions: we approximate (Δv_n) by real sequences of which we know how to compute the bounds. In the following, $\mu_?$ and $\nu_?$ respectively denote lower and upper approximation functions that are to be defined depending on the desired precision. First we get rid of the FP subtraction in (Δv_n):

$$\forall n : \Delta v_n \leq \nu_?\big(|v_{n+1} - v_n|\big) \tag{6.2}$$

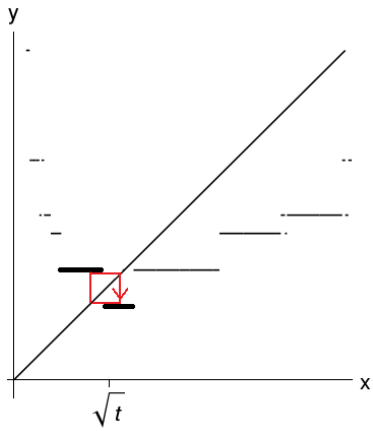
Then we get rid of the FP operations in (v_n) in a similar way: by using rational sequences (w_n^+) and (w_n^-) respectively obtained by upper and lower approximating



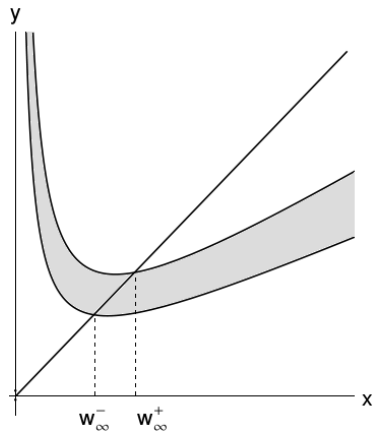
(a) When using real computations, $|u_{n+1} - u_n|$ where $u_{n+1} = f(u_n)$ strictly decreases at each step: termination is always guaranteed.



(b) When using FP computations, the function that defines the sequence becomes discrete and unintuitive.



(c) Can we be sure that situations like this never happen when using FPs?



(d) We abstract the function g, $v_{n+1} = g(v_n)$, by any function $g^{\#}$ that lays in the gray area.

FIGURE 6.8: Studying rational and FP versions of Newton's method for computing square roots

the operations in (v_n).

$$\begin{cases} \forall n : w_n^- \leq v_n \leq w_n^+ \\ w_n^- = \mu_?\Big(\big(\mu_?\big(v_n + \mu_?(t/v_n)\big)\big)/2\Big) \\ w_n^+ = \nu_?\Big(\big(\nu_?\big(v_n + \nu_?(t/v_n)\big)\big)/2\Big) \end{cases} \tag{6.3}$$

as shown in Figure 6.8(d). It follows that $|v_{n+1} - v_n| \leq |w^+_{n+1} - w^-_n|$. Then from Equation 6.2, from monotonicity of $\mu_?$ and $\nu_?$ and by considering the limits:

$$\exists m \forall m' : m \leq m' \implies \Delta v_{m'} \leq \underbrace{\nu_?\big(|w^+_\infty - w^-_\infty|\big)}_{L} \tag{6.4}$$

where $w^+_\infty = lim_{n\to\infty} w^+_n$ and $w^-_\infty = lim_{n\to\infty} w^-_n$. We can show that these limits are finite and that they each are the unique fixpoint of the function that defines the considered sequence. Now we can compute the range of t for which $L < \theta$, for example for ubuntu11.c:

```
? (* Let us use the PAA (mu2,nu2) for example. Definitions in Appendix A. *)
  theta := 1/2
  L := Nu2[Abs[wn - wp]]
  Reduce[
    (* FP division by 2 needs not to be approximated:  see FP Property 2.3 *)
    wn == Mu2[wp + Mu2[t/wp]]/2 &&
    wp == Nu2[wn + Nu2[t/wn]]/2 &&
    L < theta &&
    t > 0 && wp > 0 && wn > 0, {wp, wn}, Reals]
> 0 < t <= #
  (* # represents a big real number.
   * We do not show its exact value for readability.
   * It is such that 7.8e12 <= # < 7.9e12 *)
```

That is ubuntu11.c terminates for any value of *val* in $]0, 7.8 \cdot 10^{12}]$.

We apply the same reasoning to ubuntu12.c. We just need to modify our Mathematica definitions to meet the binary64 format and to set L to 10^{-10}. Mathematica results are such that ubuntu12.c terminates for any value of x in $]0, 9 \cdot 10^{10}]$.

We point out that the use of (μ_2, ν_2) is only an arbitrary choice. It could have been another pair of PAA. Notably we can use Theorem 3.3 to devise tighter pairs such that we will obtain potentially bigger initial ranges that lead to termination.

6.6 Conclusion

To our knowledge, we presented the first test suite that is specifically designed for termination analysis of FP computations. The test suite is built from various programs in the wild: a few of them had termination bugs due to the peculiarity of FP computations. This clearly suggests the need for developping tools that can automatically

prove or disprove their termination. We put current state-of-the-art termination analyzers against our test suite: they were unsuccessful on most of the programs. In contrast to that, the tool we developed gave better results.

Termination of some programs of the test suite could be proved only by using human intelligence while that of other ones still remains unkown. The current status of Floops as studied in this chapter is summarized in Figure 6.9. Successfully analyzing the whole test suite in a fully automatic way, with reasonable amount of memory and time, can be a milestone to reach for all the currently existing tools.

	Verdict	Juggernaut	2LS	FTerm	Semi-manual
ubuntu1.c	✗	✗			
ubuntu2.c	✗	✗			
ubuntu3.c	?				
ubuntu4.c	?				
ubuntu5.c	✓	✓		✓	
ubuntu6.c	?				
ubuntu7.c	✓			✓	
ubuntu8.c	✓			✓	
ubuntu9.c	✓			✓	
ubuntu10.c	✓	✓	✓	✓	
ubuntu11.c	✓				✓
ubuntu12.c	✓				✓

FIGURE 6.9: Floops' status following Figure 6.3, 6.6 and Section 6.5. Note that these results are valid for specific ranges of the variables. Also see Figure 6.4 for a correct interpretation of Juggernaut's verdict.

General Conclusion

We arrive at the end of our exploration of the world of Floating-Point (FP) computations. It is a peculiar world in which the equality $x + 1 = x$ can be true. This is due to the existence of rounding errors that render counter-intuitive the behavior of programs manipulating FP numbers. To prove that these programs actually do what they are supposed to do, we often need to prove among other things that they eventually terminate. This thesis developed techniques to this effect.

First in Chapter 3, we devised an algorithm called PaaTerm that approximates FP computations into the rationals and that gradually increases the tightness of the approximation until termination can be proved. We also devised an algorithm called Opt-ν which produces affine approximations that are computable efficiently and that are optimal in a certain sense. Then in Chapter 4 and 5, we respectively studied the global and local approach for proving termination of FP computations. We studied the use of the well-known Podelski-Rybalchenko (PR) algorithm for the FP case. We also showed in details how to use the Size-Change (SC) approach and its extension to monotonicity constraints to handle FP numbers. Last in Chapter 6, we built a test suite called Floops to show that current termination analyzers are yet to be improved when it comes to FP numbers. We experimentally showed that our techniques contribute to these improvements. For future work, the following ideas can be investigated:

(Fut1) On PaaTerm: instead of simply increasing the tightness when termination cannot be proved, we could also use a counter-example guided approaches [GMR15]. That is we could check whether the non-terminating executions in the abstract effectively appear in the concrete, after which we can decide how to tighten the abstraction. We point out that the core of PaaTerm, an algorithm called Paa which is the one in charge of the tightening, can be used to verify properties other than termination: safety properties for example.

(Fut2) On Opt-ν: this algorithm gives an optimal affine bound for the rounding function. We could investigate how this optimality result could be extended to specific FP computations. First we could start with basic operations such

as FP addition. Then we could continue with combinations or sequences of such basic operations. Also on a more technical note, Opt-ν uses a notion of surface as quality measure: what about other quality measures, for example the one used in the well-known Least Square method? It would have the advantage of punishing the bigs gaps between the rounding function and the affine bound. On a even more technical note, we showed that when tightening, the size of the invariants captured in the abstract *qualitatively* increases in the sense of the inclusion. Can we think about a *quantitative* measure? Notably it would allow to decide whether it is worth increasing the tightness given the new size of the set of invariants we would obtain comparatively to its current size.

(Fut3) Following the use of PR and SC for the FP case, a natural question arises: what about other approaches? For example for the global approach, we could extend the results we got on Affine Ranking Functions (ARF) to Lexicographic ARFs. Then for the local approach, we could investigate how to make the Transition Invariants approach support FP computations.

(Fut4) The future of Floops: the loops in this test suite are relatively simple and yet they challenge current termination analyzers. Once we obtain satisfactory results, we can enrich Floops with slightly more complex loops. Then we can continue so until we will hopefully be able to have a test suite that have loops with hundreds of FP operations, or even more. The Ubuntu distribution have such loops: what guarantee do we have for their termination when even two-lines loops in the ImageMagick package could end in unexpected infinite executions?

(Fut5) Cleverness and dumbness: back in Section 6.5.2, termination of ubuntu11.c could not be proven fully automatically: we had to resort to clever observations. Yet there is a very dumb way to prove it always terminates: execute it for each possible input and the answer will be obtained before time runs out, in less than five minutes. In comparison the brute force of Juggernaut is too complex to answer in the given time. Clearly, combining dumb approaches with advanced ones can be fruitful, how to do that?

By the time this last paragraph was written, the ImageMagick bug that bothered the fictional character Marc from the General Introduction has been fixed: he can now edit his selfies in complete tranquility. Unfortunately we fear that Marc will still

encounter similar bugs in other software. Far from closing the topic, we would like our work to be a call for further development of termination analysis of FP computations.

Conclusion Générale

Nous voici à la fin de notre exploration du monde des calculs flottants, un monde particulier dans lequel l'égalité $x + 1 = x$ peut être vraie. Ceci est dû aux erreurs d'arrondi qui rendent contre-intuitif le comportement des programmes manipulant les nombres flottants. Pour prouver que ces programmes font réellement ce qu'ils sont censés faire, nous devons souvent prouver entre autres choses qu'ils terminent. Cette thèse a développé des techniques à cet effet.

Tout d'abord dans le Chapitre 3, nous avons conçu un algorithme appelé PAATERM qui approxime les calculs flottants dans les rationnels et qui augmente progressivement la précision de l'approximation obtenue jusqu'à ce que la terminaison puisse être prouvée. Nous avons également conçu un algorithme appelé OPT-ν qui produit des approximations affines calculables rapidement et optimales selon un certain sens. Ensuite dans le Chapitre 4 et 5, nous avons respectivement étudié l'approche globale et locale pour prouver la terminaison de calculs flottants. Nous avons conçu une version flottante de l'algorithme bien connu de Podelski-Rybalchenko. Nous avons également montré en détail comment utiliser l'approche Size-Change et son extension aux contraintes de monotonicité pour traiter les nombres flottants. Enfin dans le Chapitre 6, nous avons construit une suite de test appelée Floops pour montrer que les analyseurs de terminaison actuels doivent encore être améliorés en ce qui concerne les calculs flottants. Nous avons montré expérimentalement que nos techniques contribuent à ces améliorations. Nous invitons le lecteur à se référer à la version anglaise de ce paragraphe pour connaitre quelques travaux futurs que nous envisageons.

A l'heure où ce dernier paragraphe a été écrit, le bug d'ImageMagick qui a dérangé le personnage fictif Marc dans l'Introduction Générale a été corrigé : il peut désormais retoucher ses selfies en toute tranquilité. Malheureusement nous craignons que Marc puisse encore rencontrer des bugs similaires dans d'autres logiciels. Loin de clore le sujet, nous souhaiterions que notre travail appelle à un développement ultérieur de l'analyse de terminaison des calculs flottants.

Appendices

A Mathematica definitions

Throughout the thesis, we use Mathematica to assist us. A quick overview of its capabilities is given in Section 6.5.1. Notably we point out that one of its strengths is the intuitiveness of its syntax. For example the rounding function ζ_n as defined in Definition 2.7 is written as follows in Mathematica for the single precision FP type.

```
? b:=2
  p:=24
  eM:=127
  em:=-126 (* 1 - eM *)
  nm:=b^em
  Exponent[x_]:=If[Abs[x]>=nm,Floor[Log[b,Abs[x]]],em]
  Ulp[x_]:=b^(-p+1)*b^Exponent[x]
  ZetaN[x_]:=Round[x,Ulp[x]]
```

Then the k-PAA $(\mu_1, \nu_1), (\mu_2, \nu_2)$ and (μ_3, ν_3) as defined in Property 3.1, 3.2 and 3.3 are written as follows.

```
? Mu1[x_]:=x-asn
  Nu1[x_]:=x+asn
  Mu2[x_]:=Piecewise[{{x(1-rn)-as,0<=x<=nM},{x(1+rn)-as,-nM<=x<0}}]
  Nu2[x_]:=Piecewise[{{x(1+rn)+as,0<=x<=nM},{x(1-rn)+as,-nM<=x<0}}]
  Mu3[x_]:=Piecewise[{{x(1-rn),nm<x<=nM},{x-as,-nm<=x<=nm},{x(1+rn),-nM<=x<-nm}}]
  Nu3[x_]:=Piecewise[{{x(1+rn),nm<x<=nM},{x+as,-nm<=x<=nm},{x(1-rn),-nM<=x<-nm}}]
  assum:=nm>0&&nM>nm&&rn>0&&as>0&&asn>as&&-nM<=x&&x<=nM
```

B Proofs of Theorem 3.1, 3.2 and Lemma 3.1

We use Mathematica to assist us. Definitions in Appendice A are assumed. We define the three quality measures as follows:

```
? S1:=Integrate[Mu1[x]-Nu1[x],{x,-nM,nM},Assumptions->assum]
  S2:=Integrate[Mu2[x]-Nu2[x],{x,-nM,nM},Assumptions->assum]
  S3:=Integrate[Mu3[x]-Nu3[x],{x,-nM,nM},Assumptions->assum]
```

Theorem 3.1 (Part 1/2). $\forall \mathbb{F} : S(\mu_1, \nu_1) > S(\mu_2, \nu_2) \iff e_{max} > e_{min} + p$

Is (μ_2, ν_2) always tighter than (μ_1, ν_1) for any possible FP type $\mathbb{F}$? We ask Mathematica for counter-examples.

```
? Reduce[S2>=S1&&assum]
> assum&&(-2as+2asn)/rn<=nM
```

Mathematica says that $\exists \mathbb{F} : S(\mu_2, \nu_2) \geq S(\mu_1, \nu_1) \iff n_{max} \geq \frac{-2\mathscr{A}_s + 2\mathscr{A}_{sn}}{\mathscr{R}_n}$ or equivalently $\forall \mathbb{F} : S(\mu_2, \nu_2) < S(\mu_1, \nu_1) \iff n_{max} < \frac{-2\mathscr{A}_s + 2\mathscr{A}_{sn}}{\mathscr{R}_n}$. We want to express that condition in terms of the parameters β, p, e_{min} and e_{max}:

$$
\begin{aligned}
S(\mu_2, \nu_2) < S(\mu_1, \nu_1) &\iff (\beta - \beta^{-p+1})\beta^{e_{max}} < \frac{-2\frac{\beta^{-p+1}\beta^{e_{min}}}{2} + 2\frac{\beta^{-p+1}\beta^{e_{max}}}{2}}{\frac{\beta^{-p+1}}{2}} \\
&\iff \beta(\beta^{e_{max}} - \beta^{-p+e_{max}}) < 2(\beta^{e_{max}} - \beta^{e_{min}}) \\
&\iff \beta^{e_{max}} - \beta^{-p+e_{max}} < \beta^{e_{max}} - \beta^{e_{min}} \qquad \text{(since } \beta \geq 2\text{)} \\
&\iff -p + e_{max} > e_{min} \qquad \text{(B.1)}
\end{aligned}
$$

Theorem 3.2 (Part 1/2). $\forall (\mathbb{F}, x) : (\mu_3, \nu_3)$ **is nested in** (μ_2, ν_2)

First we check if $\forall (\mathbb{F}, x) : \mu_2(x) \leq \mu_3(x) \leq \nu_3(x) \leq \nu_2(x)$:

```
? Reduce[!(Mu2[x]<=Mu3[x]<=Nu3[x]<=Nu2[x])&&assum]
> False
```

Then we check if $\exists x : \mu_2(x) < \mu_3(x) \vee \nu_3(x) < \nu_2(x)$:

```
? Reduce[(Mu2[x]<Mu3[x]||Nu3[x]<Nu2[x])&&assum]
> True
```

Theorem 3.2 (Part 2/2). $\exists (\mathbb{F}, x) : (\mu_2, \nu_2)$ **is not nested in** (μ_1, ν_1)

We ask Mathematica for counter-examples.

```
? Reduce[!(Mu1[x]<=Mu2[x]<=Nu2[x]<=Nu1[x])&&assum]
> assum&&(x<(-asn+as)/rn||x>(asn-as)/rn)
```

That is $\exists (\mathbb{F}, x) : \neg\big(\mu_1(x) \leq \mu_2(x) \leq \nu_2(x) \leq \nu_1(x)\big) \iff \big(x < \frac{-\mathscr{A}_{sn} + \mathscr{A}_s}{\mathscr{R}_n} \vee x > \frac{\mathscr{A}_{sn} - \mathscr{A}_s}{\mathscr{R}_n}\big)$. In terms of the parameters $\beta, p, e_{min}, e_{max}$: if $x < -L$ or $x > L$ where $L = \beta^{e_{max}} - \beta^{e_{min}}$ then (μ_2, ν_2) is not nested in (μ_1, ν_1). See Figure 3.1(b) for illustration.

Theorem 3.1 (Part 2/2).: $\forall \mathbb{F} : S(\mu_2, \nu_2) > S(\mu_3, \nu_3)$

Follows from Proposition 3.1 and from the fact that (μ_2, ν_2) is nested in (μ_3, ν_3) by Theorem 3.2.

Lemma 3.1: $\forall(\mathbb{F}, x) : (\mu_{k+1}, \nu_{k+1})$ **obtained through** PAA **is nested in** (μ_k, ν_k) **if the subnormals are isolated in** (μ_k, ν_k)

The problem boils down to proving that on any β-ade B of ULP u, the pair $(x + \frac{u}{2}, x - \frac{u}{2})$ is nested in the pair (μ_3, ν_3). Since these are all affine functions, we just need to check that $\nu_3 \leq x - \frac{u}{2} \leq x + \frac{u}{2} \leq \mu_3$ at the edges of B.

C Range analysis

The need for range analysis is recurrent throughout the thesis.

Theorem C.1. *Consider a program as specified in Definition 1.2. The most precise ranges in which the program variables lay during all possible executions are computable.*

Proof. *Similar to that of Theorem 4.1. Recall the proof of Theorem 1.1. In addition to checking non-repeating states in each partial execution, we also record all the states that were already visited.*

However computing the most precise ranges is extremely space and time consuming in the general cases. Various abstractions can be used in order to get approximate ranges in a reasonable amount of time [Min07][GGP09].

D Ramsey theorem and termination analysis

Termination Characterization 1.2.f directly comes from the original Infinite Ramsey Theorem [Ram30, Theorem A], with a few observations from [BS15, Theorem 2.1]. We show how to retrieve that characterization using a better known version of that theorem.

Lemma D.1 (Ramsey theorem in terms of graph coloring). *Consider a complete simple graph (undirected, with no loop nor multiple edges) over an infinite number of nodes. Then for any possible edge coloring, there is always an infinite set of nodes such that all edges between these nodes have the same color.*

Proof. *See [Mis12] for a short proof.*

Termination Characterization 1.2.f. *Consider the program $\mathcal{P} = (\mathcal{S}, \mathcal{S}, \mathcal{R})$. It terminates* if and only if $\mathcal{R}^+$ *is disjunctively cWF.*

Proof. *For the $\Rightarrow$-direction, derived from (the trivial) Termination Characterization 1.2.b-e. For the $\Leftarrow$-direction, reason by contradiction. Suppose there is an infinite execution $S_1S_2S_3\ldots$ of $\mathcal{P}$ but that $\mathcal{R}^+$ is disjunctively cWF: $\mathcal{R}^+ = \bigcup \mathcal{T}_k$ such that each $\mathcal{T}_k$ is cWF. Then color each edge $\{i, j\}, i < j$, with a $\mathcal{T}_k$ such that $S_i\mathcal{T}_kS_j$. This is illustrated in Figure D.1. Now by Lemma D.1 there is an infinite complete subgraph that has all of its edges colored by some same $\mathcal{T}_k$: that particular $\mathcal{T}_k$ is* not *cWF, contradiction.*

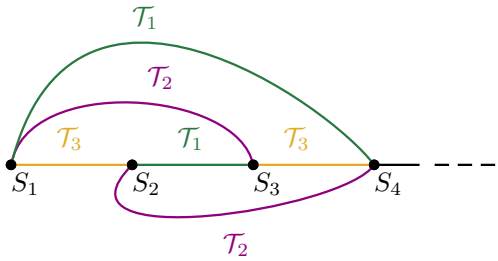


FIGURE D.1: Coloring the edges of a complete graph

A quantitative version of Termination Characterization 1.2.f can be found in [Del16, Lemma 3.4, Point (3)].

E More details for the proof of Theorem 3.6

(§1). To be precise, an ULP-increasing endpoints L_{inc} is characterized as follows: $L_{inc} = \big((-1)^s(1+\epsilon)\beta^e, (-1)^s\beta^e\big)$.

(§2, Case (a)). Similar to Property 2.7 in the sense that subnormals all have the same ULP.

(§2, Case (b)). First, ULP-increasing endpoints are all located on ν_3 of Property 3.1. Then ν_3 is convex.

(§2, Case (c)). The endpoints of abscissa between that of P_1 and P_2, P_2 not included, all have the same ULP $u = 2\epsilon\beta^e$. Thus they are placed under the line $(\mathcal{D})$ of equation $y(x) = x + \frac{u}{2} = x + \epsilon\beta^e$. Now when including P_2, it remains to show that (P_1P_2) is above $(\mathcal{D})$.

(§5, Claim 1). The piecewise function that corresponds to the concatenation of $\overline{P_{min}P_i}, \overline{P_iP_j}$ and $\overline{P_jP_{max}}$ is concave.

(§5, Claim 2). The objective function S from Equation 3.3 can be also written: $S = (x_{max} - x_{min})\left(\frac{x_{max}+x_{min}}{2}a + b\right) = (x_{max} - x_{min})\mu\left(\frac{x_{max}+x_{min}}{2}\right)$. That is we just need to minimize μ at the midpoint of I.

Bibliography

[Ali+10] Christophe Alias et al. “Multi-dimensional Rankings, Program Termination and Complexity Bounds of Flowchart Programs”. In: *Proc. of the 17th International Static Analysis Symposium (SAS'10)*. Ed. by Radhia Cousot and Matthieu Martel. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, pp. 117–133. ISBN: 978-3-642-15768-4. DOI: 10.1007/978-3-642-15769-1_8. URL: http://dx.doi.org/10.1007/978-3-642-15769-1_8.

[Bag+12] Roberto Bagnara et al. “A New Look at the Automatic Synthesis of Linear Ranking Functions”. In: *Information and Computation* 215 (2012), pp. 47–67. DOI: 10.1016/j.ic.2012.03.003. URL: http://dx.doi.org/10.1016/j.ic.2012.03.003.

[Ben09] Amir M. Ben-Amram. “Size-Change Termination, Monotonicity Constraints and Ranking Functions”. In: *Proc. of the 21st International Conference on Computer Aided Verification (CAV'09)*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 109–123. ISBN: 978-3-642-02657-7. DOI: 10.1007/978-3-642-02658-4_12. URL: https://doi.org/10.1007/978-3-642-02658-4_12.

[Ben11] Amir M. Ben-Amram. “Monotonicity Constraints for Termination in the Integer Domain”. In: *Logical Methods in Computer Science* 7.3 (2011). DOI: 10.2168/LMCS-7(3:4)2011. URL: https://doi.org/10.2168/LMCS-7(3:4)2011.

[Ben13] Amir M. Ben-Amram. “Ranking Functions for Linear-Constraint Loops”. In: *Proc. of the 1st International Workshop on Verification and Program Transformation (VPT'13)*. Ed. by Alexei Lisitsa and Andrei P. Nemytykh. Vol. 16. EPiC Series. EasyChair, 2013, pp. 1–8. URL: http://www.easychair.org/publications/?page=859286511.

[Ben15] Amir M. Ben-Amram. “Mortality of Iterated Piecewise Affine Functions over the Integers: Decidability and complexity”. In: *Computability* 4.1 (2015), pp. 19–56. DOI: 10.3233/COM-150032. URL: https://doi.org/10.3233/COM-150032.

[BG14] Amir M. Ben-Amram and Samir Genaim. "Ranking Functions for Linear-Constraint Loops". In: *Journal of the ACM* 61.4 (2014), 26:1–26:55. DOI: 10.1145/2629488. URL: http://doi.acm.org/10.1145/2629488.

[BG15] Amir M. Ben-Amram and Samir Genaim. "Complexity of Bradley-Manna-Sipma Lexicographic Ranking Functions". In: *Proc. of the 27th International Conference on Computer Aided Verification (CAV'15)*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. Lecture Notes in Computer Science. Springer, 2015, pp. 304–321. ISBN: 978-3-319-21667-6. DOI: 10.1007/978-3-319-21668-3_18. URL: http://dx.doi.org/10.1007/978-3-319-21668-3_18.

[BG17] Amir M. Ben-Amram and Samir Genaim. "On Multiphase-Linear Ranking Functions". In: *Proc. of the 29th International Conference on Computer Aided Verification (CAV'17), Part II*. ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 601–620. ISBN: 978-3-319-63389-3. DOI: 10.1007/978-3-319-63390-9_32. URL: https://doi.org/10.1007/978-3-319-63390-9_32.

[BGM12] Amir M. Ben-Amram, Samir Genaim, and Abu N. Masud. "On the Termination of Integer Loops". In: *ACM Transactions on Programming Languages and Systems* 34.4 (2012), 16:1–16:24. DOI: 10.1145/2400676.2400679. URL: http://doi.acm.org/10.1145/2400676.2400679.

[BIK14] Marius Bozga, Radu Iosif, and Filip Konecný. "Deciding Conditional Termination". In: *Logical Methods in Computer Science* 10.3 (2014). DOI: 10.2168/LMCS-10(3:8)2014. URL: http://dx.doi.org/10.2168/LMCS-10(3:8)2014.

[BJ06] Thomas Ball and Robert B. Jones, eds. *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006. ISBN: 3-540-37406-X.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[BL07] Amir M. Ben-Amram and Chin Soon Lee. "Program Termination Analysis in Polynomial Time". In: *ACM Transactions on Programming Languages and Systems* 29.1 (2007), 5:1–5:37. DOI: 10.1145/1180475.1180480. URL: http://doi.acm.org/10.1145/1180475.1180480.

[BM09] Ahmed Bouajjani and Oded Maler, eds. *Proc. of the 21st International Conference on Computer Aided Verification (CAV'09)*. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009. ISBN: 978-3-642-02657-7. DOI: 10.1007/978-3-642-02658-4. URL: https://doi.org/10.1007/978-3-642-02658-4.

[BM13] Roberto Bagnara and Frédéric Mesnard. "Eventual Linear Ranking Functions". In: *Proc. of the 15th International Symposium on Principles and Practice of*

Declarative Programming (PPDP'13). Ed. by Ricardo Peña and Tom Schrijvers. ACM, 2013, pp. 229–238. ISBN: 978-1-4503-2154-9. DOI: 10.1145/2505879.2505884. URL: http://doi.acm.org/10.1145/2505879.2505884.

[BMR12] Mohammed S. Belaid, Claude Michel, and Michel Rueher. "Boosting Local Consistency Algorithms over Floating-Point Numbers". In: *Proc. of the 18th International Conference on Principles and Practice of Constraint Programming (CP'12)*. Ed. by Michela Milano. Vol. 7514. Lecture Notes in Computer Science. Springer, 2012, pp. 127–140. ISBN: 978-3-642-33557-0. DOI: 10.1007/978-3-642-33558-7_12. URL: http://dx.doi.org/10.1007/978-3-642-33558-7_12.

[BMS05a] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. "Linear Ranking with Reachability". In: *Proc. of the 17th International Conference on Computer Aided Verification (CAV'05)*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 491–504. ISBN: 3-540-27231-3. DOI: 10.1007/11513988_48. URL: http://dx.doi.org/10.1007/11513988_48.

[BMS05b] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. "Termination of Polynomial Programs". In: *Proc. of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*. Ed. by Radhia Cousot. Vol. 3385. Lecture Notes in Computer Science. Springer, 2005, pp. 113–129. ISBN: 3-540-24297-X. DOI: 10.1007/978-3-540-30579-8_8. URL: http://dx.doi.org/10.1007/978-3-540-30579-8_8.

[BN10] Sylvie Boldo and Thi M. T. Nguyen. "Hardware-Independent Proofs of Numerical Programs". In: *Proc. of the 2nd NASA Formal Methods Symposium (NFM'10)*. Ed. by César A. Muñoz. Vol. NASA/CP-2010-216215. NASA Conference Proceedings. 2010, pp. 14–23.

[BN11] Sylvie Boldo and Thi M. T. Nguyen. "Proofs of Numerical Programs when the Compiler Optimizes". In: *Innovations in Systems and Software Engineering* 7.2 (2011), pp. 151–160. DOI: 10.1007/s11334-011-0151-6. URL: http://dx.doi.org/10.1007/s11334-011-0151-6.

[Bol15] Sylvie Boldo. "Stupid is as Stupid Does: Taking the Square Root of the Square of a Floating-Point Number". In: *Electronic Notes in Theoretical Computer Science* 317 (2015), pp. 27–32. DOI: 10.1016/j.entcs.2015.10.004. URL: http://dx.doi.org/10.1016/j.entcs.2015.10.004.

[Bra06] Mark Braverman. "Termination of Integer Linear Programs". In: *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer

Science. Springer, 2006, pp. 372–385. ISBN: 3-540-37406-X. DOI: 10.1007/11817963_34. URL: http://dx.doi.org/10.1007/11817963_34.

[BS15] Stefano Berardi and Silvia Steila. "An Intuitionistic Version of Ramsey's Theorem and its use in Program Termination". In: *Annals of Pure and Applied Logic* 166.12 (2015), pp. 1382 –1406. ISSN: 0168-0072. DOI: http://dx.doi.org/10.1016/j.apal.2015.08.002. URL: http://www.sciencedirect.com/science/article/pii/S0168007215000731.

[CC10] Patrick Cousot and Radhia Cousot. "A Gentle Introduction to Formal Verification of Computer Systems by Abstract Interpretation". In: *Logics and Languages for Reliability and Security*. Ed. by Javier Esparza, Bernd Spanfelner, and Orna Grumberg. Vol. 25. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2010, pp. 1–29. ISBN: 978-1-60750-099-5. DOI: 10.3233/978-1-60750-100-8-1. URL: http://dx.doi.org/10.3233/978-1-60750-100-8-1.

[CC12] Patrick Cousot and Radhia Cousot. "An Abstract Interpretation Framework for Termination". In: *Proc. of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 245–258. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103687. URL: http://doi.acm.org/10.1145/2103656.2103687.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proc. of the 4th ACM Symposium on Principles of Programming Languages (POPL'77)*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973.

[Cer81] Paul E. Ceruzzi. "The Early Computers of Konrad Zuse, 1935 to 1945". In: *IEEE Annals of the History of Computing* 3.3 (1981), pp. 241–262. DOI: 10.1109/MAHC.1981.10034. URL: https://doi.org/10.1109/MAHC.1981.10034.

[CFM15] Hongyi Chen, Shaked Flur, and Supratik Mukhopadhyay. "Termination Proofs for Linear Simple Loops". In: *Software Tools for Technology Transfer* 17.1 (2015), pp. 47–57. DOI: 10.1007/s10009-013-0288-8. URL: http://dx.doi.org/10.1007/s10009-013-0288-8.

[Chu36] Alonzo Church. "A Note on the Entscheidungsproblem". In: *The Journal of Symbolic Logic* 1.01 (1936), pp. 40–41.

[CLS05] Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. "Testing for Termination with Monotonicity Constraints". In: *Proc. of the 21st International Conference on Logic Programming (ICLP'05)*. Ed. by Maurizio Gabbrielli

and Gopal Gupta. Vol. 3668. Lecture Notes in Computer Science. Springer, 2005, pp. 326–340. ISBN: 3-540-29208-X. DOI: 10.1007/11562931_25. URL: http://dx.doi.org/10.1007/11562931_25.

[CM10] Radhia Cousot and Matthieu Martel, eds. *Proc. of the 17th International Static Analysis Symposium (SAS'10)*. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-15768-4. DOI: 10.1007/978-3-642-15769-1. URL: http://dx.doi.org/10.1007/978-3-642-15769-1.

[Coo+11] Byron Cook et al. "Proving Stabilization of Biological Systems". In: *Proc. of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 134–149. ISBN: 978-3-642-18275-4. DOI: 10.1007/978-3-642-18275-4_11. URL: http://dx.doi.org/10.1007/978-3-642-18275-4_11.

[Coo+13] Byron Cook et al. "Ranking Function Synthesis for Bit-vector Relations". In: *Formal Methods in System Design* 43.1 (2013), pp. 93–120. DOI: 10.1007/s10703-013-0186-4. URL: http://dx.doi.org/10.1007/s10703-013-0186-4.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. "Terminator: Beyond Safety". In: *Proc. of the 18th International Conference on Computer Aided Verification (CAV'06)*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 415–418. ISBN: 3-540-37406-X. DOI: 10.1007/11817963_37. URL: http://dx.doi.org/10.1007/11817963_37.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. "Proving Program Termination". In: *Communications of the ACM* 54.5 (2011), pp. 88–98. DOI: 10.1145/1941487.1941509. URL: http://doi.acm.org/10.1145/1941487.1941509.

[CSZ13] Byron Cook, Abigail See, and Florian Zuleger. "Ramsey vs. Lexicographic Termination Proving". In: *Proc. of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Ed. by Nir Piterman and Scott A. Smolka. Vol. 7795. Lecture Notes in Computer Science. Springer, 2013, pp. 47–61. ISBN: 978-3-642-36741-0. DOI: 10.1007/978-3-642-36742-7_4. URL: http://dx.doi.org/10.1007/978-3-642-36742-7_4.

[Cyt+91] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Transactions on Programming Languages and Systems* 13.4 (1991), pp. 451–490. DOI: 10.1145/115372.115320. URL: http://doi.acm.org/10.1145/115372.115320.

[Del16] Christian Delhommé. *Decomposition of Tree-Automatic Structures.* Tech. rep. 2016.

[DKL15] Cristina David, Daniel Kroening, and Matt Lewis. "Unrestricted Termination and Non-termination Arguments for Bit-Vector Programs". In: *Proc. of the 24th European Symposium on Programming (ESOP'15).* Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 183–204. ISBN: 978-3-662-46668-1. DOI: 10.1007/978-3-662-46669-8_8. URL: http://dx.doi.org/10.1007/978-3-662-46669-8_8.

[Fou+07] Laurent Fousse et al. "MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding". In: *ACM Transactions on Mathematical Software* 33.2 (2007), p. 13. DOI: 10.1145/1236463.1236468. URL: http://doi.acm.org/10.1145/1236463.1236468.

[GGP09] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. "The Zonotope Abstract Domain Taylor1+". In: *Proc. of the 21st International Conference on Computer Aided Verification (CAV'09).* Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 627–633. ISBN: 978-3-642-02657-7. DOI: 10.1007/978-3-642-02658-4_47. URL: https://doi.org/10.1007/978-3-642-02658-4_47.

[Gie+04] Jürgen Giesl et al. "Automated Termination Proofs with AProVE". in: *Proc. of the 15th International Conference on Rewriting Techniques and Applications (RTA'04).* Ed. by Vincent van Oostrom. Vol. 3091. Lecture Notes in Computer Science. Springer, 2004, pp. 210–220. ISBN: 3-540-22153-0. DOI: 10.1007/978-3-540-25979-4_15. URL: http://dx.doi.org/10.1007/978-3-540-25979-4_15.

[Gie+15] Jürgen Giesl et al. "Termination Competition (TermComp'15)". In: *Proc. of the 25th International Conference on Automated Deduction (CADE'15).* Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015, pp. 105–108. ISBN: 978-3-319-21400-9. DOI: 10.1007/978-3-319-21401-6_6. URL: https://doi.org/10.1007/978-3-319-21401-6_6.

[Gie+17] Jürgen Giesl et al. "Analyzing Program Termination and Complexity Automatically with AProVE". in: *Journal of Automated Reasoning* 58.1 (2017), pp. 3–31. DOI: 10.1007/s10817-016-9388-y. URL: https://doi.org/10.1007/s10817-016-9388-y.

[GLM15] Stef Graillat, Vincent Lefèvre, and Jean-Michel Muller. "On the Maximum Relative Error when Computing Integer Powers by Iterated Multiplications in Floating-point Arithmetic". In: *Numerical Algorithms* 70.3 (2015), pp. 653–

667. ISSN: 1572-9265. DOI: 10.1007/s11075-015-9967-8. URL: http://dx.doi.org/10.1007/s11075-015-9967-8.

[GMR15] Laure Gonnord, David Monniaux, and Gabriel Radanne. "Synthesis of Ranking Functions using Extremal Counterexamples". In: *Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 608–618. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737976. URL: http://doi.acm.org/10.1145/2737924.2737976.

[Gol91] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". In: *ACM Computing Surveys* 23.1 (1991), pp. 5–48. DOI: 10.1145/103162.103163. URL: http://doi.acm.org/10.1145/103162.103163.

[Hau96] John R. Hauser. "Handling Floating-Point Exceptions in Numeric Programs". In: *ACM Transactions on Programming Languages and Systems* 18.2 (1996), pp. 139–174. DOI: 10.1145/227699.227701. URL: http://doi.acm.org/10.1145/227699.227701.

[HJP10] Matthias Heizmann, Neil D. Jones, and Andreas Podelski. "Size-Change Termination and Transition Invariants". In: *Proc. of the 17th International Static Analysis Symposium (SAS'10)*. Ed. by Radhia Cousot and Matthieu Martel. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, pp. 22–50. ISBN: 978-3-642-15768-4. DOI: 10.1007/978-3-642-15769-1_4. URL: http://dx.doi.org/10.1007/978-3-642-15769-1_4.

[Ieea] "IEEE Standard for Binary Floating-Point Arithmetic". In: *ANSI/IEEE Standard 754-1985* (1985), 0_1–. DOI: 10.1109/IEEESTD.1985.82928.

[Ieeb] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Standard 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[Jon97] Neil D. Jones. *Computability and Complexity - From a Programming Perspective*. Foundations of computing series. MIT Press, 1997. ISBN: 978-0-262-10064-9.

[JR16] Claude-Pierre Jeannerod and Siegfried M. Rump. "On Relative Errors of Floating-point Operations: Optimal Bounds and Applications". In: *Mathematics of Computation* (2016). DOI: 10.1090/mcom/3234. URL: https://hal.inria.fr/hal-00934443.

[JW04] Guy L. Steele Jr. and Jon L. White. "How to Print Floating-point Numbers Accurately (with Retrospective)". In: *A Selection from 20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999*. Ed. by Kathryn S. McKinley. ACM, 2004. ISBN: 1-58113-623-4.

[Kah02] William Kahan. *Idempotent Binary–>Decimal–>Binary Conversion*. Tech. rep. 2002. URL: https://people.eecs.berkeley.edu/~wkahan/Math128/BinDecBin.pdf.

[Kah81] William Kahan. *Why do we Need a Floating-point Arithmetic Standard*. Tech. rep. 1981. URL: https://people.eecs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf.

[Kha80] Leonid G. Khachiyan. "Polynomial Algorithms in Linear Programming". In: *USSR Computational Mathematics and Mathematical Physics* 20.1 (1980), pp. 53 –72. ISSN: 0041-5553. DOI: 10.1016/0041-5553(80)90061-0. URL: http://www.sciencedirect.com/science/article/pii/0041555380900610.

[KL14] Daniel Kroening and Matt Lewis. "Second-Order SAT Solving using Program Synthesis". In: *Computing Research Repository* abs/1409.4925 (2014). URL: http://arxiv.org/abs/1409.4925.

[Lag11] Jeffrey C. Lagarias. *The Ultimate Challenge: the* $3x + 1$ *Problem*. American Mathematical Society, 2011. ISBN: 978-0821849408.

[Ler09] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Communications of the ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814. URL: http://doi.acm.org/10.1145/1538788.1538814.

[LH15] Jan Leike and Matthias Heizmann. "Ranking Templates for Linear Loops". In: *Logical Methods in Computer Science* 11.1 (2015). DOI: 10.2168/LMCS-11(1:16)2015. URL: https://doi.org/10.2168/LMCS-11(1:16)2015.

[LJB01] Chin S. Lee, Neil D. Jones, and Amir M. Ben-Amram. "The Size-change Principle for Program Termination". In: *Proc. of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. Ed. by Chris Hankin and Dave Schmidt. ACM, 2001, pp. 81–92. ISBN: 1-58113-336-7. DOI: 10.1145/360204.360210. URL: http://doi.acm.org/10.1145/360204.360210.

[LM08] Salvador Lucas and José Meseguer. "Termination of Just/Fair Computations in Term Rewriting". In: *Information and Computation* 206.5 (2008), pp. 652–675. DOI: 10.1016/j.ic.2007.11.002. URL: https://doi.org/10.1016/j.ic.2007.11.002.

[LSS97] Naomi Lindenstrauss, Yehoshua Sagiv, and Alexander Serebrenik. "TermiLog: A System for Checking Termination of Queries to Logic Programs". In: *Proc. of the 9th International Conference on Computer Aided Verification (CAV'97)*. Ed. by Orna Grumberg. Vol. 1254. Lecture Notes in Computer Science. Springer, 1997, pp. 444–447. ISBN: 3-540-63166-6. DOI: 10.1007/3-540-63166-6_44. URL: http://dx.doi.org/10.1007/3-540-63166-6_44.

[Luc05] Salvador Lucas. "Polynomials Over the Reals in Proofs of Termination: From Theory to Practice". In: *Theoretical Informatics and Applications* 39.3 (2005), pp. 547–586. DOI: 10.1051/ita:2005029. URL: https://doi.org/10.1051/ita:2005029.

[LZF16] Yi Li, Guang Zhu, and Yong Feng. "The L-Depth Eventual Linear Ranking Functions for Single-Path Linear Constraint Loops". In: *Proc. of the 10th International Symposium on Theoretical Aspects of Software Engineering (TASE'16)*. IEEE Computer Society, 2016, pp. 30–37. ISBN: 978-1-5090-1764-5. DOI: 10.1109/TASE.2016.8. URL: http://dx.doi.org/10.1109/TASE.2016.8.

[Mat68] David W. Matula. "In-and-out Conversions". In: *Communications of the ACM* 11.1 (1968), pp. 47–50. DOI: 10.1145/362851.362887. URL: http://doi.acm.org/10.1145/362851.362887.

[Min07] Antoine Miné. "Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors". In: *Computing Research Repository* abs/cs/0703-077 (2007). URL: http://arxiv.org/abs/cs/0703077.

[Mis12] Jayadev Misra. *A Proof of Infinite Ramsey Theorem*. Tech. rep. 2012. URL: http://www.cs.utexas.edu/users/misra/Notes.dir/Ramsey.pdf.

[MMP16a] Fonenantsoa Maurica, Frédéric Mesnard, and Étienne Payet. "On the Linear Ranking Problem for Simple Floating-Point Loops". In: *Proc. of the 23rd International Static Analysis Symposium (SAS'16)*. Ed. by Xavier Rival. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016, pp. 300–316. ISBN: 978-3-662-53412-0. DOI: 10.1007/978-3-662-53413-7_15. URL: http://dx.doi.org/10.1007/978-3-662-53413-7_15.

[MMP16b] Fonenantsoa Maurica, Frédéric Mesnard, and Étienne Payet. "Termination Analysis of Floating-Point Programs using Parameterizable Rational Approximations". In: *Proc. of the 31st ACM Symposium on Applied Computing (SAC'16)*. Ed. by Sascha Ossowski. ACM, 2016, pp. 1674–1679. ISBN: 978-1-4503-3739-7. DOI: 10.1145/2851613.2851834. URL: http://doi.acm.org/10.1145/2851613.2851834.

[MMP17] Fonenantsoa Maurica, Frédéric Mesnard, and Étienne Payet. "Optimal Approximation for Efficient Termination Analysis of Floating-Point Loops". In: *Proc. of the 1st IEEE Conference on on Next Generation Computing Applications (NextComp'17)*. 2017.

[MNR03] Jean-Michel Muller, Jean-Louis Nicolas, and Xavier-François Roblot. *Number of Solutions to* $(A^2 + B^2 = C^2 + C)$ *in a Binade*. Tech. rep. 2003. URL: https://hal.inria.fr/inria-00071634.

[Mon08] David Monniaux. "The Pitfalls of Verifying Floating-Point Computations". In: *ACM Transactions on Programming Languages and Systems* 30.3 (2008), 12:1–12:41. DOI: 10.1145/1353445.1353446. URL: http://doi.acm.org/10.1145/1353445.1353446.

[MR03] Frédéric Mesnard and Salvatore Ruggieri. "On Proving Left Termination of Constraint Logic Programs". In: *ACM Transactions on Computational Logic* 4.2 (2003), pp. 207–259. DOI: 10.1145/635499.635503. URL: http://doi.acm.org/10.1145/635499.635503.

[Mul+10] Jean-Michel Muller et al. *Handbook of Floating-Point Arithmetic*. Birk-häuser, 2010. ISBN: 978-0-8176-4704-9. DOI: 10.1007/978-0-8176-4705-6. URL: http://dx.doi.org/10.1007/978-0-8176-4705-6.

[Mul05] Jean-Michel Muller. *On the Definition of ulp(x)*. Tech. rep. RR-5504. INRIA, Feb. 2005, p. 16. URL: https://hal.inria.fr/inria-00070503.

[Pay08] Étienne Payet. "Loop Detection in Term Rewriting using the Eliminating Unfoldings". In: *Theoretical Computer Science* 403.2-3 (2008), pp. 307–327. DOI: 10.1016/j.tcs.2008.05.013. URL: http://dx.doi.org/10.1016/j.tcs.2008.05.013.

[PM06] Étienne Payet and Frédéric Mesnard. "Nontermination Inference of Logic Programs". In: *ACM Transactions on Programming Languages and Systems* 28.2 (2006), pp. 256–289. ISSN: 0164-0925. DOI: 10.1145/1119479.1119481. URL: http://doi.acm.org/10.1145/1119479.1119481.

[PR04a] Andreas Podelski and Andrey Rybalchenko. "A Complete Method for the Synthesis of Linear Ranking Functions". In: *Proc. of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*. Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, pp. 239–251. ISBN: 3-540-20803-8. DOI: 10.1007/978-3-540-24622-0_20. URL: http://dx.doi.org/10.1007/978-3-540-24622-0_20.

[PR04b] Andreas Podelski and Andrey Rybalchenko. "Transition Invariants". In: *Proc. of the 19th IEEE Symposium on Logic in Computer Science (LICS' 04)*. IEEE Computer Society, 2004, pp. 32–41. ISBN: 0-7695-2192-4. DOI: 10.1109/LICS.2004.1319598. URL: http://dx.doi.org/10.1109/LICS.2004.1319598.

[PR07] Andreas Podelski and Andrey Rybalchenko. "Transition Predicate Abstraction and Fair Termination". In: *ACM Transactions on Programming Languages and Systems* 29.3 (2007). DOI: 10.1145/1232420.1232422. URL: http://doi.acm.org/10.1145/1232420.1232422.

[PR11] Andreas Podelski and Andrey Rybalchenko. "Transition Invariants and Transition Predicate Abstraction for Program Termination". In: *Proc. of 17th*

International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11). Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Vol. 6605. Lecture Notes in Computer Science. Springer, 2011, pp. 3–10. ISBN: 978-3-642-19834-2. DOI: 10.1007/978-3-642-19835-9_2. URL: http://dx.doi.org/10.1007/978-3-642-19835-9_2.

[PS09] Étienne Payet and Fausto Spoto. "Experiments with Non-Termination Analysis for Java Bytecode". In: *Electronic Notes in Theoretical Computer Science* 253.5 (2009), pp. 83–96. DOI: 10.1016/j.entcs.2009.11.016. URL: http://dx.doi.org/10.1016/j.entcs.2009.11.016.

[Ram30] Frank P. Ramsey. "On a Problem of Formal Logic". In: *Proc. of the London Mathematical Society* s2-30.1 (1930), pp. 264–286. DOI: 10.1112/plms/s2-30.1.264. URL: http://dx.doi.org/10.1112/plms/s2-30.1.264.

[Riv16] Xavier Rival, ed. *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016. ISBN: 978-3-662-53412-0. DOI: 10.1007/978-3-662-53413-7. URL: http://dx.doi.org/10.1007/978-3-662-53413-7.

[SD94] Danny De Schreye and Stefaan Decorte. "Termination of Logic Programs: The Never-Ending Story". In: *Journal of Logic Programming* 19/20 (1994), pp. 199–260. DOI: 10.1016/0743-1066(94)90027-2. URL: http://dx.doi.org/10.1016/0743-1066(94)90027-2.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997. ISBN: 978-0-534-94728-6.

[SK16] Peter Schrammel and Daniel Kroening. "2LS for Program Analysis - (Competition Contribution)". In: *Proc. of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 905–907. ISBN: 978-3-662-49673-2. DOI: 10.1007/978-3-662-49674-9_56. URL: https://doi.org/10.1007/978-3-662-49674-9_56.

[SMP10] Fausto Spoto, Frédéric Mesnard, and Étienne Payet. "A Termination Analyzer for Java Bytecode Based on Path-length". In: *ACM Transactions on Programming Languages and Systems* 32.3 (2010). DOI: 10.1145/1709093.1709095. URL: http://doi.acm.org/10.1145/1709093.1709095.

[Spo16] Fausto Spoto. "The Julia Static Analyzer for Java". In: *Proc. of the 23rd International Static Analysis Symposium (SAS'16)*. Ed. by Xavier Rival. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016, pp. 39–

57. ISBN: 978-3-662-53412-0. DOI: 10.1007/978-3-662-53413-7_3. URL: https://doi.org/10.1007/978-3-662-53413-7_3.

[SS05] Alexander Serebrenik and Danny De Schreye. "Termination of Floating-Point Computations". In: *Journal of Automated Reasoning* 34.2 (2005), pp. 141–177. DOI: 10.1007/s10817-005-6546-z. URL: http://dx.doi.org/10.1007/s10817-005-6546-z.

[Ste74] Pat H. Sterbenz. *Floating-Point Computation*. Prentice-Hall series in automatic computation. Prentice-Hall, 1974. URL: https://cds.cern.ch/record/268412.

[Str65] Christopher Strachey. "An Impossible Program". In: *The Computer Journal* 7.4 (1965), p. 313. DOI: 10.1093/comjnl/7.4.313. URL: http://dx.doi.org/10.1093/comjnl/7.4.313.

[Tiw04] Ashish Tiwari. "Termination of Linear Programs". In: *Proc. of the 16th International Conference on Computer Aided Verification (CAV'04)*. Ed. by Rajeev Alur and Doron A. Peled. Vol. 3114. Lecture Notes in Computer Science. Springer, 2004, pp. 70–82. ISBN: 3-540-22342-8. DOI: 10.1007/978-3-540-27813-9_6. URL: http://dx.doi.org/10.1007/978-3-540-27813-9_6.

[Tur37] Alan Mathison Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical society* 2.1 (1937), pp. 230–265.

[Urb15] Caterina Urban. "Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs." PhD thesis. École Normale Supérieure, Paris, 2015. URL: https://tel.archives-ouvertes.fr/tel-01176641.

[ZLW16] Guang Zhu, Yi Li, and Wenyuan Wu. "Eventual Linear Ranking Functions for Multi-path Linear Loops". In: *Proc. of the 2016 IEEE Conference on Information Technology, Networking, Electronic and Automation Control (ITNEC'16)*. 2016, pp. 331–337. DOI: 10.1109/ITNEC.2016.7560376. URL: http://ieeexplore.ieee.org/document/7560376/.

POLE RECHERCHE
Ecoles **D**octorales

LETTRE D'ENGAGEMENT DE NON-PLAGIAT

Je, soussigné(e) Fonenantsoa MAURICA ANDRIANAMPOIZINIMARO,
en ma qualité de doctorant(e) de l'Université de La Réunion, déclare être conscient(e) que le plagiat est un acte délictueux passible de sanctions disciplinaires. Aussi, dans le respect de la propriété intellectuelle et du droit d'auteur, je m'engage à systématiquement citer mes sources, quelle qu'en soit la forme (textes, images, audiovisuel, internet), dans le cadre de la rédaction de ma thèse et de toute autre production scientifique, sachant que l'établissement est susceptible de soumettre le texte de ma thèse à un logiciel anti-plagiat.

Fait à Sainte-Clotilde le : 2/11/2017

Signature :

Extrait du Règlement intérieur de l'Université de La Réunion
(validé par le Conseil d'Administration en date du 11 décembre 2014)

Article 9. Protection de la propriété intellectuelle – Faux et usage de faux, contrefaçon, plagiat

L'utilisation des ressources informatiques de l'Université implique le respect de ses droits de propriété intellectuelle ainsi que ceux de ses partenaires et plus généralement, de tous tiers titulaires de tels droits.
En conséquence, chaque utilisateur doit :
- utiliser les logiciels dans les conditions de licences souscrites ;
- ne pas reproduire, copier, diffuser, modifier ou utiliser des logiciels, bases de données, pages Web, textes, images, photographies ou autres créations protégées par le droit d'auteur ou un droit privatif, sans avoir obtenu préalablement l'autorisation des titulaires de ces droits.

La contrefaçon et le faux
Conformément aux dispositions du code de la propriété intellectuelle, toute représentation ou reproduction intégrale ou partielle d'une œuvre de l'esprit faite sans le consentement de son auteur est illicite et constitue un délit pénal.
L'article 444-1 du code pénal dispose : « Constitue un faux toute altération frauduleuse de la vérité, de nature à causer un préjudice et accomplie par quelque moyen que ce soit, dans un écrit ou tout autre support d'expression de la pensée qui a pour objet ou qui peut avoir pour effet d'établir la preuve d'un droit ou d'un fait ayant des conséquences juridiques ».
L'article L335_3 du code de la propriété intellectuelle précise que : « Est également un délit de contrefaçon toute reproduction, représentation ou diffusion, par quelque moyen que ce soit, d'une œuvre de l'esprit en violation des droits de l'auteur, tels qu'ils sont définis et réglementés par la loi. Est également un délit de contrefaçon la violation de l'un des droits de l'auteur d'un logiciel (…) ».

Le plagiat est constitué par la copie, totale ou partielle d'un travail réalisé par autrui, lorsque la source empruntée n'est pas citée, quel que soit le moyen utilisé. Le plagiat constitue une violation du droit d'auteur (au sens des articles L 335-2 et L 335-3 du code de la propriété intellectuelle). Il peut être assimilé à un délit de contrefaçon. C'est aussi une faute disciplinaire, susceptible d'entraîner une sanction.
Les sources et les références utilisées dans le cadre des travaux (préparations, devoirs, mémoires, thèses, rapports de stage…) doivent être clairement citées. Des citations intégrales peuvent figurer dans les documents rendus, si elles sont assorties de leur référence (nom d'auteur, publication, date, éditeur…) et identifiées comme telles par des guillemets ou des italiques.

Les délits de contrefaçon, de plagiat et d'usage de faux peuvent donner lieu à une sanction disciplinaire indépendante de la mise en œuvre de poursuites pénales.