De l'arithmétique d'intervalles à la certification de programmes

Guillaume Melquiond

N° d'ordre : 388 N° attribué par la bibliothèque : 06ENSL0388

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

présentée et soutenue publiquement le 21 novembre 2006 par

Guillaume MELQUIOND

pour l'obtention du grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique

au titre de l'École doctorale de mathématiques et d'informatique fondamentale de Lyon

De l'arithmétique d'intervalles à la certification de programmes

Directeur de thèse : Marc DAUMAS

Après avis de : Guillaume HANROT
Christine PAULIN-MOHRING

Devant la commission d'examen formée de :

Jean-Daniel BOISSONNAT	Membre
Marc DAUMAS	Membre
Guillaume HANROT	Membre/Rapporteur
John HARRISON	Membre
Jean-Michel MULLER	Membre
Christine PAULIN-MOHRING	Membre/Rapporteur

Remerciements

Je tiens à remercier mon directeur de thèse Marc Daumas d'avoir vu le potentiel des travaux présentés ici et de m'avoir laissé une grande liberté.

Je tiens à remercier Christine Paulin-Mohring et Guillaume Hanrot d'avoir été les rapporteurs de cette thèse, Jean-Daniel Boissonnat d'avoir présidé, John Harrisson d'être venu de si loin et Jean-Michel Muller d'avoir complété le jury d'un membre local mais néanmoins compétent.

Je tiens à remercier mes coauteurs, Sylvie Boldo, Hervé Brönnimann, Florent de Dinechin, Christoph Lauter, César Muñoz et Sylvain Pion.

Je tiens à remercier Simone Criqui, Claude-Pierre Jeannerod, Nathalie Revol et Arnaud Tisserand pour leur relecture attentive de ce document. Toute faute que vous pourriez y détecter provient vraisemblablement des retouches de dernière minute que j'ai effectuées et non pas d'un manque de sérieux de leur part.

Je tiens à remercier Francisco, Nicolas, Romain et Sylvain d'avoir été des cobureaux sympathiques[1]. Je tiens à remercier les thésards du laboratoire qui ont su me faire perdre de nombreuses heures dans des jeux en réseau : Damien, Florent, Jérémie, Nicolas, Victor, Vincent. Les jeux en réseau ne sont pas la seule source de non-productivité et je tiens donc aussi à remercier Anne, Damien, Emmanuel, Florent, Laurent, Stéphane et Stéphane, Sylvain et Sylvain. Comme tous les chemins viennent de Lyon, il me faut aussi mentionner Benoît, Blaise, Christophe, Emmanuelle et Emmanuel, Laurent, Philippe. Et enfin, Camille et Camille pour leur réapparition surprise. J'ai probablement oublié de nombreuses personnes dans ces quelques listes et, tôt ou tard, je me réveillerai en sursaut au milieu de la nuit en me disant : « Flûtine, j'ai oublié de remercier Machin ! »

Je tiens à remercier les membres de l'équipe Arénaire qui ont su échapper aux énumérations précédentes, en particulier Gilles Villard et Sylvie Boyer. Il faudrait aussi que je remercie les habitants du troisième étage, Corinne, Dominique, Serge, Simone, mais il y en a trop, je ne vais pas m'en sortir. Sans compter les enseignants du département d'informatique de l'ENSL, ainsi que Jean-François Ponsignon pour l'INSA. Et enfin, les étudiants qui m'ont subi pendant ces quelques années de thèse.

Ce document tient à remercier Gedit, Latex et Rubber.

Pour finir : Pierre, on ne t'oublie pas.

[1] Si votre prénom n'apparaît pas dans cette liste, ce n'est pas que vous m'êtes antipathique, c'est juste que vous avez déjà été remercié une fois. La même remarque s'applique aux listes suivantes.

Table des matières

Chapitre 1

Introduction

L'explosion d'Ariane 5 lors de son vol inaugural [L+96], la chute de l'indice de la bourse de Vancouver dans les années 80 [MV99], la défaillance d'un missile Patriot lors de la première guerre du Golfe [Inf92] sont quelques-uns des nombreux exemples de problèmes informatiques aux conséquences parfois dramatiques. Dans certains cas, il s'agit d'une opération qui renvoie une valeur trop grande pour être stockée dans les registres, provoquant un comportement exceptionnel du programme. Dans d'autres cas, de petites erreurs de calcul s'accumulent au cours du temps et font progressivement diverger les valeurs du programme des valeurs idéales.

Ces problèmes n'ont pas été causés par un facteur extérieur : il n'y a pas eu de défaillance matérielle ou de rayon cosmique pour venir perturber les calculs. Le comportement de ces applications était conforme à leur implantation. Le problème réside donc en amont, au niveau de la conception. À l'heure où logiciels et matériels informatiques s'immiscent dans tous les domaines, il devient donc indispensable de garantir qu'une application fait bien ce qu'on attend d'elle. Une norme internationale [ISO05] propose différents niveaux (*Evaluation Assurance Level*) décrivant le soin apporté durant la conception d'un système. Le niveau le plus sûr, EAL 7, est accordé à condition que le logiciel/matériel ait non seulement été testé mais qu'en plus ses parties critiques aient été vérifiées formellement, c'est-à-dire à l'aide de méthodes mathématiques.

Cette norme est principalement conçue pour évaluer les composants qui jouent un rôle en matière de sécurité informatique (systèmes d'exploitation, pare-feux, etc). Mais cette obligation d'une vérification formelle pourrait être étendue aux applications qui effectuent des calculs numériques. Il est en effet indispensable de s'assurer que le résultat calculé par l'application répond bien au problème posé par l'utilisateur, au moins dans une certaine mesure. L'arithmétique d'intervalles est une des approches possibles : elle fournit des algorithmes et méthodes permettant de calculer des bornes rigoureuses sur des solutions approchées (erreur et valeur calculée) [Moo79, JKDW01].

Une autre approche consiste à effectuer une certification statique de l'applica-

tion. Le développeur apporte la preuve que les résultats calculés sont toujours en accord avec les spécifications et qu'il n'y a donc pas besoin de recourir à des méthodes robustes pour l'exécution de l'application. Même s'il est de plus en plus employé, ce type de certification formelle constitue un travail long, pénible et propice aux erreurs, à cause de la complexité des applications à traiter. L'outil informatique pourrait automatiser une part importante de ce travail et certifier formellement les applications effectuant des calculs numériques. Cette thèse décrit les méthodes développées pour le logiciel Gappa[1] afin qu'il accomplisse ce travail de certification.

Même si nous nous plaçons ici dans le domaine de la certification statique, cela ne signifie pas pour autant que l'arithmétique d'intervalles est abandonnée. Elle est en effet utilisée, non pas dans le programme à certifier, mais dans l'outil qui sert à le certifier. Son emploi garantit que l'outil ne s'est pas fourvoyé. En adaptant certaines de ses méthodes, elle permet même de générer un certificat formel prouvant que le programme se comportera correctement. L'un des enjeux de cette thèse consiste à sélectionner les méthodes de l'arithmétique d'intervalles suffisamment puissantes pour traiter des applications numériques tout en étant suffisamment simples pour ne pas être un frein à la construction des certificats de ces applications.

1.1 Certification de codes numériques

Les programmes considérés ici sont des codes effectuant des calculs numériques avec des arithmétiques approchées. Ces codes calculent des valeurs sans que leur flot d'exécution ne soit perturbé. Structures conditionnelles, terminaison des boucles, validité des accès mémoire sont laissées à l'analyse d'outils de certification plus généralistes [Fil03, vdBJ01]. Les portions de programme qu'il reste à analyser après le traitement de ces outils ne contiennent que des opérateurs arithmétiques et sont généralement qualifiées de *Straight-Line Programs*. Elles sont représentables par des graphes dirigés acycliques dont les nœuds sont des opérateurs arithmétiques. Afin de coller au comportement du programme tel qu'il sera exécuté sur un processeur, ces opérateurs ne sont pas les opérateurs théoriques qui calculent en précision infinie mais bien les opérateurs tels qu'ils sont implantés en matériel. En particulier, la précision des formats numériques utilisés est limitée et chaque valeur calculée est donc potentiellement entachée d'une erreur d'arrondi.

1.1.1 Bornes de variables et comportements invalides

Ces erreurs produites au cours du programme risquent de progressivement s'accumuler et de faire diverger les valeurs calculées des valeurs théoriques. À force de diverger, ces valeurs peuvent se retrouver hors des domaines prévus, conduisant ainsi à des divisions par zéro ou des racines carrées de nombres négatifs par exemple. Les valeurs peuvent aussi augmenter démesurément, franchissant alors

[1] Gappa est l'acronyme de « Génération Automatique de Preuves de Propriétés Arithmétiques ».

le seuil des valeurs représentables par le format numérique et provoquant des dépassements de capacité. Certaines fonctions du programme peuvent aussi avoir des pré-conditions qui sont vérifiées quand les calculs sont effectués en précision infinie mais pas quand ils sont perturbés par les erreurs.

Pour garantir que ces différents problèmes ne risquent pas de se produire, il faut pouvoir prouver que chacune des valeurs calculées satisfait les contraintes de domaine qu'imposent les parties du programme qui utilisent cette valeur. Considérons la fonction en langage C décrite par le code 1.1.

Code C 1.1 Exemple de fonction avec précondition

```
float f(float x) {
  assert(0 <= x && x <= 1);
  float y = x * (1 - x);
  return sqrt(0.25 - y);
}
```

Cette fonction f prend en argument un flottant simple précision x. L'assertion à son début spécifie que x est compris entre 0 et 1. La fonction calcule ensuite la valeur de y ; il s'agit du flottant $x \otimes (1 \ominus x)$. Ce ne sont pas ici la multiplication et la soustraction sur les nombres réels ; il s'agit d'opérateurs spécifiques liés à l'arithmétique flottante et à la précision limitée de ses nombres. Alors que l'on sait que $\frac{1}{4} - x \cdot (1 - x) = (x - \frac{1}{2})^2$ est positif ou nul pour toute valeur de x, il n'y a *a priori* aucune garantie que le nombre dont la fonction prend la racine carrée le soit. En particulier, quand x prend des valeurs aux alentours de 0.5, le nombre $x \otimes (1 - x)$ pourrait valoir strictement plus de 0.25 à cause des erreurs d'arrondi qui entachent les deux opérations approchées.

Pour garantir qu'aucun comportement exceptionnel ne se produit dans la fonction f, il faut prouver qu'elle ne cherchera pas à prendre la racine d'un nombre négatif. Si l'on note $\mathbb{F}$ l'ensemble des nombres flottants simple précision, cela revient à la proposition logique suivante :

$$\forall x \in \mathbb{F}, \quad 0 \leq x \leq 1 \quad \Rightarrow \quad 0.25 \ominus (x \otimes (1 \ominus x)) \geq 0.$$

Dans cet exemple, la certification du bon comportement d'une fonction a été ramenée à la preuve d'une proposition contenant des encadrements d'expressions arithmétiques. Ce genre de proposition est à la base des certifications que Gappa réalise.

1.1.2 Erreurs numériques

Comme le montrent certains exemples de programmes au comportement numérique imprévu, les variables sortant de leur domaine ne sont pas les seules causes de défaillances. Il peut aussi arriver que la valeur calculée soit tellement éloignée de la valeur espérée qu'elle provoque des réactions inattendues. Ces écarts peuvent être

attribués à plusieurs causes. Il y a les erreurs de méthode, au sens large, liées aux différences entre le problème réel et l'algorithme employé pour le résoudre. Des erreur d'approximations sont introduites quand on simplifie certaines des quantités de l'algorithme, par exemple les constantes afin de les stocker. L'erreur d'arrondi est l'erreur commise à l'exécution du programme parce que l'arithmétique employée n'est pas celle des nombres réels.

La frontière entre ces catégories n'est pas toujours très nette et leurs dénominations sont variables suivant les auteurs. Elles vont juste servir à préciser le cadre dans lequel ma thèse se situe. L'outil Gappa est spécialement conçu pour traiter les erreurs d'approximation et d'arrondi.

Toutes ces erreurs seront représentées soit sous forme absolue soit sous forme relative. Si l'on note x une valeur idéale et $\tilde{x}$ une valeur qui en est une approximation, par exemple la valeur effectivement calculée par le programme, alors l'erreur absolue est définie par $\tilde{x} - x$ tandis que l'erreur relative est définie par $\frac{\tilde{x}-x}{x}$.

Si l'on reprend la fonction exemple précédente, prouver que l'erreur absolue commise entre la valeur calculée y et la valeur idéale $x \cdot (1 - x)$ ne dépasse pas $3 \cdot 2^{-27}$ se ramène à prouver la proposition logique

$$\forall x \in \mathbb{F}, \quad 0 \leq x \leq 1 \quad \Rightarrow \quad |x \otimes (1 \ominus x)) - x \cdot (1 - x)| \leq 3 \cdot 2^{-27}.$$

1.2 L'outil Gappa

Une première approche pour montrer la proposition précédente serait de vérifier que son corps est vrai pour chaque valeur possible de la variable x. Une telle approche est par exemple employée dans la preuve de petits opérateurs arithmétiques implantés en matériel [DdD05]. Le nombre de valeurs distinctes de x est de l'ordre de 2^{30} et il est possible de réaliser un test exhaustif en un temps raisonnable.

Mais il suffit que x soit un argument en double précision et il y a alors 2^{62} valeurs à considérer. Si la fonction prend plusieurs arguments, ce nombre explose à nouveau. Il n'est alors plus possible d'effectuer un test exhaustif de toutes les valeurs. Des méthodes statistiques peuvent dans ce cas être employées. Elles n'offrent cependant qu'une estimation et non une garantie de la fiabilité du programme. Elles n'excluent pas la présence d'un comportement incorrect du programme aux conséquences inacceptables.

Gappa ne cherche pas à vérifier que la proposition est vraie pour chacune des entrées prise séparément, mais considère des intervalles de valeurs dans leur ensemble et vérifie que la proposition y est satisfaite. Cette proposition logique peut aussi bien couvrir des problèmes de bornes de variables que des questions d'erreur numérique, permettant ainsi à l'outil de participer à la certification de programmes numériques.

Un script soumis à Gappa se décompose en trois parties. La première partie fournit du sucre syntaxique permettant de définir aisément les expressions qui apparaissent dans la proposition logique. La deuxième partie, encadrée par une paire

d'accolades, est la description de cette proposition logique dont Gappa doit vérifier la validité. La troisième partie permet quant à elle de fournir des indications sur les méthodes à employer pour prouver la proposition logique au cas où l'outil échouerait.

Voici un premier exemple de script, il demande simplement à Gappa de prouver que la somme de deux nombres dans $[1, 2]$ et $[3, 4]$ est compris entre 3 et 7 :

```
{ x in [1,2] /\ y in [3,4] -> x + y in [3,7] }
```

Dans l'exemple précédent, les termes `x` et `y` n'ont pas été définis auparavant et sont donc considérés par l'outil comme étant des variables universellement quantifiées sur l'ensemble $\mathbb{R}$ des nombres réels. Les encadrements à gauche du symbole d'implication `->` correspondent aux deux hypothèses. Celui à droite est le but à atteindre. Si un point d'interrogation est utilisé à la place de l'intervalle, Gappa cherche un encadrement pour lequel il est capable de trouver une preuve de la proposition. Il répond ici $x + y \in [4, 6]$.

```
{ x in [1,2] /\ y in [3,4] -> x + y in ? }
```

Dans les deux cas, Gappa génère une preuve formelle de la propriété qu'il a calculée. Cette preuve du comportement numérique du programme peut être automatiquement vérifiée par un assistant de preuves tel que Coq [BC04]. Elle peut aussi être insérée au sein d'une preuve plus générale, preuve qui prendrait en compte non seulement les erreurs de calcul, mais aussi la terminaison des boucles, la validité des accès mémoire, etc.

Ce premier exemple de script ne contenait qu'une simple proposition. Considérons maintenant le script 1.1 associé aux propositions concernant la fonction f du code C 1.1. Les quatre premières lignes définissent un mode d'arrondi et les valeurs manipulées : y est la valeur calculée tandis que z est la valeur idéale. Puis vient la proposition à prouver : sachant que x est compris entre 0 et 1, la variable y est dans l'intervalle $[0, \frac{1}{4}]$ tandis que l'erreur $|y - z|$ commise est inférieure à $3 \cdot 2^{-27}$. Finalement, les deux dernières lignes donnent des indices à Gappa sur une façon de prouver la proposition : le premier précise que z prend les mêmes valeurs que $\frac{1}{4} - (x - \frac{1}{2})^2$ et le second indique à l'outil qu'une étude par cas sur x permet d'obtenir des encadrements de y et $|y - z|$.

Script Gappa 1.1 Exemple $x \otimes (1 \ominus x)$

```
@rnd = float< ieee_32, ne >;
x = rnd(xx);
y rnd= x * (1 - x);
z = x * (1 - x);

{ x in [0,1] -> y in [0,0.25] /\ |y - z| <= 3b-27 }

z -> 1/4 - (x - 1/2) * (x - 1/2);
y, |y - z| $ x;
```

1.3 Positionnement

Le travail de cette thèse se situe au confluent de plusieurs domaines. En particulier, on y retrouve des éléments de certification formelle de codes arithmétiques, de calcul automatique de l'erreur d'arrondi et de certification automatique de programmes. Les paragraphes ci-après regroupent quelques travaux notables dans ces domaines.

1.3.1 Certification formelle de codes arithmétiques

Les normes décrivant l'arithmétique à virgule flottante (IEEE-754 principalement, mais aussi IEEE-854) ont été spécifiées dans divers formalismes : pour la méthode Z [Bar89], pour les assistants de preuve HOL et PVS [CM95], pour l'assistant HOL Light [Har99], etc. L'arithmétique à virgule fixe a elle aussi fait l'objet de formalisations [ATD05]. Ces spécifications constituent les fondations sur lesquelles on peut s'appuyer pour certifier des applications utilisant des arithmétiques approchées.

On peut relever quelques exemples de travaux basés sur ces spécifications. L'unité des processeurs AMD consacrée aux calculs flottants a ainsi été formellement certifiée en ACL2 [MLK98, Rus00]. L'implantation en micro-code des algorithmes de division et de racine carrée en virgule flottante a, en particulier, été prouvée correcte. D'autres processeurs généralistes ont eux aussi fait l'objet de certifications formelles [KK03, SG02]. On peut même trouver des certifications d'unités flottantes complètes qui descendent jusqu'au niveau de la porte logique [JB05].

Comme exemple d'algorithme de plus haut niveau, on trouve un calcul approché de la fonction exponentielle en HOL Light [Har97]. Le formalisme employé pour sa certification fournit cette fois des primitives de haut niveau : l'algorithme a été décrit dans un langage impératif disposant directement des opérations flottantes d'addition et de multiplication.

Ces différents travaux ont eu pour but la certification d'algorithmes précis. On peut relever une approche orthogonale dans l'action de recherche coopérative intitulée Arithmétique des Ordinateurs Certifiée. L'objectif était cette fois d'explorer un grand nombre de propriétés mathématiques associées à une formalisation générique de l'arithmétique flottante. La bibliothèque Coq qui en a résulté [DRT01, Bol04] permet de garantir la validité de méthodes parfois subtiles employées en calcul flottant.

1.3.2 Calcul automatique de l'erreur d'arrondi

La certification d'un algorithme numérique, tel le calcul de l'exponentielle mentionné ci-dessus, nécessite de repérer les erreurs d'arrondi liées aux opérations approchées, de les propager le long des calculs et de borner leur influence globale. Parmi les outils capables de manipuler automatiquement ces erreurs, on peut relever Fluctuat [PGM04]. Il effectue une analyse statique d'un programme

en représentant à l'aide d'une série formelle la contribution de chacune des erreurs d'arrondi à l'erreur globale qui entache un résultat calculé. L'approche employée permet de traiter les boucles et exécutions conditionnelles d'un programme. Les résultats d'un tel outil ne sont cependant pas réutilisables au sein d'une certification formelle. Par conséquent, en l'absence d'un outil tel que Gappa, les calculs de bornes d'erreur devaient être effectués à la main pour pouvoir certifier formellement la correction des algorithmes.

Réaliser une analyse statique est l'approche la mieux adaptée pour certifier un programme, mais elle est en fait peu employée pour l'arithmétique flottante. De nombreux outils sont ainsi conçus pour transformer un programme afin qu'une estimation de l'erreur d'arrondi commise soit calculée à l'exécution. Le programme peut ainsi être enrichi pour fournir une borne fiable sur le résultat final en propageant des bornes d'erreur au cours du calcul [BFS01]. Une autre approche consiste à effectuer plusieurs le même calcul en variant les méthodes d'arrondi des résultats intermédiaires ; cela fournit une estimation statistique de la précision du résultat [Che95]. Une dernière approche consiste à estimer l'erreur globale en calculant précisément l'erreur commise à chaque étape ; cette estimation peut alors être réintroduite à la fin du calcul pour compenser l'erreur commise et ainsi améliorer la précision du résultat [Lan00]. Certaines de ces approches permettent d'obtenir des programmes au comportement certifiés, mais au prix d'une modification des algorithmes qu'ils implantent.

1.3.3 Certification automatique de programmes

Les calculs numériques et l'erreur qu'ils causent ne constituent qu'une part de l'exécution d'un programme. Une certification complète doit aussi prendre en considération les accès mémoire, les structures de contrôle et les interactions avec l'extérieur. Il existe plusieurs architectures de certification capables de générer des obligations de preuve à partir du code d'un programme. Ces architectures sont généralement dédiés à un langage particulier, Java par exemple [vdBJ01]. On peut cependant relever le cas de Why [Fil03] qui fournit un formalisme de bas niveau dans lequel plusieurs langages de programmation peuvent être exprimés ; les obligations de preuve générées à partir des assertions, pré-conditions et post-conditions du programme peuvent ensuite vérifiées à l'aide de divers assistants de preuve.

Ces outils sont cependant restreints au calcul entier et ne permettent pas pour l'instant d'attaquer des programmes flottants : il leur manque la notion de calcul approché. Parmi les résultants intéressants en arithmétique entière, on trouve une certification de l'algorithme de racine carrée de GMP par le biais d'obligations de preuve en Coq [BMZ02].

1.4 Plan du document

Les trois premiers chapitres sont consacrés au formalisme employé dans Gappa. Le chapitre 2 indique comment la preuve de propositions logiques concernant des

nombres réels est ramenée à des calculs sur des intervalles et quelles sont les propriétés de l'arithmétique d'intervalles employée. L'utilisation de cette arithmétique présente cependant un certain nombre de difficultés liées aux corrélations entre expressions manipulées. Le chapitre 3 présente des méthodes pour modérer l'impact d'une perte de corrélation. Les plus importantes d'entre elles sont basées sur la réécriture des expressions d'erreurs absolus et relatives. Afin de pouvoir traiter des propriétés sur des programmes, le formalisme de Gappa ne peut pas se limiter aux nombres réels. Le chapitre 4 l'étend aux ensembles discrets sur lesquels les programmes effectuent leurs calculs approchés. Ces ensembles sont traités par le biais d'opérateurs d'arrondi et de prédicats exprimant la précision des résultats des calculs approchés.

Viennent ensuite des considérations plus appliquées. Le chapitre 5 détaille les particularités liées à l'utilisation par l'outil des notions décrites aux chapitres précédents : propositions logiques, graphes de preuves, bissections et réécritures. Le chapitre 6 s'intéresse à l'utilisation des certificats produits par Gappa, et en particulier à leur interaction avec les assistants de preuves. Ces derniers doivent en effet pouvoir les interpréter en un temps raisonnable, ce qui oblige l'outil à en générer des versions simplifiées.

Le chapitre 7 présente quelques utilisations typiques de Gappa dans le domaine de la certification de fonctions élémentaires en arithmétique flottante. Les spécifications de ces fonctions fixent généralement l'erreur maximale qui peut exister entre les résultats calculés par leur implantation et les valeurs des fonctions mathématiques idéales ; Gappa est donc particulièrement adapté à leur certification. Le chapitre 8 décrit une méthode pour construire des filtres robustes en virgule flottante pour des prédicats géométriques homogènes. Elle est dérivée des concepts à la base de Gappa et met à profit les calculs de l'outil pour garantir la correction des filtres. Le chapitre 9 conclut ce document en présentant quelques perspectives liées à la certification d'applications numériques.

Comme l'arithmétique d'intervalles réapparaît en divers endroits tout au long de ce document, l'annexe A s'intéresse plus particulièrement à une implantation de cette arithmétique dans le langage C++. Pour finir, l'annexe B détaille le langage employé par Gappa tandis que l'annexe C fait l'inventaire de ses règles de réécriture et théorèmes.

Chapitre 2

Preuve et arithmétique d'intervalles

Ce chapitre présente le formalisme dans lequel les propositions logiques sont manipulées puis rappelle les propriétés de base de l'arithmétique d'intervalles et ses liens avec la preuve. Il détaille enfin les raisons qui nous ont conduits aux intervalles à bornes dyadiques. Ces intervalles offrent en particulier des opportunités de simplification des preuves générées afin d'accélérer leur vérification.

Gappa est conçu pour prouver des propositions logiques dont les briques élémentaires sont des encadrements d'expressions à valeurs réelles. Dans ce chapitre, les propositions ont systématiquement une forme simplifiée ; le paragraphe 5.1.1 indique comment l'outil s'y ramène. Le paragraphe 2.1 décrit le principal prédicat, l'appartenance à un intervalle, et comment il est manipulé au sein de ce formalisme. Le paragraphe 4.4 présente des prédicats complémentaires permettant de gérer des sous-ensembles non connexes de nombres réels.

Le principe de base de Gappa est d'ajouter progressivement à un groupe d'hypothèses de nouveaux faits qui en découlent, et cela jusqu'à ce que la conclusion de la proposition logique en fasse partie ou qu'une contradiction entre les hypothèses soit découverte. Le paragraphe refarithmetique-intervalles décrit l'arithmétique d'intervalles employée pour construire ces nouveaux faits. sont construits par arithmétique d'intervalles ; des bornes dyadiques[1] sont employées pour rendre cette méthode efficace.

L'arithmétique d'intervalles employée est minimaliste. Le chapitre 6 montre les avantages d'avoir une théorie réduite. Pour compenser l'absence des méthodes puissantes qui existent en arithmétique d'intervalles, le chapitre 3 présente des méthodes qui, bien que simples, n'en sont pas moins efficaces pour prouver les propositions que Gappa a à traiter.

[1] Dans ce document, l'adjectif dyadique ne fait pas référence aux nombres p-adiques de Kurt Hensel mais aux nombres rationnels dyadiques, c'est-à-dire le sous-ensemble de $\mathbb{Q}$ constitué des nombres de la forme $m \cdot 2^e$ avec m et e des entiers relatifs.

2.1 Preuves, ensembles et inclusion

Soient $v = (v_1, \ldots, v_p)$ un ensemble de variables à valeurs réelles. Considérons une proposition logique P de la forme

$$\forall v \in \mathbb{R}^p, \quad f_1(v) \in R_1 \wedge f_2(v) \in R_2 \wedge \ldots \wedge f_k(v) \in R_k \quad \Rightarrow \quad g(v) \in R.$$

Les $R_1, \ldots, R_k$ et R sont des sous-ensembles de $\mathbb{R}$. Les $f_1(v), \ldots, f_k(v)$ et $g(v)$ sont des expressions contenant les variables $v_1, \ldots, v_p$, des constantes et les opérateurs arithmétiques traditionnels : addition, soustraction, multiplication, division, racine carrée, valeur absolue. Des opérateurs d'arrondi sont ajoutés au chapitre 4. L'ajout de fonctions transcendantes comme $\sin$, $\exp$, etc correspondrait à une extension naturelle de ce formalisme. Nous avons cependant décidé de les laisser pour l'instant en dehors du cadre pratique de Gappa afin de nous concentrer sur les opérateurs arithmétiques de base. Ceux-ci constituent en effet la grande majorité des opérations rencontrées lors de la certification des programmes que nous avons considérés.

Les expressions $f_i(v)$ seront assimilées aux fonctions partielles qui les évaluent afin de simplifier les notations. Ainsi, si $f_1(v)$ est l'expression $v_3/(v_4 + v_6)$, alors f_1 nommera aussi la fonction $(x_1, \ldots, x_p) \in \mathbb{R}^p \mapsto x_3/(x_4 + x_6)$.

2.1.1 Preuve de propositions

Si l'on veut prouver la proposition P, il suffit de prouver une proposition qui implique P et d'en déduire une preuve de P. Cette nouvelle proposition sera construite sous une forme plus simple à prouver. Il est ainsi possible d'ajouter de nouvelles hypothèses $f_l(v) \in R_l$ à condition que ces hypothèses puissent être déduites des hypothèses déjà présentes.

Supposons maintenant que ces transformations mènent à deux hypothèses sur la même expression : $e(v) \in E_1$ et $e(v) \in E_2$. Si l'un des deux sous-ensembles E_1 ou E_2 contient l'autre, il suffit de garder l'hypothèse correspondante et de supprimer l'autre. Mais l'inclusion ne constitue pas un ordre total et il se peut donc qu'aucun des deux ne contienne l'autre. Il est alors possible de remplacer les deux hypothèses par l'hypothèse $e(v) \in E_1 \cap E_2$.

Si une hypothèse de la forme $e(v) \in \emptyset$ apparaît à force d'intersections, la proposition devient trivialement vraie. En effet, il n'existe alors aucun vecteur $v \in \mathbb{R}^p$ qui puisse satisfaire les hypothèses. Les autres propositions trivialement vraies sont celles où l'on obtient comme hypothèse $g(v) \in R'$ avec $R' \subseteq R$. On construit ainsi une suite de propositions dérivées de P en ajoutant des hypothèses jusqu'à obtenir une de ces propositions trivialement vraies.

Étant donnée une hypothèse $e(v) \in E$, il est aussi possible de construire deux propositions P_1 et P_2 dans lesquelles l'hypothèse est remplacée respectivement par $e(v) \in E_1$ et $e(v) \in E_2$. Contrairement aux transformations précédentes, on ne remplace pas la proposition par une nouvelle proposition mais par deux nouvelles propositions. En travaillant alors sur les deux propositions E_1 et E_2 à la fois,

on pourra en déduire une preuve de P à condition d'avoir $E \subseteq E_1 \cup E_2$. Cette transformation permet l'utilisation des méthodes de bissection présentées dans le paragraphe 3.1 au prix d'une augmentation du nombre de propositions à prouver.

2.1.2 Réduction aux intervalles

Parmi les transformations autorisées, il est aussi possible de remplacer les ensembles en hypothèses par des ensembles plus grands. Ainsi, si l'on remplace l'hypothèse $f_i(v) \in R_i$ par une hypothèse $f_i(v) \in R'_i$ sachant que $R_i \subseteq R'_i$, on obtient une nouvelle proposition qui implique P. Il est donc possible de simplifier des hypothèses en considérant que les expressions sont dans des sous-ensembles de $\mathbb{R}$ plus larges mais plus pratiques à manipuler. Les intervalles de $\mathbb{R}$ forment une telle classe de sous-ensembles puisqu'ils peuvent s'exprimer à l'aide des deux bornes qui délimitent chacun d'eux. Un problème qui concerne un nombre potentiellement infini de points se ramène ainsi à un problème qui concerne un nombre fini de bornes.

Cependant, les ensembles de nombres machine constituent rarement des intervalles : ce sont des sous-ensembles discrets de $\mathbb{R}$. Les intervalles ne sont donc pas toujours suffisants pour représenter des propositions apparaissant lors de la certification d'un code numérique. Agrandir ces ensembles pour en faire des intervalles risque de transformer la proposition P en une proposition improuvable. C'est pour pallier ce défaut que Gappa manipule aussi d'autres familles de sous-ensembles de $\mathbb{R}$. Elles sont détaillées au paragraphe 4.4.

2.2 Arithmétique d'intervalles

Ajouter de nouvelles hypothèses à P est autorisé à condition qu'elles soient conséquences des hypothèses déjà présentes dans P. Ces nouvelles hypothèses vont être construites en s'appuyant sur des théorèmes d'arithmétique d'intervalles. Ces théorèmes permettent de travailler sur des intervalles exclusivement, sans se préoccuper des propositions manipulées.

2.2.1 Types d'intervalles

Comme expliqué précédemment, les ensembles vides sont très pratiques puisqu'ils permettent de prouver n'importe quelle proposition. Il n'est cependant pas nécessaire de savoir comment les manipuler puisque dès que l'on en obtient un, le travail est terminé. Dans la suite, les intervalles seront donc généralement considérés comme des sous-ensembles non vides de $\mathbb{R}$.

Plus précisément, les intervalles sont des ensembles connexes de $\mathbb{R}$, c'est-à-dire que si deux points a et b sont présents dans un intervalle alors tout point x entre a et b est compris dans cet intervalle. Un intervalle est donc représentable par ses bornes inférieure et supérieure. Un exemple d'intervalle est $\{x \in \mathbb{R} \mid \pi \leq x < 5\}$. Ses deux bornes sont les réels π et 5. Une seule d'entre elles appartient à l'intervalle :

π. Les premières questions qui se posent sont donc : les bornes peuvent-elles être infinies ? Si elles sont finies, appartiennent-elles à l'intervalle ?

Les inégalités sont représentables à l'aide d'intervalles dont une seule des bornes est infinie. Par exemple, $x \geq 5$ s'exprime par $x \in I$ avec I un intervalle dont une borne est 5 et l'autre est $+\infty$. Les inégalités n'ont cependant qu'un intérêt limité pour le calcul. Par exemple, à partir des seules hypothèses $x \geq 5$ et $-3 \leq y \leq 2$, il est impossible de déduire une quelconque propriété concernant le produit $x \cdot y$. Dans Gappa, les inégalités ne sont donc pas manipulées d'un point de vue arithmétique mais servent seulement lors d'intersections entre plusieurs hypothèses concernant la même expression. En particulier, il n'est pas nécessaire de formaliser la notion de borne infinie : elle reste un simple détail d'implantation.

Pour suivre la pratique existante, les bornes sont incluses dans les intervalles. L'intervalle $\{x \in \mathbb{R} \mid a \leq x \leq b\}$ est ainsi représentable par la paire (a, b) de deux nombres réels. La gestion d'intervalles dont les bornes peuvent être soit fermées soit ouvertes conduirait à une explosion du nombre de théorèmes d'arithmétique d'intervalles. Ainsi, comme détaillé au paragraphe 6.3, la multiplication utilise neuf théorèmes simples. Pour conserver ce principe des théorèmes simples tout en prenant en compte différents types de bornes, ce nombre aurait été multiplié par 16 (voire par 64 si on autorisait des bornes infinies en plus des bornes ouvertes).

Pour résumer, les intervalles tels qu'ils sont manipulés dans Gappa sont donc des sous-ensembles connexes, fermés, bornés et non-vides de $\mathbb{R}$. Ils sont représentés par leurs bornes inférieures et supérieures qui sont des nombres réels. Le paragraphe 2.3.1 restreint les bornes possibles à un sous-ensemble dénombrable de $\mathbb{R}$, ne serait-ce que pour pouvoir les représenter en machine.

2.2.2 Opérateurs arithmétiques

Ce qui rend l'arithmétique d'intervalles particulièrement intéressante, c'est sa propriété fondamentale. Étant donné un opérateur binaire $\diamond$ défini sur les réels, l'arithmétique d'intervalles définit ce même opérateur sur des intervalles et il vérifie la propriété suivante :

$$\forall I_x, I_y \subseteq \mathbb{R}, \quad x \in I_x \wedge y \in I_y \quad \Rightarrow \quad x \diamond y \in I_x \diamond I_y.$$

Cette propriété est utilisée pour générer de nouvelles hypothèses. Ainsi, s'il y a deux hypothèses $e_1(v) \in R_1$ et $e_2(v) \in R_2$, on peut en déduire une troisième hypothèse $(e_1 \diamond e_2)(v) \in R_1 \diamond R_2$. Ou plutôt, puisque nous avons vu que les intervalles en hypothèse pouvaient grossir, $(e_1 \diamond e_2)(v) \in R_3$ avec l'intervalle R_3 vérifiant la propriété $R_3 \supseteq R_1 \diamond R_2$.

Il convient donc de définir des opérateurs sur les intervalles pour chacun des opérateurs pouvant apparaître dans une expression. En utilisant des intervalles, on est ramené à de simples calculs sur les bornes. Les propriétés suivantes s'obtiennent

en considérant la monotonie (par morceau) des opérateurs réels [Moo79].

$$\begin{aligned} -[a,b] &= [-b,-a] \\ [a,b]^{-1} &= [\tfrac{1}{b},\tfrac{1}{a}] \quad \text{si } 0 \notin [a,b] \\ \sqrt{[a,b]} &= [\sqrt{a},\sqrt{b}] \quad \text{si } a \geq 0 \\ |[a,b]| &= [\max(0,a,-b),\max(-a,b)] \end{aligned}$$

$$\begin{aligned} [a,b]+[c,d] &= [a+c,b+d] \\ [a,b]-[c,d] &= [a-d,b-c] \\ [a,b]\times[c,d] &= [\min(a\cdot c,b\cdot c,a\cdot d,b\cdot d),\max(a\cdot c,b\cdot c,a\cdot d,b\cdot d)] \\ [a,b]\div[c,d] &= [a,b]\times[c,d]^{-1} \end{aligned}$$

Les opérations d'intersection et d'union s'expriment elles aussi sur les bornes. L'union est ici un peu spéciale vu qu'elle renvoie en réalité l'enveloppe convexe dans le cas où les deux intervalles en entrée ne sont pas en contact. Pour l'intersection, si jamais la valeur calculée pour la borne inférieure est plus grande que celle pour la borne droite, alors l'intervalle résultant est vide et la preuve est terminée.

$$\begin{aligned} [a,b]\cap[c,d] &= [\max(a,c),\min(b,d)] \\ [a,b]\cup[c,d] &= [\min(a,c),\max(b,d)] \end{aligned}$$

2.2.3 Exemple

Voyons comment prouver l'invariant de boucle du code C 2.1. Le chapitre 4 montrera comment traiter des nombres flottants et leurs arrondis ; pour l'instant et afin de simplifier cet exemple, arithmétique flottante et arithmétique réelle sont supposées équivalentes.

Code C 2.1 Boucle avec invariant

```
float f(float x, int n) {
  for(int i = 1; i < n; ++i) {
    assert(1 <= x && x <= 2);
    x = x / (i + 1) + 1;
  }
  return x;
}
```

Prouver l'invariant demande de vérifier que, si x est compris entre 1 et 2, alors $\frac{x}{i+1}+1$ l'est aussi. Le script 2.1 montre la traduction du problème dans la syntaxe de Gappa. Elle nécessite de borner i, ce qui peut par exemple se faire en supposant que n est un entier signé 32 bits et que i ne peut donc pas dépasser 2^{31}. Dans la syntaxe Gappa, cette borne sera notée `1b31` (qui représente le nombre dyadique $1\cdot 2^{31}$).

Script Gappa 2.1 Description de l'invariant de boucle

```
{ x in [1,2] /\ i in [1,1b31] -> x / (i + 1) + 1 in [1,2] }
```

Voici l'évolution de l'ensemble des hypothèses lors de la construction d'une preuve de la proposition. Pour chaque transformation, l'opération appliquée est indiquée sur la gauche. Ses opérandes sont soulignés et l'encadrement résultant est écrit en gras. Le dernier résultat calculé est un encadrement de $\frac{x}{i+1} + 1$ dont l'intervalle est inclus dans l'intervalle $[1, 2]$ demandé.

$$
\begin{aligned}
\text{départ} \quad & x \in [1,2], i \in [1,2^{31}] \\
\text{(constante)} \quad & x \in [1,2], i \in [1,2^{31}], \mathbf{1} \in [1,1] \\
\text{(addition)} \quad & x \in [1,2], \underline{i} \in [1,2^{31}], \underline{1} \in [1,1], \mathbf{i+1} \in [2,2^{31}+1] \\
\text{(division)} \quad & \underline{x} \in [1,2], \ldots, 1 \in [1,1], \underline{i+1} \in [2,2^{31}+1], \frac{\mathbf{x}}{\mathbf{i+1}} \in \left[\frac{1}{2^{31}+1}, \frac{1}{2}\right] \\
\text{(addition)} \quad & \ldots, \underline{1} \in [1,1], \ldots, \underline{\frac{x}{i+1}} \in \left[\frac{1}{2^{31}+1}, \frac{1}{2}\right], \frac{\mathbf{x}}{\mathbf{i+1}} + \mathbf{1} \in \left[\frac{2^{31}+2}{2^{31}+1}, \frac{3}{2}\right] \\
\text{arrivée} \quad & \frac{x}{i+1} + 1 \in [1,2]
\end{aligned}
$$

L'exemple précédent suppose que les calculs sont effectués en arithmétique réelle. Le script 2.2 ne fait pas cette hypothèse simplificatrice : il prend en compte les arrondis au plus près en simple précision effectués lors des opérations flottantes de la fonction étudiée.

Script Gappa 2.2 Invariant de boucle avec opérations arrondies

```
y float<ieee_32,ne>= x / (i + 1) + 1;
{ x in [1,2] /\ i in [1,1b31] -> y in [1,2] }
```

2.3 Les intervalles en pratique

Les intervalles sont pour l'instant des paires de nombres réels. Pour pouvoir les manipuler à l'aide d'un ordinateur, il faut leur donner une représentation plus concrète. L'aspect « paire » ne présente aucune difficulté ; l'aspect « nombres réels » est par contre moins évident.

2.3.1 Bornes d'intervalles

Expressions symboliques

L'approche la plus générique consisterait à utiliser des expressions symboliques comme bornes. De tels intervalles sont par exemple utilisés pour optimiser les programmes lors de la compilation en propageant des intervalles de valeurs [Pat95]. Afin de reproduire fidèlement les contraintes sur les indices dans les nids de boucles, les intervalles les encadrant ont alors des bornes qui sont des combinaisons affines à coefficients constants d'autres indices.

Les multiplications par des constantes sont cependant rares dans les expressions que Gappa manipule. Les fonctions $\min$ et $\max$ employées dans les bornes du produit ne se simplifieront donc pas. Les bornes symboliques vont aussi limiter l'usage des théorèmes de division puisqu'il sera difficile de vérifier que l'intervalle dénominateur ne contient pas zéro. Il est donc préférable d'employer des bornes constantes.

Nombres calculables et algébriques

Les représentations les plus génériques couvrent tous les réels calculables. Connaître le signe d'un réel calculable n'est pas un problème décidable. On peut cependant contourner le problème lié aux fonctions $\min$ et $\max$ en fixant un seuil $\epsilon > 0$. Pour un nombre calculable x, il devient alors possible de prouver l'une des trois propriétés $x > 0$, $x < 0$ ou $|x| \leq \epsilon$ et de faire apparaître ϵ dans la multiplication d'intervalles au lieu de $\min$ et $\max$. Des travaux récents [Zum06] montrent que cette approche de l'arithmétique d'intervalles est certes lente, mais pas au point d'être inutilisable en pratique.

Le coût qu'engendre la manipulation de ces bornes ne nous a pas semblé justifié au regard du domaine dans lequel Gappa est spécialisé, celui des arithmétiques machine. Il n'y a en effet pas besoin de l'ensemble complet des réels calculables pour représenter les nombres machine. L'emploi des nombres algébriques a été rejeté pour la même raison et notre attention s'est portée sur les nombres rationnels.

Nombres rationnels et dyadiques

Le calcul sur les nombres rationnels peut se faire rapidement et nous les avons employés comme bornes d'intervalles pour vérifier avec l'assistant de preuves PVS [ORS92] une borne sur l'erreur commise dans l'algorithme approchant une fonction élémentaire [DMM05]. Les paragraphes 3.3.1 et 6.1 présentent des points de ce travail.

Le chapitre 4 détaille les opérateurs d'arrondi présents dans les expressions manipulées par Gappa. Les nombres les plus simples à arrondir avec ces opérateurs sont les nombres rationnels dont le dénominateur est une puissance de 2. Ces rationnels constituent l'ensemble $\mathbb{D}$ des nombres dyadiques et sont de la forme

$m \cdot 2^e$ avec (m, e) un couple d'entiers relatifs (m est appelé la mantisse et e l'exposant). Ils sont particulièrement adaptés aux arithmétiques machine puisque les variables dans des formats binaires à virgule fixe ou flottante prennent des valeurs dyadiques. Nous avons par conséquent employé des nombres dyadiques comme bornes des intervalles dans Gappa.

La multiplication de deux dyadiques s'effectue en multipliant les mantisses et en additionnant les exposants. L'addition demande quant à elle d'ajuster les exposants et d'aligner en conséquence les mantisses avant de les additionner. La comparaison de deux nombres dyadiques peut se faire en les soustrayant et en observant le signe de la mantisse résultante. Les opérations de division et racine carrée ne sont pas définies sur les nombres dyadiques, mais le paragraphe 6.3.3 montre qu'elles ne sont en fait pas nécessaires pour vérifier des preuves contenant des divisions et des racines carrées d'intervalles.

2.3.2 Croissance et arrondi des bornes

Avec des bornes rationnelles, il n'est généralement pas possible de représenter l'intervalle $\sqrt{I}$. Avec des bornes dyadiques, il n'est généralement pas non plus possible de représenter l'intervalle I/J. Les bornes réelles théoriques vont donc être arrondies vers l'extérieur de l'intervalle afin de devenir représentables. La borne inférieure est arrondie vers le bas, la borne supérieure est arrondie vers le haut. Cette façon de produire les résultats est en fait utilisée par toutes les bibliothèques d'arithmétique d'intervalles qui fournissent des résultats garantis et travaillent avec les nombres flottants disponibles en machine.

Pour pouvoir effectuer les arrondis, il faut se fixer une contrainte. Dire par exemple que ni le numérateur ni le dénominateur du résultat ne peuvent dépasser n chiffres pour les rationnels, de même pour la mantisse dans le cas des nombres dyadiques. À partir de là, il n'y a pas vraiment de raison de limiter cette contrainte uniquement à la racine carrée (et à la division pour les dyadiques). En effet, la taille aussi bien des nombres rationnels que des nombres dyadiques a tendance à croître rapidement au fur et à mesure des multiplications. En particulier, le produit de deux nombres dyadiques de mantisse impaire et de longueur respectivement k et l est un nombre dyadique de mantisse impaire et de longueur $k + l - 1$.

En fixant la précision maximale des nombres dyadiques générés, le temps nécessaire aux assistants de preuve pour vérifier les calculs contenus dans les certificats générés par Gappa n'augmente au pire que linéairement en fonction du nombre d'étapes qu'ils contiennent. Le paragraphe 6.3.4 montre par contre que le temps de vérification augmente empiriquement de façon quadratique vis-à-vis de la précision maximale des nombres dyadiques générés. Le paragraphe ci-après montre comment s'affranchir du surcoût causé par l'emploi d'une précision trop élevée dans les calculs de Gappa.

2.3.3 Simplification et oracle

Un nombre rationnel non nul est dit normalisé s'il n'y a aucun facteur commun à son numérateur et à son dénominateur. Dans le cas des nombres dyadiques, cela se traduit par la propriété : un nombre non nul $m \cdot 2^e$ est normalisé à condition que m soit un entier impair. Normaliser un nombre est simple, il suffit d'incrémenter son exposant e et de diviser sa mantisse m par deux jusqu'à ce que la mantisse devienne impaire. Cette opération de normalisation permet de limiter la taille des nombres manipulés lors de la vérification de preuves et donc de l'accélérer.

Il est cependant possible de pousser ce principe de simplification un peu plus loin. Notons $\|n\|$ le nombre de chiffres de l'écriture binaire de n et définissons une relation de simplicité entre des nombres non nuls $a = m_a \cdot 2^{e_a}$ et $b = m_b \cdot 2^{e_b}$ comme suit. Ces deux nombres sont supposés déjà normalisés, c'est-à-dire que m_a et m_b sont des entiers impairs. Le nombre a est considéré plus simple que b si l'une des deux conditions suivantes est remplie :

1. $|m_a| = |m_b| = 1$ et $|e_a| < |e_b|$
2. $\|m_a\| < \|m_b\|$ et $|e_a| + \|m_a\| \leq |e_b| + \|m_b\|$

Étant donné un nombre dyadique $m \cdot 2^e$ avec m entier impair valant au moins trois, le nombre $(m-1) \cdot 2^e$ est le plus grand nombre dyadique qui soit inférieur à $m \cdot 2^e$ tout en étant plus simple. Similairement $(m+1) \cdot 2^e$ est le plus petit nombre supérieur à m qui soit plus simple. Pour 2^e, le nombre plus simple juste inférieur est 2^{e-1} si e est strictement positif, tandis que le nombre plus simple juste supérieur est 2^{e+1} si e est strictement négatif. Notons $\leftarrow_-$ et $\rightarrow_+$ les relations entre un nombre et les nombres plus simples juste inférieur et supérieur. Voici deux exemples de chaines maximales de simplification.

$$1 \leftarrow_- 2 \leftarrow_- 4 \leftarrow_- 8 \leftarrow_- \mathbf{9} \rightarrow_+ 10 \rightarrow_+ 12 \rightarrow_+ 16$$

$$\frac{1}{4} \leftarrow_- \frac{\mathbf{5}}{\mathbf{16}} \rightarrow_+ \frac{3}{8} \rightarrow_+ \frac{1}{2} \rightarrow_+ 1$$

Gappa effectue tous ses calculs à une précision donnée afin de générer une première version de la preuve. Puis il agrandit les intervalles présents en hypothèse de chaque théorème en simplifiant leurs bornes [DM04]. Les nombres produits ayant une taille plus petite, la vérification des encadrements de la nouvelle preuve en sera accélérée. Afin de s'assurer que la proposition prouvée n'est pas modifiée lors de ces transformations, Gappa parcourt la preuve à rebours et ne simplifie l'intervalle d'une hypothèse de théorème que si cela ne modifie pas l'intervalle de la conclusion de ce théorème. Après ces transformations, la nouvelle preuve contient exactement les mêmes étapes que la preuve initiale, mais impliquant des bornes représentées avec moins de chiffres.

Considérons l'exemple de la preuve de $\sqrt{1/3} \in [\frac{1}{2}, \frac{3}{4}]$. Le script Gappa correspondant tient en une ligne : `{ sqrt(1 / 3) in [0.5,0.75] }`. La preuve initialement générée par l'outil contient les étapes suivantes :

1. $1 \in [1, 1]$,

2. $3 \in [3, 3]$,
3. $1/3 \in [384307168202282325 \cdot 2^{-60}, 768614336404564651 \cdot 2^{-61}]$,
4. $\sqrt{1/3} \in [332819770519635731 \cdot 2^{-59}, 665639541039271463 \cdot 2^{-60}]$.

Le résultat obtenu à l'étape 4 est trop précis par rapport à la proposition logique à prouver, Gappa agrandit donc l'intervalle de la conclusion à $[\frac{1}{2}, \frac{3}{4}]$. L'intervalle en hypothèse (l'encadrement de $1/3$) peut alors être simplifié jusqu'à $[\frac{1}{4}, \frac{1}{2}]$ sans que la racine carrée de cet intervalle ne déborde de $[\frac{1}{2}, \frac{3}{4}]$. La conclusion de l'étape 3 ayant grandi, ses hypothèses peuvent à leur tour être simplifiées. L'intervalle $[1, 1]$ ne peut pas être simplifié plus. L'encadrement de 3 peut par contre grandir jusqu'à $[2, 4]$ sans perturber l'encadrement du quotient. Les bornes contenues dans la preuve ne sont alors presque plus que des puissances de 2 au lieu d'être des nombres d'environ 59 bits[2].

1. $1 \in [1, 1]$,
2. $3 \in [2, 2^2]$,
3. $1/3 \in [2^{-2}, 2^{-1}]$,
4. $\sqrt{1/3} \in [2^{-1}, 3 \cdot 2^{-2}]$.

En ramenant les nombres employés à chaque étape de la preuve à la précision localement optimale, cette méthode de simplification permet de limiter le coût qu'à une précision excessive sur le temps de vérification. Dans l'exemple décrit au paragraphe 6.3.4, ce coût disparaît même complètement : le temps de vérification devient indépendant de la précision employée par Gappa en interne.

[2]Par défaut, Gappa effectue ses calculs avec une précision maximale de 60 bits pour représenter les mantisses des nombres dyadiques. Après normalisation pour éliminer les mantisses paires, la taille des nombres peut être légèrement inférieure à 60 bits.

Chapitre 3

Intervalles et perte de corrélation

Ce chapitre détaille quelques méthodes simples permettant de diminuer le mauvais impact qu'ont les corrélations d'expressions sur l'arithmétique d'intervalles naïve. Il y a d'abord la traditionnelle bissection. Mais Gappa bénéficie essentiellement de la réécriture des expressions d'erreur. L'introduction de nouveaux opérateurs arithmétiques permet aussi de limiter les phénomènes de décorrélation.

Le principal problème lié à une utilisation naïve de l'arithmétique d'intervalles est le phénomène de décorrélation. Cette arithmétique n'est en effet pas capable de garder une trace de la corrélation qui existe entre des occurrences multiples d'une même variable, voire d'une même sous-expression.

Considérons par exemple l'expression $e = \frac{x}{x+1}$. La fonction associée est croissante et continue sur la demi-droite $]-1, +\infty[$. Pour $x \in [4, 9]$, l'expression prend donc ses valeurs dans l'intervalle $[0.8, 0.9]$ et cet intervalle est optimal. Si l'on effectue une évaluation par intervalles, on obtiendrait par contre que e prend ses valeurs dans $[4, 9]/[5, 10] = [0.4, 1.8]$, c'est-à-dire un intervalle quatorze fois trop large. Modifions maintenant l'expression e afin qu'il ne reste qu'une unique occurrence de x : $e = 1 - \frac{1}{x+1}$. L'évaluation par intervalles redevient alors optimale.

L'expression la plus simple qui pose problème est $x - x$. Elle vaut évidemment zéro quelle que soit la valeur prise par x. Mais une évaluation par intervalles renvoie un intervalle de largeur $2 \cdot w(X)$ si x est inclus dans un intervalle X de largeur $w(X)$. À moins que X soit un intervalle réduit à un unique point, il est donc impossible d'obtenir un intervalle ne contenant que 0 quand on encadre $x - x$ par $X - X$.

Il faut noter que de nombreuses méthodes [JKDW01] permettent de contourner les corrélations en mettant à profit la continuité et les multiples dérivabilités des expressions manipulées. Cependant, dès que les opérateurs d'arrondi décrits au chapitre 4 font leur apparition dans les expressions, celles-ci deviennent beaucoup trop discontinues pour que ces méthodes s'appliquent. En double précision,

la fonction qui associe à tout nombre réel le flottant le plus proche a beau être constante par morceaux, elle n'en possède pas moins de 2^{63} points de discontinuité environ.

3.1 Bissection d'intervalles

Dans l'exemple de l'expression $x - x$, la largeur de l'intervalle de sortie est proportionnelle à celle de l'intervalle d'entrée. De façon plus générale, pour une expression donnée, la fonction d'intervalles correspondante est croissante pour l'ordre partiel de l'inclusion d'intervalles. Pour réduire la largeur des intervalles en sortie, il est donc naturel de chercher à considérer des intervalles moins larges en entrée. Les méthodes de bissection sont couramment rencontrées en arithmétique d'intervalles et peuvent être utilisées à cette fin.

Au lieu de considérer l'intervalle d'entrée I en entier, l'idée est donc de le découper en deux sous-intervalles I_1 et I_2 vérifiant $I_1 \cup I_2 \supseteq I$ et d'effectuer des évaluations sur chacun de ces sous-intervalles. Ces évaluations vont respectivement produire des intervalles J_1 et J_2. Sur I, l'expression est donc bornée par un intervalle J vérifiant $J \supseteq J_1 \cup J_2$. Si l'intervalle ainsi obtenu reste trop grand, la méthode peut être appliquée récursivement à I_1 et I_2.

3.1.1 Exemples et inconvénients

Reprenons l'exemple de $e = \frac{x}{x+1}$ et découpons l'intervalle $[4, 9]$ qui contient x en $[4, 6.5]$ et $[6.5, 9]$. L'évaluation par intervalle donne sur chacun de ces sous-intervalles les encadrements : $e \in [0.5, 1.3]$ et $e \in [0.6, 1.2]$. L'expression e est donc encadrée par l'union $[0.5, 1.3]$ de ces deux intervalles. Le plus mauvais encadrement se trouve sur le premier intervalle ; découpons-le à nouveau en deux morceaux : $[4, 5]$ et $[5, 6.5]$. Les encadrements sur chacun des sous-intervalles sont donc cette fois $[0.6, 1]$ et $[0.6, 1]$. Cela donne maintenant comme encadrement total $[0.6, 1.2]$. L'intervalle reste six fois trop large que l'intervalle optimal, mais il est meilleur que l'intervalle $[0.4, 1.8]$ obtenu initialement.

Reprenons maintenant l'exemple $x - x$. Si l'on découpe l'intervalle X en 2^n sous-intervalles de largeur $2^{-n} \cdot w(X)$, on va obtenir un encadrement $x - x$ de largeur $2^{1-n} \cdot w(X)$. Par bissection, il est ainsi possible de réduire d'un facteur arbitraire la taille de l'intervalle de sortie, mais sans pour autant qu'il soit possible d'atteindre l'intervalle optimal $[0, 0]$. Qui plus est, durée de l'analyse et taille de la preuve finale seront nécessairement en $O(2^n)$.

Cet exemple est un cas extrême, il n'est généralement pas nécessaire que tous les sous-intervalles aient la même largeur pour réussir à prouver une proposition ; le nombre de sous-intervalles sera donc limité. Il peut cependant falloir découper les domaines suivant plusieurs variables et la complexité de cette méthode augmente alors exponentiellement par rapport au nombre de variables. La bissection est donc loin d'être une panacée. Elle ne devrait être envisagée qu'en dernier recours et

Gappa ne l'emploie donc qu'à la demande expresse de l'utilisateur.

3.1.2 Avantages de la bissection

Dans le cadre de la preuve de propositions, la bissection présente cependant des avantages. Tout d'abord, elle permet de trouver des contre-exemples. En effet, la bissection découpe récursivement un intervalle hypothèse en sous-intervalles tant que la proposition n'est pas vérifiée. Si la proposition n'est jamais vérifiée sur tous les sous-intervalles, Gappa finira par abandonner, mais aura obtenu un intervalle très fin. Dans le cas où l'on manipule des données discrètes comme des nombres flottants, cet intervalle sera même réduit à un unique point comme le montre le paragraphe 4.3.1. Quoi qu'il en soit, il est probable que ce petit intervalle contienne un point pour lequel la proposition n'est pas satisfaisable.

Autre utilisation incontournable, la bissection permet de prouver qu'une proposition est vraie sur un domaine entier, mais en utilisant des méthodes différentes en fonction du sous-domaine. Considérons par exemple $x \in [0,3] \Rightarrow |x \cdot t| \leq 2^{-26}$ avec $t = (x \ominus 1) - (x - 1)$. L'opérateur $\ominus$ désigne un opérateur de soustraction en virgule flottante. Sans rentrer dans le détail, il suffit de savoir pour l'instant qu'un théorème général sur l'erreur d'arrondi commise lors du calcul de $x \ominus 1$ affirme que $|t| \leq 2^{-22}$ pour $x \in [0,3]$. Avec un tel résultat, il est évidemment impossible de borner $|x \cdot t|$ par 2^{-26}.

Cependant, ce même théorème général permet de prouver que $|t| \leq 2^{-25}$ pour $x \in [0, \frac{1}{2}]$, tandis qu'une succession de lemmes permet de prouver que t vaut 0 pour $x \in [\frac{1}{2}, 3]$. Si l'on ne découpait pas l'intervalle sur x en sous-intervalles, aucun de ces deux résultats ne pourrait s'appliquer. On peut alors déduire que $|x \cdot t|$ est borné par 2^{-26} pour $x \in [0, \frac{1}{2}]$ et vaut zéro sur le reste de $[0,3]$.

La bissection peut aussi servir à évaluer l'influence de la corrélation entre les différentes occurrences d'une même sous-expression. Si cette influence est importante, découper l'encadrement de cette sous-expression en quelques sous-intervalles va sensiblement réduire la taille des encadrements prouvés. Gappa pourrait lancer quelques bissections en aveugle afin de détecter les sous-expressions critiques du problème.

3.2 Réécritures d'expressions d'erreur

Comme la bissection ne constitue pas une solution satisfaisante aux exemples $x - x$ et $\frac{x}{x+1}$, ils vont être réécrits respectivement en 0 et $1 - \frac{1}{x+1}$ avant d'être évalués par intervalles. Les deux transformations précédentes sont simples, mais elles ne s'appliquent pas aux expressions d'erreur, de la forme $\tilde{e} - e$ ou $\frac{\tilde{e}-e}{e}$.

Borner de telles expressions d'erreur sert généralement à montrer que les valeurs que prennent les expressions $\tilde{e}$ et e sont proches. Cela signifie malheureusement que les expressions $\tilde{e}$ et e sont fortement corrélées. En particulier, calculer séparément les intervalles $\tilde{E}$ et E contenant les valeurs de $\tilde{e}$ et e, puis les soustraire,

ne produit pas un résultat utilisable. La largeur de l'intervalle $\tilde{E} - E$ est en effet de l'ordre de $w(\tilde{E}) + w(E)$, c'est-à-dire $2 \cdot w(E)$. Quand les valeurs de e ont une grande amplitude, la largeur de E est élevée et rend l'encadrement obtenu pour $\tilde{e} - e$ inutile.

Plutôt que d'évaluer une expression d'erreur sous sa forme initiale, il est donc préférable de la réécrire avant d'utiliser l'arithmétique d'intervalles. Les méthodes employées constituent une des fondations de l'outil et présentent une approche novatrice. Elle est liée à la forme particulière des expressions traitées par Gappa, forme à laquelle sont rarement confrontés les autres outils s'appuyant sur l'arithmétique d'intervalles.

3.2.1 À partir des arbres syntaxiques

Si $\tilde{e}$ et e contiennent respectivement des sous-expressions $\tilde{x}$ et x dont on sait borner l'écart $\delta_x = \tilde{x} - x$, alors il faut chercher à faire apparaître le terme δ_x dans $\tilde{e} - e$ afin de réduire, voire d'éliminer, la décorrélation qui apparaît quand on utilise les encadrements $\tilde{x} \in \tilde{X}$ et $x \in X$.

Considérons par exemple la différence de deux produits : $\delta = \tilde{x} \cdot \tilde{y} - x \cdot y$. Cette erreur absolue liée à la multiplication peut être réécrite d'au moins trois façons équivalentes :

$$\begin{aligned}
\delta &= \tilde{x} \cdot (\tilde{y} - y) + (\tilde{x} - x) \cdot y = \tilde{x} \cdot \delta_y + \delta_x \cdot y \\
\delta &= (\tilde{x} - x) \cdot \tilde{y} + x \cdot (\tilde{y} - y) = \delta_x \cdot \tilde{y} + x \cdot \delta_y \\
\delta &= x \cdot (\tilde{y} - y) + (\tilde{x} - x) \cdot y + (\tilde{x} - x) \cdot (\tilde{y} - y) \\
&= x \cdot \delta_y + \delta_x \cdot y + \delta_x \cdot \delta_y
\end{aligned}$$

Chacune de ces expressions réécrites fait apparaître les termes d'erreur $\delta_x = \tilde{x} - x$ et $\delta_y = \tilde{y} - y$, ce qui permet de conserver une part de corrélation entre $\tilde{x}$ et x et entre $\tilde{y}$ et y et donc d'obtenir de meilleurs encadrements.

Si l'on s'intéresse maintenant à l'erreur relative ϵ plutôt qu'à l'erreur absolue liée à la multiplication, la situation est encore plus favorable. En effet, si les expressions $\epsilon_x = \frac{\tilde{x}-x}{x}$ et $\epsilon_y = \frac{\tilde{y}-y}{y}$ ne sont pas corrélées entre elles, l'utilisation des opérateurs spécialisés décrits au paragraphe 3.3.2 va permettre une évaluation par intervalles optimale.

$$\epsilon = \frac{\tilde{x} \cdot \tilde{y} - x \cdot y}{x \cdot y} = \epsilon_x + \epsilon_y + \epsilon_x \cdot \epsilon_y$$

Chacune de ces réécritures ramène donc l'évaluation d'une expression complexe à l'évaluation de plusieurs expressions plus simples, tout en essayant de ne pas trop perdre de corrélation, contrairement à ce qu'aurait provoqué une utilisation directe de l'arithmétique d'intervalles. Ces expressions plus simples sont elles aussi des expressions d'erreur et peuvent à leur tour être réécrites, et cela jusqu'à atteindre des expressions d'erreur dont les encadrements sont connus.

Ces méthodes s'appliquent ainsi récursivement mais elles nécessitent que les membres $\tilde{e}$ et e de l'expression à encadrer aient une structure similaire et que les positions relatives de leurs sous-termes correspondent. Dans le cas contraire, on se retrouve à borner l'erreur entre des sous-expressions sans rapport, ce qui augmente d'autant la décorrélation. Par exemple, lors de la réécriture de $\tilde{y} \cdot \tilde{x} - x \cdot y$, les termes $\tilde{y} - x$ et $\tilde{x} - y$ apparaissent, ce qui ne présente aucun intérêt lors de l'évaluation.

Les réécritures précédentes concernent la multiplication ; les suivantes s'appliquent à l'erreur absolue de l'addition et de la soustraction et à l'erreur relative de la division :

$$\begin{aligned}(\tilde{x} + \tilde{y}) - (x + y) &= (\tilde{x} - x) + (\tilde{y} - y) \\ (\tilde{x} - \tilde{y}) - (x - y) &= (\tilde{x} - x) - (\tilde{y} - y) \\ \frac{\tilde{x}/\tilde{y} - x/y}{x/y} &= \frac{\epsilon_x - \epsilon_y}{1 + \epsilon_y}\end{aligned}$$

3.2.2 À partir d'expressions intermédiaires

Dans le cas particulier où l'erreur entre x et y est connue, on peut profiter de la proximité de $\tilde{x}$ et x pour borner l'expression $\tilde{x} - y$ en la réécrivant en $(\tilde{x} - x) + (x - y)$. De façon analogue, on peut chercher à faire intervenir le terme $\tilde{y}$ proche de y en réécrivant $\tilde{x} - y$ en $(\tilde{x} - \tilde{y}) + (\tilde{y} - y)$.

De telles réécritures existent aussi dans le cadre de l'erreur relative. L'identité suivante permet en effet de faire apparaître des termes intermédiaires à condition que certaines expressions ne valent pas zéro.

$$\forall a, b, c \in \mathbb{R}, \quad a \neq 0 \wedge b \neq 0 \quad \Rightarrow \quad \frac{c-a}{a} = \frac{c-b}{b} + \frac{b-a}{a} + \frac{c-b}{b} \cdot \frac{b-a}{a}$$

Pour les réécritures qui s'appuient sur la structure syntaxique, toute l'information nécessaire pour les appliquer se trouve dans l'expression manipulée. Ce n'est plus le cas ici : seuls $\tilde{x}$ et y sont présents et il faut pouvoir en déduire x ou $\tilde{y}$. Le paragraphe 5.4.2 présente les heuristiques mises en œuvre par Gappa pour construire ces expressions.

3.2.3 Autres réécritures

Même si l'utilisateur soumet une proposition ne contenant que des expressions d'erreur, Gappa aura besoin d'encadrer des expressions qui ne sont pas de cette forme et auxquelles les règles précédentes ne sont pas adaptées. L'évaluation de ces expressions peut cependant bénéficier de réécritures pour y faire apparaître des termes d'erreur. L'expression $(\tilde{e} - e) + e$ peut par exemple conduire à des encadrements plus fins que si l'on avait directement cherché à évaluer $\tilde{e}$. Et réciproquement, si l'expression à encadrer est e, la réécrire en $\tilde{e} - (\tilde{e} - e)$ peut conduire à de meilleurs encadrements.

De façon analogue, il est possible de faire intervenir l'erreur relative plutôt que l'erreur absolue pour encadrer $\tilde{e}$ ou e :

$$\begin{aligned} \tilde{e} &= e \cdot (1 + \epsilon_e) \\ e &= \frac{\tilde{e}}{1 + \epsilon_e} \end{aligned}$$

Il peut aussi arriver qu'une expression approchée apparaisse au sein d'une expression plus complexe. L'expression $\tilde{a} \cdot b$ peut ainsi être réécrite $(\tilde{a} - a) \cdot b + a \cdot b$ afin de faire disparaître le terme $\tilde{a}$ au profit de a et de l'erreur absolue $\tilde{a} - a$. Ces réécritures ne sont pas vraiment nécessaires mais elles peuvent parfois simplifier la vie de l'utilisateur et éviter qu'il n'ait à les définir lui-même. Pour ne pas que le nombre d'expressions manipulées explose, Gappa ne cherche les termes approchés que parmi les opérandes de plus haut niveau de l'expression à encadrer. Les autres termes approchés seront découverts au cours de l'encadrement par intervalles des sous-termes d'une expression.

3.3 Amélioration d'encadrements

À l'aide de la bissection et de réécritures, il est ainsi possible de réduire l'influence des phénomènes de décorrélation qui surgissent lors de l'évaluation par intervalles. Une autre approche est d'enrichir l'arithmétique d'intervalles et son évaluation au moyen de nouveaux théorèmes afin de prendre en compte certaines corrélations.

3.3.1 Dérivation d'expressions

Lorsque l'on considère l'erreur $\tilde{e} - e$, les expressions $\tilde{e}$ et e sont proches, mais il est aussi possible que leurs variations soient similaires. La corrélation entre les expressions $\tilde{e}$ et e existe alors aussi au niveau de leurs dérivées. Cependant, il peut arriver que l'évaluation de la dérivée par intervalles subisse une décorrélation plus faible, par exemple parce que les intervalles manipulés sont moins larges. C'est le cas pour $x - x$, l'expression dérivée est $1 - 1$ dont l'évaluation par intervalles ne pose aucune difficulté.

Pour mettre à profit la dérivée f' de $f = \tilde{e} - e$ par rapport à une variable x, on peut employer les inégalités des accroissements finis. Au lieu d'encadrer l'expression sur tout un intervalle X, elle n'a besoin d'être encadrée qu'en un seul point $x_0 \in X$. C'est sa dérivée qui est encadrée sur tout l'intervalle. À l'aide d'arithmétique par intervalles, ces encadrements s'expriment par

$$f(X) \subseteq F([x_0]) + (X - [x_0]) \cdot F'(X).$$

Dans cette formule, F est une fonction qui associe à tout intervalle U un intervalle contenant les éléments de l'image $f(U)$ au moins. La fonction d'intervalles F' vérifie la même propriété vis-à-vis de la dérivée f'. En prenant x_0 vers le milieu

de X, on obtient ainsi un intervalle englobant $f(X)$ et dont la largeur est de l'ordre de $w(X) \cdot w(|F'(X)|)$. Le théorème de Taylor par intervalles permet de généraliser cette méthode à un degré quelconque :

$$f(X) \subseteq \sum_{i=0}^{n-1} \frac{(X - [x_0])^i}{i!} \cdot F^{(i)}([x_0]) + \frac{(X - [x_0])^n}{n!} \cdot F^{(n)}(X).$$

En collaboration avec César Muñoz, nous avons employé cette méthode pour borner une erreur de troncature [DMM05]. Nous avions une fonction $r(\phi)$ exprimée à l'aide de fonctions trigonométriques et une approximation polynomiale $\tilde{r}(\phi)$. L'objectif était de vérifier la qualité de cette approximation, c'est-à-dire de borner l'expression $\frac{\tilde{r}-r}{r}$ sur l'intervalle $\phi \in [0, \frac{4}{9}\pi]$. Pour prouver que l'erreur est de l'ordre de 10^{-6} à l'aide d'une évaluation naïve par intervalles, il faudrait découper l'intervalle de départ en plusieurs millions de sous-intervalles à cause de la décorrélation.

En utilisant un développement au premier ordre du numérateur $\tilde{r}(\phi) - r(\phi)$, nous avons pu limiter le nombre de sous-intervalles à moins de dix mille et ainsi accroître la vitesse de vérification. Cette vérification a été faite en PVS avec une arithmétique d'intervalles à bornes rationnelles [ML05]. L'assistant PVS a permis de prouver formellement que le théorème de Taylor par intervalles permet effectivement d'obtenir un encadrement d'expression, puis il a été utilisé pour vérifier l'encadrement de $\frac{\tilde{r}-r}{r}$.

3.3.2 Nouveaux opérateurs arithmétiques

Traditionnellement, seuls les opérateurs arithmétiques courants sont étendus en opérateurs sur les intervalles. Addition, multiplication, division, racine carrée, etc. constituent en effet les bases à partir desquelles on peut construire les autres expressions arithmétiques. Les corrélations entre expressions limitent cependant leur utilité pour évaluer des expressions plus complexes. J'ai donc introduit des fonctions d'intervalles spécifiques pour des expressions complexes qui reviennent couramment dans les propositions que Gappa traite.

Évaluer $(a, b) \mapsto a + b + a \cdot b$

L'expression $a + b + a \cdot b$ apparaît quand Gappa réécrit des expressions pour y faire apparaître des termes d'erreur. Elle exprime par exemple l'erreur relative de la multiplication :

$$\frac{\tilde{x} \cdot \tilde{y} - x \cdot y}{x \cdot y} = \epsilon_x + \epsilon_y + \epsilon_x \cdot \epsilon_y$$

Elle permet aussi d'exprimer l'erreur relative entre des expressions z et x en passant par les intermédiaires $\epsilon_x = \frac{y-x}{x}$ et $\epsilon_y = \frac{z-y}{y}$:

$$\frac{z - x}{x} = \epsilon_x + \epsilon_y + \epsilon_x \cdot \epsilon_y$$

L'expression $a+b+a\cdot b$ contient plusieurs occurrences de a et b et souffre par conséquent d'une corrélation qui serait perdue lors d'une évaluation par intervalles. Par exemple, avec les hypothèses $|a| \leq \frac{1}{2}$ et $|b| \leq \frac{1}{2}$, on obtiendrait l'encadrement $a+b+a\cdot b \in [-\frac{5}{4}, \frac{5}{4}]$ alors que l'intervalle optimal est en réalité $[-\frac{3}{4}, \frac{5}{4}]$. Un autre inconvénient de cette expression est que le produit $a \cdot b$ implique vraisemblablement des intervalles contenant zéro puisque a et b représentent des erreurs. Comme indiqué au paragraphe 6.3, la vérification d'un tel produit est alors coûteuse.

L'intervalle optimal peut s'obtenir de diverses façons, par exemple en évaluant l'expression équivalente $(1+a)\cdot(1+b)-1$. Cette nouvelle forme ne fait intervenir les termes a et b qu'une fois chacun. Elle ne contient donc aucune corrélation autre que celle qui existe potentiellement entre a et b. C'est donc cette forme que l'on souhaiterait évaluer par intervalles. Malheureusement elle exige une précision élevée pour calculer les bornes. En effet, a et b représentent des erreurs d'arrondi et vont donc prendre des valeurs très petites. Elles peuvent par exemple être bien inférieures à 2^{-100} en quadruple précision. La simple évaluation de $1+a$ et $1+b$ nécessiterait alors plus de 100 bits de précision pour que les ordres de grandeur de a et b soient conservés. Cette forme produit donc un résultat un peu plus fidèle mais bien trop coûteux par rapport à l'évaluation directe de $a+b+a\cdot b$.

Regardons ce qui est effectivement calculé lors de l'évaluation naïve par intervalles de $(1+a)\cdot(1+b)-1$. Pour simplifier, supposons que les calculs s'effectuent sur des intervalles à bornes réelles et qu'il n'y a pas besoin d'arrondir ces bornes. Les expressions a et b prennent leurs valeurs dans les intervalles $A = [\underline{a}, \overline{a}]$ et $B = [\underline{b}, \overline{b}]$. Fixons en plus les contraintes : $\underline{a} \geq -1$ et $\underline{b} \geq -1$. Comme a et b représentent des erreurs relatives, ce n'est pas très limitant : avoir une erreur relative plus grande que 1 en valeur absolue n'a en effet pas grand intérêt. Les valeurs de $1+a$ et $1+b$ sont donc positives ou nulles. Il en découle successivement les deux relations d'appartenance suivantes :

$$\begin{aligned} (1+a)\cdot(1+b)-1 &\in [(1+\underline{a})\cdot(1+\underline{b})-1, (1+\overline{a})\cdot(1+\overline{b})-1] \\ a+b+a\cdot b &\in [\underline{a}+\underline{b}+\underline{a}\cdot\underline{b}, \overline{a}+\overline{b}+\overline{a}\cdot\overline{b}] \end{aligned}$$

Cette dernière relation d'appartenance permet de définir un nouvel opérateur binaire sur les intervalles. Il s'applique à condition que les intervalles A et B ne contiennent que des valeurs égales ou plus grandes que -1.

Évaluer $(x, y, a, b) \mapsto \frac{a\cdot x+b\cdot y}{x+y}$

Cette expression apparaît moins souvent que celle présentée au paragraphe précédent. Elle est utile pour évaluer l'erreur relative associée à une addition ou à une soustraction. Par exemple, pour l'erreur entre $\tilde{x}+\tilde{y}$ et $x+y$, on a :

$$\epsilon = \frac{(\tilde{x}+\tilde{y})-(x+y)}{x+y} = \frac{\epsilon_x \cdot x + \epsilon_y \cdot y}{x+y}$$

Si jamais x et y sont de même signe ou que l'un domine l'autre, l'erreur relative calculée sera très pessimiste. Comme nous parlons d'erreur relative, nous

pouvons considérer que les intervalles X et Y encadrant les expressions x et y ne contiennent pas zéro. Par conséquent, pour a et b fixés, les minimum et maximum de la fonction homographique $(x, y) \mapsto \frac{a \cdot x + b \cdot y}{x+y}$ sur le domaine rectangulaire $X \times Y$ sont nécessairement atteints aux sommets de ce domaine. Les dérivées partielles sont en effet de signe constant sur l'ensemble du domaine :

$$\frac{\partial f}{\partial x} = \frac{y \cdot (a-b)}{(x+y)^2} \quad \text{et} \quad \frac{\partial f}{\partial y} = \frac{x \cdot (b-a)}{(x+y)^2}$$

Les valeurs de a et b n'ont maintenant plus besoin d'être fixées. La valeur de l'expression complète est ainsi contenue dans l'intervalle qui est l'enveloppe convexe des quatre intervalles résultant du calcul de $\frac{x \cdot A + y \cdot B}{x+y}$ aux quatre sommets du domaine $X \times Y$. Qui plus est, les bornes de cet intervalle étant atteintes, il s'agit du résultat optimal.

Ce nouvel opérateur donne donc un intervalle optimal. Pour ce qui est des encadrements d'expression, le prédicat obtenu le sera sous deux conditions. D'une part les expressions a et b ne doivent pas être corrélées. Comme a et b seront des erreurs relatives, ce n'est pas une supposition trop contraignante. Il faut d'autre part que le domaine des valeurs effectivement prises par le couple (x, y) atteigne les quatre sommets du rectangle $X \times Y$, ce qui est généralement une façon détournée de dire que les expressions x et y ne sont pas corrélées.

Script Gappa 3.1 Erreur relative d'une somme

```
{ |(xx - x) / x| <= 0.1 /\
  |(yy - y) / y| <= 0.1 /\
  x in [50,1000] /\ y in [-25,-1] ->
  ((xx + yy) - (x + y)) / (x + y) in ? }
```

Considérons le script Gappa 3.1. Il calcule l'erreur relative qui existe entre les sommes $x' + y'$ et $x + y$ sachant que les erreurs relatives $\frac{x'-x}{x}$ et $\frac{y'-y}{y}$ sont connues. Sans l'aide de l'opérateur homographique, Gappa borne l'erreur relative globale par 4.1. Cette borne étant supérieure à 1, elle n'apporte absolument aucune information utile sur l'erreur. Avec l'aide de l'opérateur homographique, l'outil répond 0.3. Cette borne est optimale puisqu'atteinte pour les valeurs $x = 50$, $x' = 55$, $y = -25$, $y' = -22.5$.

Cet opérateur est très utile aux utilisateurs qui manipulent des erreurs relatives dans leurs scripts Gappa. Il leur évite d'avoir à indiquer de nombreuses règles de réécritures pour traiter les additions. Il s'agit cependant du seul théorème concernant l'arithmétique réelle qui n'a pas été prouvé en Coq. Nous avons jugé préférable d'attendre les prochaines évolutions de la bibliothèque Coq d'analyse réelle avant de s'attaquer à sa preuve.

Quelques expressions simples

La structure des expressions précédentes est relativement complexe. Il en existe de plus simples pour lesquelles la prise en compte d'une corrélation diminue considérablement la taille des intervalles. C'est le cas de la différence $x-x$ et du quotient $\frac{x}{x}$. L'intervalle encadrant l'expression $x - x$ est $[0, 0]$. Celui encadrant $\frac{x}{x}$ est $[1, 1]$ à condition que l'intervalle qui encadre $|x|$ ne contienne pas zéro.

Pour $x - x$ et $\frac{x}{x}$, le même genre d'optimisation pourrait s'obtenir directement par réécriture. Moins triviale est cependant l'élévation au carré $x \cdot x$. Là encore il y a un phénomène de décorrélation. L'encadrement obtenu en effectuant le produit $X \times X$ est en effet trop pessimiste : il peut contenir des valeurs négatives si l'intervalle X contient à la fois des valeurs positives et négatives. L'intervalle optimal s'obtient en éliminant les valeurs négatives de $X \times X$. Une réécriture en $|x \cdot x|$ conduirait aussi à un intervalle optimal, mais au prix d'une preuve plus longue.

De tels théorèmes sont en réalité nécessaires pour toutes les expressions de la forme x^n, mais seul le plus courant, c'est-à-dire celui pour $n = 2$, est implanté dans Gappa. Les autres pourront être ajoutés au cas par cas en fonction des besoins.

3.3.3 La question de la valeur absolue

Considérons l'encadrement $|e| \in [2, 5]$. Il signifie que la valeur absolue de l'expression e est comprise entre 2 et 5. Mais si l'on regarde maintenant l'expression elle-même, cet encadrement signifie que la valeur de e est comprise dans les intervalles $[-5, -2]$ ou $[2, 5]$. Autrement dit, e appartient à l'intervalle $[-5, 5]$ mais n'appartient pas à $]-2, 2[$. L'utilisation de la valeur absolue permet donc d'exclure des portions d'intervalles du domaine de certaines expressions. C'est utile pour appliquer un théorème dont le domaine de validité n'est pas connexe.

Exemples de domaines non connexes

C'est par exemple le cas pour l'erreur relative commise lors de l'arrondi d'un nombre réel en un nombre flottant. L'encadrement traditionnel E_0 de cette erreur relative $\epsilon = \frac{\circ(e)-e}{e}$ n'est en effet valide que si la valeur de e n'appartient pas au domaine D_0 des nombres dénormalisés. La contrainte pour appliquer le théorème pourrait donc être : avoir $e \in I$ avec $I \cap D_0 = \emptyset$. Mais supposons que l'hypothèse dont on dispose soit en fait $|e| \in [2, 5]$. On peut en déduire une hypothèse $e \in [-5, 5]$ mais la contrainte du théorème n'est alors pas respectée puisque D_0 contient des valeurs proches de zéro.

Le théorème reste cependant vrai si l'on modifie la contrainte en $|e| \in I$ avec $I \cap |D_0| = \emptyset$. Si D_0 est symétrique par rapport à zéro, cette nouvelle contrainte permet de prouver un plus grand nombre de propositions. Le seul risque est l'allongement de la preuve d'une étape s'il faut calculer un encadrement de $|e|$ à partir d'un encadrement de e.

Cette modification des contraintes ne se cantonne pas à l'erreur relative d'une opération flottante. Elle s'applique à toute réécriture impliquant un quotient. Ainsi,

pour pouvoir profiter de l'égalité $e = 1/\frac{1}{e}$, il faut disposer d'une hypothèse affirmant que e ne vaut pas zéro. Là encore, la contrainte est plus intéressante quand elle est exprimée à l'aide de la valeur absolue de l'expression : $|e| \in I$ avec $0 \notin I$.

Calculer avec des valeurs absolues

Il est donc intéressant d'avoir des contraintes sur les encadrements de valeurs absolues. L'arithmétique d'intervalles n'est malheureusement pas efficace ici. Considérons par exemple que les hypothèses soient $|x| \in [2, 5]$ et $|y| \in [0, 1]$ et qu'il faille prouver $x + y \neq 0$.

Cela signifie qu'il faut construire une hypothèse de la forme $|x + y| \in I$ avec $0 \notin I$. Les seules propriétés intéressantes qui peuvent être ajoutées sont $x \in [-5, 5]$ et $y \in [-1, 1]$. Par arithmétique d'intervalles, on peut alors en déduire $|x + y| \in [0, 6]$. Cette propriété n'est cependant pas suffisante. En faisant une trisection sur x entre les intervalles $[-5, -1.5]$, $[-1.5, 1.5]$ et $[1.5, 5]$, on obtient une contradiction sur le domaine central et la propriété $|x+y| \in [0.5, 6]$ sur les deux autres domaines. Ce qui prouve donc la validité de l'hypothèse $|x + y| \in [0.5, 6]$, mais au prix d'une trisection.

Afin d'éviter cette trisection et de réduire la taille de la preuve, il convient de mettre directement en relation l'encadrement de $|x + y|$ avec ceux de $|x|$ et $|y|$ plutôt que de passer par ceux de x et y. Les inégalités triangulaire fournissent les inégalités $|x| - |y| \leq |x + y| \leq |x| + |y|$. Par simples addition et soustraction d'intervalles, on prouve que les expressions $|x| - |y|$ et $|x| + |y|$ prennent leurs valeurs dans les intervalles $[1, 5]$ et $[2, 6]$ respectivement. On en déduit alors $|x + y| \in [1, 6]$. Ce dernier encadrement satisfait la contrainte $0 \notin [1, 6]$ et on a donc prouvé $x + y \neq 0$ sans avoir eu besoin de considérer des sous-domaines.

3.3.4 Représentations améliorées

Jusqu'à présent, les expressions n'étaient encadrées qu'à l'aide d'un intervalle à bornes constantes. Afin de conserver plus d'informations de corrélation, on pourrait chercher à employer une représentation plus riche des encadrements. Les encadrements dépendraient d'expressions non constantes sur l'ensemble du domaine défini par les hypothèses. Même sans aller jusqu'aux séries formelles de Fluctuat [PGM04], voici deux représentations qui pourraient aider. Contrairement aux améliorations décrites précédemment, celles ci-après sont seulement à l'état de perspectives, elles n'ont pas été implantées dans Gappa.

Représentation affine

Les plus gros problèmes apparaissent quand on cherche à borner l'écart entre deux termes proches $\tilde{x}$ et x. Les encadrements de ces termes n'ont en effet plus aucune information concernant leur corrélation. Supposons maintenant que $\tilde{x}$ soit encadré en utilisant deux intervalles X_r et X_a : $\tilde{x} \in x \cdot (1 + X_r) + X_a$. Sous une

notation moins condensée, le prédicat est donc devenu :

$$\exists \epsilon \in X_r \quad \exists \delta \in X_a \quad \tilde{x} = x \cdot (1 + \epsilon) + \delta.$$

Un encadrement n'est donc plus un simple intervalle constant mais un triplet composé d'une expression et deux intervalles. Si les valeurs de l'expression x sont contenues dans l'intervalle X, il est alors facile d'en déduire un intervalle contenant la distance $\tilde{x} - x$ entre les deux termes : $X \cdot X_r + X_a$. Quant à l'erreur relative, elle est contenue dans l'intervalle $X_r + X_a/X$.

La composition d'erreurs ne pose pas plus de difficulté. Si les expressions x, y et z sont reliées par les encadrements $y \in x \cdot (1 + X_r) + X_a$ et $z \in y \cdot (1 + Y_r) + Y_a$ alors

$$z \in x \cdot (1 + (X_r + Y_r + X_r \cdot Y_r)) + (X_a + Y_a + X_a \cdot Y_r).$$

Dans la formule précédente, l'intervalle $X_r + Y_r + X_r \cdot Y_r$ devrait plutôt être évalué à l'aide de l'opérateur décrit au paragraphe 3.3.2. Des formules de complexité similaire permettent d'effectuer l'addition et la multiplication de deux encadrements $\tilde{x} \in x \cdot (1 + X_r) + X_a$ et $\tilde{y} \in y \cdot (1 + Y_r) + Y_a$. Par exemple, pour l'addition, on obtiendrait la formule suivante (l'autre opérateur décrit au paragraphe 3.3.2 serait ici indispensable pour limiter la décorrélation) :

$$\tilde{x} + \tilde{y} \in (x + y) \cdot \left(1 + \frac{X \cdot X_r + Y \cdot Y_r}{X + Y}\right) + (X_a + Y_a).$$

L'avantage de cette représentation est que les réécritures présentées au paragraphe 3.2 deviennent inutiles pour des propagations directes de l'erreur : la représentation des encadrements fournit toutes les réponses. Qui plus est, il n'y a pas besoin de se préoccuper de prouver que des termes sont non nuls, contrairement à ce que l'on devrait faire si l'on manipulait directement des erreurs relatives.

Dans le cadre des arithmétiques approchées à virgule flottante, les erreurs relatives d'arrondi sont perturbées par la présence non seulement de zéro mais aussi des nombres dénormalisés. Avec la représentation décrite ici, le problème ne se pose pas : il suffit de borner la petite erreur absolue commise dans le domaine des dénormalisés. Si l'on borne par ϵ_0 l'erreur relative en dehors des dénormalisés et par η_0 sur le domaine des dénormalisés [Dem84], on obtient l'encadrement suivant :

$$\circ(x) \in x \cdot (1 + [-\epsilon_0, \epsilon_0]) + [-\eta_0, \eta_0].$$

Si la dernière opération flottante effectuée est une addition ou une soustraction, cette représentation permet d'obtenir un résultat encore plus élégant : $x \oplus y$ est alors encadré par $(x + y) \cdot (1 + [-\epsilon_0, \epsilon_0])$. En effet, si $x \oplus y$ est un nombre dénormalisé, il vaut nécessairement $x + y$ et l'erreur commise est alors nulle.

Cette représentation améliorée présente cependant l'inconvénient de sa bien plus grande complexité. Non seulement les théorèmes ont à manipuler des prédicats plus complexes et leur vérification est plus longue, mais en fait ils n'existent parfois

même pas. On se retrouve alors contraint de revenir à des prédicats d'encadrement simple et d'appliquer les théorèmes classiques d'arithmétique d'intervalles.

Une approche intermédiaire serait de ne pas utiliser systématiquement cette représentation améliorée. Ce serait juste un prédicat additionnel testé en parallèle du prédicat d'encadrement simple, au cas où il améliorerait les choses. Ce mécanisme de prédicat parallèle est déjà employé pour l'encadrement en valeur absolue présenté au paragraphe 5.2.3. Par conséquent, Gappa utiliserait généralement des encadrements simples associés à des réécritures. Mais en présence d'erreurs relatives et de nombres dénormalisés, l'outil pourrait se tourner vers les résultats fournis par ce prédicat amélioré.

Modèles de Taylor

L'utilisation de modèles de Taylor [MB03] s'apparente à la méthode décrite au paragraphe 3.3.1. Mais cette fois, au lieu de dériver formellement l'expression entière avant de l'évaluer, l'encadrement des diverses dérivées est construit au fur et à mesure de l'évaluation. Le prédicat d'encadrement est alors de la forme $e \in P(x) + I$ avec I un intervalle et P un polynôme à coefficients représentables, c'est-à-dire des nombres dyadiques dans notre cas.

L'arithmétique d'intervalles classique peut être considérée comme de l'arithmétique sur des modèles de Taylor dont le polynôme est nul. De la même façon que pour l'arithmétique d'intervalles, les opérateurs réels sont étendus aux modèles de Taylor. Ainsi, sachant que $a \in P(x) + I$ et $b \in Q(x) + J$, l'encadrement de la somme $a + b$ s'exprime

$$a + b \in (P + Q)(x) + (I + J).$$

Couplée aux réécritures d'expressions approchées, cette méthode permet même de gérer des expressions qui ne sont pas dérivables comme c'est le cas des fonctions contenant les opérateurs d'arrondi définis au chapitre 4. Considérons par exemple que l'on souhaite encadrer l'expression $\circ(e)$. Il suffit d'encadrer l'erreur absolue $\circ(e) - e$ par un intervalle J puis de réécrire l'expression $\circ(e)$ en $(\circ(e) - e) + e$. À supposer que l'expression e soit encadrée par $P(x) + I$, on obtiendra $\circ(e) \in P(x) + (I + J)$.

Comme pour la représentation affine, l'inconvénient est ici la plus grande complexité de cette représentation par rapport au simple intervalle et les difficultés que cela pose lors de la vérification des calculs. Qui plus est, il faut pouvoir décider quelle expression x sera choisie comme argument du polynôme. En effet, bien que l'on puisse construire des modèles de Taylor multivariés, il est préférable d'éviter la multiplication des indéterminées pour que les modèles puissent effectivement gérer les corrélations.

Chapitre 4

Arithmétiques approchées en machine

Afin de prendre en compte les phénomènes d'arrondi, de nouveaux opérateurs sont ajoutés. Ce chapitre décrit leurs particularités et les liens qu'ils entretiennent avec l'arithmétique d'intervalles. Il présente aussi des prédicats spécialement adaptés aux ensembles de nombres dyadiques.

Les chapitres précédents traitent d'une arithmétique idéale en précision infinie et montrent comment prouver des bornes d'expressions contenant les opérateurs arithmétiques de base sur $\mathbb{R}$. L'un des objectifs de Gappa est néanmoins d'aider à certifier des applications numériques. Il est indispensable pour cela de savoir exprimer les opérations arithmétiques qu'elles contiennent. La précision et le domaine limités des formats manipulés en machine ont en effet tendance à perturber les résultats calculés par rapport aux valeurs obtenues en arithmétique réelle.

En Gappa, ces nouvelles arithmétiques s'expriment à l'aide d'opérateurs d'arrondi. Des théorèmes exprimant les propriétés de ces opérateurs sont ensuite nécessaires pour que Gappa puisse les manipuler avec des intervalles. Cependant les intervalles ne sont pas tout à fait adaptés pour représenter des ensembles discrets de nombres machine, c'est pourquoi Gappa dispose aussi de prédicats permettant d'exprimer les propriétés de précision de ces nombres.

4.1 Opérateurs d'arrondi

Deux approches ont été successivement utilisées pour Gappa. La première, décrite au paragraphe 4.1.4, consiste à travailler directement sur les ensembles de nombres machine. La seconde, aujourd'hui utilisée dans l'outil, s'abstrait des problèmes de représentation en plongeant systématiquement les nombres machine

dans l'ensemble des nombres réels afin de pouvoir appliquer aussi souvent que possible les théorèmes d'arithmétique réelle. C'est cette seconde approche qui est décrite ci-après.

4.1.1 Opérateurs unaires

Même si les types numériques utilisés en matériel ne permettent pas de représenter tous les réels, un certain nombre de propriétés les rendent utilisables. Considérons deux nombres machine a et b. Si la somme réelle $a + b$ de ces deux nombres est représentable par un nombre machine, alors l'addition en machine produira généralement ce résultat. Si le nombre n'est en revanche pas représentable, le résultat du calcul sera généralement un nombre qui en est proche. Ces propriétés permettent d'utiliser les arithmétiques machine comme substitut d'une véritable arithmétique des nombres réels.

La norme IEEE-754 [IEE85] dicte ainsi que, en dehors des situations exceptionnelles, le résultat d'un calcul flottant s'obtient en calculant virtuellement la valeur réelle en précision infinie puis en l'arrondissant vers un nombre représentable en fonction du mode d'arrondi. C'est ce concept qui nous a conduit à introduire des opérateurs unaires d'arrondi. En effet, la somme approchée $a \oplus b$ va alors être égale à $\circ(a+b)$, avec $\circ$ la fonction de $\mathbb{R}$ dans $\mathbb{R}$ qui se charge de calculer le nombre machine correspondant au résultat en précision infinie.

Ainsi, au lieu d'avoir à définir tous les opérateurs arithmétiques approchés $\oplus$, $\ominus$, $\otimes$, $\oslash$, etc, il suffit de définir un unique opérateur d'arrondi $\circ$. Il en va de même pour les théorèmes. Les théorèmes d'arithmétique réelle n'ont besoin d'être prouvés qu'une seule fois. Il suffit ensuite de poser quelques théorèmes qui s'appliquent à un opérateur unaire et donc à toutes les opérations approchées de l'arithmétique correspondante. Un opérateur unaire différent est défini pour chaque arithmétique approchée. Le paragraphe 4.2 présente en détail quelques-uns des critères qui définissent un opérateur unaire : arithmétique à virgule fixe ou à virgule flottante, direction d'arrondi, précision du format de stockage, plus petite information représentable, etc.

4.1.2 Opérateurs généralisés

Les opérateurs unaires peuvent ainsi être utilisés pour émuler un opérateur arithmétique approché à partir du moment où le résultat de l'opération dépend entièrement de la valeur du résultat de l'opérateur réel exact. Même si ce comportement est préférable du point de vue de la qualité des calculs, pour des raisons de performances ce n'est pas nécessairement le choix qui est fait dans certaines implantations [Tex97].

Considérons l'exemple d'un multiplieur en virgule fixe qui prend en entrée des nombres sur 32 bits avec une partie fractionnaire sur 16 bits. Le résultat est renvoyé dans ce même format et il y a plusieurs façons de l'obtenir. Pour respecter la règle selon laquelle ce résultat arrondi dépend entièrement du produit exact, il faudrait

effectuer ce produit en totalité, ce qui nécessite un multiplieur entier $32 \times 32 \rightarrow 64$ bits. Si seul un multiplieur $32 \times 32 \rightarrow 32$ est disponible, une solution consiste à décaler chacune des entrées de 8 bits vers la droite, c'est-à-dire à diviser par 256 les entiers qui représentent les nombres à virgule fixe. Le produit de ces entiers représente une approximation du résultat réel dans le format initial.

Appliquons cet algorithme aux produits $(128 \cdot 2^{-16}) \otimes (512 \cdot 2^{-16})$ et $(256 \cdot 2^{-16}) \otimes (256 \cdot 2^{-16})$. Le résultat en précision infinie vaut dans les deux cas 2^{-16}. Mais la première multiplication répond $0 \cdot 2^{-16}$ tandis que la seconde répond $1 \cdot 2^{-16}$. L'utilisation d'un opérateur unaire n'est donc pas du tout adapté puisque $\circ(2^{-16})$ peut prendre plusieurs valeurs. Il est par conséquent indispensable d'encoder dans l'opération d'arrondi la valeur des entrées. Au lieu de représenter le produit $a \otimes b$ par $\circ(a \times b)$, il faut donc plutôt le représenter par $\circ_{\times}(a, b)$.

Le principe de ces opérateurs généralisés peut être poussé un peu plus loin. Le résultat d'une opération peut en effet dépendre, non seulement des valeurs de chacune des entrées, mais aussi de leur représentation en machine, voire du contexte dans lequel l'opération a lieu. En conséquence, il peut être nécessaire d'ajouter un paramètre aux opérateurs afin de différencier des opérations qui bien qu'elles prennent les mêmes arguments en entrée produiraient des valeurs différentes en sortie.

À chaque opérateur généralisé est associée une opération arithmétique réelle pour construire les énoncés des théorèmes le concernant. Ainsi, là où un théorème aurait fourni un moyen d'encadrer $\circ(a \diamond b) - a \diamond b$ à partir d'un encadrement de $a \diamond b$ pour un opérateur unaire, Gappa va chercher un théorème qui encadre $\circ_{\diamond}(a, b) - (a \diamond b)$ à partir d'un encadrement de $a \diamond b$ pour un opérateur généralisé. À partir de là, il n'y a donc plus aucune différence entre opérateur unaire et opérateur généralisé. La syntaxe à base d'opérateurs unaires sera donc utilisée par la suite pour simplifier les notations.

4.1.3 Limitations

Les opérateurs précédents sont conçus pour manipuler aussi souvent que possible des nombres réels plutôt que des nombres machine. Ils sont ainsi des fonctions de $\mathbb{R}^n$ dans $\mathbb{R}$. Seules les valeurs du format qui sont représentables dans $\mathbb{R}$ peuvent donc être manipulées et faire l'objet de théorèmes. Ainsi les formats flottants qui permettent de représenter des dépassements de capacité par des nombres se comportant comme $-\infty$ et $+\infty$ ne peuvent être traités que si ces nombres n'apparaissent jamais. Les théorèmes ne prennent jamais en compte ces situations exceptionnelles et se comportent comme si des nombres arbitrairement grands étaient représentables.

Un programme conçu pour être résistant aux dépassements de capacité ne peut donc pas être directement traité par Gappa. Il faudrait considérer, à part, les cas exceptionnels à l'aide d'un autre formalisme. Cependant, la majorité des programmes ont été écrits en supposant que seuls des nombres finis sont manipulés lors de leur exécution. Une telle propriété du programme peut par contre être vérifiée à l'aide

de Gappa. Il suffit en effet de demander à l'outil de prouver qu'aucun des résultats ne se trouve en dehors du domaine des nombres représentables, ce qui est un simple encadrement d'expressions.

Les infinis ne sont pas les seules valeurs qui échappent au formalisme utilisé par Gappa. Il n'est pas non plus possible de différencier des nombres différents représentant la même valeur réelle. C'est par exemple le cas des zéros signés en arithmétique flottante : pour Gappa, les nombres $+0$ et -0 de la norme IEEE-754 sont le même réel zéro. Le signe du zéro n'a heureusement une importance que dans les situations exceptionnelles que Gappa ne gère de toutes façons pas. Finalement le formalisme ne permet pas non plus de gérer des nombres qui n'ont pas de signification dans $\mathbb{R}$. C'est le cas des *Not-a-Number*, la racine carrée d'un nombre négatif par exemple.

4.1.4 Changer l'ensemble sous-jacent

Jusqu'à présent, nous avons considéré que l'ensemble de base est la droite réelle et que la proposition logique élémentaire est l'appartenance des valeurs d'une expression à un intervalle de $\mathbb{R}$. Cependant, le même formalisme décrit au chapitre 2 aurait pu être bâti en considérant un autre ensemble que $\mathbb{R}$. Ainsi, si l'on considère les nombres flottants, privés de la distinction entre -0 et $+0$ et des *Not-a-Number*, on obtient un ensemble totalement ordonné $\mathbb{F}$ sur lequel on peut construire des intervalles et l'arithmétique correspondante.

Un intervalle de flottants $[a, b]$ est le sous-ensemble de $\mathbb{F}$ vérifiant $\{x \in \mathbb{F} \mid a \leq x \leq b\}$. Considérons maintenant l'addition flottante $\oplus$. Elle vérifie des propriétés de monotonie similaire à l'addition réelle : $\forall u, v, x, y \in \mathbb{F},\ u \leq v \wedge x \leq y \ \Rightarrow u \oplus x \leq v \oplus y$. On peut donc aisément définir une addition d'intervalles de nombres flottants :

$$[a, b] \oplus [c, d] = [a \oplus c, b \oplus d]$$

Il est aisé de définir une arithmétique d'intervalles spécialisée pour les calculs flottants. C'est ce formalisme que nous avons élaboré lors de la première version de Gappa [DM04]. Il est adapté si l'on cherche à prouver des bornes sur des variables d'un programme. Il n'y a en effet pas besoin de passer par les nombres réels, toutes les propositions sont prouvées en restant dans le domaine des nombres machine. En particulier, cela permet de traiter des programmes qui génèrent des nombres infinis. Qui plus est, ce formalisme n'est pas sensible au phénomène de croissance des bornes décrit au paragraphe 2.3.2. La taille des nombres utilisés pour les bornes reste en effet contrainte au format des éléments de $\mathbb{F}$.

Autre avantage, ce formalisme fournit parfois des encadrements plus fins quand on cherche à quel intervalle appartient la variable d'un programme. Employer des opérateurs d'arrondi unaires peut en effet causer un phénomène de double arrondi[1] qui conduit à des encadrements pessimistes puisque deux calculs successifs vont être nécessaires pour obtenir celui de $x \oplus y = \circ(x + y)$. Supposons par exemple

[1] Un exemple plus simple de ce phénomène est décrit au paragraphe 8.1.1.

$2^{p-1} \leq x$ et $\frac{1}{2} + 2^{-k-1} \leq y$. L'entier p représente la précision de l'arrondi au plus près $\circ$, c'est-à-dire que $\circ(2^{p-1} + 0.6)$ vaut $2^{p-1} + 1$. L'entier $p + k$ représente quant à lui la précision utilisée en interne par Gappa pour ses arrondis vers le haut $\triangle$ et vers les bas $\triangledown$. En particulier, la meilleure borne inférieure sur $x + y$ que l'outil est capable de prouver est $2^{p-1} + \frac{1}{2} = \triangledown(2^{p-1} + \frac{1}{2} + 2^{-k-1}) \leq x + y$. En profitant de la monotonie de l'opérateur $\circ$, l'outil prouve finalement $2^{p-1} = \circ(2^{p-1} + \frac{1}{2}) \leq x \oplus y$. En calculant directement, Gappa aurait obtenu une meilleure borne : $2^{p-1} + 1 = 2^{p-1} \oplus (\frac{1}{2} + 2^{-k-1}) \leq x \oplus y$. Ce problème de double arrondi des bornes peut cependant être contourné dans les situations où des opérateurs arithmétiques de base (addition, soustraction, multiplication, division et racine carrée) sont appliqués à des valeurs arrondies. Il suffit en effet de demander à l'outil d'utiliser une précision interne doublée par rapport à p [Fig95].

L'emploi des nombres flottants comme ensemble sous-jacent présente un dernier avantage : l'arithmétique d'intervalles est simple à implanter si la machine supporte ce format, ce qui facilite et accélère la recherche et la génération de preuves. Dans le cas contraire, il va cependant falloir émuler tous les opérateurs de cette arithmétique. Cette contrainte se fait plus particulièrement sentir avec les outils de vérification de preuves. Il faut y définir une théorie représentant fidèlement tous les opérateurs arithmétiques de toutes les architectures pour lesquelles on souhaite certifier des applications.

En utilisant des opérateurs d'arrondi, il n'y a plus qu'un seul opérateur à définir par architecture, le reste des propriétés étant couvert par celles de l'arithmétique réelle. Comme il fallait de toutes façons développer une arithmétique d'intervalles adaptée à $\mathbb{R}$, nous avons bâti les versions suivantes de Gappa autour d'un formalisme à base d'opérateurs d'arrondi.

4.2 Représentations

Les opérateurs d'arrondi présentés précédemment permettent donc de représenter toute opération arithmétique effectuée en machine dès lors que son résultat est un nombre réel. Voyons maintenant ce qu'il en est plus précisément des formats et arithmétiques usuellement rencontrés.

4.2.1 Arithmétique à virgule fixe

Les nombres à virgule fixe sont généralement stockés dans des variables entières et un exposant virtuel leur est associé. Cet exposant n'a pas besoin d'être le même pour toutes les variables, mais pour une variable donnée, cet exposant est fixe. Soit v une variable stockant l'entier m et dont l'exposant associé est e. Cette variable représente le nombre réel $v = m \cdot 2^e$.

Les logiciels effectuant des calculs en virgule fixe contiennent généralement un grand nombre d'opérations de décalage, c'est-à-dire des multiplications ou divisions par des puissances de deux. Ces opérations n'intéressent pas Gappa. L'outil

travaille en effet directement sur les valeurs réelles représentées par les variables.

Considérons par exemple la ligne de programme `v3 = 8 * v1 + v2` et supposons que les exposants virtuels associés aux trois variables sont $e_1 = 1$, $e_2 = -2$ et $e_3 = -2$. Les entiers stockés vérifient l'équation $m_3 = 8 \cdot m_1 + m_2$. La valeur réelle représentée par le résultat est donc $v_3 = m_3 \cdot 2^{-2} = m_1 \cdot 2^1 + m_2 \cdot 2^{-2} = v_1 + v_2$. C'est cette somme $v_3 = v_1 + v_2$ qu'il convient d'exprimer dans la syntaxe Gappa. Elle correspond exactement à l'intention du développeur quand il a écrit la ligne de programme. Le fait que le contenu de `v1` soit décalé de trois bits vers la gauche pour l'aligner avec le contenu de `v2` n'est qu'un détail d'implantation et n'a pas d'incidence sur les valeurs réelles calculées.

Il se produit un arrondi quand une valeur $m_1 \cdot 2^{e_1}$ a besoin d'être stockée dans une variable `v2` associée à un exposant e_2 strictement supérieur à e_1. Cet arrondi est généralement provoqué par un décalage vers la droite. Mais là encore ce n'est pas ce qui est exprimé dans un script Gappa. Ce qui importe, c'est l'exposant e_2 et la façon dont la valeur représentée a été arrondie. Dans le cas d'un décalage vers la droite en complément à deux, cela correspond à un arrondi de la valeur représentée vers $-\infty$.

Les formats de nombres à virgule fixe sont donc représentés par l'exposant associé. Il s'agit aussi du poids réel du bit de poids faible de l'entier stocké. Cela ne suffit pas à définir une arithmétique à virgule fixe, il faut y ajouter la méthode employée quand le résultat d'un calcul n'est pas représentable et doit donc être arrondi. Au final, un programme utilisant la virgule fixe est donc décrit par la donnée de plusieurs opérateurs unaires dont les paramètres sont un exposant et une direction d'arrondi.

Accessoirement, ces opérateurs permettent aussi de décrire les opérations mathématiques de partie entière. Ainsi la partie entière supérieure $\lceil x \rceil$ peut s'exprimer `fixed<0,up>(x)` : le poids minimal est 2^0 et l'arrondi a lieu vers $+\infty$. Pour un poids minimal de zéro, le nom `int` peut aussi être employé. Par exemple, le script 4.1 demande à Gappa de prouver la propriété[2] suivante : si deux nombres réels x et y ont la même partie entière inférieure, alors ils sont séparés d'au plus 1.

Script Gappa 4.1 Conséquence de l'égalité des parties entières

```
{ int<dn>(x) - int<dn>(y) in [0,0] -> |x - y| <= 1 }
```

4.2.2 Arithmétique flottante

Gappa ne considère que les nombres flottants binaires, c'est-à-dire des nombres qui valent $m \cdot 2^e$ avec m et e deux entiers relatifs. D'autres types d'arithmétique seraient envisageables, tels les nombres flottants décimaux. Il suffit en effet de définir les opérateurs d'arrondi qui conviennent. Cette définition serait cependant

[2] Cet exemple est un cas difficile rencontré dans une procédure d'élimination des quantificateurs pour $(\mathbb{R}, <, +, \lfloor \cdot \rfloor, 0, 1)$ [Cha06].

plus complexe que pour les nombres flottants binaires puisque ceux-ci profitent de l'utilisation par Gappa de nombres dyadiques.

Gappa utilise une représentation des nombres flottants sous la forme $m \cdot 2^e$ avec $|m| < 2^p$ et $e \geq E$. Contrairement aux formats décrits dans la norme IEEE-754, un nombre flottant n'a donc généralement pas une représentation unique. Gappa n'a cependant jamais besoin de cette propriété. Qui plus est, cette représentation couvre naturellement la question des nombres dénormalisés [DRT01].

Un format de nombres flottants est donc caractérisé par sa précision p et son exposant minimal E. Pour décrire les opérateurs d'arrondi, il faut ajouter la direction d'arrondi à utiliser pour les résultats non représentables, comme c'était déjà le cas pour les arithmétiques à virgule fixe. Pour les formats simple et double précision de la norme IEEE-754, ainsi que pour le futur format quadruple précision, cela donne pour une direction `dir` :

simple	`float<24, -149, dir>`
double	`float<53, -1074, dir>`
quadruple	`float<113, -16494, dir>`

4.2.3 Directions d'arrondi

Les opérateurs d'arrondi présentés dans les paragraphes précédents décrivent non seulement le format de destination, mais aussi la direction dans laquelle a lieu l'arrondi quand le nombre à stocker n'est pas exactement représentable. Gappa supporte onze directions différentes, dont six sont des arrondis au plus près.

Ces directions peuvent être décomposées en fonction du signe des valeurs à arrondir. Par exemple, un arrondi vers $+\infty$ peut être considéré comme un arrondi en direction de zéro sur les réels négatifs et un arrondi vers l'infini sur les réels positifs. Ces deux autres arrondis ont la particularité d'être symétriques, c'est-à-dire qu'ils vérifient $\circ(-x) = -\circ(x)$. Il suffit donc à Gappa de considérer sept briques de base qui correspondent à diverses façons d'arrondir un nombre réel positif.

Pour effectuer l'arrondi d'une borne d'intervalle, c'est-à-dire d'un nombre dyadique, Gappa emploie des méthodes à base de bits r et s [EL04]. Ainsi, étant donné un nombre x, l'outil va d'abord l'arrondir vers zéro à la précision du format de destination. Cela s'effectue en tronquant la mantisse à la bonne taille. Si cela ne modifie pas le nombre, c'est qu'il est déjà représentable. Par contre, s'il est modifié, Gappa sélectionne, soit ce nombre tronqué, soit son successeur, comme arrondi du nombre positif initial. Tous les modes d'arrondi considérés ici ont en effet la propriété d'arrondir un nombre réel à l'un des deux nombres représentables qui l'encadrent.

Afin de choisir entre ces deux nombres représentables, l'outil calcule deux booléens lors de la troncature de x : un bit d'arrondi r et un bit *sticky* s. Si les n bits de poids faible de la mantisse sont tronqués, alors s est la disjonction des valeurs des $n-1$ bits de poids faibles, tandis que r a la valeur du dernier bit tronqué. Ces

deux bits r et s permettent de qualifier précisément les distances relatives de x par rapport au nombre tronqué et son successeur. Quand le choix se fait sur la parité de la mantisse tronquée, le booléen o représente la valeur du bit de poids faible de la mantisse tronquée.

Dans le tableau suivant, la troisième colonne indique la condition que Gappa vérifie pour choisir le successeur plutôt que le nombre tronqué. Dans les cas d'arrondi au plus près, le texte entre parenthèses dans la deuxième colonne indique le choix effectué lorsque x est à égale distance des deux nombres représentables les plus proches.

`zr`	vers zéro	jamais
`aw`	vers ∞	$r \vee s$
`od`	vers les mantisses impaires	$(r \vee s) \wedge \neg o$
`ne`	au plus près (mantisses paires)	$r \wedge (s \vee o)$
`no`	au plus près (mantisses impaires)	$r \wedge (s \vee \neg o)$
`nz`	au plus près (vers zéro)	$r \wedge s$
`na`	au plus près (vers ∞)	r

Les briques `zr`, `aw` et `ne` permettent d'exprimer les quatre modes d'arrondi décrits par la norme IEEE-754. Ces modes peuvent aussi être employés pour la virgule fixe. Les briques `nz` et `na` sont plus spécifiques à la virgule fixe. Les modes correspondants simplifient la description d'arithmétiques à virgule fixe qui ajoutent systématiquement $\frac{1}{2}$ avant d'effectuer un décalage vers la droite. La brique `od` est quant à elle utilisée pour l'arrondi impair décrit dans [BM05].

4.3 Théorèmes associés

Afin de pouvoir encadrer des expressions contenant des opérateurs d'arrondi, Gappa s'appuie sur des théorèmes lui permettant de borner des expressions de la forme $\circ(e)$, $\circ(e) - e$ et $\frac{\circ(e)-e}{e}$. Les autres types d'expression sont ramenés à ces trois formes par le biais de réécritures.

À chaque opérateur d'arrondi est associé un ensemble de théorèmes. Le tableau suivant indique quels types de conclusion peuvent être obtenus en fonction des types d'encadrements en hypothèse. Le cas singulier où Gappa se sert d'un encadrement de $\circ(e)$ pour en déduire un nouvel encadrement de $\circ(e)$ est expliqué en détail au paragraphe 4.3.1.

Hypothèse	Conclusion
$e \in I$	$\circ(e) \in J$
$\circ(e) \in I$	$\circ(e) \in J$
aucune	$\circ(e) - e \in J$
$e \in I$	$\circ(e) - e \in J$
$\circ(e) \in I$	$\circ(e) - e \in J$
$\lvert e \rvert \in I$	$\circ(e) - e \in J$
$\lvert \circ (e) \rvert \in I$	$\circ(e) - e \in J$
$e \in I$	$\frac{\circ(e)-e}{e} \in J$
$\circ(e) \in I$	$\frac{\circ(e)-e}{e} \in J$
$\lvert e \rvert \in I$	$\frac{\circ(e)-e}{e} \in J$
$\lvert \circ(e) \rvert \in I$	$\frac{\circ(e)-e}{e} \in J$

Les théorèmes utilisant un encadrement de e en hypothèse sont généralement capables de fournir des résultats plus précis concernant les erreurs que ceux utilisant $\circ(e)$. Cependant, un encadrement de e peut ne pas être disponible alors qu'un encadrement de $\circ(e)$ peut l'être, d'où l'intérêt de fournir deux versions. De même, un théorème s'appuyant sur des encadrements en valeur absolue peut parfois être utilisé là où les encadrements normaux ne permettent pas de conclure. Les variantes des théorèmes ont été retenues en fonction des propriétés de l'erreur d'arrondi de l'opérateur auquel ils s'appliquent.

4.3.1 Bornes de valeurs arrondies

Du point de vue de l'arithmétique d'intervalles, un opérateur d'arrondi ne présente pas de différence fondamentale avec les opérations arithmétiques élémentaires. C'est une fonction de $\mathbb{R}$ dans $\mathbb{R}$ et elle peut donc être étendue aux intervalles. Qui plus est, elle présente généralement l'avantage d'être croissante, ce qui simplifie la définition de son extension aux intervalles : $\circ([a, b]) = [\circ(a), \circ(b)]$.

Le théorème qui calcule un encadrement de $\circ(e)$ en fonction d'un encadrement de $\circ(e)$ n'est pas inutile : l'objectif est de réduire la taille d'un intervalle encadrant une expression approchée. En particulier, si un tel intervalle ne contient qu'une seule valeur approchée, il est très intéressant de réduire l'intervalle à cette unique valeur. De même, s'il ne contient aucune valeur, la preuve de la proposition courante est achevée. Ces réductions se produisent par exemple lors d'une bissection : à partir d'un certain nombre de subdivisions, les sous-intervalles considérés ne contiendront plus qu'une unique valeur approchée, voire aucune.

À ce moment-là, soit il y a une contradiction entre les hypothèses, soit les calculs sont effectués sur des intervalles points, c'est-à-dire sans aucun problème de décorrélation. Considérons que les calculs soient effectués en arithmétique flottante simple précision. Pour x entre 0 et 1, l'erreur absolue commise lors du calcul de $\circ(2 \cdot \sqrt{x} - 1)$ est bornée par 2^{-25}. Pour x au delà de 1, les théorèmes généraux sur l'erreur absolue de l'arrondi flottant ne fournissent pas de borne meilleure que 2^{-24}, ce qui est insuffisant pour prouver la proposition du script 4.2.

Script Gappa 4.2 Preuve par test exhaustif de flottants

```
@rnd = float< ieee_32, ne >;
x = rnd(xx);
y = 2 * sqrt(x) - 1;
{ x in [0,1.00001] -> |rnd(y) - y| <= 1b-25 }
|rnd(y) - y| $ x;
```

Cependant, x n'est pas ici un réel quelconque, c'est un flottant simple précision. Un sous-intervalle qui l'encadre peut donc être réduit jusqu'à être ponctuel ou vide, ce qui permet à Gappa de considérer un à un tous les flottants compris entre 1 et 1.00001 puisqu'il y en a un nombre fini. Pour chacun d'eux, il calcule un encadrement de $\circ(2\cdot\sqrt{x}-1)-(2\cdot\sqrt{x}-1)$ et vérifie que l'erreur est effectivement bornée par 2^{-25}.

Dans le cadre de l'arithmétique à virgule fixe, ce théorème qui s'appuie sur l'encadrement de $\circ(e)$ n'a pas besoin d'être implanté. Les théorèmes liés au prédicat de virgule fixe décrit au paragraphe 4.4.1 le rendent en effet redondant. Ils sont eux aussi capables de réduire la taille d'un intervalle de valeurs approchées jusqu'à ce qu'il soit vide ou que les bornes soient des nombres représentables dans le format virgule fixe.

4.3.2 Erreurs absolues

Arithmétique à virgule fixe

En arithmétique à virgule fixe, des bornes simples et constantes sont généralement connues sur l'erreur absolue $\circ(e) - e$ commise lors de l'arrondi d'une expression. Ainsi, si $\circ$ est un arrondi au plus près avec un poids minimal de 2^k, alors l'expression $\circ(e) - e$ est encadrée par $[-2^{k-1}, 2^{k-1}]$. Si l'arrondi est dirigé vers $+\infty$, l'intervalle devient $[0, 2^k]$. Et ainsi de suite. Il n'est pas nécessaire d'avoir une hypothèse sur e pour obtenir ces encadrements.

Cependant dans le cas de l'arrondi vers zéro ou vers l'infini, on obtient un intervalle $[-2^k, 2^k]$. Si jamais l'expression e ne prend que des valeurs positives ou que des valeurs négatives, alors l'intervalle pourrait être réduit de moitié. Ainsi, si e ne prend que des valeurs positives ou nulle et si l'arrondi est dirigé vers zéro, celui-ci se comporte comme un arrondi vers $-\infty$ et on peut substituer l'intervalle $[-2^k, 0]$ à $[-2^k, 2^k]$. De façon analogue, si $\circ(e)$ ne prend que des valeurs strictement positives et si l'arrondi est toujours vers zéro, Gappa utilise l'encadrement $\circ(e) - e \in [-2^k, 0]$.

En virgule fixe, pour les arrondis vers zéro et vers l'infini, des théorèmes prenant en hypothèse des encadrements de e ou $\circ(e)$ seront donc définis afin de bénéficier de meilleurs encadrements lorsque le signe de ces expressions est connu.

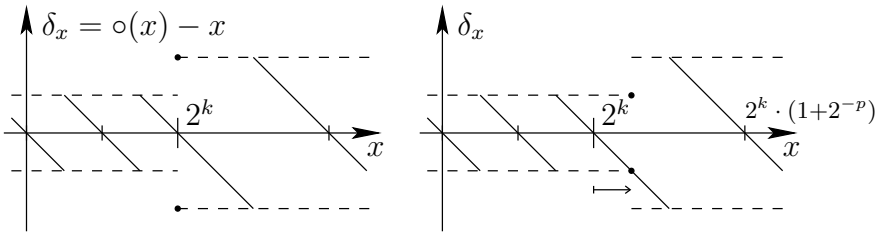

FIG. 4.1 – Bornes de l'erreur absolue aux alentours d'une puissance de deux

Arithmétique à virgule flottante

En matière d'arithmétique flottante, l'erreur relative est un peu plus naturelle que l'erreur absolue. On pourrait donc se contenter de théorèmes à son propos et laisser Gappa utiliser des réécritures et des calculs pour en déduire des encadrements d'erreur absolue. Cependant, d'une part, l'erreur relative n'est pas définie si e passe par zéro, d'autre part, les résultats obtenus sont un peu pessimistes si les valeurs de e ne couvrent qu'une binade (l'espace entre deux puissances de 2 successives). Des théorèmes sont donc aussi disponibles en arithmétique flottante afin d'encadrer une erreur absolue $\circ(e) - e$ en fonction des encadrements de e ou $\circ(e)$.

Les hypothèses de ces théorèmes demandent de majorer strictement les bornes arrondies par une puissance de deux et d'en déduire l'erreur absolue maximale qui peut être commise en fonction de la précision et de l'exposant minimal du format. Le premier graphe de la figure 4.1 montre l'erreur absolue $\circ(x) - x$ commise en fonction de la valeur de x pour l'arrondi au plus près. La courbe de l'erreur croise l'axe des abscisses pour les valeurs réelles exactement représentables. La borne utilisée dans les théorèmes est une fonction qui majore l'erreur ; son tracé (et celui de son opposée) est représenté en pointillés.

Le tracé montre que cette méthode manque de finesse aux alentours des puissances de deux : le deuxième graphe étend légèrement le domaine sur laquelle la petite borne est employée. Notons p la précision du format flottant. Au lieu d'appliquer la grande borne d'erreur dès 2^k, on peut continuer à appliquer la petite borne jusqu'à $2^k \cdot (1+2^{-p-2})$. Pour les encadrements $x \in [3, 7]$ et $y \in [6, 9]$, cela signifie donc que l'on peut diviser par deux l'encadrement que fournissent les théorèmes pour l'erreur absolue de $\circ(x + y) - (x + y)$.

Les versions avancées des théorèmes permettent donc d'améliorer les bornes d'erreur. Elles nécessitent cependant une vérification plus complexe des hypothèses puisqu'il ne suffit plus de borner des nombres dyadiques par des puissances de deux. Gappa ne les utilise donc dans la preuve générée que dans les cas où elles réduisent l'intervalle encadrant l'erreur absolue.

4.3.3 Erreurs relatives

L'erreur relative ne présente aucun intérêt pour l'arithmétique à virgule fixe. Les théorèmes la concernant sont donc principalement définis pour les opérateurs d'arrondi flottants. Elle est alors encadrée par des valeurs simples. Contrairement à l'erreur absolue, il est inutile de se préoccuper des exposants des nombres pour la calculer puisque les bornes sont en fait constantes et ne dépendent que de l'opérateur d'arrondi.

Il est cependant indispensable de s'assurer que e ou $\circ(e)$ ne s'approchent pas du domaine des nombres dénormalisés. Les bornes habituelles n'y sont en effet plus valables et Gappa devra passer par l'erreur absolue pour obtenir des résultats intéressants dans ce domaine. Comme ce domaine interdit se trouve centré en zéro, les considérations du paragraphe 3.3.3 s'appliquent : il est intéressant d'avoir des théorèmes qui prennent en hypothèse des encadrements non pas de e et $\circ(e)$ mais de leurs valeurs absolues $|e|$ et $|\circ(e)|$.

4.3.4 Réécritures

Les théorèmes décrits pour borner les erreurs d'arrondi s'appliquent aux erreurs élémentaires de la forme $\circ(e) - e$ ou $\frac{\circ(e)-e}{e}$. Ceci est indispensable pour pouvoir borner l'erreur locale à une opération, mais insuffisant pour borner l'erreur d'arrondi commise à l'échelle d'une séquence d'instructions. Les techniques de réécritures décrites au paragraphe 3.2 permettent de résoudre ce problème.

Script Gappa 4.3 Erreur relative d'un produit de sommes

```
@rnd = float< ieee_64, ne >;
r rnd= (a + b) * (c + d);
s = (a + b) * (c + d);
{ a + b in [1,100] /\ c + d in [1,100] ->
  (r - s) / s in ? }
```

Le script 4.3 demande à Gappa un encadrement de l'erreur relative $\frac{r-s}{s}$ commise lors du produit de deux sommes de deux nombres flottants chacune. Les hypothèses sur les sommes $a+b$ et $c+d$ affirment juste que les valeurs considérées ne s'aventurent jamais près de zéro et que les théorèmes d'erreur relative s'appliquent.

Notons p et q les sommes $a + b$ et $c + d$. Les valeurs calculées effectivement sont $\circ(p)$ et $\circ(q)$. Le résultat final est alors $r = \circ(\circ(p) \cdot \circ(q))$ tandis que la valeur exacte est $s = p \cdot q$. Les théorèmes décrits précédemment permettent d'encadrer les erreurs élémentaires $\frac{\circ(p)-p}{p}$ et $\frac{\circ(q)-q}{q}$.

En considérant $\circ(p)$ et $\circ(q)$ comme des valeurs approchées de p et q, la réécriture consacrée à l'erreur relative de la multiplication réelle décrite au paragraphe 3.2.1 permet d'encadrer l'erreur $\frac{\circ(p)\cdot\circ(q)-s}{s}$.

$$\frac{\circ(p) \cdot \circ(q) - p \cdot q}{p \cdot q} = \frac{\circ(p) - p}{p} + \frac{\circ(q) - q}{q} + \frac{\circ(p) - p}{p} \cdot \frac{\circ(q) - q}{q}$$

Le produit $\circ(p) \cdot \circ(q)$ n'est pas r, mais on peut considérer r comme étant une expression approchée de $\circ(p) \cdot \circ(q)$. Ainsi, $\circ(p) \cdot \circ(q)$ est une expression intermédiaire entre r et e. La réécriture décrite au paragraphe 3.2.2 s'applique alors.

Puis en considérant r comme étant une valeur approchée de $\circ(p) \cdot \circ(q)$, une deuxième réécriture fait apparaître l'erreur d'arrondi élémentaire commise lors de la multiplication dans l'erreur globale.

$$\frac{r-s}{s} = \frac{r-\circ(p)\cdot\circ(q)}{\circ(p)\cdot\circ(q)} + \frac{\circ(p)\cdot\circ(q)-s}{s} + \frac{r-\circ(p)\cdot\circ(q)}{\circ(p)\cdot\circ(q)} \cdot \frac{\circ(p)\cdot\circ(q)-s}{s}$$

Ainsi, en faisant intervenir le terme $\circ(p) \cdot \circ(q)$, c'est-à-dire r duquel on a supprimé l'opérateur d'arrondi de plus haut niveau, on s'est ramené aux erreurs d'arrondi commises lors des trois opérations flottantes. On peut alors combiner ces erreurs locales pour borner l'erreur globale.

Le cheminement décrit ici n'est pas celui suivi par Gappa, mais par la preuve que l'outil génère. À chaque fois que Gappa rencontre une expression de la forme $\circ(e)$, il la considère comme une valeur approchée de e et applique les règles de réécritures qui font intervenir des couples d'expressions approchée et exacte.

4.4 Prédicats de précision

Les prédicats décrits ci-après apportent une note de discrétion aux encadrements d'expressions arrondies. L'objectif est de pouvoir montrer que certains calculs ne sont entachés d'aucune erreur d'arrondi parce que les résultats réels sont tous représentables. Par exemple, l'erreur absolue entre un nombre et sa partie entière $\lfloor e \rfloor - e$ est normalement encadrée par $[-1, 0]$; par contre il devient possible d'obtenir l'encadrement $[0, 0]$ si l'on prouve que l'expression e ne prend que des valeurs entières.

Une première approche serait d'associer aux deux bornes d'un intervalle une troisième valeur qui indiquerait le pas entre deux valeurs consécutives de l'ensemble représenté [Pat95]. Ainsi l'encadrement $e \in [-3, 7; 1]$ signifierait que e prend une valeur de la forme $-3 + 1 \cdot k$ avec k un entier entre 0 et 10. L'inconvénient de cette approche est le lien artificiel qu'elle crée entre les deux propriétés. Il n'est en effet pas nécessaire de connaître l'intervalle dans lequel se trouve e pour savoir que $\lfloor e \rfloor - e$ vaut zéro. Qui plus est, la virgule flottante ne bénéficierait que peu de cette représentation, puisque le pas entre les valeurs flottantes n'est généralement pas constant et l'ensemble contiendrait de nombreux éléments superflus.

L'approche que j'ai retenue dans Gappa n'est donc pas une extension des encadrements. Ce sont deux nouveaux prédicats qui indiquent la « précision » suffisante pour représenter exactement les valeurs d'une expression. L'un des prédicats est adapté aux arithmétiques à virgule fixe, l'autre aux arithmétiques à virgule flottante. Dans les deux cas, ces prédicats affirment que les valeurs d'une expression sont des nombres dyadiques et ils décrivent les propriétés de ces nombres.

4.4.1 Prédicat de virgule fixe

Sans l'aide du prédicat de virgule fixe, les encadrements d'erreurs prouvés par Gappa pour des calculs en virgule fixe ne sont pas aussi fins que ceux que l'on peut facilement obtenir à la main. L'outil ne se rend pas compte que nombre d'opérations arithmétiques à virgule fixe sont en réalité des opérations exactes : l'arrondi ne perd aucune information du résultat parce que le développeur a prévu des registres suffisamment larges. Pour repérer ces opérations exactes, il suffit de savoir la précision (position du bit de poids faible) suffisante pour stocker leur résultat exactement.

Script Gappa 4.4 La somme de deux entiers est un entier

```
x = int<zr>(x_);
y = int<zr>(y_);
{ int<zr>(x + y) - (x + y) in ? }
```

Le script 4.4 correspond à la proposition selon laquelle la somme de deux entiers est représentable par un entier. Pour indiquer à Gappa que les valeurs d'une expression ont un format particulier, il suffit de la définir comme étant l'arrondi à ce format d'une autre expression. Pour vérifier que les valeurs d'une expression ont un certain format, il suffit de vérifier que l'erreur absolue, entre cette expression et son arrondi vers le format, est nulle. Sur cet exemple, Gappa répond que l'erreur est effectivement nulle. Pour ce faire, il montre d'abord que x et y sont des éléments de l'ensemble $\mathbb{Z} \cdot 2^0$. Il en déduit que leur somme $x + y$ appartient aussi à $\mathbb{Z} \cdot 2^0$, puis qu'arrondir $x + y$ ne modifie pas sa valeur et que l'erreur d'arrondi est nulle.

Pour tenir ces raisonnements, Gappa utilise le prédicat suivant :

$$\mathrm{FIX}(x, k) = \exists m, e \in \mathbb{Z}, \quad x = m \cdot 2^e \wedge e \geq k.$$

Quand Gappa sait qu'une expression x est de la forme `fixed<`k, dir`>`(e), il peut en déduire la propriété $\mathrm{FIX}(x, k)$. Réciproquement, si l'outil sait prouver $\mathrm{FIX}(e, k)$, alors il peut déduire que l'encadrement est $[0, 0]$ pour toute expression de la forme `fixed<`k', dir`>`$(e) - e$ avec $k' \geq k$.

Comme les bornes des intervalles sont représentées par des nombres dyadiques, Gappa peut aussi construire une propriété de précision à partir d'une expression encadrée par un intervalle réduit à un point. Par exemple, si les valeurs de e appartiennent à l'intervalle point $[3 \cdot 2^{-1}, 3 \cdot 2^{-1}]$, la propriété $\mathrm{FIX}(e, -1)$ est vraie.

Ensuite, tout comme pour les encadrements et leur arithmétique d'intervalles, des calculs peuvent être effectués sur ce prédicat.

$$\begin{aligned}
\mathrm{FIX}(x, k) \wedge \mathrm{FIX}(y, l) &\Rightarrow \mathrm{FIX}(x + y, \min(k, l)) \\
\mathrm{FIX}(x, k) \wedge \mathrm{FIX}(y, l) &\Rightarrow \mathrm{FIX}(x \cdot y, k + l) \\
\mathrm{FIX}(e, k) \wedge \mathrm{FIX}(e, l) &\Rightarrow \mathrm{FIX}(e, \max(k, l))
\end{aligned}$$

Finalement, une propriété $\mathrm{FIX}(e, k)$ peut être utilisée par Gappa afin d'améliorer un encadrement sur e. En effet, un encadrement de e est optimal si son intervalle est vide ou si ses bornes sont des multiples de 2^k. Par conséquent, si e appartient à $[a, b]$, e appartient aussi à l'intervalle $[a', b']$ avec a' l'arrondi vers $+\infty$ de a et b' l'arrondi vers $-\infty$ de b. Par exemple, si e est un entier appartenant à $[1.7, 3.6]$, il appartient en fait à $[2, 3]$. Et si e est un entier appartenant à $[2.3, 2.9]$, il prend ses valeurs dans l'intervalle vide et la preuve est achevée.

4.4.2 Prédicat de virgule flottante

Ce prédicat est l'analogue du prédicat précédent adapté à l'arithmétique flottante. Au lieu d'indiquer l'exposant suffisant pour représenter un nombre dyadique, il indique la taille de mantisse suffisante pour représenter ce nombre.

$$\mathrm{FLT}(x, k) = \exists m, e \in \mathbb{Z}, \quad x = m \cdot 2^e \wedge |m| < 2^k.$$

Là encore, il est possible de déduire des propriétés d'un intervalle réduit à un point. Par exemple, si $e \in [5 \cdot 2^{11}, 5 \cdot 2^{11}]$, Gappa peut prouver $\mathrm{FLT}(e, 3)$ puisque $|5| < 2^3$. Des propriétés découlent aussi directement de la présence d'opérateurs d'arrondis flottants. Ainsi, si e est un résultat au format flottant simple précision IEEE-754, alors on a $\mathrm{FLT}(e, 24)$.

Pour déduire qu'une erreur absolue est nulle, l'approche est un peu plus complexe qu'en arithmétique à virgule fixe, un format à virgule flottante étant représenté à la fois par une précision de mantisse et un exposant minimal. Donc, pour déduire `float<`p, q, dir`>`$(e) - e \in [0, 0]$, Gappa cherche à obtenir $\mathrm{FLT}(e, p')$ avec $p' \leq p$ et $\mathrm{FIX}(e, q')$ avec $q' \geq q$.

Finalement, il est possible d'effectuer des calculs sur ce prédicat :

$$\begin{aligned} \mathrm{FLT}(x, k) \wedge \mathrm{FLT}(y, l) &\Rightarrow \mathrm{FLT}(x \cdot y, k + l) \\ \mathrm{FLT}(e, k) \wedge \mathrm{FLT}(e, l) &\Rightarrow \mathrm{FLT}(e, \min(k, l)) \end{aligned}$$

4.4.3 Exemple : Sterbenz

En arithmétique flottante, il arrive sous certaines conditions que la somme de deux nombres flottants soit représentable. En particulier, un lemme communément attribué à Sterbenz [Ste74] affirme que si le quotient $\frac{a}{b}$ est encadré par $[\frac{1}{2}, 2]$, alors la soustraction $a \ominus b$ est exacte, c'est-à-dire que l'erreur absolue $\circ(a - b) - (a - b)$ est nulle. Une première approche consisterait donc à ajouter dans Gappa plusieurs théorèmes exprimant cette propriété.

Voyons cependant sur un exemple ce qu'apporte les prédicats de précision. Le script 4.5 effectue la soustraction de deux valeurs a et b, en simple précision et arrondi dirigé vers zéro. Le lemme de Sterbenz ne pourrait pas s'appliquer sur cet exemple. L'encadrement de $\frac{a}{b}$ contient en effet des valeurs en dehors de l'intervalle $[\frac{1}{2}, 2]$. Gappa répond néanmoins que l'erreur absolue est effectivement nulle.

Script Gappa 4.5 Proposition de type lemme de Sterbenz

```
@rnd = float< ieee_32, zr >;
a = rnd(a_); b = rnd(b_);
{ a in [3.2,3.3] /\ b in [1.4,1.9] ->
  rnd(a - b) - (a - b) in ? }
```

Dans la preuve générée, Gappa indique tout d'abord que, a étant un flottant simple précision, ses valeurs sont représentables par des nombres dyadiques dont la mantisse fait moins de 24 bits. Il constate d'autre part que la valeur absolue de a est plus grande que 3.2. Par conséquent, a est nécessairement un multiple de 2^{-22}. En suivant un raisonnement analogue, il en déduit que b est un multiple de 2^{-23}.

Partant de propriétés de virgule flottante et d'encadrements, l'outil a ainsi obtenu des propriétés de virgule fixe sur a et b. Il en déduit alors que $a - b$ est un multiple de 2^{-23}. Or il sait aussi prouver que la valeur absolue de $a - b$ ne dépasse pas 1.9. Il va cette fois combiner une propriété de virgule fixe avec un encadrement pour montrer que les valeurs de l'expression $a - b$ sont représentables par des nombres dyadiques avec des mantisses de taille inférieure à 24 bits.

De plus, comme $a - b$ est un multiple de 2^{-23}, c'est aussi un multiple de 2^{-149}. Par conséquent, les valeurs de $a - b$ sont représentables en simple précision et aucune erreur d'arrondi n'est commise pendant la soustraction.

Cet exemple montre que les prédicats FIX ne se cantonnent pas au domaine de l'arithmétique à virgule fixe. En utilisant des théorèmes les combinant aux encadrements et aux prédicats FLT, Gappa est capable de prouver que certaines opérations flottantes sont exactes. Dans l'exemple, tous les nombres avaient le même format, mais la méthode est plus générale et fonctionne même quand les formats diffèrent.

Chapitre 5

Fonctionnement de Gappa

Ce chapitre décrit la mise en œuvre, dans Gappa, des méthodes théoriques décrites aux chapitres précédents, c'est-à-dire le travail à effectuer pour qu'elles s'appliquent : préparation des propositions logiques, construction des graphes de preuve, gestion des bissections et des réécritures.

Le travail de Gappa sur une proposition logique se décompose en plusieurs étapes. La proposition est tout d'abord transformée en un ensemble de plusieurs propositions adaptées au formalisme décrit au chapitre 2. Cette étape est décrite au paragraphe 5.1.1. Étant donnée une proposition élémentaire, Gappa cherche alors l'ensemble des expressions sur lesquelles des propriétés pourraient être prouvables[1] et ne retient que celles qui peuvent être utiles pour prouver la proposition (paragraphe 5.1.2).

Une fois cet ensemble d'expressions déterminé, Gappa cherche à calculer les propriétés qui les concernent en appliquant les théorèmes de sa base concernant l'arithmétique d'intervalles ou les arithmétiques approchées, ainsi que des réécritures (paragraphe 5.4). L'outil a une liste de couples d'expressions « approchée » et « exacte » pour inférer les termes absents des expressions à réécrire. Si la proposition n'est pas encore prouvée après avoir essayé tous ces théorèmes, Gappa applique des méthodes de bissection pour construire un nouveau jeu de propositions élémentaires sur lesquelles il recommence son travail. Après avoir trouvé une bissection convenable, Gappa réalise une fusion optimale des sous-cas pour limiter leur nombre (paragraphe 5.3).

Tout au long de ces calculs de propriétés, Gappa tient à jour un graphe de preuve (paragraphe 5.2) qui lui servira à générer le certificat. Celui-ci est constitué d'une succession de lemmes et de leurs preuves ; le dernier d'entre eux corres-

[1] À ce stade, l'outil ne sait pas encore si les propriétés considérées sont prouvables : des théorèmes les concernant existent mais rien ne lui permet de savoir si leurs hypothèses pourront être satisfaites.

pond exactement à la proposition énoncée par l'utilisateur. Cependant, avant de produire ce certificat, les techniques de simplification des bornes décrites au paragraphe 2.3.3 sont mises en œuvre afin que le certificat ne pâtisse pas de la précision élevée employée par l'outil lors de ses calculs.

5.1 Prétraitement

5.1.1 Mise en forme des propositions logiques

Étant donnée une proposition logique, Gappa commence par la mettre sous une forme adaptée à la suite de son travail. Comme décrit au chapitre 2, cette forme est une implication ayant en membre de gauche une conjonction d'hypothèses d'encadrement. Pour des raisons d'efficacité, le membre de droite n'est pas limité à un unique encadrement mais à n'importe quelle formule contenant des conjonctions, disjonctions et encadrements.

$$e_1 \in I_1 \wedge e_2 \in I_2 \wedge \cdots \wedge e_n \in I_n \quad \Rightarrow \quad P$$

Simplification des propositions

Pour atteindre des propositions de cette forme, des transformations proches du calcul des séquents sont appliquées à la proposition initiale jusqu'à ce que chacun des termes de la conjonction de gauche ait été décomposé en un encadrement élémentaire. Ces transformations peuvent avoir pour effet d'augmenter le nombre de propositions à considérer. Par exemple, si la proposition est de la forme $e_1 \in I_1 \wedge \ldots \wedge e_n \in I_n \wedge (Q \Rightarrow R) \Rightarrow P$, elle est d'abord remplacée par les deux propositions suivantes :

$$\begin{aligned} e_1 \in I_1 \wedge \cdots \wedge e_n \in I_n &\Rightarrow P \vee Q \\ e_1 \in I_1 \wedge \cdots \wedge e_n \in I_n \wedge R &\Rightarrow P \end{aligned}$$

La proposition initiale est ainsi progressivement transformée en un ensemble d'implications dont le membre de gauche est une conjonction d'encadrements, tandis que le membre de droite ne contient, ni symbole de négation, ni symbole d'implication. L'information que Gappa peut utiliser se retrouve donc en hypothèse de ces propositions transformées.

Pour simplifier le traitement, les propositions finales ne contiennent jamais deux hypothèses encadrant la même expression. Ces cas s'éliminent facilement en effectuant l'intersection des encadrements : $e_i \in I_i \wedge e_i \in I_j$ se remplace par $e_i \in I_i \cap I_j$. Les bornes doivent aussi être transformées en nombres dyadiques. Les intervalles fournis par l'utilisateur voient donc leurs bornes arrondies vers l'extérieur quand ils font partie du membre de gauche et arrondies vers l'intérieur quand ils font partie du membre de droite.

Au lieu de fournir des bornes numériques, l'utilisateur peut représenter un encadrement par « *expr* **`in ?`** ». De tels encadrements ne sont autorisés que s'ils ne

se retrouvent que dans les parties droites des propositions finales. Gappa cherche alors un encadrement à bornes finies de $expr$ pour lequel la proposition est prouvable.

Traitement des inégalités

Les inégalités, qui sont des encadrements partiels, suivent les mêmes transformations que les encadrements normaux. Gappa effectue cependant une transformation supplémentaire pour celles du membre de droite en disjonction. Une proposition $P \Rightarrow Q \vee e \geq c$ est ainsi réécrite en $P \wedge e \leq c \Rightarrow Q \vee e \geq c$ afin d'ajouter une hypothèse supplémentaire. Celle-ci permettra peut-être de trouver une contradiction au sein des hypothèses et donc de prouver la proposition.

Gappa n'effectue pas d'autres transformations sur la proposition logique. Mais on pourrait imaginer de généraliser le traitement appliqué aux inégalités. Les encadrements sont en effet des conjonctions d'inégalités et il est donc possible de pousser la décomposition d'une proposition jusqu'à ne plus avoir dans le membre de droite qu'une disjonction d'inégalités. Celles-ci peuvent alors toutes être retournées au sein du membre de gauche (en utilisant, le cas échéant, des inégalités strictes). Son membre de droite ne contenant alors plus aucune propriété, la seule façon de prouver la proposition est de trouver une contradiction entre ses hypothèses. Un tel traitement permettrait de s'assurer que toute l'information contenue dans la proposition logique initiale se retrouve en hypothèse des propositions sur lesquelles Gappa va travailler. L'algorithme commencerait donc comme les méthodes de résolution [RV01] puis calculerait par saturation les encadrements de toutes les expressions retenues.

Alors que ce traitement permettrait de prouver un plus grand nombre de propositions, Gappa ne l'applique pas jusqu'au bout. Il se limite à des propositions dont les membres de gauche sont des conjonctions d'inégalités larges et les membres de droite des conjonctions et disjonctions d'inégalités larges. En effet, le traitement complet empêche d'une part de traiter les encadrements « `in ?` » dont l'intervalle n'est pas fixé à l'avance et il provoque d'autre part une explosion du nombre de preuves à construire. Aucune des certifications réalisées n'a pour l'instant eu besoin de ce surcroît de complexité.

5.1.2 Recherche des chemins de calcul

Une fois la proposition décomposée en propositions élémentaires, Gappa considère les propositions les unes à la suite des autres. L'objectif est, à partir des encadrements en hypothèse, d'obtenir des encadrements qui satisfont le membre de droite de l'implication. La première étape consiste à rechercher les expressions intermédiaires que l'outil a besoin d'encadrer et les théorèmes qui vont donner ces encadrements.

Cette recherche se fait itérativement en partant de l'ensemble des expressions qui apparaissent, non seulement dans le membre de droite, mais aussi dans le

membre de gauche. En effet, s'il existe une contradiction entre les hypothèses, elle apparaîtra vraisemblablement dans leurs encadrements. Par exemple, dans la proposition $x \in [1,2] \wedge x+1 \in [3,4] \Rightarrow x \in [5,6]$, la contradiction se manifeste lors du calcul d'un autre encadrement de $x+1$.

Pour chacune des expressions de cet ensemble, Gappa génère la liste des théorèmes qui peuvent engendrer leurs encadrements. Chacun de ces théorèmes a lui même des hypothèses qui consistent en des encadrements d'autres expressions. Celles-ci sont ajoutées à l'ensemble des expressions à encadrer afin d'être à leur tour considérées. Certaines des expressions de l'ensemble peuvent s'avérer impossible à encadrer, par exemple une variable universellement quantifiée sur laquelle il n'existe aucune hypothèse. Elle est considérée inutile et retirée de l'ensemble et les théorèmes qui en dépendent sont supprimés. Ces suppressions peuvent laisser des expressions sans aucun théorème capable de calculer leurs encadrements. Elles sont à leur tour retirées de l'ensemble, ce qui entraîne à nouveau la suppression de théorèmes. Le processus se poursuit jusqu'à ce que l'outil n'ait plus que des expressions « utiles » et les théorèmes pour les encadrer à partir des encadrements d'autres expressions « utiles ». Le terme « encadrement » a été employé dans ce qui précède, mais il peut tout aussi bien s'agir de prédicats de précision.

Le procédé employé par Gappa termine en pratique. Les règles de réécritures et les théorèmes ont en effet été choisis pour que, même si l'outil peut avoir à considérer des expressions plus complexes que l'expression initiale, il n'existe aucune itération capable de générer une suite d'expressions arbitrairement complexes. Deux remarques permettent de s'en persuader.

Premier point, Gappa ne cherche pas à rassembler une nouvelle fois les théorèmes qui encadrent une expression qu'il a déjà rencontrée auparavant. Pour illustrer ce point, supposons qu'il lui faille encadrer $\circ(a)$. Une réécriture lui dit qu'il peut passer par l'encadrement de $(\circ(a)-a)+a$. Puis le théorème d'addition lui annonce que cela s'obtient en encadrant séparément $\circ(a)-a$ et a. L'outil construit alors la liste des théorèmes permettant d'encadrer $\circ(a)-a$, mais pas celle de l'expression a qui a déjà été rencontrée.

Deuxième point, les réécritures ne s'appliquent qu'à l'expression que Gappa cherche à encadrer et non à ses sous-expressions. En particulier, $\circ(a)$ se réécrit en $(\circ(a)-a)+a$, mais cette dernière ne se réécrit pas en $(((\circ(a)-a)+a)-a)+a$. L'utilisateur peut fournir ses propres règles, mais elles ne posent pas de difficultés non plus. En effet comme elles ne contiennent pas des termes joker qui seraient satisfaits par des sous-expressions quelconques, elles s'appliquent tout au plus une fois chacune. Ce ne serait pas le cas avec une règle $(\%_1) \rightarrow (2 \cdot \%_1 - \%_1)$ dans laquelle $\%_1$ est un terme joker.

5.2 Graphes de preuve

En suivant les chemins de calculs qu'il vient d'obtenir, Gappa construit des propriétés d'expressions (encadrement ou précision). Il part de la conjonction d'enca-

drements présente en hypothèse et ajoute progressivement de nouvelles propriétés en combinant celles déjà obtenues. Cette itération se poursuit jusqu'à ce que les encadrements de la proposition à prouver soient satisfaits. Si un de ces encadrements est « ? », l'itération se poursuit jusqu'à ce que plus aucun théorème n'ajoute de nouvelle propriété ou n'améliore les propriétés déjà présentes.

5.2.1 Théorèmes et graphes de preuves

Les chemins de calcul indiquent à Gappa quels théorèmes s'appliquent quand telles hypothèses sont disponibles. À chaque fois qu'une nouvelle propriété est disponible ou qu'une propriété a été améliorée, les théorèmes qui ont cette propriété en hypothèse sont placés à la fin d'une queue indiquant les théorèmes potentiellement applicables. Pour chaque théorème applicable, Gappa vérifie que les contraintes (par exemple l'intervalle dénominateur d'une division ne doit pas contenir zéro) sont bien remplies puis calcule la propriété conclusion qui en découle.

Dans le cas des prédicats de précision, ce calcul est simplement effectué en arithmétique entière. Pour ce qui est des prédicats d'encadrements, les bornes dyadiques sont manipulées à l'aide de MPFR [FHL+05], la précision des calculs étant fixée de façon globale par un paramètre utilisateur. La bibliothèque d'arithmétique d'intervalles de Boost, présentée dans l'annexe A, calcule les bornes de ces encadrements.

Une fois la propriété calculée, deux nœuds sont ajoutés au graphe de preuve pour en garder une trace. Le premier décrit l'instance du théorème appliqué ; il ne dépend de rien et exprime un résultat vrai dans l'absolu. Le second nœud relie le premier aux nœuds du graphe qui ont pour conclusions les propriétés en hypothèses du théorème. Dans la figure 5.1, les nœuds du premier type sont indiqués par un cadre en trait plein tandis que les nœuds du second type sont indiqués par un cadre en trait discontinu.

Un graphe acyclique est donc généré au fur et à mesure des calculs. Les arêtes dirigées de ce graphe indiquent les dépendances entre les différents faits, c'est-à-dire tels nœuds impliquent tels nœuds. Ce graphe sera parcouru pour générer le script de preuve une fois que les propriétés en conclusion de la proposition auront été satisfaites. La figure 5.1 montre le graphe créé pour la proposition

$$x \in [0, \tfrac{1}{4}] \quad \Rightarrow \quad x \cdot (1 - x) \in [0, \tfrac{1}{4}].$$

Initialement le graphe ne contient qu'un seul nœud $x \in [0, \frac{1}{4}]$ qui correspond à l'encadrement en partie gauche de l'implication dans la proposition logique. Partant de là, l'objectif pour Gappa est de trouver un encadrement de $x \cdot (1 - x)$ qui satisfait la partie droite de l'implication : $x \cdot (1 - x) \in [0, \frac{1}{4}]$. Le nœud $1 \in [1, 1]$ sur la gauche et les deux nœuds contenant des implications sur la droite du graphe sont des instances de théorèmes d'arithmétique par intervalles : encadrement d'une constante, encadrement d'une différence, encadrement d'un produit. Les propriétés qu'ils expriment sont vraies pour n'importe quelle valeur réelle de x. Les autres

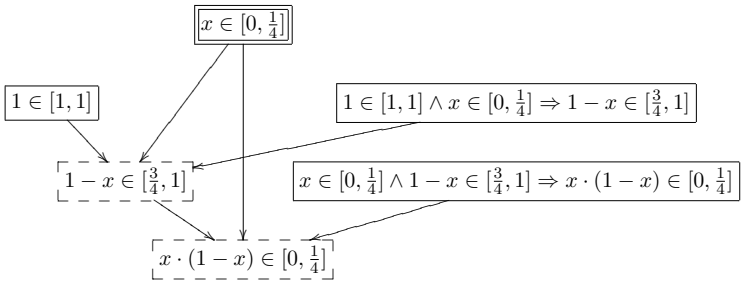

FIG. 5.1 – Graphe de preuve de $x \in [0, \frac{1}{4}] \Rightarrow x \cdot (1 - x) \in [0, \frac{1}{4}]$

nœuds contiennent des propriétés vraies du fait des hypothèses de la proposition, c'est-à-dire quand x prend ses valeurs dans $[0, \frac{1}{4}]$.

5.2.2 Intersections

Deux nouveaux nœuds sont ajoutés pour chaque nouvelle propriété obtenue. La propriété qu'ils prouvent peut concerner une expression et un prédicat pour lesquels il existe déjà une propriété prouvée. Trois cas se présentent alors. Premier cas, la nouvelle propriété n'apporte rien par rapport à l'ancienne ; elle peut donc être aussitôt oubliée. C'est par exemple le cas si l'ancienne propriété affirmait que $x \in [2, 6]$ alors que la nouvelle donne $x \in [1, 7]$.

Deuxième cas, la nouvelle propriété est strictement meilleure que l'ancienne, par exemple si la nouvelle est maintenant $x \in [3, 5]$. Tous les théorèmes appliqués à partir de ce moment pourront donc utiliser cette propriété afin de produire des résultats potentiellement plus fins. Les théorèmes déjà appliqués ne sont par contre pas modifiés et continuent à dépendre de l'ancien nœud. Les faire dépendre du nouveau nœud pourrait en effet créer des cycles dans le graphe de preuve.

Le dernier cas ne se présente qu'avec les prédicats d'encadrement. En effet, l'inclusion d'intervalles n'étant qu'un ordre partiel, la nouvelle propriété peut être ni strictement meilleure ni strictement plus mauvaise que l'ancienne. C'est le cas par exemple de $x \in [2, 6]$ et $x \in [4, 7]$. Il est cependant possible de déduire de ces deux propriétés que x appartient à l'intersection des deux intervalles : $x \in [4, 6]$. Un nœud supplémentaire est ajouté au graphe pour refléter cette déduction, la propriété étant maintenant strictement meilleure.

Ces nœuds d'intersection sont aussi le seul moyen de gérer les inégalités qui se trouvent en hypothèse. Comme aucun de ses théorèmes ne manipule une inégalité sur une expression, Gappa attend d'obtenir, par un chemin de calcul, un premier encadrement de cette expression, puis il utilise l'inégalité pour réduire cet encadrement.

Dans les situations d'intersection, il peut arriver que les deux encadrements soient disjoints, par exemple si les propriétés sont $x \in [2, 4]$ et $x \in [5, 7]$. Les valeurs de x appartiennent alors à l'intervalle vide ; une contradiction a donc été trouvée et il n'est pas nécessaire de continuer plus avant. C'est le seul endroit de

Gappa où des intervalles vides sont envisagés. Aucun autre calcul ou théorème n'en renvoie, ce qui permet de ne pas les prendre en compte lors de la génération de la preuve.

Pour satisfaire cette contrainte, certains théorèmes sont forcés de renvoyer des intervalles non vides là où il serait naturel de trouver un intervalle vide. Supposons par exemple que x soit un entier appartenant à l'intervalle $[a, b]$. Un théorème de Gappa renvoie un intervalle précis encadrant x : $[\lceil a \rceil, \lfloor b \rfloor]$. Si l'intervalle $[a, b]$ est $[2.5, 2.75]$, cela conduit à un intervalle vide caractérisé par des bornes inversées : $x \in [3, 2]$. Pour éviter cela, le théorème renvoie l'intervalle $[3, 3]$ (l'intervalle $[2, 2]$ conviendrait aussi), non vide mais moins fin. Ce manque de finesse ne pose cependant pas de problème. D'une part parce que le calcul reste formellement vérifiable : $3 \leq \lceil 2.5 \rceil$ et $\lfloor 2.75 \rfloor \leq 3$. D'autre part parce qu'il y a maintenant deux encadrements de x et que leur intersection est vide. La fin de la preuve n'a donc été repoussée que d'une étape.

5.2.3 Encadrements en valeur absolue

Jusqu'à présent, un prédicat d'encadrement et deux prédicats de précision ont été présentés. Le sujet de l'encadrement de la valeur absolue d'une expression a aussi été abordé au paragraphe 3.3.3 mais un tel encadrement peut être exprimé simplement à l'aide du prédicat d'encadrement standard. Cependant, pour deux raisons, Gappa l'implante à l'aide d'un prédicat à part. L'une d'elle est indiquée au paragraphe 6.3.2 : il est possible d'écrire des théorèmes plus simples à vérifier si la borne inférieure de l'intervalle est positive ou nulle, ce qui est le cas pour un encadrement de valeur absolue.

L'autre raison est la diminution du nombre de théorèmes nécessaires. Considérons par exemple qu'il n'y ait pas de prédicat spécifique pour les valeurs absolues et que Gappa cherche à construire les chemins de calcul qui permettent de borner $x \cdot |y|$. Un de ces chemins borne la valeur absolue et en déduit l'encadrement de l'expression originelle. Cela consiste à borner l'expression $|x \cdot |y||$. La valeur absolue du produit étant le produit des valeurs absolues, Gappa peut alors chercher à évaluer $|x| \cdot |(|y|)|$. En répétant ces étapes, des expressions de plus en plus complexes mais complètement inutiles vont donc être créées. Pour éviter cela, un théorème fournissant la propriété $|x \cdot |y|| = |x| \cdot |y|$ serait par exemple nécessaire. Pour des raisons de symétrie, il en faudrait un autre concernant le membre de gauche. L'opération de multiplication n'est pas la seule à souffrir de ce problème et d'autres théorèmes devraient donc être spécialisés pour traiter les expressions contenant des valeurs absolues.

L'ajout d'un prédicat spécifique à la valeur absolue permet d'éviter cette explosion du nombre de théorèmes. Notons ABS ce nouveau prédicat et BND le prédicat standard d'encadrement. Dans l'exemple précédent, Gappa doit alors évaluer $\mathrm{BND}(x \cdot |y|)$, ce qui peut se faire en passant $\mathrm{ABS}(x \cdot |y|)$. La valeur absolue du produit étant le produit des valeurs absolues, cela revient à évaluer $\mathrm{ABS}(x)$ et $\mathrm{ABS}(|y|)$. Ce dernier vaut aussi $\mathrm{ABS}(y)$. Ce prédicat évite donc d'écrire plusieurs

versions des théorèmes. L'inconvénient est que quelques-unes des expressions ne sont plus considérées et certaines propositions ne sont donc pas prouvables par Gappa. C'est le cas de la proposition $|x| \cdot |y| \in [1, 2] \Rightarrow x \cdot |y| \in [-2, 2]$.

En conclusion, Gappa implante un prédicat d'encadrement supplémentaire :

$$\mathrm{ABS}(e, [a, b]) = 0 \leq a \leq |e| \leq b.$$

5.3 Bissections

Une fois que l'outil a calculé les encadrements et les bornes de précision de chacune des expressions qu'il avait à considérer, il cherche à appliquer d'éventuelles méthodes de bissection. Si Gappa a prouvé qu'une expression e est encadrée par $[a, b]$, il découpe l'intervalle en sous-intervalles et chercher à réévaluer les propriétés concernant les autres expressions pour chacun de ces sous-intervalles.

5.3.1 Déroulement d'une bissection

Script Gappa 5.1 Découpage en quatre sous-intervalles de même largeur

```
t = x * x;
{ x in [-2,3] -> t * (1 - t) in ? }
$ t;
```

Dans le script 5.1, la dernière ligne « `$ t;` » indique à Gappa qu'il doit recalculer les propriétés concernant chacune des expressions après avoir découpé l'encadrement de t en sous-intervalles. Plus précisément, Gappa effectue tout d'abord un calcul complet de l'ensemble des propriétés. Il obtient ainsi les encadrements $t \in [0, 9]$ et $t \cdot (1-t) \in [-72, 9]$. Puis il constate qu'un découpage lui est demandé. En l'absence de précision, il choisit un découpage en quatre sous-intervalles.

Comme $t \in [0, 9]$, les quatre intervalles considérés sont $[0, \frac{9}{4}]$, $[\frac{9}{4}, \frac{9}{2}]$, etc. Pour chacun de ces intervalles, Gappa crée un nouveau graphe de preuve. Ce graphe dérive du graphe courant, c'est-à-dire que toutes les propriétés déjà prouvées restent valides, mais il contient une hypothèse supplémentaire sur l'encadrement de t par un sous-intervalle. Gappa recalcule toutes les propriétés sur ce graphe afin d'affiner ses résultats. Pour $t \in [0, \frac{9}{4}]$, l'outil montre que $t \cdot (1 - t)$ est compris dans l'intervalle $[-\frac{45}{16}, \frac{9}{4}]$.

Dans les trois autres graphes de preuve, les intervalles encadrant $t \cdot (1-t)$ sont $[-\frac{63}{4}, -\frac{45}{16}]$, $[-\frac{621}{16}, -\frac{63}{4}]$ et $[-72, -\frac{621}{16}]$. Comme Gappa a obtenu un encadrement dans chacun des quatre nouveaux graphes, il peut créer une propriété les englobant tous dans le graphe initial. Ainsi, après avoir calculé l'enveloppe convexe, un nœud prouvant $t \cdot (1-t) \in [-72, \frac{9}{4}]$ est créé. Ses parents, c'est-à-dire les hypothèses qui assurent la validité du nouveau nœud, sont le nœud prouvant $t \in [0, 9]$ et les quatre nœuds prouvant les encadrements de $t \cdot (1 - t)$ sur chacun des sous-intervalles.

Si Gappa trouve une contradiction dans certains graphes, deux cas se présentent. Premier cas, tous les graphes contiennent une contradiction et l'outil peut en déduire une contradiction globale et terminer la preuve. Deuxième cas, l'outil calcule l'enveloppe convexe de tous les encadrements obtenus dans des graphes sans contradiction. Cette enveloppe est valable globalement puisqu'il est possible de la déduire de chacun des graphes : soit il y a une contradiction, soit l'encadrement du graphe est inclus dans l'enveloppe par définition.

Pour finir, une bissection échoue si aucune contradiction globale n'a été obtenue et si pour aucune des expressions encadrées l'enveloppe convexe calculée n'est meilleure que l'encadrement initialement calculé.

5.3.2 Déclaration de bissections

Comme les bissections multiplient le nombre de graphes de preuve à remplir, elles ne sont jamais réalisées si elles n'ont pas été explicitement demandées par l'utilisateur. La syntaxe pour demander à Gappa de réaliser une bissection est un indice de la forme :

découpage ::= *expr* | *expr* **in** *entier* | *expr* **in (***borne***, ...)**
bissection ::= [*expr***, ...**] **$** *découpage***, ...;**

Un indice de bissection est donc la donnée d'une liste potentiellement vide d'expressions et d'une liste de découpages. Ces découpages se composent de deux parties. La première est une expression qui indique quel encadrement est découpé. La deuxième, optionnelle, indique comment le découpage est réalisé. Quand elle est écrite sous la forme « **in** *entier* », l'entier indique le nombre de sous-intervalles à considérer. Sous la forme « **in (***borne***, ...)** », l'encadrement est découpé par rapport aux points donnés en paramètre.

Ainsi, en utilisant le script 5.2 à la place du précédent, l'encadrement de t est maintenant découpé en trois intervalles $[0, \frac{1}{2}]$, $[\frac{1}{2}, 1]$ et $[1, 9]$. Cela conduit à un meilleur résultat final : $t \cdot (1 - t) \in [-72, \frac{1}{2}]$.

Script Gappa 5.2 Découpage par points

```
t = x * x;
{ x in [-2,3] -> t * (1 - t) in ? }
$ t in (0.5,1);
```

Si plusieurs découpages sont indiqués à droite du dollar, ils sont effectués récursivement. Cela signifie que, dans chaque graphe correspondant à un sous-intervalle du premier découpage, une nouvelle bissection est effectuée en utilisant le deuxième découpage et elle crée de nouveaux graphes. Et ainsi de suite. Par conséquent, si un indice de bissection contient d'abord un découpage en trois sous-intervalles, puis un autre découpage en cinq sous-intervalles et enfin un découpage

en quatre sous-intervalles, il y aura en tout $1 + 3 + 3 \cdot 5 + 3 \cdot 5 \cdot 4 = 79$ graphes de preuve à remplir.

Quand un script Gappa contient plusieurs indices de bissection, ils ne sont pas appliqués récursivement. Gappa cherche d'abord à améliorer le graphe initial à l'aide du premier indice. Les autres indices ne sont pas utilisés dans les graphes créés. Puis l'outil applique le deuxième indice au graphe initial. Là encore, les autres indices ne sont pas appliqués aux graphes créés. Et ainsi de suite. Cette façon d'indiquer les bissections permet de limiter l'explosion du nombre de graphes de preuve. Il n'y en aura ici que $1 + 3 + 5 + 4 = 13$, mais ce nombre réduit peut ne plus être suffisant à Gappa pour prouver la proposition.

5.3.3 Découpage ciblé

Il reste le cas des découpages qui n'indiquent qu'une expression sans préciser comment se fait la découpe. Le comportement de Gappa dépend alors de la présence ou non d'expressions à gauche du dollar. S'il n'y en a pas, les découpages sont alors équivalents à des découpages en quatre sous-intervalles. Cela permet à l'utilisateur de tester rapidement l'impact que pourrait avoir des bissections selon telles ou telles sous-expressions.

Si des expressions sont indiquées à gauche du dollar par contre, Gappa effectue un découpage en sous-intervalles de telle sorte que la partie de la conclusion qui correspond à ces variables soit prouvée. Dans le script 5.3, il est demandé à Gappa des sous-intervalles de $[0, 1000]$ de plus en plus petits jusqu'à avoir prouvé sur chacun d'eux la propriété correspondant à $\frac{1}{x}$ dans la conclusion, c'est-à-dire l'inégalité $\frac{1}{x} \leq 2$. Gappa ne s'acharne cependant pas à encadrer $\frac{1}{x}$ s'il est capable de prouver la conclusion sur un des sous-intervalles, par exemple s'il obtient une contradiction entre les hypothèses ou une preuve de $x^2 \leq 1$.

Script Gappa 5.3 Découpage avec expression cible

```
{ x in [0,1000] -> x * x <= 1 \/ 1 / x <= 2 }
1 / x $ x;
```

Soit α un nombre réel strictement compris entre $\frac{1}{2}$ et 1. À partir de la propriété $x \in [0, \alpha]$, il est possible de déduire $x^2 \leq 1$. Quant à la propriété $x \in [\alpha, 1000]$, on peut en déduire $\frac{1}{x} \leq 2$. La preuve de la proposition peut donc se faire en considérant séparément les cas $x \leq \alpha$ et $x \geq \alpha$. C'est ce que fait Gappa dans sa preuve : il prend $\alpha = \frac{1000}{1024}$ comme frontière et décompose le problème en deux sous-problèmes.

Découpage récursif

Pour obtenir cette valeur de α, Gappa fait une première tentative de découpage avec les deux encadrements $x \in [0, 500]$ et $x \in [500, 1000]$. Sur le deuxième,

il arrive effectivement à prouver $\frac{1}{x} \leq 2$ puisqu'il obtient une contradiction avec $\frac{1}{x} \geq 2$. Sur le premier sous-intervalle, l'outil est par contre incapable de prouver la proposition. Il découpe alors cet intervalle en deux nouveaux sous-intervalles. Là encore, la proposition est prouvable sur le sous-intervalle de droite, mais pas sur celui de gauche. Et ainsi de suite, Gappa découpe récursivement les sous-intervalles de gauche en deux jusqu'à atteindre $x \in [0, \frac{1000}{1024}]$. Sur ce dernier encadrement, il obtient une contradiction avec $x^2 \geq 1$ et peut donc en déduire une preuve de $x^2 \leq 1$.

Gappa a ainsi découpé l'encadrement $x \in [0, 1000]$ jusqu'à obtenir une contradiction sur tous les sous-intervalles considérés. À ce moment-là, onze graphes de preuve ont été générés pour les sous-intervalles $[0, \alpha]$, $[\alpha, \frac{1000}{512}]$, ..., $[250, 500]$ et $[500, 1000]$. La situation n'est pas vraiment satisfaisante puisqu'elle conduit à une preuve bien plus longue que si des graphes de preuve avaient été générés pour les seuls intervalles $[0, \alpha]$ et $[\alpha, 1000]$. Gappa termine donc en réalisant une fusion des différents sous-cas afin de réduire leur nombre.

Tout comme les simplifications décrites au paragraphe 2.3.3, ce traitement est caractéristique de l'approche Gappa. Avec un outil classique employant l'arithmétique d'intervalles, l'encadrement d'une expression est l'unique objectif, l'outil n'a pas besoin d'effectuer d'autres opérations. Avec Gappa, l'utilisateur attend non seulement un encadrement mais aussi une preuve formelle de cet encadrement. Pour qu'elle soit utilisable, un assistant de preuves doit pouvoir l'interpréter en un temps raisonnable. Les traitements effectués par Gappa doivent donc, non seulement obtenir des encadrements fins, mais aussi générer des preuves courtes.

Fusion des graphes

La méthode consiste à tester, pour chaque sous-intervalle obtenu, si la proposition reste vraie quand on le fusionne avec le sous-intervalle qui le précède. Dans l'affirmative, les deux sous-intervalles sont retirés et remplacés par leur réunion. Puis l'outil passe au sous-intervalle suivant.

Par exemple, Gappa considère le sous-intervalle $[\frac{1000}{512}, \frac{1000}{256}]$ et se rend compte que la proposition reste vraie quand il le fusionne avec le sous-intervalle précédent, c'est-à-dire $[\alpha, \frac{1000}{512}]$. L'intervalle $[\alpha, \frac{1000}{256}]$ leur est alors substitué dans la liste. Puis Gappa passe au sous-intervalle suivant de la liste triée et recommence jusqu'à ce qu'il n'y ait plus que les deux sous-intervalles finaux au lieu des onze initiaux. Il est possible de modifier l'exemple pour obtenir des scripts pour lesquels la réduction de la preuve est arbitrairement grande par cette technique simple.

Étudions le coût de la méthode. La bissection produit un découpage en n sous-intervalles après avoir étudié $2 \cdot n + 1$ sous-cas. Le mécanisme de fusion cherche alors à fusionner chaque intervalle au plus une fois avec celui qui précède, ce qui correspond donc à l'étude de $n - 1$ sous-cas supplémentaires. Le surcoût est donc de l'ordre de 50%.

Optimalité du découpage

Vis-à-vis du découpage initialement obtenu, la fusion réalisée est optimale : aucune fusion des sous-intervalles ne permet de réduire plus le nombre de sous-cas. Cela suppose cependant qu'une propriété prouvable par Gappa sur un intervalle I est prouvable sur tout sous-intervalle de I. Cette hypothèse n'est pas très contraignante parce qu'elle est vérifiée au moins pour les fractions rationnelles [Moo79] et lorsque les bornes des intervalles sont arrondies.

La justification d'optimalité peut se faire en considérant trois découpages repérés par les points séparant les sous-intervalles. Tout d'abord le découpage initial construit par bissection $D = (x_0, \ldots, x_n)$. Ensuite celui obtenu par Gappa $D_g = (x_0, x_{g(1)}, \ldots, x_{g(p-1)}, x_n)$. Enfin, un découpage optimal $D_o = (x_0, x_{o(1)}, \ldots, x_{o(q-1)}, x_n)$ qui s'appuie sur les points de D et pour lequel Gappa saurait prouver la proposition sur chacun des sous-intervalles $[x_{o(i)}, x_{o(i+1)}]$. Par définition, le découpage optimal ne contient pas plus de sous-intervalles que celui obtenu par Gappa, donc $p \geq q$. L'objectif est de montrer que p vaut q en réalité. Appelons i le plus grand indice pour lequel les découpages vérifient $g(i) \geq o(i)$. Si $o(i) = n$ alors les deux découpages ont la même longueur.

Supposons par l'absurde que ce ne soit pas le cas. Cela implique $i < q$ et $g(i+1) < o(i+1)$. Au moment où Gappa a considéré l'intervalle élémentaire $[x_{g(i+1)}, x_{g(i+1)+1}]$, l'intervalle $[x_{g(i)}, x_{g(i+1)}]$ existait déjà et il a donc cherché à les fusionner. Puisque cette fusion a échoué, cela signifie que Gappa n'a pas réussi à prouver la propriété sur l'intervalle $[x_{g(i)}, x_{g(i+1)+1}]$. Or Gappa est capable de la prouver sur l'intervalle $[x_{o(i)}, x_{o(i+1)}]$ qui en est un sur-ensemble puisque $o(i) \leq g(i)$ et $g(i+1) < o(i+1)$. C'est ici que réside la contradiction. Le découpage obtenu par Gappa a donc une longueur optimale par rapport aux points obtenus par bissection.

Si l'on considère maintenant un découpage D' qui ne s'appuie par sur les points obtenus par bissection, comment se compare-t-il à un découpage de Gappa ? Dans le pire des cas (du point de vue de Gappa), sa longueur est deux fois moindre. En effet, un intervalle de D' ne peut pas couvrir deux intervalles complets de D_g, sinon l'outil les aurait fusionnés. Le découpage D_g est donc intéressant du point de vue de la longueur de la preuve.

5.4 Règles de réécriture

Comme détaillé dans les chapitres précédents, l'utilisation de réécritures permet de contourner les problèmes de décorrélation lors de l'évaluation par arithmétique d'intervalles d'expressions représentant des erreurs. Gappa contient ainsi toute une base de règles permettant de réécrire les expressions à encadrer sous des formes contenant des termes potentiellement moins corrélés. L'utilisateur peut ajouter ses propres règles pour exprimer des identités mathématiques plus complexes. Toutes ces réécritures ne sont appliquées qu'aux prédicats d'encadrement.

5.4.1 Règles prédéfinies

Les règles peuvent se décomposer en plusieurs catégories comme indiqué au paragraphe 3.2. Il y a tout d'abord les règles qui s'appliquent à des expressions d'erreur entre deux expressions dont les arbres syntaxiques commencent de façon similaire. Afin de pouvoir capturer l'erreur absolue de l'addition, il y a ainsi une règle réécrivant les soustractions d'additions :

$$(a+b)-(c+d) \quad \rightarrow \quad (a-c)+(b-d).$$

Si a et c désignent une même expression, Gappa doit évaluer par intervalles le terme $a-a$. C'est la pire forme de corrélation, mais Gappa sait la détecter et l'encadrer par $[0,0]$. L'encadrement est donc bon mais son certificat est inutilement complexe : il contient une preuve de $a-a \in [0,0]$ puis calcule l'intervalle $[0,0]+I$. Pour éviter ces étapes, Gappa n'applique pas la règle quand a et c (ou b et d) désignent la même expression. Deux autres règles prennent le relais pour traiter directement ces situations où une même expression se trouve de part et d'autre de l'opérateur de soustraction :

$$\begin{aligned}(a+b)-(a+c) \quad &\rightarrow \quad b-c,\\ (a+b)-(c+b) \quad &\rightarrow \quad a-c.\end{aligned}$$

Que ce soit pour l'addition, la soustraction ou la multiplication, les règles qui concernent leurs erreurs absolues sont purement syntaxiques : elles s'appliquent quelles que soient les valeurs prises par les sous-expressions. Dans le cadre des erreurs relatives, ce n'est plus le cas à cause de la division. Voici par exemple la réécriture employée pour l'erreur relative entre les produits $a \cdot b$ et $a \cdot c$.

$$\frac{a \cdot b - a \cdot c}{a \cdot c} \quad \rightarrow \quad \frac{b-c}{c}.$$

Pour pouvoir appliquer cette règle, il faut prouver que ni a ni c ne vaut zéro. Gappa le garantit par deux propriétés $\mathrm{ABS}(a, [\underline{a}, \overline{a}])$ et $\mathrm{ABS}(c, [\underline{c}, \overline{c}])$ vérifiant les inégalités $\underline{a} > 0$ et $\underline{c} > 0$.

5.4.2 Expressions approchées et expressions exactes

Les règles qui décomposent les erreurs en faisant intervenir des expressions intermédiaires constituent une catégorie à part. C'est le cas, par exemple, de la règle dédiée à l'erreur absolue :

$$a-b \quad \rightarrow \quad (a-c)+(c-b).$$

Pour évaluer l'erreur entre a et b, Gappa cherche si elle ne s'exprime pas plus facilement sous la forme de deux erreurs : l'erreur entre a et c d'une part et celle entre c et b d'autre part. La différence par rapport aux règles précédentes est que le terme c n'apparaît pas dans l'expression à réécrire. Gappa doit l'inférer à partir d'informations annexes.

Pour cela, l'outil maintient une liste de couples d'expressions. Un couple (e, f) signifie que l'expression e prend des valeurs proches de celles de f. Du point de vue de Gappa, cela signifie surtout que l'erreur entre e et f est probablement simple à évaluer et qu'il peut donc être intéressant de la faire apparaître chaque fois que possible. Ainsi, la règle précédente est appliquée si (a, c) fait partie de ces couples. Il en existe une variante mathématiquement identique qui est appliquée quand le couple est (c, b) et non pas (a, c)[2].

Gappa applique plusieurs heuristiques pour construire sa liste de couples. Tout d'abord, si une expression est de la forme $\circ(e)$, $(\circ(e), e)$ est ajouté à l'ensemble des couples considérés. Ensuite, si une hypothèse est de la forme $e - f \in I$ ou $\frac{e-f}{f} \in I$, l'outil suppose qu'elle représente en fait un encadrement d'erreur absolue ou relative et ajoute le couple (e, f) à la liste. Cette dernière heuristique s'applique aussi quand l'expression encadrée est $|e - f|$ ou $|\frac{e-f}{f}|$.

Tout cela peut ne pas être suffisant et la syntaxe `e ~ f;` permet alors à l'utilisateur d'ajouter ses propres couples (e, f). Dans le script 5.4, l'utilisateur demande à Gappa de déduire de $\lfloor a \rfloor + b \in [5, 10]$ un encadrement de $a + b$. S'il ne précise pas que $\lfloor a \rfloor + b$ est une valeur qui approche $a + b$, Gappa est incapable de prouver $a + b \in [5, 11]$.

Script Gappa 5.4 Paire expression approchée – expression exacte

```
{ int<dn>(a) + b in [5,10] -> a + b in ? }
int<dn>(a) + b ~ a + b;
```

5.4.3 Règles utilisateur

La définition de couples permet donc à Gappa de généraliser les règles de réécriture qu'il connaît et sait appliquer. Mais cela peut ne pas suffire à prouver une proposition, en particulier quand les expressions ont besoin d'être réécrites en profondeur. L'utilisateur peut alors fournir ses propres règles.

Script Gappa 5.5 Règle utilisateur de réécriture

```
{ x in [0,1] -> x * (1 - x) in [0, 0.25] }
x * (1 - x) -> 1/4 - (x - 1/2) * (x - 1/2);
```

Ainsi, la meilleure façon de borner l'expression $x \cdot (1 - x)$ par arithmétique d'intervalles est de la réécrire en $\frac{1}{4} - (x - \frac{1}{2})^2$ afin de faire disparaître la corrélation entre les termes x et $1 - x$. Aucun des théorèmes de Gappa ne lui permet d'effectuer cette réécriture, l'utilisateur doit donc la lui fournir. C'est le cas du script 5.5.

[2] Si Gappa connaît les deux couples (a, c) et (c, b) à la fois, les deux règles feront parties des chemins de calcul. Comme elles propagent toutes deux l'encadrement de $(a - c) + (c - b)$, le résultat de la dernière règle appliquée n'améliorera pas celui de la première. L'outil ne le gardera donc pas dans le graphe de preuve.

Un indice de la forme « e_1 -> e_2 » indique à l'outil que, s'il a besoin d'encadrer une expression e_1, il peut utiliser l'intervalle obtenu en encadrant l'expression e_2. Pour que cette manipulation soit valable, il suffit que les expressions e_1 et e_2 aient la même valeur dans tous les cas.

Ces règles de réécriture posées par l'utilisateur introduisent de nouveaux axiomes au sein de la preuve. Pour limiter le risque d'erreur, Gappa cherche à vérifier que les membres de gauche et de droite de la règle sont effectivement égaux. Idéalement, il devrait aussi en générer une preuve pour éviter que les réécritures ne soient de simples axiomes dans le certificat final. Dans cette optique, les méthodes probabilistes [Sch80] ne présentent qu'un intérêt limité puisqu'elles ne garantissent pas l'identité d'expressions multivariées. Elles pourraient cependant être utilisées pour repérer rapidement les fautes de frappe de l'utilisateur.

Pour effectuer la comparaison des deux expressions, Gappa manipule leur différence sous forme symbolique : un quotient d'expressions polynomiales à coefficients entiers. Ces expressions polynomiales sont sous forme normale et les sous-termes qui y jouent le rôle d'indéterminées sont les valeurs numériques non entières, les variables universellement quantifiées ainsi que les résultats d'arrondi, de valeur absolue et de racine carrée. Si la forme normale au numérateur du quotient est nulle, alors les deux expressions sont égales. Gappa profite aussi de la normalisation pour repérer les expressions qui doivent prendre des valeurs non nulles afin que la réécriture soit licite.

Malheureusement, ces formes normales sont coûteuses à manipuler, aussi bien en temps qu'en mémoire, en particulier quand il s'agit de réécrire les erreurs relatives d'une fonction longue. Du point de vue de la théorie de la complexité, ce n'est pas surprenant : vérifier qu'un simple polynôme est nul est déjà un problème notoirement difficile [KI04]. Il doit cependant y avoir des pistes à explorer pour réduire les ressources nécessaires. D'une part, d'autres formes normales [GM05] pourraient être essayées. D'autre part les expressions à comparer sont loin d'être quelconques et il n'est peut-être pas nécessaire de les normaliser.

Chapitre 6

Vérification automatique

Après avoir vérifié une proposition logique, Gappa génère une preuve formelle de sa validité. Ce chapitre présente les méthodes employées pour qu'un assistant de preuves puisse traiter automatiquement le contenu de cette preuve.

La raison d'être de Gappa est la vérification de propriétés qui garantissent la fiabilité de programmes s'appuyant sur des arithmétiques approchées. Si une proposition logique est une transcription fidèle des spécifications d'une part et de l'implantation d'un code d'autre part, alors le code en question est correct si Gappa affirme que la proposition est vraie. Deux questions se posent alors. Premièrement, cette affirmation est-elle une garantie suffisante ou Gappa a-t-il pu se tromper ? Deuxièmement, la proposition prouvée ne correspond qu'à la partie numérique du travail de certification. Comment interfacer ce résultat de validité avec le reste de la certification ?

Plusieurs approches sont envisageables. La première est tout simplement la confiance aveugle dans le résultat donné par Gappa. Cette approche n'est pas nécessairement déraisonnable. En effet, les méthodes employées par Gappa sont conçues pour fournir des résultats garantis et leur implantation s'appuie sur des bibliothèques spécialisées qui ne sont pas spécifiques à l'outil mais couramment employées. Ne pas avoir à certifier les résultats favorise un développement rapide de l'outil et donc un plus grand nombre de ses fonctionnalités. Quant à l'inclusion des résultats au sein de preuves plus générales, elle se fait alors par le biais de nouveaux axiomes. Du point de vue de Gappa, les graphes de preuve décrits au chapitre 5 n'ont plus besoin d'être conservés dans leur intégralité : seuls les nœuds aux feuilles, c'est-à-dire les derniers encadrements obtenus, sont nécessaires au fonctionnement de l'outil.

La confiance en l'outil pourrait être accrue par sa certification partielle ou totale, ainsi que celle des bibliothèques sur lesquelles il s'appuie. Cela éliminerait les dernières appréhensions dans la validité des résultats produits. Ce travail serait

cependant colossal et n'éviterait pas pour autant d'avoir à représenter les résultats par des axiomes dans les preuves plus générales. Une autre approche serait alors non pas de garantir que Gappa produit systématiquement un résultat correct mais de valider le résultat produit pour une proposition donnée à l'aide d'un certificat. Ce certificat pourrait alors prendre la place de l'axiome associé au résultat.

Cette approche présente cependant une grosse contrainte : toute méthode employée par Gappa doit pouvoir générer un certificat concernant les encadrements qu'elle calcule. Cela explique que les méthodes détaillées aux chapitres précédents soient d'apparence simple : les adapter pour qu'elles génèrent des certificats est réalisable en un temps raisonnable. Malgré leur simplicité, le nombre de fois où elles sont appliquées lors d'une exécution les met hors de portée d'un assistant de preuves tel que Coq. En effet, l'arithmétique y est actuellement interprétée bit à bit, ce qui la rend inadaptée à l'usage intensif qu'en fait Gappa.

L'outil va donc être une entité distincte de l'assistant de preuves, afin de ne pas souffrir de la lenteur[1] des calculs de ce dernier. Cette lenteur indésirable est cependant la conséquence de la vérification approfondie effectuée par l'assistant de preuves, vérification qui est quant à elle désirée. Il est donc important de limiter autant que faire se peut le volume des calculs présents dans le certificat, même si cela requiert plus de travail de la part de Gappa. C'est dans cette optique qu'est effectué le travail de fusion des bissections présenté au paragraphe 5.3.3.

6.1 Certificats et approche oracle

L'outil est donc implanté sous la forme d'un oracle. Étant donnée une proposition logique, Gappa affirme qu'elle est valide ou déclare forfait. Il fournit aussi des indications sous la forme d'un certificat pour qu'un assistant de preuves puisse aboutir au même résultat. Comme Gappa n'est pas lié à un assistant de preuves donné, on pourrait imaginer qu'il génère des certificats pour d'autres assistants que Coq.

Intéressons-nous à ces indications. Tout d'abord, il est bien évident que des données inutiles ne feraient qu'alourdir le certificat. Toute trace des tentatives qui se sont soldées par un échec doit donc être retirée du certificat : parmi les très nombreux[2] chemins de preuve que Gappa a explorés, seul celui qui mène réellement à la preuve de la proposition a un quelconque intérêt du point de vue du certificat.

Sont aussi à éviter les indications qui conduiraient à un résultat intermédiaire trop puissant, même si la preuve de la proposition en découle ensuite. Considérons l'exemple de l'erreur d'approximation décrite au paragraphe 3.3.1. Son expression fait intervenir des fonctions élémentaires dont les résultats sont encadrés par des intervalles à bornes rationnelles. La précision des bornes de ces encadrements est

[1] À précision donnée, la vérification d'une opération intervalle à l'aide des entiers relatifs standards de Coq demande plusieurs milliers de fois le temps nécessaire à Gappa pour l'effectuer.

[2] En l'absence de fusion des sous-cas de bissection, pour chaque théorème retenu dans la preuve finale, Gappa a cherché à appliquer environ une centaine de théorèmes.

fixée par un paramètre indiquant à quel ordre tronquer les séries les approchant. Ce paramètre est l'une des données fournies par l'oracle : celui-ci affirme que, si l'assistant PVS fixe le paramètre à cette valeur, il sera capable de prouver la propriété demandée. Le coût pour évaluer ces séries augmente rapidement en fonction de l'ordre auquel elles sont tronquées. L'oracle a donc tout intérêt à effectuer quelques calculs supplémentaires pour s'assurer que le paramètre a vraiment besoin d'avoir une valeur aussi élevée.

Un dernier point concernant les certificats est le niveau de détails auquel ils doivent descendre. Vont-ils conduire l'assistant de preuve pas à pas ou vont-ils ne lui donner que les grandes lignes ? Les auteurs de [HT98] s'intéressent à l'utilisation d'outils de calcul symbolique comme oracles et prennent l'exemple du calcul d'un plus grand diviseur commun pour illustrer ce point. Supposons que l'une des étapes d'une preuve soit le calcul du PGCD de deux polynômes p et q. Le certificat pourrait alors se contenter d'indiquer que ce PGCD doit être calculé et l'assistant de preuves pourrait alors appliquer l'algorithme d'Euclide pour effectivement le calculer. Si le certificat contenait comme détail supplémentaire la valeur d de ce PGCD, l'assistant de preuves ne serait pas plus avancé pour autant : la vérification de la propriété « commun diviseur » en serait simplifié mais il resterait à vérifier que c'est le « plus grand ». Le certificat serait bien plus intéressant s'il contenait en plus du résultat d les coefficients de Bézout u et v. L'assistant de preuves n'aurait plus alors qu'à vérifier l'égalité $u \cdot p + v \cdot q = d$ et quelques propriétés sur les polynômes pour s'assurer que d est effectivement le PGCD.

Pour [DMM05], nous avons fait le choix de ne fournir à PVS que le découpage en sous-intervalles et l'ordre auquel tronquer les séries des fonctions élémentaires. Le détail des calculs à effectuer pour encadrer les expressions est laissé à l'appréciation de l'assistant de preuves. Celui-ci dispose en effet d'une bibliothèque d'arithmétique par intervalles [ML05] lui permettant d'encadrer directement des expressions compliquées.

Dans Gappa, nous avons au contraire décidé de produire des certificats de très bas niveau. Chaque opération arithmétique de base y est détaillée, c'est-à-dire que le certificat contient son résultat et le nom du théorème à appliquer pour valider ce résultat intermédiaire. La taille du certificat augmente alors considérablement et son contenu n'est plus lisible. Comme dans le cas du PGCD, ce niveau de détail accru permet d'appliquer des algorithmes de vérification moins coûteux. Il permet aussi de limiter la quantité de calculs nécessaires dans ces algorithmes en fournissant des encadrements aussi simplifiés que possible, comme cela est décrit au paragraphe 2.3.3.

6.2 Vérification par le calcul

Considérons la somme de deux expressions x et y. Connaissant les encadrements $x \in [a, b]$ et $y \in [c, d]$ avec a, b, c et d des nombres dyadiques explicites, la vérification de l'encadrement $x + y \in [u, v]$ peut se faire à l'aide de l'arithmétique

d'intervalles :

$$\forall a, b, c, d, u, v \in \mathbb{D}, \quad \forall x, y \in \mathbb{R}, \quad x \in [a, b] \ \wedge \ y \in [c, d] \quad \Rightarrow \\ [a, b] + [c, d] = [u, v] \quad \Rightarrow \quad x + y \in [u, v].$$

Ce théorème est cependant gênant parce que l'hypothèse $[a, b] + [c, d] = [u, v]$ est trop forte. En la remplaçant par $[a, b] + [c, d] \subseteq [u, v]$, le théorème reste vrai et il peut s'appliquer même si l'intervalle $[u, v]$ présent dans le certificat est trop large, ce qui risque de se produire à cause des arrondis détaillés au chapitre 2.

Ce théorème nécessite à la fois une théorie suffisamment complète de l'arithmétique d'intervalles et son implantation dans l'assistant de preuve. Si elles sont absentes, il n'est pas nécessaire de les ajouter : on peut simplifier le théorème en revenant directement aux définitions des opérateurs par intervalles. Cela donne alors :

$$\forall a, b, c, d, u, v \in \mathbb{D}, \quad \forall x, y \in \mathbb{R}, \quad x \in [a, b] \ \wedge \ y \in [c, d] \quad \Rightarrow \\ u \leq a + c \ \wedge \ b + d \leq v \quad \Rightarrow \quad x + y \in [u, v].$$

6.2.1 Opérations sur les nombres dyadiques

Lorsque le théorème d'addition est appliqué durant la vérification du certificat, les hypothèses $x \in [a, b]$ et $y \in [c, d]$ sont exactement des résultats déjà prouvés auparavant. L'assistant de preuves n'a donc aucune difficulté à constater qu'elles sont effectivement satisfaites. Pour l'hypothèse $u \leq a + c \wedge b + d \leq v$, la situation n'est pas aussi simple. Pour vérifier qu'elle est satisfaite, il faut calculer les nombres dyadiques $a + c$ et $b + d$ et les comparer à u et v. C'est ici qu'intervient la bibliothèque Pff [Bol04].

La bibliothèque Pff supporte des nombres de la forme $m \cdot \beta^e$ avec m et e des entiers relatifs et β un entier valant au moins 2. À β fixé, ces nombres forment un ensemble fermé pour les opérations d'addition, soustraction et multiplication. Ces opérations sur les nombres suffisent à vérifier les opérations correspondantes étendues aux intervalles. Comme le montre le paragraphe 6.3, elles sont aussi suffisantes pour traiter les divisions et racines carrées d'intervalles. En ce qui concerne Gappa, la valeur de β est fixée à 2 pour calculer avec des nombres dyadiques.

Il faut noter que Coq utilise une représentation des entiers relatifs par leur signe et une chaîne de bits, bit de poids faible en tête. Sur cette représentation, les opérations de décalages sont particulièrement faciles à faire. Alors que Pff est obligé d'utiliser des multiplications et des élévations à la puissance de β pour être générique, Gappa n'a pas besoin d'une telle complexité. La bibliothèque Pff a ainsi été abandonnée au profit d'une implantation de l'arithmétique dyadique spécifique à Gappa. Comme le montrera le paragraphe 6.3.4, cette optimisation permet de réduire le temps nécessaire à Coq pour interpréter des preuves d'environ 12%. Cela permet aussi de considérablement réduire les dépendances sur d'autres bibliothèques de théorèmes qu'auraient des preuves de propositions n'utilisant que les opérateurs arithmétiques sur les nombres réels.

Cette implantation s'est faite en représentant les nombres dyadiques de la même façon que dans Pff, c'est-à-dire par des paires d'entiers relatifs. Étant donnés deux dyadiques (m_1, e_1) et (m_2, e_2), l'algorithme de multiplication renvoie le dyadique $(m_1 \cdot m_2, e_1 + e_2)$. Les nombres dyadiques sont définis par plongement dans les nombres réels, c'est-à-dire qu'il existe un opérateur $\mathcal{R}(m, e) = m \cdot 2^e$. La correction de l'algorithme de multiplication $\times$ est alors prouvée en vérifiant $\mathcal{R}(m_1, e_1) \cdot \mathcal{R}(m_2, e_2) = \mathcal{R}((m_1, e_1) \times (m_2, e_2))$.

Addition, soustraction et comparaison suivent une approche similaire, si ce n'est qu'elles sont toutes les trois basées sur un même algorithme. Étant donné deux dyadiques (m_1, e_1) et (m_2, e_2), il calcule trois entiers n_1, n_2 et e tels que $\mathcal{R}(m_1, e_1) = \mathcal{R}(n_1, e)$ et $\mathcal{R}(m_2, e_2) = \mathcal{R}(n_2, e)$. Somme et différence valent alors $(n_1 + n_2, e)$ et $(n_1 - n_2, e)$. La comparaison se fait quant à elle directement sur les entiers n_1 et n_2. Il y a aussi quelques comparaisons spécifiques quand l'un des opérandes vaut zéro : il suffit de regarder le signe de l'autre opérande, aucun décalage n'est nécessaire.

Comme les constantes employées par l'utilisateur de Gappa ne sont pas nécessairement des nombres dyadiques, il faut aussi pouvoir prendre en compte les nombres décimaux, c'est-à-dire des réels de la forme $m \cdot 10^e$ avec m et e entiers relatifs. Gappa traite de tels nombres en les encadrant par des intervalles de nombres dyadiques. Il n'est pas nécessaire de savoir calculer avec des nombres décimaux, il suffit de savoir les comparer à des nombres dyadiques. La bibliothèque Coq met à profit l'égalité $10^e = 5^e \cdot 2^e$ pour n'effectuer que des multiplication par 5 : celles de facteur 2 seront directement prises en charge par la comparaison de nombres dyadiques.

Script Coq 6.1 Comparaison entre un nombre dyadique et un nombre décimal

```
Definition Dcompare (x : float2) (y : float10) :=
 let m := Fnum10 y in let e := Fexp10 y in
 match e with
 | Zpos p => Fcomp2 x (Float2 (m * Zpower_pos 5 p) e)
 | Zneg p => Fcomp2 (Float2 (Fnum x * Zpower_pos 5 p)
                            (Fexp x)) (Float2 m e)
 | Z0 => Fcomp2 x (Float2 m 0)
 end.

Lemma Dcompare_correct :
 forall x : float2, forall y : float10,
 match (Dcompare x y) with
 | Lt => (x < y)%R
 | Eq => (x = y :>R)%R
 | Gt => (x > y)%R
 end.
```

Le script Coq 6.1 présente l'algorithme employé et l'énoncé de son lemme de correction. Les fonctions `Fnum` et `Fexp` permettent d'accéder à la mantisse et à

l'exposant d'un nombre dyadique, tandis que `Fnum10` et `Fexp10` font de même pour un nombre décimal. La fonction `Float2` permet quant à elle de construire un nombre dyadique à partir d'une mantisse et d'un exposant. À partir du nombre dyadique x et du nombre décimal y, l'algorithme construit deux nouveaux nombres dyadiques en fonction du signe de l'exposant du nombre décimal, puis il les compare (`Fcomp2`). Le lemme de correction assure quant à lui que, quelque soit le résultat renvoyé par l'algorithme, il est compatible avec la comparaison entre les valeurs réelles de x et y.

6.2.2 Fonctions booléennes

Dans Pff, addition et multiplication sont toutes deux implantées sous forme de fonctions récursives et Coq est capable d'évaluer leur résultat pour peu que leurs opérandes soient des constantes explicites. Les comparaisons sont fournies par des fonctions booléennes elles aussi calculables.

En notant $\leq_2$ l'opérateur qui compare deux nombres dyadiques et renvoie un booléen, l'hypothèse arithmétique du théorème d'addition devient alors $((u \leq_2 a+c) = true) \wedge ((b+d \leq_2 v) = true)$. Coq peut vérifier que cette conjonction est satisfaite en réduisant les termes jusqu'à ce que ses membres soient deux égalités $true = true$. Il s'agit en fait ici d'un mécanisme de réflexion [Bou97] entre les propriétés et les booléens et il est possible de l'étendre à la conjonction afin que l'hypothèse arithmétique ne soit plus qu'une simple égalité entre deux expressions booléennes dont l'une est simplement $true$. Ce mécanisme est cependant implicite à chacun des théorèmes d'arithmétique ; sa correction est prouvée pour chacun d'eux et non pas de façon générale. Le script 6.2 en décrit l'énoncé en Coq.

Script Coq 6.2 Définition en Coq de l'addition d'intervalles

```
Definition add_helper (xi yi zi : FF) :=
 Fle2 (lower zi) (Fplus2 (lower xi) (lower yi)) &&
 Fle2 (Fplus2 (upper xi) (upper yi)) (upper zi).

Theorem add :
 forall x y : R, forall xi yi zi : FF,
 BND x xi -> BND y yi ->
 add_helper xi yi zi = true ->
 BND (x + y) zi.
```

Le type `FF` représente des paires de nombres dyadiques, c'est-à-dire des intervalles dont les bornes sont accessibles par les fonctions `lower` et `upper`. Le prédicat `BND` correspond à l'encadrement d'une expression réelle par un intervalle dyadique. Les fonctions `Fle2` et `Fplus2` sont respectivement la comparaison et l'addition de deux nombres dyadiques.

Le script Coq 6.3 montre comment le théorème est ensuite appliqué dans le certificat. Les termes `p7`, `p11` et `p6` sont tous les trois des encadrements. Une fois

les hypothèses `p7` et `p11` chargés sous les noms `h0` et `h1`, il reste à prouver le but `p6`. Pour ce faire, le théorème d'addition est appliqué. Il nécessite deux hypothèses d'encadrement et une hypothèse d'arithmétique d'intervalles. Les encadrements lui sont fournis par les deux hypothèses `h0` et `h1`, c'est-à-dire `p7` et `p11`. Il reste alors à prouver l'hypothèse arithmétique, c'est-à-dire une égalité entre deux booléens. Un appel à la tactique `finalize` résout ce but : elle cherche à vérifier que les deux membres de l'égalité se réduisent à $true$ (si le certificat est correct) et sont donc égaux par réflexivité de l'égalité. Cela conclut la preuve du lemme `t11`, l'une des nombreuses étapes d'un certificat généré par Gappa.

Script Coq 6.3 Application d'un théorème de la bibliothèque de support

```
Lemma t11 : p7 -> p11 -> p6.
 intros h0 h1.
 apply add with (1 := h0) (2 := h1) ;
 finalize.
Qed.
```

Ces notions de réduction et réflexion sont relativement spécifiques à l'emploi de Coq pour interpréter le certificat. Gappa n'est pas pour autant lié à Coq, des approches différentes sont accessibles aux autres assistants de preuve. Dans notre travail sur le théorème de Taylor par intervalles en PVS [DMM05], nous avons ainsi montré que l'assistant PVS est tout à fait capable d'interpréter des certificats utilisant de l'arithmétique d'intervalles à bornes rationnelles.

6.3 Calculs simplifiés

Comme mentionné précédemment, le prix payé par Coq pour des calculs sûrs est la lenteur de leur exécution. Il est donc important que les théorèmes employés soient les plus simples possibles. Plus la fonction booléenne qui sert d'hypothèse arithmétique sera simple, plus vite Coq vérifiera que les théorèmes sont appliqués correctement.

6.3.1 Spécialisation des théorèmes

Voyons comment simplifier le travail de vérification en partant de l'exemple du théorème de multiplication : étant donnés x et y deux réels, alors leur produit $x \cdot y$ appartient à l'intervalle $[u, v]$ si

$$x \in [a, b], y \in [c, d] \text{ et } [a, b] \times [c, d] \subseteq [u, v].$$

Rappel : l'opérateur de multiplication d'intervalles est défini par

$$[a, b] \times [c, d] = [\min(a \cdot c, b \cdot c, a \cdot d, b \cdot d), \max(a \cdot c, b \cdot c, a \cdot d, b \cdot d)].$$

Une première approche est d'implanter directement dans le langage de l'assistant de preuves un théorème inspiré de la multiplication d'intervalles, comme c'est le cas pour l'addition. Pour satisfaire les hypothèses de ce théorème, il faut effectuer les produits de quatre paires de nombres dyadiques puis repérer les valeurs minimale $\underline{m}$ et maximale $\overline{m}$ parmi ces quatre produits. Comme pour l'addition, le test d'inclusion se fait en testant les inégalités $e \leq \underline{m}$ et $\overline{m} \leq f$.

La plupart du temps, il n'est cependant pas nécessaire d'effectuer l'ensemble de ces quatre produits. C'est par exemple le cas lorsque les inégalités $b < 0$ et $c > 0$ sont vérifiées. Tous les éléments de l'intervalle $[a, b]$ sont alors négatifs tandis que ceux de $[c, d]$ sont positifs. Les valeurs $\underline{m}$ et $\overline{m}$ sont donc respectivement $a \cdot d$ et $b \cdot c$. Les hypothèses du théorème précédent se réduisent alors à

$$x \in [a, b], y \in [c, d], b < 0, c > 0, u \leq a \cdot d \text{ et } b \cdot c \leq v.$$

Parmi ces hypothèses, les comparaisons à zéro sont triviales : il suffit de tester le signe de la mantisse des nombres dyadiques. Les seules opérations coûteuses qui restent sont donc deux multiplications et deux comparaisons, au lieu des quatre multiplications et six comparaisons initialement présentes. En contrepartie de ces hypothèses simplifiées, il faut prévoir autant de théorèmes que de cas différents. Dans le cas de la multiplication, neuf théorèmes sont ainsi nécessaires suivant d'une part que $a > 0$, $b < 0$ ou $a \leq 0 \leq b$ et d'autre part que $c > 0$, $d < 0$, $c \leq 0 \leq d$. Gappa indique dans le certificat quel théorème l'assistant de preuves doit appliquer.

Script Gappa 6.1 Performance de la multiplication

```
{ x in [-2.1,2.1] /\ y in [-2.1,2.1] /\ z in [-2.1,2.1] ->
  x * y * z * x * y * z in ? }
$ x, y, z;
```

Le script Gappa 6.1 produit une preuve contenant $5 \cdot 4^3$ applications d'un théorème de multiplication. Cinq correspond au nombre de multiplications de l'expression à encadrer et 4^3 au nombre de cas étudiés pour les trois variables. Si un théorème générique basé sur la multiplication d'intervalles est employé pour vérifier la preuve, il faut 46 secondes à Coq pour charger la preuve. En utilisant un réseau de tri à la place des $\min$ et $\max$ pour réduire le nombre de comparaisons de 8 à 6, le temps nécessaire tombe à 37 secondes. En utilisant les neuf théorèmes spécialisés de la multiplication, le temps n'est plus que de 25 secondes. Sur cet exemple, même la version optimisée du théorème générique reste 30% plus lente que les versions spécialisées.

Sur l'exemple de la multiplication, les théorèmes concernant l'élévation au carré et la valeur absolue sont déclinés en plusieurs versions en fonction du signe des valeurs contenues dans l'intervalle d'entrée.

6.3.2 Gestion des valeurs absolues

La valeur absolue du produit est le produit des valeurs absolues. Gappa connaît un théorème qui exprime cette propriété. Comme les expressions en valeur absolue sont à valeurs positives, on peut construire un théorème simplifié de multiplication. Mais malheureusement rien ne garantit que les bornes inférieures des encadrements soient positives. C'est l'une des raisons pour lesquelles Gappa utilise un prédicat spécialisé dans les encadrements de valeurs absolues : $\mathrm{ABS}(e, [a, b]) = 0 \leq a \leq |e| \leq b$. Grâce à cette contrainte sur la borne inférieure, le corps de l'énoncé de ce théorème de multiplication devient

$$\begin{array}{c}\forall a, b, c, d, u, v \in \mathbb{D}, \quad \forall x, y \in \mathbb{R}, \quad \mathrm{ABS}(x, [a, b]) \ \wedge \ \mathrm{ABS}(y, [c, d]) \quad \Rightarrow \\ 0 \leq u \ \wedge \ u \leq a \cdot c \ \wedge \ b \cdot d \leq v \quad \Rightarrow \quad \mathrm{ABS}(x \cdot y, [u, v]).\end{array}$$

Le cas de l'addition de deux expressions dont les valeurs absolues sont encadrées est un peu plus complexe. Les bornes finales sont fournies par les inégalités triangulaires. Pour la borne supérieure, l'hypothèse est identique à celle de l'addition normale : $b + d \leq v$. Pour la borne inférieure, il faut par contre garantir $u \leq ||x| - |y|| \leq |x + y|$. Le traitement dépend de la position de l'intervalle $[a, b] - [c, d]$ par rapport à zéro. Un théorème différent est utilisé dans chacun des trois cas. Voici celui qui intervient quand $[a, b] - [c, d]$ ne contient que des valeurs strictement positives :

$$\begin{array}{c}\forall a, b, c, d, u, v \in \mathbb{D}, \quad \forall x, y \in \mathbb{R}, \quad \mathrm{ABS}(x, [a, b]) \ \wedge \ \mathrm{ABS}(y, [c, d]) \quad \Rightarrow \\ 0 \leq u \ \wedge \ u \leq a - d \ \wedge \ b + d \leq v \quad \Rightarrow \quad \mathrm{ABS}(x + y, [u, v]).\end{array}$$

6.3.3 Utilisation des multiplications

Jusqu'à présent, chacun des théorèmes de vérification n'employait que des additions et des multiplications de nombres dyadiques, c'est-à-dire des opérations exactes. Si l'on continue à suivre la logique de l'arithmétique d'intervalles, cela implique d'employer des divisions pour effectuer les calculs, divisions qui sont en général inexactes. Voici ce à quoi ressemblerait le théorème de division pour des intervalles ne contenant que des valeurs positives :

$$\begin{array}{c}\forall a, b, c, d, u, v \in \mathbb{D}, \quad \forall x, y \in \mathbb{R}, \quad x \in [a, b] \ \wedge \ y \in [c, d] \quad \Rightarrow \\ 0 \leq a \ \wedge \ 0 < c \ \wedge \ u \leq a/d \ \wedge \ b/c \leq v \quad \Rightarrow \quad x/y \in [u, v].\end{array}$$

Pour effectuer ces divisions exactement, il faudrait employer des nombres rationnels, voire des nombres rationnels munis en facteur d'une puissance de deux sur le modèle des nombres dyadiques afin que la conversion des dyadiques vers les rationnels ne soit pas trop coûteuse. Il est cependant possible de contourner ce problème en modifiant les théorèmes pour qu'ils n'utilisent que des multiplications :

$$\begin{array}{c}\forall a, b, c, d, u, v \in \mathbb{D}, \quad \forall x, y \in \mathbb{R}, \quad x \in [a, b] \ \wedge \ y \in [c, d] \quad \Rightarrow \\ 0 \leq a \ \wedge \ 0 < c \ \wedge \ u \cdot d \leq a \ \wedge \ b \leq v \cdot c \quad \Rightarrow \quad x/y \in [u, v].\end{array}$$

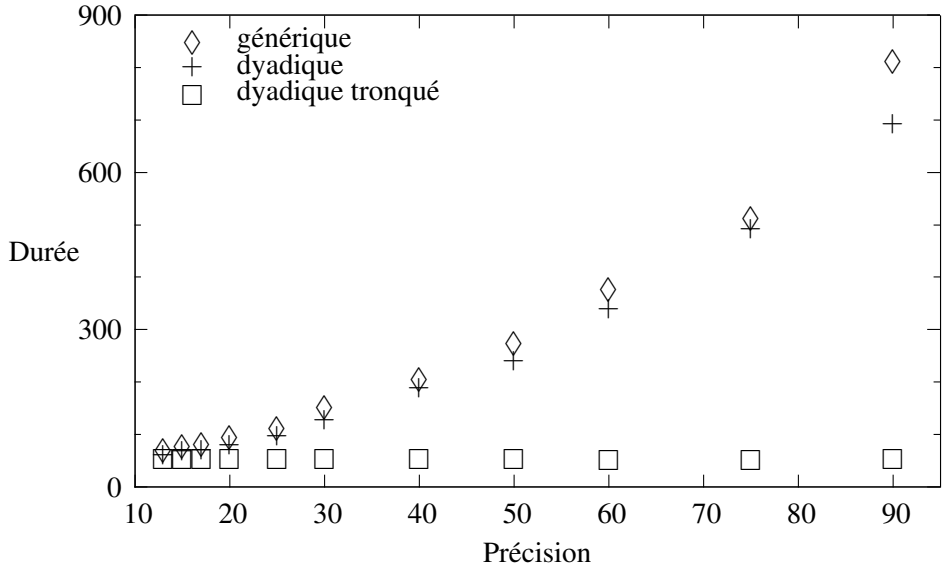

FIG. 6.1 – Impact de la précision sur le temps de vérification

Une telle méthode s'applique aussi aux racines carrées. En conclusion, tous les encadrements d'opérations arithmétiques générés par Gappa peuvent être vérifiés sans faire intervenir de division ou de racine carrée de nombres dyadiques. Seules addition et multiplication sont employées.

6.3.4 Influence de la précision

Script Gappa 6.2 Approximation de la racine carrée

```
p = (1 + 3b-2*(x-1)) / (1 + 1b-2*(x-1));
{ x in [0.5,1.5] -> sqrt(x) - p in [-1b-7,1b-8] }
sqrt(x) - p $ x;
```

Le script Gappa 6.2 vérifie que la fraction rationnelle p est proche de la fonction racine carrée sur l'intervalle $[0.5, 1.5]$. Il va servir à mesurer l'impact de la précision des calculs sur le temps nécessaire à l'assistant de preuve Coq pour compiler le fichier. Pour que Gappa génère une preuve de cette proposition en un temps raisonnable, la précision doit être d'au moins 13 bits. Cette preuve contient plus de 3300 lemmes effectuant des vérifications de calculs par intervalles. Le nombre de lemmes décroît légèrement et se stabilise à partir de 17 bits de précision.

Le graphe 6.1 indique le temps nécessaire à Coq (en secondes) pour charger une preuve en fonction de la précision (en nombre de bits) des bornes qu'a utilisée Gappa. Trois types de preuves ont été générés. Les preuves « génériques » utilisent les nombres flottants définis par la bibliothèque Pff tandis que les preuves

« dyadiques » utilisent la bibliothèque spécialisée écrite pour Gappa. L'usage de la bibliothèque spécialisée permet un gain de 12% en performance. Les preuves « tronquées » utilisent quant à elles la simplification des bornes décrite au paragraphe 2.3.3 en plus de la bibliothèque spécialisée. Cette simplification permet de passer d'une complexité en temps au moins quadratique à une complexité indépendante de la précision interne de Gappa.

Le temps de chargement de la preuve simplifiée est même inférieur à celui nécessaire pour la preuve utilisant la précision globale minimale de treize bits. Toutes les étapes de la preuve ne demandent en effet pas une précision de treize bits. Cette méthode de simplification est donc plus avantageuse que de simplement diminuer la précision globale jusqu'à ce que la preuve ne passe plus.

6.4 Théorèmes et arrondis

Les opérations arithmétiques de base ne posent pas de difficulté lors de leur formalisation. Par rapport à un développement traditionnel en Coq, il faut cependant privilégier l'efficacité des algorithmes sur la simplicité des théorèmes. Chaque optimisation implantée pour accélérer l'interprétation des certificats se paie par une complexité accrue des théorèmes dans la bibliothèque de support.

Pour les opérateurs d'arrondi, la situation se complique puisqu'il n'y a plus de modèle pour servir de référence. Dans le cas des opérateurs arithmétiques, il s'agissait de la théorie des nombres réels. Il faut ici en construire un pour les arithmétiques approchées. Mais voyons d'abord l'approche employée dans la bibliothèque Pff. Les arrondis y sont définis par des prédicats : « `Closest` $\ldots r\ f$ » signifie que f est un des nombres flottants les plus proches du réel r dans le format courant (dont les caractéristiques sont normalement décrites au niveau des pointillés).

Il n'y a pas forcément unicité du nombre dyadique f. D'une part plusieurs représentations dyadiques du nombre flottant peuvent satisfaire les contraintes du prédicat. D'autre part il peut même y avoir plusieurs flottants qui satisfont le prédicat `Closest`. Ce n'est heureusement pas le cas d'autres prédicats comme `Even-Closest` qui vérifie la propriété de nombre flottant le plus proche au sens de la norme IEEE-754. En adaptant les théorèmes d'existence de ce nombre flottant, on pourrait montrer l'existence d'une fonction d'arrondi sur les nombres réels adaptée à Gappa. Une telle fonction ne fournirait aucun moyen de calcul effectif des bornes arrondies, il faudrait donc définir un algorithme et vérifier qu'il correspond bien à la restriction de cette fonction d'arrondi aux nombres dyadiques.

Le problème de cette approche réside dans la spécialisation de Pff aux arithmétiques flottantes. Ses prédicats d'arrondi et leurs propriétés ne permettent pas de décrire une arithmétique à virgule fixe. Partant de cette constatation, j'ai bâti un formalisme directement inspiré de l'approche adoptée dans Gappa et qui soit ainsi adapté aussi bien à la virgule fixe qu'à la virgule flottante. Le paragraphe 4.2 décrit cette approche.

6.4.1 Fonction d'exposant

Du point de vue de Gappa, tout nombre machine $x \in \mathbb{F}$ est un multiple de 2^ζ, la seule différence entre virgule fixe et virgule flottante étant le choix de ζ. En arithmétique à virgule fixe, ζ est constant sur l'ensemble des nombres dyadiques. En virgule flottante, sa valeur dépend de la binade b où réside le nombre : $2^{b-1} \leq |x| < 2^b$. Pour couvrir les deux scénarios, il suffit de se donner une fonction ζ : $b \mapsto \zeta(b)$. À partir de là, il n'y a plus aucune différence.

La fonction ζ décrit un ensemble de nombres machine $\mathbb{F}_\zeta$. Le réel zéro fait partie de cet ensemble. Un réel non nul en fait partie s'il est représentable par un nombre dyadique $m \cdot 2^e$ vérifiant $\zeta(||m|| + e) \leq e$. L'entier $||m||$ désigne ici la longueur de la mantisse en nombre de bits ; la somme $||m|| + e$ est donc la binade b dans laquelle se trouve le réel représenté. Pour un arrondi `fixed<`n`,_>`, la fonction d'exposant est $\zeta(b) = n$. Pour un arrondi `float<`p`,`n`,_>`, la fonction est $\zeta(b) = \max(b - p, n)$.

Afin de garantir la monotonie de la fonction d'arrondi, le formalisme impose deux contraintes sur la fonction d'exposant. La première traite des puissances de deux et s'énonce : $\forall b, \quad \zeta(b) < b \Rightarrow \zeta(b + 1) \leq b$. Elle peut se traduire ainsi : si 2^{b-1} fait partie de $\mathbb{F}_\zeta$, alors 2^b en fait partie aussi et cette puissance est donc arrondie vers elle-même. La deuxième contrainte concerne les réels compris entre zéro et le plus petit nombre non nul de $\mathbb{F}_\zeta$ en valeur absolue. La fonction ζ doit être constante sur toutes les binades b concernées et $2^{\zeta(b)}$ doit être représentable. Cette contrainte s'énonce :

$$\forall b \in \mathbb{Z}, \quad b \leq \zeta(b) \Rightarrow \begin{cases} \zeta(\zeta(b) + 1) \leq \zeta(b) \\ \forall l \in \mathbb{Z}, \quad l \leq \zeta(b) \Rightarrow \zeta(l) = \zeta(b) \end{cases}$$

Les contraintes sur ζ sont suffisamment lâches pour que d'autres ensembles que virgule flottante au sens IEEE-754 et virgule fixe soient représentables. Par exemple, si un processeur implante des nombres flottants sans les dénormalisés, le formalisme reste adapté. En notant p la précision et 2^n le plus petit nombre normalisé, la fonction est $\zeta(b) = n$ pour $b \leq n$ et $\zeta(b) = b - p$ sinon.

6.4.2 Arrondi des nombres dyadiques

Le script 6.4 décrit les fonctions utilisées pour réaliser l'arrondi d'un nombre dyadique. La fonction `round` extrait la direction d'arrondi de l'enregistrement `rdirs` en fonction du signe du nombre dyadique à arrondir. Elle invoque ensuite `round_pos` pour arrondir la valeur absolue du nombre dyadique. Cette fonction calcule d'abord l'exposant `e'` fixé par l'argument `rexp` (la fonction ζ) que doit prendre l'arrondi du nombre donné par la valeur absolue `m` de la mantisse et par l'exposant `e`. Si `e'` est plus petit ou égal à `e`, le nombre est déjà représentable et est directement renvoyé. Dans le cas contraire, la fonction `shr` effectue la troncature de l'entier positif `m` à la taille qu'impose `e'`. Au fur et à mesure de la troncature, les bits r et s sont tenus à jour. La fonction `rdir` de direction d'arrondi les utilisera pour choisir le résultat renvoyé parmi le nombre tronqué et son successeur.

Script Coq 6.4 Algorithme d'arrondi d'un nombre dyadique

```
Definition shr_aux (p : rnd_record) : rnd_record :=
 let s := rnd_r p || rnd_s p in
 match (rnd_m p) with
 | N0 => rnd_record_mk N0 false s
 | Npos m1 =>
  match m1 with
  | xH => rnd_record_mk N0 true s
  | xO m2 => rnd_record_mk (Npos m2) false s
  | xI m2 => rnd_record_mk (Npos m2) true s
  end
 end.

Definition shr (m : positive) (d : positive) :=
 iter_pos d _ shr_aux (rnd_record_mk (Npos m) false false).

Definition round_pos (rdir : rnd_record -> bool)
   (rexp : Z -> Z) (m : positive) (e : Z) :=
 let e' := rexp (e + Zpos (digits m))%Z in
 match (e' - e)%Z with
 | Zpos d =>
    let r := shr m d in
    ((if rdir r then Nsucc (rnd_m r) else rnd_m r), e')
 | _ => (Npos m, e)
 end.

Definition round (rdirs : round_dir) (rexp : Z -> Z)
   (f : float2) :=
 match (Fnum f) with
 | Z0 => Float2 Z0 Z0
 | Zpos p =>
    match (round_pos (rpos rdirs) rexp p (Fexp f)) with
    | (N0, _) => Float2 Z0 Z0
    | (Npos q, e) => Float2 (Zpos q) e
    end
 | Zneg p =>
    match (round_pos (rneg rdirs) rexp p (Fexp f)) with
    | (N0, _) => Float2 Z0 Z0
    | (Npos q, e) => Float2 (Zneg q) e
    end
 end.
```

La fonction `round` est ensuite étendue à l'ensemble des nombres réels. La première étape consiste à prouver que `round_pos` est constante par morceaux. Cette constance est ensuite combinée à la densité des dyadiques dans les réels pour prouver que tout nombre réel est encadré par deux nombres dyadiques qui s'arrondissent vers la même valeur. C'est cette valeur qui est choisie pour l'extension. La constance sert aussi à prouver que la fonction `round_pos` est croissante. Par construction, l'extension l'est aussi. Qui plus est, tous les nombres renvoyés par l'extension sont représentables dans $\mathbb{F}_\zeta$.

6.4.3 Exemple de théorème

Connaissant un encadrement $[a, b]$ d'une expression e et un entier n vérifiant $\exists m \in \mathbb{Z},\ e = m \cdot 2^n$, Gappa est capable de prouver que les valeurs de e sont en fait contenues dans un intervalle plus petit : $[\triangle(a), \triangledown(b)]$. Ici $\triangledown$ et $\triangle$ désignent les opérateurs d'arrondi vers le bas et vers le haut associés au format à virgule fixe dont le poids minimal est 2^n. L'intérêt de ce théorème est décrit au paragraphe 4.3.1.

La preuve de ce théorème présent dans la bibliothèque de support de Gappa utilise la propriété de monotonie des extensions de fonctions d'arrondi. Notons $\triangledown'$ et $\triangle'$ les extensions de $\triangledown$ et $\triangle$. Par hypothèse, les inégalités $a \leq e$ et $e \leq b$ sont vérifiées. On en déduit donc les inégalités $\triangle'(a) \leq \triangle'(e)$ et $\triangledown'(e) \leq \triangledown'(b)$. Par définition de n, les valeurs de e sont représentables dans le format des opérateurs $\triangledown$ et $\triangle$, on a donc $e = \triangledown'(e) = \triangle'(e)$. Cela permet de conclure que $\triangle(a) \leq e$ et $e \leq \triangledown(b)$ en utilisant le fait que a et b sont dyadiques.

Chapitre 7

Application de Gappa aux fonctions élémentaires

Ce chapitre présente quelques exemples un peu plus réalistes et les techniques à mettre en œuvre pour pouvoir les certifier à l'aide de Gappa.

La bibliothèque CRlibm[1] propose des fonctions élémentaires correctement arrondies en double précision. Les directions d'arrondi proposées sont les quatre de la norme IEEE-754. Ainsi, pour x un nombre flottant double précision, l'expression `exp_rn`(x) renvoie le nombre flottant double précision le plus proche de la valeur mathématique $\exp x$, c'est-à-dire $\circ(\exp x)$.

Réduction de l'argument x à un petit intervalle, évaluation d'une approximation polynomiale, reconstruction du résultat permettent de calculer un nombre y en précision élevée qui soit proche de $\exp x$. C'est l'arrondi $\circ(y)$ de ce nombre qui est renvoyé par `exp_rn`(x). La fonction est donc correcte si l'égalité $\circ(y) = \circ(\exp x)$ est vérifiée. Or il existe une borne δ garantissant que y et $\exp x$ s'arrondissent au même flottant quand l'inégalité $|y - \exp x| \leq \delta$ est vérifiée. Pour certifier que la fonction `exp_rn` satisfait ses spécifications, il suffit donc de prouver que la valeur calculée y n'est pas entachée d'une erreur totale plus grande que δ. C'est précisément pour prouver ce genre de proposition que Gappa a été conçu.

L'évolution des fonctionnalités de Gappa a en fait été fortement influencée par les besoins en preuve des développeurs de CRlibm et en particulier de Florent de Dinechin et de Christoph Lauter. Les bornes sur les erreurs étaient auparavant calculées à la main [DdDM01] et l'emploi de Gappa a permis d'automatiser et de simplifier ce travail [dDLM06]. Qui plus est, l'outil a amélioré certaines bornes qui souffraient de sur-estimation, permettant ainsi d'accélérer les fonctions de CRlibm. En effet, chacune de ces fonctions se décompose en deux phases : une phase rapide qui donne une première approximation puis une phase lente appelée si cette

[1] `http://lipforge.ens-lyon.fr/www/crlibm/`

première approximation n'est pas suffisamment bonne. Cette approche est similaire à celle décrite pour les prédicats du chapitre 8. Grâce aux bornes calculées par Gappa, il a été possible de prouver que les phases rapides étaient plus précises que prévu et qu'il était donc moins souvent nécessaire de se rabattre sur les phases lentes.

7.1 L'exponentielle de Tang

Même si l'on ne s'intéresse pas à l'arrondi correct mais seulement à des approximations garanties, l'approche reste similaire : il s'agit de vérifier que la valeur calculée par le logiciel ou le matériel ne s'écarte pas de la valeur mathématique de la fonction plus que la borne spécifiée. Cette vérification peut se faire de manière formelle. Une implantation de l'exponentielle en simple précision [Tan90] est ainsi entièrement certifiée par une preuve formelle [Har97], ce qui a d'ailleurs permis de constater que la preuve papier originelle était incorrecte.

Il faut remarquer que la preuve formelle écrite par Harrison va bien plus loin que ce que Gappa sait traiter. Elle formalise en effet un langage de programmation impératif suffisamment puissant pour pouvoir exprimer tous les détails de l'implantation de l'exponentielle. De plus, le script HOL light [Har96] contient des preuves de l'erreur de troncature, de la réduction d'argument et de la reconstruction du résultat final. Enfin, il traite le cas des valeurs flottantes exceptionnelles, les infinis par exemple.

Gappa n'est utilisé ici que pour borner l'erreur totale commise sur le résultat non reconstruit. Le script fait des hypothèses sur la précision de la réduction d'argument et sur celle de l'approximation polynomiale. La preuve ainsi écrite est donc moins exhaustive mais l'obtenir n'est plus l'affaire que de quelques minutes. Elle n'en couvre pas moins les parties délicates du problème, c'est-à-dire la propagation des erreurs au cours des calculs approchés, et ne nécessite pas d'un développeur qu'il se spécialise dans les méthodes formelles. Voici une brève description de l'algorithme de Tang.

7.1.1 Présentation de l'algorithme

Pour approcher $\exp x$ avec x un nombre flottant simple précision, la première étape est la réduction d'argument pour se ramener dans un intervalle centré en zéro et de rayon $\frac{\ln 2}{64}$:

$$x = n \cdot \frac{\ln 2}{32} + R_0 \quad \text{avec } n \text{ entier.}$$

Cette réduction d'argument ne peut généralement pas se faire de façon exacte. La valeur R_0 est donc approchée par la somme R de deux nombres flottants simple précision r_1 et r_2 [LBD03]. L'algorithme calcule ensuite la division euclidienne de n par 32 : $n = 32 \cdot m + j$. On a alors

$$\exp x = 2^m \cdot 2^{\frac{j}{32}} \cdot \exp R_0.$$

Le nombre $S_0 = 2^{\frac{j}{32}}$ est approché par un nombre S somme de deux flottants simple précision s_1 et s_2. Dans l'exemple ci-dessous, seul le cas $j = 1$ est présenté ; les autres cas utilisent une approche similaire. Il reste à approcher la valeur $\exp R_0$ en évaluant $1 + P(r)$. Le polynôme $P(r)$ vaut $r + a_1 \cdot r^2 + a_2 \cdot r^3$ avec a_1 et a_2 deux nombres flottants simple précision. Une borne sur l'erreur Z est fournie en hypothèse à Gappa : $P(r) + Z = \exp r - 1$.

7.1.2 Formalisation Gappa

Le script 7.1 expose le problème dans la syntaxe Gappa. Les lignes 1 à 14 décrivent l'implantation de l'algorithme et les constantes flottantes qu'il utilise. La ligne 14 indique par exemple que l'expression e est l'expression $s_1 + (s_2 + s \cdot p)$ dans laquelle les trois opérations arithmétiques ont été arrondies par `rnd`, c'est-à-dire que les calculs sont effectués en simple précision et en arrondi au plus près.

Script Gappa 7.1 Erreur commise lors de l'évaluation de l'exponentielle

```
1  @rnd = float< ieee_32, ne >;
2
3  a1 = 8388676b-24;
4  a2 = 11184876b-26;
5  l2 = 12566158b-48;
6  s1 = 8572288b-23;
7  s2 = 13833605b-44;
8
9  r2 rnd= -n * l2;
10 r rnd= r1 + r2;
11 q rnd= r * r * (a1 + r * a2);
12 p rnd= r1 + (r2 + q);
13 s rnd= s1 + s2;
14 e rnd= s1 + (s2 + s * p);
15
16 R = r1 + r2;
17 S = s1 + s2;
18
19 E = s1 + (s2 + S0 * (r1 + (r2 + R0 * R0 * (a1 + R0 * a2))));
20 Er = S + S0 * (R + (a1 * R0 * R0 + a2 * R0 * R0 * R0) + 0);
21 E0 = S0 + S0 * (R0 + (a1 * R0 * R0 + a2 * R0 * R0 * R0) + Z);
22
23 { Z in [-55b-39,55b-39] /\ S - S0 in [-1b-41,1b-41] /\
24   R - R0 in [-1b-34,1b-34] /\ R in [0,0.0217] /\
25   n in [-10176,10176] ->
26   e in ? /\ e - E0 in ? }
27
28 e - E0 -> (e - E) + (Er - E0);
29 r1 -> R - r2;
```

Viennent ensuite les définitions de R et S comme des sommes de flottants. Il faut noter que les valeurs r et s calculées par l'algorithme sont des approximations de R et S qui sont elles-mêmes des approximations de R_0 et S_0. Quant à E, il s'agit de l'expression qui serait évaluée si le processeur effectuait ses calculs en précision infinie. Elle a exactement la même structure que l'expression e effectivement calculée, si ce n'est que S_0 et R_0 y remplacent leurs valeurs approchées s et r. Finalement, E_0 est $S_0 \cdot (1 + P(R_0) + Z)$, c'est-à-dire une façon correcte d'écrire $\exp x \cdot 2^{-m}$.

Les lignes 23 à 25 sont les hypothèses de la proposition. Elles expriment les bornes sur la qualité de la réduction d'argument ($R - R_0$), sur celle de l'approximation polynomiale (Z) et sur celle de la représentation de la constante $2^{\frac{j}{32}}$ pour $j = 1$ ($S - S_0$). Elles expriment aussi les propriétés de la réduction d'argument, c'est-à-dire les domaines de n et R.

7.1.3 Ajout d'indices

Mettons pour l'instant de côté la définition de E_r et les indices des lignes 28 et 29. Le script correspond alors à une transcription directe du problème. Cependant, si on le soumet à Gappa, celui-ci n'est capable de borner ni e ni $e - E_0$. Intéressons nous d'abord à e. Vues les hypothèses et les constantes, l'outil connaît directement toutes les feuilles de l'arbre syntaxique de e, mis à part r_1. L'indice de la ligne 29 fournit une façon d'encadrer la valeur de r_1 et permet effectivement à Gappa d'encadrer e.

Décomposons maintenant $e - E_0$ en $(e - E) + (E - E_0)$. L'expression $e - E$ caractérise la propagation des erreurs au cours du calcul et Gappa n'a aucune difficulté à l'encadrer de façon précise. Le problème ne vient donc pas d'ici. Si l'on essaie maintenant d'encadrer l'autre morceau $E - E_0$ de l'erreur, l'outil échoue. La difficulté pour Gappa réside dans l'absence de ressemblance entre les expressions E et E_0. Ainsi, la dernière opération de E (par ordre d'évaluation) est une addition alors que celle de E_0 est une multiplication, ce qui empêche Gappa d'appliquer toute méthode autre que l'évaluation naïve.

L'objectif est donc de normaliser les deux expressions. C'est ici qu'intervient l'expression E_r. Elle est mathématiquement égale à E mais elle présente une structure similaire à celle de E_0. Pour que le terme Z de E_0 ait son pendant dans E_r, un zéro est artificiellement ajouté à l'expression E_r. L'outil évalue alors la différence $E_r - E_0$ en faisant correspondre les termes S et S_0, R et R_0, et 0 et Z et en appliquant les méthodes du paragraphe 3.2.1.

L'indice de réécriture à la ligne 28 indique à Gappa l'existence de la décomposition de $e - E_0$ en $(e - E) + (E_r - E_0)$. Elle correspond à la façon dont le développeur a négligé des termes et réordonné les calculs dans son algorithme afin d'améliorer la précision du résultat. Ces optimisations ont nécessité une grande part de réflexion et d'expertise de la part du développeur ; il n'est donc pas surprenant que quelques indications soient nécessaires à Gappa.

7.2 Arithmétique double-double

Cette implantation de l'exponentielle contient des traces d'arithmétique multi-précision. La constante S_0 est ainsi approchée par la somme de deux nombres flottants s_1 et s_2 qui ne se chevauchent pas : les bits les plus significatifs de S_0 se trouvent tous représentés par s_1. Cette approche multi-précision reste limitée puisque le terme d'erreur s_2 n'est pas généré par l'algorithme : il est calculé à l'avance par le développeur.

La bibliothèque CRlibm utilise la multi-précision de façon plus systématique. Elle permet d'obtenir des résultats d'une précision plus élevée que celle employée dans les calculs intermédiaires. Il est en effet facile de calculer les termes d'erreurs des opérations flottantes d'addition et de multiplication pour peu qu'il n'y ait pas de dépassement de capacité [Dek71]. Le logarithme est utilisé ci-après comme exemple de certification d'un algorithme employant des opérations arithmétiques multi-précision.

7.2.1 Opérations exactes

Plutôt que de s'attaquer directement au code complet du logarithme, il est préférable de commencer par le code qui calcule le terme d'erreur d'une addition flottante [Knu69]. Il s'agit ici de montrer que la somme $x + y$ des entrées flottantes est égale à la somme $s + e$ des valeurs calculées. Le script 7.2 présente la transcription de ce problème.

Script Gappa 7.2 Propriété fondamentale de l'opérateur d'addition exacte

```
@rnd = float< ieee_64, ne >;
x = rnd(x_);
y = rnd(y_);
s rnd= x + y;
t rnd= s - x;
e rnd= (x - (s - t)) + (y - t);
{ (s + e) - (x + y) in [0,0] }
```

Dans le formalisme de Gappa, la preuve de cette proposition nécessite une étude par cas sur les positions relatives de x et y et les binades qu'ils occupent. Le nombre de sous-cas que cela demande met cette preuve hors de portée de l'outil.

Pour traiter des programmes qui s'appuient sur de telles portions de code, une première solution serait d'apprendre à Gappa à les repérer. Il n'existe malheureusement pas une unique façon de les agencer ; par exemple, `x - (s - t)` peut aussi s'écrire `(t - s) + x`. Qui plus est, il existe des versions de l'algorithme utilisant moins d'opérations flottantes quand les positions relatives de x et y sont connues puisque $\circ(s-t)$ peut alors valoir exactement x. Une deuxième solution serait que Gappa fournisse des opérateurs calculant les termes d'erreur. La technique

consistant à décomposer un opérateur arithmétique approché en un opérateur arithmétique sur les réels suivi d'une fonction de projection sur l'ensemble des nombres machine ne s'applique cependant pas dans ce cas.

Aucune de ces deux solutions n'a semblé satisfaisante et le script 7.2 ne peut donc pas être traité par Gappa. *A fortiori*, Gappa ne sait pas traiter un script traduisant fidèlement du code s'appuyant sur un additionneur exact. Il est néanmoins possible d'étudier un tel code en simplifiant la définition de e. Il suffit de le définir directement comme étant un terme d'erreur : $e = (x + y) - \circ(x + y)$. Connaissant un encadrement de $x + y$, Gappa est alors capable d'en déduire un encadrement de e. Cependant, au vu des conventions de représentation des erreurs en Gappa, il est préférable d'écrire $e = -(\circ(x + y) - (x + y))$ pour qu'un plus grand nombre de théorèmes reconnaissent le terme d'erreur et puissent s'appliquer.

Cette approche est bien adaptée à la façon dont Gappa manipule des expressions, mais certaines contraintes que doit vérifier l'algorithme initial sont perdues lors de la transcription. C'est le cas lors du calcul du terme d'erreur e de la multiplication $\circ(x \cdot y)$. En Gappa, cela se traduit par la définition $e = -(\circ(x \cdot y) - (x \cdot y))$. Le terme d'erreur n'est cependant pas représentable par un nombre flottant s'il est trop petit. Prouver la correction de l'algorithme nécessite donc de prouver que $x \cdot y$ ne peut pas être petit. Cette contrainte n'apparaît pas dans la définition de e et elle doit donc être ajoutée par l'utilisateur aux conclusions de la proposition logique. La situation est donc similaire à la gestion des dépassements de capacité ; eux aussi doivent faire l'objet de contraintes explicites.

Pour illustrer cette représentation des termes d'erreur en Gappa, considérons le code C 7.1 tiré de CRlibm. Il s'agit de l'évaluation polynomiale $p(x) = x - \frac{1}{2} \cdot x^2 + x^3 \cdot q(x)$ effectuée après la réduction d'argument lors de l'approximation du logarithme [dDLM06]. La réduction a produit un double-double $Z = z_h + z_l$ et le code cherche à calculer $p(Z)$. Pour ce faire, le terme $x^3 \cdot q(x)$ de p est approché en arithmétique double précision au point z_h. Le terme $-\frac{1}{2} \cdot x^2$ est quant à lui approché en négligeant le terme z_l^2.

Code C 7.1 Évaluation polynomiale du logarithme

```
q = c3 + zh * (c4 + zh * (c5 + zh * (c6 + zh * c7)));
Mul12(&zhSquareh, &zhSquarel, zh, zh);
zhCube = zh * zhSquareh;
polyUpper = zhCube * q;
zhSquareHalfh = zhSquareh * -0.5;
zhSquareHalfl = zhSquarel * -0.5;
zhzl = -1 * (zh * zl);
Add12(t1h, t1l, polyUpper, zhzl);
Add22(&t2h, &t2l, zh, zl, zhSquareHalfh, zhSquareHalfl);
Add22(&ph, &pl, t2h, t2l, t1h, t1l);
```

Le script 7.3 retranscrit en Gappa les huit premières lignes du code C. Là où

ce dernier effectue un `Mul12`[2] pour calculer exactement le carré z_h^2 et le stocker dans les variables `zhSquareh` et `zhSquarel`, le script Gappa lui donne directement le nom `ZhSquarehl` puis manipule les deux variables en bloc, par exemple lors du calcul de `ZhSquareHalfhl`. De même, le résultat `T1hl` de l'opérateur `Add12` d'addition exacte est directement considéré comme étant le réel somme de `polyUpper` et `zhzl`. Les opérations exactes sont ainsi assimilées à des opérations sur les nombres réels. La représentation des résultats par des paires de flottants double précision n'apparaît que pour la variable Z.

Script Gappa 7.3 Approximation du logarithme (partie 1)

```
zh = float64ne(Z);
zl = Z - zh;
q float64ne= c3 + zh * (c4 + zh * (c5 + zh * (c6 + zh*c7)));
ZhSquarehl = zh * zh;
zhSquareh = float64ne(ZhSquarehl);
zhCube float64ne= zh * zhSquareh;
polyUpper float64ne= zhCube * q ;
ZhSquareHalfhl = -0.5 * ZhSquarehl;
zhzl = -1 * float64ne(zh * zl);
T1hl = polyUpper + zhzl;
```

7.2.2 Opérations en précision élevée

Pour l'instant, seules les opérations exactes ont été considérées. Il est cependant possible de définir des opérateurs plus complexes qui prendraient des paires de flottants en entrée et renverraient une paire de flottants en sortie. Par exemple, un opérateur de multiplication prendrait les paires (x_1, x_2) et (y_1, y_2) et renverrait une paire (z_1, z_2) telle que la somme $z_1 + z_2$ soit une valeur approchée du produit $p = (x_1 + x_2) \cdot (y_1 + y_2)$. En fait, il n'est même pas nécessaire de se restreindre à des opérations sur des paires de flottants : CRlibm utilise des triplets de flottants dans la phase lente de ses algorithmes [Lau05]. La définition de ces opérations n'est cependant pas aussi simple que pour les précédentes. Le résultat ne vérifie en effet ni $z_1 = \circ(p)$ ni $z_2 = p - z_1$ ni même $z_2 = \circ(p - z_1)$.

Les deux dernières lignes de l'exemple du logarithme montrent l'emploi de ce genre d'opération. Ce sont deux additionneurs qui prennent chacun deux doubles-double en entrée et renvoient un double-double approchant leur somme en sortie. Le script 7.4 traduit ces additionneurs à l'aide d'opérateurs d'arrondi généralisés. Le concept de ces opérateurs généralisés est présenté au paragraphe 4.1.2.

[2] Le nom des macros employées par CRlibm reflète l'opération évaluée et la taille des expansions flottantes [Pri91] employées comme opérandes et résultat. Ainsi `Mul12` effectue la multiplication de deux flottants et stocke le résultat dans la somme de deux flottants. La macro `Add22` effectue quant à elle l'addition de deux sommes de deux flottants et stocke le résultat dans une somme de deux flottants.

Script Gappa 7.4 Approximation du logarithme (partie 2)

```
T2hl = add_rel<103>(Z, ZhSquareHalfhl);
Phl = add_rel<103>(T2hl, T1hl);
```

La fonction `add_rel<103>` est associée à l'addition et Gappa sait donc que `Phl` approche la somme des doubles-double `T2hl` et `T1hl`. Plus précisément, l'entier passé en paramètre indique que cette approximation vérifie

$$\left|\frac{\texttt{Phl} - (\texttt{T2hl} + \texttt{T1hl})}{\texttt{T2hl} + \texttt{T1hl}}\right| \leq 2^{-103}.$$

La définition de `Phl` n'indique donc pas comment il a été calculé mais fournit directement une propriété d'approximation le caractérisant. Cette approche permet de représenter efficacement les résultats de briques arithmétiques sur des expansions flottantes. Elle souffre malheureusement d'un défaut dans son formalisme : l'écriture `add_rel<103>`(x, y) nécessite que le résultat soit défini de façon unique en fonction de x et y. Ce n'est pas le cas ici puisque le résultat dépend en fait de la représentation sous forme de paires de flottants des nombres x et y et cette représentation n'est pas toujours unique.

Ainsi, en fonction du contexte, `add_rel<103>`(x, y) pourrait prendre des valeurs différentes. Dans les scripts Gappa, la question ne se pose normalement pas : les expressions passées en argument des différents `add_rel` étant syntaxiquement différentes, l'outil n'effectue aucune simplification malheureuse. Pour contourner ce problème de multiplicité, il faudrait cependant ajouter un autre paramètre en plus de `103`. Il servirait à encoder précisément la façon dont les expressions x et y ont été obtenues. Cela permettrait d'assurer que le résultat est bien défini de façon unique.

Cette syntaxe date d'une époque où le fonctionnement de Gappa tournait principalement autour des règles de réécriture impliquant des opérateurs d'arrondi. Pour profiter de la puissance de Gappa, il fallait donc que les `Add22` et `Mul22` apparaissent comme des expressions arrondies dans les scripts. Depuis que Gappa gère les règles de réécriture à l'aide de couples d'expressions approchée et exacte, comme décrit au paragraphe 5.4.2, l'utilisation d'une syntaxe spécialisée n'est plus nécessaire. Les définitions du script 7.4 peuvent donc être remplacées par des hypothèses supplémentaires, moins lisibles certes, mais plus satisfaisantes du point de vue formel :

```
|(T2hl - (Z + ZhSquareHalfhl)) /
                  (Z + ZhSquareHalfhl)| <= 1b-103 /\
|(Phl - (T2hl + T1hl)) / (T2hl + T1hl)| <= 1b-103
```

7.3 Gestion des erreurs

Le script Gappa 7.5 est inspiré d'une portion de la preuve du logarithme. Les opérateurs d'arrondi ont cependant été supprimés du problème afin de simplifier son énoncé. Le script demande à Gappa de borner l'erreur relative qui existe entre la valeur approchée $z_h^2 + 2 \cdot z_h \cdot z_l$ et le carré exact $Z^2 = (z_h + z_l)^2$. Une erreur de cette forme apparaît quand on néglige des termes dans un algorithme, ici z_l^2. L'outil répond que cette erreur est bornée par 2^{384}, ce qui est un résultat correct mais inutile.

Script Gappa 7.5 Erreur relative du carré simplifié

```
zh = float<ieee_64,ne>(Z);
zl = Z - zh;
z2 = zh*zh + 2*zh*zl;

epsilon0 = (zh - Z) / Z;
epsilon1 = (z2 - Z*Z) / (Z*Z);

{ Z in [1b-200,1b-8] -> epsilon1 in ? }
```

L'outil est confronté à une difficulté : il ne peut pas se ramener à des encadrements d'expressions qu'il connaît, c'est-à-dire à celui de ϵ_0 principalement. Il est possible de guider Gappa à l'aide de nombreux indices, mais le plus simple reste d'effectuer un peu de calcul symbolique pour directement fournir à l'outil la relation qui lie ϵ_0 et ϵ_1.

$$\begin{aligned} \epsilon_1 &= \frac{z_h^2 + 2 \cdot z_h \cdot z_l - Z^2}{Z^2} \\ &= -\frac{(z_h - Z)^2}{Z^2} \quad \text{car } z_l = Z - z_h \\ &= -\epsilon_0^2 \end{aligned}$$

L'ajout de la règle « `epsilon1 -> -(epsilon0*epsilon0);` » permet à Gappa de donner une réponse précise : $\epsilon_1 \in [-2^{-106}, 0]$. Il faut noter que Gappa utilise la même priorité des opérateurs arithmétiques que le C afin de faciliter la transcription de programmes. En particulier, si le membre de droite était écrit « `- epsilon0 * epsilon0` », son membre de droite serait interprété $(-\epsilon_0) \cdot \epsilon_0$ au lieu de $-\epsilon_0^2$. La réponse de Gappa aurait alors été un peu moins précise puisque l'élévation au carré aurait été remplacée par une multiplication de termes corrélés lors de l'évaluation des encadrements.

Chapitre 8

Prédicats géométriques homogènes

Ce chapitre détaille des filtres semi-statiques dont la robustesse est impossible à prouver à partir des théorèmes présents dans Gappa. L'ajout d'opérateurs d'arrondi spécifiques permet néanmoins d'étendre Gappa pour qu'il calcule automatiquement les constantes utilisées dans ces filtres.

Ce travail a été réalisé en collaboration avec Sylvain Pion. Il a été présenté à [MP05] puis publié dans [MP06]. Il offre une méthode pour construire des filtres robustes en virgule flottante pour des prédicats. Ces filtres ont été implantés dans CGAL[1], une bibliothèque pour la géométrie algorithmique, par Sylvain Pion. Les théorèmes employés pour garantir leur robustesse ne correspondent pas à ceux implantés dans Gappa, rendant l'outil *a priori* inutile pour ce travail. La définition de nouveaux opérateurs d'arrondi permet cependant de faire correspondre les intervalles qui apparaissent dans les théorèmes de Gappa et ceux qui apparaissent dans notre formalisme des prédicats homogènes. Les preuves alors générées par Gappa, même si elles ne correspondent pas aux propositions demandées pour une certification formelle, fournissent les constantes numériques nous permettant de garantir la robustesse des filtres.

8.1 Géométrie algorithmique et calcul exact

La géométrie algorithmique consiste généralement à produire des structures combinatoires discrètes à partir de données numériques. Un algorithme d'enveloppe convexe en deux dimensions produit ainsi une liste de certains des points en entrée, tandis qu'un algorithme de triangulation produit un graphe de ces points. Ces algorithmes s'appuient sur des fonctions appelées prédicats géométriques et chargées de calculer la position relative de quelques objets géométriques.

[1] `http://www.cgal.org/`

Par exemple, le prédicat d'orientation prend en entrée les coordonnées de trois points du plan et indique s'ils sont alignés ou s'ils sont orientés dans le sens des aiguilles d'une montre ou dans le sens contraire. Une implantation naïve en arithmétique flottante est rapide mais peut produire des résultats faux. Il est possible de concevoir des algorithmes géométriques qui supportent des réponses incohérentes de la part des prédicats. L'approche courante est cependant l'emploi de prédicats exacts, même si c'est au prix de calculs plus longs [YD95].

8.1.1 Orientation de trois points du plan

Considérons l'exemple du prédicat d'orientation de trois points p, q et r du plan. Les entrées sont les coordonnées cartésiennes de ces trois points. L'orientation est alors donnée par le signe de ces déterminants :

$$\mathrm{orient}_2(p,q,r) = \mathrm{sgn}\begin{vmatrix} p_x & q_x & r_x \\ p_y & q_y & r_y \\ 1 & 1 & 1 \end{vmatrix} = \mathrm{sgn}\begin{vmatrix} q_x - p_x & r_x - p_x \\ q_y - p_y & r_y - p_y \end{vmatrix}$$

Le déterminant 2×2 est un peu plus rapide à évaluer, c'est lui que nous considérerons. Les coordonnées des points sont des nombres flottants en double précision. Une implantation naïve en C est donnée par le code 8.1.

Code C 8.1 Implantation naïve du prédicat d'orientation

```
double pqx = qx - px, pqy = qy - py;
double prx = rx - px, pry = ry - py;
double det = pqx * pry - pqy * prx;
if (det > 0) return POSITIVE;
if (det < 0) return NEGATIVE;
return ZERO;
```

Si un type arithmétique exact était employé à la place de `double`, cette implantation conduirait à un résultat correct quelles que soient les entrées. Elle serait cependant bien plus lente qu'une implantation flottante. En effet, le stockage de la valeur exacte du déterminant peut nécessiter plusieurs milliers de bits. L'utilisation de nombres flottants dans cette fonction contourne ce problème de performance mais risque par contre de causer divers dépassements de capacité et des erreurs d'arrondi.

Domaine limité

De nombreux problèmes risquent de découler de l'utilisation d'une telle fonction. Considérons d'abord les problèmes de domaine. Soit M la plus grande puissance de 2 qui soit représentable en double précision. Prenons les trois points sui-

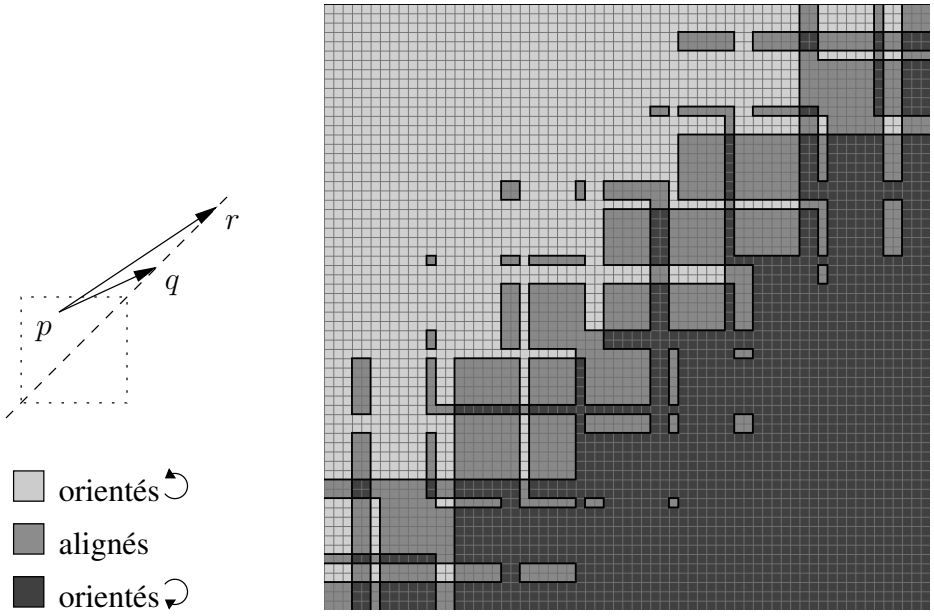

FIG. 8.1 – Valeur approchée d'$\mathrm{orient}_2(p,q,r)$ pour p proche de la droite (q,r)

vants :

$$(p,q,r) = \begin{pmatrix} -M & M & -7M/8 \\ 1 & 3 & 17/16 \end{pmatrix}$$

Le calcul de `qx - px` produit une valeur non représentable $2 \cdot M$ et la variable `pqx` contient donc $+\infty$. Les autres variables `pqy`, `prx` et `pry` contiennent par contre des valeurs finies. Il en va de même pour le résultat de `pqy * prx`. Par conséquent, l'infini contenu dans `pqx` se propage jusque dans la dernière variable `det` et la fonction répond `POSITIVE` alors que la véritable valeur du déterminant est le nombre négatif $-M/8$.

Précision limitée

L'autre inconvénient à utiliser des nombres flottants est la précision limitée de ces nombres. Si les points sont relativement proches de l'alignement, les erreurs commises en arrondissant chacun des résultats intermédiaires peuvent faire basculer le signe calculé. La figure 8.1 (inspirée des figures publiées dans [KMP+04]) montre le résultat du prédicat quand il est évalué en simple précision pour $q = (8.1, 8.1)$ et $r = (12.1, 12.1)$ fixés et p se déplaçant sur la grille des 65×65 paires de flottants entourant le point $(1.5, 1.5)$.

Idéalement, le prédicat devrait répondre `ZERO` sur la première diagonale de la figure représentant les points alignés avec q et r. Les deux triangles séparés par cette diagonale devraient quant à eux avoir des signes constants et différents. Ce

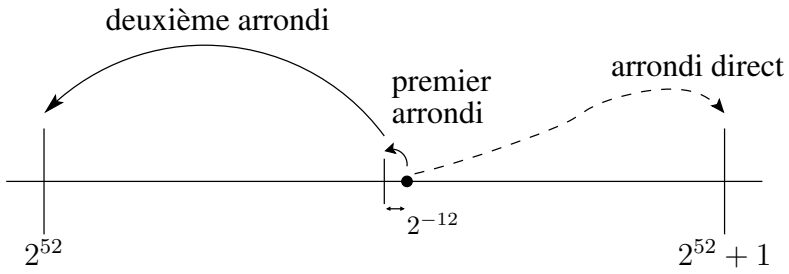

FIG. 8.2 – Franchissement des bornes d'erreur par double arrondi

n'est pas le cas ici car la propagation des erreurs d'arrondi provoque des changements de signes.

Double arrondi

Le dernier problème qui peut être rencontré est lié à une mauvaise interaction entre compilateur, architecture et environnement logiciel. Considérons l'exemple d'un utilisateur compilant à l'aide de GCC des exécutables pour architectures de type x86 tournant sous GNU/Linux. Un processeur x86 utilise des registres flottants au format double étendu (64 bits de mantisse au lieu des 53 attendus pour `double`). La précision des calculs peut néanmoins être contrôlée pour que l'arrondi des résultats s'effectue bien au niveau du 53^{e} bit.

Cependant le mode d'arrondi utilisé par défaut sous GNU/Linux demande l'emploi de la précision maximale et le compilateur devrait donc générer des instructions de changement de précision à chaque fois que des calculs impliquent des `double`. Ces instructions sont malheureusement extrêmement coûteuses car le processeur doit vider son pipeline d'instructions. Par conséquent, le compilateur ne les génère pas, estimant que les utilisateurs seront bien plus satisfaits d'un code plus rapide et généralement plus précis.

Il y a cependant une situation dans laquelle, malgré l'utilisation d'une précision plus élevée, le code produit un résultat moins précis. Appuyons-nous sur l'exemple de la somme approchée $2^{52} \oplus ((2^{11}+1) \cdot 2^{-12})$ représenté sur la figure 8.2. En double précision (53 bits de mantisse), les nombres flottants 2^{52} et $2^{52}+1$ sont consécutifs. Or la somme exacte vaut légèrement plus de $2^{52} + \frac{1}{2}$. Le résultat approché au plus près est donc $2^{52}+1$.

En utilisant la précision étendue (64 bits), le résultat stocké dans les registres du processeur est cependant $2^{52}+\frac{1}{2}$. Il est ensuite arrondi une deuxième fois lors du stockage de la variable en mémoire avec une précision de 53 bits. La valeur finale est cette fois 2^{52}. L'erreur relative effectivement commise est donc supérieure à la valeur traditionnellement admise de 2^{-53}.

8.1.2 Approche par filtre

Supposons que les composantes des vecteurs pq et pr soient toutes bornées par 1 en valeur absolue et qu'il n'y ait pas de phénomène de double arrondi. Le script Gappa 8.1 borne l'erreur commise lors du calcul ; il s'agit de la distance entre la valeur calculée `det` et la valeur réelle du déterminant `Det`.

Script Gappa 8.1 Erreur commise lors du calcul du déterminant 2×2

```
@rnd = float< ieee_64, ne >;

pqx = rnd(qx - px); pqy = rnd(qy - py);
prx = rnd(rx - px); pry = rnd(ry - py);
det rnd= pqx * pry - pqy * prx;

Det = (qx-px) * (ry-py) - (qy-py) * (rx-px);

{ pqx in [-1,1] /\ pqy in [-1,1] /\
  prx in [-1,1] /\ pry in [-1,1]
-> det - Det in ? }
```

Gappa répond que l'erreur absolue commise δ est de l'ordre de 2^{-51}. Quand la valeur calculée du déterminant `det` est supérieure à δ, son signe est identique à celui de la valeur réelle `Det`. Les trois dernières lignes du code 8.2 prennent en compte cette propriété. Par conséquent, soit le signe correct est renvoyé, soit la valeur retournée indique qu'aucun résultat garanti n'a été obtenu.

Code C 8.2 Implantation bornée robuste du prédicat d'orientation

```
double pqx = qx - px, pqy = qy - py;
double prx = rx - px, pry = ry - py;
double det = pqx * pry - pqy * prx;
if (det > +delta) return POSITIVE;
if (det < -delta) return NEGATIVE;
return UNKNOWN
```

La fonction ne constitue plus alors un prédicat dans son intégralité, mais seulement sa première phase. Il s'agit d'une approche par filtres [DP98]. En cas d'échec de cette première phase, une deuxième plus lente mais plus précise est lancée. Il peut s'agir d'un filtre à base d'arithmétique d'intervalles. Si ce n'est toujours pas suffisant, une évaluation à base d'arithmétique exacte peut être exécutée en dernier recours pour garantir que le prédicat renvoie toujours un résultat correct.

8.2 Nouvelle arithmétique

La tentative précédente suppose que chacune des composantes des vecteurs est bornée par 1 en valeur absolue. Plaçons-nous maintenant dans le cas général. Multiplier une des lignes du déterminant par un coefficient multiplie d'autant la valeur du déterminant. S'il en était de même pour les bornes sur l'erreur, la proposition suivante serait vérifiée :

$$|\mathtt{det} - \mathtt{Det}| \leq \delta \cdot m_x \cdot m_y$$

Dans cette formule, δ désigne à nouveau la borne sur l'erreur absolue obtenue quand toutes les composantes sont bornées par 1 en valeur absolue. Les facteurs m_x et m_y valent quant à eux $\max(|\mathtt{pqx}|, |\mathtt{prx}|)$ et $\max(|\mathtt{pqy}|, |\mathtt{pry}|)$.

Cette proposition n'est malheureusement pas vérifiée. L'emploi de δ conduit à sous-estimer l'erreur totale causée par les arrondis. Nous allons donc construire par induction une nouvelle borne, légèrement plus grande que δ, pour laquelle la proposition est vérifiée.

8.2.1 Addition et multiplication

Supposons que a et b soient deux expressions encadrées par les intervalles $f(m_x, m_y) \cdot A$ et $g(m_x, m_y) \cdot B$. Dans ces termes, A et B désignent des intervalles et f et g des fonctions à valeurs réelles positives ou nulles sur l'ensemble des paires de réels positifs ou nuls. Dans le cas particulier du prédicat d'orientation, ce sont de simples monômes à coefficients positifs.

En réordonnant les termes et en appliquant des règles d'arithmétique d'intervalles, on construit pour le produit $a \cdot b$ un encadrement de structure similaire ($f \cdot g$ y désigne la fonction produit de f et g) :

$$a \cdot b \in (f \cdot g)(m_x, m_y) \cdot (A \cdot B).$$

Pour l'addition, si l'on suppose $f = g$, on peut construire l'encadrement suivant :

$$a + b \in f(m_x, m_y) \cdot (A + B).$$

8.2.2 Arrondis

Supposons dans ce qui suit qu'il n'y a aucun dépassement de capacité. Soit ϵ_0 une borne de l'erreur relative commise en arrondissant un nombre. Soit η_0 une borne de l'erreur absolue qui couvre le domaine des nombres dénormalisés sur lequel la borne d'erreur relative n'est pas valide. L'erreur d'arrondi est alors majorée par la formule suivante :

$$\forall x \in \mathbb{R}, \quad \epsilon_0 \cdot |x| \geq \eta_0 \quad \Rightarrow \quad |\circ(x) - x| \leq \epsilon_0 \cdot \max(|x|, \eta_0/\epsilon_0)$$

En double précision, les valeurs de ces constantes sont $\epsilon_0 = 2^{-53}$ et $\eta_0 = 2^{-1075}$. Pour prendre en compte le phénomène de double arrondi, il est nécessaire d'agrandir un peu ces constantes : $\epsilon_0 = 2^{-53} \cdot (1+2^{-11})$ et $\eta_0 = 2^{-1075} \cdot (1+2^{-12})$.

Supposons à nouveau que l'expression a soit encadrée par $f(m_x, m_y) \cdot A$. Ajoutons comme hypothèse $[-\frac{\eta_0}{\epsilon_0}, \frac{\eta_0}{\epsilon_0}] \subseteq f(m_x, m_y) \cdot A$ pour contourner le maximum qui apparaît dans la proposition précédente. En notant $|A|$ la magnitude de l'intervalle A, l'encadrement de l'erreur absolue devient :

$$\circ(a) - a \in f(m_x, m_y) \cdot [-\epsilon_0 \cdot |A|, \epsilon_0 \cdot |A|].$$

L'encadrement précédent ne convient pas pour borner les erreurs d'arrondi associées aux quatre premières soustractions du filtre, c'est-à-dire au calcul des composantes des vecteurs. La valeur de $f(m_x, m_y) \cdot A$ est en effet inconnue lors de l'exécution de l'algorithme en machine. Plutôt que d'utiliser un encadrement de a, il est préférable d'utiliser un encadrement de la valeur arrondie : $\circ(a) \in f(m_x, m_y) \cdot A'$. L'expression a étant ici une soustraction, l'existence de nombres dénormalisés permet d'utiliser l'erreur relative sur l'ensemble du domaine :

$$\circ(a) - a \in f(m_x, m_y) \cdot [-\epsilon_0 \cdot |A'|, \epsilon_0 \cdot |A'|].$$

8.2.3 Derniers détails

Les théorèmes précédents permettent de calculer l'encadrement d'une expression composée. Encore faut-il savoir encadrer une expression élémentaire, c'est-à-dire les termes `pqx`, `prx`, `pqy` et `pry`. Par définition de m_x et m_y, ce sont :

$$\begin{aligned} \texttt{pqx}, \texttt{prx} &\in m_x \cdot [-1, 1] \\ \texttt{pqy}, \texttt{pry} &\in m_y \cdot [-1, 1] \end{aligned}$$

Ces encadrements permettent donc d'initialiser la récurrence, les théorèmes la font ensuite progresser jusqu'à encadrer $\texttt{det} - \texttt{Det}$. Des règles de réécriture sont employées afin de faire apparaître les termes nécessaires. Par exemple, pour encadrer l'erreur entre `det` et `Det`, le terme $t = \texttt{pqx} \otimes \texttt{pry} - \texttt{pqy} \otimes \texttt{prx}$ est introduit :

$$\texttt{det} - \texttt{Det} = (\texttt{det} - t) + (t - \texttt{Det}).$$

Il y a maintenant suffisamment de briques pour construire la preuve d'une borne d'erreur du prédicat. Pour ce qui est des théorèmes d'addition et de multiplication, la seule différence avec les théorèmes traditionnels d'arithmétique par intervalles réside dans la présence des facteurs de la forme $f(m_x, m_y)$. Pour les arrondis, la différence est plus marquée puisque ce sont des intervalles un peu plus grands que si l'on avait calculé une erreur absolue ; le paragraphe 8.2.4 ci-après contourne ce problème en introduisant des opérateurs d'arrondi spécifiques. Quant aux règles de réécritures, elles sont identiques à celles utilisées pour traiter les erreurs absolues.

Par conséquent, si l'on en retire toute notion de facteur d'homogénéité, la preuve n'est pas différente de ce qu'on obtiendrait en cherchant à borner l'erreur absolue `det − Det` quand les termes `pqx`, `prx`, `pqy` et `pry` prennent leurs valeurs dans l'intervalle $[-1, 1]$.

8.2.4 Transcription Gappa

À la présence des facteurs près, Gappa est donc directement utilisable pour encadrer la différence `det − Det`. J'ai introduit deux opérateurs d'arrondi pour gérer les prédicats homogènes. Le premier est utilisé pour chacune des soustractions initiales. Le second est utilisé pour les opérations suivantes. Le fait d'utiliser deux opérateurs au lieu d'un seul oblige Gappa à respecter les contraintes d'homogénéité des théorèmes.

Le script 8.2 est la transcription du problème dans la syntaxe de Gappa. L'intervalle calculé donne la constante à utiliser dans le filtre. En revanche, la preuve générée n'a que peu d'utilité : ce n'est réellement qu'une preuve concernant une erreur absolue et la vérification des facteurs d'homogénéité n'en fait pas partie.

Script Gappa 8.2 Erreur homogène du déterminant

```
pqx = homogen80x_init(qx - px);
pqy = homogen80x_init(qy - py);
prx = homogen80x_init(rx - px);
pry = homogen80x_init(ry - py);
det homogen80x= pqx * pry - pqy * prx;
Det = (qx-px)*(ry-py) - (qy-py)*(rx-px);

{ pqx in [-1,1] /\ pqy in [-1,1] /\
  prx in [-1,1] /\ pry in [-1,1] ->
  det - Det in ? }
```

8.3 Filtre semi-statique robuste

Gappa répond une borne δ de l'erreur absolue de l'ordre de $2^{-50} \simeq 8.9 \cdot 10^{-16}$. Nous savons donc que si la distance $|\texttt{det} - \texttt{Det}|$ est inférieure à $\delta \cdot m_x \cdot m_y$ alors les deux termes ont le même signe. L'implantation ne peut cependant pas directement calculer le produit $\delta \cdot m_x \cdot m_y$ en arithmétique flottante : cela introduirait potentiellement une erreur de calcul supplémentaire. Nous avons donc artificiellement augmenté la valeur de δ afin que cette erreur de calcul n'influe pas sur le résultat de la comparaison. Gappa a servi à vérifier que cette inflation était suffisante.

8.3.1 Implantation

Pour que le filtre se comporte bien, il faut aussi assurer que le calcul de ce produit $\delta \cdot m_x \cdot m_y$ ne fait intervenir aucun résultat dénormalisé. De plus, aucun des calculs dans le filtre ne doit provoquer un dépassement de capacité. Le filtre contient donc des comparaisons pour éviter ces situations. Là encore, Gappa a été employé pour vérifier que ces comparaisons sont suffisantes. Une fois les diverses comparaisons ajoutées, le filtre écrit en C++ ressemble au code 8.3.

Code C 8.3 Implantation robuste du prédicat d'orientation

```
double pqx = qx - px, pqy = qy - py;
double prx = rx - px, pry = ry - py;
double det = pqx * pry - pqy * prx;

double maxx = max(abs(pqx), abs(prx));
double maxy = max(abs(pqy), abs(pry));
double eps = 8.8872057372592758e-16 * maxx * maxy;
if (maxx > maxy) swap(maxx, maxy);

if (maxx < 1e-146) {
  if (maxx == 0) return ZERO;
} else if (maxy < 1e153) {
  if (det > eps)  return POSITIVE;
  if (det < -eps) return NEGATIVE;
}

// fall back to a more accurate, slower method
...
```

8.3.2 Performances

Les performances des prédicats ont été mesurées sur une triangulation de Delaunay de 10^5 points tirés au hasard dans le cube unité [DP03]. L'algorithme employé est celui de CGAL et il utilise intensivement les prédicats `orientation` et `insphere`. Les tests ont été compilés à l'aide du compilateur GNU g++ version 4.0 (options de compilation : `-O3 -DNDEBUG`). Le tableau 8.1 montre, pour différentes implantations des prédicats, le temps moyen d'exécution sur trois tests.

Le jeu de points ne provoque pas d'échec du premier filtre des prédicats. Ce sont donc les performances de cette première étape qui sont mesurées. À titre de comparaison, le temps obtenu à l'aide des filtres statiques écrits par Shewchuk [She97] est aussi indiqué. Nos filtres sont légèrement plus rapides, mais ce n'est qu'anecdotique. Ce qui importe vraiment, c'est qu'ils produisent des résultats corrects même si les calculs intermédiaires subissent *underflow* ou *overflow*. Qui plus est, ils résistent aussi aux phénomènes potentiels de double arrondi et ne nécessitent pas que de la double précision correcte soit présente : certains calculs

Type de prédicat	Durée (s)
algorithmes naïfs non robustes	3.29
nos filtres + ...[2]	4.33
calcul par intervalles + ...[2]	12.5
calcul exact	296
prédicats de Shewchuk	4.39

TAB. 8.1 – Performances d'une triangulation de Delaunay en trois dimensions

peuvent être effectués en précision étendue. Ces filtres robustes ont été ajoutés à CGAL.

8.3.3 Généralisation

Écrire à la main les différents scripts Gappa pour le prédicat d'orientation dans le plan ne pose pas de difficultés. En revanche, les prédicats de degré supérieur sont vite fastidieux. La méthode décrite dans ce chapitre s'applique en effet à n'importe quel prédicat qui implique l'évaluation d'un polynôme homogène. Il s'avère que la plupart des prédicats géométriques le sont puisqu'ils manipulent des longueurs.

En plus du prédicat d'orientation 2D, des filtres ont ainsi été écrits pour le prédicat d'orientation 3D ainsi que pour les prédicats indiquant si un point est inscrit dans un cercle ou dans une sphère définis respectivement par trois ou quatre points. Le plus complexe des prédicats est ainsi un déterminant 5×5.

Comme CGAL possède des versions génériques des prédicats qui prennent en paramètre *template* une classe décrivant un type numérique, nous avons écrit une classe C++ qui s'interface avec ces prédicats afin de générer les scripts Gappa correspondants.

[2]Comme le jeu de points ne provoque pas d'échec du premier filtre, le choix des filtres employés en deuxième, voire en troisième, étape du prédicat n'influe pas sur les performances mesurées. Ces filtres ne sont donc pas précisés ; il peut s'agir d'une évaluation en arithmétique exacte.

Chapitre 9

Conclusion et perspectives

9.1 Un outil d'aide à la certification

Nous avons défini au cours de cette thèse un langage de haut niveau capable d'exprimer les propositions logiques dont la validité est requise lors de la certification d'applications numériques. L'outil Gappa a été conçu pour vérifier ces propositions et en générer des preuves formelles qui peuvent être réutilisées au sein de certifications qui couvrent plus que la partie numérique des applications.

Écrire de telles preuves à la main est un travail long et fastidieux. Il devient facile de négliger de prendre en compte certains termes comme le produit des erreurs relatives dans l'erreur relative du produit. Qui plus est, modifier légèrement l'algorithme peut nécessiter d'en refaire complètement la preuve. Gappa permet au développeur de contourner ces difficultés en automatisant toutes les étapes fastidieuses de la preuve, qu'elles concernent des encadrements de variables ou des calculs d'erreurs. Le temps nécessaire pour certifier un algorithme numérique en est considérablement diminué. Le développeur peut alors se permettre d'explorer les différentes variantes de son algorithme pour en retenir les plus performantes, Gappa se chargeant de valider l'algorithme à chaque nouveau changement.

Toutefois, si le développeur a mis à profit son expertise pour optimiser cet algorithme (en négligeant des termes, en réordonnant des calculs, etc), Gappa peut ne pas réussir à le certifier. Dans ce cas, le savoir du développeur doit être traduit dans la syntaxe de Gappa sous la forme de quelques lignes indiquant quelles bissections effectuer ou comment réécrire certaines expressions sous des formes plus facilement traitables.

L'intérêt de Gappa réside dans la facilité avec laquelle il permet de borner les erreurs d'arrondi qui apparaissent lors d'un calcul approché. Il a été employé avec succès pour des fonctions élémentaires de CRlibm [dDLM06], des filtres flottants dans CGAL [MP06], l'optimisation de polynômes [Rev06] et des opérateurs arithmétiques matériels [MTVC06]. Les nombreuses interactions avec les utilisateurs de l'outil ont permis une évolution rapide de ses fonctionnalités. Cela s'est traduit par la publication d'une trentaine de versions ; Gappa compte maintenant 6500

lignes de code C++ et la bibliothèque de support 8000 lignes de Coq.

9.2 Particularités de Gappa

Pour que l'outil génère des preuves formelles, il a fallu construire un système qui soit à la fois assez puissant pour pouvoir vérifier les propositions qui nous intéressent mais aussi assez compact pour que les théorèmes soient formalisables rapidement. Le choix s'est donc porté sur des intervalles fermés et bornés de nombres réels à bornes dyadiques associés à une arithmétique d'intervalles simple.

Afin de réduire l'influence de la corrélation et d'améliorer les bornes calculées, l'arithmétique a été augmentée de quelques opérateurs spécifiques tels que $\frac{x}{x}$ ou $x+y+x\cdot y$. Pour traiter les arithmétiques approchées, des opérateurs d'arrondi $\circ(x)$ ont aussi été ajoutés. Les erreurs d'arrondi causées par ces arithmétiques ont demandé l'ajout de règles pour borner $\circ(x)-x$ et $\frac{\circ(x)-x}{x}$ en fonction de x ou $\circ(x)$. Les intervalles n'offrent cependant pas une puissance suffisante pour prendre en compte le caractère discret des ensembles de valeurs dans les arithmétiques approchées. L'outil a donc été enrichi à l'aide de prédicats de précision qui permettent en particulier d'obtenir des bornes d'erreur fines en arithmétique à virgule fixe.

Les propriétés concernant les erreurs d'arrondi ne sont utiles que pour borner localement ces erreurs. Leur propagation à l'ensemble du problème se fait par un mécanisme de réécriture des expressions : les termes sont réordonnés pour faire apparaître dans l'erreur globale les expressions des erreurs locales. Les autres types de corrélation sont traités en laissant la possibilité à l'utilisateur de définir ses propres règles de réécriture ou de demander d'effectuer des bissections sur certains termes.

Les points précédents concernent la vérification des propositions, mais Gappa en génère aussi des preuves. Celles-ci ne sont pas la trace exhaustive de l'ensemble des calculs effectués par Gappa ; elles ont la particularité d'être concises. L'outil profite en effet de la puissance de calcul à sa disposition pour réduire autant que possible la taille du certificat : il élimine les étapes de raisonnement inutiles, fusionne des sous-cas de bissection et diminue la précision nécessaire pour effectuer les calculs.

9.3 Perspectives

Employer l'arithmétique d'intervalles à des fins de certification de programmes constitue un vaste sujet. Ce paragraphe va donc se restreindre à présenter quelques-unes des voies raisonnables que Gappa pourrait emprunter.

Une première piste qui peut être explorée est l'emploi de Gappa pour servir de fondation à une tactique Coq afin de vérifier des systèmes d'inéquations sur les nombres réels. Elle n'aurait pas la puissance de résolution de la tactique `omega` [Pug91] mais aurait par contre bien moins de restrictions, à commencer par l'absence des nombres réels justement. Elle permettrait aussi de gérer des

multiplications par des expressions non constantes, des divisions, des racines carrées, etc. La tactique s'apparenterait alors à la stratégie `numerical` implantée en PVS [ML05]. L'existence des intervalles à bornes dyadiques serait complètement transparente pour l'utilisateur : la tactique manipulerait directement des inégalités écrites dans le formalisme Coq concernant les nombres réels.

Une autre piste qu'il convient d'explorer est l'interface de Gappa avec des systèmes de certification plus généraux tels que Why [Fil03]. Cela nécessiterait une formalisation des arithmétiques approchées commune aux différents outils. Pour l'instant, Caduceus [FM04] assimile les types flottants `float` et `double` du langage C aux nombres réels `real` de Why. Il n'est donc pas possible d'employer ces outils pour certifier des programmes flottants. Cela serait pourtant très utile pour une bibliothèque comme CRlibm : son code contient de nombreuses structures conditionnelles et tables de valeurs. Comme la gestion de telles structures dépasse le cadre de Gappa, la certification de CRlibm passe par la génération de centaines de scripts Gappa. Ils correspondent aux différentes branches prises et aux valeurs des indices utilisés pour accéder aux tables.

Pour finir, il faut signaler le problème des expressions corrélées. Gappa n'a aucune difficulté à les traiter quand il s'agit d'expressions d'erreur et l'outil remplit donc son objectif. Pour des expressions purement mathématiques, la situation n'est pas aussi simple. L'emploi de bissections et de réécritures par l'utilisateur offre alors un moyen de contourner cette difficulté. Elles ne sont cependant efficaces que si les expressions n'ont qu'un degré de liberté. Pour décorréler des expressions aux variations plus complexes, ne serait-ce que des polynômes multivariés, il faudrait formaliser une arithmétique d'intervalles plus puissante. Les modèles de Taylor par exemple sont une des pistes à explorer en matière de preuve de propriétés numériques.

Annexe A

Travailler avec les intervalles

Plusieurs variations autour de l'arithmétique d'intervalles transparaissent dans ce document. Considérons quatre de celles qui ont été implantées et utilisées. Le paragraphe 2.3.1 expose les intervalles à bornes dyadiques choisis pour Gappa. Les intervalles employés au paragraphe 3.3.1 ont quant à eux des bornes rationnelles. Pour accélérer la recherche des paramètres au paragraphe 6.1, l'oracle peut substituer des bornes flottantes aux bornes rationnelles. Enfin, comme expliqué au paragraphe 4.1.4, la première version de Gappa utilisait elle aussi des intervalles à bornes flottantes. Mais contrairement aux trois premiers exemples, ce n'était pas pour manipuler des sous-ensembles de $\mathbb{R}$: ces intervalles représentaient des sous-ensembles de nombres flottants et l'arithmétique associée.

Ces exemples montrent que l'arithmétique d'intervalles se décline, d'une part suivant l'arithmétique des nombres manipulés, $\mathbb{R}$ ou $\mathbb{F}$ ici, et d'autre part suivant la représentation des bornes, $\mathbb{D}$, $\mathbb{Q}$ ou $\mathbb{F}$. D'autres variations d'implantation apparaissent dans ces différents exemples : Gappa emploie en interne des intervalles qui peuvent devenir vides ou non bornés, mais cette propriété n'est pas utile aux intervalles à bornes rationnelles et compliquerait leur implantation.

Ces différentes implantations utilisent la bibliothèque Boost[1] pour l'arithmétique d'intervalles [BMP03, BMP06b] conçue en collaboration avec Hervé Brönnimann et Sylvain Pion. Les travaux réalisés sur cette bibliothèque C++ nous ont conduits à écrire une proposition d'inclusion de l'arithmétique d'intervalles dans la norme décrivant le langage C++.

A.1 Boost

Pour être utilisable dans chacun des cas présentés ci-dessus, la bibliothèque d'arithmétique d'intervalles fournit une classe générique `interval` paramétrée par le type des bornes stockées dans l'intervalle et trois politiques influant sur les algorithmes implantés. La plus importante de ces politiques, *Rounding*, indique comment effectuer des opérations en arrondi dirigé sur les bornes. Ces opérations

[1] `http://www.boost.org/`

caractérisent l'arithmétique d'intervalles que la bibliothèque fournit. Par exemple, pour des intervalles à bornes flottantes, si les calculs approchés sont effectués au plus près (ou tout autre mode d'arrondi fixé), alors l'arithmétique résultante étend $\mathbb{F}$ aux intervalles. Si par contre les calculs sont arrondis vers le bas pour les bornes inférieures et vers le haut pour les bornes supérieures, l'arithmétique est cette fois celle des intervalles de $\mathbb{R}$. La même arithmétique s'obtient en utilisant des bornes rationnelles et des calculs exacts sur ces bornes, la différence résidant dans les intervalles représentables.

La deuxième politique, *Checking*, permet à la bibliothèque de traiter les cas exceptionnels que sont les intervalles non bornés ou vides et les nombres invalides. Par exemple, pour un type d'intervalles pour lequel les intervalles vides sont interdits, la bibliothèque accélère l'exécution de ses fonctions en considérant que les intervalles fournis en entrée contiennent toujours des éléments. La dernière politique, *Comparison*, indique comment sont définis les opérateurs de comparaison. Cette dernière politique est en fait fournie par des espaces de noms plutôt que comme paramètre de classe. Cela évite à l'utilisateur d'avoir à modifier le type des intervalles s'il a localement besoin d'un type de comparaison particulier. Il lui suffit de charger l'espace de noms correspondant à ce type de comparaison.

En matière d'intervalles à bornes flottantes, la bibliothèque fournit des types spécialisés qui optimisent le nombre de changements de mode d'arrondi effectués. Ces changements sont coûteux et il est préférable de regrouper les opérations effectuées avec la même direction d'arrondi. L'approche retenue dans plusieurs bibliothèques est d'imposer à l'utilisateur un mode d'arrondi dirigé pour l'ensemble du programme, perturbant de ce fait toute opération flottante qui n'est pas liée à ces bibliothèques d'intervalles. Nous avons jugée cette approche trop restrictive. L'utilisateur n'a donc pas à se préoccuper du mode d'arrondi du processeur : chaque opération sur les intervalles s'en chargera. Il peut cependant changer localement le type de ses intervalles pour indiquer à la bibliothèque qu'elle est seule à utiliser l'unité flottante dans cette portion du programme et qu'elle peut donc optimiser les changements de direction d'arrondi sans risque de perturber le comportement du reste du programme.

A.2 Normalisation

Suite à nos travaux sur cette bibliothèque, nous avons soumis une proposition [BMP06c] pour incorporer l'arithmétique d'intervalles à la norme du langage C++. Le caractère générique de la bibliothèque Boost lui permet de traiter un grand nombre d'arithmétiques mais elle implique l'existence de nombreux paramètres pour l'utilisateur. Afin de simplifier la proposition, nous avons préféré nous concentrer sur une arithmétique d'intervalles unique. Le choix s'est porté sur l'extension de $\mathbb{R}$ aux intervalles.

Le seul paramètre de la classe `std::interval` est le type des bornes. L'absence de politique *Rounding* empêche de spécifier comment arrondir les bornes.

Les spécialisations de la classe pour des types utilisateurs ne sont donc pas définies. La situation est par conséquent analogue à celle de `std::complex` : seules les spécialisations associées aux types flottants sont détaillées.

Les fonctions proposées respectent toutes la propriété d'inclusion, y compris les fonctions d'entrée-sortie et les fonctions transcendantes. Les intervalles vides sont des intervalles à part entière et les fonctions les génèrent quand leurs entrées sont complètement en dehors du domaine des opérateurs qu'elles implantent. Par exemple, $\sqrt{[-3,1]} = [0,1]$ et $\sqrt{[-3,-2]} = \emptyset$; le résultat est défini dans les deux cas et aucune exception n'est déclenchée.

Contrairement aux opérateurs arithmétiques, les opérateurs de comparaison sur les intervalles ne sont pas accessibles par défaut. L'approche adoptée est la même que pour Boost : les opérateurs sont rangés dans des espaces de noms. L'un de ces espaces fournit des opérateurs étendant les comparaisons entre nombre réels et renvoyant des ensembles de booléens [BMP06a]. Ainsi $([0,1] < [2,3]) = \{true\}$, $([2,3] < [1,2]) = \{false\}$, $([0,2] < [1,3]) = \{false, true\}$ et $([0,1] < \emptyset) = \emptyset$. En projetant les ensembles $\{true\}$ et $\{false, true\}$ sur le booléen $true$ et les autres sur $false$, on obtient le jeu des comparaisons « possibles » fournies par un autre espace de noms. Il existe aussi un espace de comparaisons « certaines » et un espace fournissant des opérateurs basés sur l'inclusion d'intervalles.

L'objectif de notre proposition est de fournir systématiquement aux utilisateurs du langage C++ les moyens d'effectuer des calculs par intervalles et donc de favoriser la création de programmes robustes. De plus, la présence d'intervalles flottants pourrait pousser les concepteurs de compilateurs à mieux traiter les calculs flottants et les modes d'arrondi dirigés décrits par la norme IEEE-754.

Annexe B

Syntaxe et sémantique du langage Gappa

Cette annexe décrit le langage employé par Gappa. Dans celui-ci, espaces, tabulations et retours à la ligne n'ont pas de signification autre que de séparer explicitement les lexèmes du langage. Un retour à la ligne marque cependant la fin d'un commentaire entamé par `#`.

B.1 Expressions réelles

B.1.1 Nombres

Les nombres peuvent être représentés sous trois formats différents : le format décimal scientifique traditionnel `0.17e+2`, le format flottant hexadécimal introduit dans le langage C99 `0x1.1p+4` et un format spécifique permettant de représenter les nombres dyadiques `34b-1`.

Lors de l'interprétation d'un nombre au format décimal ou hexadécimal, le « point décimal » est déplacé vers la droite jusqu'à disparaître. Les nombres précédents vont donc être lus respectivement comme `17e+0` et `0x11p+0`. Le problème ne se pose pas pour le format dyadique puisqu'il n'autorise pas de point décimal. Une fois cette transformation effectuée, on obtient un nombre sous la forme $m \cdot \beta^e$ avec m et e des entiers relatifs. La base β vaut 10 si le nombre était explicitement au format décimal scientifique, 2 dans le cas contraire. Une dernière simplification est appliquée au nombre. Tant que e est strictement négatif et que m est divisible par β, le nombre est remplacé par $(m/\beta) \cdot \beta^{e-1}$.

Deux nombres sont syntaxiquement égaux s'ils valent zéro, s'ils ont la même mantisse et un exposant égal à zéro, ou si les triplets mantisse–base–exposant sont égaux. Ainsi, les trois chaînes de caractères `0.17e+2`, `0x1.1p+4` et `34b-1` sont associées à la même expression réelle (qui vaut 17). Par contre, les chaînes `1b1` et `2` sont deux représentations syntaxiquement différentes du nombre 2.

B.1.2 Opérateurs, fonctions et expressions

Gappa reconnaît les opérateurs `+`, `-`, `*`, `/` ainsi que la fonction `sqrt(`e`)` avec leur signification traditionnelle. La valeur absolue est notée par une paire de barres `|`e`|`. La fonction `fma(`a`,`b`,`c`)` est interprétée en $a \cdot b + c$.

Les opérateurs d'arrondi sont des fonctions. Certaines de ces fonctions nécessitent des paramètres (indication de précision, direction d'arrondi, etc) passés entre angles à la suite du nom de fonction. Exemple : `fixed<`$prec$`,`dir`>(`e`)`. Les opérateurs d'arrondi généralisés sont des fonctions à plusieurs arguments.

L'ensemble des expressions est constitué des nombres, potentiellement préfixés d'un signe, des identifiants et de leurs constructions avec les opérateurs et fonctions décrites précédemment. La grammaire est la suivante (les priorités usuelles des opérateurs binaires infixes sont respectées).

$$
\begin{array}{rcl}
\mathit{round} & ::= & \mathit{ident}\ [\ \text{`<'}\ \mathit{param}\ \{\ \text{`,'}\ \mathit{param}\ \}\ \text{`>'}\ ] \\
\mathit{expr} & ::= & \mathit{number} \\
& | & \mathit{ident} \\
& | & \text{`(' } \mathit{expr} \text{ `)'} \\
& | & \text{`|' } \mathit{expr} \text{ `|'} \\
& | & (\ \text{`+'} \mid \text{`-'}\ )\ \mathit{expr} \\
& | & \mathit{expr}\ (\ \text{`+'} \mid \text{`-'} \mid \text{`*'} \mid \text{`/'}\ )\ \mathit{expr} \\
& | & \texttt{sqrt}\ \text{`(' } \mathit{expr} \text{ `)'} \\
& | & \texttt{fma}\ \text{`(' } \mathit{expr} \text{ `,' } \mathit{expr} \text{ `,' } \mathit{expr} \text{ `)'} \\
& | & \mathit{round}\ \text{`(' } \mathit{expr}\ \{\ \text{`,'}\ \mathit{expr}\ \}\ \text{`)'}
\end{array}
$$

B.2 Script Gappa

Un script se décompose en trois parties. La première partie permet de d'associer des sous-expressions à des identifiants, afin de simplifier les expressions qui apparaissent dans la deuxième partie. Celle-ci contient la proposition logique que l'outil aura à vérifier. Finalement, la troisième partie contient les indices devant aider Gappa à vérifier la proposition.

$$\mathit{script} ::= \{\ \mathit{def}\ \}\ \text{`\{' } \mathit{prop} \text{ `\}'}\ \{\ \mathit{hint}\ \}$$

B.2.1 Définitions d'expression

Pour éviter d'avoir à fournir tous les paramètres des opérateurs d'arrondi à chaque fois qu'ils sont utilisés, il est possible d'associer à des identifiants leurs versions spécialisées en préfixant la définition d'un `@`.

Les définitions d'expression se déclinent quant à elles sous deux formes. En l'absence d'un opérateur d'arrondi devant le symbole =, l'identifiant de gauche désigne exactement l'expression de droite. Par contre, si un opérateur unaire d'arrondi est indiqué, il sera implicitement appliqué dans l'expression de droite au résultat de chaque opération binaire `+`, `-`, `*`, `/` ainsi qu'au résultat de chaque fonction `sqrt` ou `fma`.

La présence d'un opérateur d'arrondi permet donc d'écrire des expressions telles qu'elles apparaîtraient dans des langages comme C, C++ ou Java, et la sémantique d'évaluation serait la même. Ce ne serait pas le cas en Fortran par exemple, puisque le langage autorise les compilateurs à considérer les opérations flottantes comme présentant des propriétés de commutativité, associativité et distributivité.

Un identifiant ne peut pas être défini après avoir été utilisé. S'il n'a pas été défini avant son utilisation, il représente une expression réelle à la valeur quelconque. Voici la grammaire des définitions.

$$
\begin{array}{rcl}
\mathit{def} & ::= & \text{`@'}\ \mathit{ident}\ \text{`='}\ \mathit{round}\ \text{`;'} \\
& | & \mathit{ident}\ [\ \mathit{round}\ ]\ \text{`='}\ \mathit{expr}\ \text{`;'}
\end{array}
$$

B.2.2 Proposition logique

Un encadrement d'expression s'exprime avec un intervalle dont les deux bornes sont des nombres. Le symbole `?` peut être substitué à l'intervalle ; Gappa cherchera alors un intervalle telle que la proposition soit satisfaite. Des inégalités larges peuvent être employées si une seule des bornes de l'intervalle est connue. Dans le cas particulier où l'expression est une valeur absolue, l'inégalité $\leq r$ est remplacée par l'encadrement $\in [0, r]$.

L'implication est le moins prioritaire des opérateurs logiques, suivie de la disjonction, puis de la conjonction, et enfin de l'opérateur de négation. La proposition est fournie à Gappa entre accolades et suit la grammaire suivante.

$$
\begin{array}{rcl}
\mathit{range} & ::= & \mathit{expr}\ \texttt{in}\ \text{`?'} \\
& | & \mathit{expr}\ \texttt{in}\ \text{`['}\ \mathit{number}\ \text{`,'}\ \mathit{number}\ \text{`]'} \\
& | & \mathit{expr}\ (\ \text{`<='} \mid \text{`>='}\ )\ \mathit{number} \\
\mathit{prop} & ::= & \mathit{range} \\
& | & \text{`('}\ \mathit{prop}\ \text{`)'} \\
& | & \texttt{not}\ \mathit{prop} \\
& | & \mathit{prop}\ (\ \text{`->'} \mid \text{`\/'} \mid \text{`/\'}\ )\ \mathit{prop}
\end{array}
$$

Gappa décompose la proposition logique en des implications dont le membre de gauche est une conjonction d'encadrements et le membre de droite des conjonctions et disjonctions d'encadrements. Chacune de ces implications est traitée séparément par l'outil.

B.2.3 Indications de résolution

Quand deux expressions sont séparées par le symbole `~`, celle de gauche est considérée par Gappa comme étant une approximation de celle de droite quand il instancie ses règles de réécritures. Quand deux expressions sont séparées par le symbole `->`, Gappa considère que tout encadrement de l'expression de droite peut être utilisé comme encadrement de celle de gauche.

Le symbole `$` demande à Gappa d'effectuer une étude de cas. Les expressions à droite du symbole indique quels encadrements seront découpés. Quand `in` est suivi d'un entier n, l'outil découpe l'encadrement considéré en n sous-intervalles égaux. Quand une liste de nombres est utilisée, le découpage se fait par rapport à ces nombres. Quand la partie `in` est omise, Gappa considère que le découpage doit se faire en quatre sous-intervalles égaux s'il n'y a pas d'expressions à gauche du `$`. S'il y en a, le découpage se fera par bisection jusqu'à ce que les encadrements des expressions sur la gauche satisfassent les contraintes exprimées dans la proposition logique.

split ::= *expr* [`in` (*integer* | '`(`' *number* { '`,`' *number* } '`)`')]
hint ::= *expr* '`~`' *expr* '`;`'
| *expr* '`->`' *expr* '`;`'
| [*expr* { '`,`' *expr* }] '`$`' *split* { '`,`' *split* } '`;`'

Annexe C

Liste des théorèmes

Cette annexe constitue un inventaire presque exhaustif des règles de réécriture et théorèmes présents dans Gappa. Les lettres de a à e représentent des expressions quelconques. Quand a est la partie exacte d'un couple d'expressions, $\tilde{a}$ représente la partie approchée. Le symbole $\circ$ est une fonction d'arrondi quelconque. Les lettres x et y désignent des nombres dyadiques et ξ un nombre explicite (décimal ou dyadique). Les différents prédicats employés par Gappa sont :

$$\begin{aligned}
\mathrm{BND}(e, [x, y]) &= x \leq e \leq y \\
\mathrm{ABS}(e, [x, y]) &= 0 \leq x \leq |e| \leq y \\
\mathrm{FIX}(e, k) &= \exists m, n \in \mathbb{Z}, e = m \cdot 2^n \wedge n \geq k \\
\mathrm{FLT}(e, k) &= \exists m, n \in \mathbb{Z}, e = m \cdot 2^n \wedge |m| < 2^k
\end{aligned}$$

Pour alléger les notations, la partie numérique des prédicats est omise, seule l'expression est indiquée dans les tableaux. Des informations sur la partie numérique sont cependant parfois indiquées ; $\mathrm{BND}(e, \geq -1)$ désigne ainsi un encadrement de e dont l'intervalle ne contient que des valeurs supérieures ou égales à -1.

Le tableau C.1 détaille les règles de réécriture présentes dans Gappa. Chaque ligne correspond à une réécriture. Quand l'outil cherche à encadrer une expression de la première colonne, il calcule l'intervalle auquel appartiennent les valeurs de l'expression de la deuxième colonne. Cet intervalle est alors utilisé dans l'encadrement de l'expression de la première colonne. La dernière colonne indique quant à elle les contraintes que l'outil doit vérifier pour pouvoir appliquer la réécriture. Une contrainte de la forme $a \neq b$ signifie que les expressions a et b doivent être syntaxiquement différentes.

Le tableau C.2 décrit succinctement les propriétés que Gappa peut calculer et les types d'hypothèses que cela nécessite. Là où ils sont simples, résultats et contraintes ont été indiqués.

Cible	Intermédiaire	Contraintes
$-a--b$	$-(a-b)$	$a \neq b$
$(-a--b)/-b$	$(a-b)/b$	$\mathrm{ABS}(b,>0), a \neq b$
$\tilde{a}+c$	$(\tilde{a}-a)+(a+c)$	
$c+\tilde{a}$	$(c+a)+(\tilde{a}-a)$	
$(a+b)-(c+d)$	$(a-c)+(b-d)$	$a \neq c, b \neq d$
$((a+b)-(c+d))/(c+d)$	$((a-c)/c*c+(b-d)/d*d)/(c+d)$	$\mathrm{ABS}(c,>0), \mathrm{ABS}(d,>0), \mathrm{ABS}(c+d,>0)$
$(a+b)-(a+c)$	$b-c$	$b \neq c$
$(a+b)-(c+b)$	$a-c$	$a \neq c$
$\tilde{a}-c$	$(\tilde{a}-a)+(a-c)$	$a \neq c, \tilde{a} \neq c$
$c-\tilde{a}$	$(c-a)+-(\tilde{a}-a)$	$\tilde{a} \neq c$
$(a-b)-(c-d)$	$(a-c)+-(b-d)$	$a \neq c, b \neq d$
$((a-b)-(c-d))/(c-d)$	$((a-c)/c*c+(b-d)/d*-d)/(c+-d)$	$\mathrm{ABS}(c,>0), \mathrm{ABS}(d,>0), \mathrm{ABS}(c-d,>0)$
$(a-b)-(a-c)$	$-(b-c)$	$b \neq c$
$(a-b)-(c-b)$	$a-c$	$a \neq c$
$\tilde{a}*c$	$(\tilde{a}-a)*c+a*c$	
$c*\tilde{a}$	$c*(\tilde{a}-a)+c*a$	
$a*b-a*c$	$a*(b-c)$	$b \neq c$
$a*c-b*c$	$(a-b)*c$	$a \neq b$
$a*b-c*d$	$a*(b-d)+(a-c)*d$	$a \neq c, b \neq d$
$a*b-c*d$	$(a-c)*b+c*(b-d)$	$a \neq c, b \neq d$
$a*b-c*d$	$a*(b-d)+(a-c)*b+-((a-c)*(b-d))$	$a \neq c, b \neq d$
$a*b-c*d$	$c*(b-d)+(a-c)*d+(a-c)*(b-d)$	$a \neq c, b \neq d$
$(a*b-c*d)/(c*d)$	$(a-c)/c+(b-d)/d+((a-c)/c)*((b-d)/d)$	$\mathrm{ABS}(c,>0), \mathrm{ABS}(d,>0), a \neq c, b \neq d$
$(a*b-a*c)/(a*c)$	$(b-c)/c$	$\mathrm{ABS}(a,>0), \mathrm{ABS}(c,>0), b \neq c$
$(a*b-c*b)/(c*b)$	$(a-c)/c$	$\mathrm{ABS}(b,>0), \mathrm{ABS}(c,>0), a \neq c$
$(a/b-c/d)/(c/d)$	$((a-c)/c-(b-d)/d)/(1+(b-d)/d)$	$\mathrm{ABS}(b,>0), \mathrm{ABS}(c,>0), \mathrm{ABS}(d,>0), b \neq d$
$(a/b-c/b)/(c/b)$	$(a-c)/c$	$\mathrm{ABS}(b,>0), \mathrm{ABS}(c,>0), a \neq c$
$\sqrt{a}-\sqrt{b}$	$(a-b)/(\sqrt{a}+\sqrt{b})$	$\mathrm{BND}(a,\geq 0), \mathrm{BND}(b,\geq 0), a \neq b$
$(\sqrt{a}-\sqrt{b})/\sqrt{b}$	$\sqrt{1+(a-b)/b}-1$	$\mathrm{BND}(a,\geq 0), \mathrm{BND}(b,>0), a \neq b$
$c-a$	$(c-\tilde{a})+(\tilde{a}-a)$	$a \neq c, \tilde{a} \neq c$
$(\tilde{a}-c)/c$	$(\tilde{a}-a)/a+(a-c)/c+((\tilde{a}-a)/a)*((a-c)/c)$	$\mathrm{ABS}(c,>0), \mathrm{ABS}(a,>0), a \neq c, \tilde{a} \neq c$
$(c-a)/a$	$(c-\tilde{a})/\tilde{a}+(\tilde{a}-a)/a+((c-\tilde{a})/\tilde{a})*((\tilde{a}-a)/a)$	$\mathrm{ABS}(a,>0), \mathrm{ABS}(\tilde{a},>0), a \neq c, \tilde{a} \neq c$
$a-b$	$((a-b)/b)*b$	$\mathrm{ABS}(b,>0), a \neq b$
$1+(a-b)/b$	a/b	$\mathrm{ABS}(b,>0), a \neq b$
$\tilde{a}$	$a+(\tilde{a}-a)$	
a	$\tilde{a}+-(\tilde{a}-a)$	
$\tilde{a}$	$a*(1+(\tilde{a}-a)/a)$	$\mathrm{ABS}(a,>0)$
a	$\tilde{a}/(1+(\tilde{a}-a)/a)$	$\mathrm{ABS}(a,>0), \mathrm{ABS}(\tilde{a},>0)$
$\sqrt{a}*\sqrt{a}$	a	$\mathrm{BND}(a,\geq 0)$

TAB. C.1 – Règles de réécriture

Cible	Hypothèses
$\mathrm{BND}(\circ(a) - a)$	
$\mathrm{BND}(\circ(a) - a)$	$\mathrm{BND}(a)$
$\mathrm{BND}(\circ(a) - a)$	$\mathrm{BND}(\circ(a))$
$\mathrm{BND}(\circ(a) - a)$	$\mathrm{ABS}(a)$
$\mathrm{BND}(\circ(a) - a)$	$\mathrm{ABS}(\circ(a))$
$\mathrm{BND}((\circ(a) - a)/a)$	$\mathrm{BND}(a)$
$\mathrm{BND}((\circ(a) - a)/a)$	$\mathrm{BND}(\circ(a))$
$\mathrm{BND}((\circ(a) - a)/a)$	$\mathrm{ABS}(a)$
$\mathrm{BND}((\circ(a) - a)/a)$	$\mathrm{ABS}(\circ(a))$
$\mathrm{BND}(\circ(a))$	$\mathrm{BND}(a)$
$\mathrm{BND}(\circ(a))$	$\mathrm{BND}(\circ(a))$
$\mathrm{BND}(-a)$	$\mathrm{BND}(a)$
$\mathrm{BND}(\lvert a\rvert)$	$\mathrm{BND}(a)$
$\mathrm{BND}(\sqrt{a})$	$\mathrm{BND}(a, \geq 0)$
$\mathrm{BND}(a - a, \ni 0)$	
$\mathrm{BND}(a/a, \ni 1)$	$\mathrm{ABS}(a, > 0)$
$\mathrm{BND}(a * a)$	$\mathrm{BND}(a)$
$\mathrm{BND}(a + b)$	$\mathrm{BND}(a), \mathrm{BND}(b)$
$\mathrm{BND}(a - b)$	$\mathrm{BND}(a), \mathrm{BND}(b)$
$\mathrm{BND}(a * b)$	$\mathrm{BND}(a), \mathrm{BND}(b)$
$\mathrm{BND}(a/b)$	$\mathrm{BND}(a), \mathrm{BND}(b, \not\ni 0)$
$\mathrm{ABS}(-a)$	$\mathrm{ABS}(a)$
$\mathrm{ABS}(\lvert a\rvert)$	$\mathrm{ABS}(a)$
$\mathrm{ABS}(\sqrt{a})$	$\mathrm{ABS}(a)$
$\mathrm{ABS}(a + b)$	$\mathrm{ABS}(a), \mathrm{ABS}(b)$
$\mathrm{ABS}(a - b)$	$\mathrm{ABS}(a), \mathrm{ABS}(b)$
$\mathrm{ABS}(a * b)$	$\mathrm{ABS}(a), \mathrm{ABS}(b)$
$\mathrm{ABS}(a/b)$	$\mathrm{ABS}(a), \mathrm{ABS}(b, > 0)$
$\mathrm{BND}(a)$	$\mathrm{ABS}(a)$
$\mathrm{BND}(a)$	$\mathrm{BND}(a), \mathrm{ABS}(a)$
$\mathrm{BND}(\lvert a\rvert)$	$\mathrm{ABS}(a)$
$\mathrm{ABS}(a)$	$\mathrm{BND}(\lvert a\rvert)$
$\mathrm{BND}(a + b + a * b)$	$\mathrm{BND}(a, \geq -1), \mathrm{BND}(b, \geq -1)$
$\mathrm{BND}((c * a + d * b)/(a + b))$	$\mathrm{BND}(a), \mathrm{BND}(b), \mathrm{BND}(c), \mathrm{BND}(d)$
$\mathrm{BND}(\xi)$	
$\mathrm{FIX}(a + b)$	$\mathrm{FIX}(a), \mathrm{FIX}(b)$
$\mathrm{FIX}(a - b)$	$\mathrm{FIX}(a), \mathrm{FIX}(b)$
$\mathrm{FIX}(a * b)$	$\mathrm{FIX}(a), \mathrm{FIX}(b)$
$\mathrm{FLT}(a * b)$	$\mathrm{FLT}(a), \mathrm{FLT}(b)$
$\mathrm{FIX}(a)$	$\mathrm{FLT}(a), \mathrm{ABS}(a)$
$\mathrm{FLT}(a)$	$\mathrm{FIX}(a), \mathrm{ABS}(a)$
$\mathrm{FIX}(a)$	$\mathrm{BND}(a, = [x, x])$
$\mathrm{FLT}(a)$	$\mathrm{BND}(a, = [x, x])$
$\mathrm{FIX}(\texttt{fixed}(a))$	
$\mathrm{BND}(\texttt{fixed}(a) - a, \ni 0)$	$\mathrm{FIX}(a)$
$\mathrm{BND}(a)$	$\mathrm{BND}(a), \mathrm{FIX}(a)$
$\mathrm{FIX}(\texttt{float}(a))$	
$\mathrm{FLT}(\texttt{float}(a))$	
$\mathrm{BND}(\texttt{float}(a) - a, \ni 0)$	$\mathrm{FIX}(a), \mathrm{FLT}(a)$

TAB. C.2 – Théorèmes avec calcul

Table des scripts Gappa

Bibliographie

[ATD05] Behzad Akbarpour, Sofiène Tahar, and Abdelkader Dekdouk. Formalization of fixed-point arithmetic in HOL. *Formal Methods in System Design*, 27(1–2):173–200, 2005. Ref. pp. 6

[Bar89] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989. Ref. pp. 6

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004. Ref. pp. 5

[BFS01] Christoph Burnikel, Stefan Funke, and Michael Seel. Exact geometric computation using cascading. *International Journal of Computational Geometry and Applications*, 11(3):245–266, 2001. Ref. pp. 7

[BM05] Sylvie Boldo and Guillaume Melquiond. When double rounding is odd. In *Proceedings of the 17th IMACS World Congress on Computational and Applied Mathematics*, Paris, France, 2005. Ref. pp. 40

[BMP03] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The Boost interval arithmetic library. In *Proceedings of the 5th Conference on Real Numbers and Computers*, pages 65–80, Lyon, France, 2003. Ref. pp. 103

[BMP06a] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. Bool_set: multi-valued logic. Technical Report 2136, C++ standardization committee, 2006. Ref. pp. 105

[BMP06b] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The design of the Boost interval arithmetic library. *Theoretical Computer Science*, 351:111–118, 2006. Ref. pp. 103

[BMP06c] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. A proposal to add interval arithmetic to the C++ standard library. Technical Report 2137, C++ standardization committee, 2006. Ref. pp. 104

[BMZ02] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3–4):225–252, 2002. Ref. pp. 7

[Bol04] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, École Normale Supérieure de Lyon, 2004. Ref. pp. 6, 68

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software*, pages 515–529, 1997. Ref. pp. 70

[Cha06] Amine Chaieb. Verifying mixed real-integer quantifier elimination. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lectures Notes in Artificial Intelligence*, pages 528–540, Seattle, WA, USA, 2006. Ref. pp. 38

[Che95] Jean-Marie Chesneaux. *L'arithmétique stochastique et le logiciel CADNA*. Habilitation à diriger des recherches, Université Paris VI, Paris, France, 1995. Ref. pp. 7

[CM95] Victor A. Carreño and Paul S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *Proceedings of the International Workshop on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, UT, USA, 1995. Ref. pp. 6

[DdD05] Jérémie Detrey and Florent de Dinechin. A parameterizable floating-point logarithm operator for FPGAs. In *Proceedings of the 39th Asilomar Conference on Signals, Systems and Computers*, pages 1186–1190, Pacific Grove, CA, USA, 2005. IEEE Signal Processing Society. Ref. pp. 4

[DdDM01] David Defour, Florent de Dinechin, and Jean-Michel Muller. Correctly rounded exponential function in double precision arithmetic. In Franklin T. Luk, editor, *Advanced Signal Processing Algorithms, Architecture and Implementations XI*, pages 156–167, San Diego, CA, USA, 2001. Ref. pp. 79

[dDLM06] Florent de Dinechin, Christoph Lauter, and Guillaume Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1318–1322, Dijon, France, 2006. Ref. pp. 79, 84, 99

[Dek71] Theodorus J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971. Ref. pp. 83

[Dem84] James Demmel. Underflow and the reliability of numerical software. *SIAM Journal on Scientific and Statistical Computing*, 5(4):887–919, 1984. Ref. pp. 30

[DM04] Marc Daumas and Guillaume Melquiond. Generating formally certified bounds on values and round-off errors. In Vasco Brattka, Christiane Frougny, and Norbert Müller, editors, *Proceedings of the 6th Conference on Real Numbers and Computers*, pages 55–70, Schloß Dagstuhl, Germany, 2004. Ref. pp. 17, 36

[DMM05] Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed proofs using interval arithmetic. In Paolo Montuschi and Eric Schwarz, editors, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, pages 188–195, Cape Cod, MA, USA, 2005. Ref. pp. 15, 25, 67, 71

[DP98] Olivier Devillers and Franco Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete and Computational Geometry*, 20(4):523–547, 1998. Ref. pp. 93

[DP03] Olivier Devillers and Sylvain Pion. Efficient exact geometric predicates for Delaunay triangulations. In *Proceedings of the 5th Workshop on Algorithm Engineering and Expereriments*, pages 37–44, Baltimore, MA, USA, 2003. Ref. pp. 97

[DRT01] Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001. Ref. pp. 6, 39

[EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004. Ref. pp. 39

[FHL+05] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. Technical Report RR-5753, INRIA, 2005. Ref. pp. 53

[Fig95] Samuel A. Figueroa. When is double rounding innocuous? *SIGNUM Newsletter*, 30(3):21–26, 1995. Ref. pp. 37

[Fil03] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, 2003. Ref. pp. 2, 7, 101

[FM04] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Proceedings of the 6th International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, 2004. Ref. pp. 101

[GM05] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In John Hurd and Tom Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113, Oxford, UK, 2005. Ref. pp. 63

[Har96] John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*,

volume 1166 of *Lecture notes on Computer Sciences*, pages 265–269. Springer-Verlag, 1996. Ref. pp. 80

[Har97] John Harrison. Floating point verification in HOL light: The exponential function. In *Algebraic Methodology and Software Technology*, pages 246–260, 1997. Ref. pp. 6, 80

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 113–130, Nice, France, 1999. Ref. pp. 6

[HT98] John Harrison and Laurent Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998. Ref. pp. 67

[IEE85] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, 1985. Ref. pp. 34

[Inf92] Information Management and Technology Division. Patriot missile defense: software problem led to system failure at Dhahran, Saudi Arabia. Technical Report B-247094, United States General Accounting Office, 1992. Ref. pp. 1

[ISO05] ISO. *ISO/IEC 15408-2005, Common Criteria for Information Technology Security Evaluation*, 2005. Ref. pp. 1

[JB05] Christian Jacobi and Christoph Berg. Formal verification of the VAMP floating point unit. *Formal Methods and System Design*, 26(3):227–266, 2005. Ref. pp. 6

[JKDW01] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer-Verlag, 2001. Ref. pp. 1, 19

[KI04] Valentine Kabanets and Russell Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. *Journal of Computational Complexity*, 13:1–46, 2004. Ref. pp. 63

[KK03] Roope Kaivola and Katherine R. Kohatsu. Proof engineering in the large: formal verification of Pentium 4 floating-point divider. *International Journal on Software Tools for Technology Transfer*, 4(3):323–334, 2003. Ref. pp. 6

[KMP+04] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. In *Proceedings of the 12th European Symposium on Algorithms*, volume 3221 of *Lecture Notes on Computer Sciences*, pages 702–713. Springer-Verlag, 2004. Ref. pp. 91

[Knu69] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, 1969. Ref. pp. 83

[L+96] Jacques-Louis Lions et al. Ariane 5 flight 501 failure report by the inquiry board. Technical report, European Space Agency, Paris, France, 1996. Ref. pp. 1

[Lan00] Philippe Langlois. Applications of the CENA method to automatic accuracy enhancement. In *Proceedings of the SCAN 2000 International Conference on Scientific Computing, Computer Arithmetic and Validated Numerics*, Karlsruhe, Germany, 2000. Ref. pp. 7

[Lau05] Christoph Q. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR2005-38, Laboratoire de l'Informatique du Parallélisme, 2005. Ref. pp. 85

[LBD03] Ren-Cang Li, Sylvie Boldo, and Marc Daumas. Theorems on efficient argument reductions. In Jean-Claude Bajard and Michael Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, pages 129–136, Santiago de Compostela, Spain, 2003. Ref. pp. 80

[MB03] Kyoko Makino and Martin Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 4:379–456, 2003. Ref. pp. 31

[ML05] César Muñoz and David Lester. Real number calculations and theorem proving. In John Hurd and Tom Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 195–210, Oxford, UK, 2005. Ref. pp. 25, 67, 101

[MLK98] J. Strother Moore, Tom W. Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating-point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, 1998. Ref. pp. 6

[Moo79] Ramon E. Moore. *Methods and Applications of Interval Analysis*. SIAM, 1979. Ref. pp. 1, 13, 60

[MP05] Guillaume Melquiond and Sylvain Pion. Formal certification of arithmetic filters for geometric predicates. In *Proceedings of the 17th IMACS World Congress on Computational and Applied Mathematics*, Paris, France, 2005. Ref. pp. 89

[MP06] Guillaume Melquiond and Sylvain Pion. Formally certified floating-point filters for homogeneous geometric predicates. *Theoretical Informatics and Applications*, 2006. To appear. Ref. pp. 89, 99

[MTVC06] Romain Michard, Arnaud Tisserand, and Nicolas Veyrat-Charvillon. Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. In *SympA Symposium en Architecture de Machines*, Perpignan, France, 2006. Ref. pp. 99

[MV99] Bruce D. McCullough and Hrishikesh D. Vinod. The numerical reliability of econometric software, 1999. Ref. pp. 1

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, USA, 1992. Springer-Verlag. Ref. pp. 15

[Pat95] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–78, 1995. Ref. pp. 15, 45

[PGM04] Sylvie Putot, Éric Goubault, and Matthieu Martel. Static analysis-based validation of floating-point computations. *Lecture Notes in Computer Science*, 2991:306–313, 2004. Ref. pp. 6, 29

[Pri91] Douglas M. Priest. Algorithms for arbitrary precision floating point arithmetic. In Peter Kornerup and David W. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–144, Grenoble, France, 1991. Ref. pp. 85

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 4–13, Albuquerque, NM, USA, 1991. Ref. pp. 100

[Rev06] Guillaume Revy. Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Technical Report RR2006-02, Laboratoire de l'Informatique du Parallélisme, 2006. Ref. pp. 99

[Rus00] David M. Russinoff. A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon processor. *Lecture Notes in Computer Science*, 1954:3–36, 2000. Ref. pp. 6

[RV01] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*, volume 1. Elsevier and MIT Press, 2001. Ref. pp. 51

[Sch80] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of ACM*, 27(4):701–717, 1980. Ref. pp. 63

[SG02] Jun Sawada and Ruben Gamboa. Mechanical verification of a square root algorithm using Taylor's theorem. In Mark Aagaard and John W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 274–291, Portland, OR, USA, 2002. Ref. pp. 6

[She97] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Computational Geometry*, 18(3):305–363, 1997. Ref. pp. 97

[Ste74] Pat H. Sterbenz. *Floating Point Computation*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1974. Ref. pp. 47

[Tan90] Ping-Tak Peter Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378–400, 1990. Ref. pp. 80

[Tex97] Texas Instrument. *TMS320C3x User's Guide*, 1997. Ref. pp. 34

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. *Lecture Notes in Computer Science*, 2031:299–313, 2001. Ref. pp. 2, 7

[YD95] Chee-Keng Yap and Thomas Dubé. The exact computation paradigm. In *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995. Ref. pp. 90

[Zum06] Roland Zumkeller. Formal global optimisation with Taylor models. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lectures Notes in Artificial Intelligence*, pages 408–422, Seattle, WA, USA, 2006. Ref. pp. 15

Résumé

Parce que les nombres manipulés en machine ont généralement un domaine et une précision limités, il est nécessaire de certifier soigneusement que les applications les utilisant se comportent correctement. Réaliser une telle certification à la main est cependant un travail propice à de nombreuses erreurs. Les méthodes formelles permettent de garantir l'absence de ces erreurs, mais le processus de certification est alors long, fastidieux et généralement réservé à des spécialistes.

Le travail effectué au cours de cette thèse vise à rendre ces méthodes accessibles en automatisant leur application. L'approche adoptée s'appuie sur une arithmétique d'intervalles accompagnée d'une base de théorèmes sur les propriétés des arithmétiques approchées et d'un mécanisme de réécriture d'expressions permettant le calcul de bornes fines sur les erreurs d'arrondi.

Ce travail s'est concrétisé par le développement de l'outil Gappa d'aide à la certification. Il permet de vérifier les propriétés de codes numériques qui utilisent de l'arithmétique à virgule fixe ou à virgule flottante. Cette vérification s'accompagne de la génération d'une preuve formelle de ces propriétés utilisable par l'assistant de preuves Coq. Gappa a été utilisé avec succès pour certifier la correction de fonctions dans les bibliothèques CRlibm, CGAL et FLIP par exemple.

Mots-clés : certification de programmes, arithmétique à virgule flottante, arithmétique à virgule fixe, arithmétique d'intervalles, méthodes formelles, assistant de preuve Coq.

Abstract

Computer numbers are usually limited, both in range and in precision. As a consequence, a careful certification has to be performed for applications that compute with these sets of numbers. Unfortunately, performing such a certification by hand is error-prone. Formal methods can ensure that the certification is correct, but making use of them is usually long and tedious, even for experts.

This thesis aims at improving the availability of these methods to developers by automatizing their implementation. The key concepts are the use of interval arithmetic, a database of theorems on computer arithmetics, and a system for rewriting expressions in order to compute tight bounds on rounding errors.

This approach has led to the development of the Gappa tool. It is designed to verify the numeric properties of programs relying on floating-point or fixed-point arithmetic. When verifying these properties, the tool also generates formal proofs of their correctness. These proofs can later be mechanically checked by the Coq proof assistant. Gappa has been successfully used for certifying some functions of the CRlibm, CGAL, and FLIP libraries, among others.

Keywords: program certification, floating-point arithmetic, fixed-point arithmetic, interval arithmetic, formal methods, Coq proof assistant.