Arithmétiques relationnelles pour l'analyse par interprétation abstraite de propriétés de précision numérique

Maxime Jacquemin

▶ **To cite this version:**

Maxime Jacquemin. Arithmétiques relationnelles pour l'analyse par interprétation abstraite de propriétés de précision numérique. Arithmétique des ordinateurs. Université Paris-Saclay, 2021. Français. NNT : 2021UPASG050 . tel-03566701

HAL Id: tel-03566701
https://theses.hal.science/tel-03566701

Submitted on 11 Feb 2022

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

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

université
PARIS-SACLAY

Arithmétiques relationnelles pour l'analyse par interprétation abstraite de propriétés de précision numérique

Relational Arithmetics for Abstract Interpretation Based Analysis of Numerical Accuracy Properties

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Unité de recherche : Université Paris-Saclay, CEA, Institut LIST, 91191, Gif-sur-Yvette, France
Référent : Faculté des sciences d'Orsay

Thèse présentée et soutenue à Paris-Saclay, le 15 juillet 2021, par

Maxime JACQUEMIN

Composition du jury :

Sylvie BOLDO Directrice de recherche, INRIA Saclay-Île-de-France Université Paris-Saclay	Présidente
Pierre-Loïc GAROCHE Professeur, École nationale de l'aviation civile	Rapporteur & Examinateur
Laure GONNORD Maître de conférence, Université Lyon 1	Rapportrice & Examinatrice
Antoine MINÉ Professeur, Sorbonne Université	Examinateur
Laura TITOLO Chargée de recherche, National Institut of Aerospace	Examinatrice

Direction de la thèse :

Julien SIGNOLES Ingénieur chercheur, HDR, CEA-LIST Université Paris-Saclay	Directeur de thèse
Franck VÉDRINE Ingénieur chercheur, CEA-LIST Université Paris-Saclay	Coencadrant

Thèse de doctorat
NNT : 2021UPASG050

À ma grand-mère, Michèle,
partie avant que je ne finisse.

Remerciements

Écrire ces pages apporte une conclusion à pratiquement quatre années de travail, avec ses moments de joie passionnés et ses passages difficiles. Naturellement, beaucoup de personnes m'ont aidé au cours de cette thèse, que ce soit intellectuellement, en réfléchissant avec moi, ou émotionnellement, en m'apportant soutien et réconfort. L'ouverture de ce manuscrit, ainsi que son point final en terme d'écriture, leurs sont donc dédiées.

J'aimerais commencer par remercier Laure Gonnord et Pierre-Loïc Garoche pour avoir accepter de rapporter ce manuscrit. Vos retours ont été très instructifs et intéressants à lire, malgré des délais un peu courts.

Je remercie également Sylvie Boldo, Laura Titolo et Antoine Miné pour avoir accepté de faire parti du jury de ma soutenance. J'ai apprécié de pouvoir vous présenter mon travail et d'en discuter avec vous durant les questions. Je remercie tout particulièrement Sylvie pour avoir accepté d'endosser le rôle de présidente. I would also like to address a special thanks to Laura, who have accepted to be part of my jury even if she is not a native french speaker.

Je remercie Sylvie Putot pour avoir été ma directrice pendant la première moitié de ma thèse. Malgré nos différents, ce fut pour moi une expérience enrichissante.

Julien Signoles, merci d'avoir accepté d'endosser le rôle de directeur pour la seconde moitié de ma thèse alors que mon travail est très éloigné de ta zone de confort. Ce fut pour moi un vrai plaisir d'essayer de te réconcilier avec les mathématiques dans les réels. Ton soutien et tes conseils, tant sur l'écriture que sur le métier de chercheur, m'ont été d'une aide précieuse.

Franck Védrine, merci pour ces quatre ans et demi de collaborations passionnantes. De mes débuts en recherche au cours de mon stage, jusqu'à aujourd'hui où je termine ce manuscrit, j'ai adoré travaillé avec toi. Merci pour cette opportunité. Je pense que je chérirai longtemps nos débats, pour le moins animés, devant ces tableaux couverts de symboles cabalistiques.

J'aimerais également remercié l'ensemble de mes collègues au laboratoire pour tous les échanges enrichissants et les excellents moments que nous avons pu passer ensemble, que ce soit au cours de pauses café légendaires ou en dehors du travail.

Je remercie tout particulièrement les membres de l'équipe *Eva* — David Bühler, Valentin Perrelle, André Maronoze et Michele Malberti — pour toute l'aide qu'ils m'ont apporté pour le développement de mes travaux. La qualité de mon code est en bonne partie dû à vos conseils.

Une pensée également pour tous mes camarades doctorants — Yaëlle, Lesly, Julien, Olivier, Quentin, Dara, Yackolley, et beaucoup d'autres — les dynamiques de soutien mutuel que nous avons mis en place m'ont beaucoup aidé à affronter les difficultés inhérentes aux dysfonctionnements systémiques de la thèse.

Je remercie également mes amis — Thibault Soret, Jérôme Duclert, Romain Chevet, Corentin Toutois et tous les autres — ainsi que ma famille — Natacha, Léo, Kristina et ma maman — pour leur soutien indéfectible et pour tous les bons moments passés ensemble.

Merci à toi Claire Dechoux pour toutes ces années. Notre relation fut incroyable, et je sais que je n'aurais pas pu voir le bout de cette thèse sans toi. Tes petits animaux trop mignons sont maintenant devenus grands.

J'oublie probablement beaucoup de monde et j'en suis désolé. A toutes celles et ceux qui se sentiraient bafouer de ne pas figurer sur ces pages, n'hésitez pas à venir m'en parler, ce sera l'occasion d'aller prendre un café.

Table des matières

Table des figures

Introduction

L'informatique est aujourd'hui présente dans toutes nos activités. Nos échanges sont numériques, nos rencontres aussi, nos divertissements, notre énergie, nos modes de transports, nos réveils, nos bloc-notes, à chaque instant, pour chaque besoin, l'outil informatique est partie intégrante de nos solutions. Même les discours politiques sont empreints de *digital*. Aussi impropre le choix de mot soit-il, il englobe malgré tout l'importance du logiciel dans notre vie quotidienne. Cette omniprésence du logiciel reste cependant relativement invisible, et lorsque l'on parle de ceux cachés derrières nos services numériques, c'est plus souvent pour en critiquer les défauts et problèmes que pour en vanter les mérites.

En effet, le logiciel est partout, et semble systématiquement accompagné de problèmes, ces bugs toujours déplaisants et dont nous parlons si souvent. La sortie d'un grand jeu vidéo est ainsi quasiment toujours accompagnée d'une avalanche de reproches sur ses défauts, tant et si bien que certains éditeurs sont mêmes réputés pour leur expertise à mettre en vente des jeux remplis de problèmes flagrants, souvent irritants, parfois amusants. Cependant, si cela reste le plus souvent acceptable dans les jeux vidéos, il y a des domaines pour lesquels l'existence d'un bug est tout simplement inacceptable. Personne ne monterait par exemple dans un avion si son logiciel de contrôle commande était développé avec le même soin que la dernière application mobile en vogue.

L'aéronautique n'est qu'un des nombreux exemples de ces domaines dit *critiques*, c'est-à-dire des domaines dans lesquels il est nécessaire de pouvoir avoir confiance dans les logiciels utilisés, notamment car des vies pourraient être en jeu, ou une erreur pourrait représenter des pertes économiques catastrophiques. Nous pourrions également citer le nucléaire, l'automobile, les télécommunications, la santé, l'armement ou encore les systèmes bancaires. Pour tous ces domaines, et pour plein d'autres encore, la problématique est la même : comment assurer qu'un logiciel fait ce que nous attendons de lui ?

Vérification de programmes

La première approche venant généralement à l'esprit lorsque l'on cherche à

augmenter la confiance que l'on peut accorder à un logiciel est de le tester le plus possibles, c'est-à-dire de vérifier que le programme se comporte correctement pour plein de données d'entrées différentes. Cette approche est très efficace, et permet d'éliminer généralement une grande partie des bugs. Cependant, dans la plupart des cas, l'espace des entrées est beaucoup trop vaste pour permettre de mener une campagne de test exhaustive. Cela signifie, comme le remarquait E. W. Dijkstra [27], qu'une approche par tests permet d'éliminer des bugs, mais pas d'en prouver l'absence. Atteindre les niveaux d'exigence recherchés dans les domaines critiques demande donc tellement de tests qu'il devient économiquement intéressant d'aborder le problème différemment, et de chercher à vérifier formellement les logiciels critiques.

Malheureusement, prouver qu'un logiciel est correct est une tâche ardue. En effet, il faut pour cela écrire, en plus du programme lui-même, une *spécification*, c'est-à-dire une description, écrite dans un langage mathématico-logique formel afin de pouvoir raisonner dessus, de ce que le programme est supposé faire, et une *preuve* que le logiciel correspond effectivement à cette spécification. Chacune de ces tâches représente une difficulté. En effet, d'une part, écrire la spécification nécessite d'utiliser des langages souvent très éloignés de nos langues naturels, ce qui requiert une certaine expertise et, d'autre part, l'écriture de la preuve repose souvent sur des raisonnements mathématiques complexes, ce qui rend l'exercice impossible sans recourir à des outils informatiques d'aide à la preuve. Ces difficultés nuisent tant à nos capacités à prouver qu'un logiciel est correct qu'à la confiance que l'on peut accorder à une telle preuve. Ainsi, il est légitime de se demander, d'une part, comment assurer que la spécification décrit bien le comportement que nous souhaitions spécifier, et d'autre part, comment vérifier, voire automatiser, le processus de preuve.

La première interrogation dispose d'une réponse simple, mais peu satisfaisante : il n'existe pas de manière de garantir qu'une spécification décrit bien ce qu'elle est supposée spécifier. Cependant, d'une part, la spécification d'un programme est souvent plus simple, et plus courte, que ce dernier, ce qui réduit les possibilités de bug et, d'autre part, bien qu'il soit possible d'écrire une spécification erronée pour un programme présentant un bug, il reste fortement improbable de réussir à prouver que le second satisfait la première. Notons d'ailleurs qu'en pratique, ce risque est minoré en séparant les deux activités de spécification et de vérification, qui sont réalisées indépendamment et par des équipes différentes.

La seconde interrogation, concernant la vérification et l'automatisation du processus de preuve, dispose, quant à elle, de plusieurs réponses, correspondant à différents choix pour contourner les difficultés de l'exercice. En effet, H. G. Rice a démontré [70] en 1953 qu'il n'existait pas d'algorithme permettant de vérifier en temps fini et systématiquement qu'un programme satisfait une propriété non tri-

viale arbitraire. Dit autrement, la construction de la preuve qu'un logiciel satisfait sa spécification est, sauf pour des spécifications très simple, un processus indécidable. Il est nécessaire de faire des concessions pour pouvoir satisfaire le besoin en outils de vérification de programmes, c'est-à-dire de relâcher au moins l'une des contraintes du théorème de Rice.

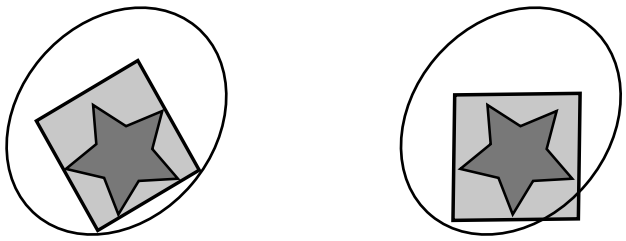
Une première contrainte que nous pouvons relâcher est celle de l'automatisation. Cette idée est au cœur de la *vérification déductive*, méthode reposant sur le calcul de plus faible précondition [28] introduit par Edsger W. Dijkstra et consistant à calculer, pour une instruction donnée, la plus faible condition à satisfaire avant son exécution pour garantir qu'une propriété donnée est satisfaite après l'exécution de l'instruction. Pour que cette approche fonctionne, il est cependant nécessaire de spécifier manuellement, pour chaque boucle du programme, un invariant, c'est-à-dire une propriété satisfaite après chaque itération, et un variant, c'est-à-dire une mesure qui décroit strictement à chaque itération et assure ainsi la terminaison. De plus, la méthode nécessite de faire appel à des prouveurs externes, adressant eux-mêmes des problèmes indécidables, et qu'il faut donc parfois guider afin de réussir à terminer la preuve.

Une deuxième contrainte qu'il est possible de relâcher concerne l'objet d'étude. L'idée est de chercher à construire la preuve, non pas pour le programme lui-même, mais pour un modèle de ce dernier. C'est l'approche explorée par le *model checking*, introduit au même moment par, d'un côté, E. Allen Emerson et Edmund M. Clarke [30], et de l'autre, Jean-Pierre Queille et Joseph Sifakis [69]. La simplification du problème à la vérification d'un modèle permet de rendre la vérification de propriétés non triviales décidables. Cependant, cette simplification peut amener à un problème d'explosion combinatoire, c'est-à-dire à la construction d'un modèle tellement grand que vérifier n'importe quelle propriété devient impossible à faire en un temps raisonnable, et ce quand bien même cette vérification est décidable. De plus, cela soulève la question de comment vérifier que le modèle correspond bien au programme.

Une troisième contrainte que nous pouvons relâcher est le caractère systématique de la réponse. L'idée est de calculer une *sur-approximation* des comportements du logiciel, c'est-à-dire une représentation mathématique décrivant un ensemble au moins aussi grand que l'ensemble des comportements possibles du programme, mais pouvant également décrire des comportements impossibles, puis de vérifier si cette sur-approximation satisfait la spécification. Si c'est le cas, alors le programme vérifie bien sa spécification car tous ces possibles comportements appartiennent à la sur-approximation, et vérifient donc la spécification. Par contre, si la sur-approximation ne vérifie pas la spécification, nous ne pouvons pas conclure que le programme est incorrect. Cela pourrait, en effet, être dû à la présence d'un comportement impossible dans la sur-approximation. Cette approche néces-

site donc de s'autoriser à pouvoir répondre « je ne sais pas ».

La figure 0.1 est une illustration de cette notion de sur-approximation. Dans les deux illustrations, la grande ellipse blanche correspond aux comportements jugés corrects par la spécification et l'étoile grise foncée correspond aux comportements possibles d'un programme. Ces derniers sont tous inclus dans l'ellipse, le programme n'a donc pas de bug. Les deux illustrations présentent également des carrés gris clair différents, correspondant chacun à une sur-approximation des comportements possibles du programme. Dans la première illustration, le carré est inclus dans l'ellipse. Cette sur-approximation constitue une preuve que le programme est correct. Dans la seconde illustration, en revanche, le carré n'est pas inclus dans l'ellipse. Cette sur-approximation ne permet donc pas de déterminer si le programme satisfait ou non sa spécification.



(a) Précise (b) Imprécise

FIGURE 0.1 – Illustration d'une sur-approximation précise (a) et imprécise (b). L'ellipse blanche correspond à la spécification, l'étoile grise foncée aux comportements possibles d'un programme et les carrés gris clairs à différentes sur-approximations.

Cette approche pour contourner l'indécidabilité de la vérification automatique de programme correspond à la théorie de *l'interprétation abstraite*, introduite par Radhia et Patrick Cousot [18]. Cette théorie définit un cadre formel général permettant de construire énormément d'analyses de programmes différentes. Les *domaines abstraits* sont les éléments centraux de ce cadre, spécifiant, entre autres, la sur-approximation utilisée, et régissant ce que les analyses sur lesquelles elles reposent peuvent prouver. Un exemple de domaine abstrait très simple est le domaine des signes, qui sur-approxime les comportements d'un programme en ne s'intéressant qu'aux signes de ses variables, et permet donc de prouver notamment des propriétés sur le signe de son résultat, comme le fait d'être un nombre positif.

Construire une analyse par interprétation abstraite consiste ainsi à définir un domaine abstrait adapté aux propriétés que l'on cherche à prouver. Cependant, il est souvent avantageux de ne pas repartir de zéro lorsque l'on définit un nouveau domaine abstrait, et de plutôt chercher à combiner des domaines existants afin de répondre à notre besoin. Une approche standard pour réaliser une telle

combinaison est le *produit réduit*, consistant à définir, en quelque sorte, un protocole d'échange d'informations entre deux domaines. Considérons deux domaines simples pour illustrer cela. Le premier associe à chaque variable du programme un ensemble de taille bornée à l'avance d'entiers, ou bien une valeur spéciale spécifiant que l'on ne sait pas à quels entiers une variable peut être associée. Par exemple, ce domaine peut associer à une variable x l'ensemble $\{2, 3, 4, 7\}$. Le second permet de garder des informations de congruence sur chaque variable. Par exemple, ce domaine peut permettre de prouver que la valeur de x congrue modulo deux vaut toujours zéro, ou dit autrement, que x est un nombre pair. Séparément, aucun de ces deux domaines ne peut prouver que x est plus petit que 5. Cependant, si l'on définit un protocole d'échange spécifiant d'éliminer tous les entiers de l'ensemble $\{2, 3, 4, 7\}$ qui ne sont pas congrus modulo deux, alors nous pouvons associer x à l'ensemble $\{2, 4\}$ et ainsi prouver la propriété désirée.

Ainsi, il existe plusieurs approches permettant de vérifier formellement la correction d'un programme, chacune offrant un compromis différent face à l'indécidabilité du problème. Dans le cadre des travaux présentés ici, nous nous intéressons spécifiquement à des programmes manipulant des nombres réels. Ce type de programmes représentent une difficulté particulière, découlant du fait qu'il est impossible de représenter la grande majorité des nombres réels dans nos ordinateurs.

Problème de confiance et nombres réels

Cette problématique est d'autant plus complexe lorsque le logiciel auquel nous nous intéressons effectue de nombreux calculs mathématiques utilisant des nombres réels. Prenons un exemple tout à fait classique, mais illustrant parfaitement pourquoi assurer que ce type de logiciels se comporte correctement est si complexe. Le 25 février 1991, une batterie anti-missile américaine de type Patriot MIM-104 échoue à intercepter une frappe sur les casernes de Dhahran, en Arabie Saoudite, pendant la première guerre du Golfe, ce qui cause la mort de 28 personnes. L'investigation menée par le gouvernement américain [7] indique que cet échec découle d'une erreur de logiciel du système de coordination. En effet, la batterie était en fonctionnement depuis plus d'une centaine heures. Or, avec le temps, le système accumulait de petites erreurs d'arrondis sur le calcul de son estimation du temps écoulé depuis le démarrage, impliquant un décalage progressif de la position perçue de la cible avec sa position réelle. Au moment de la tentative d'interception, l'erreur accumulée était environ 0.35 seconde, ce qui représente, à la vitesse du missile, un décalage de plus de 600 mètres. Ce qu'il faut comprendre ici, c'est que, dans une vision idéale dans laquelle nos ordinateurs seraient des machines à calculer infiniment précises, le logiciel du système de coordination ne présenterait pas de bug et il fonctionnerait correctement. Le problème, ici, est qu'il est strictement

impossible de représenter certains nombres réels avec une mémoire finie. Cela signifie qu'il est nécessaire de faire une approximation : calculer avec des réels dans nos ordinateurs implique forcément de n'utiliser qu'un sous-ensemble fini d'entre eux, disposant d'une représentation finie. La restriction utilisée représente un compromis entre précision et performance, et c'est l'approximation induite par cette restriction qui n'a pas été prise en compte et a causé la dérive dans le calcul du temps.

Le choix maximisant la précision correspond au domaine du *calcul formel*, consistant à utiliser des représentations symboliques des nombres, et à étudier les algorithmes permettant de calculer avec ces symboles. Nous retrouvons cette solution dans des outils comme MATLAB [37] ou SageMath [1] par exemple, souvent utilisés durant la phase de modélisation d'un logiciel effectuant des calculs complexes. Les représentations symboliques dont disposent ces outils sont, en pratique, suffisantes pour la plupart des applications, et permettent donc d'effectuer les calculs sans introduire d'erreurs. Cependant, les temps de calcul sont la plupart du temps bien trop grands pour que ces outils soient utilisables en production. Ces logiciels sont principalement utilisés en amont du développement, pendant leurs modélisations. Ainsi, de nos jours, le compromis le plus largement répandu dans les codes des logiciels critiques, et présentant des performances spectaculaires, est *l'arithmétique flottante* [2, 34]. L'idée est, intuitivement, de représenter les nombres en *notation scientifique*, mais en limitant le nombre de chiffres significatifs et l'intervalle dans lequel l'exposant évolue et en utilisant une base binaire plutôt que décimale. Par exemple, le nombre 42 est représenté sous la forme 1.3125×2^5. Cela présente plusieurs avantages : d'abord, la mémoire nécessaire pour représenter un nombre flottant est constante, ensuite, leur répartition au sein de l'ensemble des réels permet de représenter précisément à la fois des nombres très proches de zéro et des nombres gigantesques, et enfin, les opérations arithmétiques sont suffisamment simples pour être réalisées directement et efficacement par un processeur. Cependant, malgré ces immenses qualités en termes de précision et de performance, ce compromis présente le défaut d'être contre intuitif pour un humain et donc difficile à appréhender. La plupart des développeurs ont ainsi appris durant leurs études que les nombres flottants sont compliqués, mais que ce n'est pas si grave et qu'ils peuvent s'en servir comme des réels, ce qui est souvent faux et peut donc provoquer des erreurs.

Cette notion d'erreur, consistant en l'écart entre le comportement du programme lorsqu'il repose sur les nombres flottants et celui lorsqu'il repose sur les nombres réels peut être définie de deux manières différentes mais complémentaires. Pour pouvoir construire une intuition de ce que sont ces deux définitions, considérons un programme calculant une distance arbitraire et imaginons que, pour une certaine entrée, le calcul idéal, utilisant des nombres réels, retourne 12 mètres

alors que le programme retourne, quant à lui, 11.7 mètres. La première manière de définir l'erreur que commet notre programme par rapport au calcul idéal est simplement de considérer l'écart entre les deux valeurs, ici 0.3 mètre. On appelle cette définition *l'erreur absolue*. La seconde approche consiste à définir l'erreur comme un pourcentage d'écart avec le résultat attendu : c'est *l'erreur relative*. Dans notre exemple, l'erreur relative commise est de 2.5%. Ces deux définitions apportent chacune un éclairage différent sur l'erreur commise par notre programme, et, en fonction du contexte, il peut être intéressant d'utiliser l'une ou l'autre de ces définitions. Elles sont également liées : 2.5% de 12 mètres correspond en effet à 0.3 mètre, soit l'erreur absolue. C'est à ces deux notions d'erreurs que nous nous sommes intéressés au cours de cette thèse, et en particulier aux liens existants entre elles.

Apport de la présente thèse

Les travaux présentés dans cette thèse ont pour objectif la définition d'une analyse de programme pour l'évaluation des erreurs introduites par l'arithmétique flottante. Pour ce type de programme, la spécification est toute trouvée : nous voulons montrer que le programme se comporte le plus possible comme s'il reposait sur des réels et non des flottants. Dit autrement, nous voulons montrer que l'erreur introduite par l'utilisation de l'arithmétique flottante est négligeable. Nous nous intéressons donc à vérifier une spécification implicite.

Nous avons choisi d'utiliser une approche par interprétation abstraite, car cette méthode présente deux caractéristiques intéressantes pour la vérification de programmes flottants. La première est son caractère automatique, permettant de construire des analyses utilisables par des non experts, ce qui est particulièrement intéressant pour la vérification de programmes reposant sur l'arithmétique flottante, complexe et contre intuitive. La seconde est son fonctionnement par sur-approximation. En effet, cela présente l'avantage de permettre de fournir à l'utilisateur des bornes sur les erreurs introduites à chaque étape du programme, et ce même si ces bornes sont trop larges pour prouver que le programme satisfait sa spécification. L'utilisateur dispose ainsi de données pouvant lui apporter une meilleure compréhension de son programme, et ainsi l'aider à y trouver les bugs.

Nos travaux consistent donc en la définition d'un domaine abstrait dédié à l'analyse des erreurs d'arrondi. Comme nous les présentons par la suite, ils existent d'autres outils utilisant l'interprétation abstraite pour l'analyse des erreurs d'arrondi, comme *Fluctuat* [29] et *PRECiSA* [83] par exemple. Cependant, ces deux outils, tout comme une majeure partie des outils d'analyse automatique dédiés aux erreurs d'arrondi, sont focalisés sur l'erreur absolue. Ce choix est motivé par le fait que, dans la majorité des cas d'usage, il est préférable de disposer de bornes sur

les erreurs absolues. Cependant, le fonctionnement de l'arithmétique flottante fait qu'il est souvent à la fois plus simple et plus précis de réfléchir en termes d'erreurs relatives.

Afin de prendre ces deux aspects en considération, notre première contribution consiste donc en la définition et la formalisation d'un nouveau domaine abstrait permettant de calculer des bornes à la fois pour les erreurs absolues et pour les erreurs relatives, et profitant du lien entre ces deux définitions pour améliorer, via un produit réduit, la précision de l'analyse. Notre formalisation est générique, dans le sens où elle permet de construire un domaine abstrait dédié à l'analyse des erreurs numériques, et utilisant ce produit réduit, à partir de n'importe quel domaine abstrait permettant de sur-approximer le comportement dans les réels d'un programme. Le domaine ainsi construit exploite la relation entre les deux erreurs pour améliorer leurs bornes après chaque opération, mais également lors de l'interprétation des conditions. Nous avons également instancié notre formalisme, d'abord en utilisant le domaines des intervalles [19, 63], et ensuite en utilisant le domaine des zonotopes [42] reposant sur l'arithmétique affine et permettant de prendre en considération les relations entre les variables, ce qui nous a permis de montrer expérimentalement les gains de précision qu'apporte notre approche.

Cependant, ces expériences ont également mis en avant certaines limites de l'utilisation de l'arithmétique affine pour les erreurs relatives. Elles découlent du fait que la précision des calculs de ces erreurs dépend beaucoup de la précision du traitement des multiplications et divisions, et que ces deux opérations sont globalement imprécises en arithmétique affine. Nous avons donc travaillé sur deux approches pour traiter ce problème. La première part du constat que l'approche standard pour contrebalancer les imprécisions de l'arithmétique affine sur les multiplications et divisions, utilisée dans *Fluctuat* par exemple, consiste à construire un produit réduit avec l'arithmétique d'intervalles, mais que ce dernier est généralement naïf et sous-exploité. Notre deuxième contribution consiste donc en une nouvelle approche tirant mieux parti du produit réduit, ce qui permet d'améliorer la précision des analyses reposant sur l'arithmétique affine, et donc en particulier l'instanciation de notre approche reposant sur le domaine des zonotopes.

La seconde approche pour traiter le problème des imprécisions de l'arithmétique affine sur les multiplications et divisions part, quant à elle, du constat que, comme la précision des calculs des erreurs relatives dépend beaucoup de la précision du traitement de ces opérations, l'arithmétique affine n'est pas l'outil le plus adapté pour l'analyser précisément. Notre troisième contribution consiste donc en la définition et la formalisation d'un nouveau domaine abstrait dédié spécifiquement à l'analyse de l'erreur relative et permettant de prendre en considération les relations entre les variables du programme. L'idée centrale est de définir une sur-approximation permettant de représenter simplement et précisément le com-

portement et la propagation des erreurs relatives.

En plus de ces trois contributions théoriques, nous avons également travaillé à l'implémentation de nos approches au sein de la plateforme *Frama-C* [49], sous la forme d'un domaine abstrait pour le greffon *Eva* d'analyse par interprétation abstraite. Enfin, cette implémentation nous a permis de mettre en avant l'intérêt de nos solutions au travers d'une évaluation expérimentale de comparaison.

Autres approches existantes

Nous ne sommes évidemment pas les premiers à vouloir définir des méthodes permettant d'augmenter la confiance que l'on peut accorder à des programmes utilisant l'arithmétique flottante. Nous présentons donc ici différentes approches ayant ce même objectif. Ces approches peuvent être séparées en deux catégories. La première correspond aux solutions cherchant à construire des preuves de correction formelles, et reposant sur les méthodes de vérification de programmes que nous venons de présenter. Ces solutions offrent ainsi de fortes garanties, mais souffrent d'un problème de passage à l'échelle, les empêchant de traiter de grands programmes. La deuxième catégorie correspond aux solutions cherchant à simplifier la recherche de bugs liés à l'arithmétique flottante, mais sans construire de preuve de correction. Ces solutions permettent de traiter des programmes bien plus grands, mais ne permettent pas d'atteindre le même niveau d'exigence vis-à-vis de la correction du programme. Si nous présentons ces solutions, ce n'est donc pas pour les comparer à notre approche, mais plutôt pour offrir un aperçu plus large des méthodes existantes.

Recherche de bugs : analyses dynamiques

Les différentes méthodes de recherche de bugs présentées ci-après présentent toutes un fonctionnement similaire : elles consistent à exécuter le programme sur un jeu d'entrées donné et à analyser, au cours de l'exécution, son comportement afin de déterminer les instructions introduisant le plus d'erreurs. Ce sont donc des analyses dynamiques. Ces méthodes se différencient les unes des autres de deux manières : d'une part par l'approche utilisée pour estimer les erreurs introduites et, d'autre part, par la solution adoptée pour intégrer l'analyse à l'exécution du programme.

Une première solution pour intégrer l'analyse à l'exécution du programme consiste à modifier le processus de compilation de ce dernier de manière à intégrer l'analyse à l'exécutable généré, en utilisant par exemple les possibilités de surcharge des opérateurs de langage comme C++ pour instrumenter les opérations flottantes, ou bien en nécessitant l'introduction d'annotations d'instrumentation

dans le code source. C'est sur cette seconde solution que repose par exemple l'outil *Atomu* [91], qui propose d'estimer les erreurs introduites par chaque opération par le biais de son conditionnement, notion couramment utilisée en analyse numérique et correspondant à une mesure de la dépendance de la solution à un problème numérique vis-à-vis des données du problème. Un conditionnement élevé signifie qu'un calcul est sensible à des perturbations sur ses entrées, et qu'il risque donc d'amplifier la moindre erreur, causant une divergence avec le comportement attendu. L'outil *Shaman* [25] propose, quant à lui, d'évaluer l'erreur introduite par chaque opération en utilisant les transformations sans erreur (*error free transformations* [73] en anglais) des opérateurs arithmétiques, puis de propager ces erreurs en utilisant une approximation au premier ordre du comportement de l'arithmétique flottante. Cette approche permet d'obtenir des informations sur le nombre de chiffres réellement significatifs de chaque calcul, et présente l'avantage d'être *thread safe*, permettant l'instrumentation et l'analyse d'immenses programmes fortement parallèles. L'outil *NSan* [17] vise les mêmes objectifs de performance et de parallélisme que *Shaman*, mais repose sur l'utilisation d'une *mémoire d'ombre* pour exécuter toutes les instructions flottantes une seconde fois, mais avec une plus grande précision. Tout écart suffisamment grand entre les deux mémoires correspond alors à une instabilité du programme. Cette approche permet de fournir un rapport détaillé sur des sources d'imprécision, et offre de grandes performances grâce à son intégration à la chaîne de compilation *LLVM* [53]. *CADNA* [46] et *Verificarlo* [26] sont deux outils reposant sur des méthodes stochastiques pour évaluer les erreurs introduites par chaque opération. Intuitivement, l'idée est d'évaluer l'impact de changements dans le mode d'arrondi utilisé sur le comportement du programme. En changeant de manière aléatoire l'arrondi de chaque opération, on obtient une mesure des incertitudes introduites par le programme. En produisant suffisamment de mesures différentes, il est ensuite possible de construire une estimation statistique des erreurs vis-à-vis du comportement idéal du programme. Ces deux solutions offrent des estimations bien plus précises que les deux premiers, mais sont, en contrepartie, plus coûteuses.

Une seconde solution pour intégrer l'analyse à l'exécution du programme consiste à exécuter ce dernier dans un environnement virtuel contrôlé, en utilisant des outils comme Valgrind [66] par exemple, et d'instrumenter les instructions assembleur afin de réaliser l'analyse. Cette solution présente l'avantage de ne pas nécessiter l'accès au code source du programme, permettant, par exemple, d'analyser des programmes reposant sur des librairies externes. L'outil *Herbgrind* [76] propose ainsi une analyse confrontant l'exécution du programme utilisant l'arithmétique flottante avec une seconde exécution menée en parallèle de la première et calculant chaque instruction flottante dans une machine abstraite utilisant les nombres flottants à précision arbitraire que propose la librairie MPFR [35]. Ces

deux exécutions en parallèle permettent d'estimer les erreurs introduites, mais également de traquer les instructions introduisant de grandes erreurs et de trouver les potentielles divergences dans le flot d'exécution causées par des *tests instables* [84], c'est-à-dire des tests dont l'évaluation ne produit pas la même valeur de vérité en arithmétique flottante qu'en arithmétique réelle à cause des erreurs d'arrondi. L'outil *VERROU* [32], quant à lui, propose une évaluation stochastique des erreurs très proche de l'approche de *CADNA*, mais au niveau binaire.

Pour terminer, notons qu'au cours de cette thèse, nous avons eu l'occasion de travailler sur une analyse dynamique pour des programmes utilisant l'arithmétique flottante. Notre approche [87] consiste en une chaîne d'outils permettant, à partir d'un programme et de spécifications sur son comportement dans les réels, de générer un exécutable contenant un moteur d'analyse par interprétation abstraite dédié à l'évaluation des erreurs introduites par l'arithmétique flottante. Notre approche représente donc un hybride entre les méthodes statiques et dynamiques, permettant d'obtenir une sur-approximation précise des erreurs pour des jeux de tests présentant de petites incertitudes. De plus, ces travaux permettent de prendre en compte les *tests instables*, c'est-à-dire les conditions du programme dont la valeur de vérité change à cause des erreurs de l'arithmétique flottante, ce qui implique que le programme n'exécute pas le même code que celui qu'il aurait exécuté s'il avait pu utiliser des nombres réels. Ces divergences de flot d'exécution sont difficiles à analyser sans perdre trop de précision. Notre approche apporte une solution à ce problème en détectant automatiquement à la compilation les tests potentiellement instables. Pour chacun de ces tests, l'analyse détecte la partie du programme dont le flot d'exécution peut être directement impacté et la transforme de manière à pouvoir l'exécuter plusieurs fois. Cela nous permet d'explorer toutes les exécutions possibles et ainsi de construire une sur-approximation précise des erreurs introduites. Cette approche n'est cependant pas présentée dans la suite de ce manuscrit, et nous avons préféré nous concentrer sur les approches par analyse statique sur lesquels nous avons travaillé.

Preuve de correction : analyses statiques

Nous présentons maintenant différents outils permettant de construire des preuves de correction pour la précision des calculs flottants. Notons que la vaste majorité de ces outils construisent ces preuves de manière automatique. *Fluctuat* [29] est un des outils reposant sur l'interprétation abstraite et l'arithmétique affine [15] pour calculer des bornes sur l'erreur absolue de chaque variable du programme analysé. L'outil ne permet d'analyser que des programmes écrits en C, et a été utilisé à plusieurs reprises sur des codes industriels conséquents [12]. C'est également un outil présentant beaucoup de similarités avec notre proposition, mais qui ne prend pas en considération les erreurs relatives. *PRECiSA* [83] est un autre

outil reposant sur l'interprétation abstraite. Il a été spécifiquement conçu pour prendre en considération les tests instables, ce qui représente une tâche difficile à accomplir sans perdre trop de précision. *FPTaylor* [81] est un outil permettant d'évaluer très précisément les bornes d'erreurs absolues et relatives pour des programmes consistant en une suite d'opérations flottantes sans structure de contrôle. Il repose sur l'utilisation d'un calcul de dérivées partielles et sur des expansions de Taylor symboliques pour simplifier le problème de recherche de borne des erreurs numériques, avant d'utiliser des techniques d'optimisations pour le résoudre. Cette approche souffre d'un grand problème de passage à l'échelle par rapport à celle reposant sur l'interprétation abstraite, mais permet généralement de calculer des bornes plus précises.

Daisy [22, 45] est un outil un peu particulier, car son objectif n'est pas, en soi, de calculer des bornes d'erreurs, mais plutôt de générer, à partir d'une spécification d'un programme reposant sur les nombres les réels, une implémentation utilisant l'arithmétique flottante, satisfaisant des contraintes de précisions fournies par l'utilisateur et combinant si possible différentes précisions pour maximiser les performances du programme généré. Pour atteindre cet objectif, l'outil a besoin de calculer des bornes d'erreurs, mais repose également sur des algorithmes génériques pour trouver la meilleure implémentation possible. *Daisy* propose différentes approches pour le calcul des bornes d'erreurs et, en particulier, une analyse dédiée à l'estimation de l'erreur relative et reposant sur des techniques d'optimisation.

Gappa [58] est un outil permettant d'aider à raisonner sur des programmes reposant sur l'arithmétique flottante. Il repose sur l'arithmétique d'intervalles et diverses réécritures pour mener à bien les calculs de bornes d'erreur. Bien que présentant un mode d'analyse automatique, l'outil est conçu pour être utilisé d'une manière interactive et offre ainsi à l'utilisateur la possibilité de fournir des indices pour guider l'analyse. *Gappa* s'adresse donc plutôt à des experts de l'analyse numérique, et a par exemple été utilisé pour prouver l'implémentation de fonctions élémentaires dans *Metalibm* [51], un outil permettant de générer des implémentations des fonctions mathématiques élémentaires qui sont performantes tout en satisfaisant un objectif de précision défini par l'utilisateur de ces fonctions. *Gappa* est également utilisé par *Why3* [33], un outil de vérification déductive, pour mener à bien les raisonnements sur l'arithmétique flottante.

Pour terminer et bien que ce ne soit pas un outil d'analyse de programmes utilisant l'arithmétique flottante, il nous semble quand même important de présenter *Flocq* [11], qui est une bibliothèque écrite en *Coq* [6] proposant une formalisation de l'arithmétique flottante, ainsi que des théorèmes et outils permettant de raisonner très finement sur des calculs flottants. L'objectif de *Flocq* est d'offrir un cadre formel permettant de construire des preuves complexes, et a été utilisé pour prouver des bornes d'erreur très précises pour différents algorithmes, tel que

l'algorithme de somme compensée de Kahan [10].

Plan

La présentation de notre travail est organisée en trois parties.

La première introduit le cadre d'étude dans lequel sont placés nos travaux. Les connaissances nécessaires sur l'arithmétique flottante sont ainsi présentées au chapitre 1. Le chapitre 2 présente, quant à lui, le langage de programmation sur lequel nous avons fondé la construction de nos analyses. Enfin, les chapitres 3 et 4 présentent le cadre de l'interprétation abstraite, d'un point de vue formel dans un premier temps, puis plus concrètement par l'étude des domaines abstraits que nous avons utilisés dans nos travaux.

La deuxième partie représente le cœur de nos contributions théoriques. La première, consistant en la définition d'une forme de collaboration entre les notions d'erreurs absolues et d'erreurs relatives au cours de l'analyse, est ainsi présentée au chapitre 5. Notre deuxième contribution, consistant en différentes propositions permettant d'améliorer les possibilités de collaborations entre formes affines et intervalles, est présentée au chapitre 6. Enfin, notre troisième contribution, consistant en la définition d'une abstraction spécialisée dans l'analyse de l'erreur relative, est présentée au chapitre 7.

La troisième partie est, elle, concentrée à la mise en application de nos contributions théoriques. Ainsi, le chapitre 8 présente notre implémentation, au sein de la plateforme d'analyse *Frama-C* [49], d'un domaine abstrait spécialisé dans l'analyse des erreurs numériques et fondé sur nos contributions. Le chapitre 9 présente, quant à lui, une évaluation expérimentale de ce prototype et sa comparaison vis-à-vis de différents outils de l'état de l'art. Enfin, le chapitre 10 conclut ce manuscrit et présente les différentes réflexions que nous avons menées sur les évolutions et extensions futures de nos travaux.

Première partie

Contexte

Chapitre 1

Arithmétique flottante

Calculer avec un ordinateur nécessite de représenter les nombres sous la seule forme manipulable par ces derniers, à savoir une suite finie de bits, qui *encode* les nombres d'une manière définie. Par exemple, la suite de bits `1001`, dans laquelle le premier bit est nommé *bit de poids fort* et le dernier *bit de poids faible*, encode l'entier naturel $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 9$. Cette suite est forcément finie car la mémoire de nos machines est elle-même finie. Cette limitation pose déjà quelques problèmes lorsque l'on calcule avec des entiers naturels car elle impose l'existence d'un plus grand nombre représentable. Elle est bien plus gênante lorsque l'on cherche à travailler avec des nombres réels. Il n'est en effet pas possible d'encoder la plupart des réels avec une mémoire finie, et une représentation symbolique, manipulant les objets mathématiques comme des symboles abstraits et sans valeurs, se heurte rapidement à d'énormes problèmes de performance. Il devient alors nécessaire de trouver un compromis ne permettant de représenter qu'une partie des réels en échange de performances acceptables. L'arithmétique à virgule flottante, ou plus simplement arithmétique flottante, est l'un de ces compromis, présent dans une écrasante majorité des ordinateurs de notre temps.

L'arithmétique flottante est définie par la norme ANSI/IEEE-754 [2] datant de 1985. Elle a subit une importante révision en 2008 [34] et une plus légère, dernièrement, en 2019. Ce standard définit différents formats, dits arithmétiques, qui spécifie les contraintes qu'un nombre réel soit encodable par un nombre flottant dans nos ordinateurs. Ces derniers comprennent une sous partie finie des réels, deux zéros signés, deux infinis et un ensemble de valeurs spéciales NaNs pour représenter les résultats de calculs non définis mathématiquement, comme $0 \div 0$. Le standard définit également les règles d'arrondi, les opérations sur les formats arithmétiques, les règles de gestion des erreurs et l'encodage par des chaînes de bits des nombres flottants. Ces deux derniers points sont en dehors du cadre de ce manuscrit et ne seront donc que survolés. Un lecteur désireux d'en apprendre davantage peut se renseigner en étudiant la source suivante [65].

Nous présentons l'arithmétique flottante, les formats et les modes d'arrondis en toute généralité dans ce chapitre, bien que la suite de ce manuscrit ne s'intéresse qu'aux formats binaires et au mode d'arrondi au plus proche pair. Il est en effet tout à fait envisageable, bien qu'en dehors du cadre de ce travail, d'étendre les résultats de cette thèse aux formats décimaux ou à d'autres modes d'arrondi. Nous avons choisi ces restrictions car elles permettent de simplifier à la fois la formalisation et le travail d'implémentation, tout en englobant une vaste majorité des programmes utilisant l'arithmétique flottante.

Ce chapitre commence par présenter en section 1.1 les formats arithmétiques introduit par ANSI/IEEE-754 de manière générique, avant de définir en sections 1.2 et 1.3 les différents modes d'arrondi et les opérations arithmétiques. Finalement, la section 1.4 introduit une modélisation standard du fonctionnement de l'arrondi au plus proche pair, avant de terminer sur les définitions d'erreurs absolue et relative.

1.1 Formats arithmétiques et nombres flottants

Pour présenter les nombres flottants, il est nécessaire de définir d'abord ce qu'est un format, car un nombre flottant n'a de sens que vis-à-vis d'un format.

Définition 1.1.1. Un format arithmétique est un triplet $(\beta, p, emax)$, avec $\beta \in \mathbb{N}^*$ la base, $p \in \mathbb{N}$ la précision et $emax \in \mathbb{N}$ l'exposant maximal.

Définition 1.1.2. Un nombre flottant dans un format arithmétique donné est soit un infini, soit un NaN, soit un nombre fini représenté par un triplet (s, m, e) où $s \in \{0, 1\}$ est nommé le signe, $m \in \mathbb{N}$ la mantisse tel que $m < \beta^p$ et $e \in \mathbb{Z}$ l'exposant tel que $1 - emax \leq e \leq emax$. La valeur que représente ce triplet est alors $(-1)^s \times m \times \beta^{e+1-p}$.

Corollaire 1.1.1. Tout nombre flottant a un opposé.

Démonstration. Pour tout triplet (s, m, e) représentant un flottant valide dans un format donné, son opposé est le triplet $(1 - s, m, e)$. La valeur de la mantisse et de l'exposant ne changeant pas, ce nouveau triplet est également valide dans le format. □

Exemple 1.1.1. Dans le format *binary32* de la norme ANSI/IEEE-754, correspondant au type `float` du langage `C` et spécifié par les paramètres $(2, 24, 127)$, le triplet $(0, 352321536, 0)$ représente $352321536 \times 2^{0+1-24} = 42$.

Les contraintes sur la mantisse et l'exposant qu'impose le format permettent de garantir que tous les nombres finis disposent bien d'un encodage machine et d'une représentation sous la forme d'une suite de bits. La norme spécifie comment

encoder des nombres flottants pour des formats ayant une base de deux ou de dix. Cependant, par souci de simplicité, nous ne présentons ici que l'encodage des formats ayant une base de deux, que nous illustrerons à l'aide d'un format artificiel et simpliste. Les paramètres de ce format, nommé ici $\mathbb{F}_m$, sont $(2, 3, 3)$. L'encodage machine de ce format doit permettre de représenter tous les nombres finis de $\mathbb{F}_m$, ainsi que les infinis et les NaNs, tout en minimisant le nombre de bits nécessaires.

Une première chose que l'on peut facilement gérer est le signe, en utilisant un bit pour l'encoder. Ensuite, les exposants sont compris entre $1 - emax = -2$ et $emax = 3$, soit six valeurs possibles. Il suffit donc de trois bits pour toutes les encoder. Les exposants extrêmes, à savoir les valeurs binaires 0_b et 7_b, sont utilisés pour encoder respectivement les deux zéros, ainsi que les NaNs et les infinis. Le signe de l'exposant est, quant à lui, géré en utilisant le biais β^p. Ainsi, l'exposant -2 est encodé par le nombre binaire 1_b, -1 est encodé par 2_b, 0 par 3_b, etc. Enfin, il nous faut déterminer le nombre de bits nécessaires pour représenter toutes les mantisses du format. Comme la précision est $p = 3$, le format dispose de $\beta^p = 8$ mantisses différentes. Il nous suffit donc également de trois bits pour toutes les encoder. Cependant, il y a ici une optimisation à faire afin d'économiser un bit.

Pour expliquer cette optimisation, commençons par représenter les nombres finis positifs du format $\mathbb{F}_m$ pour les trois exposants les plus petits, comme en figure 1.1. Chaque exposant y est représenté sur un axe séparé. Sur chaque axe, les valeurs des mantisses sont ordonnées les unes par rapport aux autres avec un espace entre chacune représentatif des écarts entre les nombres représentés. Les valeurs alignées verticalement et reliées d'un axe à l'autre par un trait en pointillé représentent le même nombre réel. Ainsi, la mantisse 4 de l'axe $e = -2$ est alignée avec la mantisse 2 de l'axe $e = -1$ car les triplets $(0, 4, -2)$ et $(0, 2, -1)$ représentent tous deux le nombre réel 0.25. C'est ce qu'on appelle le phénomène de *cohorte* : un nombre réel peut avoir plusieurs représentations flottantes au sein d'un même format arithmétique. L'existence de ces cohortes ne posent de problèmes que pour l'encodage machine car cela peut rendre complexe l'implémentation électronique de certaines opérations. Il faut donc choisir un représentant unique à chaque cohorte.

Ainsi, le choix fait par le standard ANSI/IEEE-754 est d'utiliser les triplets avec le plus petit exposant possible. Les mantisses qui ne constituent pas le représentant d'une cohorte sont grisées sur la figure 1.1. On remarque tout de suite que, mis à part pour l'exposant $e = -2$, toutes les mantisses des représentants de chaque cohorte sont plus grandes ou égales à quatre. Ces dernières sont illustrées en gras. La norme nomme nombres *normaux* ou *normalisés* les triplets qui sont les représentants de leur cohorte et pour lesquels la mantisse est supérieure ou égale à $\beta^{p-1} = 4$. Les nombres pour lesquelles le représentant de la cohorte à une mantisse plus petite que quatre sont nommés nombres *sous-normaux* ou *dénormalisés*. Ils sont en rouge et en italique sur la figure.

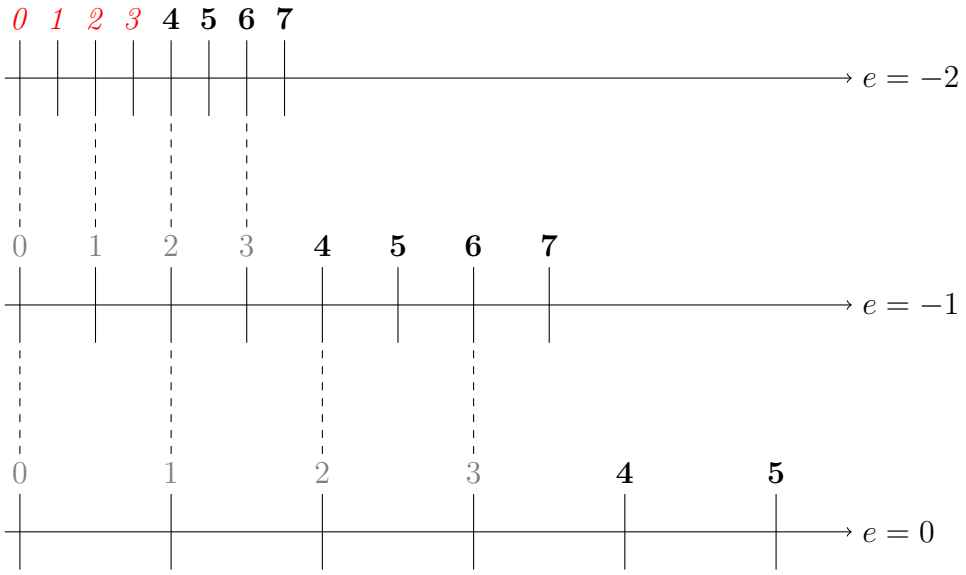


FIGURE 1.1 – Représentation graphique des nombres représentables dans $\mathbb{F}_m$.

Cette séparation entre nombre normaux et sous-normaux est intéressante. En effet, les mantisses des nombres normaux ne peuvent avoir pour valeurs que 4, 5, 6 et 7. Leur encodage sur trois bits a donc forcément un comme bit de poids fort. À l'inverse, les mantisses des nombres sous-normaux ne peuvent avoir pour valeur que 0, 1, 2 et 3. Leur encodage sur trois bits à donc forcément zéro comme bit de poids fort. Ainsi, le bit de poids fort est inutile à partir du moment où il est possible de déterminer si un nombre est normal ou sous-normal d'une autre manière. Or, l'exposant 0_b ne sert pour le moment qu'à représenter les zéros. Les

Exposant	**Mantisse = 0**	**Mantisse ≠ 0**
0_b	zéro	nombres sous-normaux
$1_b, \ldots, 6_b$	nombres normaux	
7_b	$\pm\infty$	NaNs

TABLE 1.1 – Résumé de l'encodage du format $\mathbb{F}_m$.

autres valeurs possibles de la mantisse ne nous servent à rien dans ce cas. La solution adoptée par le format est donc d'encoder tous les nombres sous-normaux en utilisant l'exposant 0_b, et de réserver les autres exposants aux nombres normaux. Ceci permet d'économiser un bit pour représenter la mantisse. Il nous faut ainsi six bits pour encoder tous les nombres du format $\mathbb{F}_m$. La table 1.1 résume les différents

cas possibles de l'encodage machine de ce format.

Exemple 1.1.2. Pour mieux illustrer, prenons pour exemple la suite de bits `110001` et cherchons à retrouver le nombre qu'elle encode. Son bit de poids fort est `1`, c'est donc un nombre négatif. Les trois bits suivants, à savoir `100` $= 4$ encodent l'exposant. Le nombre encodé est donc un nombre normal. En appliquant le biais de 3, nous avons $e = 1$. Finalement, les deux bits restants encodent la mantisse. Comme le nombre est normal, c'est la suite de bits `101` qu'il faut considérer. Nous avons donc $m = 5$. Au final, le nombre encodé par la suite de bits `110001` est $-5 \times 2^{1+1-3} = -2.5$.

Les formats binaires de la norme ANSI/IEEE-754 fonctionne comme le format $\mathbb{F}_m$ (mais avec beaucoup plus de précision, ce qui rend toute tentative d'illustration absolument illisible). Ainsi, un nombre au format *binary32* est encodé sous la forme d'une chaine de 32 bits telle que présentée dans la figure 1.2. Le bit de poids fort encode le signe, les huit bits suivants servent à représenter l'exposant et les vingt-trois restants sont réservés à la mantisse. Comme pour le format $\mathbb{F}_m$, un bit est économisé sur la mantisse car l'exposant permet de déterminer si le nombre encodé est normal ou sous-normal. Le biais utilisé est ici 127, c'est-à-dire que l'exposant 0 est représenté par le nombre binaire $\texttt{127}_\texttt{b}$. Les encodages binaires $\texttt{0}_\texttt{b}$ et $\texttt{255}_\texttt{b}$

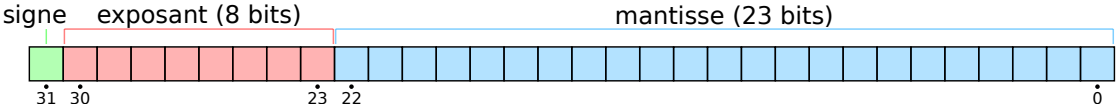


FIGURE 1.2 – Encodage binaire du format *binary32*.

de l'exposant sont, comme dans $\mathbb{F}_m$, réservés pour les valeurs spéciales. Enfin, le nombre zéro dispose de deux encodages dans lesquels tous les bits de la mantisse et de l'exposant sont à zéro, et qui ne diffèrent que par le signe. On a donc un $0+$ et un $0-$. Cependant, ces deux encodages ne sont pas différentiables à l'utilisation, exception faite de quelques cas particuliers, comme $1 \div 0$ qui retourne un infini du même signe que le zéro.

1.2 Arrondis

Un arrondi est une fonction permettant de passer de l'ensemble infini et continu des réels à l'ensemble fini et discontinu des flottants. La norme ANSI/IEEE-754 définit cinq arrondis différents : vers $\pm\infty$, vers zéro, et deux arrondis différents vers le flottant le plus proche. La différence entre ces deux derniers arrondis réside dans la gestion des nombres réels équidistants à deux nombres flottants consécutifs. Ainsi, le premier choisit le nombre flottant le plus éloigné de zéro, alors que le

second arrondit en garantissant le fait que le dernier chiffre de la mantisse soit pair (c'est-à-dire égal à zéro en base deux). Ce dernier arrondi, appelé *arrondi au plus proche pair*, est le plus couramment utilisé. C'est donc celui que nous allons présenter plus en détails.

Comme toutes les fonctions d'arrondi définies par la norme, l'arrondi au plus proche pair, que nous notons $\circ^{\phi}()$ où ϕ est le format de destination, est une fonction totale et monotone, c'est-à-dire que, pour tout x et y appartenant à $\mathbb{R}$, si $x \leq y$ alors on a forcément $\circ^{\phi}(x) \leq \circ^{\phi}(y)$ pour tout format ϕ. De plus, elle est suffisamment spécifiée pour être calculable. Ainsi, une définition standard de cette fonction est la suivante :

$$\circ^{\phi}(x) = x(1 + e_x) + d_x \tag{1.1}$$

avec $|e_x| \leq \epsilon_M$, $|d_x| \leq \delta_M$, $e_x \times d_x = 0$ dans lesquelles ϵ_M et δ_M sont des constantes dépendantes du format utilisé. La première est souvent appelée l'*epsilon machine* et vaut β^{1-p}. La seconde, quant à elle, est appelée le *delta machine* et correspond au plus petit nombre sous-normal positif représentable, à savoir $\beta^{2-emax-p}$. Cette définition gère séparément l'arrondi vers un nombre sous-normal de l'arrondi vers un nombre normal. Ainsi, d_x représente l'erreur introduite par un arrondi vers un nombre sous-normal, alors que e_x représente celle introduite par un arrondi vers un nombre normal. La condition $e_x \times d_x = 0$ permet d'interdire d'arrondir en même temps vers un nombre normal et un nombre sous-normal.

Exemple 1.2.1. Le nombre réel 4.5 n'est pas représentable exactement dans le format $\mathbb{F}_m$. Les deux flottants encadrants 4.5 sont 4, encodé par `010100`, et 5, encodé par `010101`. En utilisant l'arrondi au plus proche pair, 4.5 est arrondi vers 4 car sa mantisse, à savoir `00`, à comme dernier bit `0`, qui est pair. Cet arrondi provoque une erreur, modélisée dans l'équation 1.3 par le symbole e_x car on arrondit vers un nombre normal. La valeur de ce dernier est $4 \div 4.5 - 1 = -0.111\cdots$.

Nous ne présentons pas ici comment calculer e_x et d_x, car cela nous est inutile dans la suite de ce manuscrit. Cependant, cette définition ne tire pas pleinement parti des propriétés des nombres flottants, ce qui nous poserait un problème de précision par la suite. En effet, l'erreur introduite par un arrondi vers un nombre normal, à savoir xe_x, est en fait bornée par la distance entre les deux nombres flottants autour de x. Comme montré par les auteurs de [74], cette distance est égale à $\mathrm{ufp}(x)\,\epsilon_M$ où $\mathrm{ufp}(x)$ est définit de la manière suivante :

$$\mathrm{ufp}(x) = \begin{cases} 0 & \text{si } x = 0 \\ \beta^{\lfloor \log_\beta |x| \rfloor} & \text{si } x \neq 0 \end{cases} \tag{1.2}$$

Exemple 1.2.2. Nous avons en base deux $\mathrm{ufp}(159) = 128$, $\mathrm{ufp}(4.5) = 4$ et $\mathrm{ufp}(-9) = 8$.

Cette notion, appelée l'*unité en première place* (où « unit in the first place » en anglais), revient à calculer le flottant avec le même exposant que $\circ^{\phi}(x)$ mais avec une mantisse à β^{p-1}. Ce nombre est forcément plus petit que x en valeur absolue car, comme dit précédemment, la mantisse d'un nombre normal vaut toujours plus que β^{p-1}. Ainsi, nous utiliserons par la suite la définition suivante pour l'arrondi au plus proche pair :

$$\circ^{\phi}(x) = x + \mathrm{ufp}(x)\, e_x + d_x \tag{1.3}$$

Exemple 1.2.3. La valeur de e_x pour l'arrondi du réel 4.5 dans le format $\mathbb{F}_m$ change avec cette nouvelle définition. On a ainsi :

$$e_x = \frac{\circ^{\phi}(x) - x}{\mathrm{ufp}(x)} = \frac{4 - 4.5}{4} = -0.125$$

1.3 Opérations

La norme introduit un certain nombre d'opérations définies sur les flottants. Certaines sont requises à toute implémentation souhaitant être conforme, alors que d'autres sont seulement recommandées. Sans vouloir être exhaustif, on note tout de même que les opérations arithmétiques élémentaires, les conversions entre formats et les comparaisons sont requises alors que les fonctions trigonométriques et transcendantales ne sont que recommandées.

La norme impose des contraintes de précision sur toutes les opérations requises. Ainsi, toute comparaison doit répondre correctement (ce qui veut dire que la réponse doit être infiniment précise) et les conversions entre formats doivent retourner le flottant représentant correctement — vis-à-vis du mode d'arrondi et dans le format cible — le nombre flottant encodé dans le format source. De plus, pour toute opération arithmétique $\widetilde{\diamond} : \mathbb{F}^n \mapsto \mathbb{F}$, la norme impose que, pour toute entrée $\mathbf{x} \in \mathbb{F}^n$, la propriété suivante soit valide :

$$\widetilde{\diamond}(\mathbf{x}) = \circ^{\phi}(\diamond(\mathbf{x})) \tag{1.4}$$

où $\diamond : \mathbb{R}^n \mapsto \mathbb{R}$ est l'équivalent dans les réels de l'opération $\widetilde{\diamond}$. L'opération $\widetilde{\diamond}$ est alors qualifiée de *correctement arrondi*. Cette propriété peut se voir de la manière suivante. Une opération est dite correctement arrondi si pour toute entrée, le résultat de son calcul est équivalent à faire le calcul dans les réels avant d'arrondir le résultat. Cela garantit que le flottant retourné est bien l'arrondi correct, c'est-à-dire le flottant le plus proche, vis-à-vis du mode d'arrondi, du calcul en réel.

1.4 Modélisation et erreurs

L'arithmétique flottante introduit, à cause des arrondis calculés à chaque étape, une *erreur*, c'est-à-dire un écart entre le résultat fourni et celui que l'on pourrait attendre d'un calcul effectué avec des nombres réels. De manière générale, la mesure de l'erreur entre une valeur *idéale* (ici le calcul réel) et une valeur *approximée* (le calcul flottant) peut être faite de deux manières différentes : soit en regardant l'écart *absolu* entre la valeur idéale et celle approximée, soit en regardant l'écart *relativement* à la valeur idéale. On parlera ainsi d'*erreur absolue* et d'*erreur relative*. Dans la suite de ce manuscrit, la notation $\mathcal{E}_a^\phi$ représentera l'erreur absolue (d'une valeur, d'un calcul, *etc*) dans un format ϕ donné, et la notation $\mathcal{E}_r^\phi$ représentera l'erreur relative dans ce même format. Cependant, par soucis de clarté, on dénotera différemment l'erreur (absolue ou relative) introduite par une simple opération d'arrondi. En effet, cette notion, que l'on nommera *erreur élémentaire*, reviendra suffisamment régulièrement pour nécessiter sa propre notation. Ainsi, $\Gamma_a^\phi(x)$ (respectivement Γ_r^ϕ) dénotera l'erreur absolue (respectivement relative) introduite en arrondissant le réel x vers le format ϕ. Les définitions de ces deux fonctions sont :

$$\Gamma_a^\phi(x) = \circ^\phi(x) - x \tag{1.5}$$

$$\Gamma_r^\phi(x) = \frac{\circ^\phi(x) - x}{x} \tag{1.6}$$

Exemple 1.4.1. En reprenant l'exemple 1.2.1, l'arrondi du réel 4.5 dans le format $\mathbb{F}_m$ provoque une erreur absolue élémentaire $\Gamma_a^{\mathbb{F}_m}(4.5) = 4 - 4.5 = -0.5$ et une erreur relative élémentaire $\Gamma_r^{\mathbb{F}_m}(4.5) = -0.5 \div 4.5 = -1 \div 9 = -0.111\cdots$.

Ces définitions étant désormais posées, nous pouvons maintenant nous intéresser à la propagation et à l'accumulation des erreurs d'arrondi au travers d'un programme. Cependant, il nous faut pour cela commencer par définir un langage de programmation sur lequel travailler. C'est l'objectif du prochain chapitre.

Chapitre 2

Langage d'étude

L'objectif de ce chapitre est de présenter le langage que nous utiliserons dans la suite de ce manuscrit. Ce langage n'a pas pour but de retranscrire entièrement un réel langage de programmation comme le `C`, ce qui serait long et fastidieux. À la place, nous présentons ici un langage simple et minimal, ne contenant que les constructions qui nous intéresserons par la suite. Ainsi, nous écarterons par exemple toutes les questions de gestion mémoire ou d'arithmétique entière, pour nous concentrer sur l'arithmétique flottante et sur les erreurs d'arrondi.

La présentation de notre langage commencera par introduire sa syntaxe, soit les différentes constructions utilisables et comment les écrire. Cette première section sera très classique, notre langage n'étant qu'une variante du langage `While` [89]. Ensuite, pour pouvoir analyser la propagation des erreurs d'arrondi et apporter des preuves que ce que nous faisons fait sens, il nous est nécessaire de donner une formalisation mathématique du comportement de notre langage, à travers une *sémantique*. Pour nous permettre de définir une analyse évaluant les erreurs d'arrondis accumulées au cours de l'exécution d'un programme, il est nécessaire que notre sémantique formalise à la fois le comportement *idéal*, utilisant l'arithmétique réelle, de notre langage, mais également son comportement *approximé*, utilisant l'arithmétique flottante, et, plus important encore, le comportant des erreurs absolues et relatives. C'est ainsi une sémantique en quatre parties que nous introduirons au cours de ce chapitre. Les trois premières sémantiques sont complètement standards. La dernière, la sémantique de l'erreur relative est, quant à elle, moins souvent utilisée.

2.1 Syntaxe

Comme dit précédemment, notre langage, nommé MiniFloat par la suite, est un classique langage `While` ne comportant ni gestion de la mémoire, ni arithmé-

tique entière, mais étendu avec de l'arithmétique flottante. Sa syntaxe est présentée sous forme BNF en figure 2.1. Un programme dans ce langage est une suite d'*instructions*, chainées à l'aide de la construction de *séquence*. Les autres instructions du langage sont l'*affectation*, permettant d'affecter la valeur d'une expression à une variable dans un format donné, l'*identité*, c'est-à-dire une instruction n'ayant aucun effet, la *boucle*, répétant une instruction tant qu'une *condition* est vérifiée, et le *test*, exécutant une de ses deux branches en fonction d'une condition. Les expres-

Variables :	$x \in \mathbb{V}$	
Formats :	$\phi \in \mathcal{F}$	
Opérateurs :	$\blacklozenge \in \{+, -, \times, \div\}$	
Expressions :	$e ::= n \in \mathbb{R}$	*nombres*
	$\mid x \in \mathbb{V}$	*variables*
	$\mid e \blacklozenge e \mid -e \mid \mathbf{sqrt}(e)$	*arithmétique*
	$\mid (\phi)\,(e)$	*annotation de format*
Conditions :	$c ::= (\phi)\,(e \leq e) \mid c \wedge c \mid c \vee c \mid \neg c$	
Instructions :	$s ::= x = (\phi)\,(e)\ ;$	*affectation*
	$\mid s\ s$	*séquence*
	$\mid$ **skip ;**	*identité*
	$\mid$ **while** c **do** s **done**	*boucle*
	$\mid$ **if** c **then** s **else** s **end**	*test*

FIGURE 2.1 – Syntaxe du langage MINIFLOAT.

sions du langage comportent les opérations arithmétiques classiques : additions, soustractions, multiplications, divisions et négations, auxquelles nous ajoutons la racine carrée, cette dernière faisant partie des opérations arithmétiques imposées comme correctement arrondies par la norme ANSI/IEEE-754. En termes de notations, les opérateurs en gras feront références aux constructions du langage, là où ceux dans une police standard feront références aux opérateurs mathématiques. Les expressions comportent également constantes réelles et variables. Ces dernières

appartiennent toutes à l'ensemble infini dénombrable $\mathbb{V}$. Enfin, l'annotation de format permet de spécifier le format flottant à utiliser pour évaluer une expression. Pour terminer, les conditions de notre langage comportent l'opérateur inférieur ou égal, annoté par le format à utiliser pour l'évaluation de ses sous-expressions, ainsi que les opérateurs logiques usuels. Le langage dispose ainsi du nombre minimal de constructions permettant de reconstruire tous les opérateurs usuels de comparaison. Ainsi, par abus de langage, nous nous réservons le droit d'utiliser par la suite les notations standards de tous les opérateurs usuels en lieu et place de leurs équivalents avec les constructions de notre langage. Les trois constructions que sont l'affectation, la comparaison et l'annotation de format permettent d'imposer une forme de typage à notre langage.

2.2 Sémantiques

L'exécution d'un programme peut se formaliser comme une suite de modifications opérées par les instructions sur un *état machine*. Pour nos ordinateurs, cet état machine contient les valeurs associées à chaque adresse de la mémoire. Pour notre langage minimal, l'état machine contient plus simplement les valeurs associées à chaque variable. Chaque étape d'exécution, chaque instruction donc, va utiliser les informations contenues dans l'état machine courant pour opérer et produire un nouvel état en altérant le précédent. Définir une sémantique pour notre langage revient donc à définir formellement comment chaque construction du langage interagit avec un état machine.

Il existe trois principales manières de définir une sémantique. La première, la *sémantique opérationnelle* [68], décrit formellement l'exécution d'un programme comme un système de transitions entre états. Ce formalisme s'intéresse particulièrement à l'exécution d'un programme, ce qui est très utile pour prouver la correction d'une transformation de programme, mais qui ne nous aiderait pas à prouver que l'analyse que nous développerons par la suite, basée sur la théorie de l'interprétation abstraite, que nous présenterons au chapitre 3, offre de bonnes garanties. La deuxième, appelée *sémantique axiomatique* [44], décrit indirectement le comportement des instructions en spécifiant, pour chacune d'entre elle, les hypothèses qui doivent être valides avant son exécution et les propriétés qui découlent de cette exécution. Cette description correspond à utiliser la logique de Hoare. Cependant, comme pour la sémantique opérationnelle, la sémantique axiomatique est rarement un outil pratique pour prouver la correction d'une analyse par interprétation abstraite, notre travail ne faisant pas exception. Enfin, la *sémantique dénotationnelle* décrit le comportement des instructions en utilisant des objets mathématiques, appelés *dénotations*, devant respectés certaines propriétés présentées dans les travaux fondateurs de Christopher Strachey et Dana Scott [77]. C'est ce

dernier formalisme qui est le plus souvent utilisé dans le cadre de l'interprétation abstraite, et c'est donc ce dernier formalisme que nous utiliserons.

La présentation de notre sémantique est séparée en plusieurs parties. Tout d'abord, nous présenterons la formalisation de l'état machine et du mécanisme de gestion des calculs n'ayant pas de sens mathématique, comme $0 \div 0$ par exemple. Ensuite, nous présenterons la sémantique dédiée aux expressions du langage, que nous séparons en quatre parties, chacune représentant un des comportements que nous souhaitons formaliser. Cette séparation a pour but de simplifier la présentation en rendant la compréhension de chacun de ces comportements plus explicite. Enfin, nous terminerons en présentant l'interprétation des conditions et des instructions. Nous présenterons au fur et à mesure de l'exposé les éléments théoriques nécessaires pour justifier de la correction de notre sémantique.

2.2.1 État machine

Comme nous le disions en début de section, nos états machine contiennent les valeurs associées à chaque variable. Mathématiquement, on peut formaliser cela comme une fonction Σ, dont le domaine est l'ensemble des variables $\mathbb{V}$ et le co-domaine est l'ensemble des valeurs associées aux variables, qu'il nous faudra définir. Cependant, il est tout à fait possible qu'une variable x ne soit associée à aucune valeur, si par exemple aucune instruction d'affectation n'a été exécutée pour x. Les fonctions utilisées pour formaliser les états machine doivent donc également être *partielles*, c'est-à-dire n'être définies que pour une sous partie de leur domaine $\mathbb{V}$. Nous utiliserons la notation $A \rightharpoonup B$ pour dénoter la fonction partielle de domaine A et de co-domaine B.

Afin de pouvoir correctement rendre compte des quatre comportements que nous souhaitons formaliser, il nous faut associer aux variables différentes informations. En effet, le comportement exact nécessite que les variables soient associées à des réels. De manière similaire, le comportement flottant nécessite des associations flottantes. Enfin, les comportements des erreurs nécessitent de pouvoir associer une erreur absolue et une erreur relative à toutes les variables, mais également d'avoir accès à leurs valeurs réelles et flottantes. Ainsi donc, l'état machine doit contenir les quatre informations nécessaires aux quatre parties de notre sémantique. La représentation formelle que nous utiliserons pour les états machine est donc la suivante :

$$\Sigma : \mathbb{V} \rightharpoonup \mathbb{R} \times \biguplus_{\phi \in \mathcal{F}} \mathbb{F}^{\phi} \times \mathbb{R} \times \mathbb{R} \tag{2.1}$$

où $\uplus$ représente la somme disjointe. Les éléments du quadruplet du co-domaine représentent, de gauche à droite, la valeur en réel, la valeur en flottant, l'erreur absolue et l'erreur relative. Il est important de noter que chaque variable peut, de

part l'utilisation de la somme disjointe, disposer d'une valeur flottante dans un format différent parmi les formats disponibles.

Cette définition de l'état machine contient toutes les informations dont nous aurons besoin. Cependant, il nous faut en définir des *projections*, c'est-à-dire une manière de n'accéder qu'à une sous partie des informations stockées. Ainsi, nous noterons $\rho_r : \mathbb{V} \rightharpoonup \mathbb{R}$ la projection de l'état machine ρ permettant d'accéder exclusivement à la valeur exacte, utilisant les réels, de chaque variable. De la même manière, nous noterons $\rho_f : \mathbb{V} \rightharpoonup \biguplus_{\phi \in \mathcal{F}} \mathbb{F}^\phi$, $\rho_{e_a} : \mathbb{V} \rightharpoonup \mathbb{R}$ et $\rho_{e_r} : \mathbb{V} \rightharpoonup \mathbb{R}$ les projections permettant d'accéder respectivement aux valeurs flottantes, aux erreurs absolues et aux erreurs relatives.

Pour terminer, il nous reste à montrer que ce formalisme respecte bien les propriétés nécessaires à toute dénotation, c'est-à-dire que nos états machine forment bien un *domaine* au sens de Strachey et Scott. Avant d'introduire la définition d'un domaine, il nous faut commencer par introduire les différentes notions qui nous seront nécessaires. Nous ne présentons ici que le strict minimum nécessaires à nos besoins. Un lecteur souhaitant en apprendre plus peut se renseigner en étudiant les sources suivantes [3].

Définition 2.2.1. Soit un ensemble quelconque $\mathbb{K}$ et une relation $\leq$ sur $\mathbb{K}$. Cette relation forme un *ordre partiel* sur $\mathbb{K}$ si et seulement si elle respecte les propriétés suivantes :

- $\leq$ est réflexive, c'est-à-dire que pour tout $x \in \mathbb{K}$, nous avons $x \leq x$;
- $\leq$ est transitive, c'est-à-dire que pour tout $x, y, z \in \mathbb{K}$, si $x \leq y$ et $y \leq z$ alors $x \leq z$;
- $\leq$ est antisymétrique, c'est-à-dire que pour tout $x, y \in \mathbb{K}$, si $x \leq y$ et $y \leq x$ alors c'est que $x = y$

Une illustration classique de cette notion est de considérer l'ensemble des sous parties de $\{a, b, c\}$ muni de la relation d'inclusion. Il est aisé de voir qu'il existe un ordre entre les éléments, par exemple $\{a, b\}$ ou $\{b, c\}$ sont tous deux plus petits que $\{a, b, c\}$. Cet ordre n'est cependant pas total. Par exemple, ni $\{a, b\}$ ni $\{b, c\}$ n'est plus petit que l'autre.

Définition 2.2.2. Soit $(\mathbb{K}, \leq)$ un ordre partiel, et $\mathbb{P}$ une sous partie de $\mathbb{K}$.

Un élément $x \in \mathbb{K}$ est un *majorant* de $\mathbb{P}$ s'il est plus grand que tous les éléments de $\mathbb{P}$. Le plus petit de tous les majorants de $\mathbb{P}$, s'il existe, est son *supremum*. Il sera noté $\sup(\mathbb{P})$ ou $\sqcup\mathbb{P}$ en fonction du contexte.

Un élément $x \in \mathbb{K}$ est un *minorant* de $\mathbb{P}$ s'il est plus petit que tous les éléments de $\mathbb{P}$. Le plus grand de tous les minorants de $\mathbb{P}$, s'il existe, est son *infimum*. Il sera noté $\inf(\mathbb{P})$ ou $\sqcap\mathbb{P}$ en fonction du contexte.

Pour reprendre notre exemple précédent, la sous partie $\{\{a\},\{b\}\}$ dispose de deux majorants, à savoir $\{a,b\}$ et $\{a,b,c\}$. Parmi ces deux, $\{a,b\}$ est le supremum car il est le plus petit.

Muni de ces notions, nous pouvons maintenant présenter la définition d'un domaine au sens de Strachey et Scott :

Définition 2.2.3. Un ensemble quelconque $\mathbb{K}$ muni d'une relation $\leq$ est un *domaine de Scott* si et seulement si :

- la relation $\leq$ forme un *ordre partiel* sur $\mathbb{K}$;
- $\mathbb{K}$ dispose d'un plus petit élément vis-à-vis de $\leq$;
- toute sous partie totalement ordonnée, nommée *chaîne* par la suite, de $\mathbb{K}$ dispose d'un supremum.

Cependant, pour éviter toute confusion avec la notion de domaine abstrait que nous verrons au chapitre 3, nous utiliserons exclusivement, dans la suite de ce manuscrit, le nom d'*ordre partiel complet et borné*, ou son acronyme *cpo* en anglais.

Ainsi donc, pour que nos états machine forment un cpo, et soient donc utilisables pour définir notre sémantique, il nous faut leur trouver une relation respectant toutes ces propriétés. Nous utilisons ici la relation $\leq$ suivante :

$$\rho_1 \leq_\Sigma \rho_2 \iff \mathrm{dom}\,(\rho_1) \subseteq \mathrm{dom}\,(\rho_2) \wedge \forall x \in \mathrm{dom}\,(\rho_1)\,, \rho_1(x) = \rho_2(x) \tag{2.2}$$

avec $\rho_1, \rho_2 \in \Sigma$ et $\mathrm{dom}\,(\rho) \subseteq \mathbb{V}$ la sous partie des variables pour lesquelles ρ disposent d'une affectation. Ainsi, un état machine est plus grand qu'un autre s'il contient des informations sur plus de variables. De la même manière, deux états machine disposant d'informations différentes sur les mêmes variables donnent la même quantité d'informations, et sont donc incomparables. Démontrons maintenant que $(\Sigma, \leq_\Sigma)$ forme bien un cpo.

Démonstration. Commençons par montrer que $(\Sigma, \leq_\Sigma)$ forme un ordre partiel. La réflexivité et la transitivité de $\leq_\Sigma$ découlent directement de celles de l'inclusion et de l'égalité. L'antisymétrie, quant à elle, découle de l'antisymétrie de l'inclusion et de la symétrie de l'égalité.

Démontrons ensuite que Σ dispose d'un plus petit élément. L'état machine n'associant aucune valeur à aucune variable, que nous noterons $\bot$, est le plus petit élément de Σ vis-à-vis de $\leq_\Sigma$. Il est en effet trivial de montrer que, comme $\mathrm{dom}\,(\bot) = \{\}$, pour tout état machine ρ, nous avons bien $\bot \leq_\Sigma \rho$.

Enfin, démontrons que toute chaîne de Σ dispose d'un supremum. Soit $\{\rho_i\}_{i\in\mathbb{N}}$ une suite, potentiellement infinie, d'états machine totalement ordonnée. Montrons

que l'état machine ρ_{sup} définit ci-après est le supremum de cette chaîne :

$$\begin{cases} \text{dom}\left(\rho_{\text{sup}}\right) = \bigcup\limits_{i \in \mathbb{N}} \text{dom}\left(\rho_i\right) \\ \forall x \in \text{dom}\left(\rho_{\text{sup}}\right), \rho_{\text{sup}}(x) = \rho_i(x) \text{ avec } i \in \mathbb{N} \text{ tel que} \\ \qquad x \in \text{dom}\left(\rho_i\right) \wedge \forall j \leq i, x \notin \text{dom}\left(\rho_j\right) \end{cases}$$

Commençons par montrer que ρ_{sup} est un majorant. Il découle trivialement des propriétés de l'union ensembliste que nous avons bien $\text{dom}\left(\rho_i\right) \subseteq \text{dom}\left(\rho_{\text{sup}}\right)$ pour tout i. De plus, l'état machine ρ_{sup} associe à toute variable x la même valeur que celle associée par tous les éléments de la chaîne associant une valeur à x. Ceci découle trivialement de la définition de $\leq_\Sigma$. Ainsi, ρ_{sup} est bien un majorant de cette chaîne.

Montrons maintenant qu'il n'existe pas d'autre élément ρ' de Σ qui soit à la fois un majorant de cette chaîne et plus petit que ρ_{sup}. Procédons par l'absurde et supposons qu'il existe un ρ' différent de ρ_{sup} qui soit le supremum de cette chaîne. L'état machine ρ' doit donc définir une affectation pour, au minimum, toutes les variables affectées par au moins un des états de la chaîne. Ainsi donc, il est évident que $\bigcup_{i \in \mathbb{N}} \text{dom}\left(\rho_i\right) \subseteq \text{dom}\left(\rho'\right)$. Cependant, comme ρ_{sup} est un majorant, ρ' est nécessairement plus petit que ρ_{sup}. Ainsi donc, $\text{dom}\left(\rho'\right) \subseteq \text{dom}\left(\rho_{\text{sup}}\right) = \bigcup_{i \in \mathbb{N}} \text{dom}\left(\rho_i\right)$. Il découle donc que $\text{dom}\left(\rho'\right) = \text{dom}\left(\rho_{\text{sup}}\right)$. Finalement, par définition de $\leq_\Sigma$, les affectations de ρ' et ρ_{sup} sont les mêmes pour toutes les variables affectées, et donc $\rho' = \rho_{\text{sup}}$. Ainsi donc, ρ_{sup} est bien le supremum de cette chaîne. □

Maintenant que nous avons montré que nos états machine disposent des propriétés nécessaires, nous pouvons revenir à la présentation de notre sémantique. Il nous faut cependant commencer par faire un autre détour, et exposer comment nous allons gérer les constructions n'ayant pas de sens.

2.2.2 Gestion des exceptions

La plupart des langages contiennent des constructions qui n'ont pas de sens, et le nôtre ne fait pas exception. Par exemple, les expressions $\textbf{sqrt}(-5)$ ou $0 \div 0$ n'ont aucun sens mathématique et leur exécution ne peut donc pas en avoir non plus. Exécuter ce genre de constructions ne peut produire aucun état si ce n'est un état d'erreur, traduisant qu'un problème a été rencontré. Pour éviter toute confusion avec les erreurs de l'arithmétique flottante, nous qualifierons ces états d'*exceptions*. Comme nous ne nous intéresserons pas à la gestion de ces exceptions, nous avons choisi de n'en utiliser qu'une seule, que nous noterons $\omega \in \Omega$. Il est important de noter que nous considérerons les NaNs et les infinis de l'arithmétique flottante comme des états d'exception afin de simplifier les définitions à venir. De toute façon, il est quelque peu difficile d'imaginer parler de l'erreur (absolue ou

relative) de ces objets. Les opérateurs arithmétiques (réels ou flottants) peuvent donc produire des exceptions : ce sont ainsi des fonctions de domaine $\mathbb{K}^n$, où $\mathbb{K}$ représente l'ensemble de nombres sur lequel s'applique l'opérateur, et de co-domaine $\mathbb{K} \uplus \Omega$.

Finalement, pour éviter de devoir gérer les exceptions dans les définitions à venir des sémantiques, nous introduisons la notation op_ω pour décrire une version modifiée de l'opérateur *op* capable de gérer les exceptions sur ses entrées. Ainsi, pour tout opérateur $op : \mathbb{K}^n \mapsto \mathbb{K} \uplus \Omega$, on définit l'opérateur $\mathrm{op}_\omega : (\mathbb{K} \uplus \Omega)^n \mapsto \mathbb{K} \uplus \Omega$ comme suit :

$$\mathrm{op}_\omega(x_1, \ldots, x_n) \triangleq \begin{cases} \omega & \text{si } \exists i \in \{1, \ldots, n\}, x_i = \omega \\ op(x_1, \ldots, x_n) & \text{sinon} \end{cases} \tag{2.3}$$

Cette version modifiée gérant les exceptions est intéressante car elle permet de conserver certaines propriétés de l'opérateur modifié qui nous seront nécessaires par la suite. Ainsi, pour toute opération binaire $\diamond$ définie sur un ensemble S, les trois propriétés suivantes sont valides.

Lemme 2.2.1. Si $\diamond$ est commutative alors $\diamond_\omega$ est commutative sur $S \uplus \Omega$.

Lemme 2.2.2. Si $\diamond$ est associative alors $\diamond_\omega$ est associative sur $S \uplus \Omega$.

Lemme 2.2.3. Si $\diamond$ est distributive par rapport à une autre opération $\circ$, alors $\diamond_\omega$ est distributive sur $S \uplus \Omega$ par rapport à $\circ_\omega$.

Démonstration. Nous ne présentons la preuve que pour la commutativité, car les preuves pour l'associativité et la distributivité utilisent exactement le même raisonnement. On cherche à prouver que pour tout $a, b \in S$, si la propriété $a \diamond b = b \diamond a$ est valide alors $a' \diamond_\omega b' = b' \diamond_\omega a'$ est valide également, avec $a' \in \{a, \omega\}$ et $b' \in \{b, \omega\}$. Faisons une analyse de cas :

- Si $a' = \omega$ ou $b' = \omega$ alors $a' \diamond_\omega b' = b' \diamond_\omega a' = \omega$;
- Sinon, par la définition 2.3, nous avons $a' \diamond_\omega b' = a \diamond b$ et $b' \diamond_\omega a' = b \diamond a$. Comme $\diamond$ est commutative, la propriété est vérifiée.

□

Pour terminer, il nous reste à montrer que l'ajout d'une gestion des exceptions est compatible avec la notion de cpo. Nous présentons donc deux manières de construire des cpos incorporant les exceptions. La première s'applique à n'importe quel ensemble, ceux nous intéressant ici étant naturellement $\mathbb{R}$ et $\mathbb{F}^\phi$. La seconde s'applique à un cpo, et nous permet ainsi de montrer que l'ajout des exceptions aux états machine ne nous fait pas perdre leurs propriétés.

Théorème 2.2.1. Pour tout ensemble $\mathbb{K}$, l'ensemble $\mathbb{K} \uplus \Omega$ muni de la relation $x \leq_\omega y \iff x = \omega \vee x = y$ est un cpo, appelé *cpo plat*.

Démonstration. Les propriétés d'ordre partiel sont démontrés trivialement. Le plus petit élément est ω comme il est, par définition, plus petit que tout élément de $\mathbb{K}$. Les chaînes ne peuvent contenir qu'au maximum deux éléments. L'un deux est ω, et l'autre est forcément supremum. □

Théorème 2.2.2. Soit un domaine $(\mathbb{K}, \leq)$. L'ensemble $\mathbb{K} \uplus \Omega$ muni de la relation $x \leq_\omega y \iff x = \omega \vee x \leq y$ forme un cpo.

Démonstration. La preuve découle trivialement du fait que l'ajout de ω ne fait qu'ajouter un nouveau plus petit élément et, donc, que l'ordre partiel précédent et toutes ces propriétés restent inchangés. □

Le premier cas n'est finalement qu'un cas particulier du deuxième dans lequel tous les éléments de $\mathbb{K}$ sont incomparables. Avec ces deux théorèmes, nous avons la certitude que toutes les dénotations que nous utiliserons respectent bien les propriétés attendues. Il est désormais temps de présenter notre sémantique.

2.2.3 Sémantique exacte des expressions

Commençons par définir la plus simple des sémantiques dont nous aurons besoin, la sémantique exacte, formalisant l'exécution des expressions en utilisant des nombres réels. La notation $\mathbb{R}[\![e]\!]$ représentera la dénotation de e, c'est-à-dire la fonction qui, à partir d'un état machine, produit un réel ou une exception :

$$\mathbb{R}[\![e]\!] : \Sigma \mapsto \mathbb{R} \uplus \Omega \tag{2.4}$$

Sa définition, présentée en figure 2.2, revient à interpréter les expressions *via* les opérateurs réels correspondants. Ainsi, les constantes réelles sont interprétées telles quelles, les variables prennent leurs valeurs dans l'environnement et les opérateurs arithmétiques sont définis récursivement, en appliquant les opérateurs réels correspondants sur le résultat de l'interprétation de leurs opérandes. Pour finir, la sémantique exacte ignore les annotations de format. Ces dernières n'auront de sens que pour l'arithmétique flottante.

Exemple 2.2.1. L'interprétation de l'expression $\mathbf{sqrt}(3 + x)$ dans un environnement ρ où x est associé à 6 se calcule comme suit :

$$\begin{aligned}\mathbb{R}[\![\mathbf{sqrt}(3+x)]\!](\rho) &= \sqrt[\omega]{\mathbb{R}[\![3+x]\!](\rho)} = \sqrt[\omega]{\mathbb{R}[\![3]\!](\rho) +_\omega \mathbb{R}[\![x]\!](\rho)} \\ &= \sqrt[\omega]{3 +_\omega \rho_r(x)} = \sqrt[\omega]{3 +_\omega 6} = \sqrt[\omega]{9} = 3\end{aligned}$$

$$
\begin{aligned}
\mathbb{R}[\![n]\!](\rho) &\triangleq n \\
\mathbb{R}[\![x]\!](\rho) &\triangleq \rho_r(x) \\
\mathbb{R}[\![e_1 \blacklozenge e_2]\!](\rho) &\triangleq \mathbb{R}[\![e_1]\!](\rho) \diamond_\omega \mathbb{R}[\![e_2]\!](\rho) \\
\mathbb{R}[\![-e]\!](\rho) &\triangleq -_\omega \mathbb{R}[\![e]\!](\rho) \\
\mathbb{R}[\![\mathbf{sqrt}(e)]\!](\rho) &\triangleq \sqrt[\omega]{\mathbb{R}[\![e]\!](\rho)} \\
\mathbb{R}[\![(\phi)\,(e)]\!](\rho) &\triangleq \mathbb{R}[\![e]\!](\rho)
\end{aligned}
$$

FIGURE 2.2 – Sémantique exacte des expressions.

2.2.4 Sémantique approximée des expressions

La sémantique approximée, formalisant l'exécution des expressions en utilisant l'arithmétique flottante, est paramétrée pour permettre de spécifier le format à utiliser. Ainsi nous noterons $\mathbb{F}^\phi[\![e]\!]$ la dénotation de e dans le format $\phi \in \mathcal{F}$. Comme pour la sémantique exacte, cette dénotation est une fonction produisant des valeurs à partir d'un état machine. Cependant ici, la dénotation devra également produire des flottants dans le format considéré :

$$\mathbb{F}^\phi[\![e]\!] : \Sigma \mapsto \mathbb{F}^\phi \uplus \Omega \tag{2.5}$$

Sa définition, présentée en figure 2.3, revient à interpréter les expressions avec l'arithmétique flottante. Les constantes réelles sont arrondies vers leurs représentants flottants. Les variables prennent leurs valeurs dans l'environnement. Cependant, ces valeurs peuvent ne pas être dans le bon format. Il est donc nécessaire de les arrondir. Les opérateurs arithmétiques sont définies récursivement sur leurs opérandes. Comme tous les opérateurs du langage sont *correctement arrondis*, nous avons choisi d'utiliser directement les propriétés présentées dans la Section 1.4 dans la définition de cette sémantique. La négation ne nécessite pas d'arrondi, contrairement aux autres opérateurs, car, comme démontré par le corollaire 1.1.1, tout nombre flottant à un opposé. Enfin, l'annotation de format interprète son opérande avec le nouveau format avant d'arrondir le résultat vers l'ancien.

Exemple 2.2.2. Reprenons l'exemple 2.2.1 et cherchons à calculer l'expression $\mathbf{sqrt}(3 + x)$ dans le format $\phi = \mathbb{F}_m$ et dans un environnement où x est associé à 6 dans le format $\mathbb{F}_m$, nombre représentable par le triplet $(0, 6, 2)$, avec dans l'ordre, et pour rappel, le signe (positif ici), la mantisse et l'exposant. Ce nombre diposant d'une représentation, il n'introduit donc pas d'arrondi. L'interprétation de cette

$$
\begin{aligned}
\mathbb{F}^\phi[\![n]\!](\rho) &\triangleq \circ^\phi(n) \\
\mathbb{F}^\phi[\![x]\!](\rho) &\triangleq \circ^\phi(\rho_f(x)) \\
\mathbb{F}^\phi[\![e_1 \blacklozenge e_2]\!](\rho) &\triangleq \circ^\phi_\omega\Big(\mathbb{F}^\phi[\![e_1]\!](\rho) \diamond_\omega \mathbb{F}^\phi[\![e_2]\!](\rho)\Big) \\
\mathbb{F}^\phi[\![-e]\!](\rho) &\triangleq -_\omega \mathbb{F}^\phi[\![e]\!](\rho) \\
\mathbb{F}^\phi[\![\mathbf{sqrt}(e)]\!](\rho) &\triangleq \circ^\phi_\omega\Big(\sqrt[\omega]{\mathbb{F}^\phi[\![e]\!](\rho)}\Big) \\
\mathbb{F}^\phi[\![(\phi_1)\,(e)]\!](\rho) &\triangleq \circ^\phi_\omega\Big(\mathbb{F}^{\phi_1}[\![e]\!](\rho)\Big)
\end{aligned}
$$

FIGURE 2.3 – Sémantique approximée des expressions.

expression se déroule comme suit :

$$
\begin{aligned}
\mathbb{F}^\phi[\![\mathbf{sqrt}(3+x)]\!](\rho) &= \circ^\phi_\omega\Big(\sqrt[\omega]{\mathbb{F}^\phi[\![3+x]\!](\rho)}\Big) && \text{(a)} \\
&= \circ^\phi_\omega\Big(\sqrt[\omega]{\circ^\phi_\omega(\mathbb{F}^\phi[\![3]\!](\rho) +_\omega \mathbb{F}^\phi[\![x]\!](\rho))}\Big) && \text{(b)} \\
&= \circ^\phi_\omega\Big(\sqrt[\omega]{\circ^\phi_\omega(\circ^\phi(3) +_\omega \circ^\phi(\rho_f(x)))}\Big) && \text{(c)} \\
&= \circ^\phi_\omega\Big(\sqrt[\omega]{\circ^\phi_\omega(3 +_\omega 6)}\Big) = \circ^\phi_\omega\Big(\sqrt[\omega]{8}\Big) = 3 && \text{(d)}
\end{aligned}
$$

Les trois premières étapes de ce calcul ne sont que des applications successives de la définition de la sémantique approximée. Ensuite, au passage entre les étapes (c) et (d), l'arrondi de 3 et de la valeur associée à x, à savoir 6, n'introduisent pas d'erreur, car 3 et 6 sont tous deux représentables dans le format $\mathbb{F}_m$, respectivement par $(0, 6, 1)$ et $(0, 6, 2)$. Cependant, l'addition de 3 et 6 n'est, quant à elle, pas représentable. Le résultat réel, à savoir 9, est arrondi vers 8. En effet, 9 est exactement à mi-chemin entre 8 et 10, les deux nombres flottants du format $\mathbb{F}_m$ les plus proches de 9. Cependant, 8 est représenté par le triplet $(0, 4, 3)$, là ou 10 est représenté par $(0, 5, 3)$. C'est donc 8 qui est retenu, car sa mantisse est paire. Finalement, la racine carrée de 8 est arrondi vers 3. On retrouve le bon résultat bien que des erreurs d'arrondi aient été commises en chemin.

2.2.5 Sémantiques d'erreur absolue et relative des expressions

Les sémantiques d'erreur formalisent l'accumulation des erreurs d'arrondi au travers de l'exécution d'une expression en les exprimant respectivement comme des

erreurs absolues et des erreurs relatives. Comme pour la sémantique approximée, il est nécessaire de savoir dans quel format le calcul est effectué pour pouvoir parler d'erreur. Ainsi, nous noterons $\mathcal{E}_a^\phi[\![e]\!]$ et $\mathcal{E}_r^\phi[\![e]\!]$ les dénotations de e permettant de calculer respectivement l'erreur absolue et l'erreur relative accumulée lors de l'exécution de e dans le format ϕ. Ces dénotations produisent un réel, représentant l'erreur de e dans un état machine donné :

$$\mathcal{E}_a^\phi[\![e]\!] : \Sigma \mapsto \mathbb{R} \uplus \Omega \quad \text{et} \quad \mathcal{E}_r^\phi[\![e]\!] : \Sigma \mapsto \mathbb{R} \uplus \Omega \tag{2.6}$$

La définition la plus simple que l'on puisse donner à la sémantique de l'erreur absolue est la suivante. Pour toute expression e, l'erreur absolue commise en l'exécutant dans le format ϕ est la différence entre son interprétation utilisant la sémantique approximée dans le format ϕ et celle utilisant la sémantique exacte :

$$\mathcal{E}_a^\phi[\![e]\!](\rho) \triangleq \mathbb{F}^\phi[\![e]\!](\rho) -_\omega \mathbb{R}[\![e]\!](\rho) \tag{2.7}$$

De la même manière, la définition la plus simple pour la sémantique de l'erreur relative est la suivante. Pour toute expression e, l'erreur relative commise en l'exécutant dans le format ϕ est la rapport entre l'erreur absolue commise en interprétant e dans ϕ et son interprétation utilisant la sémantique exacte, ou encore le rapport entre l'interprétation dans la sémantique approximée et celle dans la sémantique exacte décalée de un :

$$\mathcal{E}_r^\phi[\![e]\!](\rho) \triangleq \frac{\mathbb{F}^\phi[\![e]\!](\rho) -_\omega \mathbb{R}[\![e]\!](\rho)}{\mathbb{R}[\![e]\!](\rho)}{}_\omega = \frac{\mathbb{F}^\phi[\![e]\!](\rho)}{\mathbb{R}[\![e]\!](\rho)}{}_\omega -_\omega 1 \tag{2.8}$$

Il est important de noter que si l'interprétation exacte de e retourne 0, alors la sémantique de l'erreur relative lèvera une exception. Ce comportement est celui attendu, l'erreur relative n'étant tout simplement pas définie pour 0.

Ces définitions ne nous avancent cependant pas beaucoup, car elles ne nous permettent pas de mieux comprendre ce qu'il se passe concrètement lorsque l'on calcule une expression. Pour obtenir une meilleure intuition, il est nécessaire d'étudier chaque construction indépendamment, en particulier pour les opérateurs arithmétiques, afin de raffiner ces premières définitions. Ces raffinements seront tous résumés à la figure 2.4 présentée en fin de section.

Commençons par les constructions les plus simples, à savoir les constantes et les variables. En remplaçant dans les équations 2.7 et 2.8 les sémantiques exactes et approximées des constantes par leurs définitions, on remarque que l'on retrouve les définitions des *erreurs absolues et relatives élémentaires* présentées à la section 1.4. Ainsi donc :

$$\mathcal{E}_a^\phi[\![n]\!](\rho) = \mathbb{F}^\phi[\![n]\!](\rho) -_\omega \mathbb{R}[\![n]\!](\rho) = \circ^\phi(n) -_\omega n = \Gamma^\phi_{a_\omega}(n) \tag{2.9}$$

$$\mathcal{E}_r^\phi[\![n]\!](\rho) = \frac{\mathbb{F}^\phi[\![n]\!](\rho)}{\mathbb{R}[\![n]\!](\rho)}{}_\omega -_\omega 1 = \frac{\circ^\phi(n)}{n}{}_\omega -_\omega 1 = \Gamma^\phi_{r_\omega}(n) \tag{2.10}$$

Pour les variables, comme à l'accoutumée, il suffit de récupérer les erreurs absolues et relatives stockées dans l'état machine. Ces dernières seront, comme nous le verrons lors de la définition de la sémantique des instructions, équivalentes à interpréter la dernière expression affectée à la variable en utilisant les sémantiques d'erreur. Ainsi :

$$\mathcal{E}_a^\phi[\![x]\!](\rho) = \rho_{e_a}(x) \quad \text{et} \quad \mathcal{E}_r^\phi[\![x]\!](\rho) = \rho_{e_r}(x) \tag{2.11}$$

Continuons avec les différentes opérations arithmétiques. Pour ce faire, commençons par introduire quelques notations qui simplifierons ensuite la lecture. Nous noterons, par abus de langage, e l'interprétation dans la sémantique exacte de l'expression e. Nous noterons également $\widetilde{e} = \mathbb{F}^\phi[\![e]\!](\rho)$ l'interprétation de e dans la sémantique approximée où le format ϕ et l'état machine ρ seront déterminés par le contexte. Enfin, les notations $\mathcal{E}_a^\phi(e) = \mathcal{E}_a^\phi[\![e]\!](\rho)$ et $\mathcal{E}_r^\phi(e) = \mathcal{E}_r^\phi[\![e]\!](\rho)$ représenteront respectivement l'interprétation de e dans la sémantique d'erreur absolue et celle dans la sémantique d'erreur relative.

Une première application des définitions des sémantiques exactes et approximées nous permet, pour toute opération n-aire $\diamond$ associée à une construction $\blacklozenge$ du langage, de réécrire les définitions des sémantiques d'erreur absolue et relative comme suit :

$$\mathcal{E}_a^\phi[\![\blacklozenge(e_1, \ldots, e_n)]\!](\rho) = \circ_\omega^\phi(\diamond_\omega(\widetilde{e_1}, \ldots, \widetilde{e_n})) -_\omega \diamond_\omega(e_1, \ldots, e_n) \tag{2.12}$$

$$\mathcal{E}_r^\phi[\![\blacklozenge(e_1, \ldots, e_n)]\!](\rho) = \frac{\circ_\omega^\phi(\diamond_\omega(\widetilde{e_1}, \ldots, \widetilde{e_n}))}{\diamond_\omega(e_1, \ldots, e_n)}\omega -_\omega 1 \tag{2.13}$$

La norme ANSI/IEEE-754 impose que toutes les opérations de notre langage soient *correctement arrondies*, ce qui nous permet d'utiliser les propriétés présentées en section 1.4. On peut donc réécrire les définitions 2.12 et 2.13 de la manière suivante :

$$\mathcal{E}_a^\phi[\![\blacklozenge(e_1, \ldots, e_n)]\!](\rho) = \diamond_\omega(\widetilde{e_1}, \ldots, \widetilde{e_n}) -_\omega \diamond_\omega(e_1, \ldots, e_n) +_\omega \Gamma_{a_\omega}^\phi(\diamond_\omega(\widetilde{e_1}, \ldots, \widetilde{e_n})) \tag{2.14}$$

$$\mathcal{E}_r^\phi[\![\blacklozenge(e_1, \ldots, e_n)]\!](\rho) = \frac{\diamond_\omega(\widetilde{e_1}, \ldots, \widetilde{e_n})}{\diamond_\omega(e_1, \ldots, e_n)}\omega \cdot_\omega \left(\Gamma_{r_\omega}^\phi(\diamond_\omega(\widetilde{e_1}, \ldots, \widetilde{e_n})) +_\omega 1\right) -_\omega 1 \tag{2.15}$$

Ce sont de ces définitions que nous partirons pour définir les sémantiques d'erreur de toutes les opérations arithmétiques du langage. Cependant, avant de présenter un raffinement des définitions 2.7 et 2.8 pour ces opérateurs, il nous semble important de souligner que les définitions 2.7 et 2.8 nous permettent d'exprimer la valeur approximée d'une expression en termes de sa valeur exacte et de son erreur absolue ou relative. On a ainsi :

$$\mathbb{F}^\phi[\![e]\!](\rho) = e +_\omega \mathcal{E}_a^\phi(e) = e \times_\omega \left(\mathcal{E}_r^\phi(e) +_\omega 1\right) \tag{2.16}$$

Addition et soustraction : pour l'addition et la soustraction, les définitions 2.7 et 2.8 se réécrivent comme suit :

$$\mathcal{E}_a^\phi[\![e_1 \pm e_2]\!](\rho) = \mathcal{E}_a^\phi(e_1) \pm_\omega \mathcal{E}_a^\phi(e_2) +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \pm_\omega \widetilde{e_2}) \tag{2.17}$$

$$\begin{aligned}\mathcal{E}_r^\phi[\![e_1 \pm e_2]\!](\rho) = \frac{e_1 \cdot_\omega \left(\mathcal{E}_r^\phi(e_1) +_\omega 1\right) \pm_\omega e_2 \cdot_\omega \left(\mathcal{E}_r^\phi(e_2) +_\omega 1\right)}{e_1 \pm_\omega e_2}{}_\omega \\ \times_\omega \left(\Gamma_{r_\omega}^\phi(\widetilde{e_1} \pm_\omega \widetilde{e_2}) +_\omega 1\right) -_\omega 1\end{aligned} \tag{2.18}$$

Démonstration. La définition pour l'erreur absolue s'obtient trivialement en appliquant la propriété 2.16 à la définition 2.7 avant de simplifier. Pour l'erreur relative, il suffit de commencer par remplacer les sémantiques exacte et approximée par leurs définitions avant de suivre le même raisonnement que pour l'erreur absolue. □

Pour ces opérations, c'est l'erreur absolue qui nous donne le plus d'intuition. En effet, l'équation 2.17 montre que les erreurs des opérandes sont propagées et cumulées avant d'ajouter l'erreur d'arrondi induite par l'opération elle même.

Exemple 2.2.3. Prenons pour exemple l'expression $(3+6)-7$ et calculons l'erreur absolue commise en l'évaluant dans le format $\mathbb{F}_m$:

$$\begin{aligned}\mathcal{E}_a^\phi[\![(3+6)-7]\!](\rho) &= \mathcal{E}_a^\phi(3+6) -_\omega \mathcal{E}_a^\phi(7) +_\omega \Gamma_{a_\omega}^\phi(\widetilde{(3+6)} -_\omega \widetilde{7}) \\ &= \left(\mathcal{E}_a^\phi(3) +_\omega \mathcal{E}_a^\phi(6) +_\omega \Gamma_{a_\omega}^\phi(\widetilde{3} +_\omega \widetilde{6})\right) -_\omega \mathcal{E}_a^\phi(7) \\ &+_\omega \Gamma_{a_\omega}^\phi(\widetilde{(3+6)} -_\omega \widetilde{7})\end{aligned}$$

Arrêtons nous un instant pour étudier chaque terme de cette expression. Tout d'abord, les termes $\mathcal{E}_a^\phi(3)$, $\mathcal{E}_a^\phi(6)$ et $\mathcal{E}_a^\phi(7)$ valent tous zéro car ces trois nombres sont représentables dans $\mathbb{F}_m$. Ensuite, le terme $\Gamma_a^\phi(\widetilde{3}+\widetilde{6})$ est intéressant. En effet, il représente l'erreur induite par l'arrondi de $3+6=9$ vers $\mathbb{F}_m$, qui donne $\widetilde{8}$. La valeur de $\Gamma_a^\phi(\widetilde{3}+\widetilde{6})$ est donc $8-9=-1$. Pour finir, l'erreur élémentaire $\Gamma_{a_\omega}^\phi(\widetilde{(3+6)} -_\omega \widetilde{7})$ vaut elle zéro. En effet, elle s'intéresse à l'erreur induite par l'arrondi de la différence entre le résultat flottant de $3+6$, à savoir 8, et 7. Cette différence se calcule correctement dans $\mathbb{F}_m$ et donc cette erreur vaut 0. Ainsi, l'erreur absolue de l'expression $(3+6)-7$ dans le format $\mathbb{F}_m$ vaut -1.

Multiplication : pour la multiplication, les définitions 2.7 et 2.8 se réécrivent comme suit :

$$\begin{aligned}\mathcal{E}_a^\phi[\![e_1 \times e_2]\!](\rho) = e_1 \cdot_\omega \mathcal{E}_a^\phi(e_2) +_\omega e_2 \cdot_\omega \mathcal{E}_a^\phi(e_1) +_\omega \mathcal{E}_a^\phi(e_1) \cdot_\omega \mathcal{E}_a^\phi(e_2) \\ +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \pm_\omega \widetilde{e_2})\end{aligned} \tag{2.19}$$

$$\begin{aligned}\mathcal{E}_r^\phi[\![e_1 \times e_2]\!](\rho) = \left(\mathcal{E}_r^\phi(e_1) +_\omega 1\right) \times_\omega \left(\mathcal{E}_r^\phi(e_2) +_\omega 1\right) \\ \times_\omega \left(\Gamma_{r_\omega}^\phi(\widetilde{e_1} \cdot_\omega \widetilde{e_2}) +_\omega 1\right) -_\omega 1\end{aligned} \tag{2.20}$$

Démonstration. La définition pour l'erreur relative se déduit de la manière suivante :

$$\begin{aligned}
\mathcal{E}_r^\phi[\![e_1 \times e_2]\!](\rho) &= \frac{\widetilde{e_1} \times_\omega \widetilde{e_2}}{e_1 \times_\omega e_2}\omega \cdot_\omega \left(\Gamma^\phi_{r_\omega}(\widetilde{e_1} \cdot_\omega \widetilde{e_2}) +_\omega 1\right) -_\omega 1 && \text{(a)}\\
&= \frac{\widetilde{e_1}}{e_1}\omega \cdot_\omega \frac{\widetilde{e_2}}{e_2}\omega \cdot_\omega \left(\Gamma^\phi_{r_\omega}(\widetilde{e_1} \cdot_\omega \widetilde{e_2}) +_\omega 1\right) -_\omega 1 && \text{(b)}\\
&= \left(\mathcal{E}_r^\phi(e_1) +_\omega 1\right) \times_\omega \left(\mathcal{E}_r^\phi(e_2) +_\omega 1\right)\\
&\quad \times_\omega \left(\Gamma^\phi_{r_\omega}(\widetilde{e_1} \cdot_\omega \widetilde{e_2}) +_\omega 1\right) -_\omega 1 && \text{(c)}
\end{aligned}$$

Le raisonnement commence à l'étape (a) en partant de la définition 2.15. L'étape (b) consiste à regrouper les termes pour chaque sous-expression, faisant apparaitre récursivement la définition 2.8 pour e_1 et e_2. Ceci nous permet de conclure, à l'étape (c), avec l'équation attendue.

Pour l'erreur absolue, le raisonnement est le suivant :

$$\begin{aligned}
\mathcal{E}_a^\phi[\![e_1 \times e_2]\!](\rho) &= \widetilde{e_1} \cdot_\omega \widetilde{e_2} -_\omega e_1 \cdot_\omega e_2 +_\omega \Gamma^\phi_{a_\omega}(\widetilde{e_1} \cdot_\omega \widetilde{e_2}) && \text{(a)}\\
&= \left(e_1 +_\omega \mathcal{E}_a^\phi(e_1)\right) \cdot_\omega \left(e_2 +_\omega \mathcal{E}_a^\phi(e_2)\right) -_\omega e_1 \cdot_\omega e_2\\
&\quad +_\omega \Gamma^\phi_{a_\omega}(\widetilde{e_1} \cdot_\omega \widetilde{e_2}) && \text{(b)}\\
&= e_1 \cdot_\omega \mathcal{E}_a^\phi(e_2) +_\omega e_2 \cdot_\omega \mathcal{E}_a^\phi(e_1) +_\omega \mathcal{E}_a^\phi(e_1) \cdot_\omega \mathcal{E}_a^\phi(e_2)\\
&\quad +_\omega \Gamma^\phi_{a_\omega}(\widetilde{e_1} \cdot_\omega \widetilde{e_2}) && \text{(c)}
\end{aligned}$$

On commence à l'étape (a) en partant de la définition 2.14. Ensuite, l'étape (b) consiste à appliquer la réécriture de la sémantique flottante des opérandes en termes de leurs erreurs absolues, à l'aide de la propriété 2.16. Finalement, l'étape (c) consiste à développer les premiers termes avant de simplifier ce qui peut l'être. □

Ici, c'est la définition de l'erreur relative qui nous apporte le plus d'intuition. En effet, les erreurs relatives des opérandes sont propagées et multipliées entre elles, avant d'être complétées par l'erreur d'arrondi induite par l'opération elle même.

Exemple 2.2.4. Prenons pour exemple l'expression 1.6×0.4 et calculons l'erreur relative commise en l'évaluant dans le format $\mathbb{F}_m$ (comme ici, aucune exception n'est levée, nous nous permettons de retirer les annotations associées) :

$$\begin{aligned}
\mathcal{E}_r^\phi[\![1.6 \times 0.4]\!](\rho) &= \left(\mathcal{E}_r^\phi(1.6) + 1\right)\left(\mathcal{E}_r^\phi(0.4) + 1\right)\left(\Gamma_r^\phi(\widetilde{1.6} \times \widetilde{0.4}) + 1\right) - 1\\
&= \left(\Gamma_r^\phi(1.6) + 1\right)\left(\Gamma_r^\phi(0.4) + 1\right)\left(\Gamma_r^\phi(\widetilde{1.6} \times \widetilde{0.4}) + 1\right) - 1
\end{aligned}$$

Il nous faut maintenant calculer les trois erreurs relatives élémentaires de l'équation. Nous rappelons que l'erreur relative élémentaire d'un calcul est définie comme suit :

$$\Gamma_r^\phi(x) = \frac{\circ^\phi(x)}{x} - 1$$

Or, comme dans notre équation chaque erreur relative se voit ajouter 1, nous n'avons qu'à calculer la partie fractionnaire pour 1.6, 0.4 et $\widetilde{1.6} \times \widetilde{0.4}$. Pour les deux premiers, les flottants les représentants dans le format $\mathbb{F}_m$ sont respectivement 1.5 et 0.375. Le rapport entre le flottant et le réel est, dans ces deux cas, égal à $\frac{15}{16}$. Le dernier rapport est calculé comme suit :

$$\frac{\circ^{\mathbb{F}_m}\left(\widetilde{1.6} \times \widetilde{0.4}\right)}{\widetilde{1.6} \times \widetilde{0.4}} = \frac{\circ^{\mathbb{F}_m}(1.5 \times 0.375)}{1.5 \times 0.375} = \frac{0.5}{0.5625} = \frac{8}{9}$$

Ainsi donc, l'erreur relative commise en calculant cette expression dans le format $\mathbb{F}_m$ est :

$$\mathcal{E}_r^\phi[\![1.6 \times 0.4]\!](\rho) = \frac{15}{16} \times \frac{15}{16} \times \frac{8}{9} - 1 = -\frac{7}{32}$$

Division : pour la division, les définitions 2.7 et 2.8 se réécrivent comme suit :

$$\mathcal{E}_a^\phi[\![e_1 \div e_2]\!](\rho) = \frac{e_2 \cdot_\omega \mathcal{E}_a^\phi(e_1) -_\omega e_1 \cdot_\omega \mathcal{E}_a^\phi(e_2)}{\widetilde{e_2} \cdot_\omega e_2}{}_\omega +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2}) \tag{2.21}$$

$$\mathcal{E}_r^\phi[\![e_1 \div e_2]\!](\rho) = \frac{\mathcal{E}_r^\phi(e_1) +_\omega 1}{\mathcal{E}_r^\phi(e_2) +_\omega 1}{}_\omega \cdot_\omega \left(\Gamma_{r_\omega}^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2}) +_\omega 1\right) -_\omega 1 \tag{2.22}$$

Démonstration. La démonstration pour l'erreur relative suit rigoureusement le même raisonnement que pour la multiplication, en remplaçant simplement les multiplications par des divisions là où cela est nécessaire. Pour l'erreur absolue, le raisonnement est le suivant :

$$\begin{aligned}
\mathcal{E}_a^\phi[\![e_1 \div e_2]\!](\rho) &= \frac{\widetilde{e_1}}{\widetilde{e_2}}{}_\omega -_\omega \frac{e_1}{e_2}{}_\omega +_\omega \Gamma_a^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2}) && \text{(a)}\\
&= \frac{\widetilde{e_1} \cdot_\omega e_2 -_\omega e_1 \cdot_\omega \widetilde{e_2}}{\widetilde{e_2} \cdot_\omega e_2}{}_\omega +_\omega \Gamma_a^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2}) && \text{(b)}\\
&= \frac{\left(e_1 +_\omega \mathcal{E}_a^\phi(e_1)\right) \cdot_\omega e_2 -_\omega e_1 \cdot_\omega \left(e_2 +_\omega \mathcal{E}_a^\phi(e_2)\right)}{\widetilde{e_2} \cdot_\omega e_2}{}_\omega && \text{(c)}\\
&\quad +_\omega \Gamma_a^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2})\\
&= \frac{e_2 \cdot_\omega \mathcal{E}_a^\phi(e_1) -_\omega e_1 \cdot_\omega \mathcal{E}_a^\phi(e_2)}{\widetilde{e_2} \cdot_\omega e_2}{}_\omega +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2}) && \text{(d)}
\end{aligned}$$

L'étape (a) est simplement la définition 2.14 appliquée à la division. L'étape (b) regroupe la fraction concernant les flottants et celle concernant les réels en une seule. L'étape (c) consiste à appliquer la réécriture de la sémantique flottante des opérandes en termes de leurs erreurs absolues, à l'aide de la propriété 2.16. Finalement, l'étape (d) consiste à développer et simplifier ce qui peut l'être. □

Ici encore, c'est la définition de l'erreur relative qui nous apporte le plus d'intuition. En effet, comme pour la multiplication, les erreurs relatives des opérandes sont propagées et divisées entre elles, avant d'être complétées par l'erreur d'arrondi induite par l'opération elle-même.

Exemple 2.2.5. Prenons pour exemple l'expression $1.6 \dot{\div} 0.4$ et calculons l'erreur relative commise en l'évaluant dans le format $\mathbb{F}_m$. Comme pour la multiplication, aucune exception n'est levée par cette évaluation, et nous nous permettons donc de retirer les annotations associées. Nous avons ainsi :

$$\mathcal{E}_r^\phi[\![1.6 \dot{\div} 0.4]\!](\rho) = \frac{\Gamma_r^\phi(1.6)+1}{\Gamma_r^\phi(0.4)+1}\left(\Gamma_r^\phi(\widetilde{1.6} \div \widetilde{0.4})+1\right) - 1$$

Nous avions calculé, lors de l'exemple de la multiplication, les erreurs relatives élémentaires, décalées de un, associées à 1.6 et 0.4. Ces dernières valent toutes deux $\frac{15}{16}$. Il nous reste ainsi à calculer le troisième terme d'erreur relative élémentaire. Nous ne calculons ici que le rapport entre le flottant et le réel, étant donné que le reste est annulé par l'ajout de 1 dans l'équation que nous cherchons à évaluer :

$$\frac{\circ^{\mathbb{F}_m}\left(\widetilde{1.6} \div \widetilde{0.4}\right)}{\widetilde{1.6} \div \widetilde{0.4}} = \frac{\circ^{\mathbb{F}_m}(1.5 \div 0.375)}{1.5 \div 0.375} = \frac{\circ^{\mathbb{F}_m}(4)}{4} = 1$$

L'erreur relative commise en évaluant cette expression est donc nulle. Dit autrement, le résultat de ce calcul dans le format $\mathbb{F}_m$ est exact.

Négation : la négation d'une expression flottante n'introduit pas d'arrondi. En effet, tout flottant dispose d'un opposé (voir corollaire 1.1.1). Ainsi donc :

$$\mathcal{E}_a^\phi[\![-e]\!](\rho) = -\mathcal{E}_a^\phi[\![e]\!](\rho) \tag{2.23}$$
$$\mathcal{E}_r^\phi[\![-e]\!](\rho) = \mathcal{E}_r^\phi[\![e]\!](\rho) \tag{2.24}$$

On notera tout de même que l'erreur absolue d'une négation est la négation de la sous-expression, ce qui découle naturellement de sa définition.

Racine carrée : pour la racine carrée, les définitions 2.7 et 2.8 se réécrivent comme suit :

$$\mathcal{E}_a^\phi[\![\mathbf{sqrt}(e)]\!](\rho) = \overset{\omega}{\sqrt}{\widetilde{e}} \left(\overset{\omega}{\sqrt}{\frac{\mathcal{E}_a^\phi(e)}{e}} \omega +_\omega 1 \right) -_\omega 1 +_\omega \Gamma_{a_\omega}^\phi(\overset{\omega}{\sqrt}{\widetilde{e}}) \tag{2.25}$$

$$\mathcal{E}_r^\phi[\![\mathbf{sqrt}(e)]\!](\rho) = \overset{\omega}{\sqrt}{\mathcal{E}_r^\phi(e) +_\omega 1} \cdot_\omega \left(\Gamma_{r_\omega}^\phi(\overset{\omega}{\sqrt}{\widetilde{e}}) +_\omega 1 \right) -_\omega 1 \tag{2.26}$$

Démonstration. Pour l'erreur relative, le raisonnement est le suivant :

$$\begin{aligned}
\mathcal{E}_r^\phi[\![\sqrt{e}]\!](\rho) &= \frac{\overset{\omega}{\sqrt}{\widetilde{e}}}{\overset{\omega}{\sqrt}{e}} \omega \cdot_\omega \left(\Gamma_{r_\omega}^\phi(\overset{\omega}{\sqrt}{\widetilde{e}}) +_\omega 1 \right) -_\omega 1 && \text{(a)} \\
&= \frac{\overset{\omega}{\sqrt}{e \cdot_\omega \left(\mathcal{E}_r^\phi(e) +_\omega 1 \right)}}{\overset{\omega}{\sqrt}{e}} \omega \cdot_\omega \left(\Gamma_{r_\omega}^\phi(\overset{\omega}{\sqrt}{\widetilde{e}}) +_\omega 1 \right) -_\omega 1 && \text{(b)} \\
&= \overset{\omega}{\sqrt}{\mathcal{E}_r^\phi(e) +_\omega 1} \cdot_\omega \left(\Gamma_{r_\omega}^\phi(\overset{\omega}{\sqrt}{\widetilde{e}}) +_\omega 1 \right) -_\omega 1 && \text{(c)}
\end{aligned}$$

L'étape (a) est simplement l'application de la définition 2.15 à la racine carrée. Ensuite, nous appliquons la définition 2.16 pour faire apparaitre l'erreur relative de l'opérande, avant de simplifier à l'étape (c) les deux termes en $\sqrt{e}$ et ainsi retrouver l'équation attendue. L'erreur absolue est, quant à elle, obtenue trivialement à partir de l'erreur relative. Cette définition ne nous apporte d'ailleurs pas tellement d'intuition. □

Ici encore, c'est l'erreur relative qui nous apporte le plus d'intuition. En effet, la définition nous montre que l'erreur relative de l'opérande est propagée au travers d'une racine carrée, avant d'être complétée par l'erreur d'arrondi induite par l'opération elle même. Comme nous l'avons déjà dit, l'erreur absolue ne nous avance pas plus vu qu'elle découle directement de l'erreur relative.

Exemple 2.2.6. Prenons pour exemple l'expression **sqrt**(4.41) et calculons l'erreur relative commise en l'évaluant dans le format $\mathbb{F}_m$. Ici encore, aucune exception n'est levée et nous nous permettons donc de retirer les annotations associées. Nous avons ainsi :

$$\mathcal{E}_r^\phi[\![\mathbf{sqrt}(4.41)]\!](\rho) = \sqrt{\Gamma_r^\phi(4.41) + 1} \left(\Gamma_r^\phi \left(\sqrt{\widetilde{4.41}} \right) + 1 \right) - 1$$

Comme pour les exemples précédents, nous devons calculer les rapports entre flottant et réel pour chaque erreur relative élémentaire. Pour la première, le flottant représentant 4.41 dans $\mathbb{F}_m$ est 4. Leur rapport vaut, une fois mis à la racine carrée, $\frac{20}{21}$. Pour la seconde, le terme $\widetilde{4.41}$ vaut, comme nous venons de le dire, 4. La racine de ce nombre est tout à fait représentable dans $\mathbb{F}_m$. Il n'y a donc ici pas de nouvelle erreur. Ainsi, l'erreur relative commise en évaluant **sqrt**(4.41) dans le format $\mathbb{F}_m$ est $-\frac{1}{21}$.

Annotation de format : pour l'annotation de format, les définitions 2.7 et 2.8 se réécrivent comme suit :

$$\mathcal{E}_a^{\phi_1}[\![(\phi_2)\,(e)]\!](\rho) = \mathcal{E}_a^{\phi_2}[\![e]\!] +_\omega \Gamma_{a_\omega}^{\phi_1}\left(\mathbb{F}^{\phi_2}[\![e]\!](\rho)\right) \tag{2.27}$$

$$\mathcal{E}_r^{\phi_1}[\![(\phi_2)\,(e)]\!](\rho) = \left(\mathcal{E}_r^{\phi_2}[\![e]\!](\rho) +_\omega 1\right) \cdot_\omega \left(\Gamma_{r_\omega}^{\phi_1}\left(\mathbb{F}^{\phi_2}[\![e]\!](\rho)\right) +_\omega 1\right) -_\omega 1 \tag{2.28}$$

Démonstration. Pour l'erreur absolue, le raisonnement est le suivant :

$$\begin{aligned}
\mathcal{E}_a^{\phi_1}[\![(\phi_2)\,(e)]\!](\rho) &= \mathbb{F}^{\phi_1}[\![(\phi_2)\,(e)]\!](\rho) -_\omega \mathbb{R}[\![(\phi_2)\,(2)]\!](\rho) && \text{(a)}\\
&= \circ_\omega^{\phi_1}\left(\mathbb{F}^{\phi_2}[\![e]\!](\rho)\right) -_\omega \mathbb{R}[\![e]\!](\rho) && \text{(b)}\\
&= \mathbb{F}^{\phi_2}[\![e]\!](\rho) -_\omega \mathbb{R}[\![e]\!](\rho) +_\omega \Gamma_{a_\omega}^{\phi_1}\left(\mathbb{F}^{\phi_2}[\![e]\!](\rho)\right) && \text{(c)}\\
&= \mathcal{E}_a^{\phi_2}[\![e]\!](\rho) +_\omega \Gamma_{a_\omega}^{\phi_1}\left(\mathbb{F}^{\phi_2}[\![e]\!](\rho)\right) && \text{(d)}
\end{aligned}$$

Nous commençons en (a) avec la définition 2.7 de l'erreur absolue. Ensuite, l'étape (b) consiste à appliquer les définitions des sémantiques exactes et approchées. En (c), l'opérateur d'arrondi est exprimé en fonction de l'erreur absolue élémentaire, ce qui fait apparaitre l'erreur absolue de l'opérande. Cette réécriture conclue le raisonnement en (d). Le raisonnement pour l'erreur relative est rigoureusement identique, en appliquant les définitions liées à l'erreur relative en lieu et place de celles liées à l'erreur absolue. □

Ici, les deux réécritures nous offrent la même intuition, à savoir que l'erreur causée par l'évaluation dans le format ϕ_2 de la sous-expression est conservée et complétée par l'erreur d'arrondi de la conversion dans le format ϕ_1.

Nous avons désormais défini toutes les sémantiques pour les expressions dont nous aurons besoin par la suite. La figure 2.4 ci-dessous en présente un résumé complet. Pour compléter la présentation de la sémantique de notre langage, nous devons à présent présenter l'interprétation des conditions et des instructions.

2.2.6 Sémantiques des conditions et des instructions

Comme nous l'avions dit en début de chapitre, nous cherchons à définir une sémantique en quatre parties, permettant de rendre compte du comportement idéal, du comportement approximé et des deux erreurs, afin de pouvoir définir par la suite notre analyse. Nous avons gardé les présentations de ces quatre parties séparées jusqu'à maintenant afin de simplifier notre exposé. Il est désormais temps de les regrouper, afin de présenter une unique sémantique des conditions et une unique des instructions pour notre langage.

Nous allons commencer par présenter la sémantique des conditions. La notation $\mathcal{C}[\![c]\!]$ représentera la dénotation de c, c'est-à-dire la fonction qui, à partir d'un état

$$\mathcal{E}_a^\phi[\![n]\!](\rho) = \Gamma_{a_\omega}^\phi(n) \qquad \mathcal{E}_a^\phi[\![x]\!](\rho) = \rho_{e_a}(x)$$
$$\mathcal{E}_r^\phi[\![n]\!](\rho) = \Gamma_{a_\omega}^\phi(n) \qquad \mathcal{E}_r^\phi[\![x]\!](\rho) = \rho_{e_r}(x)$$

$$\mathcal{E}_a^\phi[\![e_1 \pm e_2]\!](\rho) = \mathcal{E}_a^\phi(e_1) \pm_\omega \mathcal{E}_a^\phi(e_2) +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \pm_\omega \widetilde{e_2})$$
$$\mathcal{E}_r^\phi[\![e_1 \pm e_2]\!](\rho) = \frac{e_1 \cdot_\omega \left(\mathcal{E}_r^\phi(e_1) +_\omega 1\right) \pm_\omega e_2 \cdot_\omega \left(\mathcal{E}_r^\phi(e_2) +_\omega 1\right)}{e_1 \pm_\omega e_2}{}_\omega$$
$$\times_\omega \left(\Gamma_{r_\omega}^\phi(\widetilde{e_1} \pm_\omega \widetilde{e_2}) +_\omega 1\right) -_\omega 1$$

$$\mathcal{E}_a^\phi[\![e_1 \times e_2]\!](\rho) = e_1 \cdot_\omega \mathcal{E}_a^\phi(e_2) +_\omega e_2 \cdot_\omega \mathcal{E}_a^\phi(e_1) +_\omega \mathcal{E}_a^\phi(e_1) \cdot_\omega \mathcal{E}_a^\phi(e_2)$$
$$+_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \pm_\omega \widetilde{e_2})$$
$$\mathcal{E}_r^\phi[\![e_1 \times e_2]\!](\rho) = \left(\mathcal{E}_r^\phi(e_1) +_\omega 1\right) \times_\omega \left(\mathcal{E}_r^\phi(e_2) +_\omega 1\right)$$
$$\times_\omega \left(\Gamma_{r_\omega}^\phi(\widetilde{e_1} \cdot_\omega \widetilde{e_2}) +_\omega 1\right) -_\omega 1$$

$$\mathcal{E}_a^\phi[\![e_1 \div e_2]\!](\rho) = \frac{e_2 \cdot_\omega \mathcal{E}_a^\phi(e_1) -_\omega e_1 \cdot_\omega \mathcal{E}_a^\phi(e_2)}{\widetilde{e_2} \cdot_\omega e_2}\omega +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2})$$
$$\mathcal{E}_r^\phi[\![e_1 \div e_2]\!](\rho) = \frac{\mathcal{E}_r^\phi(e_1) +_\omega 1}{\mathcal{E}_r^\phi(e_2) +_\omega 1}\omega \cdot_\omega \left(\Gamma_{r_\omega}^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2}) +_\omega 1\right) -_\omega 1$$

$$\mathcal{E}_a^\phi[\![-e]\!](\rho) = -\mathcal{E}_a^\phi[\![e]\!](\rho)$$
$$\mathcal{E}_r^\phi[\![-e]\!](\rho) = \mathcal{E}_r^\phi[\![e]\!](\rho)$$

$$\mathcal{E}_a^\phi[\![\mathbf{sqrt}(e)]\!](\rho) = \sqrt[\omega]{e +_\omega \mathcal{E}_a^\phi(e)} -_\omega \sqrt[\omega]{e} +_\omega \Gamma_{a_\omega}^\phi(\sqrt[\omega]{\widetilde{e}})$$
$$\mathcal{E}_r^\phi[\![\mathbf{sqrt}(e)]\!](\rho) = \sqrt[\omega]{\mathcal{E}_r^\phi(e) +_\omega 1} \cdot_\omega \left(\Gamma_{r_\omega}^\phi(\sqrt[\omega]{\widetilde{e}}) +_\omega 1\right) -_\omega 1$$

$$\mathcal{E}_a^{\phi_1}[\![(\phi_2)\,(e)]\!](\rho) = \mathcal{E}_a^{\phi_2}[\![e]\!] +_\omega \Gamma_{a_\omega}^{\phi_1}\left(\mathbb{F}^{\phi_2}[\![e]\!](\rho)\right)$$
$$\mathcal{E}_r^{\phi_1}[\![(\phi_2)\,(e)]\!](\rho) = \left(\mathcal{E}_r^{\phi_2}[\![e]\!](\rho) +_\omega 1\right) \cdot_\omega \left(\Gamma_{r_\omega}^{\phi_1}\left(\mathbb{F}^{\phi_2}[\![e]\!](\rho)\right) +_\omega 1\right) -_\omega 1$$

FIGURE 2.4 – Sémantiques d'erreurs absolue et relative des expressions.

machine, produit un booléen, indiquant si la condition est vraie ou non, ou une exception :

$$\mathcal{C}[\![c]\!] : \Sigma \mapsto \{\text{Vrai}, \text{Faux}, \omega\}$$

Sa définition, présentée en figure 2.5, interprète tous les opérateurs booléens avec le sens attendu. La seule vraie particularité se trouve au niveau de l'interprétation de l'inégalité. En effet, cette dernière est interprétée en utilisant la sémantique

$$\begin{aligned}
\mathcal{C}[\![(\phi)\,(e_1 \leq e_2)]\!](\rho) &\triangleq \mathbb{F}^{\phi}[\![e_1]\!](\rho) \leq \mathbb{F}^{\phi}[\![e_2]\!](\rho) \\
\mathcal{C}[\![c_1 \wedge c_2]\!](\rho) &\triangleq \mathcal{C}[\![c_1]\!](\rho) \wedge \mathcal{C}[\![c_2]\!](\rho) \\
\mathcal{C}[\![c_1 \vee c_2]\!](\rho) &\triangleq \mathcal{C}[\![c_1]\!](\rho) \vee \mathcal{C}[\![c_2]\!](\rho) \\
\mathcal{C}[\![\neg c]\!](\rho) &\triangleq \neg \mathcal{C}[\![c]\!](\rho)
\end{aligned}$$

FIGURE 2.5 – Sémantique des conditions.

approximée du format ϕ qui est syntaxiquement explicite dans cette construction. Ceci est dû au fait que nous nous intéressons à la propagation des erreurs au travers du programme, et non à la sémantique de la différence entre une exécution idéale et une exécution flottante. Nous faisons donc ici le choix, par soucis de simplicité, d'ignorer la problématique des *tests instables* [84], c'est-à-dire les conditions pour lesquelles l'exécution idéale ne suit pas la même branche que l'exécution flottante. Il est donc important de noter que les propriétés de l'analyse que nous allons présenter ne sont valides que sous l'hypothèse que le programme analysé ne contient pas de tests instables. Leur support est une extension que nous souhaitons mener dans de futures développements, et qui profitera de l'expertise acquise durant mon stage de Master, ayant mené à une publication [88].

Pour terminer notre sémantique, il ne nous reste plus qu'à présenter l'interprétation des instructions du langage. La notation $\mathcal{S}[\![s]\!]$ représentera la dénotation de l'instruction s, c'est-à-dire la fonction qui, à partir d'un état machine, produit un nouvel état machine, représentant les évolutions que l'instruction applique, ou une exception :

$$\mathcal{S}[\![s]\!] : \Sigma \mapsto \Sigma \uplus \Omega \tag{2.29}$$

La définition de cette dernière sémantique est présentée en figure 2.6. L'affectation met à jour l'état machine, en affectant à la variable x le quadruplet composé des évaluations par les sémantiques exactes, approchées, d'erreur absolue et d'erreur relative de l'expression e si et seulement si aucune d'entre elles ne lève d'exception. Sinon, l'affectation lève à son tour une exception. Nous utiliserons la notation $\rho\left[x \overset{\omega}{\Leftarrow} v\right]$ pour représenter ce comportement. Nous ne fournirons pas ici sa définition, cette dernière étant classique. La séquence interprète la première instruction

$$\mathcal{S}[\![x = (\phi)\,(e)\ ;]\!](\rho) \triangleq \rho\left[x \overset{\omega}{\leftarrow} \left(\mathbb{R}[\![e]\!](\rho), \mathbb{F}^{\phi}[\![e]\!](\rho), \mathcal{E}_a^{\phi}[\![e]\!](\rho), \mathcal{E}_r^{\phi}[\![e]\!](\rho)\right)\right]$$

$$\mathcal{S}[\![s_1\ s_2]\!](\rho) \triangleq \mathcal{S}[\![\mathrm{s}_2]\!]_{\omega}(\mathcal{S}[\![s_1]\!](\rho)) \qquad \mathcal{S}[\![\textbf{skip ;}]\!](\rho) \triangleq \rho$$

$$\mathcal{S}[\![\textbf{if } c \textbf{ then } s_t \textbf{ else } s_e \textbf{ end}]\!](\rho) \triangleq \begin{cases} \omega & \text{si } \rho = \omega \vee \mathcal{C}[\![c]\!](\rho) = \omega \\ \mathcal{S}[\![s_t]\!](\rho) & \text{si } \rho \neq \omega \wedge \mathcal{C}[\![c]\!](\rho) = \text{Vrai} \\ \mathcal{S}[\![s_e]\!](\rho) & \text{si } \rho \neq \omega \wedge \mathcal{C}[\![c]\!](\rho) = \text{Faux} \end{cases}$$

$$\mathcal{S}[\![\textbf{while } c \textbf{ do } s \textbf{ done}]\!](\rho) \triangleq \mathrm{fix}\,(F_{c,s})\,(\rho)$$
$$\text{avec } F_{c,s}(f) = \lambda\rho'.\begin{cases} \omega & \text{si } \rho' = \omega \vee \mathcal{C}[\![c]\!](\rho') = \omega \\ f(\mathcal{S}[\![s]\!](\rho')) & \text{si } \rho' \neq \omega \wedge \mathcal{C}[\![c]\!](\rho') = \text{Vrai} \\ \rho' & \text{si } \rho' \neq \omega \wedge \mathcal{C}[\![c]\!](\rho') = \text{Faux} \end{cases}$$

FIGURE 2.6 – Sémantique des instructions.

dans l'état machine fourni, puis la seconde en utilisant l'état machine fourni par la première instruction si cette dernière n'a pas levé d'exception. L'identité retourne simplement l'état machine fourni. Le test interprète la branche **then** si l'évaluation de la condition c renvoie vraie, interprète la branche **else** si l'évaluation de c renvoie faux, et lève une exception sinon. Enfin, l'évaluation de la boucle consiste à calculer le plus petit point fixe d'une fonction un peu particulière, sur laquelle il convient donc de nous arrêter. En effet, cette fonction est en fait d'ordre supérieure, elle manipule et renvoie des fonctions :

$$F_{c,s} : (\Sigma \uplus \Omega \mapsto \Sigma \uplus \Omega) \mapsto (\Sigma \uplus \Omega \mapsto \Sigma \uplus \Omega) \tag{2.30}$$

Plus précisément, l'objectif du calcul de point fixe de cette fonction est de représenter les boucles comme la construction d'une fonction décrivant, de plus en plus précisément, ce que la boucle fait. À la fin du processus, s'il termine, la fonction retournée décrit entièrement la boucle, lui donnant ainsi un sens mathématique.

Cependant, il nous reste un problème à résoudre. En effet, rien pour le moment ne nous garantit que $F_{c,s}$ dispose d'un point fixe, ni qu'il en existe un plus petit ! Cette garantie est apportée par le théorème de Knaster-Tarski [67]. Avant de l'introduire, il nous faut commencer par présenter deux notions sur les fonctions appliquées à des cpos.

Définition 2.2.4. La *puissance n-ième* d'une fonction f, notée f^n, est définie

récursivement comme suit :

$$\begin{cases} f^0(x) & = x \\ f^{n+1}(x) & = f(f^n(x)) \end{cases} \tag{2.31}$$

Définition 2.2.5. Une fonction $f : A \mapsto B$, où $(A, \leq_A)$ et $(B, \leq_B)$ sont deux cpos, est dite *monotones* si elle préserve l'ordre, c'est-à-dire :

$$\forall x, y \in A, \text{ si } x \leq_A y \text{ alors } f(x) \leq_B f(y) \tag{2.32}$$

Définition 2.2.6. Une fonction monotone f d'un cpo A à un cpo B est dite *continue* si elle préserve les supremums des chaînes, c'est-à-dire :

$$\forall \text{ chaîne } C \subseteq A, f(\sup(C)) = \sup(\{f(c) \mid c \in C\}) \tag{2.33}$$

Nous pouvons désormais énoncé le théorème de Knaster-Tarski :

Théorème 2.2.3. Le plus petit point fixe d'une fonction f continue sur un cpo D de plus petit élément $\perp$ existe et est le supremum des puissances itérées de f à partir de $\perp$, c'est-à-dire :

$$\text{fix}(f) = \sup\left(\{f^i(\perp) \mid \forall i \in \mathbb{N}\}\right) \tag{2.34}$$

Démonstration. La preuve de ce théorème se fait en trois étapes, que nous n'allons qu'esquisser ici. Une démonstration plus complète, dont nous nous sommes d'ailleurs largement inspirés, est celle du cours d'Andrew M. Pitts, Glynn Winskel et Marcelo Fiore [67].

1. La première étape consiste à montrer que le supremum des puissances itérées de f à partir de $\perp$ existe. Ceci s'obtient facilement par récurrence, en utilisant la monotonie de f pour l'induction, et la définition de f^0 pour le cas de base.
2. La deuxième étape consiste à montrer que $\text{fix}(f)$ est effectivement un point fixe de f, c'est-à-dire que $f(\text{fix}(f)) = \text{fix}(f)$. Ceci s'obtient à partir de la continuité de f, et en prenant en considération que supprimer les plus petits éléments d'une chaîne ne change pas son supremum.
3. La dernière étape consiste à montrer que ce point fixe est bien le plus petit vis-à-vis de l'ordre sur D. Ceci vient du fait que tout point fixe de f est nécessairement majorant de ses puissance itérées, et que par définition du supremum, $\text{fix}(f)$ est le plus petit majorant de cette chaîne.

□

Ainsi, afin de prouver que la sémantique des boucles est correct, il nous reste donc à montrer que $\Sigma \uplus \Omega \mapsto \Sigma \uplus \Omega$ est bien un cpo, et que $F_{c,s}$ est continue sur ce cpo. Pour le premier point, il nous suffit d'équiper l'ensemble $\Sigma \uplus \Omega \mapsto \Sigma \uplus \Omega$ de la relation suivante :

$$\forall f, g \in \Sigma \uplus \Omega \mapsto \Sigma \uplus \Omega, f \leq_{\mapsto} g \iff \forall x \in \Sigma \uplus \Omega, f(x) \leq_{\omega} g(x) \tag{2.35}$$

où $\leq_{\omega}$ est la relation d'ordre du cpo $\Sigma \uplus \Omega$ que nous avions introduite au théorème 2.2.2. Ce premier point est une généralisation du cpo des états machine que nous avions présenté en 2.2. La preuve reprend d'ailleurs, dans les grandes lignes, celle pour les états machine, et nous ne la ferrons donc pas à nouveau ici. Enfin, la continuité de $F_{c,s}$ se démontre, quant à elle, par analyse de cas sur la syntaxe des instructions du langage.

Ainsi donc, nous avons défini une sémantique complète pour notre langage, permettant de rendre compte des comportements exacts, utilisant l'arithmétique réelle, approximé, utilisant les flottants, et des deux erreurs, absolue et relative. Cependant, cette sémantique n'est pas calculable, ce qui limite grandement son utilisabilité pour construire une analyse quelconque. En effet, bien que le théorème de Knaster-Tarski nous donne une solution constructive au calcul de point fixe, rien ne nous garantit que ce calcul soit exécutable sur un ordinateur en un temps fini. Pire encore, l'indécidabilité du problème de l'arrêt, prouvé par Turing en 1936 [85], et sa généralisation par Rice [70] stipulant que toute propriété non triviale d'un programme est indécidable, nous garantit exactement l'inverse. Cette limitation peut heureusement être contournée de différentes manières, chacune sacrifiant une qualité différente. Dans le prochain chapitre, nous présenterons comment construire une analyse automatique en utilisant l'une de ces méthodes, à savoir l'interprétation abstraite.

Chapitre 3

Interprétation Abstraite

L'interprétation abstraite est une théorie permettant de définir des analyses statiques de programmes dont le but est de prouver automatiquement des propriétés sur ces derniers. Ces propriétés peuvent aller de l'absence de bugs, comme des accès mémoire invalides ou des divisions par zéro, à des notions plus fonctionnelles, permettant de garantir qu'un programme se comporte comme nous le souhaitons. Définir prouver automatiquement et à coup sûr ce genre de propriétés non triviales est cependant une question épineuse. En effet, le théorème de Rice [70] nous impose d'entrée de jeu un mur : cette question est indécidable, il n'existe pas d'algorithme permettant cela.

Afin de contourner ce mur, il va être nécessaire de revoir nos attentes à la baisse. Nous disposons de plusieurs options ici, chacune étant à l'origine d'une technique d'analyse statique :

— relâcher la contrainte d'automatisme, en demandant à l'utilisateur de guider l'analyse au travers d'annotations, amène à la *vérification déductive* [28] ;
— simplifier l'objet d'étude pour rendre la question décidable amène au *model checking* [30, 69] ;
— autoriser l'analyse à répondre « je ne sais pas » amène à l'*interprétation abstraite* [18].

Chacune de ces méthodes présente des avantages et des défauts par rapport aux autres. Dans le cadre de l'analyse d'erreurs numériques, l'utilisation de la vérification déductive ou du model checking sont cependant difficiles. En effet, écrire des annotations ou extraire un modèle en prenant en considération le fonctionnement de l'arithmétique flottante est difficile et contre-intuitive. Ces raisons, couplées au fait que conserver l'automatisme de l'analyse est, en pratique, une caractéristique particulièrement intéressante, font que l'interprétation nous semble la technique la plus adaptée pour notre besoin.

Ainsi, dans ce chapitre, nous allons présenter le cadre théorique de l'interprétation abstraite dont nous aurons besoin, ainsi que les deux instanciations que

nous avons utilisés durant nos travaux. Plus précisément, nous définissons une *sémantique de trace*, puis une *sémantique collectrice* pour notre langage, permettant de raisonner sur de multiples exécutions d'un même programme, avant de présenter les outils théoriques dont nous aurons besoin, à savoir les domaines abstraits, les correspondances de Galois, l'abstraction des opérateurs, le calcul de points fixe abstraits et le produit réduit. Cette présentation est très fortement inspirée de l'excellent tutoriel d'Antoine Miné [60] présentant l'analyse par interprétation abstraite pour la preuve de propriétés numérique. Enfin, nous présenterons les deux domaines abstraits que nous avons utilisés au cours de ces travaux, à savoir le domaine des *intervalles* et le domaine des *zonotopes*, au chapitre 4.

3.1 Pas à pas vers l'abstraction

La sémantique que nous avons présentée au chapitre précédent formalise l'exécution d'un programme à partir d'un état machine. Cependant, elle présente des limites en deux points, empêchant que l'on ne s'en serve telle quelle pour prouver des propriétés. Premièrement, elle ne construit que l'état machine final, résultant de l'exécution complète du programme. Cela pose problème car l'on ne peut pas spécifier de propriétés à vérifier au cours de l'exécution. Deuxièmement, elle ne représente qu'une seule exécution. Or, nous souhaiterions pouvoir prouver automatiquement des propriétés pour tout ou partie de ses exécutions possibles, et donc tout ou partie des états machines initiaux que l'on pourrait lui fournir. Il est donc nécessaire de construire de nouveaux outils permettant de prendre ces besoins en considération.

Commençons donc par le premier point et construisons la *sémantique de traces* de notre langage, représentant une exécution comme la suite ordonnée des instructions qui sont exécutées, associées aux états machines résultant de leurs exécutions. Pour pouvoir la définir, il nous faut commencer par introduire une manière d'indicer toutes les instructions d'un programme qui ne sont pas des instructions de contrôle, à savoir la séquence, le test et la boucle, de manière unique. Nous utilisons pour cela la notion classique d'*étiquettes*. Nous notons $\text{labels}\,(p)$ l'ensemble des étiquettes d'un programme, $\text{label}\,(s, p)$ l'étiquette de l'instruction s dans le programme p et $\text{last}\,(p)$ l'étiquette de la dernière instruction du programme p. Ces fonctions se définissent facilement récursivement sur le langage. Nous ne présentons pas ici comment les définir plus en détail car cela ne nous est pas nécessaire. En effet, il nous suffit de savoir que chaque instruction à un identifiant unique, peu importe comment ce dernier est construit.

Exemple 3.1.1. Prenons pour exemple le programme suivant :

$$p = \textbf{if } (\phi)\,(x \leq 0) \textbf{ then } x = (\phi)\,(0)\ ; \textbf{ else skip ; end}$$

Une assignation d'étiquettes possible serait de désigner $x = (\phi)\,(0)$; par 1 et **skip ;** par 2. Ainsi donc, nous aurions ici $\text{labels}\,(p) = \{1, 2\}$.

Intéressons nous maintenant à définir la sémantique de traces de notre langage. Elle est noté $\mathscr{T}[\![p]\!]$ et est définie comme une fonction produisant, à partir d'un état machine initial, une liste associant des états machines aux étiquettes des instructions du programme p :

$$\mathscr{T}[\![p]\!] : \Sigma \mapsto [\text{labels}\,(p) \times (\Sigma \uplus \Omega)] \tag{3.1}$$

Sa définition est donnée à la figure 3.1. La liste produite contient les étiquettes de toutes les instructions qui ont été exécutées à partir de l'état machine initial, dans l'ordre inverse dans lequel elles l'ont été. De plus, chaque étiquette est associée à l'état machine résultant de l'exécution de l'instruction associée. Ainsi, parcourir la liste permet de remonter l'exécution du programme en commençant par la fin. Nous utilisons les notations classiques associées aux listes, avec ε désignant la liste vide, x désignant la liste singleton ne contenant que x, et $x \bullet \tau$ désignant l'ajout de l'élément x en tête de la liste τ. Par abus de notation, nous dénotons la concaténation de liste de la même manière.

$$\mathscr{T}[\![s_1\ s_2]\!](\rho) = \begin{cases} \mathscr{T}[\![s_2]\!](\mathcal{S}[\![s_1]\!](\rho)) \bullet (\text{label}\,(s_1, p)\,, \mathcal{S}[\![s_1]\!](\rho)) & \text{si } \mathcal{S}[\![s_1]\!](\rho) \neq \omega \\ (\text{label}\,(s_1, p)\,, \omega) & \text{sinon} \end{cases}$$

$$\mathscr{T}[\![\textbf{if } c \textbf{ then } s_t \textbf{ else } s_e \textbf{ end}]\!](\rho) = \begin{cases} \varepsilon & \text{si } \mathcal{C}[\![c]\!](\rho) = \omega \\ \mathscr{T}[\![s_t]\!](\rho) & \text{si } \mathcal{C}[\![c]\!](\rho) = \text{Vrai} \\ \mathscr{T}[\![s_e]\!](\rho) & \text{si } \mathcal{C}[\![c]\!](\rho) = \text{Faux} \end{cases}$$

$$\mathscr{T}[\![\textbf{while } c \textbf{ do } s \textbf{ done}]\!](\rho) = \text{fix}\,(F)\,(\rho)$$

$$\text{avec } F(f) = \lambda\rho. \begin{cases} \varepsilon & \begin{array}{l}\text{si } \rho = \omega \\ \text{ou } \mathcal{C}[\![c]\!](\rho) = \omega \\ \text{ou } \mathcal{C}[\![c]\!](\rho) = \text{Faux}\end{array} \\ f(\mathcal{S}[\![s]\!](\rho)) \bullet \mathscr{T}[\![s]\!](\rho) & \text{sinon} \end{cases}$$

$$\mathscr{T}[\![s]\!](\rho) = [(\text{label}\,(s, p)\,, \mathcal{S}[\![s]\!](\rho))] \text{ avec } s = x = (\phi)\,(e) \text{ ; ou } s = \textbf{skip ;}$$

FIGURE 3.1 – Sémantique de trace du langage MINIFLOAT.

Les trois instructions de contrôle disposent d'un traitement spécifique. La séquence évalue sa première instruction, puis utilise l'état machine produit par cette

évaluation pour interpréter sa seconde instruction. Cependant, elle ajoute également, en queue de la liste produite, l'étiquette et l'état machine de la première instruction. Si la première instruction lève une exception, alors la liste associée à la construction $s_1\ s_2$ est le singleton ne contenant que l'étiquette de s_1, à laquelle est associée l'état d'exception ω. Le test n'évalue, et n'ajoute à la trace, que la branche correspondant à la valeur de vérité de sa condition dans le contexte. Une liste vide est associée à l'instruction de test si l'évaluation de sa condition lève une exception. La sémantique de trace de la boucle fonctionne d'une manière analogue à celle de la sémantique des instructions décrite au chapitre 2. En effet, elle consiste en un calcul de point fixe sur une fonction d'ordre supérieur, dont le paramètre est une fonction qui décrit, de plus en plus précisément la sémantique du corps de la boucle :

$$F : (\Sigma \uplus \Omega \mapsto [\mathrm{labels}\,(s) \times \Sigma \uplus \Omega]) \mapsto (\Sigma \uplus \Omega \mapsto [\mathrm{labels}\,(s) \times \Sigma \uplus \Omega])$$

L'existence et l'évaluation de ce point fixe sont des conséquences directes du théorème 2.2.3 de Knaster-Tarski. En effet, l'ensemble des traces, équipé de l'ordre partiel spécifiant qu'une liste τ_1 est plus petite qu'une liste τ_2 si cette dernière est de la forme $\tau' \bullet \tau_1$ forme trivialement un domaine, qui peut être étendu, en utilisant l'ordre sur les fonctions énoncé à l'équation 2.35, en domaine sur $(\Sigma \uplus \Omega \mapsto [\mathrm{labels}\,(s) \times (\Sigma \uplus \Omega)])$. Enfin, les instructions d'affectation et d'identité sont, aux yeux de la sémantique de traces, des feuilles. Il convient donc de leur associer une liste singleton, ne contenant que l'étiquette associée à l'instruction et l'état machine résultant de son exécution.

Abordons maintenant notre second problème, et cherchons à formaliser comment exprimer des propriétés sur de multiples exécutions. Une première possibilité est de construire, pour un ensemble d'états machines donné $X \in \mathcal{P}(\Sigma)$, l'ensemble des traces possibles du programme p. Cependant, cette construction est assez inconfortable. En effet, exprimer la moindre propriété sur l'état machine après l'exécution d'une instruction revient systématiquement à devoir chercher toutes les occurrences de cette instruction dans toutes les traces, avant d'exprimer la propriété sur chacun des états machines associés. Il est donc intéressant d'intégrer ce processus nécessaire directement dans la *sémantique collectrice* de notre langage, représentant l'ensemble des exécutions possibles d'un programme à partir d'un ensemble d'états machines. Elle est notée $\mathscr{C}[\![p]\!]$ et est définie comme une fonction produisant, à partir d'un ensemble d'états machines initiaux, une fonction partielle associant à chaque étiquette du programme l'ensemble des états machines qu'il est possible de voir associé à cette instruction dans au moins une des traces existantes :

$$\mathscr{C}[\![p]\!] : \mathcal{P}(\Sigma) \mapsto (\mathrm{labels}\,(p) \rightharpoonup \mathcal{P}(\Sigma \uplus \Omega)) \tag{3.2}$$

Sa définition découle directement du processus nécessaire à exprimer des propriétés sur un ensemble de traces que nous décrivions en début de paragraphe. En effet, la

fonction partielle associée à un ensemble d'états initiaux X construit, pour toute étiquette s, l'ensemble des états associés à s dans au moins une trace :

$$\mathscr{C}[\![p]\!](X) = \lambda s.\{\sigma \mid \exists \rho \in X, (s,\sigma) \in \mathscr{T}[\![p]\!](\rho)\} \tag{3.3}$$

La sémantique collectrice perd de l'information par rapport à l'ensemble des traces possibles. Il n'est en effet plus possible de connaitre la suite d'instructions ayant conduit à un état machine. Ce n'est cependant pas un problème, car nous ne cherchons pas à retrouver la source des erreurs d'arrondi, mais seulement à prouver qu'elles impactent suffisamment peu le programme pour que nous puissions accorder notre confiance au résultat que ce dernier calcule.

Exemple 3.1.2. Soit le programme p suivant :

$$\begin{array}{ll}
\textbf{if } (\mathbb{F}_m)\,(x \leq 2) & \\
\textbf{then } y = (\mathbb{F}_m)\,(3)\,; & (1)\\
\textbf{else } y = (\mathbb{F}_m)\,(5)\,; & (2)\\
z = (\mathbb{F}_m)\,(y)\,; & (3)
\end{array}$$

dans lequel chaque instruction disposant d'une étiquette est annotée à droite par celle qui lui est associée. Étudions les sémantiques de traces et collectrice de ce programme avec les deux états initiaux $\{x \leftarrow 1\}$ et $\{x \leftarrow 3\}$. Nous disposons des deux traces suivantes :

$$\begin{array}{l}
(3, \{x \leftarrow 1; y \leftarrow 3; z \leftarrow 3\}) \bullet (1, \{x \leftarrow 1; y \leftarrow 3\}) \bullet \varepsilon\\
(3, \{x \leftarrow 1; y \leftarrow 5; z \leftarrow 5\}) \bullet (2, \{x \leftarrow 1; y \leftarrow 5\}) \bullet \varepsilon
\end{array}$$

nous indiquant que pour être dans un état ou $y = 3$ à l'étiquette 3, il faut être passé par l'instruction d'étiquette 1. La sémantique collectrice, quant à elle, produit, à partir d'un état initial $\{\{x \leftarrow 1\}, \{x \leftarrow 3\}\}$, une fonction partielle qui associe les deux états $\{x \leftarrow 1; y \leftarrow 3; z \leftarrow 3\}$ et $\{x \leftarrow 1; y \leftarrow 5; z \leftarrow 5\}$ à l'étiquette 3. Il n'est plus possible de savoir quelles séquences d'instructions permettent de produire ces états.

Cette sémantique collectrice est suffisante car elle contient toutes les informations dont nous avons besoin pour exprimer les propriétés qui nous intéressent. Cependant, elle n'est pas plus calculable que la sémantique des instructions de notre langage, ce qui nous empêche toujours de définir notre analyse. Il est donc nécessaire de la simplifier davantage afin de pouvoir la calculer. Cette simplification ne pourra pas être effectuée sans perte. L'interprétation abstraite offre ainsi

un cadre théorique permettant de définir comment simplifier la sémantique collectrice afin d'obtenir une analyse calculable en temps fini tout en ne manquant aucun mauvais comportement, quitte à ne pas pouvoir toujours prouver que le programme est correct.

3.2 Cadre théorique de l'interprétation abstraite

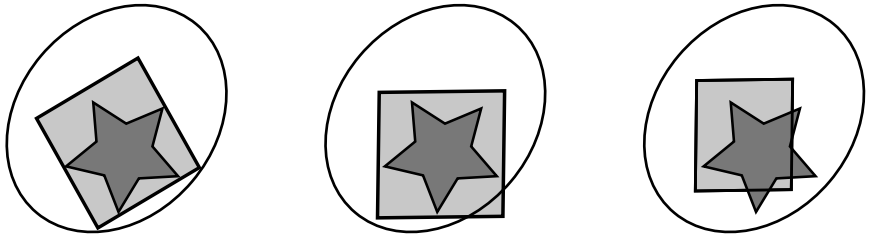
Toutes les sémantiques que nous avons définies jusque ici ont pour but de formaliser les comportements *concrets* présentés par les programmes écrits dans notre langage. L'objectif de l'interprétation abstraite est de fournir un cadre permettant de définir de nouvelles sémantiques calculant des *approximations* de ces comportements, plus simples à manipuler mais ne pouvant pas, en contrepartie, représenter exactement tous les comportements concrets.

La sémantique collectrice que nous avons définie précédemment associe un ensemble d'états machine, ou encore *états concrets*, à chaque étiquette d'un programme. Une sémantique collectrice abstraite, quant à elle, opère sur des états abstraits, qui représentent un ensemble d'états machine. Cette sémantique abstraite, et les états abstraits qu'elle manipule, ne peuvent pas être définis n'importe comment. Le but de la théorie de l'interprétation abstraite est de nous fournir un cadre permettant de garantir qu'une sémantique abstraite soit à la fois calculable en temps fini et *correcte*. On dit d'une sémantique abstraite qu'elle est correcte si elle capture au moins tous les comportements concrets possibles, c'est-à-dire que l'état abstrait produit pour chaque étiquette contient au moins tous les états machine produits par notre sémantique collectrice concrète. La propriété de correction est nécessaire pour pouvoir déduire des propriétés intéressantes sur notre programme. En effet, comme tous les comportements du programme sont représentés par une sémantique abstraite correcte, prouver que l'état abstrait ne contient pas de comportements indésirables permet de déduire qu'il n'y en a pas non plus parmi les états concrets. De la même manière, si l'erreur numérique obtenue dans l'abstrait est bornée, alors elle est effectivement bornée pour toutes les exécutions concrètes.

Cependant, il est important de noter que l'état abstrait produit peut contenir plus de comportements que ceux calculés par la sémantique concrète. C'est grâce à cette relaxation que nous pouvons définir une sémantique abstraite calculable, mais cela implique que nous ne pourrons pas toujours prouver certaines propriétés. Par exemple, si l'état abstrait contient une exception e, nous ne pouvons pas en déduire qu'il existe une exécution du programme levant e, car sa présence dans l'abstrait pourrait être uniquement due à une sur-approximation de l'analyse. Une

analyse par interprétation abstraite est donc amenée à devoir répondre « je ne sais pas » de temps en temps.

La figure 3.2 est une illustration de ces différents phénomènes. Dans les trois illustrations, la grande ellipse blanche correspond aux comportements concrets acceptables et l'étoile grise foncée correspond aux comportements concrets d'un programme. Ces derniers sont tous inclus dans l'ellipse, le programme n'a donc pas de bug. Ces trois illustrations présentent des carrés en gris clair différents, correspondant à une approximation des comportements concrets possibles du programme. Dans les deux premiers cas, l'abstraction est correcte car elle contient



(a) Précise (b) Imprécise (c) Incorrecte

FIGURE 3.2 – Illustration d'une abstraction correcte et précise (a), correcte mais imprécise (b) et incorrecte (c). L'ellipse blanche correspond aux comportements acceptables, l'étoile grise foncée aux états concrets d'un programme et les carrés gris clairs à différentes abstractions.

toute l'étoile. Cependant, comme le carré de la première est incluse dans l'ellipse, nous pouvons en déduire que le programme n'a pas de bug, alors que le carré de la seconde n'est pas inclus, ce qui ne permet aucune déduction. Dans le dernier cas, l'étoile n'est pas incluse dans le carré, ce qui signifie que l'analyse n'est pas correcte. Bien que son résultat soit inclus dans les comportements acceptables, il n'est pas possible de déduire que le programme n'a pas de bug car l'analyse aurait pu manquer un des comportements du programme.

3.2.1 Domaine abstrait

Comme nous l'avons dit précédemment, la sémantique abstraite manipule des états abstraits représentant une approximation des états concrets. Cependant, il nous faut pouvoir, d'une certaine manière, relier les états concrets et les états abstraits. En particulier, un état concret contenant plus d'informations doit nécessairement être représenté par un état abstrait approximant plus d'informations. Dit autrement, nous avons besoin d'ordres partiels sur les états concrets et sur les états abstraits, modélisant la quantité d'informations représentées. Ces structures d'ensembles partiellement ordonnés sont appelées *domaine*. Pour notre langage,

le domaine concret est composé d'ensembles d'états machines et utilise l'inclusion ensembliste comme ordre partiel. Cependant, par soucis de généralité, les présentations qui suivent sont faites en utilisant un domaine concret générique, noté $(C, \leq)$. Le domaine abstrait, quant à lui, est noté $(A, \sqsubseteq)$, où A est un ensemble et $\sqsubseteq$ une relation formant un ordre partiel sur A. Il nous faut également un lien entre le monde concret et l'abstrait, afin de pouvoir spécifier ce qu'est une bonne approximation. Cette connexion minimale est représentée par une fonction de *concrétisation*, traditionnellement [18] notée γ.

Définition 3.2.1. Une fonction de concrétisation $\gamma : (A, \sqsubseteq) \mapsto (C, \leq)$ est une fonction monotone permettant d'obtenir l'ensemble des états concrets représenté par un état abstrait.

La monotonie garantit simplement que de plus petits états abstraits, au sens de $\sqsubseteq$, représentent bien de plus petits états concrets au sens de $\leq$. Nous pouvons désormais formaliser la notion de correction évoquée précédemment :

Définition 3.2.2. Un état abstrait $a \in A$ est une abstraction *correcte* d'un état concret $c \in C$ si $c \leq \gamma(a)$. De plus, cette abstraction est *exacte* si $c = \gamma(a)$.

Enfin, nous pouvons également définir formellement ce que signifie le fait de prouver qu'une propriété soit respectée en utilisant une analyse par interprétation abstraite. Soit P l'ensemble des états machine respectant une propriété donnée, c'est-à-dire une représentation ensembliste de ce que cette propriété signifie. Comme P est un ensemble d'états machine, donc un état concret, il est possible de l'abstraire par un état $P^\sharp$. Soit $R^\sharp$ un état abstrait correct obtenu par le biais d'une analyse par interprétation abstraite d'un programme quelconque. Alors, si $P^\sharp$ est une abstraction exacte de P, c'est-à-dire que notre propriété est parfaitement représentable dans l'abstrait, et qu'il est possible de prouver que $R^\sharp \sqsubseteq P^\sharp$, nous pouvons déduire que le programme satisfait la propriété dans le concret, c'est-à-dire que $R \leq P$. C'est ce que nous illustrions déjà à la figure 3.2 : si l'ellipse, qui représente l'ensemble des comportements acceptables, c'est-à-dire la propriété que l'on cherche à démontrer, est représentable dans l'abstrait, alors il suffit de vérifier que le carré gris clair représentant l'abstraction des comportements concrets est plus petite que l'ellipse pour prouver que tous les comportements concrets du programme, illustrés par l'étoile grise foncée, sont correct.

Une chose importante de noter ici que le calcul de $R^\sharp$ et la vérification sont faits entièrement dans l'abstrait. La précision et l'expressivité de l'abstraction utilisée sont donc des points clé pour pouvoir déduire des propriétés intéressantes. Cependant, l'efficacité, au sens algorithmique, de cette abstraction est également cruciale afin de pouvoir analyser des programmes conséquents. Ainsi, la création d'une abstraction est toujours un compromis entre précision et efficacité. Le cadre

minimal que nous venons de fournir n'est donc pas suffisant. En effet, il permet de déterminer la correction d'une analyse, mais ne nous apporte aucune aide pour évaluer sa précision ou son efficacité. Le second point est en dehors du cadre de cette section, mais nous pouvons adresser le premier. Pour cela, avant de présenter comment abstraire la sémantique de notre langage, nous allons commencer par présenter rapidement la notion de *correspondance de Galois*.

3.2.2 Correspondance de Galois

La fonction de concrétisation ne représente qu'une faible connexion entre les mondes concrets et abstraits. Nous pouvons cependant requérir l'existence d'un lien de plus, une seconde fonction monotone appelée la *fonction d'abstraction*. Elle est traditionnellement [18] notée α, et associe un état abstrait à tout état concret. Nous pouvons également demander à ce que la paire (α, γ) forme une *correspondance de Galois* :

Définition 3.2.3. Pour $(C, \leq)$ et $(A, \sqsubseteq)$ deux ensembles partiellement ordonnés, la paire $(\alpha : C \mapsto A, \gamma : A \mapsto C)$ est une *correspondance de Galois* si et seulement si :

$$\forall a \in A, c \in C, c \leq \gamma(a) \iff \alpha(c) \sqsubseteq a \tag{3.4}$$

Cette définition donne un lien très fort entre les fonctions d'abstraction et de concrétisation. Cependant, une seconde caractérisation permet d'apporter un éclairage plus intuitif sur ce lien :

Définition 3.2.4. La paire (α, γ) est une correspondance de Galois si et seulement si elle respecte les deux propriétés suivantes :

1. $\gamma \circ \alpha$ est extensive, c'est-à-dire que $\forall c \in C, c \leq \gamma(\alpha(c))$;
2. $\alpha \circ \gamma$ est réductrice, c'est-à-dire que $\forall a \in A, \alpha(\gamma(a)) \sqsubseteq a$.

La première propriété stipule que faire un aller-retour en partant du monde concret peut produire un état moins précis que celui de départ, ce qui fait sens car un état concret n'a pas forcément de représentation exacte dans le monde abstrait. La seconde propriété stipule, quant à elle, que faire un aller-retour en partant de l'abstrait peut améliorer la précision, ce qui n'arrive en pratique que si un état concret à plusieurs représentations possibles dans l'abstrait. Cette particularité ne nous impactera pas par la suite. L'équivalence entre ces deux définitions se démontre facilement. Un lecteur curieux peut en apprendre plus en lisant [71].

La figure 3.3 représente une illustration d'une correspondance de Galois entre un domaine concret $(C, \leq)$ et un domaine abstrait $(A, \sqsubseteq)$. Les points c_1, c_2 et c_3 représentent trois états concrets, et les points a_1 et a_2 deux états abstraits. Ces points sont mis en relation de différentes manières, chacune illustrée par une flèche.

Deux points d'un même domaine sont en relation si l'un est plus petit que l'autre.

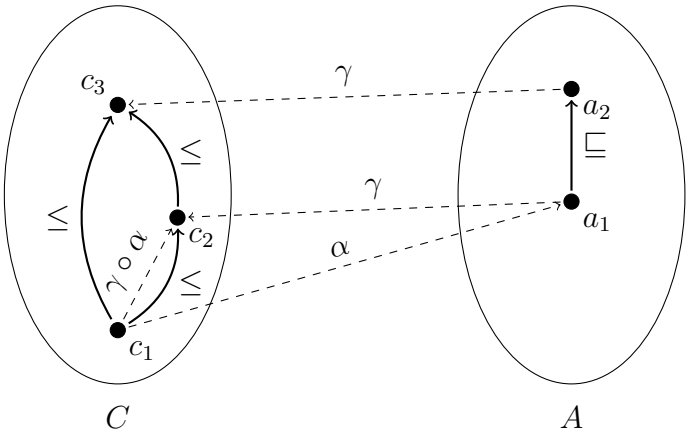


FIGURE 3.3 – Illustration d'une correspondance de Galois.

Ainsi, nous avons $c_1 \leq c_2 \leq c_3$ et $a_1 \sqsubseteq a_2$. Deux points de domaines différents sont en relation si l'un peut être construit à partir de l'autre, au travers des fonctions d'abstraction et de concrétisation. Ainsi, nous avons $a_1 = \alpha(c_1)$, $c_2 = \gamma(a_1)$ et $c_3 = \gamma(c_2)$. Le chemin partant de c_1 pour atteindre c_2 en passant par a_1 est ainsi une illustration du fait que $\gamma \circ \alpha$ est extensive. De la même manière, le chemin c_1, a_1, a_2, c_3 est une illustration de la définition 3.2.3.

Avoir une correspondance de Galois offre de nombreuses propriétés intéressantes. Cependant, avant de les présenter, il nous faut faire un point sur les notations. Par la suite, nous utiliserons $\cup X$ et $\cap X$ pour dénoter le supremum et l'infimum d'un ensemble d'*états concrets*, et nous utiliserons $\sqcup A$ et $\sqcap A$ pour dénoter le supremum et l'infimum d'un ensemble d'*états abstraits*. De plus, par abus de notation, si A ne contient que deux états abstraits a_1 et a_2, alors nous dénoterons $a_1 \sqcup a_2$ et $a_1 \sqcap a_2$ son supremum et son infimum. Nous ferons de même avec les opérateurs concrets.

Théorème 3.2.1. Si (α, γ) forme une correspondance de Galois entre un $(C, \leq)$ et $(A, \sqsubseteq)$, alors :

1. $\gamma \circ \alpha \circ \gamma = \gamma$ et $\alpha \circ \gamma \circ \alpha = \alpha$;
2. $\gamma \circ \alpha$ et $\alpha \circ \gamma$ sont idempotents, c'est-à-dire que pour tout $n \geq 1$, $(\gamma \circ \alpha)^n = \gamma \circ \alpha$ et $(\alpha \circ \gamma)^n = \alpha \circ \gamma$;
3. $\forall c \in C, \alpha(c) = \sqcap\{a \mid c \leq \gamma(a)\}$;
4. $\forall a \in A, \gamma(a) = \cup\{c \mid \alpha(c) \sqsubseteq a\}$;
5. α associe les supremums abstraits aux supremums concrets :
   $\forall X \subseteq C$, si $\cup X$ existe, alors $\alpha(\cup X) = \sqcup\{\alpha(x) \mid x \in X\}$

6. γ associe les infimums concrets aux infimums abstraits :
   $\forall X \subseteq A, \text{si } \sqcap X \text{ existe, alors } \gamma(\sqcap X) = \cap\{\gamma(x) \mid x \in X\}$

Démonstration. Nous présentons une idée de la preuve point par point :

1. Soit $a \in A$. En appliquant le premier point de la définition 3.2.4 à $\gamma(a) \in C$, nous avons $\gamma(a) \leq (\gamma \circ \alpha \circ \gamma)(a)$. De plus, en partant du second point de cette définition et en utilisant la monotonie de γ, nous avons également $(\gamma \circ \alpha \circ \gamma)(a) \leq \gamma(a)$. Ainsi, nous avons bien $\gamma \circ \alpha \circ \gamma = \gamma$. La preuve pour $\alpha \circ \gamma \circ \alpha$ utilise le même raisonnement.
2. La preuve réutilise le même procédé que pour le point précédent afin d'encadrer, pour tout $c \in C$, $(\gamma \circ \alpha)(c)$ par $(\gamma \circ \alpha \circ \gamma \circ \alpha)(c)$, et pour tout $a \in A$, $(\alpha \circ \gamma)(a)$ par $(\alpha \circ \gamma \circ \alpha \circ \gamma)(a)$. La généralisation pour tout n est faite facilement par récurrence.
3. Le fait que $\alpha(c)$ appartient bien à $\{a \mid c \leq \gamma(a)\}$ découle directement du premier point de la définition 3.2.4. Il nous faut maintenant montrer que $\alpha(c)$ est le plus petit des éléments de $\{a \mid c \leq \gamma(a)\}$ pour conclure, c'est-à-dire que $\forall x \in \{a \mid c \leq \gamma(a)\}, \alpha(c) \sqsubseteq x$. Ceci découle trivialement de la monotonie de α et du deuxième point de la définition 3.2.4.
4. La preuve réutilise le même raisonnement que celle du point précédent.
5. Soit $X \subseteq C$. Commençons par prouver que $\alpha(\cup X)$ est bien un majorant de l'ensemble $\{\alpha(x) \mid x \in X\}$. Ceci est équivalent à démontrer que pour tout $x \in X$, on a bien $\alpha(x) \sqsubseteq \alpha(\cup X)$, ce qui découle trivialement du fait que $\cup X$ est majorant de X et de la monotonie de α. Prouvons maintenant que $\alpha(\cup X)$ est bien le plus petit majorant. Dit autrement, nous voulons démontrer que :

   $$\forall y \in X, (\forall x \in X, \alpha(x) \sqsubseteq \alpha(y)) \implies \alpha(\cup X) \sqsubseteq \alpha(y)$$

   Soit $y \in X$. Nous commençons par utiliser la monotonie de γ et le premier point de la définition 3.2.4 pour déduire de la prémisse que $\forall x \in X, x \leq (\gamma \circ \alpha)(y)$. Ensuite, la définition du supremum induit que $\cup X \leq (\gamma \circ \alpha)(y)$. Enfin, la monotonie de α et la première propriété énoncée dans ce théorème nous permettent de conclure.
6. La preuve réutilise un raisonnement similaire à celui du point précédent.

□

Ce théorème nous permet de voir qu'une correspondance de Galois nous offre des propriétés intéressantes sur le lien entre l'abstrait et le concret. Tout d'abord, les points 1 et 2 permettent de montrer que, même si un aller-retour à partir du concret peut nous faire perdre en précision, en faire plusieurs ne changera plus rien. Ensuite, les points 3 et 4 montrent qu'il est possible de construire l'une des

deux fonctions à partir de l'autre. De plus, le point 3 nous garantit que pour tout $c \in C$, $\alpha(c)$ est la meilleure abstraction correcte possible. Enfin, les deux derniers points permettent de mettre en relation les mondes concrets et abstraits si nous disposons de structures plus riches qu'un ordre partiel.

Ainsi, nous disposons désormais d'un cadre formel permettant de parler d'abstraction, au travers de la fonction de concrétisation, et de précision, au travers de la correspondance de Galois. D'autres propriétés et définitions peuvent par ailleurs découler de cette notion, elles ne nous seront cependant pas utiles. Par contre, il nous manque encore des outils formels pour pouvoir décrire la relation entre les constructions de notre langage et leurs nécessaires équivalents dans le monde abstrait.

3.2.3 Opérateurs abstraits

Exactement comme l'abstraction des états concrets doit garantir de ne manquer aucun comportement, il est nécessaire que tous les opérateurs utilisés dans la sémantique concrète dispose d'un équivalent dans l'abstrait qui garantisse le même genre de propriété. Cette notion d'*opérateur correct* découle logiquement de la notion d'abstraction correcte, et ne nécessite pas de correspondance de Galois pour être définie :

Définition 3.2.5. Soit γ une fonction de concrétisation entre un domaine abstrait $(A, \sqsubseteq)$ et un domaine concret $(C, \leq)$. L'opérateur $f^\sharp$ est une approximation *correcte* de l'opérateur $f : C \mapsto C$ si $\forall a \in A, f(\gamma(a)) \leq \gamma(f^\sharp(a))$. Comme précédemment, $f^\sharp$ est une approximation *exacte* de f dans le cas où nous avons l'égalité.

L'intuition ici, illustrée à la figure 3.4, est que l'opérateur abstrait doit produire, pour toute état abstrait, une abstraction contenant plus de comportements que l'application de l'opérateur concret sur la concrétisation de l'état abstrait.

Dans le cas où nous avons une correspondance de Galois, nous pouvons de plus définir une notion de *meilleure abstraction* pour les opérateurs :

Définition 3.2.6. Un opérateur abstrait correct ${f_1}^\sharp$ est une *meilleure abstraction* d'un opérateur concret f qu'un autre opérateur abstrait correct ${f_2}^\sharp$ si et seulement si, pour tout $a \in A$, nous avons ${f_1}^\sharp(a) \sqsubseteq {f_2}^\sharp(a)$.

Théorème 3.2.2. Si (α, γ) est une correspondance de Galois pour $(C, \leq)$ et $(A, \sqsubseteq)$ alors, pour tout opérateur $f : C \mapsto C$, la *meilleure abstraction* de f est la fonction $\alpha \circ f \circ \gamma$.

Démonstration. La définition 3.2.5 de correction, indiquant que $\forall a \in A, (f \circ \gamma)(a) \leq (\gamma \circ f^\sharp)(a)$, peut être réécrite en utilisant les propriétés des correspondances de Galois comme $\forall a \in A, (\alpha \circ f \circ \gamma)(a) \sqsubseteq (\alpha \circ \gamma \circ f^\sharp)(a) \sqsubseteq f^\sharp(a)$. Nous

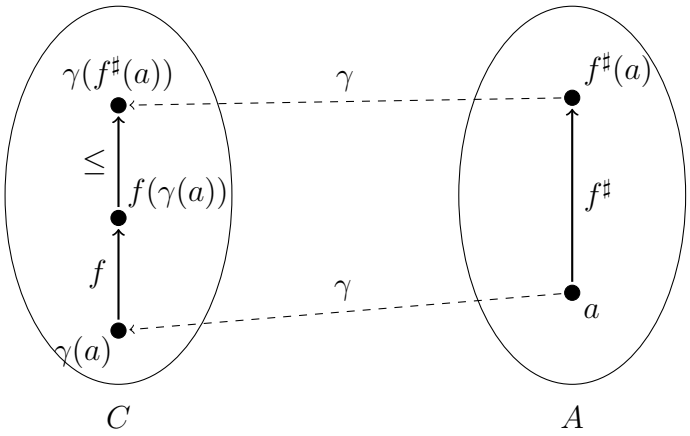


FIGURE 3.4 – Illustration d'un opérateur correct.

remarquons ainsi que toute abstraction correcte de f est nécessairement moins précise que $\alpha \circ f \circ \gamma$. □

Il est important de noter que, bien que la meilleure abstraction donnée par le théorème 3.2.2 permette de raisonner sur la précision d'une analyse par interprétation abstraite, et en particulier sur la précision maximale qu'il est possible d'atteindre, elle ne permet pas en général de définir comment implémenter un opérateur abstrait. En effet, l'objectif de l'implémentation est de ne pas retourner dans le monde concret. Or c'est la première chose qu'il est nécessaire d'effectuer dans la définition de la meilleure abstraction. Ainsi, ce théorème sert principalement à vérifier si l'implémentation d'un opérateur abstrait est la plus précise possible.

Pour terminer, il faut nous arrêter un instant sur la composition d'opérateurs abstraits. En effet, comme nous l'avons vu au chapitre précédent, la sémantique concrète de notre langage est obtenue en combinant des fonctions sémantiques atomiques. Il est donc intéressant que nous réutilisions ce schéma pour l'abstraction, afin de pouvoir combiner des opérateurs abstraits atomiques. Il nous faut cependant vérifier pour cela que la composition d'opérateurs corrects est elle-même correcte :

Théorème 3.2.3. Soient f et g deux opérateurs concrets, et $f^\sharp, g^\sharp$ les abstractions correctes respectives de ces opérateurs. Alors :

1. Si f est monotone, alors $f^\sharp \circ g^\sharp$ est une abstraction correcte de $f \circ g$;
2. Si $f^\sharp$ et $g^\sharp$ sont des abstractions *exactes* de f et g respectivement, alors $f^\sharp \circ g^\sharp$ est une approximation exacte de $f \circ g$.

Démonstration. Soit $(A, \sqsubseteq)$ un domaine abstrait et soit $a \in A$. Commençons par le premier point. Comme $g^\sharp$ est une abstraction correcte de g, et que f est monotone,

nous avons $f(g(\gamma(a))) \leq f(\gamma(g^\sharp(a)))$. Or, $f^\sharp$ est une abstraction correcte de f. En particulier pour $g^\sharp(a)$, nous avons $f(\gamma(g^\sharp(a))) \leq \gamma(f^\sharp(g^\sharp(a)))$, ce qui conclut la preuve. La preuve pour le second point suit le même raisonnement, mais ne nécessite pas la monotonie de f car elle repose sur des égalités. □

La condition de monotonie sur f n'est en pratique pas un problème car elle ne concerne que le monde concret, pour lequel nous avons montré au chapitre précédent la monotonie de toutes les constructions de notre langage.

La composition d'opérateurs conserve ainsi la correction, mais pas nécessairement l'optimalité. En effet, la composition des meilleures approximations de f et g est la fonction $\alpha \circ f \circ \gamma \circ \alpha \circ g \circ \gamma$. Nous remarquons que nous sommes amenés ici à faire un aller-retour de trop entre le concret et l'abstrait, représenté par le $\gamma \circ \alpha$ au milieu, ce qui peut être une source d'imprécision. En pratique, cette limitation peut être contournée en implémentant des opérateurs abstraits spécialisés pour les schémas revenant le plus souvent.

3.2.4 Points fixes et élargissement

La dernière brique formelle manquante pour pouvoir définir des sémantiques abstraites à notre langage concerne la gestion des boucles, et plus particulièrement l'approximation du plus petit point fixe utilisée pour la sémantique de cette instruction. Il existe plusieurs méthodes permettant, à partir d'une approximation correcte $f^\sharp$ de f, de calculer une approximation correcte de $\mathrm{fix}\,(f)$. Nous n'en présentons ici qu'une seule, classique, qui nous est suffisante par la suite. Cependant, nous avons besoin d'une structure plus riche que les cpos que nous avions utilisé au chapitre précédent. Il nous faut donc commencer par l'introduire.

Définition 3.2.7. Un ensemble partiellement ordonné $(C, \leq)$ est un *treillis complet* si et seulement si toute sous partie $A \subseteq C$ dispose d'un supremum et d'un infimum dans C.

Un treillis complet dispose forcément d'un plus grand élément, à savoir le supremum de C tout entier, et d'un plus petit élément, à savoir l'infimum de l'ensemble vide. Ils sont notés respectivement$\top$ et $\bot$.

Théorème 3.2.4. Si $f : C \mapsto C$ est une fonction monotone dans un treillis complet $(C, \leq)$, et que $f^\sharp : A \mapsto A$ est une approximation correcte de f dans un ensemble partiellement ordonné $(A, \sqsubseteq)$, alors le supremum de la suite $\{f^{\sharp i}(\bot^\sharp)\}_{i\in\mathbb{N}}$, s'il existe, est une approximation correcte de $\mathrm{fix}\,(f)$.

Démonstration. La démonstration repose sur le fait que, pour tout $i \in \mathbb{N}$, $f^{\sharp i}(\bot^\sharp)$ est une approximation correcte de $f^i(\bot)$. Ceci peut se démontrer trivialement par

récurrence en utilisant le fait que $f^\sharp$ et $\bot^\sharp$ sont des approximations correctes de f et $\bot$. □

L'idée derrière ce théorème est de reprendre le processus de construction du plus petit point fixe du théorème 2.2.3 de Knaster-Tarski. Il est cependant important de noter que le théorème 3.2.4 nous garantit la correction, mais pas l'optimalité, et ce même si $f^\sharp$ est la meilleure abstraction de f. En effet, l'approximation de fix (f) passe ici par de multiples compositions de $f^\sharp$, et nous avons montré précédemment que la composition de meilleures abstractions n'est pas forcément la meilleure abstraction de la composition. De plus, ce théorème ne nous fournit pas non plus de garantie de terminaison. En effet, la suite des puissances itérées de $f^\sharp$ n'est pas nécessairement stable. Et, même s'il existe, le supremum peut être difficile à calculer. Il existe plusieurs méthodes dans la littérature pour résoudre cette contrainte de convergence. Nous utiliserons ici la solution standard, introduite par Cousot et Cousot [18], consistant à utiliser un opérateur d'*élargissement* (*widening* en anglais) :

Définition 3.2.8. Un opérateur $\nabla : A \times A \mapsto A$ est un opérateur d'élargissement pour le domaine abstrait $(A, \sqsubseteq)$ si :

1. il calcule une borne supérieure : $\forall x, y \in A, x \sqsubseteq x \nabla y \wedge y \sqsubseteq x \nabla y$;
2. il converge en un nombre fini d'étapes : pour toute suite $\{y_i\}_{i\in\mathbb{N}}$ dans A, la suite $\{x_i\}_{i\in\mathbb{N}}$ définie comme $x_0 = y_0$, $x_{i+1} = x_i \nabla y_{i+1}$ est telle qu'il existe un $k \geq 0$ tel que $x_{k+1} = x_k$.

Aucune propriété de symétrie n'est attendu d'un opérateur d'élargissement.

Cette définition permet de construire des suites croissantes en itérant des opérateurs abstraits qui ne sont pas nécessairement monotones, et garantit que ces suites sont convergentes, et ce même si le domaine A contient des chaînes infinies. Un exemple naïf, utilisable pour n'importe quel domaine contenant un plus grand élément $\top$, est de retourner $\top$ dès que l'itération n'est pas stable :

$$x \nabla y \triangleq \begin{cases} x & \text{si } y \sqsubseteq x \\ \top & \text{sinon} \end{cases} \tag{3.5}$$

Nous verrons d'autres exemples plus intéressants en pratique par la suite.

Cet opérateur va nous permettre de calculer en un temps fini une approximation correcte de plus petits point fixes :

Théorème 3.2.5. Si $f : C \mapsto C$ est une fonction monotone dans un treillis complet concret $(C, \leq)$, et que $f^\sharp : A \mapsto A$ est une approximation correcte de f

dans le domaine abstrait $(A, \sqsubseteq)$ disposant d'un plus petit élément $\bot^\sharp$, alors la suite suivante :

$$\begin{aligned} x_0 &\triangleq \bot^\sharp \\ x_{i+1} &\triangleq x_i \nabla f^\sharp(x_i) \end{aligned} \tag{3.6}$$

converge en temps fini, et son supremum x est une approximation correcte de $\mathrm{fix}\,(f)$.

Démonstration. Il nous suffit de montrer que, pour tout $i \in \mathbb{N}$, nous avons $f^i(\bot) \leq \gamma(x_i)$. Le cas de base $f^0(\bot) = \bot = \gamma(x_0)$ découle trivialement de la définition de f^0 et de x_0. Supposons maintenant que $f^i(\bot) \leq \gamma(x_i)$ et montrons que la propriété est valide au rang $i+1$. Par monotonie de f et la définition 3.2.5 d'opérateur correct, nous avons $f(f^i(\bot)) \leq f(\gamma(x_i)) \leq \gamma(f^\sharp(x_i))$. Or, par définition de la suite x_i et des opérateurs d'élargissement, ainsi que par monotonie de γ, nous avons $\gamma(f^\sharp(x_i)) \leq \gamma(x_i \nabla f^\sharp(x_i)) = \gamma(x_{i+1})$, ce qui conclut la démonstration. □

Ainsi, nous pouvons construire une approximation correcte du point fixe de toute fonction monotone en temps fini, et disposons ainsi de tous les outils pour définir une sémantique abstraite à notre langage. Il est important de noter que le théorème 3.2.5 peut également être démontré en relâchant la contrainte de treillis complet, et en exigeant à la place que f soit continue, c'est-à-dire qu'elle conserve les supremums de toutes les sous parties totalement ordonnées de C. Dans notre contexte, les deux voies reviennent au même car notre domaine concret est un treillis complet, et donc un domaine de Scott, et que les fonctions dont nous cherchons à abstraire les points fixes sont continues, et donc monotones.

Pour terminer, notons que l'utilisation d'un opérateur d'élargissement n'est qu'une des approches possibles pour traiter les boucles. L'état de l'art propose d'autres méthodes, comme par exemple la synthèse d'invariants déductifs par subdivision [48], ou encore en commençant par calculer la matrice jacobienne décrivant les dérivées de la sémantique du corps de la boucle étudiée avant d'étudier sa stabilité en utilisant les exposants de Lyapunov [56]. Cependant, bien que ces approches soient très intéressantes, nous ne les avons pas étudié plus en détail au cours de ses travaux. Ainsi, lorsque nous donnerons, au chapitre 4, les définitions des deux domaines abstraits que nous avons utilisé, nous nous concentrerons sur leurs opérateurs d'élargissement.

3.2.5 Produit réduit

Les outils que nous avons présenté jusque là permettent de définir des domaines et des opérateurs abstraits corrects vis-à-vis des comportements concrets de notre

langage. Cependant, comme nous l'avons déjà dit, un état abstrait perd nécessairement de l'information par rapport à l'état concret qu'il représente. Vu dans l'autre sens, un domaine abstrait dispose d'une certaine *expressivité*, permettant de tenir compte de certaines caractéristiques de la sémantique concrète. Naturellement, différents domaines présenteront différentes expressivités, et il pourrait être intéressant de combiner ces expressivités, afin d'améliorer nos analyses. L'interprétation abstraite dispose d'un outil générique pour cela, à savoir le *produit réduit*, permettant de construire un nouveau domaine à partir de deux autres.

Théorème 3.2.6. Soit deux domaines abstraits, $(A_1, \sqsubseteq_1)$ et $(A_2, \sqsubseteq_2)$, disposant chacun d'une fonction de concrétisation, que nous dénotons γ_1 et γ_2. L'ensemble $A_1 \times A_2$ associé à la relation $\sqsubseteq_{1\times2}$ et la fonction de concrétisation $\gamma_{1\times2}$ définies comme :

$$\gamma_{1\times2}(a_1, a_2) = \gamma_1(a_1) \cap \gamma_2(a_2) \tag{3.7}$$

$$(a_1, a_2) \sqsubseteq_{1\times2} (b_1, b_2) \iff a_1 \sqsubseteq_1 b_1 \wedge a_2 \sqsubseteq_2 b_2 \tag{3.8}$$

est un domaine abstrait. De plus, si $(A_1, \sqsubseteq_1)$ et $(A_2, \sqsubseteq_2)$ disposent tous deux d'une abstraction correcte pour un opérateur f quelconque, alors l'opérateur $f^\sharp_{1\times2}$ définie comme

$$f^\sharp_{1\times2}(a_1, a_2) = \Omega(f^\sharp_1(a_1), f^\sharp_2(a_2)) \tag{3.9}$$

est une abstraction correcte de f, avec $\Omega : A_1 \times A_2 \mapsto A_1 \times A_2$ une *fonction de réduction* telle que :

$$(y_1, y_2) = \Omega(x_1, x_2) \implies \begin{cases} \gamma_{1\times2}(y_1, y_2) = \gamma_{1\times2}(x_1, x_2) \\ \gamma_1(y_1) \subseteq \gamma_1(x_1) \\ \gamma_2(y_2) \subseteq \gamma_2(x_2) \end{cases} \tag{3.10}$$

Ce domaine est appelé le *produit réduit* de $(A_1, \sqsubseteq_1)$ et $(A_2, \sqsubseteq_2)$.

Démonstration. Prouver que $\sqsubseteq_{1\times2}$ forme un ordre partiel sur $A_1 \times A_2$ et que $\gamma_{1\times2}$ est monotone est triviale. La correction de $f^\sharp_{1\times2}$ découle, quant à elle, directement des caractéristiques de la fonction de réduction. □

Le produit réduit de deux domaines dispose ainsi de toutes les informations qu'auraient acquis chacun d'entre eux, c'est-à-dire qu'il représente la conjonction des expressivités. Ainsi, il est naturel de définir sa fonction de concrétisation comme l'intersection des concrétisations pour chaque domaine car cela représente correctement cette idée de recoupement d'informations. De la même manière, l'ordre partiel $\sqsubseteq_{1\times2}$ peut se comprendre comme suit : un état abstrait est moins précis qu'un autre si et seulement si il l'est pour les deux domaines sous-jacents. Par ailleurs, nous pouvons aisément remarquer que si $(A_1, \sqsubseteq_1)$ et $(A_2, \sqsubseteq_2)$ disposent

tous deux d'un plus petit élément, dénotons les $\bot_1$ et $\bot_2$, alors le plus petit élément de leur produit réduit est la paire $(\bot_1, \bot_2)$. Il en va de même pour le plus grand élément. Enfin, l'abstraction d'un opérateur quelconque dans le produit réduit consiste simplement à calculer cette opération dans les deux domains sous-jacents et à mettre en commun les informations, ce qui est représenté par la fonction de réduction. Cette dernière doit être fourni lors de la construction du produit réduit et c'est elle qui permet de gagner en précision en tirant partie de l'expressivité des deux domaines sous-jacents.

Le dernier point qu'il nous faut aborder est à la définition d'un opérateur d'élargissement pour le produit réduit. En effet, nous serions ici également tentés d'utiliser la fonction de réduction pour améliorer notre analyse à chaque itération du processus. Cependant, faire cela ne permet plus de garantir la convergence. Ainsi, l'opérateur d'élargissement $\nabla_{1\times 2}$ est simplement défini comme

$$(a_1, a_2)\nabla_{1\times 2}(b_1, b_2) = (a_1\nabla_1 b_1, a_2\nabla_2 b_2) \tag{3.11}$$

Ses propriétés de convergence découlent directement de celles de opérateurs d'élargissement des domaines sous-jacents.

3.3 Sémantique abstraite

Nous disposons désormais de tous les outils formels dont nous aurons besoin pour définir une sémantique abstraite à notre langage. Cette sémantique est paramétrée par le domaine abstrait utilisé, nous permettant ainsi de définir plusieurs analyses sans reprendre le travail du début. Nous commençons en premier lieu par définir l'ensemble des opérateurs abstraits qu'un domaine abstrait doit fournir pour pouvoir être utilisé dans notre sémantique, avant de présenter en second lieu notre sémantique abstraite. Cette présentation est rapide, car elle est classique et suit le même fil que la présentation de la sémantique concrète.

Définition 3.3.1. Pour qu'un domaine abstrait $(A, \sqsubseteq)$ soit utilisable dans notre sémantique, il doit posséder :

- une fonction de concrétisation monotone $\gamma : A \mapsto \mathcal{P}(\Sigma)$;
- un plus petit élément $\bot^\sharp$ représentant l'ensemble vide ;
- un plus grand élément $\top^\sharp$ représentant l'ensemble Σ complet ;
- des opérateurs abstraits corrects, au sens de la définition 3.2.5, pour l'affectation, ainsi que pour le supremum et l'infimum ;
- un opérateur abstrait correct permettant de *réduire* un état abstrait vis-à-vis d'une comparaison flottante, c'est-à-dire un opérateur

$$(\phi)\,(e_1 \leq^\sharp e_2) : A \mapsto A$$

tel que, pour toutes expressions e_1 et e_2, tout format ϕ, tout état abstrait $\rho^\sharp \in A$ et tout état concret $\rho \in \gamma(\rho^\sharp)$ tel que $\mathcal{C}[\![(\phi)\,(e_1 \leq e_2)]\!](\rho)$ est vrai, nous avons $\rho \in \gamma((\phi)\,(e_1 \leq^\sharp e_2)(\rho^\sharp))$;
— un opérateur d'élargissement ∇ respectant la définition 3.2.8 ;
— (optionnel) une fonction d'abstraction monotone α telle que (α, γ) soit une correspondance de Galois.

Plusieurs choses sont à noter dans cette définition. Tout d'abord, il est évident que tous les éléments abstraits doivent disposer d'une représentation informatique, et que tous les opérateurs abstraits doivent être algorithmiquement les plus efficaces possibles. Cela veut dire que définir un domaine abstrait nécessite donc de trouver un équilibre entre efficacité et expressivité. Ensuite, la définition n'exige aucune autre structure que celle de domaine abstrait, disposant donc d'un ordre partiel et d'une fonction de concrétisation, ce qui nous donne une grande latitude. Également, le domaine abstrait n'a pas à représenter les exceptions. Nous considérons que ces dernières sont gérées au travers d'un système d'*alarmes*, indépendant du domaine, dont l'objectif est, de manière simplifié, de prévenir d'un potentiel bug et de continuer l'analyse en supposant être face à un faux positif. Ce genre de mécanisme est présent dans la plupart des analyseurs par interprétation abstraite modernes, comme *Astrée* [8], *Frama-C/Eva* [9], ou encore *Mopsa* [61]. Enfin, l'abstraction de la comparaison flottante ne cherche pas à retourner une valeur de vérité abstraite sur la comparaison, mais plutôt à filtrer de l'état abstrait un maximum d'états ne validant pas la comparaison. Cela revient à dire que, pour tout $X \in \mathcal{P}(\Sigma)$ et tout $\rho^\sharp \in A$ tels que $\rho^\sharp$ est une abstraction correcte de X, l'état abstrait $(e_1 \leq^\sharp e_2)(\rho^\sharp)$ doit être une abstraction correcte de l'ensemble $\{x \in X \mid \mathcal{C}[\![e_1 \leq e_2]\!](x) = \text{Vrai}\}$ des états concrets validant la condition $e_1 \leq e_2$. Cette gestion de la comparaison et des conditions, différentes de la sémantique concrète, nous est utile, par exemple, pour interpréter, dans l'abstrait, les tests. En effet, comme nous le verrons par la suite, l'interprétation des tests consiste à évaluer chaque branche dans l'abstrait en utilisant un état abstrait réduit par la condition (ou sa négation pour la branche **else**), avant de calculer le supremum dans l'abstrait. Par ailleurs, dans le cadre de notre analyse, notre domaine concret est ordonné par inclusion, et donc le supremum et l'infimum correspondent respectivement à l'union et à l'intersection ensembliste.

Nous pouvons désormais définir la sémantique abstraite de notre langage. Commençons pour cela par présenter comment interpréter les opérateurs logiques du langage dans l'abstrait en définissant la *sémantique abstraite des conditions*, que nous noterons $\mathcal{C}^\sharp[\![c]\!]$. Comme pour la comparaison flottante, le but de cette sémantique est de réduire l'état abstrait en fonction de la condition interprétée. Ainsi, pour un domaine abstrait dont les éléments appartiennent à A, nous avons :

$$\mathcal{C}^\sharp[\![c]\!] : A \mapsto A \tag{3.12}$$

Sa définition est donnée à la figure 3.5. L'interprétation des comparaisons flottantes nous est donnée par le domaine abstrait. Les opérateurs $\wedge$ et $\vee$ sont respectivement

$$\begin{aligned}
\mathcal{C}^\sharp[\![e_1 \leq e_2]\!](\rho^\sharp) &\text{ donné} \\
\mathcal{C}^\sharp[\![c_1 \wedge c_2]\!](\rho^\sharp) &= \mathcal{C}^\sharp[\![c_1]\!](\rho^\sharp) \sqcap \mathcal{C}^\sharp[\![c_2]\!](\rho^\sharp) \\
\mathcal{C}^\sharp[\![c_1 \vee c_2]\!](\rho^\sharp) &= \mathcal{C}^\sharp[\![c_1]\!](\rho^\sharp) \sqcup \mathcal{C}^\sharp[\![c_2]\!](\rho^\sharp) \\
\mathcal{C}^\sharp[\![\neg(e_1 \leq e_2)]\!](\rho^\sharp) &= \mathcal{C}^\sharp[\![e_2 \leq e_1]\!](\rho^\sharp) \\
\mathcal{C}^\sharp[\![\neg(c_1 \wedge c_2)]\!](\rho^\sharp) &= \mathcal{C}^\sharp[\![\neg c_1]\!](\rho^\sharp) \sqcup \mathcal{C}^\sharp[\![\neg c_2]\!](\rho^\sharp) \\
\mathcal{C}^\sharp[\![\neg(c_1 \vee c_2)]\!](\rho^\sharp) &= \mathcal{C}^\sharp[\![\neg c_1]\!](\rho^\sharp) \sqcap \mathcal{C}^\sharp[\![\neg c_2]\!](\rho^\sharp)
\end{aligned}$$

FIGURE 3.5 – Sémantique abstraite des conditions.

interprétés comme l'intersection et l'union abstraite des états abstraits produits par l'interprétation dans l'abstrait de chacune de leurs opérandes. Enfin, l'interprétation de la négation dépend de son opérande. Si cette dernière est un $\wedge$ ou un $\vee$, alors nous utilisons les lois de Morgan. Si c'est une comparaison, alors nous n'avons qu'à l'inverser. Ici, nous sommes obligés d'interpréter $\neg(e_1 \leq e_2)$ comme $e_2 \leq e_1$, et de perdre donc le caractère strict provenant de la négation, car nous n'avons pas d'opérateur abstrait pour la comparaison stricte. Ce n'est cependant pas un problème, l'état abstrait résultant étant une abstraction correcte, bien que non optimale.

Comme l'interprétation des expressions et des conditions est laissée à la charge du domaine abstrait, l'abstraction de notre sémantique collectrice, que nous noterons $\mathscr{C}^\sharp[\![p]\!]$ n'a que les instructions à interpréter. Comme pour la sémantique collectrice concrete, elle doit associer une valeur abstraite à chaque étiquette du programme. Dit autrement, la sémantique collectrice abstraite est une fonction

$$\mathscr{C}^\sharp[\![p]\!] : A \mapsto (\text{labels}\,(p) \rightharpoonup A) \tag{3.13}$$

qui associe à un état abstrait initial une fonction partielle associant, à chaque étiquette du programme, l'état abstrait résultant de son exécution. Sa définition est donnée à la figure 3.6. La sémantique abstraite d'un programme ne contenant qu'une affectation est une fonction utilisant l'opérateur abstrait fourni par le domaine pour évaluer l'état abstrait résultant de l'interprétation de l'affectation. Pour l'identité, cela fonctionne d'une manière similaire : sa sémantique est une fonction associant l'état abstrait initial à l'instruction. Dans ces deux cas, nous n'avons pas besoin de gérer d'autres instructions. En effet, la seule étiquette pour laquelle il est possible de demander l'état abstrait associé est celle de l'instruction traitée. Pour la séquence, la fonction partielle associe, à toute étiquette l, l'état abstrait produit

$$\mathscr{C}^\sharp[\![x = (\phi)\,(e)\ ;]\!](\rho^\sharp) \text{ donné}$$
$$\mathscr{C}^\sharp[\![\textbf{skip ;}]\!](\rho^\sharp) = \lambda l.\rho^\sharp$$

$$\mathscr{C}^\sharp[\![s_1\ s_2]\!](\rho^\sharp) = \lambda l. \begin{cases} \mathscr{C}^\sharp[\![s_1]\!](\rho^\sharp)(l) & \text{si } l \in \text{labels}\,(s_1) \\ \mathscr{C}^\sharp[\![s_2]\!](\mathscr{C}^\sharp[\![s_1]\!](\rho^\sharp)(\text{last}\,(s_1)))(l) & \text{sinon} \end{cases}$$

$$\mathscr{C}^\sharp[\![\textbf{if } c \textbf{ then } s_t \textbf{ else } s_e \textbf{ end}]\!](\rho^\sharp) = \lambda l.\mathscr{C}^\sharp[\![s_t]\!](t^\sharp)(l) \sqcup \mathscr{C}^\sharp[\![s_e]\!](f^\sharp)(l)$$
$$\text{avec } f^\sharp = \mathcal{C}^\sharp[\![\neg c]\!](\rho^\sharp) \text{ et } t^\sharp = \mathcal{C}^\sharp[\![c]\!](\rho^\sharp)$$

$$\mathscr{C}^\sharp[\![\textbf{while } c \textbf{ do } s \textbf{ done}]\!](\rho^\sharp) = \lambda l. \begin{cases} \bot^\sharp & \text{si } l \notin \text{labels}\,(s) \\ \mathcal{C}^\sharp[\![\neg c]\!](\sqcup\{x_i\}_{i\in\mathbb{N}}) & \text{sinon} \end{cases}$$
$$\text{avec } x_0 = \bot^\sharp \text{ et } x_{i+1} = x_i \nabla (\rho^\sharp \sqcup \mathscr{C}^\sharp[\![s]\!](\mathcal{C}^\sharp[\![c]\!](x_i))(l))$$

FIGURE 3.6 – Sémantique abstraite du langage MINIFLOAT.

par l'interprétation de sa première instruction s_1, en utilisant l'état abstrait en argument, si l appartient à s_1. Sinon, elle interprète sa seconde instruction et utilise la fonction partielle ainsi crée pour calculer l'état abstrait à associer à l. Pour les instructions dont l'étiquette appartient à sa seconde instruction, elle associe l'état abstrait produit par son interprétation mais en utilisant l'état abstrait associé à la dernière étiquette de la première instruction. Le test utilise la sémantique abstraite des conditions décrites à la figure 3.5 pour réduire l'état d'entrée, une première fois avec la condition du test et une seconde fois avec sa négation. Ces états sont ensuite utilisés lors de l'interprétation des branches du test, le premier pour interpréter la branche **then** et le second pour la branche **else**. Finalement, pour chaque étiquette, la fonction retournée construit l'union abstraite des états issus de l'interprétation des deux branches. Pour terminer, la sémantique de la boucle associe $\bot^\sharp$ pour toute étiquette n'appartenant pas à son corps, et, pour celles lui appartenant, le supremum de la suite construite en utilisant le théorème 3.2.5 et réduite par la négation de la condition.

Ainsi, il nous suffit, pour créer une analyse d'erreurs numériques, de définir un domaine abstrait répondant aux exigences de la définition 3.3.1. Nous allons pour cela présenter les deux domaines classiques sur lesquelles nous nous sommes penchés au cours de ce travail, à savoir le domaines des *intervalles* et le domaine des *zonotopes*. Cependant, ces deux domaines partagent une particularité : ils

s'intéressent à définir comment abstraire l'ensemble des *valeurs* qu'une variable peut avoir, et non comment abstraire l'ensemble des *états* dans lequel le programme peut être. Le lien entre l'abstraction de valeurs et l'abstraction d'états est fait par un mécanisme générique, qu'il convient donc de présenter d'abord.

3.4 Des valeurs abstraites aux domaines

Notre sémantique collectrice concrète construit une fonction partielle qui associe, à chaque étiquette du programme, un ensemble d'états machine, ces états étant eux-mêmes des fonctions partielles associant une valeur (dans notre cas, un réel, un flottant, une erreur absolue et une erreur relative) à chaque variable. Ainsi, chaque variable est associée, au travers de tous les états retournés par la sémantique collectrice, à un ensemble de valeurs possibles. L'idée derrière les domaines des intervalles et des zonotopes est de fournir une abstraction pour ces ensembles de valeurs, et de construire une abstraction des ensembles d'états machine à partir de là. Ce choix est motivé par le fait que définir des abstractions pour les valeurs est généralement plus simple. Cependant, il a également tendance à limiter l'expressivité du domaine résultant car les informations représentées dans l'abstraction ne portent que sur les valeurs des variables, ce qui restreint la possibilité de raisonnements relationnelles.

Afin de pouvoir construire un domaine abstrait respectant les exigences de la définition 3.3.1, une abstraction de valeurs doit également respecter certaines exigences minimales.

Définition 3.4.1. Un ensemble partiellement ordonné $(R^\sharp, \sqsubseteq_r)$ est une *valeur abstraite* utilisable pour définir un domaine abstrait satisfaisant la définition 3.3.1 s'il possède :

- une fonction de concrétisation monotone produisant des ensembles de réels : $\gamma_r : R^\sharp \mapsto \mathcal{P}(\mathbb{R})$;
- un plus petit élément $\bot_r^\sharp$ représentant l'ensemble vide ;
- un plus grand élément $\top_r^\sharp$ représentant $\mathbb{R}$ en entier ;
- des opérateurs abstraits corrects pour tous les opérateurs arithmétiques de notre langage, l'union et l'intersection ;
- des opérateurs abstraits corrects pour les erreurs absolues et relatives élémentaires ;
- un opérateur d'élargissement ∇_r respectant la définition 3.2.8 ;
- (optionnel) une fonction d'abstraction monotone α_r telle que (α_r, γ_r) soit une correspondance de Galois.

Deux choses sont à noter dans cette définition. Premier point, nous n'exigeons pas des valeurs abstraites qu'elles fournissent un opérateur abstrait pour la compa-

raison flottante. En effet, une abstraction correcte mais imprécise pour la comparaison est de ne pas réduire l'état abstrait. En pratique, les domaines des intervalles et des zonotopes proposent des techniques pour gérer les comparaisons. Nous les présentons par la suite. Second point, nous ne demandons pas aux valeurs abstraites de représenter des ensembles de quadruplets (réels, flottants, erreurs absolues et erreurs relatives), mais simplement des ensembles de réels. Cela peut sembler surprenant de prime abord, car la sémantique concrète utilise ces quadruplets comme valeurs. Cependant, nous pouvons reconstruire tout ce dont nous avons besoin sur les quadruplets à partir de ce que nous exigeons des valeurs abstraites.

Pour cela, la première étape est de construire un ensemble partiellement ordonné disposant d'une fonction de concrétisation vers des ensembles de valeurs concrètes. Nous pouvons trivialement montrer que l'ensemble $R^\sharp \times R^\sharp \times R^\sharp \times R^\sharp$ muni de la relation d'ordre et de la fonction de concrétisation

$$(r_1^\sharp, f_1^\sharp, e_{a_1}^\sharp, e_{r_1}^\sharp) \sqsubseteq_v (r_2^\sharp, f_2^\sharp, e_{a_2}^\sharp, e_{r_2}^\sharp) \iff r_1^\sharp \sqsubseteq_r r_2^\sharp \wedge f_1^\sharp \sqsubseteq_r f_2^\sharp \wedge e_{a_1}^\sharp \sqsubseteq_r e_{a_2}^\sharp \wedge e_{r_1}^\sharp \sqsubseteq_r e_{r_2}^\sharp \quad (3.14)$$

$$\gamma_v((r^\sharp, f^\sharp, e_a^\sharp, e_r^\sharp)) = \{(r, f, e_a, e_r) \mid r \in \gamma_r(r^\sharp) \wedge f \in \gamma_r(f^\sharp) \wedge e_a \in \gamma_r(e_a^\sharp) \wedge e_r \in \gamma_r(e_r^\sharp)\} \quad (3.15)$$

satisfait ce besoin. On notera tout de même que cette abstraction représente les valeurs flottantes comme des réels. C'est une abstraction correcte, car les réels contiennent tous les flottants, mais cela introduit une grosse approximation.

Cette abstraction dispose trivialement d'un plus grand et d'un plus petit élément, ainsi que d'un opérateur d'élargissement et d'opérateurs corrects pour l'union et l'intersection. Il suffit pour cela de réutiliser les éléments provenant de la valeur abstraite sur chacune des quatre abstractions. De la même manière, si la valeur abstraite dispose d'une fonction d'abstraction, nous pouvons en reconstruire une pour $R^\sharp \times R^\sharp \times R^\sharp \times R^\sharp$ en utilisant le même procédé que pour la fonction de concrétisation. Ainsi, pour construire une valeur abstraite pour les quadruplets concrets, il ne nous reste qu'à fournir des opérateurs abstraits pour l'arithmétique. Cependant, il faut ici faire attention au fait que l'arithmétique sur les erreurs n'a pas le même sens que l'arithmétique sur les valeurs réelles. Dit autrement, il nous faut ici fournir une sémantique abstraite des expressions respectant le même découpage que celui que nous avions utilisé au chapitre 2 pour présenter la sémantique concrète des expressions.

Nous présentons à présent les sémantiques abstraites des expressions à partir d'une valeur abstraite $V^\sharp$ permettant de représenter nos quadruplets de valeurs concrètes. Cette valeur abstraite est construite en suivant la procédure que nous venons de décrire, et nous utilisons les notations $f_v^\sharp$ pour dénoter l'objet $f^\sharp$ associé

à la valeur abstraite $V^\sharp$. Tout d'abord, abstraire la sémantique exacte, travaillant sur les valeurs réelles, se fait trivialement en utilisant les opérateurs abstraits fournis par la valeur abstraite. Ensuite, Les sémantiques abstraites des erreurs absolues et relatives s'obtiennent d'une manière analogue, en calculant dans l'abstrait les équations données à la figure 2.4. En effet, ces dernières ne contiennent que des opérateurs arithmétiques réels et les erreurs élémentaires, pour lesquels nous disposons de versions abstraites. Enfin, abstraire la sémantique approximée peut se faire d'une manière triviale, bien que potentiellement imprécise, en utilisant les relations 2.7 et 2.8 afin de calculer une approximation de la valeur flottante à partir des approximations des valeurs réelles et des erreurs. Nous remarquons par ailleurs qu'il n'est du coup pas nécessaire de disposer d'une abstraction pour la valeur flottante, car nous pouvons la reconstruire à chaque instant à partir des abstractions des réels et des erreurs.

Nous disposons donc d'une valeur abstraite permettant de représenter les valeurs concrètes de notre langage. Il nous faut désormais construire un domaine abstrait satisfaisant les exigences de la définition 3.3.1. Pour cela, commençons par construire l'ensemble partiellement ordonné qui va permettre d'abstraire les ensembles d'états machine concrets. Ces états abstraits utilisent la même structure qu'un état concret : ce sont des fonctions partielles associant une valeur abstraite à chaque variable du programme. La relation d'ordre que nous utilisons pour ces états abstraits, qui reprend la même idée que celle présentée en 2.35, est la suivante :

$$\rho_1^\sharp \sqsubseteq_d \rho_2^\sharp \iff \mathrm{dom}\left(\rho_1^\sharp\right) \subseteq \mathrm{dom}\left(\rho_2^\sharp\right) \wedge \forall x \in \mathrm{dom}\left(\rho_1^\sharp\right), \rho_1^\sharp(x) \sqsubseteq_v \rho_2^\sharp(x) \tag{3.16}$$

Nous ne fournissons pas la preuve que $(\mathbb{V} \rightharpoonup V^\sharp, \sqsubseteq_d)$ forme un ordre partiel, cette dernière reprenant très largement les raisonnements que nous avions utilisés pour prouver que la relation 2.2 forme un ordre partiel sur les états machine concrets au chapitre précédent.

Cet ordre partiel dispose trivialement d'un plus petit élément, à savoir la fonction partielle n'associant de valeur abstraite à aucune variable, et d'un plus grand élément, à savoir la fonction partielle associant $\top_v^\sharp$ à toutes les variables de $\mathbb{V}$.

Afin que $V^\sharp$ forme un domaine abstrait, nous avons besoin d'une fonction de concrétisation. Nous utilisons pour cela la fonction γ_d suivante, dont l'objectif est de construire l'ensemble des états machine concrets pour lesquels l'affectation de chaque variable est contenue dans l'abstraction associée dans l'abstrait à cette variable, et dont la monotonie découle directement de celle de γ_v :

$$\gamma_d(\rho^\sharp) = \left\{\rho \mid \mathrm{dom}\left(\rho\right) \subseteq \mathrm{dom}\left(\rho^\sharp\right) \wedge \forall x \in \mathrm{dom}\left(\rho\right), \rho(x) \in \gamma_v(\rho^\sharp(x))\right\} \tag{3.17}$$

De plus, si nous disposons d'une correspondance de Galois sur $V^\sharp$, nous pouvons facilement l'étendre au domaine abstrait en utilisant la fonction d'abstraction suivante, dont l'objectif est d'associer à chaque variable l'abstraction de l'ensemble

des valeurs concrètes qu'elle peut prendre, sa monotonie découlant également directement de celle de α_v :

$$\alpha_d(E) = \lambda x.\alpha_v\left(\{\rho(x) \mid \rho \in E \wedge x \in \mathrm{dom}\,(\rho)\}\right) \tag{3.18}$$

La paire (γ_d, α_d) forme trivialement une correspondance de Galois pour le domaine abstrait.

Nous pouvons facilement définir un opérateur d'élargissement pour ce domaine abstrait à partir de celui sur les valeurs abstraites :

$$\rho_1^\sharp \nabla_d \rho_2^\sharp = \lambda x. \begin{cases} \rho_1^\sharp(x) & \text{si } x \in \mathrm{dom}\left(\rho_1^\sharp\right) \wedge x \notin \mathrm{dom}\left(\rho_2^\sharp\right) \\ \rho_2^\sharp(x) & \text{si } x \notin \mathrm{dom}\left(\rho_1^\sharp\right) \wedge x \in \mathrm{dom}\left(\rho_2^\sharp\right) \\ \rho_1^\sharp(x) \nabla_v \rho_2^\sharp(x) & \text{si } x \in \mathrm{dom}\left(\rho_1^\sharp\right) \wedge x \in \mathrm{dom}\left(\rho_2^\sharp\right) \end{cases} \tag{3.19}$$

Les propriétés de calcul de borne supérieure et de convergence en temps fini découlent trivialement des propriétés de l'opérateur d'élargissement sur les valeurs abstraites.

$$\rho_1^\sharp \sqcup_d \rho_2^\sharp \triangleq \lambda x.\rho_1^\sharp(x) \sqcup_v \rho_2^\sharp(x) \quad \text{et} \quad \rho_1^\sharp \sqcap_d \rho_2^\sharp \triangleq \lambda x.\rho_1^\sharp(x) \sqcap_v \rho_2^\sharp(x)$$

$$D[\![x = (\phi)\,(e)\;;]\!](\rho^\sharp) \triangleq \lambda y. \begin{cases} V[\![e]\!](\rho^\sharp) & \text{si } x = y \\ \rho^\sharp(y) & \text{sinon} \end{cases}$$

FIGURE 3.7 – Opérateurs abstraits construits à partir d'une valeur abstraite pour le supremum, l'infimum et l'affectation.

Ainsi, il ne nous reste qu'à définir des opérateurs abstraits pour l'affectation, le supremum et l'infimum afin de terminer la construction de notre domaine abstrait. En effet, comme nous l'avons dit précédemment, la comparaison peut trivialement être abstraite avec l'identité. Ces opérateurs sont résumés à la figure 3.7. Pour le supremum et l'infimum, il nous suffit d'utiliser la même idée que pour l'opérateur d'élargissement, en appliquant variable par variable les opérateurs abstraits provenant de la valeur abstraite. Comme pour l'opérateur d'élargissement, la preuve de correction découle trivialement de la correction des opérateurs abstraits sur les valeurs. L'affectation, quant à elle, est simplement abstraite en associant à la variable l'abstraction produite par l'évaluation de la sémantique abstraite des expressions. La correction découle directement de la correction de la sémantique abstraite des expressions.

Nous sommes donc en mesure de produire un domaine abstrait satisfaisant la définition 3.3.1 à partir d'une simple valeur abstraite satisfaisant 3.4.1. Nous

pouvons désormais présenter les deux valeurs abstraites qui nous ont intéressés au cours de ces travaux : les intervalles et les formes affines, éléments atomiques composant le domaine des zonotopes.

Chapitre 4

Domaines Abstraits

Dans ce chapitre, nous présentons les deux domaines abstraits que nous avons utilisés au cours de ces travaux : le domaine des intervalles et celui des zonotopes. Pour chacun de ces domaines, la présentation s'axe autour de la construction de l'abstraction de valeurs qui lui sert de base, la création du domaine se faisant en suivant la méthodologie décrite à la Section 3.4 du chapitre précédent. Le domaine des intervalles est un grand classique en interprétation abstraite, et est utilisé par tous les analyseurs utilisant l'interprétation abstraite dont nous ayons connaissance. Sa présentation est donc rapide. Le domaine des zonotopes, quant à lui, est bien moins utilisé et spécialisé pour l'analyse d'erreurs numériques. Sa présentation est donc plus détaillée. Un soin tout particulier est apporté à la présentation de ses opérateurs arithmétiques.

4.1 Domaine des intervalles

Le domaine des intervalles, probablement le plus connu des domaines abstraits, a été introduit par Cousot et Cousot [19] et est fondé sur l'arithmétique des intervalles introduite par Moore pour l'analyse numérique en 1966 [63]. Il présente comme grands avantages d'être simple et performant, tout en offrant suffisamment d'expressivité pour inférer des propriétés intéressantes quoique simples. Un ensemble de valeurs y est représenté par un *intervalle*, c'est-à-dire une paire définissant ses bornes inférieures et supérieures, ou par $\bot_{\mathbb{I}}^{\sharp}$, représentant l'ensemble vide. Ainsi, l'ensemble $\mathbb{I}$ des intervalles est formellement défini par :

$$\mathbb{I} = \{[a\,;\,b] \mid a, b \in \mathbb{R} \cup \{\pm\infty\} \wedge a \leq b\} \cup \{\bot_{\mathbb{I}}^{\sharp}\} \tag{4.1}$$

Il est important de noter que nous utilisons des bornes réelles dans cette formalisation. En pratique, utiliser des réels est impossible et nous utilisons des rationnels à la place. Nous reparlerons plus en détail de ce problème au chapitre 8. Les

bornes peuvent également valoir $\pm\infty$ afin de pouvoir représenter $\mathbb{R}$ tout entier, nous dispensant ainsi d'un plus grand élément. La notation $\top_{\mathbb{I}}^{\sharp}$ correspond ainsi à l'intervalle $[-\infty\,;+\infty]$. À l'inverse, il nous est nécessaire d'ajouter un plus petit élément afin de représenter l'ensemble vide. Nous le notons $\bot_{\mathbb{I}}^{\sharp}$.

Introduisons donc les différents éléments nécessaires pour faire de $\mathbb{I}$ une valeur abstraite. La première étape est de construire un ordre partiel sur $\mathbb{I}$, noté $\sqsubseteq_{\mathbb{I}}$:

$$\begin{gathered}\forall a,b,c,d \in \mathbb{R}\cup\{\pm\infty\}, [a\,;b] \sqsubseteq_{\mathbb{I}} [c\,;d] \iff c \leq a \wedge b \leq d \text{ ; et} \\ \forall X \in \mathbb{I}, \bot_{\mathbb{I}}^{\sharp} \sqsubseteq_{\mathbb{I}} X\end{gathered} \tag{4.2}$$

Un intervalle est ainsi plus petit qu'un autre si ses deux bornes sont incluses dans l'autre. De plus, $\bot_{\mathbb{I}}^{\sharp}$ est plus petit que n'importe quel intervalle, ce qui correspond au fait qu'il représente l'ensemble vide.

Théorème 4.1.1. L'ensemble $\mathbb{I}$ muni de la relation $\sqsubseteq_{\mathbb{I}}$ forme un ordre partiel, dont $\bot_{\mathbb{I}}^{\sharp}$ et $\top_{\mathbb{I}}^{\sharp}$ sont respectivement le plus petit et le plus grand élément.

Démonstration. Les preuves découlent trivialement de la définition de $\sqsubseteq_{\mathbb{I}}$ et des propriétés d'ordre sur les réels. □

Continuons avec la fonction de concrétisation. Sa définition découle trivialement de la notion d'intervalle. Ainsi, la concrétisation d'un intervalle est l'ensemble des réels compris entre ses bornes, ou l'ensemble vide pour $\bot_{\mathbb{I}}^{\sharp}$:

$$\begin{aligned}\gamma_{\mathbb{I}}([a\,;b]) &= \{x \in \mathbb{R} \mid a \leq x \leq b\} \\ \gamma_{\mathbb{I}}(\bot_{\mathbb{I}}^{\sharp}) &= \emptyset\end{aligned} \tag{4.3}$$

Théorème 4.1.2. La fonction $\gamma_{\mathbb{I}} : \mathbb{I} \mapsto \mathcal{P}(\mathbb{R})$ est monotone.

Démonstration. La preuve découle trivialement de la définition de $\gamma_{\mathbb{I}}$ et de l'ordre partiel sur les intervalles. □

Il nous faut ensuite donner des opérateurs abstraits pour l'union, l'intersection et l'arithmétique. Commençons par les deux premiers opérateurs. Pour l'union, il nous suffit de prendre comme bornes les valeurs minimales et maximales des intervalles, $\bot_{\mathbb{I}}^{\sharp}$ se comportant comme un élément neutre pour cet opérateur :

$$\begin{aligned}[a\,;b] \sqcup_{\mathbb{I}} [c\,;d] &= [\min(a,c)\,;\max(b,d)] \\ \bot_{\mathbb{I}}^{\sharp} \sqcup_{\mathbb{I}} X = X \sqcup_{\mathbb{I}} \bot_{\mathbb{I}}^{\sharp} &= X\end{aligned} \tag{4.4}$$

Pour l'intersection, il suffit à l'inverse de prendre la plus grande des bornes inférieures comme borne inférieure et la plus petite des bornes supérieures pour la borne supérieure. Cependant, il est nécessaire ici de s'assurer que l'intervalle résultant est bien formé. Il est en effet possible que la plus grande des bornes inférieures

soit plus grande que la plus petite des bornes supérieures si les deux intervalles sont disjoints. Enfin, $\bot_{\mathbb{I}}^\sharp$ est ici élément absorbant :

$$[a\,;\,b] \sqcap_{\mathbb{I}} [c\,;\,d] = \begin{cases} [\max(a,c)\,;\,\min(b,d)] & \text{si } \max(a,c) \leq \min(b,d) \\ \bot_{\mathbb{I}}^\sharp & \text{sinon} \end{cases} \tag{4.5}$$
$$\bot_{\mathbb{I}}^\sharp \sqcap_{\mathbb{I}} X = X \sqcap_{\mathbb{I}} \bot_{\mathbb{I}}^\sharp = \bot_{\mathbb{I}}^\sharp$$

Théorème 4.1.3. Les opérateurs abstraits $\sqcup_{\mathbb{I}}$ et $\sqcap_{\mathbb{I}}$ sont respectivement des abstractions correctes de l'union et l'intersection.

Démonstration. La preuve pour l'union découle trivialement du fait que $\sqcup_{\mathbb{I}}$ construit un intervalle contenant nécessairement ses deux opérandes. Pour l'intersection, il suffit de remarquer que l'intersection de deux intervalles, si elle existe, ne peut être qu'un intervalle dont les bornes inférieures et supérieures sont respectivement le maximum des bornes inférieures des opérandes et le minimum de leurs bornes supérieures, et que cette intersection n'existe que si ces deux valeurs sont correctement ordonnées. □

Passons maintenant aux opérateurs arithmétiques. Nous ne fournissons pas les preuves de correction, un lecteur curieux peut les trouver ici [64]. Elles consistent principalement à étudier la monotonie de l'opérateur sur la zone bornée par les deux intervalles pour déterminer comment calculer les bornes supérieures et inférieures. Les définitions des opérateurs abstraits sont fournies à la figure 4.1. Le cas $\bot_{\mathbb{I}}^\sharp$ n'y est pas explicité par soucis de simplicité, mais ce dernier est élément absorbant pour tous les opérateurs.

$$\begin{aligned}
[a\,;\,b] + [c\,;\,d] &= [a+c\,;\,b+d] \\
[a\,;\,b] - [c\,;\,d] &= [a-d\,;\,b-c] \\
[a\,;\,b] \times [c\,;\,d] &= [\min(ac,ad,bc,bd)\,;\,\max(ac,ad,bc,bd)] \\
[a\,;\,b] \div [c\,;\,d] &= \begin{cases} \top_{\mathbb{I}}^\sharp & \text{si } 0 \in [c\,;\,d] \\ \left[\min\left(\frac{a}{c},\frac{a}{d},\frac{b}{c},\frac{b}{d}\right)\,;\,\max\left(\frac{a}{c},\frac{a}{d},\frac{b}{c},\frac{b}{d}\right)\right] & \text{sinon} \end{cases} \\
-[a\,;\,b] &= [-b\,;\,-a] \\
\sqrt{[a\,;\,b]} &= \begin{cases} \bot_{\mathbb{I}}^\sharp & \text{si } b < 0 \\ \left[0\,;\,\sqrt{b}\right] & \text{si } a \leq 0 \text{ et } b > 0 \\ \left[\sqrt{a}\,;\,\sqrt{b}\right] & \text{sinon} \end{cases}
\end{aligned}$$

FIGURE 4.1 – Arithmétique correcte sur les intervalles.

L'addition consiste simplement à additionner les bornes inférieures et les bornes supérieures entre elles. Pour la soustraction, la borne inférieure est la différence entre la borne inférieure de sa première opérande et la borne supérieure de sa deuxième. Sa borne supérieure est calculée en inversant ce schéma. Pour la multiplication, il suffit de calculer les produits de toutes les bornes puis de prendre les valeurs minimales et maximales. La division suit exactement le même raisonnement, à ceci prêt qu'il faut auparavant vérifier que l'intervalle diviseur ne contient pas zéro. Si c'est le cas, il convient alors de renvoyer $\top_{\mathbb{I}}^{\sharp}$. Pour la négation, il suffit d'inverser l'ordre des bornes et de prendre leurs opposées. Enfin pour la racine carrée, il suffit de l'appliquer sur chaque borne pour obtenir l'intervalle résultat. Il convient cependant de vérifier que l'opérande ne contient pas de valeurs strictement négatives. Si tout l'intervalle est strictement négatif, alors le seul résultat possible est $\bot_{\mathbb{I}}^{\sharp}$: le calcul ne fait aucun sens. Si l'intervalle contient des valeurs strictement négatives et des valeurs positives ou nulles, il convient d'ignorer les valeurs négatives, un mécanisme d'alarmes non présenté ici permettant de prévenir l'utilisateur qu'une opération non autorisée peut potentiellement se produire.

Nous pouvons également définir une fonction d'abstraction pour le domaine des intervalles, formant une connexion de Galois avec $\gamma_{\mathbb{I}}$:

$$\begin{aligned} \alpha_{\mathbb{I}}(X) &= [\inf(X)\ ; \sup(X)] \\ \alpha_{\mathbb{I}}(\emptyset) &= \bot_{\mathbb{I}}^{\sharp} \end{aligned} \tag{4.6}$$

Théorème 4.1.4. La fonction $\alpha_{\mathbb{I}} : \mathcal{P}(\mathbb{R}) \mapsto \mathbb{I}$ est monotone et forme une connexion de Galois avec $\gamma_{\mathbb{I}}$.

Démonstration. La monotonie découle trivialement de la définition de $\alpha_{\mathbb{I}}$. Démontrer la connexion de Galois se fait facilement en utilisant la définition 3.2.4. En effet, nous pouvons, dans un premier temps, montrer trivialement que l'intervalle $[\inf(X)\ ; \sup(X)]$ est une abstraction correcte de X pour toute sous-partie de $\mathbb{R}$. En effet, pour que cela ne soit pas le cas, il faudrait qu'une valeur plus grande que le supremum de X (ou plus petite que son infimum) appartienne à X, ce qui est impossible. Ainsi, nous avons $X \subseteq (\gamma_{\mathbb{I}} \circ \alpha_{\mathbb{I}})(X)$. Dans un second temps, il est trivial de montrer que la fonction $\alpha_{\mathbb{I}} \circ \gamma_{\mathbb{I}}$ correspond à l'identité. En effet, le minimum et le maximum de l'ensemble produit par la concrétisation d'un intervalle sont ses bornes. L'abstraction de cet ensemble produira donc bien le même intervalle, ce qui conclut la démonstration. □

Définissons maintenant un opérateur d'élargissement sur les intervalles plus intéressant que celui présenté à la section 3.2.4. L'idée ici n'est de diverger sur une

seule borne lorsque c'est possible :

$$[a\,;\,b]\,\nabla_{\mathbb{I}}\,[c\,;\,d] = \begin{cases} [c\,;\,d] & \text{si } a \geq c \wedge b \leq d \\ [c\,;\,+\infty] & \text{si } a \geq c \wedge b > d \\ [-\infty\,;\,d] & \text{si } a < c \wedge b \leq d \\ [-\infty\,;\,+\infty] & \text{sinon} \end{cases} \tag{4.7}$$

Cet opérateur peut encore être amélioré, en ajoutant des *seuils* [52] pour éviter de diverger trop vite. Cela permet d'améliorer la précision, mais peut ralentir la convergence.

Théorème 4.1.5. L'opérateur $\nabla_{\mathbb{I}}$ satisfait la définition 3.2.8.

Démonstration. La propriété de borne supérieure découle trivialement de la définition. La convergence est assurée par le fait que les bornes divergent en une étape. □

Pour terminer la présentation du domaine des intervalles, il ne nous reste qu'à présenter les opérateurs abstraits calculant les erreurs absolues et relatives élémentaires. Nous ne présentons ici qu'une version naive de ces opérateurs. Une version plus fine et précise sera présentée en détails au chapitre 5.

En partant des équations 1.5 et 1.6, ainsi que de la définition 1.1, introduites au chapitre 1, nous pouvons obtenir les équations suivantes pour les erreurs élémentaires :

$$\Gamma_a^{\phi}(x) = xe_x + d_x \tag{4.8}$$

$$\Gamma_r^{\phi}(x) = e_x + \frac{d_x}{x} \tag{4.9}$$

où, pour rappel, e_x et d_x sont deux réels bornés en fonction du format ϕ. Ainsi, il est possible de définir des abstractions correctes pour ces paramètres qui comprennent toutes leurs valeurs possibles. Comme les opérations utilisées dans ces équations disposent toutes d'une version correcte sur les intervalles, nous pouvons facilement calculer une abstraction correcte des deux erreurs élémentaires dans les intervalles.

Nous disposons donc de tous les éléments nécessaires pour faire des intervalles une valeur abstraite sur laquelle fonder une analyse de notre langage. Il nous reste cependant à présenter une méthodologie standard permettant d'améliorer la méthodologie naïve pour la gestion des comparaisons : la *propagation arrière* [18]. Nous ne présentons ici qu'une intuition de cette technique et, en particulier, ne présentons pas l'ensemble des propagateurs arrières aujourd'hui existants. Un lecteur curieux peut se référer à [64] pour obtenir plus d'informations.

Considérons un exemple. Imaginons que nous ayons deux variables, x et y, respectivement associées dans notre état abstrait aux intervalles $[1\,;\,12]$ et $[3\,;\,7]$, et

que nous analysions un programme contenant un test avec la condition $x \leq y$. Dans cette situation, notre sémantique abstraite analyse les deux branches de chaque test, en supposant que la condition est vérifiée pour la branche `then`, et violée pour la branche `else`. Or, dans notre cas, il existe des valeurs de x pour lesquelles il est impossible d'aller dans une des branches. Par exemple, il est impossible de valider la condition avec $x = 10$ car il n'existe aucune valeur de y plus grande que 10. Nous pouvons donc *réduire* l'intervalle de x dès lors que l'hypothèse $x \leq y$ est vérifiée. La technique de propagation arrière consiste à définir une manière de réduire les intervalles des variables apparaissants dans une condition à partir de ces hypothèses. Dans notre exemple, l'intervalle de x peut ainsi être réduit à $[1\ ;\ 7]$ pour l'analyse de la branche `then`, et à $[3\ ;\ 12]$ pour celle de la branche `else`. Notons ici que, pour la branche `else`, nous sommes obligés de conserver la valeur 3 car nous ne disposons que d'intervalles fermés de $\mathbb{R}$. L'intervalle de y n'est, quant à lui, pas modifié. Ce n'est cependant pas toujours le cas, et l'application de cette technique à des conditions plus complexes, faisant notamment intervenir plus de variables, peut amener à réduire les abstractions associées à plusieurs variables.

Le domaine des intervalles est ainsi un domaine simple, formé sur une arithmétique facile à manipuler et rapide à calculer, et permettant de décrire des propriétés intéressantes sur les programmes analysés. Cependant, il souffre également d'un problème majeur : il ne permet pas de garder trace des relations entre les variables. Un exemple typique, bien qu'extrême, de ce problème, est le calcul de l'expression $x - x$, dont le résultat est trivialement zéro. Cependant, l'arithmétique des intervalles ne permet pas de le déterminer dans le cas général. Par exemple, si la valeur abstraite de x est l'intervalle $[0\ ;\ 2]$ alors l'évaluation de l'expression $x - x$ est l'intervalle $[-4\ ;\ 4]$ et non $[0\ ;\ 0]$. Cette limitation fait que le domaine des intervalles devient rapidement très imprécis lors de l'analyse de programmes faisant apparaitre des relations. Plusieurs réponses à ce problème existent, faisant différents compromis entre précision et performance. Dans le cadre de cette thèse, nous nous sommes concentrés sur l'une de ces réponses : le domaine des zonotopes.

4.2 Domaine des zonotopes

Le domaine des zonotopes [15, 40, 41] est une réponse à la limitation du domaine des intervalles, cherchant à garder trace de certaines relations entre les variables du programme analysé tout en impactant le moins possible les performances. Son fonctionnement peut être vu au travers de différents prismes. Le premier, plus global et abstrait, est fondé sur l'algèbre linéaire et la notion de *transformation affine*. Le second, plus terre à terre et orienté vers l'apport d'une réponse aux limitations des intervalles, est concentré autour de la définition de valeurs abstraites relationnelles appelées *formes affines*. Ces deux visions sont équivalentes, chacune

permettant de reconstruire l'autre. Cependant, c'est au travers du second prisme que le domaine des zonotopes a été abordé au cours de cette thèse. De plus, nous pensons qu'il permet de construire plus facilement une bonne intuition du fonctionnement de ce domaine. Ainsi, c'est au travers de ce prisme que nous allons le présenter.

Nous allons commencer cette présentation en nous aidant d'un programme comme exemple, ce qui nous permettra d'illustrer l'intuition derrière le fonctionnement du domaine des zonotopes et d'en montrer l'intérêt par rapport aux intervalles. Ce programme, présenté à la figure 4.2, est écrit en utilisant notre langage d'étude et consiste en une série d'affectations dans un format arbitraire ϕ. Ce dernier ne nous intéresse pas ici, car nous allons nous limiter à la sémantique exacte, suffisante pour mettre en avant ce que nous souhaitons illustrer.

$$
\begin{aligned}
a &= (\phi)\,(2 \times x)\,;\\
b &= (\phi)\,(y \div 3)\,;\\
z &= (\phi)\,(a + b)\,;\\
r &= (\phi)\,(z - x - y)\,;
\end{aligned}
$$

FIGURE 4.2 – Programme d'illustration du domaine des zonotopes.

Ce programme consiste en une série de quatre affectations, dans lesquelles nous pouvons trouver deux variables à valeurs arbitraires, à savoir x et y. Nous souhaitons ici calculer une surapproximation pour la variable r afin de prouver, par exemple, que si la valeur de x est compris entre -2 et 4, et celle de y entre -6 et 6, alors toutes les valeurs possibles de r sont comprises entre -10 et 10. La figure 4.3 ci-dessous présente une première analyse de ce programme utilisant le domaine des intervalles. Chaque ligne y est annotée avec l'intervalle surapproximant la sémantique exacte à partir de l'état initial pour sa variable gauche. Ainsi, les valeurs exactes possibles pour a sont comprises entre -4 et 8 par exemple.

$$
\begin{aligned}
a &= (\phi)\,(2 \times x)\,; && \rightarrow && a \in [-4\,;\,8]\\
b &= (\phi)\,(y \div 3)\,; && \rightarrow && b \in [-2\,;\,2]\\
z &= (\phi)\,(a + b)\,; && \rightarrow && z \in [-6\,;\,10]\\
r &= (\phi)\,(z - x - y)\,; && \rightarrow && r \in [-16\,;\,18]
\end{aligned}
$$

FIGURE 4.3 – Exemple d'analyse avec le domaine des intervalles.

Ces sur-approximations s'obtiennent facilement en utilisant les opérateurs arithmétiques abstraits du domaine des intervalles. Cette analyse borne la variable r entre -16 et 18, ne permettant pas de prouver la propriété qui nous intéresse. Cette situation vient du fait que le programme est construit avec des expressions relationnelles qui ne sont pas prises en compte par le domaine des intervalles. En effet, afin de calculer une approximation précise pour r, il est nécessaire de considérer que l'approximation de z contient en quelque sorte celles de x et y.

Afin de pouvoir mener à bien ce genre de raisonnement, le domaine des zonotopes propose de procéder en deux temps. Dans un premier temps, pour chaque entrée du programme (ici x et y), le domaine crée une quantité symbolique nommée *symbole de bruit*, notée ε_i, bornée entre -1 et 1. L'objectif de chacun de ces symboles de bruit est de représenter l'incertitude qu'introduit l'entrée à laquelle il est associé dans l'analyse du programme. Dans un second temps, le domaine associe à chaque entrée une expression symbolique construite en utilisant ces symboles de bruit. Dans notre exemple, le domaine crée ε_1 et ε_2, deux symboles de bruit représentant respectivement les incertitudes introduites par x et y. Le domaine associe ensuite x à l'expression symbolique $1 + 3\varepsilon_1$ et y à $6\varepsilon_2$. Il est important de noter que ces expressions symboliques sont des abstractions représentant les mêmes ensembles de valeurs que les intervalles associés à x et y par le domaine des intervalles. Il suffit, pour s'en convaincre, d'appliquer l'arithmétique d'intervalles sur ces expressions symboliques. La seule différence pour le moment est que le domaine des zonotopes dispose d'une représentation symbolique de l'incertitude des entrées. Cependant, cette différence va permettre à ce domaine de mener des raisonnement relationnels.

$$
\begin{array}{lcl}
a = (\phi)\,(2 \times x)\,; & \rightarrow & a \in 2 + 6\varepsilon_1 \\
b = (\phi)\,(y \div 3)\,; & \rightarrow & b \in 2\varepsilon_2 \\
z = (\phi)\,(a + b)\,; & \rightarrow & z \in 2 + 6\varepsilon_1 + 2\varepsilon_2 \\
r = (\phi)\,(z - x - y)\,; & \rightarrow & r \in 1 + 3\varepsilon_1 - 4\varepsilon_2
\end{array}
$$

FIGURE 4.4 – Exemple d'analyse avec le domaine des zonotopes.

Pour cela, le domaine des zonotopes permet à l'analyse de propager symboliquement ces expressions au travers du programme. Nous n'avons pas encore défini les opérateurs qui le permettent, cependant, pour notre exemple, l'intuition mathématique est suffisante. La figure 4.4 présente ainsi l'expression symbolique associée à chaque variable gauche par l'analyse : la variable a est associée à l'expression $2 \times (1 + 3\varepsilon_1) = 2 + 6\varepsilon_1$, la variable b est associée à $6\varepsilon_2 \div 3 = 2\varepsilon_2$ et la variable z à $2 + 6\varepsilon_1 + 2\varepsilon_2$. Ici encore, les expressions symboliques ne permettent

pas d'améliorer les bornes pour a, b et z. Elles permettent par contre de garder trace du lien entre z et les entrées x et y du programme. Grâce à cela, le domaine des zonotopes permet d'obtenir de meilleures bornes pour r. En effet, l'expression symbolique pour r est la suivante :

$$r = z - x - y = (2 + 6\varepsilon_1 + 2\varepsilon_2) - (1 + 3\varepsilon_1) - 6\varepsilon_2 = 1 + 3\varepsilon_1 - 4\varepsilon_2 \tag{4.10}$$

Cette expression permet, au travers de l'arithmétique d'intervalles, de prouver que la valeur de r est comprise entre -6 et 8, ce qui est plus précis que l'objectif que nous nous étions fixés, qui était, pour rappel, l'intervalle $[-10\,;\,10]$.

Le domaine des zonotopes permet donc, au travers de raisonnements symboliques, de tenir compte des relations entre variables afin de fournir une analyse plus précise. Cependant, afin de maintenir de bonnes performances, le domaine ne permet que certains raisonnements symboliques, et ne garde donc trace que de certaines relations. Plus précisément, les expressions symboliques que le domaine associe aux variables doivent toutes être de la forme :

$$\alpha_0 + \sum_{i=1}^{n} \alpha_i \varepsilon_i + \sum_{j=1}^{m} \beta_i \eta_i \tag{4.11}$$

où $\alpha_0 \in \mathbb{R}$ est nommé le *centre*, les $\alpha_i \in \mathbb{R}$ sont les *coefficients centraux* et les $\beta_i \in \mathbb{R}$ sont les *coefficients de perturbation*. Les ε_i sont les symboles de bruit représentant les incertitudes introduites par les entrées du programme, et sont nommés *symboles centraux*. Les $\eta_i \in [-1\,;\,1]$ sont, comme les ε_i, des symboles de bruit, mais représentant les incertitudes introduites par l'analyse. Ils sont nommés *symboles de perturbation*. Les expressions symboliques de cette forme sont nommées des *formes affines*.

Avant de continuer, il nous faut prendre le temps de détailler une remarque importante à propos de cette définition des formes affines. La distinction que nous faisons entre les symboles centraux et ceux de perturbation permet de rendre compte des différences que nous souhaitons accorder au cours de l'analyse aux incertitudes provenant des entrées de celles provenant de l'analyse. En effet, notre objectif est avant tout de conserver les relations avec les entrées du programme car elles permettent de mieux comprendre son comportement. À l'inverse, conserver les relations avec les incertitudes de l'analyse ne nous apportent pas d'éclairage sur le programme, et sont donc beaucoup plus facile à sacrifier pour améliorer les performances. Cependant, la distinction n'est pas toujours nécessaire pour comprendre ce que nous présentons par la suite. Ainsi, à des fins pédagogiques, nous ne distinguons les symboles que lorsque cela est nécessaire. Dans le cas contraire, nous utiliserons les notations α_i et ε_i pour désigner indifféremment les coefficients et symboles centraux et de perturbations.

Notons également que sacrifier des symboles de bruit pour améliorer les performances est une question importante au niveau de l'implémentation du domaine

des zonotopes. Plusieurs approches existent pour cela, comme celles proposées par F. Messine [59] par exemple. Nous ne nous sommes cependant pas concentrés sur cet aspect dans notre travail, et nous ne présentons donc pas ces approches en détails.

De part leur définition, il est évident que la somme de formes affines ou leur multiplication par une constante produit une forme affine, ce qui explique pourquoi le domaine des zonotopes est aussi précis sur l'exemple de la figure 4.2 : toutes les expressions y apparaissant sont exactement manipulables par les raisonnements symboliques du domaine. Ce n'est cependant pas le cas de toutes les expressions de notre langage. En effet, la multiplication de deux formes affines peut produire un polynôme de degré supérieur à un, ne satisfaisant donc pas la forme attendue. De la même manière, la racine carrée et la division de formes affines ne produisent pas nécessairement une forme affine. Les opérations qui conservent les formes affines sont appelées des *opérations linéaires*. Les autres sont, à l'opposé, des opérations non linéaires. Pour ces dernières, il est donc nécessaire de définir des opérateurs abstraits introduisant une surapproximation.

Cependant, afin de pouvoir parler de la correction de ces opérateurs abstraits, encore faut-il avoir défini une fonction de concrétisation pour les formes affines, notée $\gamma_{\mathbb{AF}} : \mathbb{AF} \mapsto \mathcal{P}(\mathbb{R})$, où $\mathbb{AF}$ est l'ensemble des formes affines. Avant de donner sa définition, nous introduisons ici quelques notations. Pour toute variable $x \in \mathbb{V}$, nous dénotons $\hat{x}$ la forme affine qui lui est associée et α_i^x le i-ème coefficient de $\hat{x}$. De plus, par abus de langage, la notation $\hat{x}(\varepsilon)$, où $\varepsilon \in [-1\,;\,1]^n$, correspond à l'évaluation de $\hat{x}$ pour une valuation des symboles de bruit donnée. Nous pouvons maintenant définir la fonction de concrétisation pour toute forme affine $\hat{x}$:

$$\gamma_{\mathbb{AF}}(\hat{x}) \triangleq \{\hat{x}(\varepsilon) \mid \varepsilon \in [-1\,;\,1]^n\} \tag{4.12}$$

Elle consiste simplement à calculer la valeur associée à $\hat{x}$ pour chaque affectation possible des symboles de bruit.

De plus, nous introduisons deux fonctions, notées $\Phi_{\mathbb{I}}^{\mathbb{AF}} : \mathbb{AF} \mapsto \mathbb{I}$ et $\Phi_{\mathbb{AF}}^{\mathbb{I}} : \mathbb{I} \mapsto \mathbb{AF}$, permettant de convertir des formes affines en intervalles, et réciproquement :

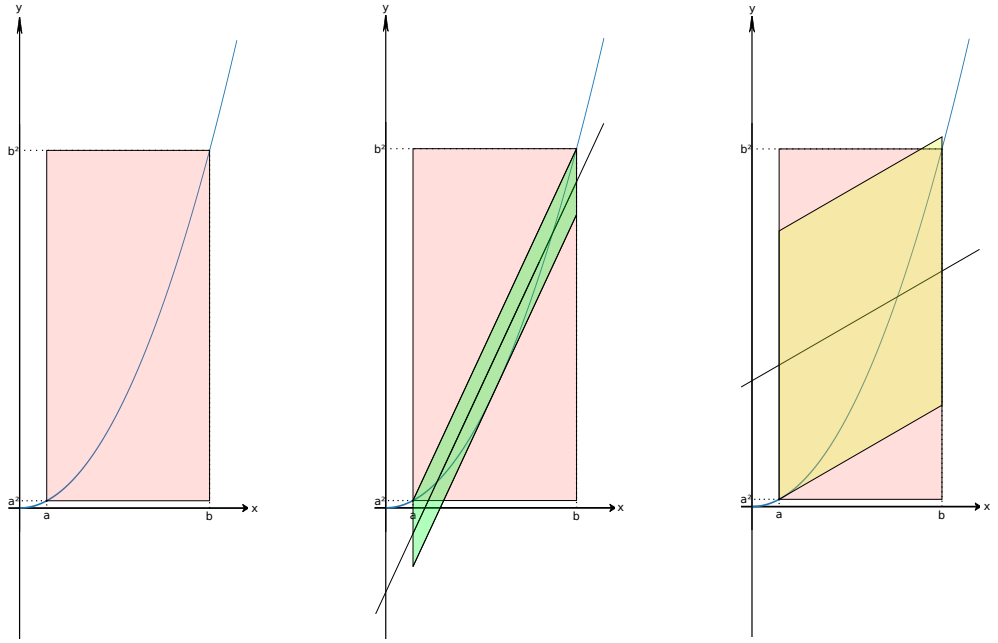
$$\Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x}) \triangleq \left[\alpha_0^x - \sum_{i=1}^{n} |\alpha_i^x|\,;\,\alpha_0^x + \sum_{i=1}^{n} |\alpha_i^x|\right] \tag{4.13}$$

$$\Phi_{\mathbb{AF}}^{\mathbb{I}}([a\,;\,b]) \triangleq \frac{a+b}{2} + \frac{b-a}{2}\varepsilon_{n+1} \tag{4.14}$$

La première calcule le minimum et le maximum de $\hat{x}$. La seconde représente l'intervalle sous la forme *centre plus déviation*. Elle est par ailleurs obligée d'introduire un nouveau symbole de bruit, afin de ne pas ajouter de fausses corrélations avec d'autres formes affines. Pour la même raison, nous avons $\alpha_i = 0$ pour tout i entre 1 et n.

4.2.1 Linéarisation des opérateurs arithmétiques

Afin de donner une meilleure compréhension des problèmes engendrés par les opérations non linéaires et de construire une intuition de comment les résoudre, commençons avec un exemple simple et cherchons à analyser le programme $y = (\phi)\,(x \times x)$; pour x compris entre 1 et 4. Le domaine des intervalles nous permet de déterminer trivialement que $y \in [1\,;16]$. L'ensemble des couples (x, y), correspondant donc à une surapproximation de l'ensemble des états concrets du programme, est illustrée à la figure 4.5a. Nous notons ici que, bien que l'intervalle pour y ne contient aucun réel ne pouvant être obtenu par le programme, l'ensemble des couples (x, y) lui contient beaucoup d'états ne pouvant pas être obtenu en réalité, par exemple le couple $(0.5, 0.5)$.



(a) Intervalles. (b) Forme affine précise. (c) Forme affine imprécise.

FIGURE 4.5 – Approximations de la fonction x^2.

L'expression $x \times x$ n'étant pas linéaire, le domaine des zonotopes ne peut pas en donner de représentation symbolique exacte. La seule information que le domaine peut représenter est une relation de la forme $y = ax + b$ entre les deux variables x et y. Géométriquement, cela veut dire que nous voudrions représenter l'ensemble des états concrets par une droite, ce qui en manquerait nécessairement certains, puisque la fonction exacte décrit une parabole, et ne serait donc pas correct. Cependant, le principe même du domaine des zonotopes est de manipuler symboliquement des incertitudes, et l'erreur que l'on commettrait en associant

une droite de la forme $ax + b$ à la variable y peut parfaitement être vue comme une incertitude, non pas introduite par les entrées, mais par l'analyse elle-même. Ainsi, nous pouvons représenter symboliquement y comme l'expression $ax + b + \delta\eta_{m+1}$ où η_{m+1} est un nouveau symbole de bruit de perturbation et $\delta \in \mathbb{R}$ son coefficient. Cette expression symbolique est correcte, c'est-à-dire que tous les états concrets de notre programme sont contenus dans l'ensemble des pairs $(x, ax + b + \delta\eta_{m+1})$, si nous choisissons δ correctement, ce qu'illustre la figure 4.5b. Le choix de a et b aura, quant à lui, un impact sur la précision de l'analyse. En effet, comme l'illustre la figure 4.5c, le choix de la droite peut avoir un impact conséquent sur l'incertitude introduite, et donc sur la valeur de δ. Il est cependant important de constater que nos exemples d'approximation de $f(x) = x^2$ par des formes affines présentent un inconvénient par rapport aux intervalles. En effet, bien que les formes affines permettent d'éliminer une grande partie des états machines impossibles que les intervalles propagent à cause de leurs imprécisions (les zones rouges non recouvertes dans les figures 4.5b et 4.5c), elles propagent également des états machines impossibles que les intervalles ne propagent pas. En particulier, la forme affine de la figure 4.5b ne permet plus de prouver que x^2 est positif ! Définir des opérateurs abstraits pour les opérations non linéaires demande donc nécessairement de faire un compromis entre deux métriques d'évaluation de la précision : l'incertitude vis-à-vis de la fonction approximée et la sur-approximation sur la projection de la forme affine résultante, c'est-à-dire sur l'ensemble des valeurs que peut prendre $f(x)$. Cependant, l'objectif des formes affines et du domaine des zonotopes est de permettre de garder traces des relations entre les variables. Dans le cas des opérations non linéaires, cela se traduit par trouver l'hyperplan décrivant au mieux l'opération, et donc minimisant l'incertitude introduite : c'est cette métrique qui nous intéressera tout particulièrement par la suite.

Ainsi, déterminer un opérateur abstrait correct pour les opérations non linéaires de notre langage consiste à définir comment choisir les différents coefficients de l'expression linéarisée de manière à minimiser l'incertitude introduite, puis à déterminer cette incertitude et à l'ajouter, en utilisant un nouveau symbole de perturbation, à l'expression linéarisée pour obtenir une forme affine linéarisée correcte. Dans notre exemple définissant une variable à partir d'une seule autre, l'expression linéarisée correspondait à une droite. Dans le cas général d'une opération n-aire $op(x_1, \ldots, x_n)$, il s'agira de trouver l'hyperplan $\sum_{i=1}^{n} a_i x_i + b$ minimisant l'incertitude. Cette dernière caractérisant à quel point l'hyperplan approxime correctement l'opération, nous utilisons la notion standard de distance sur l'espace des fonctions pour la mesurer. Cette dernière peut être définie, pour toute paire de fonctions, comme le maximum de la différence absolue entre leurs images. Dans

notre cas, cela donne la définition suivante pour l'incertitude δ :

$$\delta \triangleq \sup_{(x_1,\ldots,x_n)\in\mathbb{R}^n} \left| \left(\sum_{i=1}^{n} a_i x_i + b \right) - op(x_1, \ldots, x_n) \right| \tag{4.15}$$

et la recherche de l'hyperplan minimisant δ peut s'exprimer comme le problème d'optimisation consistant à déterminer les coefficients $(a_1, \ldots, a_n, b)$ minimisant l'incertitude :

$$\inf_{(a_1,\ldots,a_n,b)\in\mathbb{R}^{n+1}} \sup_{(x_1,\ldots,x_n)\in\mathbb{R}^n} \left| \left(\sum_{i=1}^{n} a_i x_i + b \right) - op(x_1, \ldots, x_n) \right| \tag{4.16}$$

Cependant, cette définition n'est pas utilisable directement en pratique. En effet, les techniques d'optimisation permettant de résoudre ce genre de problème sont souvent soit limitées à certains types d'expressions, soit très coûteuses, soit fondées sur des heuristiques ne nous permettant pas de garantir la correction de la solution. Il est donc nécessaire de définir, pour chacune des opérations non linéaires de notre langage, une méthodologie spécifique permettant d'approcher le plus possible, et le plus rapidement possible, l'hyperplan idéal que calculerait la résolution du problème d'optimisation 4.16 sans avoir à le résoudre.

Opérations unaires

L'approximation de fonctions de $\mathbb{R}$ dans lui-même par des objets plus simples étant un domaine largement étudié des mathématiques, nous disposons d'outils puissants pour les opérations unaires. En particulier, l'approximation par une droite que nous cherchons à faire n'est qu'un cas particulier *d'approximation polynomiale*. Il existe plusieurs approches pour approximer des fonctions par des polynômes, comme les séries de Taylor par exemple. Ici, nous nous concentrons sur une méthodologie fondée sur le *théorème d'équioscillation de Chebyshev* [57] et présentée dans [86]. Bien que notre langage ne contienne qu'une opération unaire nécessitant une linéarisation, à savoir la racine carrée, nous présenterons la méthodologie en toute généralité, afin de faciliter de futures extensions du langage.

Commençons par présenter le théorème d'équioscillation de Chebyshev, qui caractérise la meilleure approximation polynomiale d'une fonction continue f sur un intervalle donné. Nous ne présentons pas la preuve ici, car elle est longue, complexe et ne nous servira pas par la suite. Un lecteur désireux d'en apprendre plus peut se référer à [57].

Théorème 4.2.1. Soit $f : [a\,;b] \mapsto \mathbb{R}$ une fonction continue. Soit δ_n le minimum des écarts absolus maximaux avec f pour tout polynôme p de degré inférieur ou égal à n :

$$\delta_n = \inf \left\{ \sup_{x\in[a\,;b]} |f(x) - p(x)| \right\}$$

La meilleure approximation polynomiale de degré n, notée p_n, de f, c'est-à-dire le polynôme minimisant δ_n, respecte nécessairement le schéma suivante : p_n est l'unique polynôme de degré inférieur ou égal à n qui surestime et sous-estime en alternance f d'exactement δ_n au moins $n+2$ fois. Les valeurs de $[a\,;b]$ pour lesquelles l'erreur est exactement δ_n sont appelées les *points d'incertitude maximale* de p_n.

Exemple 4.2.1. La meilleure approximation affine, c'est-à-dire polynomiale de degré 1, d'une fonction continue quelconque présente son erreur maximale en au moins trois points, ce qui est illustré à la figure 4.6. La droite rouge est la meilleure approximation affine de la courbe bleue dans l'intervalle $[-a\,;a]$. L'incertitude maximale d est atteinte en $x=-a$, $x=a$ et $x=0$.

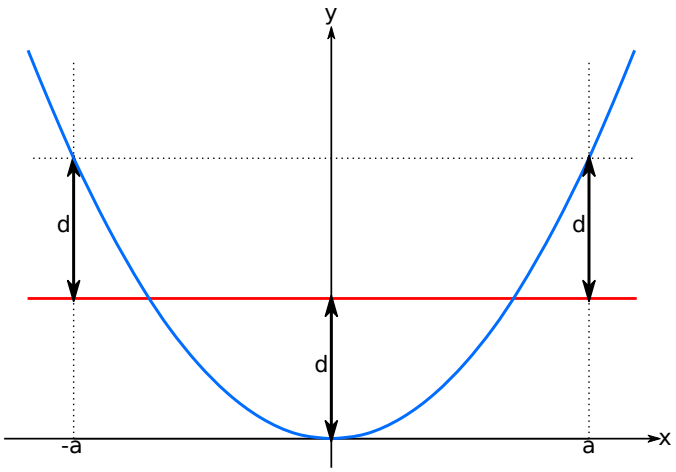


FIGURE 4.6 – Illustration des trois points d'incertitude maximale d'une approximation affine.

Ce théorème nous donne une caractérisation de la meilleure approximation affine d'une fonction continue au sens de la minimisation de l'incertitude introduite. Cependant, ni l'énoncé ni sa preuve n'apportent de méthodologie constructive permettant de déterminer concrètement cette meilleure approximation. Dans le cas général, il n'est d'ailleurs pas possible de définir un algorithme construisant la meilleure approximation polynomiale d'une fonction continue f quelconque. Uewichitrapochana et Surarerks [86] ont cherché à résoudre ce problème en proposant des solutions pour trouver analytiquement les points d'incertitude maximale de la meilleure approximation affine dans deux cas. Le premier cas, celui qui va nous intéresser ici, survient quand la fonction est soit entièrement convexe, soit entièrement concave sur tout l'intervalle $[a\,;b]$. Le second cas, plus complexe mais permettant de traiter les fonctions trigonométriques notamment, prend en considération les fonctions pour lesquelles il existe un $c\in\mathbb{R}$ tel que $f(c-x)=-f(c+x)$

pour tout $x \in \mathbb{R}$. Nous ne présentons ici que le premier cas, cependant, il nous semblait important d'évoquer le second afin de souligner les possibles extensions du langage que le domaine des zonotopes seraient en mesure de traiter.

Afin de déterminer la meilleure approximation affine d'une fonction f continue et qui est soit entièrement convexe, soit entièrement concave sur un intervalle $[a\,;b]$, il est nécessaire de pouvoir construire, pour tout triplet (x_0, x_1, x_2) de $[a\,;b]$ tel que $x_0 < x_1 < x_2$, la droite $p(x) = \alpha x + \beta$ telle que les écarts entre f et p en x_0, x_1 et x_2 soient égaux en valeurs absolues et alternés en signe.

Théorème 4.2.2. Soit $f : [a\,;b] \mapsto \mathbb{R}$ une fonction continue. Soit x_0, x_1, x_2 trois points de $[a\,;b]$ tels que $x_0 < x_1 < x_2$. La droite $p(x) = \alpha x + \beta$ où les coefficients α et β de p sont définis par

$$\alpha \triangleq \frac{f(x_2) - f(x_0)}{x_2 - x_0} \qquad \text{et} \qquad \beta \triangleq \frac{r + f(x_1) - \alpha x_1}{2}$$
$$\text{avec} \qquad r \triangleq \frac{x_2 f(x_0) - x_0 f(x_2)}{x_2 - x_0}$$

satisfait la propriété suivante :

$$f(x_0) - p(x_0) = p(x_1) - f(x_1) = f(x_2) - p(x_2) \tag{4.17}$$

De plus, la valeur absolue de ces écarts, notée δ, peut être calculée comme suit

$$\delta = \frac{|r - f(x_1) + \alpha x_1|}{2}$$

Démonstration. L'égalité entre l'écart en x_0 et celui en x_2 découle facilement de la définition de la pente α :

$$\begin{aligned}
\alpha = \frac{f(x_2) - f(x_0)}{x_2 - x_0} &\iff \alpha(x_2 - x_0) - \beta = f(x_2) - f(x_0) - \beta \\
&\iff f(x_0) - (\alpha x_0 + \beta) = f(x_2) - (\alpha x_2 + \beta) \\
&\iff f(x_0) - p(x_0) = f(x_2) - p(x_2)
\end{aligned}$$

L'égalité entre l'écart en x_1 et celui en x_0, et donc en x_2, découle elle de la définition de β :

$$\begin{aligned}
&\beta = \frac{r + f(x_1) - \alpha x_1}{2} \\
&\iff 2\beta = r + f(x_1) - \alpha x_1 = \frac{x_2 f(x_0) - x_0 f(x_2)}{x_2 - x_0} + f(x_1) - \alpha x_1 \\
&\iff 2\beta = \frac{(x_2 - x_0) f(x_0)}{x_2 - x_0} - \frac{f(x_2) - f(x_0)}{x_2 - x_0} x_0 + f(x_1) - \alpha x_1 \\
&\iff 2\beta = f(x_0) - \alpha x_0 - \alpha x_1 + f(x_1) \\
&\iff f(x_0) - p(x_0) = p(x_1) - f(x_1)
\end{aligned}$$

Enfin, δ est l'écart absolu entre f et p en x_0, x_1 et x_2. La définition présentée dans le théorème est ainsi obtenu en calculant la valeur absolue de $p(x_1) - f(x_1)$. □

Le théorème 4.2.2 nous permet de construire une droite satisfaisant les conditions d'alternance et d'écarts absolus constants pour n'importe quel triplet de points. Il nous reste cependant à pouvoir déterminer trois points pour lesquels cet écart est maximal, c'est-à-dire à déterminer les points d'incertitude maximale.

Théorème 4.2.3. Soit $f : [a\,;b] \mapsto \mathbb{R}$ une fonction continue et dérivable au moins deux fois. Si la dérivée seconde de f est de signe constant sur $[a\,;b]$, c'est-à-dire qu'elle est soit entièrement convexe, soit entièrement concave, alors a, b et u sont trois points d'incertitude maximale de la droite $p(x)$ construite par le théorème 4.2.2 avec

$$f'(u) = \frac{f(b) - f(a)}{b - a} \tag{4.18}$$

Démonstration. Tout d'abord, l'existante d'un $u \in [a\,;b]$ satisfaisant l'égalité 4.18 découle directement du théorème des accroissements finis, l'unicité découlant elle du fait que f est soit entièrement convexe, soit entièrement concave sur $[a\,;b]$. Ensuite, l'égalité des écarts absolus en a, b et u provient directement de la construction de p par le théorème 4.2.2. Ainsi, il ne nous reste qu'à démontrer que cet écart absolu est bien maximal. Il suffit pour cela d'étudier la fonction $g(x) = f(x) - p(x)$. Nous avons directement que $g'(x) = f'(x) - \alpha$ et $g''(x) = f''(x)$. Comme $f''(x)$ est de signe constant, disons positif pour simplifier la démonstration, $g'(x)$ est monotone croissante sur $[a\,;b]$. De plus, par l'égalité 4.18, nous avons :

$$g'(u) = f'(u) - \alpha = \frac{f(b) - f(a)}{b - a} - \alpha = \alpha - \alpha = 0$$

Ainsi, $g'(x)$ est de signe négatif entre a et u, et positif entre u et b. Nous en déduisons que u est un minimum de g sur $[a\,;b]$, et que, comme $g(a) = g(b)$, a et b sont des maximums de g. □

Nous disposons ainsi de tous les éléments permettant de construire la meilleure approximation affine pour toute fonction continue et entièrement convexe ou concave, ainsi que pour déterminer l'incertitude que cette approximation induit. Il ne nous reste plus qu'à l'appliquer à la racine carrée. Ainsi, la forme affine associée à l'expression $\mathbf{sqrt}(x)$, pour x une expression associée à la forme affine $\hat{x}$, et dont les valeurs sont comprises entre a et b deux réels positifs différents, est la suivante :

$$\alpha\hat{x} + \frac{1}{2}\left(\alpha\sqrt{ab} + \frac{1}{4\alpha}\right) + \frac{1}{2}\left|\alpha\sqrt{ab} - \frac{1}{4\alpha}\right|\varepsilon_{n+1} \tag{4.19}$$

avec α la pente de la droite reliant $(a, \sqrt{a})$ et $(b, \sqrt{b})$, telle qu'énoncée au théorème 4.2.2. En effet, la fonction $f(x) = \sqrt{x}$ est évidemment entièrement convexe sur $\mathbb{R}^+$, et ses points critiques sur $[a\,;b]$ sont a, b et u tels que

$$\frac{1}{2\sqrt{u}} = \frac{\sqrt{b}-\sqrt{a}}{b-a} = \frac{1}{\sqrt{a}+\sqrt{b}} \iff 2\sqrt{u} = \sqrt{a}+\sqrt{b} \iff u = \frac{\left(\sqrt{a}+\sqrt{b}\right)^2}{4}$$

Le théorème 4.2.2 nous donne ainsi :

$$\begin{aligned}
\alpha &= \frac{\sqrt{b}-\sqrt{a}}{b-a} = \frac{1}{\sqrt{a}+\sqrt{b}} \\
r &= \frac{b\sqrt{a}-a\sqrt{b}}{b-a} = \frac{\sqrt{ab}(\sqrt{b}-\sqrt{a})}{b-a} = \alpha\sqrt{ab} \\
\beta &= \frac{r+\sqrt{u}-\alpha u}{2} = \frac{\alpha\sqrt{ab}+\frac{1}{2\alpha}-\alpha\frac{1}{4\alpha^2}}{2} = \frac{1}{2}\left(\alpha\sqrt{ab}+\frac{1}{4\alpha}\right) \\
\delta &= \frac{|r-\sqrt{u}+\alpha u|}{2} = \frac{|\alpha\sqrt{ab}-\frac{1}{2\alpha}+\alpha\frac{1}{4\alpha^2}|}{2} = \frac{1}{2}\left|\alpha\sqrt{ab}-\frac{1}{4\alpha}\right|
\end{aligned}$$

nous permettant de construire l'expression présentée à l'équation 4.19. Cette expression ne contient bien que des opérations linéaires parfaitement représentables par le domaine des zonotopes, ce qui nous permet de construire une surapproximation correcte de la racine carrée.

Multiplication

La solution présentée précédemment ne peut malheureusement pas être généralisée pour des fonctions à plusieurs paramètres. Ainsi, pour la multiplication comme pour la division, il est nécessaire de trouver d'autres schémas d'approximation. Le cas de la multiplication est cependant abordable d'une manière assez directe. En effet, développer l'expression mathématiques $\hat{x}\hat{y}$ où $\hat{x}$ et $\hat{y}$ sont les formes affines associées à des expressions x et y de notre langage nous donne

$$\hat{x}\hat{y} = \alpha_0^x\alpha_0^y + \sum_{i=1}^{n}(\alpha_0^x\alpha_i^y + \alpha_0^y\alpha_i^x)\varepsilon_i + \left(\sum_{i=1}^{n}\alpha_i^x\varepsilon_i\right)\left(\sum_{i=1}^{n}\alpha_i^y\varepsilon_i\right) \tag{4.20}$$

où les α_i^x sont les coefficients de $\hat{x}$ et les α_i^y ceux de $\hat{y}$. Dans cette formulation, seul le dernier terme est non linéaire : il fait en effet apparaitre des éléments quadratiques de la forme $\varepsilon_i\varepsilon_j$. Ainsi, une solution pour sur-approximer la multiplication avec une forme affine est d'utiliser la forme suivante :

$$\alpha_0^x\alpha_0^y + \sum_{i=1}^{n}(\alpha_0^x\alpha_i^y + \alpha_0^y\alpha_i^x)\varepsilon_i + \beta + \delta\varepsilon_{n+1} \tag{4.21}$$

où β et δ sont deux réels tels que, pour toute valuation des symboles de bruits, il existe une valuation de $\varepsilon_{n+1} \in [-1\,;\,1]$ telle que l'évaluation du terme non linéaire de l'équation 4.20 soit égale à $\beta + \delta\varepsilon_{n+1}$.

Une première solution, couramment utilisée et présentée, par exemple, par E. Goubault et S. Putot [38], est de calculer une sur-approximation du terme non linéaire en utilisant l'arithmétique d'intervalles afin d'ignorer les corrélations entre les symboles de bruit et donc simplifier le calcul. Une application naïve de cette méthode nous donne

$$\beta = 0 \qquad \text{et} \qquad \delta = \left(\sum_{i=1}^{n} |\alpha_i^x|\right)\left(\sum_{i=1}^{n} |\alpha_i^y|\right) \tag{4.22}$$

Cependant, l'utilisation de ces définitions pour β et δ induit une grosse surapproximation dans la majorité des cas, car elles ignorent l'entièreté des relations entre les deux sommes. Il est heureusement tout à fait possible d'améliorer significativement cette méthode en réorganisant et regroupant les différents éléments du terme non linéaire :

$$\left(\sum_{i=1}^{n} \alpha_i^x \varepsilon_i\right)\left(\sum_{i=1}^{n} \alpha_i^y \varepsilon_i\right) = \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i^x \alpha_j^y \varepsilon_i \varepsilon_j = \sum_{i=1}^{n} \alpha_i^x \alpha_i^y \varepsilon_i^2 + \sum_{i \neq j} \alpha_i^x \alpha_j^y \varepsilon_i \varepsilon_j$$

Rendre les termes en ε_i^2 apparents est intéressant car ils sont évalués en $[0\,;\,1]$ par arithmétique d'intervalles, contrairement aux termes en $\varepsilon_i\varepsilon_j$ avec $i \neq j$ qui sont, quant à eux, évalués en $[-1\,;\,1]$. Ainsi, nous obtenons, grâce à cette réorganisation, les valeurs suivantes :

$$\beta = \sum_{i=1}^{n} \frac{\alpha_i^x \alpha_i^y}{2} \qquad \text{et} \qquad \delta = \sum_{i=1}^{n} \frac{|\alpha_i^x \alpha_i^y|}{2} + \sum_{i=1}^{n}\sum_{\substack{j=1 \\ j\neq i}}^{n} |\alpha_i^x \alpha_j^y| \tag{4.23}$$

Exemple 4.2.2. Soient $\hat{x} = 2 + 3\varepsilon_1 - 2\varepsilon_2 + \varepsilon_3$ et $\hat{y} = 1 - 2\varepsilon_1 + 4\varepsilon_2$. En utilisant la version naïve décrite par l'équation 4.22, nous obtenons les coefficients $\beta = 0$ et $\delta = (|3| + |-2| + |1|)(|-2| + |4| + |0|) = 36$. Avec la version améliorée présentée par l'équation 4.23, nous obtenons $\beta = (3 \times (-2) + (-2) \times 4 + 1 \times 0) \div 2 = -7$ et $\delta = (|3 \times (-2)| + |(-2) \times 4|) \div 2 + |3 \times 4| + |(-2) \times (-2)| = 23$.

Cette première solution offre une réponse simple à l'approximation de la multiplication par une forme affine. Cependant, rien ne garantit que l'incertitude qu'elle introduit en approximant le terme non linéaire soit optimale. Pour obtenir cette garantie, il est nécessaire d'utiliser une méthodologie plus complexe, présentée en 2003 par S. Miyajima, T. Miyata et M. Kashiwagi [75] et, plus récemment, par I. Skalna et M. Hladìk [80]. L'idée derrière cette solution repose sur trois éléments : d'abord, le fait de calculer les coefficients β et δ revient à chercher les extremums

de la fonction $f(u, v) = uv$ pour u et v appartenant à un ensemble *convexe* de points du plan, c'est-à-dire pour lequel tout segment reliant deux de ces points est nécessairement contenu dans l'ensemble ; ensuite, f présente la particularité que pour tout ensemble convexe du plan, ses extremums se trouvent nécessairement sur le bord de l'ensemble ; enfin, le bord des ensembles convexes à considérer dans notre cas n'est composé que de segments simples à déterminer et sur lesquels il est aisé de calculer les extremums de f. Cependant, comme cette solution n'est pas limitée à la multiplication, et que nous avons travaillé à l'améliorer, nous la présenterons plus formellement au chapitre 6, dédié à la présentation de nos contributions visant à améliorer les opérateurs non linéaires binaires de l'arithmétique affine.

Division

Pour la division, la solution standard, présentée, par exemple, par S. Miyajima et M. Kashiwagi [62], consiste à utiliser le fait que la fonction inverse satisfait les hypothèses nécessaires à l'application de la méthode de meilleure approximation des opérations unaires, présentée précédemment, pour exprimer la division comme le produit entre son numérateur et l'inverse de son dénominateur. Ainsi, en appliquant cette méthode, la meilleure approximation affine de la fonction $f(x) = x^{-1}$ pour $x \in [a\,;b]$ un intervalle ne contenant pas 0, c'est-à-dire que soit $0 < a < b$ soit $0 > b > a$, est

$$-\frac{x}{ab} + \frac{a+b+2\sqrt{ab}}{2ab} + \frac{|a+b-2\sqrt{ab}|}{2ab}\varepsilon_{n+1} \tag{4.24}$$

Démonstration. Les trois points critiques pour cette fonction sur l'intervalle $[a\,;b]$ sont a, b et $u = \sqrt{ab}$. Ce dernier est obtenu à partir de l'équation 4.18 :

$$\begin{aligned} f'(u) = \frac{f(b)-f(a)}{b-a} &\iff -\frac{1}{u^2} = \frac{\frac{1}{b}-\frac{1}{a}}{b-a} = \frac{\frac{a-b}{ab}}{b-a} = -\frac{1}{ab} \\ &\iff u^2 = ab \iff u = \sqrt{ab} \end{aligned}$$

Les calculs des coefficients α et β, ainsi que de l'incertitude δ s'obtiennent également directement en appliquant le théorème 4.2.2 :

$$\begin{aligned} \alpha &= \frac{b^{-1}-a^{-1}}{b-a} = \frac{\frac{a-b}{ab}}{b-a} = -\frac{1}{ab} \\ r &= \frac{\frac{b}{a}-\frac{a}{b}}{b-a} = \frac{b^2-a^2}{ab(b-a)} = \frac{a+b}{ab} \end{aligned}$$

$$\beta = \frac{r + \frac{1}{u} - \alpha u}{2} = \frac{\frac{a+b}{ab} + \frac{1}{\sqrt{ab}} + \frac{\sqrt{ab}}{ab}}{2} = \frac{a + b + 2\sqrt{ab}}{2ab}$$
$$\delta = \frac{|r - \frac{1}{u} + \alpha u|}{2} = \frac{|\frac{a+b}{ab} - \frac{1}{\sqrt{ab}} - \frac{\sqrt{ab}|}{ab}}{2} = \frac{|a + b - 2\sqrt{ab}|}{2ab}$$

ce qui conclut la preuve. □

Nous disposons ainsi d'opérateurs abstraits permettant de gérer toutes les opérations non linéaires de notre langage. Nous pouvons maintenant construire une abstraction de valeur respectant la définition 3.4.1 en utilisant ces opérations. Il est cependant important de noter que, dans les cas de la division et de la racine carrée, ces opérateurs supposent que leurs entrées sont valides, c'est-à-dire nécessairement positive pour la racine carrée, et avec un dénominateur non nul pour la division. Notre abstraction de valeur devra donc traiter ces cas en amont, et disposer d'une manière de représenter l'intervalle $[-\infty\,;\,\infty]$, ce que ne peuvent pas faire les formes affines.

4.2.2 Les formes affines comme abstractions de valeurs

Maintenant que nous disposons d'une arithmétique correcte adaptée aux formes affines, nous pouvons nous atteler à la construction d'une abstraction de valeurs basée sur elles et respectant la définition 3.4.1. Commençons par étendre $\mathbb{AF}$ avec deux nouvelles valeurs, $\bot_{\mathbb{AF}}$ et $\top_{\mathbb{AF}}$, permettant de représenter respectivement l'ensemble vide et tout $\mathbb{R}$. Nous notons cet ensemble $\mathbb{AF}^+$ et c'est avec lui que nous allons construire notre abstraction de valeurs.

Une intuition importante à prendre en considération est que nous souhaitons refléter dans notre ordre partiel que nous accordons plus d'importance aux symboles centraux qu'à ceux de perturbations. Cette différence de traitement découle du fait que les symboles centraux sont associés aux entrées du programme analysé alors que les symboles de perturbations ne représentent que les incertitudes liées à l'analyse. La manière dont l'ordre partiel définie ci-après reflète cela peut être compris de la manière suivante : une forme affine $\hat{x}$ est plus petite qu'une forme affine $\hat{y}$ si et seulement si les écarts de comportement de leurs parties centrales sont systématiquement compensables en utilisant les symboles de perturbations.

Théorème 4.2.4. La relation $\sqsubseteq_{\mathbb{AF}^+}$ définit comme

$$\begin{aligned} &\forall \hat{x}, \hat{y} \in \mathbb{AF}, \hat{x} \sqsubseteq_{\mathbb{AF}^+} \hat{y} \iff \\ &\quad \forall \varepsilon \in [-1\,;\,1]^n, \forall \eta^x \in [-1\,;\,1]^n, \exists \eta^y \in [-1\,;\,1]^n, \hat{x}(\varepsilon, \eta^x) = \hat{y}(\varepsilon, \eta^y) \end{aligned} \tag{4.25}$$

$$\forall \hat{x} \in \mathbb{AF}, \bot_{\mathbb{AF}} \sqsubseteq_{\mathbb{AF}^+} \hat{x} \wedge \hat{x} \sqsubseteq_{\mathbb{AF}^+} \top_{\mathbb{AF}} \tag{4.26}$$

forme un préordre sur l'ensemble $\mathbb{AF}^+$, c'est-à-dire que $\sqsubseteq_{\mathbb{AF}^+}$ est réflexive et transitive mais pas nécessairement antisymétrique. De ce préordre découle la relation d'équivalence $\hat{x} \sim \hat{y} \iff \hat{x} \sqsubseteq_{\mathbb{AF}^+} \hat{y} \wedge \hat{y} \sqsubseteq_{\mathbb{AF}^+} \hat{x}$. La relation $\sqsubseteq_{\mathbb{AF}^+} / \sim$, c'est-à-dire la relation qui accepte les mêmes paires que notre préordre et qui considère les paires en relation par $\sim$ comme équivalente, forme un ordre partiel pour $\mathbb{AF}^+$. Nous le noterons, par abus de langage, également $\sqsubseteq_{\mathbb{AF}^+}$.

Démonstration. Nous ne nous intéressons ici qu'à prouver les propriétés de l'ordre partiel pour des formes affines. Les cas présentant les valeurs $\bot_{\mathbb{AF}}$ et $\top_{\mathbb{AF}}$ se démontrent en effet trivialement à partir de 4.26. La réflexivité et l'antisymétrie de l'ordre partiel découle directement de sa construction. Il nous reste donc à démontrer la transitivité. Soit $\hat{x}$, $\hat{y}$ et $\hat{z}$ trois formes affines telles que $\hat{x} \sqsubseteq_{\mathbb{AF}^+} \hat{y}$ et $\hat{y} \sqsubseteq_{\mathbb{AF}^+} \hat{z}$. Nous ne considérons ici que le cas où $\hat{x}$, $\hat{y}$ et $\hat{z}$ sont toutes les trois différentes de $\top_{\mathbb{AF}}$ et $\bot_{\mathbb{AF}}$ car le traitement de ces différents cas est trivial. Nous avons donc les deux relations suivantes :

$$\forall \varepsilon \in [-1\,;\,1]^n, \forall \eta^x \in [-1\,;\,1]^n, \exists \eta^y \in [-1\,;\,1]^n, \hat{x}(\varepsilon, \eta^x) = \hat{y}(\varepsilon, \eta^y)$$
$$\forall \varepsilon \in [-1\,;\,1]^n, \forall \eta^y \in [-1\,;\,1]^n, \exists \eta^z \in [-1\,;\,1]^n, \hat{y}(\varepsilon, \eta^y) = \hat{z}(\varepsilon, \eta^z)$$

La seconde relation nous garantit que, pour tout ε et tout η^y, il existe un η^z tel que $\hat{y}(\varepsilon, \eta^y)$ et $\hat{z}(\varepsilon, \eta^z)$ soient égaux. C'est donc en particulier le cas pour tout η^y permettant, pour un ε et un η^x donné, de satisfaire la première relation. Dit autrement, pour tout ε et tout η^x, la première relation nous garantit l'existence d'un η^y tel que $\hat{x}(\varepsilon, \eta^x) = \hat{y}(\varepsilon, \eta^y)$, et la seconde nous garantit l'existence d'un η^z tel que $\hat{y}(\varepsilon, \eta^y) = \hat{z}(\varepsilon, \eta^z)$. La relation $\sqsubseteq_{\mathbb{AF}^+}$ est donc bien transitive, ce qui conclut la preuve. □

La concrétisation et les opérateurs abstraits pour l'arithmétique ont déjà été traités précédemment sur l'ensemble des formes affines $\mathbb{AF}$. La généralisation de leurs définitions sur $\mathbb{AF}^+$ est fait de manière intuitive. En effet, la concrétisation de $\bot_{\mathbb{AF}}$ est nécessairement l'ensemble vide, tandis que celle de $\top_{\mathbb{AF}}$ est $\mathbb{R}$ tout entier. L'application d'un opérateur à des paramètres dont au moins un vaut $\bot_{\mathbb{AF}}$ produit nécessairement $\bot_{\mathbb{AF}}$. Si aucun des paramètres ne vaut $\bot_{\mathbb{AF}}$ mais qu'au moins un vaut $\top_{\mathbb{AF}}$, alors l'évaluation produit $\top_{\mathbb{AF}}$.

Définir des opérateurs abstraits pour les erreurs élémentaires peut également être effectuée très facilement. En effet, il suffit de reprendre les opérateurs du domaine des intervalles et de les composer avec les fonctions de conversions $\Phi_{\mathbb{I}}^{\mathbb{AF}}$ et $\Phi_{\mathbb{AF}}^{\mathbb{I}}$ que nous avons définies en 4.13 et 4.14. Ceci nous donne, par composition d'opérateurs corrects, une abstraction correcte des erreurs élémentaires pour les formes affines. De plus, cela assure que l'introduction d'une nouvelle erreur d'arrondi est faite au travers d'un nouveau symbole de bruit, garantissant son indépendance vis-à-vis du reste de l'analyse. Nous pouvons par ailleurs utiliser

un processus analogue pour construire une fonction d'abstraction pour les formes affines.

L'abstraction de l'union et de l'intersection pose néanmoins un problème. En effet, l'ordre partiel que nous utilisons ne garantit l'existence ni d'une plus petite borne supérieure, ni d'une plus grande borne inférieure, mais seulement d'une infinité de bornes supérieures (respectivement inférieures) minimales (respectivement maximales) incomparables entre elles. La preuve, trop complexe pour que nous la présentions ici, découle d'une propriété beaucoup plus générale prouvée par M. G. Krein et M. A. Rutman [50]. Ainsi, pour ces deux opérations, il est nécessaire de définir des critères permettant de sélectionner quelle abstraction nous voulons utiliser. Ces critères sont arbitraires et guidés par des considérations pratiques. Dans notre cas, les deux critères que nous aimerions utiliser sont la garantie que les concrétisations des formes affine calculées par ces opérateurs ne soient pas moins précises que les concrétisations des intervalles calculés par l'union et l'intersection abstraite de ce domaine, et la garantie que la nouvelle incertitude introduite soit minimale. La définition de tels opérateurs a été largement étudiée [39, 42], nous ne présentons donc ici que leurs définitions sans apporter de preuves sur sa correction ou ses propriétés. De plus, ces opérateurs ne garantissent pas, dans le cas général, la minimisation de la nouvelle incertitude introduite. C'est en effet impossible, et nous utilisons donc les solutions proposées dans [39]. Ces dernières ont, par ailleurs, l'intérêt de nous donner gratuitement un opérateur d'élargissement, simplifiant ainsi la définition de notre abstraction de valeurs.

Théorème 4.2.5. Soit $\hat{x}$ et $\hat{y}$ deux formes affines. La forme affine $\hat{z}$ dont les coefficients sont définis comme :

$$\begin{aligned}
\alpha_0^z &= \mathrm{mid}(\Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x}) \sqcup_{\mathbb{I}} \Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{y})) \\
\alpha_i^z &= \mathrm{argmin}(\Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x}) \sqcup_{\mathbb{I}} \Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{y})) \\
\alpha_{n+1}^z &= \sup\left(\gamma_{\mathbb{I}}(\Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x}) \sqcup_{\mathbb{I}} \Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{y}))\right) - \alpha_0^z - \sum_{i=1}^{n} |\alpha_i^z|
\end{aligned}$$

avec $\mathrm{mid} : \mathbb{I} \mapsto \mathbb{R}$ une fonction évaluant le centre d'un intervalle et $\mathrm{argmin} : \mathbb{I} \mapsto \mathbb{R}$ une fonction évaluant le plus petit réel en valeur absolue d'un intervalle, est une abstraction correcte de l'union dont la projection n'est pas moins précise que $\Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x}) \sqcup_{\mathbb{I}} \Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{y})$.

Théorème 4.2.6. Soit $\hat{x}$ et $\hat{y}$ deux formes affines. La forme affine $\hat{z}$ dont les

coefficients sont définis comme :

$$\begin{aligned}\alpha_0^z &= \text{mid}(\Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x}) \sqcap_{\mathbb{I}} \Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{y}))\\ \alpha_i^z &= \text{argmax}(\Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x}) \sqcap_{\mathbb{I}} \Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{y}))\\ \alpha_{n+1}^z &= \sup\left(\gamma_{\mathbb{I}}(\Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x}) \sqcap_{\mathbb{I}} \Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{y}))\right) - \alpha_0^z - \sum_{i=1}^{n} |\alpha_i^z|\end{aligned}$$

avec mid : $\mathbb{I} \mapsto \mathbb{R}$ une fonction évaluant le centre d'un intervalle et argmax : $\mathbb{I} \mapsto \mathbb{R}$ une fonction évaluant le plus grand réel en valeur absolue d'un intervalle, est une abstraction correcte de l'intersection dont la projection n'est pas moins précise que $\Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x}) \sqcap_{\mathbb{I}} \Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{y})$.

L'opérateur abstrait de l'union présentée en 4.2.5 présente également de bonnes propriétés de convergence. En effet, des applications successives réduisent nécessairement les coefficients de tous les symboles à l'exception du dernier introduit, nous amenant vers un comportement de plus en plus proche des intervalles. Cependant, afin de garantir une convergence en temps fini, il est nécessaire de diverger à $\top_{\mathbb{AF}}$ si la projection de la forme affine construite par itérations succesives dépasse d'un intervalle donné. L'ajout de cette limite artificielle permet d'utiliser l'union comme un opérateur d'élargissement pour les formes affines. L'état de l'art [23] présente également différentes méthodes et techniques d'itération permettant d'améliorer la précision de l'invariant obtenu. Nous ne les présentons pas ici car nous ne nous sommes pas intéressés à l'amélioration de l'analyse des boucles.

4.2.3 Problème, limitation et raffinement du domaine

Nous avons défini une abstraction de valeurs fondée sur les formes affines et permettant de générer un domaine abstrait pour notre analyse. Cependant, ce dernier souffre d'un problème handicapant. En effet, l'application point-à-point de l'ordre partiel présenté au théorème 4.2.4 ne permet pas de garantir l'inclusion géométrique de deux états abstraits considérés comme ordonnés. Prenons un exemple pour illustrer ce propos. Soit $\rho_1^\sharp$ et $\rho_2^\sharp$ deux états abstraits associant les formes affines suivantes aux variables x et y :

$$\rho_1^\sharp = \begin{Bmatrix} x \leftarrow 1 - 4\varepsilon_1 \\ y \leftarrow 3 + 2\varepsilon_1 + 0.5\varepsilon_2 \end{Bmatrix} \quad \text{et} \quad \rho_2^\sharp = \begin{Bmatrix} x \leftarrow 3 - \varepsilon_1 - 7\varepsilon_3 \\ y \leftarrow 1.5 + \varepsilon_1 + 4\varepsilon_3 \end{Bmatrix}$$

En utilisant l'ordre partiel point-à-point, nous avons $\rho_1^\sharp \sqsubseteq \rho_2^\sharp$. En effet, $\hat{x_1}$, la forme affine associée à x dans $\rho_1^\sharp$, est bien plus petite que $\hat{x_2}$, et il en va de même pour $\hat{y_1}$ et $\hat{y_2}$. Cependant, si nous représentons graphiquement les ensembles de couples (x, y) que représentent ces états abstraits, ils sont partiellement disjoints, ce qui est illustré par la figure 4.7.

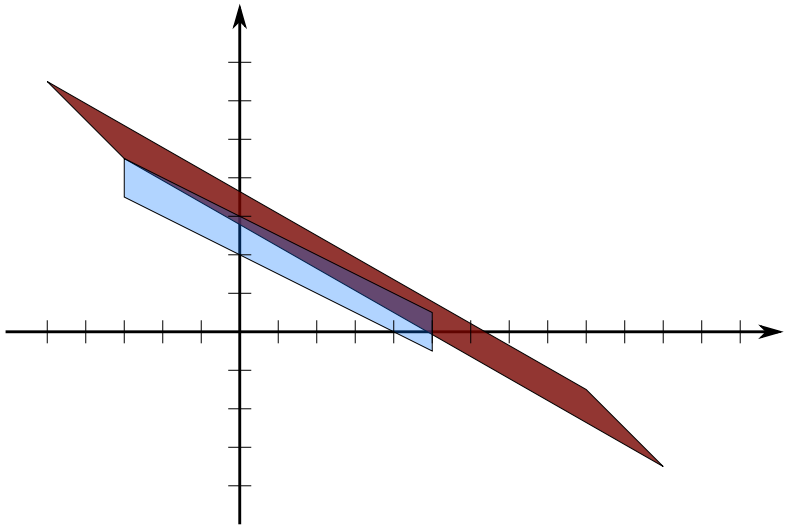


FIGURE 4.7 – Deux états abstraits géométriquement partiellement disjoints mais considérés comme ordonnés par l'ordre partiel point-à-point.

Cet ordre partiel n'est ainsi pas utilisable en pratique, car il ne permet pas de représenter la notion centrale en interprétation abstraite, qui est celle de la sur-approximation de l'information. La problème vient du fait que, par construction, un ordre partiel point-à-point ne prend pas en considération d'éventuelles relations entre les différentes variables. Ainsi, il est nécessaire d'utiliser, au niveau du domaine abstrait, l'ordre partiel introduit par E. Goubault et S. Putot[39, 42] et fondé sur une notion de norme. Cependant, ce changement d'ordre partiel n'impacte pas les opérateurs abstraits du domaine. En particulier, pour le supremum et l'infimum, les définitions que nous avons présenté des opérateurs abstrait pour l'union et l'intersection correspondent en réalité à une restriction à une seule variable des opérateurs de supremum et d'infimum associés à l'ordre partiel de E. Goubault et S. Putot.

En plus de ce problème au niveau de l'ordre partiel, le domaine abstrait crée à partir de notre abstraction de valeurs présente une limitation. En effet, il ne dispose en l'état d'aucune méthode pour réduire un état abstrait lors de l'interprétation d'une condition. L'idée générale, présentée par K. Gorbal, E. Goubault et S. Putot [47, 36], est de réduire non pas sur les formes affines mais les intervalles des symboles de bruit qui les composent. Par exemple, la condition $x \leq 1$ avec x une variable associée à la forme affine $\hat{x} = 1 + 2\varepsilon_1$ peut être correctement représentée en réduisant l'intervalle associé au symbole ε_1 à $[-1\,;0]$. Cette solution présente un avantage important : elle permet de réduire toutes les formes affines partageant des relations avec celles impliquées dans une condition en un coup. Dans notre exemple, si nous avions une seconde variable y associée à $\hat{y} = 2 - \varepsilon_1$, alors l'inter-

prétation de la condition $x \leq 1$ impacterait également y et permettrait de prouver que, si x est plus petit que 1, alors y est plus petit que 2.

Cette méthodologie de gestion des conditions implique certains changements sur l'ordre partiel et les opérateurs abstraits du domaine. Cependant, ces changements n'impactent pas les travaux théoriques présentés dans ce manuscrit, et nous ne présentons donc pas en détail le fonctionnement de l'approche. Un lecteur désireux d'en apprendre plus peut lire la thèse de Khalil Ghorbal [36]. Par contre, l'implémentation de cette méthode a demandé des efforts particuliers, que nous présentons au chapitre 8.

Deuxième partie

Contributions

Chapitre 5

Un produit réduit entre erreurs absolues et relatives

Comme nous l'avons vu au chapitre précédent, nous pouvons construire une analyse d'erreurs numériques pour notre langage pour un domaine abstrait satisfaisant la définition 3.4.1. Cependant, en particulier pour des domaines construits par dessus des abstractions de valeur respectant la définition 3.4.1, l'analyse ainsi construite est naïve et produit des résultats imprécis, en particulier pour ce qui nous intéresse, à savoir les bornes d'erreurs. En effet, les opérateurs arithmétiques abstraits sont construits par composition des opérateurs arithmétiques des valeurs abstraites servant de base, ce qui induit, dans la grande majorité des cas, une non-optimalité. L'état de l'art s'est principalement intéressé à développer des opérateurs plus précis, et donc moins performants, pour des domaines abstraits plus riches et expressifs. C'est l'une des motivations à l'utilisation du domaine des zonotopes et de ses formes affines par exemple. Ces efforts sont, pour la plupart, concentrés sur l'amélioration des bornes d'erreurs absolues, et permettent d'obtenir de très bons résultats, mais au prix de temps d'analyse souvent élevés.

Cependant, comme nous l'avons vu au chapitre 1, les erreurs d'arrondi introduites par les calculs flottants sont bornées en erreur relative, et pas en erreur absolue. Ainsi, une autre voie permettant de combler cette non optimalité est de s'intéresser à améliorer les bornes d'erreurs absolues et relatives en même temps, et de profiter du fait que ces deux erreurs sont les deux faces d'une même pièce pour compenser les faiblesses de l'une par les forces de l'autre. En effet, l'équation 2.16, permettant d'exprimer la sémantique approximée à partir de la sémantique exacte et des sémantiques d'erreurs, met en avant une relation forte entre ces deux notions d'erreurs, relation que nous avons cherchée à exploiter dans les travaux présentés dans cette section. Un exemple permettant d'illustrer l'intérêt d'une analyse croisée est de chercher à évaluer l'erreur absolue de l'expression $t \div (t + 1)$ dans le format *binary64*, en utilisant l'arithmétique d'intervalle et avec t un flottant, sans

erreur associée donc, et appartenant à $[0\,;\,999]$. En effet, une analyse utilisant les opérateurs abstraits naïfs borne l'erreur absolue de cette expression comme suit :

$$\begin{aligned}
\mathcal{E}_a^{\phi}(t \div (t+1)) &= \frac{(t+1).\mathcal{E}_a^{\phi}(t) - t.\mathcal{E}_a^{\phi}(t+1)}{\widetilde{(t+1).(t+1)}} + \Gamma_a^{\phi}(\widetilde{t} \div \widetilde{(t+1)}) \\
&= \frac{[1\,;\,1000]\,[0\,;\,0] - [0\,;\,999]\,[-5.68\times10^{-14}\,;\,5.68\times10^{-14}]}{([1\,;\,1000] + [-5.68\times10^{-14}\,;\,5.68\times10^{-14}])\,[1\,;\,1000]} \\
&\qquad + [-5.68e-14\,;\,5.68e-14] \\
&= \frac{[-5.68\times10^{-11}\,;\,5.68\times10^{-11}]}{[1\,;\,1000000]} + [-5.68e-14\,;\,5.68e-14] \\
&= \left[-5.68\times10^{-11}\,;\,5.68\times10^{-11}\right]
\end{aligned}$$

Cependant, comme nous le détaillerons par la suite, le calcul de l'erreur absolue d'une division peut être exprimé naturellement en utilisant l'erreur relative de son dénominateur. Cette simple réécriture permet d'améliorer significativement les bornes, réduisant l'intervalle associé à l'erreur absolue à $[-1.67e^{-13}\,;\,1.67e^{-13}]$, comme nous l'expliquerons par la suite. Ainsi, cette idée, qui est notre première contribution et qui a mené à la publication [54], permet d'améliorer grandement la précision de l'analyse sans nécessité de domaines abstraits ou d'opérateurs abstraits complexes, ce qui permet de construire des analyses d'erreurs numériques à la fois précises et efficaces.

La présentation de cette contribution contient trois parties. Premièrement, nous montrerons comment définir un simple *produit réduit* entre les deux erreurs, applicable après chaque opération, et permettant d'améliorer la précision en combinant les informations déduites par chaque erreur. Nous en profiterons également pour améliorer l'exploitation de ce produit réduit, en l'intégrant directement à la formulation de la sémantique d'erreur absolue pour la division et la racine carrée. Deuxièmement, nous présenterons comment améliorer le calcul des erreurs élémentaires en prenant en considération la modélisation améliorée de l'arrondi que nous présentions à l'équation 1.3. Enfin, nous montrerons dans une troisième partie qu'il est possible d'utiliser ce produit réduit lors de l'évaluation des conditions afin de réduire les bornes d'erreurs absolues lors de l'évaluation des branches d'un test. Bien qu'en grande partie générique vis-à-vis du domaine abstrait utilisé, le développement initial de cette contribution a été axé autour des intervalles. En particulier, la partie dédiée aux erreurs élémentaires est spécifique aux intervalles. Ainsi, notre présentation sera effectuée en toute généralité, mais nos exemples et la deuxième section de ce chapitre se fonderont sur les intervalles.

5.1 Produit réduit et opérateurs abstraits

La définition 2.16 explicite comment représenter, dans le monde concret, la valeur approximée à partir de la valeur exacte et des erreurs. Elle permet également d'expliciter chacune des erreurs à partir de l'autre (nous reprenons les notations simplifiées introduites lors de la présentation des sémantiques concrètes d'erreurs au chapitre 2) :

$$\mathcal{E}_a^\phi(e) = e \times_\omega \mathcal{E}_r^\phi(e) \qquad \mathcal{E}_r^\phi(e) = \frac{\mathcal{E}_a^\phi(e)}{e}_\omega \tag{5.1}$$

Dit autrement, cela signifie qu'il est toujours possible, dans le concret, de déterminer l'une des erreurs à partir de l'autre. Malheureusement, le passage vers la sémantique abstraite ne permet pas de conserver ces égalités. Cependant, sa correction nous permet malgré tout de conserver un lien intéressant : pour tout état abstrait $\rho^\sharp$ et tout état concret ρ appartenant à la concrétisation de $\rho^\sharp$, nous avons

$$\mathcal{E}_a^\phi(e) \in \gamma_d\left(e^\sharp \times^\sharp \mathcal{E}_r^{\phi\sharp}(e)\right) \tag{5.2}$$

$$\mathcal{E}_r^\phi(e) \in \gamma_d\left(\mathcal{E}_a^{\phi\sharp}(e) \div^\sharp e^\sharp\right) \tag{5.3}$$

où $e^\sharp$, $\mathcal{E}_a^{\phi\sharp}(e)$ et $\mathcal{E}_r^{\phi\sharp}(e)$ correspondent respectivement aux sémantiques abstraites exactes, d'erreur absolue et d'erreur relative. Pour le domaine des intervalles par exemple, ces abstractions s'obtiennent, comme nous le décrivions à la section 3.4 du chapitre 3, en interprétant dans l'abstrait les différentes sémantiques concrètes.

Les équations $e^\sharp \times^\sharp \mathcal{E}_r^{\phi\sharp}(e)$ et $\mathcal{E}_a^{\phi\sharp}(e) \div^\sharp e^\sharp$ sont ainsi respectivement des abstractions correctes de l'erreur absolue et de l'erreur relative. Cela signifie que nous disposons de deux abstractions correctes pour le calcul des erreurs : celles produites par la sémantique abstraite et celles produites par les équations 5.2 et 5.3. Les seuls états concrets pertinents, c'est-à-dire ceux qui peuvent réellement survenir au cours de l'exécution du programme, sont donc ceux qui appartiennent aussi bien à l'une qu'à l'autre des concrétisations de ces deux abstractions. Dit autrement, nous avons pour tout état abstrait $\rho^\sharp$ et tout état concret ρ appartenant à la concrétisation de $\rho^\sharp$:

$$\mathcal{E}_a^\phi(e) \in \gamma_d\left(\mathcal{E}_a^{\phi\sharp}(e)\right) \cap \gamma_d\left(e^\sharp \times^\sharp \mathcal{E}_r^{\phi\sharp}(e)\right) \tag{5.4}$$

$$\mathcal{E}_r^\phi(e) \in \gamma_d\left(\mathcal{E}_r^{\phi\sharp}(e)\right) \cap \gamma_d\left(\mathcal{E}_a^{\phi\sharp}(e)) \div^\sharp e^\sharp\right) \tag{5.5}$$

Ainsi, lorsque nous disposons d'une abstraction correcte de l'intersection pour les valeurs abstraites, nous pouvons effectuer ces calculs entièrement dans l'abstrait, ce qui définit un *produit réduit* entre les erreurs absolues et relatives : à chaque étape, nous calculons l'intersection dans l'abstrait des informations obtenues par chacune des erreurs afin d'améliorer la précision des deux.

Exemple 5.1.1. Cherchons à évaluer les bornes d'erreurs absolues de l'expression $r = (\phi)\,(x \times \mathbf{sqrt}(y))$; en utilisant le format *binary64* et un état abstrait initial $\rho^\sharp$ dans lequel x prend valeur dans l'intervalle $[1\,;5]$, avec une erreur absolue bornée par $[-4\epsilon_M\,;4\epsilon_M]$ et une erreur relative bornée par $[-\epsilon_M\,;\epsilon_M]$ (on rappelle que ϵ_M vaut 2^{-53} pour le format *binary64*), et y prend valeur dans l'intervalle $[2\,;9]$, avec une erreur absolue bornée par $[-8\epsilon_M\,;8\epsilon_M]$ et avec une erreur relative bornée par $[-\epsilon_M\,;\epsilon_M]$.

L'évaluation de la sémantique abstraite exacte pour r par le domaine des intervalles nous donne directement :

$$r^\sharp = x^\sharp \times_{\mathbb{I}} \sqrt[\mathbb{I}]{y^\sharp} = [1\,;5] \times_{\mathbb{I}} \sqrt[\mathbb{I}]{[2\,;9]} = [1\,;5] \times_{\mathbb{I}} \left[\sqrt{2}\,;3\right] = \left[\sqrt{2}\,;15\right] \tag{5.6}$$

L'évaluation des erreurs absolues et relatives de r nécessite de calculer celle de $\mathbf{sqrt}(y)$ d'abord. Une application directe des équations 2.25 et 2.26 dans l'abstrait nous donne

$$\begin{aligned}
\mathcal{E}_a^{\phi^\sharp}(\mathbf{sqrt}(y)) &= \sqrt[\mathbb{I}]{[2\,;9]} \times_{\mathbb{I}} \left(\sqrt[\mathbb{I}]{\frac{[-8\epsilon_M\,;8\epsilon_M]}{[2\,;9]} + 1} - 1\right) +_{\mathbb{I}} \Gamma_a^{\phi^\sharp}(\sqrt{\widetilde{y}}) \\
&= \left[\sqrt{2}\,;3\right] \times_{\mathbb{I}} \left(\sqrt[\mathbb{I}]{[-4\epsilon_M\,;4\epsilon_M] + 1} - 1\right) +_{\mathbb{I}} [-2\epsilon_M\,;2\epsilon_M] \\
&= \left[\sqrt{2}\,;3\right] \times_{\mathbb{I}} \left[\sqrt{1-4\epsilon_M} - 1\,;\sqrt{1+4\epsilon_M} - 1\right] +_{\mathbb{I}} [-2\epsilon_M\,;2\epsilon_M] \\
&= \left[3\left(\sqrt{1-4\epsilon_M} - 1\right) - 2\epsilon_M\,;3\left(\sqrt{1+4\epsilon_M} - 1\right) + 2\epsilon_M\right] \\
&= \left[-8.88e^{-16}\,;8.88e^{-16}\right] \\
\mathcal{E}_r^{\phi^\sharp}(\mathbf{sqrt}(y)) &= \sqrt[\mathbb{I}]{[-\epsilon_M\,;\epsilon_M] + 1} \times_{\mathbb{I}} \left(\Gamma_r^{\phi^\sharp}(\sqrt{\widetilde{y}}) + 1\right) - 1 \\
&= \left[\sqrt{1-\epsilon_M}\,;\sqrt{1+\epsilon_M}\right] \times_{\mathbb{I}} [1-\epsilon_M\,;1+\epsilon_M] - 1 \\
&= \left[(1-\epsilon_M)\sqrt{1-\epsilon_M} - 1\,;(1+\epsilon_M)\sqrt{1+\epsilon_M} - 1\right] \\
&= \left[-1.67e^{-16}\,;1.67e^{-16}\right]
\end{aligned}$$

Sur cette première étape, notre produit réduit ne permet pas d'améliorer les bornes. C'est sur le calcul de la multiplication qu'il va nous permettre de gagner en précision. En effet, une application des équations 2.19 et 2.20 nous donne les bornes suivantes (nous utilisons ici la variable temporaire $z = \mathbf{sqrt}(y)$ pour alléger les

écritures) :

$$\begin{aligned}
\mathcal{E}_a^{\phi\sharp}(r) &= x^\sharp \mathcal{E}_a^{\phi\sharp}(z) +_{\mathbb{I}} z^\sharp \mathcal{E}_a^{\phi\sharp}(x) +_{\mathbb{I}} \mathcal{E}_a^{\phi\sharp}(x)\mathcal{E}_a^{\phi\sharp}(z) +_{\mathbb{I}} \Gamma_a^{\phi\sharp}(\widetilde{x} \times \widetilde{z}) \\
&= [1\,;5] \times_{\mathbb{I}} \left[-8.88e^{-16}\,; 8.88e^{-16}\right] +_{\mathbb{I}} \left[\sqrt{2}\,; 3\right] \times_{\mathbb{I}} [-4\epsilon_M\,; 4\epsilon_M] \\
&\qquad +_{\mathbb{I}} [-4\epsilon_M\,; 4\epsilon_M] \times_{\mathbb{I}} \left[-8.88e^{16}\,; 8.88e^{-16}\right] +_{\mathbb{I}} [-8\epsilon_M\,; 8\epsilon_M] \\
&= \left[-6.66e^{-15}\,; 6.66e^{-15}\right] \\
\mathcal{E}_r^{\phi\sharp}(r) &= \left(\mathcal{E}_r^{\phi\sharp}(x) + 1\right) \times_{\mathbb{I}} \left(\mathcal{E}_r^{\phi\sharp}(z) + 1\right) \times_{\mathbb{I}} \left(\Gamma_r^{\phi\sharp}(\widetilde{x} \times \widetilde{z}) + 1\right) - 1 \\
&= [1 - \epsilon_M\,; 1 + \epsilon_M] \times_{\mathbb{I}} \left[1 - 1.67e^{-16}\,; 1 + 1.67e^{-16}\right] \\
&\qquad \times_{\mathbb{I}} [1 - \epsilon_M\,; 1 + \epsilon_M] - 1 \\
&= \left[-3.89e^{-16}\,; 3.89e^{-16}\right]
\end{aligned}$$

Notre produit réduit permet ici d'améliorer l'erreur absolue. En effet, le calcul $r^\sharp \mathcal{E}_r^{\phi\sharp}(r)$ permet d'obtenir les bornes $\left[\sqrt{2}\,; 15\right] \times_{\mathbb{I}} [-3.89e^{-16}\,; 3.89e^{-16}] = [-5.83e^{-15}\,; 5.82e^{-15}]$ pour l'erreur absolue de r.

Ainsi, ce simple produit réduit permet d'améliorer les bornes des deux erreurs et, ainsi, d'obtenir une analyse plus précise. Cependant, dans les cas de la division et de la racine carrée, il est possible d'aller plus loin et d'intégrer la relation entre les deux erreurs directement dans les sémantiques de ces opérations. Nous allons montrer que ces deux réécritures sont strictement équivalentes aux formulations 2.21 et 2.25 dans le concret. C'est au passage dans l'abstrait que la différence est créée, les réécritures permettant, en pratique, d'être plus précis sans réduire significativement les performances.

Théorème 5.1.1. L'erreur absolue de la division peut être calculée comme

$$\mathcal{E}_a^\phi[\![e_1 \div e_2]\!](\rho) = \frac{\mathcal{E}_a^\phi(e_1) -_\omega e_1 \cdot_\omega \mathcal{E}_r^\phi(e_2)}{\widetilde{e_2}}{}_\omega +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2}) \tag{5.7}$$

Démonstration. Pour prouver que cette réécriture est valide, il nous faut repartir de la définition 2.14 appliquée à la division :

$$\begin{aligned}
\mathcal{E}_a^\phi[\![e_1 \div e_2]\!](\rho) &= \frac{\widetilde{e_1}}{\widetilde{e_2}}{}_\omega -_\omega \frac{e_1}{e_2}{}_\omega +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2}) \\
&= \frac{\widetilde{e_1} \cdot_\omega e_2 -_\omega e_1 \cdot_\omega \widetilde{e_2}}{\widetilde{e_2} \cdot_\omega e_2}{}_\omega +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2})
\end{aligned}$$

Pour la première réécriture, nous avions transformé, en utilisant 2.16, $\widetilde{e_1}$ et $\widetilde{e_2}$ afin d'expliciter les erreurs absolues des deux expressions. Cette transformation ne

permet pas de simplifier le dénominateur, ce qui pose problème lors du passage dans l'abstrait, en particulier avec des domaines abstraites non relationnels comme les intervalles. Ici, pour $\widetilde{e_2}$, il sera plus intéressant d'expliciter son erreur relative :

$$\mathcal{E}_a^\phi[\![e_1 \div e_2]\!](\rho) = \frac{(e_1 +_\omega \mathcal{E}_a^\phi(e_1)) \cdot_\omega e_2 -_\omega e_1 \cdot_\omega e_2 \cdot_\omega (\mathcal{E}_r^\phi(e_2) +_\omega 1)}{\widetilde{e_2} \cdot_\omega e_2}{}_\omega \\ +_\omega \Gamma_{a_\omega}^\phi(\widetilde{e_1} \div_\omega \widetilde{e_2})$$

En effet, cette transformation permet de simplifier les termes e_2 du numérateur et du dénominateur, nous amenant, une fois le numérateur simplifié en développant sa deuxième partie, à l'équation 5.7 attendue. □

Exemple 5.1.2. En utilisant l'équation 5.7 pour évaluer $t \div (t+1)$ avec $t \in [0\,;\,999]$, nous obtenons les bornes suivantes :

$$\begin{aligned}
\mathcal{E}_a^\phi(t \div (t+1)) &= \frac{\mathcal{E}_a^\phi(t) - t.\mathcal{E}_r^\phi(t+1)}{\widetilde{(t+1)}} + \Gamma_a^\phi(\widetilde{t} \div \widetilde{(t+1)}) \\
&= \frac{-\,[0\,;\,999]\,[-1.11 \times 10^{-16}\,;\,1.11 \times 10^{-16}]}{[1\,;\,1000] + [-5.68 \times 10^{-14}\,;\,5.68 \times 10^{-14}]} \\
&\qquad + \left[-5.68 \times 10^{-14}\,;\,5.68 \times 10^{-14}\right] \\
&= \left[-1.67e^{-13}\,;\,1.67e^{-13}\right]
\end{aligned}$$

Comparées aux bornes $[-5.68e^{-11}\,;\,5.68e^{-11}]$ obtenues avec le calcul naïf, cette réécriture nous permet d'améliorer la précision de l'analyse d'un facteur 340 environ.

Théorème 5.1.2. L'erreur absolue de la racine carrée peut être calculée comme

$$\mathcal{E}_a^\phi[\![\mathbf{sqrt}(e)]\!](\rho) = \sqrt[\omega]{e} \times_\omega \sqrt[\omega]{\mathcal{E}_r^\phi(e) +_\omega 1} +_\omega \Gamma_a^\phi(\sqrt[\omega]{\widetilde{e}}) \tag{5.8}$$

Démonstration. Cette réécriture est trivialement obtenue à partir de la définition 2.25 et de la relation 5.1 entre les deux erreurs. □

Exemple 5.1.3. En utilisant cette nouvelle formulation pour la racine carrée, nous pouvons encore améliorer les bornes de l'erreur absolue de l'expression $\mathbf{sqrt}(y)$ dans l'exemple 5.1.1, pour obtenir l'intervalle :

$$\begin{aligned}
\mathcal{E}_a^{\phi\sharp}(\mathbf{sqrt}(y)) &= \sqrt[\mathbb{I}]{[2\,;\,9]} \times_{\mathbb{I}} \left(\sqrt[\mathbb{I}]{[-\epsilon_M\,;\,\epsilon_M] + 1} - 1\right) +_{\mathbb{I}} \Gamma_a^\phi(\sqrt{\widetilde{y}}) \\
&= \left[\sqrt{2}\,;\,3\right] \times_{\mathbb{I}} \left[\sqrt{1-\epsilon_M} - 1\,;\,\sqrt{1+\epsilon_M} - 1\right] +_{\mathbb{I}} [-2\epsilon_M\,;\,2\epsilon_M] \\
&= \left[3\left(\sqrt{1-\epsilon_M} - 1\right) - 2\epsilon_M\,;\,3\left(\sqrt{1+\epsilon_M} - 1\right) + 2\epsilon_M\right] \\
&= \left[-3.96e^{-16}\,;\,3.96e^{-16}\right]
\end{aligned}$$

Cette amélioration permet de réduire, en suivant exactement le même calcul que dans l'exemple 5.1.1, l'erreur absolue de r à $[-4.16e^{-15}\,;\,4.16e^{-15}]$.

Ainsi, nous avons pu, au travers d'un produit réduit et de réécritures sémantiques simples, améliorer significativement la précision de notre analyse d'erreurs numériques sans réduire significativement ses performances, ce que nous montrerons au chapitre 9 par une validation expérimentale. Nous pouvons cependant continuer à améliorer ce produit réduit, en nous intéressant tout d'abord au calcul des erreurs élémentaires.

5.2 Erreurs élémentaires

Les équations 4.8 et 4.9 que nous avons proposées d'utiliser pour abstraire les erreurs élémentaires durant la présentation du domaine des intervalles sont correctes mais peu précises. En effet, elles ne sont fondées que sur la définition simple de l'opérateur d'arrondi présenté à la section 1.2, et ne peuvent donc pas être optimisées pour profiter de la notion d'ufp introduite par la définition 1.3. De plus, ces équations ne profitent pas non plus de la séparation entre nombres normaux et sous-normaux. Afin de résoudre ces problèmes, nous présentons ici des opérateurs abstraits pour les erreurs élémentaires absolues et relatives permettant de prendre en considération toutes ces particularités. Ces opérateurs abstraits ne sont pas définis sur des valeurs abstraites génériques mais uniquement sur des intervalles. Nous faisons ce choix tant par soucis de simplicité que parce que cela nous est suffisant par la suite. En particulier, ces opérateurs seront réutilisables lors de l'utilisation de formes affines comme valeurs abstraites. Enfin, l'opérateur abstrait pour l'erreur absolue est standard [65] alors que celui pour l'erreur relative est une de nos contributions.

En partant de la définition 1.3 de l'arrondi et des définitions 1.5 et 1.6, nous obtenons, dans le concret, les définitions suivantes pour les deux erreurs :

$$\Gamma_a^\phi(x) = \mathrm{ufp}(x)\, e_x + d_x$$
$$\Gamma_r^\phi(x) = \frac{\mathrm{ufp}(x)}{x} e_x + \frac{d_x}{x}$$

dans lesquelles, pour rappel, $e_x \in [-\epsilon_M\,;\,+\epsilon_M]$, $d_x \in [-\delta_M\,;\,+\delta_M]$ et $e_x \times d_x = 0$, avec ϵ_M et δ_M deux constantes fixées par le format. Afin de mieux prendre en considération le comportement des valeurs e_x et d_x, nous introduisons les deux fonctions ϵ et δ suivantes, permettant de calculer un intervalle qui abstrait correctement e_x et d_x pour tout intervalle $[\underline{x}\,;\,\overline{x}]$ abstrayant correctement x et tel que $\underline{x}$ et $\overline{x}$ soient dans les bornes du format ϕ utilisé. Cette vérification peut être

effectuée aisément en calculant l'intersection dans l'abstrait de $[\underline{x}\,;\overline{x}]$ avec l'intervalle contenant l'ensemble des flottants finis du format. Ce dernier découle de la définition 1.1.2 et est défini, pour un format $(\beta, p, emax)$ quelconque, comme $[-(\beta^{-p}-1)\beta^{emax+1-p}\,;+(\beta^{-p}-1)\beta^{emax+1-p}]$. Les définitions de ces fonctions sont les suivantes :

$$\epsilon([a\,;b]) = \begin{cases} [-\epsilon_M\,;+\epsilon_M] & \text{si } \left[+\beta^{1-emax}\,;+\infty\right] \cap_{\mathbb{I}}^{\sharp} [\underline{x}\,;\overline{x}] \neq \bot_{\mathbb{I}}^{\sharp} \\ & \text{ou } [\underline{x}\,;\overline{x}] \cap_{\mathbb{I}}^{\sharp} \left[-\infty\,;-\beta^{1-emax}\right] \neq \bot_{\mathbb{I}}^{\sharp} \\ [0\,;0] & \text{sinon} \end{cases} \tag{5.9}$$

$$\delta([a\,;b]) = \begin{cases} [-\delta_M\,;+\delta_M] & \text{si } [-\beta^{1-emax}\,;+\beta^{1-emax}] \cap_{\mathbb{I}}^{\sharp} [\underline{x}\,;\overline{x}] \neq \bot_{\mathbb{I}}^{\sharp} \\ [0\,;0] & \text{sinon} \end{cases} \tag{5.10}$$

La fonction ϵ (respectivement δ) renvoie l'abstraction correcte $[-\epsilon_M\,;+\epsilon_M]$ (respectivement $[-\delta_M\,;+\delta_M]$) si l'intervalle $[\underline{x}\,;\overline{x}]$ contient des flottants normaux (respectivement sous-normaux), ce qui peut être vérifié simplement en s'assurant que l'intersection entre $[\underline{x}\,;\overline{x}]$ et les deux demi-droites contenant tous les flottants normaux (respectivement entre $[\underline{x}\,;\overline{x}]$ et l'intervalle contenant tous les flottants sous-normaux) est non vide, c'est-à-dire différente de $\bot_{\mathbb{I}}{}^{\sharp}$. Sinon, elle renvoie simplement l'intervalle représentant le singleton zéro. Nous notons que ces deux fonctions renvoient toujours un intervalle symétrique centré en zéro.

Pour le calcul de l'erreur absolue élémentaire, il nous reste à étudier plus en détail la définition 1.2 de la fonction $\mathrm{ufp}(x)$. Comme cette fonction retourne la puissance de deux inférieure la plus proche de $|x|$, il est trivial de remarquer qu'elle est toujours positive, constante par morceaux, décroissante pour $x \leq 0$, croissante pour $x \geq 0$ et symétrique par rapport à l'axe $y = 0$. Ainsi, comme elle est toujours positive, il nous suffit de trouver son maximum sur l'intervalle $[\underline{x}\,;\overline{x}]$, et de le multiplier par l'intervalle $\epsilon([\underline{x}\,;\overline{x}])$ pour obtenir une abstraction correcte de $\mathrm{ufp}(x)\, e_x$. Pour trouver son maximum, il nous suffit d'étudier trois cas possibles :

- soit $[\underline{x}\,;\overline{x}]$ est toujours positif et par monotonie croissante sur $\mathbb{R}^+$, $\mathrm{ufp}(\overline{x})$ est le maximum ;
- soit $[\underline{x}\,;\overline{x}]$ est toujours négatif et par monotonie décroissante sur $\mathbb{R}^-$, $\mathrm{ufp}(\underline{x})$ est le maximum ;
- soit $0 \in [\underline{x}\,;\overline{x}]$ et il suffit de comparer le maximum de la sous partie positive et celui de la sous partie négative.

Ainsi, le maximum de $\mathrm{ufp}(x)$ pour tout $x \in [\underline{x}\,;\overline{x}]$ est le maximum entre $\mathrm{ufp}(\underline{x})$ et $\mathrm{ufp}(\overline{x})$. Par symétrie par rapport à l'axe $y = 0$, cela revient à calculer $\mathrm{ufp}(\max(|\underline{x}|, |\overline{x}|))$. Il en découle que $\mathrm{ufp}(\max(|\underline{x}|, |\overline{x}|))\, \epsilon([\underline{x}\,;\overline{x}])$ est une abstraction correcte de $\mathrm{ufp}(x)\, e_x$. Ainsi, l'opérateur abstrait $\Gamma_a^{\phi\sharp}$ produisant une abstraction correcte de Γ_a^{ϕ} pour tout intervalle $[\underline{x}\,;\overline{x}]$ peut être comme :

$$\Gamma_a^{\phi\sharp}([\underline{x}\,;\overline{x}]) = \mathrm{ufp}(\max(|\underline{x}|, |\overline{x}|))\, \epsilon([\underline{x}\,;\overline{x}]) \sqcup_{\mathbb{I}} \delta([\underline{x}\,;\overline{x}]) \tag{5.11}$$

Nous utilisons une union abstraite plutôt que l'opérateur abstrait de l'addition ici car cela nous permet de prendre en considération que, bien que $[\underline{x}\,;\overline{x}]$ peut contenir à la fois des réels pouvant être arrondis vers des flottants normaux et des réels pouvant être arrondis vers des flottants sous-normaux, aucun réel ne peut être arrondi en même temps vers un flottant normal et un sous-normal. Utiliser une somme ici introduirait donc une surapproximation inutile.

Pour l'erreur relative élémentaire, nous pouvons également améliorer les bornes en prenant la définition 1.3 de l'arrondi en compte. Commençons par la partie concernant les flottants normaux. Comme pour l'erreur absolue élémentaire, nous allons chercher à calculer le maximum de $|\mathrm{ufp}(x) \div x|$ pour tout x appartenant à $[\underline{x}\,;\overline{x}]$. La valeur absolue vient du fait que l'abstraction de e_x est, comme nous le soulignions précédemment, forcément un intervalle symétrique centré en zéro, et que le signe de $\mathrm{ufp}(x) \div x$ n'est donc pas utile. Nous pouvons naturellement utiliser les propriétés de l'ufp pour calculer un encadrement du numérateur et l'arithmétique d'intervalles pour terminer. Cependant, cette solution introduirait de la surapproximation que nous pouvons éviter. En effet, pour tout $x \neq 0$, l'expression $|\mathrm{ufp}(x) \div x|$ peut être réécrite, en utilisant la définition 1.2, comme suit :

$$\left|\frac{\mathrm{ufp}(x)}{x}\right| = \frac{\beta^{\lfloor \log_\beta |x| \rfloor}}{\beta^{\log_\beta |x|}} = \frac{1}{\beta^{\log_\beta |x| - \lfloor \log_\beta |x| \rfloor}} \tag{5.12}$$

Or, comme l'écart entre tout réel x et sa partie entière $\lfloor x \rfloor$ est toujours plus grand que 0 et strictement plus petit que 1. Ainsi, l'intervalle $[\beta^{-1}\,;1]$ est une abstraction correcte de $|\mathrm{ufp}(x) \div x|$ pour tout $x \in \mathbb{R}^*$.

Cette borne est donc utilisable pour n'importe quelle sous partie $[\underline{x}\,;\overline{x}]$ de même signe des réels (le cas où $0 \in [\underline{x}\,;\overline{x}]$ reviendrait à vouloir calculer une erreur relative pour une valeur nulle, ce qui ne fait pas sens). Cependant, nous pouvons légèrement l'améliorer s'il existe un entier n tel que $\beta^n \leq \underline{x}$ et $\overline{x} < \beta^{n+1}$. En effet, dans ce cas, la valeur de $\lfloor \log_\beta |x| \rfloor$ est constante et donc le dénominateur de l'équation 5.12 est monotone et soit strictement croissant si $[\underline{x}\,;\overline{x}]$ est toujours positif, soit strictement décroissant si $[\underline{x}\,;\overline{x}]$ est toujours négatif. Évaluer $|\mathrm{ufp}(m) \div m|$ pour $m = \min(|\underline{x}|, |\overline{x}|)$ permet donc de calculer une borne supérieure à cette expression sur tout l'intervalle $[\underline{x}\,;\overline{x}]$. Ainsi, nous avons :

$$\max_{x \in [\underline{x}\,;\overline{x}]} \left|\frac{\mathrm{ufp}(x)}{x}\right| = \begin{cases} |\mathrm{ufp}(m) \div m| & \text{si } \exists n \in \mathbb{Z}, \beta^n \leq \underline{x} \wedge \overline{x} < \beta^{n+1} \\ 1 & \text{sinon} \end{cases} \tag{5.13}$$

Dans le cas où le format utilisé est soit *binary32*, soit *binary64*, $|\mathrm{ufp}(m) \div m|$ est égal à la mantisse de $\circ^\phi(m)$ multipliée par β^{p-1}, ce qui revient à interpréter la partie de la chaîne de bits encodant la mantisse de $\circ^\phi(m)$ comme un nombre décimal compris entre 1 et 2.

Passons maintenant à la sous expression $d_x \div x$ concernant l'arrondi vers des nombres sous-normaux. Comme précédemment, nous considérons x strictement positif, le cas strictement négatif étant symétrique. Il nous faut ici repartir de la définition 1.3. Dans cette dernière, si x est arrondi vers un nombre sous-normal, nous avons :

$$\circ^{\phi}(x) = x + d_x \iff d_x = \circ^{\phi}(x) - x \iff \frac{d_x}{x} = \frac{\circ^{\phi}(x)}{x} - 1 \tag{5.14}$$

Ainsi, il nous suffit pour borner $d_x \div x$ d'étudier le rapport entre un réel et le flottant le représentant, quand ce dernier est sous-normal. Comme nous l'avons illustré sur le format $\mathbb{F}_m$ à la section 1.1, les nombres sous-normaux de n'importe quel format ne sont qu'un multiple du plus petit d'entre eux, à savoir δ_M. Un nombre réel est ainsi arrondi (en utilisant l'arrondi au plus proche) vers le flottant sous-normal $n\delta_M$ si et seulement si :

$$\left(n - \frac{1}{2}\right)\delta_M \leq x \leq \left(n + \frac{1}{2}\right)\delta_M \tag{5.15}$$

Il est important de noter ici que nous ne spécifions pas de stratégie pour gérer les cas limites où x est équidistant de deux flottants consécutifs. Cela ne nous est en effet d'aucune utilité pour déterminer un encadrement correct de $\circ^{\phi}(x) \div x$. Les inégalités 5.15 peuvent également être exprimées comme suit : un nombre réel x est arrondi vers le flottant sous-normal $n\delta_M$ si et seulement si il existe un $t \in \left[-\frac{1}{2}\,;\,\frac{1}{2}\right]$ tel que $x = (n+t)\delta_M$. Ainsi, pour tout $x \in \mathbb{R}^{+*}$ tel que $\circ^{\phi}(x)$ soit sous-normal, il existe un $n \in \mathbb{N}$ tel que nous ayons :

$$\frac{\circ^{\phi}(x)}{x} = \frac{n\delta_M}{(n+t)\delta_M} = \frac{n}{n+t} \quad \text{avec } t \in \left[-\frac{1}{2}\,;+\frac{1}{2}\right] \tag{5.16}$$

Étudions cette expression pour différentes valeurs de n :

- Si $n = 0$ alors $n \div (n+t)$ vaut forcément zéro. En effet, le cas $t = 0$ est impossible car, sinon, x vaudrait également zéro et son erreur relative ne serait pas définie.
- Si $n \geq 1$, alors il est trivial de montrer que son maximum vaut 2 et est atteint quand $n = 1$ et $t = -1/2$.

Ainsi, en prenant en compte la symétrie autour de zéro, nous pouvons borner l'expression $d_x \div x$ par l'intervalle $[-1\,;+1]$. Cette abstraction est correcte, et n'est naturellement utilisée que si l'intervalle $[\underline{x}\,;\,\overline{x}]$ pour lequel nous cherchons à évaluer l'erreur relative élémentaire contient des nombres pouvant être arrondis vers des flottants sous-normaux. Sinon, le singleton zéro est à préférer. De plus, comme pour l'erreur élémentaire absolue, il convient de ne pas additioner les abstractions pour la partie normale et la partie sous-normale, mais de privilégier l'utilisation de l'union abstraite.

5.3 Améliorer les conditions

La définition 3.3.1 de domaine abstrait exige un opérateur permettant d'interpréter l'inégalité, que nous utilisons pour construire la sémantique abstraite des conditions. Cependant, définir cet opérateur est en pratique complexe pour répondre à notre besoin. En effet, d'une part, la plupart des domaines abstraits, et en particulier les domaines abstraits des intervalles et des zonotopes, présentent classiquement de tels opérateurs, mais ces derniers ne sont généralement aptes qu'à traiter des conditions portant sur la sémantique exacte, et ne sont donc pas adaptés à notre analyse. D'autre part, l'état de l'art [21] présente bien des solutions pour traiter certaines conditions portant sur la sémantique approximée. Cependant, définir ces solutions est souvent un processus long et complexe, prouver qu'elles sont correctes vis-à-vis de la norme ANSI/IEEE-754 est une tâche encore plus ardue, et par conséquent, ces solutions ne permettent de traiter qu'une partie des constructions de notre langage. De plus, ni les opérateurs classiques des domaines abstraits, ni les opérateurs adaptés à la sémantique approximée ne permettent de réduire les erreurs.

Ainsi, dans cette section, nous proposons une solution simple pour définir, à partir d'un opérateur abstrait ne traitant que des conditions sur la sémantique exacte, un opérateur abstrait pour l'inégalité permettant de réduire les informations sur la sémantique exacte dans le contexte d'une condition sur la sémantique approximée et de réduire les informations sur la sémantique d'erreur absolue. Nous ne garantissons pas ici l'optimalité de cet opérateur abstrait, juste sa correction. Nous montrerons cependant aux travers d'exemples, mais également au chapitre 9, qu'il donne des résultats très satisfaisants en pratique. Cependant, afin d'aider la compréhension, commençons avec un exemple. Prenons le programme suivant utilisant le format *binary64*

$$x = (\phi)\,(i \times i)\ ;\ \textbf{if}\ x \leq 2.0\ \textbf{then}\ r = (\phi)\,(x)\ ;\ \textbf{else}\ r = (\phi)\,(2.0)\ ;\ \textbf{end}$$

et cherchons à analyser ses erreurs numériques en utilisant le domaine des intervalles comme base, avec i associé à l'intervalle $[1\,;100]$ et ne présentant pas d'erreur. Avant la condition, x est associé à l'intervalle $[1\,;10000]$ et ses erreurs sont associées à l'intervalle suivant pour l'erreur absolue :

$$\begin{aligned}\mathcal{E}_a^{\phi\sharp}(x) &= i \times_{\mathbb{I}} \mathcal{E}_a^{\phi\sharp}(i) +_{\mathbb{I}} i \times_{\mathbb{I}} \mathcal{E}_a^{\phi\sharp}(i) +_{\mathbb{I}} \mathcal{E}_a^{\phi\sharp}(i) \times_{\mathbb{I}} \mathcal{E}_a^{\phi\sharp}(i) +_{\mathbb{I}} \Gamma_a^{\phi\sharp}(\widetilde{i} \times \widetilde{i}) \\ &= \mathrm{ufp}(\max(1, 10000)) \times_{\mathbb{I}} [-\epsilon_M\,;\,\epsilon_M] \sqcup_{\mathbb{I}} [0\,;0] \\ &= 8192 \times \left[-1.11e^{-16}\,;\,1.11e^{-16}\right] = \left[-9.09e^{-13}\,;\,9.09e^{-13}\right]\end{aligned}$$

et à l'intervalle suivant pour l'erreur relative :

$$\begin{aligned}\mathcal{E}_r^{\phi\sharp}(x) &= \left(\mathcal{E}_r^{\phi\sharp}(i)+1\right) \times_{\mathbb{I}} \left(\mathcal{E}_r^{\phi\sharp}(i)+1\right) \times_{\mathbb{I}} \left(\Gamma_r^{\phi\sharp}(\widetilde{i}\times\widetilde{i})+1\right) - 1 \\ &= \max_{v\in[1\ ;10000]} \left|\frac{\mathrm{ufp}(v)}{v}\right| \times_{\mathbb{I}} [-\epsilon_M\ ;\ \epsilon_M] \sqcup_{\mathbb{I}} [0\ ;\ 0] \\ &= \left[-1.11e^{-16}\ ;\ 1.11e^{-16}\right]\end{aligned}$$

Dans la branche **then**, la variable r est trivialement associée à l'intervalle $[1\ ;\ 2]$. Comme l'interprétation naïve des conditions ne permet pas de réduire les erreurs, nous avons également $\mathcal{E}_a^{\phi\sharp}(r) = \mathcal{E}_a^{\phi\sharp}(x)$ et $\mathcal{E}_r^{\phi\sharp}(r) = \mathcal{E}_r^{\phi\sharp}(x)$. Cependant, cela est très imprécis. En effet, intuitivement, les erreurs absolues maximales associées à x sont liées au plus grandes valeurs que peut prendre x. Ainsi, les bornes d'erreurs absolues devraient être plus petites, car il ne reste de x que les petites affectations. Comme nous allons le détailler par la suite, nous pouvons effectivement faire cette réduction grâce au produit réduit entre les deux erreurs. En effet, la relation $\mathcal{E}_a^{\phi}(x) = x\mathcal{E}_r^{\phi}(x)$ est vraie en chaque point de programme, et donc en particulier juste après la réduction de x par la condition. Ainsi nous pouvons réduire les bornes d'erreurs absolues de x (et donc de r) après l'interprétation de la condition en les recalculant comme :

$$[1\ ;\ 2]\left[-1.11e^{-16}\ ;\ 1.11e^{-16}\right] = \left[-2.22e^{-16}\ ;\ 2.22e^{-16}\right]$$

Le produit entre les erreurs permet ici d'améliorer drastiquement les bornes de l'erreur absolue, avec un gain d'un facteur 4100 environ.

Revenons maintenant à une présentation formelle de l'utilisation du produit réduit pour l'interprétation des conditions. Supposons donc que nous ayons un domaine abstrait $(A, \sqsubseteq)$, disposant d'une fonction de concrétisation $\gamma : A \mapsto \mathcal{P}(\Sigma)$ et d'un opérateur abstrait $e_1 \leq^\sharp_{\mathbb{R}} e_2 : A \mapsto A$ tels que, pour toute expression e_1 et e_2, tout état abstrait $\rho^\sharp$ et toute variable $x \in \mathbb{V}$, les propriétés suivantes soient satisfaites :

$$\{\rho_r(x) \mid \rho \in \gamma(\rho^\sharp) \wedge \mathbb{R}[\![e_1]\!](\rho) \leq \mathbb{R}[\![e_2]\!](\rho)\} \subseteq \gamma\left(((e_1 \leq^\sharp_{\mathbb{R}} e_2)(\rho^\sharp))_r(x)\right) \tag{5.17}$$

$$\rho^\sharp_f(x) = ((e_1 \leq^\sharp_{\mathbb{R}} e_2)(\rho^\sharp))_f(x) \tag{5.18}$$

$$\rho^\sharp_{e_a}(x) = ((e_1 \leq^\sharp_{\mathbb{R}} e_2)(\rho^\sharp))_{e_a}(x) \tag{5.19}$$

$$\rho^\sharp_{e_r}(x) = ((e_1 \leq^\sharp_{\mathbb{R}} e_2)(\rho^\sharp))_{e_r}(x) \tag{5.20}$$

Ces propriétés spécifient que $\leq^\sharp_{\mathbb{R}}$ n'a réduit que les informations relatives à la sémantique exacte, et l'a fait d'une manière correcte vis-à-vis de l'interprétation de la condition dans cette sémantique.

Nous souhaitons définir un opérateur abstrait permettant de réduire les informations relatives à la sémantique exacte mais d'une manière correcte vis-à-vis de la sémantique approximée cette fois. Afin de mieux illustrer ce que cela signifie, commençons par un exemple et étudions comment interpréter la condition $b \times c \leq d - e$ en utilisant $\leq^\sharp_{\mathbb{R}}$ mais en respectant la sémantique approximée. Une solution directe consiste à rendre la sémantique approximée apparente de la manière suivante :

$$\widetilde{b \times c} \leq \widetilde{d - e}$$
$$\widetilde{b} \times \widetilde{c} + \Gamma^\phi_a(\widetilde{b} \times \widetilde{c}) \leq \widetilde{d} - \widetilde{e} + \Gamma^\phi_a(\widetilde{d} - \widetilde{e})$$
$$(b + \mathcal{E}^\phi_a(b))(c + \mathcal{E}^\phi_a(c)) + \Gamma^\phi_a(\widetilde{b} \times \widetilde{c}) \leq (d + \mathcal{E}^\phi_a(d)) - (e + \mathcal{E}^\phi_a(e)) + \Gamma^\phi_a(\widetilde{d} - \widetilde{e})$$

Concrètement, cette réécriture consiste à appliquer la définition de la sémantique approximée présentée à la figure 2.3 récursivement sur chaque sous-expression, puis à faire disparaitre les opérateurs d'arrondi en utilisant la définition 1.3 et à réécrire, pour toute variable x, les expressions de la forme $\widetilde{x}$ comme $x + \mathcal{E}^\phi_a(x)$. Cela produit une condition interprétable par $\leq^\sharp_{\mathbb{R}}$ à partir du moment où nous fournissons des abstractions pour les erreurs élémentaires.

Cette réécriture nous permet de produire un état abstrait $\rho^\sharp_2$ identique à l'état de départ $\rho^\sharp$, sauf pour les informations relatives à la sémantique exacte, qui ont été réduites en respectant la sémantique approximée. L'état abstrait résultant est nécessairement correct car cette réécriture est strictement équivalente à la condition originale dans le concret et l'opérateur $\leq^\sharp_{\mathbb{R}}$ satisfait la propriété 5.17. Il est également non optimal. En effet, remplacer la sémantique approximée par sa réécriture mettant en avant les erreurs va nécessairement introduire une sur-approximation dans l'abstrait, qui va être propagée au travers de la réduction. Ce n'est cependant pas un problème car nous cherchons à définir un opérateur correct et, si possible, précis, mais que notre but n'est pas l'optimalité car cette dernière est difficile à obtenir dès qu'il s'agit de la sémantique approximée.

Cette première étape n'a permis que de réduire les informations relatives à la sémantique exacte. Les informations pour les erreurs n'ont pas changées, ni celles pour la sémantique approximée, ce qui est cependant moins problématique car elles se reconstruisent trivialement à partir de l'exact et des erreurs. Nous pouvons cependant améliorer les choses grâce à la relation entre les deux erreurs. Plus précisément, nous pouvons réduire les informations sur l'erreur absolue. En effet, comme nous le présentions au début de cette section, la relation 5.1, permettant d'exprimer l'erreur absolue comme le produit entre l'exact et l'erreur relative, est vraie à chaque point de programme, et donc, en particulier, après l'interprétation d'une condition. Or, comme les informations sur l'exact ont été potentiellement réduites lors de cette interprétation, il est possible que ce produit soit plus précis que l'erreur absolue pour certaines variables.

Nous disposons ainsi de plusieurs outils permettant d'améliorer notre analyse d'erreurs numériques en profitant du lien existant entre les notions d'erreurs absolues et d'erreurs relatives. Nous présenterons au chapitre 9 une évaluation de cette approche, et mettrons en avant que ces outils permettent de créer une analyse précise, parfois même plus précise que l'état de l'art, tout en restant très simple et peu coûteuse. Cependant, notre première preuve de concept, fondée sur le domaine des intervalles, montre également que l'absence de raisonnement relationnel peut rapidement devenir un problème pour la précision de l'analyse. C'est pourquoi nous avons cherché à définir une seconde analyse utilisant nos améliorations autour du produit réduit, mais présentant également des améliorations relationnelles, tant par l'utilisation et l'amélioration du domaine des zonotopes que par la définition de nouveaux opérateurs, spécifiquement adaptés à l'analyse relationnelle de l'erreur relative.

Chapitre 6

Produit réduit et opérateurs non linéaires

L'analyse des erreurs numériques que nous avons présentée au chapitre 5 permet de tirer parti des relations entre l'erreur absolue et l'erreur relative pour réduire les bornes estimées au travers d'un produit réduit, permettant ainsi d'obtenir une analyse précise sans impacter significativement les performances. Cependant, la précision de cette analyse dépend aussi grandement du domaine abstrait sur lequel elle est construite, ce qui amène naturellement à vouloir utiliser des domaines plus précis, comme le domaine des zonotopes, adapté à l'analyse de l'erreur absolue. Cependant, comme nous l'avons montré au chapitre 4, les formes affines, qui composent le domaine des zonotopes, présentent de vrais difficultés vis-à-vis des opérations non linéaires. En effet, sur ces opérations, les opérateurs abstraits des formes affines permettent de conserver une partie des relations entre les opérandes et le résultat, mais cela se fait au détriment de la concrétisation de ce dernier, pouvant être moins précis que le résultat obtenu en utilisant l'arithmétique d'intervalles.

La solution adoptée majoritairement dans l'état de l'art, comme dans l'outil *Fluctuat* [29] par exemple, pour limiter ce problème consiste à utiliser un produit réduit entre les formes affines et les intervalles. Cependant, ce produit réduit est le plus souvent assez naïf, et, en particulier, ne redéfinit pas les opérateurs arithmétiques abstraits afin de tenir compte des informations portées par les intervalles. Dans ce chapitre, nous présentons deux améliorations des opérateurs arithmétiques abstraits profitant du produit réduit. La première consiste à utiliser les formes affines pour calculer un encadrement par intervalle optimal de la division. La seconde consiste à tenir compte des intervalles des opérandes pour améliorer l'encadrement de l'incertitude introduite par la multiplication et la division.

Ce chapitre est séparé en deux sections. Dans la première, nous présentons un nouvel algorithme permettant de calculer un encadrement par intervalle optimal

de la division entre deux formes affines. Dans la seconde, nous présentons comment tenir compte des intervalles pour améliorer l'encadrement de l'incertitude introduite par la multiplication et la division. Ce travail repose sur la méthodologie de calcul de l'incertitude de la multiplication de I. Skalna et M. Hladík [80], que nous avons modifiée pour tenir compte des intervalles. Il est important de noter que ces deux contributions ne prennent pas en considération la méthodologie de gestion des conditions. Améliorer ces contributions afin de combler ce manque fait partie de nos travaux futurs.

Cependant, avant de commencer cette présentation, il nous faut présenter rapidement la fonction de réduction $\Omega : \mathbb{AF}^+ \times \mathbb{I} \mapsto \mathbb{AF}^+ \times \mathbb{I}$ du produit réduit entre les intervalles et les formes affines. Intuitivement, cette dernière se contente d'utiliser la projection en intervalle de la forme affine pour réduire l'intervalle qui lui est associé. Réduire la forme affine à partir de l'intervalle n'est pas particulièrement intéressant, car nous ne pouvons extraire aucune information relationnelle de ce dernier. Or, c'est exactement pour ces informations que les formes affines sont utiles. L'application de la fonction de réduction Ω à la paire $(\hat{x}, [x])$ se contente donc de réduire l'intervalle $[x]$ en calculant son intersection avec la projection de $\hat{x}$. Ainsi, la définition de cette fonction est la suivante :

$$\Omega(\hat{x}, [x]) \triangleq (\hat{x}, [x] \sqcap_{\mathbb{I}} \Phi_{\mathbb{I}}^{\mathbb{AF}}(\hat{x})) \tag{6.1}$$

où, pour rappel, $\Phi_{\mathbb{I}}^{\mathbb{AF}} : \mathbb{AF}^+ \mapsto \mathbb{I}$ est la fonction de projection des formes affines vers les intervalles.

6.1 Bornes optimales de la division

Dans cette section, nous présentons un algorithme permettant de calculer un encadrement optimal par intervalle d'une division entre deux formes affines. La présentation est faite pour le calcul de la borne supérieure, celui pour la borne inférieure étant symétrique. Soit $\hat{x}$ et $\hat{y}$ deux formes affines telles que, pour toute affectation des symboles de bruit, $\hat{y}$ est soit toujours strictement positive, soit toujours strictement négative. Notre objectif est de calculer la borne supérieure $\lambda^+ \in \mathbb{R}$ de l'expression $\hat{x} \div \hat{y}$. Pour tout $\varepsilon \in [-1\ ;\ 1]^n$, nous avons ainsi $\hat{x}(\varepsilon) \div \hat{y}(\varepsilon) \leq \lambda^+$, c'est-à-dire $\hat{x}(\varepsilon) - \lambda^+ \hat{y}(\varepsilon) \leq 0$. Cette équation peut être réécrite comme suit :

$$(\alpha_0^x - \lambda^+ \alpha_0^y) + \sum_{i=1}^{n} (\alpha_i^x - \lambda^+ \alpha_i^y)\varepsilon_i \leq 0 \tag{6.2}$$

Cette inégalité est vraie pour toute affectation des symboles de bruit, et donc en particulier pour celle maximisant le terme de gauche. Comme chaque symbole de

bruit évolue dans $[-1\,;1]$, l'affectation maximisant le terme de gauche est celle rendant toutes les expressions $\alpha_i^x - \lambda^+\alpha_i^y$ positives. Ainsi, nous avons :

$$(\alpha_0^x - \lambda^+\alpha_0^y) + \sum_{i=1}^{n} |\alpha_i^x - \lambda^+\alpha_i^y| \leq 0 \tag{6.3}$$

L'inégalité 6.3 ne permet cependant pas de calculer λ^+ directement à cause des valeurs absolues. Pour contourner ce problème, il est nécessaire de pouvoir déterminer le signe de l'expression $e_i(\lambda) = \alpha_i^x - \lambda\alpha_i^y$ en fonction de λ. Précisons immédiatement que nous ne nous intéressons qu'aux indices pour lesquels α_i^y est différent de zéro. Il est en effet inutile de s'intéresser aux autres car e_i est alors constante.

Soit $\lambda_i \in \mathbb{R}$ tel que $e_i(\lambda_i) = 0$ et $\text{sign} : \mathbb{R} \mapsto \mathbb{R}$ une fonction telle que $\text{sign}(x)$ vaut 1 si x est strictement positif, -1 si x est strictement négatif, et 0 si $x = 0$. Nous pouvons déduire les relations suivantes :

$$\lambda^+ > \lambda_i \iff \begin{cases} e_i(\lambda^+) < 0 & \text{si } \alpha_i^y > 0 \\ e_i(\lambda^+) > 0 & \text{si } \alpha_i^y < 0 \end{cases} \iff |e_i(\lambda^+)| = -\,\text{sign}(\alpha_i^y) e_i(\lambda^+)$$

$$\lambda^+ < \lambda_i \iff \begin{cases} e_i(\lambda^+) > 0 & \text{si } \alpha_i^y > 0 \\ e_i(\lambda^+) < 0 & \text{si } \alpha_i^y < 0 \end{cases} \iff |e_i(\lambda^+)| = +\,\text{sign}(\alpha_i^y) e_i(\lambda^+)$$

Concrètement, cela signifie que nous pouvons supprimer les valeurs absolues des termes $|e_i(\lambda^+)|$ en divisant la ligne des réels en deux au niveau de λ_i, et en déterminant sur quelle demi-droite se trouve λ^+. Ainsi, si nous trions tous les λ_i par ordre croissant, nous obtenons un partitionnement de la droite des réels en $n+1$ partitions, chacune nous permettant de déterminer comment réécrire tous les $|e_i(\lambda^+)|$ si λ^+ en fait partie.

Exemple 6.1.1. Soit $\hat{x} = 2 + \varepsilon_1 - 2\varepsilon_2$ et $\hat{y} = 7 + \varepsilon_1 + 4\varepsilon_2 - \varepsilon_3$. Nous avons trivialement $\lambda_1 = 1$, $\lambda_2 = -1 \div 2$ et $\lambda_3 = 0$, ce qui nous donne le partitionnement illustré à la figure 6.1. La droite des réels est divisée en quatre partitions. La première contient tous les réels plus petits que $-1 \div 2$, c'est-à-dire plus petits que le plus petit des λ_i. Si λ^+ appartient à cette partition alors nous avons $e_1(\lambda^+) > 0$, $e_2(\lambda^+) > 0$ et $e_3(\lambda^+) < 0$. La deuxième partition contient tous les réels entre λ_2 et λ_3, nous donnant $e_1(\lambda^+) > 0$, $e_2(\lambda^+) < 0$ et $e_3(\lambda^+) < 0$ si λ^+ en fait partie. Les autres partitions fonctionnent de la même manière.

Dans chaque partition ainsi définie, il est possible de déterminer le signe de $e_i(\lambda^+)$ et donc de supprimer toutes les valeurs absolues et, ainsi, résoudre l'inéquation 6.3. Si la borne obtenue est conforme aux limites de la partition considérée, alors elle correspond à un candidat valide pour la borne supérieure de la division

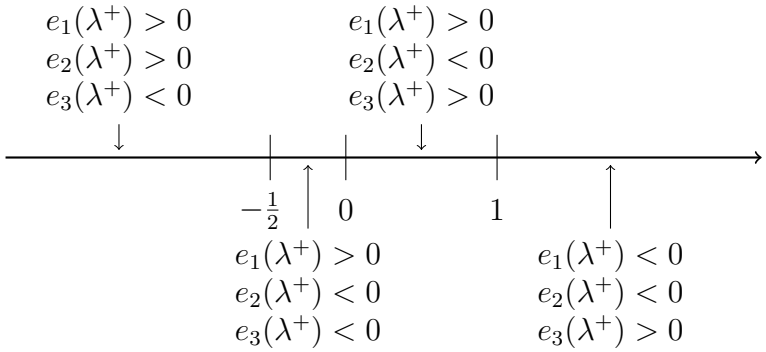


FIGURE 6.1 – Partionnement de la ligne des réels dans l'exemple 6.1.1.

entre $\hat{x}$ et $\hat{y}$. Sinon, cela signifie que l'hypothèse selon laquelle λ^+ appartient à la partition considérée est fausse, et qu'il faut donc essayer sur une autre. Cela nous permet de définir un algorithme très simple pour évaluer la borne supérieure :

1. calculer les λ_i pour tout $i \in [1\,;n]$ et les trier par ordre croissant ;
2. pour chaque morceau de la droite des réels :
   — faire l'hypothèse que λ^+ appartient à cette partition ;
   — simplifier et résoudre l'inéquation 6.3 ;
   — vérifier si la borne calculée satisfait l'hypothèse ;
3. calculer le maximum des candidats valides.

La complexité de cet algorithme est en $O(n \log n)$ à cause du tri.

Exemple 6.1.2. Dans l'exemple 6.1.1, nous obtenons les bornes suivantes sur chaque partition :

— si $\lambda^+ \leq \lambda_2$, alors nous avons $\lambda^+ \geq 1 \div 13$, réfutant l'hypothèse ;
— si $\lambda_2 \leq \lambda^+ \leq \lambda_3$, alors nous avons $\lambda^+ \geq 1$, réfutant l'hypothèse ;
— si $\lambda_3 \leq \lambda^+ \leq \lambda_1$, alors nous avons $\lambda^+ \geq 5 \div 3$, réfutant l'hypothèse ;
— si $\lambda_1 \leq \lambda^+$ alors nous avons $\lambda^+ \geq 3$, validant l'hypothèse.

La borne supérieure est donc 3.

6.2 Encadrement optimal des incertitudes

Dans cette section, nous présentons en détail comment borner les incertitudes introduites par les opérateurs abstraits de la multiplication et de la division sur le produit réduit entre formes affines et intervalles. Cette méthode est fondée sur les travaux de I. Skalna et M. Hladík [80], que nous avons améliorés pour prendre en considération les intervalles associés dans le produit réduit. Elle permet de calculer exactement et efficacement les extremums de toute fonction $f : D \mapsto \mathbb{R}$,

où $D \subseteq \mathbb{R}^2$, sur le sous-ensemble $S \subseteq D$ des paires de valeurs possibles de ces opérandes, lorsque ces dernières sont décrites par des paires $\mathbb{AF}^+ \times \mathbb{I}$ et que f présente certaines caractéristiques que nous allons détailler par la suite.

Dans un premier temps, nous présentons les caractéristiques attendues des fonctions considérées, et montrons que les incertitudes de la multiplication et de la division entre formes affines peuvent être exprimées comme des fonctions présentant ces caractéristiques. Dans un deuxième temps, nous présentons l'algorithme original ne prenant en compte que les formes affines. Enfin, dans un troisième temps, nous présentons notre proposition pour prendre également en considération les intervalles associés.

6.2.1 Fonctions considérées

Soit $f : D \mapsto \mathbb{R}$, où $D \subseteq \mathbb{R}^2$, une fonction dont nous voulons connaître les extremums sur un sous-ensemble S de D. Pour pouvoir utiliser l'algorithme que nous présentons dans la suite de cette section, il faut tout d'abord que f soit continue et dérivable au moins deux fois sur tout S. De plus, il est nécessaire que f soit une *fonction col* sur S. Cependant, afin de pouvoir définir ce que cela signifie, il nous faut commencer par donner une caractérisation des extremums de f comme les images de certains points particulier de D.

Définition 6.2.1. Un point (x_0, y_0) de D est appelé *point stationnaire* de f si les dérivées partielles de f pour chacune des deux composantes de D s'annulent toutes les deux en ce point, c'est-à-dire si :

$$\frac{\partial f}{\partial x}(x_0, y_0) = \frac{\partial f}{\partial y}(x_0, y_0) = 0 \tag{6.4}$$

Théorème 6.2.1. Si f possède des extremums locaux, alors chacun d'entre eux correspond à l'image par f d'un de ses points stationnaires.

Démonstration. Nous ne fournissons ici qu'une idée de la preuve, ce théorème étant très largement connu et utilisé pour l'analyse de fonction. Une preuve plus complète est présentée dans le cours d'analyse fonctionnelle de W. Rudin [72]. Intuitivement, les dérivées partielles d'une fonction f se comportent en tout point x_0 comme des approximations linéaires de f en x_0, la pente de ces approximations correspondant à la dérivée partielle de f en x_0. Si les pentes de ces dernières en un point ne sont pas nulles, cela signifie qu'il est possible de faire un petit déplacement faisant augmenter (ou diminuer) l'image de f. Afin d'atteindre un maximum, il faut donc que les pentes de ces approximations linéaires s'annulent, et donc que toutes les dérivées partielles soient nulles. □

L'inverse n'est cependant pas forcément vrai, un point stationnaire n'étant pas nécessairement associé à un extremum local. Un tel point stationnaire est alors nommé un *point col*.

Exemple 6.2.1. La fonction $f(x, y) = xy$ définie sur $\mathbb{R}^2$ ne possède qu'un seul point critique pour $x = y = 0$. Ce point ne correspond cependant pas à un extremum de f, il suffit pour s'en convaincre de remarquer qu'une variation, même infinitésimale, de x et y sur la droite $y = x$ permet d'obtenir des valeurs de f strictement positives, alors qu'une variation sur la droite $y = -x$ permet d'obtenir des valeurs strictement négatives.

Définition 6.2.2. Soit S un sous-ensemble de D. La fonction f est une *fonction col* sur S si et seulement si l'ensemble de ses points stationnaires appartenant à S sont des points col.

Exemple 6.2.2. La fonction $f(x, y) = x^2$ dispose d'une infinité de points critiques pour $x = 0$, correspondant tous à un minimum local de f. Cependant, sur le sous ensemble défini par $x \in [1\,;3]$ et $y \in [-3\,;3]$, f est une fonction col car aucun de ses points critiques n'appartient au sous ensemble.

Trouver les extremums d'une fonction col sur un sous-ensemble de D est compliqué, car nous n'avons, à priori, aucun indice pour les chercher. Cependant, si le sous-ensemble considéré est convexe et fermé, alors nous pouvons grandement réduire l'espace de recherche. Il convient donc de définir formellement les caractéristiques d'un tel sous-ensemble.

Définition 6.2.3. Soit S un sous-ensemble de D. S est dit *convexe* si, pour toute paire de points $p_0 = (x_0, y_0)$ et $p_1 = (x_1, y_1)$ de S, l'ensemble de points P défini comme

$$P \triangleq \{(x_0 + t(x_1 - x_0), y_0 + t(y_1 - y_0)) \mid t \in [0\,;1]\} \tag{6.5}$$

appartient à S. L'ensemble P correspond au segment reliant p_0 et p_1.

Définition 6.2.4. Soit S un sous-ensemble de D. Un point $p \in D$ est un *point limite* de S si et seulement s'il existe, pour tout $\epsilon > 0$ un point p' de S tel que $|p - p'| < \epsilon$, où $|p|$ correspond à la norme euclidienne. L'ensemble des points limites de S est noté limite(S). S est dit *fermé* s'il contient tous ses points limites.

Exemple 6.2.3. Intuitivement, un point limite est un point qui possède, dans son voisinage proche, des points de S, et ce, peu importe à quel point nous sommes stricts sur la notion de proche. Par exemple, le point 1 est un point limite de l'intervalle ouvert $[0\,;1)$. Cet intervalle n'est par contre pas fermé, car 1 n'en fait justement pas parti.

Définition 6.2.5. Soit S un sous-ensemble convexe et fermé de D. Le *bord* de S, que l'on dénote bord(S), est l'ensemble des points limites de S possédant également des points n'appartenant pas à S dans leur voisinage :

$$p \in \text{bord}\,(S) \iff p \in \text{limite}\,(S) \wedge \forall \epsilon > 0, \exists p' \in D \backslash S, |p - p'| < \epsilon \tag{6.6}$$

Nous pouvons maintenant définir comment restreindre l'espace de recherche des extremums de f sur un sous-ensemble convexe et fermé.

Théorème 6.2.2. Si f est une fonction col sur un sous-ensemble convexe fermé S de D, alors ses extremums sur S se trouvent nécessairement sur le bord de S.

Démonstration. Nous ne fournissons ici qu'un aperçu de la preuve. Une preuve plus complète est disponible dans le cours d'analyse multivariée de J. Stewart [82] par exemple. L'idée est que, en partant de n'importe quel point de S qui ne soit pas sur le bord, il existe forcément au moins une direction vers laquelle un déplacement infinitésimale va augmenter (ou diminuer) la valeur de f, soit parce que le point n'est pas critique et donc qu'une de ses dérivées est non nulle, soit parce que le point est critique, mais est un point col et n'est donc pas un extremum local. En continuant de se déplacer de proche en proche, nous finissons nécessairement par atteindre un point du bord de S. Pour continuer d'augmenter la valeur de f, il ne nous reste comme seule possibilité de nous déplacer le long du bord ou de sortir de S. □

Pour que l'algorithme de recherche des extremums soit utilisable sur une fonction col f, il faut que cette dernière présente une dernière caractéristique : il est nécessaire de pouvoir évaluer les extremums de f pour tout segment de S. C'est, cependant, généralement facile à faire en pratique, car il suffit de se ramener à un cas univarié. Avant de passer à la présentation de l'algorithme en lui-même, commençons par montrer que les incertitudes de la multiplication et de la division entre deux formes affines peuvent être exprimées comme des fonctions col et que nous pouvons évaluer leurs extremums sur des segments.

Multiplication. À partir de la définition 4.20, nous pouvons exprimer la multiplication de deux formes affines comme suit :

$$\hat{x}\hat{y} = \alpha_0^x \hat{y} + \alpha_0^y \hat{x} - \alpha_0^x \alpha_0^y + \Delta \tag{6.7}$$

où Δ représente l'incertitude introduite. Nous pouvons donc, par simple réécriture, exprimer Δ comme une fonction à deux variables :

$$\Delta(x, y) = xy - \alpha_0^x y - \alpha_0^y x + \alpha_0^x \alpha_0^y \tag{6.8}$$

Cette fonction est trivialement continue et dérivable sur $\mathbb{R}^2$. Montrons maintenant que $\Delta(x, y)$ est une fonction col. Pour ce faire, commençons par calculer ses points stationnaires. Les dérivées partielles de $\Delta(x, y)$ sont trivialement calculables :

$$\frac{\partial \Delta}{\partial x}(x, y) = y - \alpha_0^y \quad \text{et} \quad \frac{\partial \Delta}{\partial y}(x, y) = x - \alpha_0^x \tag{6.9}$$

Le seul point annulant les deux dérivées partielles est (α_0^x, α_0^y). Montrons maintenant que ce point est un point col de $\Delta(x, y)$. Pour ce faire, nous pouvons utiliser le *test des dérivées partielles secondes*. Nous ne rentrons pas dans les détails ici, car ce test est classique en analyse de fonctions multivariées. Un lecteur désireux d'en apprendre plus peut lire le cours d'analyse multivariée de J. Stewart [82] par exemple. Ce test consiste, pour toute fonction f, à approximer f par la fonction suivante :

$$P_f(x, y) = \frac{\partial^2 f}{\partial x^2}(x, y)\frac{\partial^2 f}{\partial y^2}(x, y) - \left(\frac{\partial^2 f}{\partial x \partial y}(x, y)\right)^2 \tag{6.10}$$

et à vérifier si $P_f(a, b) < 0$ pour chaque point stationnaire (a, b), permettant de déduire si (a, b) est un point col.

Dans notre cas, il est trivial de montrer que $P_\Delta(x, y)$ est constant et vaut toujours -1. En effet, les dérivées secondes en ∂x^2 et en ∂y^2 sont nulles, et celle en $\partial x \partial y$ est constante et vaut 1. Ainsi (α_0^x, α_0^y) est un point col de $\Delta(x, y)$. Pour pouvoir calculer les extremums de cette fonction sur un sous-ensemble convexe et fermé de $\mathbb{R}^2$, il nous reste ainsi à définir comment les calculer pour tout segment de $\mathbb{R}^2$ entre deux points (x_0, y_0) et (x_1, y_1). En reprenant la définition 6.5, caractérisant un segment au travers d'un unique paramètre t prenant valeur dans $[0\,;1]$, nous pouvons transformer $\Delta(x, y)$ en une fonction univariée sur t :

$$\Delta(t) = (x_0 + \delta_x t)(y_0 + \delta_y t) - \alpha_0^x(y_0 + \delta_y t) - \alpha_0^y(x_0 + \delta_x t) + \alpha_0^x \alpha_0^y \tag{6.11}$$

Ce changement de paramètre permet de revenir à un problème de recherche des extremums d'un polynôme univarié de degré deux, qui est simple à résoudre. L'incertitude introduite par la multiplication de deux formes affines peut ainsi être calculée de manière optimale par l'algorithme présenté en section 6.2.2.

Division. En utilisant la définition 4.24 de l'inverse et la définition 4.20 de la multiplication, nous pouvons exprimer directement la division de deux formes affines comme :

$$\frac{\hat{x}}{\hat{y}} = \frac{\sqrt{\overline{y}\underline{y}}}{\overline{y}\underline{y}}\hat{x} - \frac{\alpha_0^x}{\overline{y}\underline{y}}\hat{y} + \frac{\alpha_0^x \alpha_0^y}{\overline{y}\underline{y}} + \Delta \tag{6.12}$$

où $\underline{y}$ et $\overline{y}$ sont respectivement la borne inférieure et la borne supérieure de $\hat{y}$, et telles que soit $0 < \underline{y} \leq \overline{y}$, soit $\underline{y} \leq \overline{y} < 0$. Exactement comme pour la multiplication,

nous pouvons exprimer l'incertitude Δ comme une fonction bivariée. Cependant, afin de simplifier les formules, nous introduisons les constantes $a \triangleq \sqrt{\overline{y}\underline{y}} \div (\overline{y}\underline{y})$, $b \triangleq -\alpha_0^x \div (\overline{y}\underline{y})$ et $c \triangleq (\alpha_0^x \alpha_0^y) \div (\overline{y}\underline{y})$. L'incertitude introduite par la division peut donc s'exprimer comme :

$$\Delta(x, y) = \frac{x}{y} - ax - by - c \tag{6.13}$$

Cette fonction est continue et dérivable sur $\mathbb{R} \times \mathbb{R}^{+*}$, ainsi que sur $\mathbb{R} \times \mathbb{R}^{-*}$. Montrons maintenant que cette fonction est une fonction col. Pour ce faire, commençons par calculer ses points stationnaires. Les dérivées partielles de $\Delta(x, y)$ sont :

$$\frac{\partial \Delta}{\partial x}(x, y) = \frac{1}{y} - a \quad \text{et} \quad \frac{\partial \Delta}{\partial y}(x, y) = -\frac{x}{y^2} - b \tag{6.14}$$

La dérivée partielle en x est nulle pour tout point (x, y) tel que $y = a^{-1}$. Il faut maintenant déterminer les valeurs de x pour lesquelles $-a^2 x - b = 0$. Cela nous donne directement $x = -ba^{-2}$. Montrons maintenant que le point $(-ba^{-2}, a^{-1})$ est un point col de $\Delta(x, y)$. Comme pour la multiplication, nous utilisons ici le test des dérivées partielles secondes. Calculons les dérivées partielles secondes de $\Delta(x, y)$:

$$\frac{\partial^2 \Delta}{\partial x^2}(x, y) = 0 \;\; ; \;\; \frac{\partial^2 \Delta}{\partial y^2}(x, y) = \frac{2x}{y^3} \;\; ; \;\; \frac{\partial^2 \Delta}{\partial x \partial y}(x, y) = -\frac{1}{y^2}$$

En utilisant ces dérivées partielles dans l'équation 6.10, nous obtenons la fonction $P_\Delta(x, y)$ suivante :

$$P_\Delta(x, y) = 0 \times \frac{2x}{y^3} - \left(-\frac{1}{y^2}\right)^2 = -\frac{1}{y^4}$$

Cette fonction est trivialement strictement négative pour tout point (x, y) et, en particulier, pour $(-ba^{-2}, a^{-1})$. Ce point est donc un point col, et $\Delta(x, y)$ est donc une fonction col, sur $\mathbb{R} \times \mathbb{R}^{+*}$ comme sur $\mathbb{R} \times \mathbb{R}^{-*}$.

Pour terminer, il nous reste à montrer comment calculer les extremums de $\Delta(x, y)$ sur un segment reliant les deux points quelconques (x_0, y_0) et (x_1, y_1). Pour des raisons de simplicité, nous ne présentons le calcul que pour des segments appartenant à $\mathbb{R} \times \mathbb{R}^{+*}$. La méthodologie est similaire pour les segments de $\mathbb{R} \times \mathbb{R}^{-*}$.. Commençons, comme pour la multiplication, par introduire un changement de variable pour transformer $\Delta(x, y)$ en fonction univariée :

$$\Delta(t) = \frac{x_0 + \delta_x t}{y_0 + \delta_y t} - a(x_0 + \delta_x t) - b(y_0 + \delta_y t) - c \tag{6.15}$$

Calculons ensuite la dérivée de cette fonction :

$$\begin{aligned}\frac{d\Delta}{dt}(t) &= \frac{\delta_x(y_0+\delta_y t)-\delta_y(x_0+\delta_x t)}{(y_0+\delta_y t)^2} - a\delta_x - b\delta_y \\ &= \frac{\delta_x y_0 - \delta_y x_0}{(y_0+\delta_y t)^2} - a\delta_x - b\delta_y\end{aligned}$$

Cherchons maintenant les valeurs de t qui annulent cette dérivée. Afin de simplifier les formules, nous introduisons les deux constantes $k = a\delta_x + b\delta_y$ et $r = \delta_x y_0 - \delta_y x_0$:

$$\begin{aligned}\frac{d\Delta}{dt}(t) = 0 &\iff \frac{r}{(y_0+\delta_y t)^2} - k = 0 \iff k(y_0+\delta_y t)^2 - r = 0 \\ &\iff k{\delta_y}^2 t^2 + 2ky_0\delta_y t + k{y_0}^2 - r = 0\end{aligned}$$

On peut trouver les racines de ce polynôme en calculant son discriminant ρ :

$$\rho = (2ky_0\delta_y)^2 - 4 \times k\delta_y^2 \times (ky_0^2 - r) = 4k^2y_0^2\delta_y^2 - 4k^2y_0^2\delta_y^2 - 4k\delta_y^2 r = -4k\delta_y^2 r$$

Si ρ est positif, alors la dérivée de $\Delta(t)$ s'annule aux deux valeurs de t suivantes :

$$t_\pm = \frac{-2ky_0\delta_y \pm \sqrt{-4k\delta_y^2 r}}{2k\delta_y^2} = -\frac{y_0}{\delta_y} \pm \frac{\sqrt{-kr}}{k\delta_y} \tag{6.16}$$

Ainsi, pour déterminer les extremums de $\Delta(t)$, il suffit de l'évaluer pour $t = 0$, $t = 1$ et pour t_+ et t_- si $\rho \geq 0$ et que les racines appartiennent bien à l'intervalle $[0\,;1]$. Nous disposons donc bien d'une méthode pour calculer les extremums de $\Delta(x, y)$ sur un segment, et l'incertitude introduite par la division peut bien être calculée de manière optimale par l'algorithme présenté ci-après.

6.2.2 Algorithme sur les formes affines uniquement

Bien que le théorème 6.2.2 permette de grandement restreindre l'espace de recherche des extremums d'une fonction sur un sous ensemble convexe fermé S, il ne suffit généralement pas pour les déterminer. En effet, dans le cas général, le bord de S est un objet trop complexe pour permettre une recherche algorithmique. Cependant, dans notre contexte, nous ne nous intéressons qu'à des sous ensembles particuliers, définis comme l'ensemble des points $\hat{S}$ que peuvent générer deux formes affines :

$$\hat{S} \triangleq \left\{ \left(\alpha_0^x + \sum_{i=1}^{n} \alpha_i^x \varepsilon_i, \alpha_0^y + \sum_{i=1}^{n} \alpha_i^y \varepsilon_i \right) \mid \varepsilon \in [-1\,;1]^n \right\} \tag{6.17}$$

Ces ensembles sont convexes et fermés. Pour se convaincre de la convexité, il suffit de remarquer que le segment reliant deux points A et B quelconques de $\hat{S}$ peut

être construit simplement en faisant varier les ε_i entre leurs affectations pour A et celles pour B de manière à former une droite. Ces affectations intermédiaires appartiennent nécessairement, pour chaque symbole, à $[-1\,;\,1]$, et donc le segment entre A et B appartient bien à $\hat{S}$. Le caractère fermé, quant à lui, découle du fait que la transformation permettant de construire $\hat{S}$ à partir de $[-1\,;\,1]^n$ est une *transformation affine* qui conserve le caractère fermé et projette le bord du sous-ensemble fermé de départ sur celui du sous-ensemble d'arrivée. Ces particularités des transformations affines sont largement étudiées en *analyse fonctionnelle*, et il serait trop long de détailler tous les éléments nécessaires à construire les preuves. Un lecteur désireux d'en apprendre plus peut lire le cours de W. Rudin [72] par exemple.

Ainsi donc, nous admettons sans preuve que $\hat{S}$ est un polygone dans $\mathbb{R}^2$ dont le bord peut être décrit par $2n$ segments, avec n le nombre de symboles de bruit, reliant des sommets correspondant à une affectation des symboles de bruit telle que, pour tout i, nous ayons $\varepsilon_i = 1$ ou $\varepsilon_i = -1$. Nous admettons également que deux sommets sont reliés par un segment du bord si et seulement si les affectations les ayant générés ne diffèrent que sur un seul symbole. La recherche des maximums de f revient donc à étudier f sur des droites. Nous verrons par la suite que c'est en pratique simple à faire pour les fonctions qui nous intéressent. Il reste cependant un problème à résoudre. En effet, il nous faut encore définir un algorithme permettant de déterminer les sommets et les segments formant le tour de $\hat{S}$ sans avoir à énumérer les 2^n sommets de $[-1\,;\,1]^n$. La présentation de cet algorithme est en deux parties. Dans un premier temps, nous présentons comment construire un point de départ à partir duquel nous allons pouvoir déterminer tous les sommets et les segments formant le bord de $\hat{S}$. Dans un second temps, nous présentons comment construire les autres sommets du bord de $\hat{S}$ d'une manière itérative et efficace. Cependant, avant de commencer, nous avons besoin d'ajouter une hypothèse sur $\hat{x}$ et $\hat{y}$: il n'existe pas de $i \in [1\,;\,n]$ tel que $\alpha_i^x = \alpha_i^y = 0$. Si tel est le cas, alors nous pouvons sans perte de généralité ignorer complètement le symbole associé, car il ne participe en rien aux formes affines engendrant $\hat{S}$.

Construction du point de départ. Pour pouvoir commencer l'algorithme de parcours du bord, il nous faut commencer par construire un point de départ.

Théorème 6.2.3. Le point (x_0^+, y_0^+) de $\hat{S}$ construit comme suit :

$$x_0^+ = \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x e_i^0 \quad \text{et} \quad y_0^+ = \alpha_0^y + \sum_{i=1}^{n} \alpha_i^y e_i^0$$

$$\text{avec } e_i^0 = \begin{cases} \text{sign}(\alpha_i^x) & \text{si } \alpha_i^x \neq 0 \\ \text{sign}(\alpha_i^y) & \text{sinon} \end{cases}$$

appartient au bord de $\hat{S}$.

Démonstration. Il est trivial de constater que l'affectation e^0 des symboles de bruit construit un point maximisant la valeur de $\hat{x}$. De plus, s'il existe plusieurs points maximisant $\hat{x}$, alors l'affectation e^0 construit celui qui maximise $\hat{y}$. □

Corollaire 6.2.1. Le point (x_0^-, y_0^-) construit en utilisant l'affectation $-e^0$ appartient également au bord de $\hat{S}$.

Nous disposons ainsi de deux sommets du bord. Ils sont également symétriques par rapport au point (α_0^x, α_0^y). Cette particularité n'est pas spécifique à ces deux points : l'intégralité de $\hat{S}$ présente une symétrie par rapport à (α_0^x, α_0^y).

Théorème 6.2.4. $\hat{S}$ est symétrique par rapport à $c \triangleq (\alpha_0^x, \alpha_0^y)$, c'est-à-dire que pour tout point p de $\hat{S}$, il existe un point $p' \in \hat{S}$ tel que $p - c = -(p' - c)$.

Démonstration. Il suffit de remarquer que, pour toute affectation e des symboles de bruit, les points $(\hat{x}(e), \hat{y}(e))$ et $(\hat{x}(-e), \hat{y}(-e))$ satisfont l'égalité ci-dessus. □

Ainsi, il suffit de construire la moitié du bord de $\hat{S}$, l'autre étant calculable simplement par symétrie. Voyons maintenant comment construire cette moitié du bord, et plus particulièrement la moitié permettant de relier (x_0^+, y_0^+) à (x_0^-, y_0^-) dans le sens antihoraire. Une illustration géométrique de ce que nous entendons par cela est présentée à la figure 6.2. Notons que ce choix est parfaitement arbitraire et que nous aurions pu construire le bord de $\hat{S}$ dans l'autre sens.

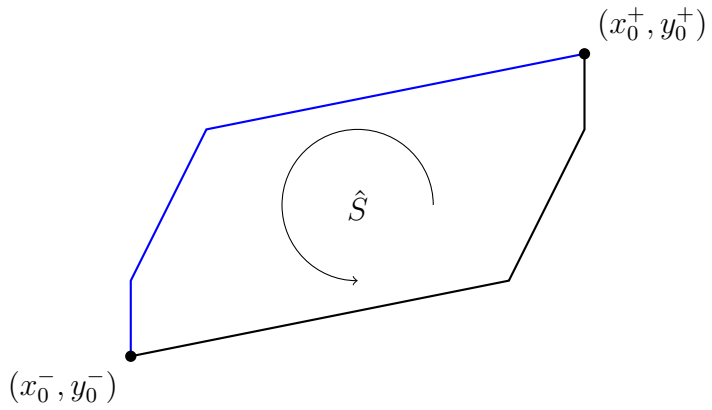


FIGURE 6.2 – Illustration de la moitié du bord construite dans le sens antihoraire.

Construction du bord. Nous l'avons dit précédemment, deux sommets de $\hat{S}$ sont reliés si et seulement si les affectations les ayant générés ne diffèrent que sur un seul symbole. Intéressons nous à caractériser le vecteur reliant deux sommets, et utilisons pour cela le sommet (x_0^+, y_0^+) que nous savons être sur le bord. Soit $k \in [1\,;n]$ et e^k une affectation des symboles de bruit définie comme :

$$\forall i \in [1\,;n]\,, e_i^k = \begin{cases} e_i^0 & \text{si } i \neq k \\ -e_i^0 & \text{sinon} \end{cases} \tag{6.18}$$

La coordonnée k est ainsi l'unique coordonnée à laquelle e^0 et e^k diffèrent. De plus, e^k correspond à un potentiel sommet de $\hat{S}$ car tous ses symboles sont affectés soit à 1, soit à -1. Calculons les coordonnées des vecteurs (δ_x^k, δ_y^k) reliant les points (x_0^+, y_0^+) et $(\hat{x}(e^k), \hat{y}(e^k))$:

$$\delta_x^k = \left(\alpha_0^x + \sum_{i\neq k} \alpha_i^x e_i^0 - \alpha_k^x e_k^0\right) - \left(\alpha_0^x + \sum_{i\neq k} \alpha_i^x e_i^0 + \alpha_k^x e_k^0\right) = -2\alpha_k^x e_k^0$$

$$\delta_y^k = \left(\alpha_0^y + \sum_{i\neq k} \alpha_i^y e_i^0 - \alpha_k^y e_k^0\right) - \left(\alpha_0^y + \sum_{i\neq k} \alpha_i^y e_i^0 + \alpha_k^y e_k^0\right) = -2\alpha_k^y e_k^0$$

L'ensemble de ces vecteurs représentent, une fois mis dans le bon ordre, tous les déplacements qu'il est nécessaire de faire pour relier (x_0^+, y_0^+) et (x_0^-, y_0^-) dans un parcours en sens antihoraire des sommets de $\hat{S}$. Pour s'en convaincre, il suffit de remarquer que se déplacer d'un sommet à ses voisins nécessite de ne modifier qu'un seul symbole, et que comme (x_0^-, y_0^-) est généré par $-e^0$, il est nécessaire de faire varier chaque symbole un part un, ce que représente exactement chaque vecteur (δ_x^k, δ_y^k). La dernière difficulté à résoudre consiste donc à déterminer l'ordre dans lequel il faut suivre les vecteurs (δ_x^k, δ_y^k).

Théorème 6.2.5. Soit s_i la pente du vecteur (δ_x^i, δ_y^i) définie comme :

$$s_i \triangleq \begin{cases} \alpha_i^y \div \alpha_i^x & \text{si } \alpha_i^x \neq 0 \\ +\infty & \text{sinon} \end{cases} \tag{6.19}$$

Soit h_i une suite d'entiers définie pour $i \in [1\,;n]$, prenant valeur dans $[1\,;n]$ et telle que, pour tout i, nous ayons $s_{h_i} \leq s_{h_{i+1}}$. Dit autrement, h_i est la suite des indices permettant d'ordonner les pentes par ordre croissant. La suite de points p_i définie pour $i \in [0\,;n]$ comme :

$$p_0 = (x_0^+, y_0^+) \tag{6.20}$$

$$p_{i+1} = p_i + (\delta_x^{h_{i+1}}, \delta_y^{h_{i+1}}) \tag{6.21}$$

correspond à la suite des sommets de $\hat{S}$ formant la moité du bord de $\hat{S}$ permettant de relier (x_0^+, y_0^+) à (x_0^-, y_0^-) dans le sens antihoraire. En particulier $p_n = (x_0^-, y_0^-)$.

Ébauche de démonstration. L'idée est de remarquer que tout autre ordre des vecteurs (δ_x^i, δ_y^i) que l'ordres croissant (ou décroissant) induirait nécessairement une violation de la convexité de $\hat{S}$. Cela est dû au fait que tout autre ordre ferait nécessairement apparaître le type de schéma illustré géométriquement à la figure 6.3. La droite en rouge reliant les points A et D est supposée appartenir à $\hat{S}$ car ce dernier est convexe. Ce n'est cependant pas le cas si l'on construit les segments formant le bord de $\hat{S}$ avec un autre ordre que l'ordre croissant ou décroissant. □

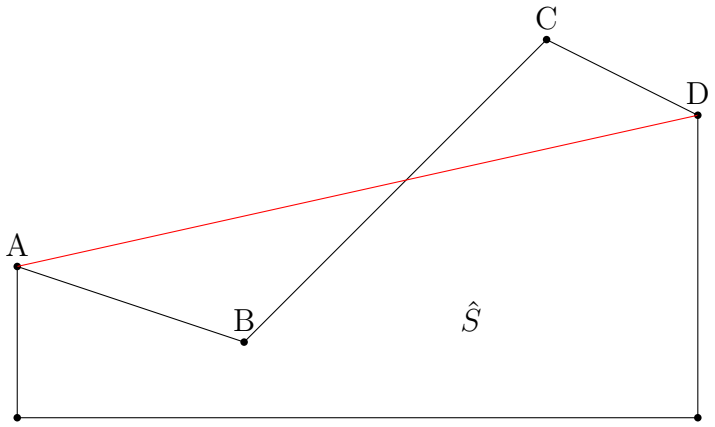


FIGURE 6.3 – Illustration du problème de convexité induit par tout autre ordre que l'ordre croissant ou décroissant.

Ainsi, le théorème 6.2.5 nous donne un algorithme permettant de calculer tous les sommets de $\hat{S}$ dans l'ordre, nous permettant de reconstruire tous les segments formant le bord de $\hat{S}$. Cet algorithme peut être décomposé en quatre phases :

1. calculer le point de départ (x_0^+, y_0^+) ;
2. calculer tous les vecteur (δ_x^i, δ_y^i) et les ordonner par ordre croissant de leurs pentes ;
3. appliquer itérativement chaque vecteur au sommet courant, en commençant par (x_0^+, y_0^+) ;
4. construire la seconde moitié par symétrie.

La complexité de l'algorithme est en $O(n \log n)$, avec n le nombre de symboles de bruit. En pratique, l'algorithme présenté par I. Skalna et M. Hladík est spécifique à la multiplication et profite des symétries de cette opération et de $\hat{S}$ pour améliorer les performances. Nous n'avons présenté ici que la version générale pour deux raisons : la première est que la division ne présente pas les mêmes symétries que la multiplication ; la seconde est que nous ne pouvons pas à la fois utiliser ces symétries et prendre en compte les intervalles. En effet, intuitivement, prendre en

compte les intervalles consiste à chercher les extremums non plus sur le bord de $\hat{S}$, mais sur le bord de l'intersection entre le sous-ensemble $\hat{S}$ et le sous-ensemble B représentant l'ensemble des paires de valeurs que x et y peuvent prendre dans les intervalles. Cette intersection n'est pas nécessairement symétrique, comme l'illustre la figure 6.4. Dans cette dernière, l'opérande x est associé à la forme affine $\hat{x} = \varepsilon_1 + \varepsilon_2$ et à l'intervalle $[x] = [-1\,;2]$, et l'opérande y est associé à $\hat{y} = \varepsilon_1 - \varepsilon_2$ et à $[y] = [-1.5\,;2]$. Le sous-ensemble, représenté en violet hachuré et en pointillé, sur lequel il faut rechercher les extremums est l'intersection entre $\hat{S}$, représenté en bleu et hachuré, et B, représenté en rouge et en pointillé.

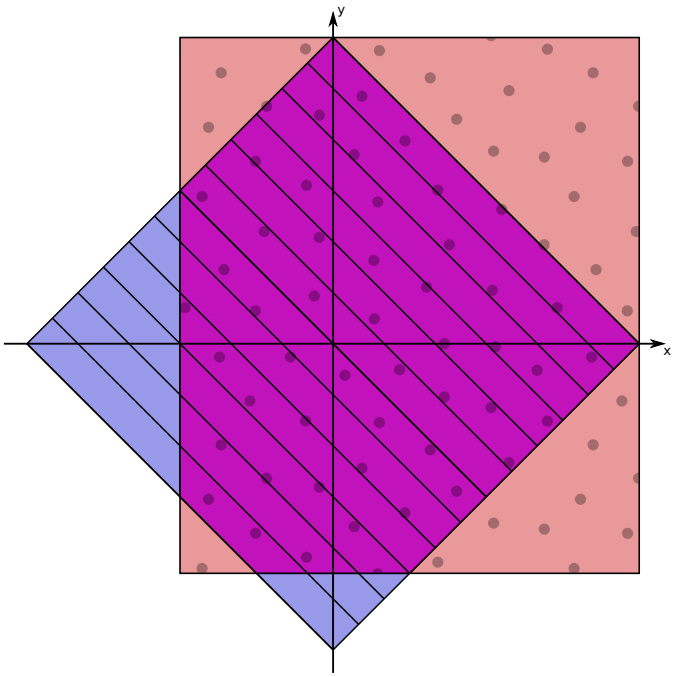


FIGURE 6.4 – Exemple d'intersection entre formes affines et intervalles.

6.2.3 Prendre en compte les intervalles

Notre algorithme permettant de calculer le bord de l'intersection entre le sous-ensemble $\hat{S}$ formé par les formes affines et le sous-ensemble B formé par les intervalles est décrit sous forme d'un pseudo-code présenté à l'algorithme 1. Il consiste en une fonction prenant en entrées le point (x_0^+, y_0^+) et les vecteurs (δ_i^x, δ_i^y) calculés par la version limitée aux formes affines que nous avons présentée à la section précédente, et retourne la liste des sommets de l'intersection. Pour simplifier la présentation, nous faisons l'hypothèse que le point (x_0^+, y_0^+) appartient bien à B. Si ce n'est pas le cas, alors soit nous sommes face à un cas trivial pour lequel

l'intersection est égale à B, soit nous pouvons sélectionner un autre sommet du bord de $\hat{S}$ appartenant à B, satisfaisant ainsi l'hypothèse.

L'algorithme repose sur trois fonctions et deux types auxiliaires qu'il nous faut commencer par présenter. Le premier type dont nous avons besoin sert à représenter le statut du point courant vis-à-vis de B :

$$\textbf{type } \texttt{Status} = \textbf{Inside} \mid \textbf{Out } \mathbb{R}^2 \tag{6.22}$$

Il dispose de deux constructeurs : **Inside** spécifie que le point courant appartient à B, et **Out** p spécifie que le point courant est en dehors de B et que le point par lequel nous sommes sortis de B est p. Le second type sert à représenter le résultat du calcul des potentielles intersections entre un segment et le bord de B :

$$\textbf{type } \texttt{Intersection} = \textbf{Zero} \mid \textbf{One } \mathbb{R}^2 \mid \textbf{Two } \mathbb{R}^2 \times \mathbb{R}^2 \tag{6.23}$$

Il dispose de trois constructeurs : **Zero** spécifie qu'il n'y a pas d'intersection, **One** p spécifie que le point p est l'unique intersection avec le bord de B, et **Two** (p_1, p_2) spécifie qu'il y a deux intersections p_1 et p_2 entre le segment et le bord de B. Présentons maintenant les fonctions auxiliaires. Nous ne faisons ici qu'une présentation informelle de leurs fonctionnements car nous considérons qu'elles sont suffisamment simples pour ne pas nécessiter une présentation détaillée de leurs pseudo-codes. La première fonction dont nous avons besoin est `insideBox` : $\mathbb{R}^2 \mapsto$ Bool qui permet d'évaluer si un point du plan appartient à B. Sa définition consiste à vérifier que chaque coordonnée du point appartient bien à l'intervalle associé à cette coordonnée. La deuxième fonction est `intersectBox` : $\mathbb{R}^2 \times \mathbb{R}^2 \mapsto$ `Intersection` permettant de calculer les potentielles intersections entre un segment et le bord de B. La définition de cette fonction est assez simple : il suffit de calculer l'intersection entre la droite support du segment et celle de chaque face de B, et de vérifier que cette intersection appartient bien à la fois au segment et à la face. Le seul cas particulier à gérer est le cas où le segment est confondu avec une des faces. La fonction retourne alors les deux sommets correspondant aux limites de la partie commune au segment et à la face. De plus, quand il existe deux intersections entre le segment (a, b) et le bord de B, la fonction retourne le résultat **Two** (p_1, p_2) en garantissant que p_1 est l'intersection la plus proche du point a. La troisième fonction est `boxBetween` : $\mathbb{R}^2 \times \mathbb{R}^2 \mapsto$ List $\mathbb{R}^2$. L'appel `boxBetween` (u, v) calcule tous les sommets du bord de B entre u et v lorsque l'on en fait le tour dans le sens antihoraire, et en supposant que u et v appartiennent au bord. Elle retourne la liste de ces sommets dans l'ordre inverse du parcours, en ajoutant u en queue et v en tête.

Exemple 6.2.4. La figure 6.5 est une illustration du fonctionnement de la fonction `boxBetween` pour la boîte B formée par les sommets s_1, s_2, s_3 et s_4. La construction

de la liste des sommets commence par u et finit par v. Seul le sommet s_4, illustré en rouge, n'appartient pas au chemin, dans le sens antihoraire, le long du bord entre u et v. La liste renvoyée par l'appel `boxBetween`(u, v) est donc $v \bullet s_3 \bullet s_2 \bullet s_1 \bullet u \bullet \varepsilon$.

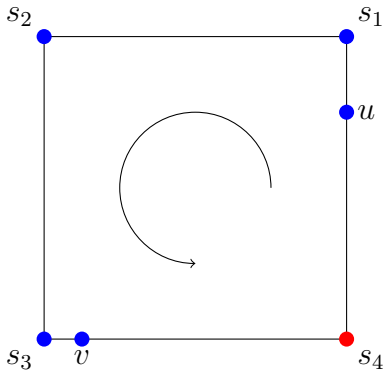


FIGURE 6.5 – Illustration d'un appel à `boxBetween`.

Nous pouvons maintenant présenter l'algorithme en lui-même. Son fonctionnement est assez simple. L'idée est de calculer le bord de $\hat{S}$ exactement comme dans la version originale, mais de vérifier à chaque déplacement si ce dernier nous fait changer de statut vis-à-vis de B, c'est-à-dire nous fait sortir de ou nous fait entrer dans B. En effet, lorsque un déplacement nous fait sortir de B, nous savons que tous les sommets de $\hat{S}$ qui suivent sont en dehors de B, et donc de l'intersection, jusqu'à en trouver un qui soit de nouveau à l'intérieur.

À la place de ces sommets rejetés de $\hat{S}$, il nous faut ajouter des sommets de B afin de compléter le bord de l'intersection. Il nous suffit ici d'ajouter tous les sommets se trouvant, lorsque l'on fait le tour de B dans le sens antihoraire, entre le point de sortie, c'est-à-dire l'intersection entre la face nous ayant fait sortir et B, et le point d'entrée, c'est-à-dire l'intersection entre la face nous faisant rentrer de nouveau et B. Il faut cependant ici faire attention à bien considérer que ce n'est pas parce que les deux sommets d'une face sont en dehors de B que toute la face l'est. Il est en effet tout à fait possible que cette dernière traverse B et ressorte, en passant prêt d'un sommet de B par exemple. Ainsi, il conviendra, pour chaque face dont les deux sommets sont en dehors de B, de vérifier s'il y a zéro, une ou deux intersections entre la face et B et de traiter chaque cas convenablement.

Présentons maintenant en détail l'algorithme. Ce dernier se présente sous la forme d'une boucle itérant sur les vecteurs permettant de construire le bord de $\hat{S}$ et modifiant un état à chaque itération. Cet état est composé de trois variables : `current` est le sommet courant du parcours, `status` est son statut vis-à-vis de B, et `submits` est la liste en cours de construction des sommets de l'intersection. À

Algorithme 1 : reduceWithBox

Entrée : La liste $\{v_i \in \mathbb{R}^2\}_{i\in[1\ ;2n]}$ des vecteurs formant le bord de $\hat{S}$.
Entrée : Le point $(x_0^+, y_0^+) \in \mathbb{R}^2$, correspondant au sommet de départ du bord de $\hat{S}$.
Sortie : La liste des sommets de l'intersection entre $\hat{S}$ et B.

```
submits ← ε ; current ← (x0+, y0+) ; status ← Inside ;
Pour chaque vi de la liste des vecteurs faire
    next ← current + vi ;
    Filtrer status avec
    | Inside ⇒
        submits ← current • submits ;
        Si ¬ insideBox(next) alors
            Filtrer intersectBox(current, next) avec
            -- Si le prochain sommet est à l'extérieur, il y
            -- a forcément intersection avec le bord de B.
1           | Zero ⇒ impossible
            -- Une seule intersection, le cas standard.
2           | One outPoint ⇒ status ← Out outPoint ;
            -- Le point courant est forcément sur le bord.
            -- On peut ignorer la première intersection.
3           | Two _ outPoint ⇒ status ← Out outPoint ;
        Fin
    | Out outPoint ⇒
        Filtrer (intersectBox(current, next), insideBox(next)) avec
        -- Si le prochain sommet est à l'intérieur, il y
        -- a forcément intersection avec le bord de B.
4       | (Zero, True) ⇒ impossible
        -- On reste à l'extérieur, il n'y a rien à faire.
5       | (Zero, False) ⇒ continue
        -- On passe par un sommet de B.
6       | (One inPoint, False) ⇒
            between ← boxBetween(outPoint, inPoint) ;
            submits ← (between\inPoint) • submits ;
            status ← Out inPoint ;
        -- Le cas standard. Il y a deux intersections si
        -- next est sur le bord de B.
7       | (One inPoint, True) | (Two inPoint _, True) ⇒
            submits ← boxBetween(outPoint, inPoint) • submits ;
            status ← Inside ;
        -- On coupe un sommet de B.
8       | (Two inPoint outPoint', False) ⇒
            submits ← boxBetween(outPoint, inPoint) • submits ;
            status ← Out outPoint' ;
    current ← next ;
Fin
Retourne submits ;
```

l'initialisation, `submits` est vide, le point courant est le point de départ (x_0^+, y_0^+) et son statut est, par hypothèse, **Inside**. À chaque itération, l'algorithme calcule le prochain sommet du tour de $\hat{S}$, représenté par la variable `next`, et met à jour son état interne en fonction du statut du sommet courant et de la position du sommet suivant vis-à-vis de B. Chaque cas à traiter correspond à une ligne de code numérotée. Étudions les donc une par une :

1. Le sommet courant appartient à B, le sommet suivant est à l'extérieur, mais nous n'avons pas trouvé d'intersection entre le segment liant `current` et `next` et le bord de B. Ce cas est purement impossible.
2. Le sommet courant appartient à B, le suivant est à l'extérieur et nous avons trouvé une intersection. C'est le cas le plus simple à conceptualiser. La mise à jour consiste simplement à changer le statut pour refléter que nous sommes maintenant en dehors de B et que le point de sortie est l'intersection que nous venons de calculer.
3. Le sommet courant appartient à B, le suivant est à l'extérieur et nous avons trouvé deux intersections. La seule configuration amenant à ce cas survient lorsque le sommet courant est sur le bord de B, et donc confondu avec la première des deux intersections. Nous pouvons donc l'ignorer et nous ramener au cas précédent.
4. Le somment courant est en dehors de B, le suivant est à l'intérieur et nous n'avons pas trouvé d'intersection. Ce cas est également impossible.
5. Le sommet courant est en dehors de B, le suivant est à l'extérieur et nous n'avons pas trouvé d'intersection. Il n'y a rien à faire, sinon à continuer le parcours du bord de $\hat{S}$.
6. Le sommet courant est en dehors de B, le suivant l'est également mais nous avons trouvé une unique intersection, dénotée `inPoint`. La seule configuration possible est que `inPoint` soit un sommet de B. Nous ajoutons donc aux sommets valides tous les sommets de B entre le point de sortie et `inPoint` et continuons le parcours du bord de $\hat{S}$ en considérant `inPoint` comme point de sortie. Afin d'éviter que `inPoint` soit ajouté deux fois à la liste des sommets validés, il convient ici de faire attention de le supprimer de la liste construite par `boxBetween`.
7. Le sommet courant est en dehors de B, le suivant est à l'intérieur et nous avons une ou deux intersections. Ces deux cas sont identiques car la seule configuration permettant d'avoir deux intersections survient lorsque le sommet suivant est sur le bord de B, et est donc confondu avec la seconde intersection. Ici, nous n'avons qu'à ajouter les sommets pertinents et mettre à jour le statut pour spécifier que nous sommes désormais à l'intérieur.

8. Le sommet courant est en dehors de B, le suivant l'est également et nous avons trouvé deux intersections. Le segment liant `current` à `next` coupe B prêt d'un sommet. La première intersection est donc un point d'entrée et la seconde un nouveau point de sortie. Nous ajoutons les sommets pertinents et continuons en mettant à jour le statut pour prendre en considération le nouveau point de sortie.

Nous n'avons pas encore de preuve de correction pour l'algorithme. Elle ne devrait cependant pas représenter une difficulté majeure et cela fait parti de nos objectifs futurs que de construire cette preuve.

La complexité de l'algorithme est en $O(n)$ car la boucle itère sur les $2n$ vecteurs formant le bord de $\hat{S}$, et chaque itération est en $O(1)$. En effet, les trois fonctions se rapportant au bord de B, c'est-à-dire `intersectBox`, `insideBox` et `boxBetween` ont une complexité en $O(1)$ car le bord de B est nécessairement composé d'exactement quatre faces. En prenant en considération le tri des vecteurs formant le bord de $\hat{S}$, nous avons ainsi une complexité totale en $O(n \log n)$.

Une comparaison de la précision et des performances de nos travaux par rapport aux méthodes standard est présentée au chapitre 9. Nous y montrons que nos travaux permettent de profiter des informations additionnelles propagées par le produit réduit entre les intervalles et les formes affines pour améliorer la précision des opérations non linéaires sans fortement impacter les performances. Cette amélioration de la précision permet de construire des analyses plus précises, mais n'est malheureusement pas suffisante pour l'analyse de l'erreur relative. En effet, la sémantique de cette dernière utilise énormément d'opérations non linéaires, et bien que nous ayons réduit l'incertitude introduite par chacune d'entre elles, l'accumulation de ces incertitudes reste un problème lors de l'analyse de comportements aussi fortement non linéaires. Nous avons donc travaillé à développer une nouvelle abstraction de valeur, spécialisée dans l'analyse de l'erreur relative.

Chapitre 7

Erreurs relatives relationnelles

Améliorer les opérateurs abstraits des formes affines pour les opérations non linéaires permet de gagner en précision, mais n'est pas suffisant pour l'erreur relative. En effet, comme nous le présentons au chapitre 9, la précision d'une analyse utilisant des formes affines pour borner la sémantique exacte et les deux sémantiques d'erreur n'est pas meilleure que celle d'une analyse utilisant des formes affines pour la sémantique exacte et celle d'erreur absolue, mais des intervalles pour celle d'erreur relative. Ces résultats ne sont finalement pas si étonnants, puisque la sémantique de l'erreur relative utilise beaucoup d'opérations non linéaires pour lesquelles les formes affines introduisent de la sur-approximation. En particulier, chaque nouvelle erreur élémentaire est introduite de manière multiplicative. Cela signifie qu'à chaque opération, l'analyse introduit de la sur-approximation, amenant invariablement à des bornes imprécises.

Exemple 7.0.1. L'application des définitions 2.20 et 2.18 à $e = (x - y) \times z$ donne l'expression symbolique suivante :

$$\begin{aligned}\mathcal{E}_r^\phi(e) &= \left(1 + \mathcal{E}_r^\phi(x - y)\right)\left(1 + \mathcal{E}_r^\phi(z)\right)\left(1 + \Gamma_r^\phi(\widetilde{(x - y)}.\widetilde{z})\right) - 1 \\ &= \frac{x\left(1 + \mathcal{E}_r^\phi(x)\right) - y\left(1 + \mathcal{E}_r^\phi(y)\right)}{x - y}\left(1 + \Gamma_r^\phi(\widetilde{x} - \widetilde{y})\right)\left(1 + \mathcal{E}_r^\phi(z)\right) \times \\ &\qquad \left(1 + \Gamma_r^\phi(\widetilde{(x - y)}.\widetilde{z})\right) - 1\end{aligned}$$

Nous remarquons ici clairement que chaque terme d'erreur est introduit au travers d'opérations non linéaires complexes à traiter précisément avec des intervalles ou des formes affines.

Nous avons donc travaillé à développer un nouveau domaine abstrait spécifiquement adapté à représenter et à propager l'erreur relative en introduisant le

moins de sur-approximation possible. Cette nouvelle abstraction est en partie fondée sur le fonctionnement des formes affines, mais est adaptée pour représenter le caractère non linéaire de la sémantique de l'erreur relative. Ces travaux ont été menés par Kévin Youyou, un étudiant du Master FIIL de l'Université Paris-Saclay que j'ai eu le plaisir d'encadrer pendant son stage de fin d'étude [90].

La présentation de ces travaux est divisée en deux parties. Dans la première partie, nous présentons l'abstraction ainsi que son ordre partiel et sa fonction de concrétisation. Dans la seconde partie, nous présentons les opérateurs arithmétiques dédiés à la sémantique de l'erreur relative. Malheureusement, nous n'avons pas eu le temps de définir des opérateurs abstraits pour l'union, l'intersection, ni d'opérateur d'élargissement. Cela signifie que notre abstraction ne satisfait pas complètement les prérequis d'un domaine abstrait. Cela fait parti de nos objectifs futurs que de définir ces derniers opérateurs.

7.1 Abstraction et ordre partiel

Comme nous l'avons vu dans l'exemple 7.0.1, la propagation de l'erreur relative donne naissance à des expressions symboliques ne correspondant pas du tout à des formes affines. Cependant, l'exemple illustre également que ces expressions semblent suivre un schéma particulier, apparaissant clairement lors de l'introduction des erreurs élémentaires. Ce schéma consiste en une série de produits entre termes de la forme $1 + x$. Nous remarquons également qu'une soustraction par 1 apparait systématiquement. C'est donc sur ce schéma que nous nous sommes fondés pour construire notre abstraction relationnelle pour l'erreur relative.

Définition 7.1.1. Soit $\mu_1, \ldots, \mu_n$ et $\nu_1, \ldots, \nu_m$ deux jeux de symboles de bruit, dont les valeurs sont comprises entre -1 et 1, et $\lambda_1, \ldots, \lambda_n, \psi_1, \ldots, \psi_m$ des réels tels que $|\lambda_i| < 1$ pour tout $i \in [1\,;n]$ et $|\psi_j| < 1$ pour tout $j \in [1\,;m]$. Nous appelons *forme relative* toute expression symbolique $\overset{\star}{x}$ de la forme :

$$\overset{\star}{x} \triangleq c^x \times \prod_{i=1}^{n} (1 + \lambda_i \mu_i)^{\kappa_i^x} \times \prod_{j=1}^{m} (1 + \psi_j \nu_j)^{\tau_j^x} - 1 \tag{7.1}$$

avec $c^x \in \mathbb{R}^{+*}$ le *centre* de la forme relative, $\kappa_i^x \in \mathbb{R}$ ses *exposants centraux*, et τ_i^x ses *exposants de perturbations*. L'ensemble des formes relatives est dénoté $\mathbb{RF}$. La notation $\overset{\star}{x}(\mu, \nu)$ correspond à l'évaluation de la forme relative $\overset{\star}{x}$ en utilisant les symboles centraux μ_i et les symboles de perturbation ν_j.

L'expression symbolique 7.1, en plus de représenter le schéma que nous avons identifié pour l'erreur relative, mime une partie des caractéristiques des formes affines. En effet, les deux sont construites autour de variables symboliques comprises

entre -1 et 1, les deux présentent un centre et des coefficients propres permettant de représenter l'influence de chaque symbole dans le résultat de l'expression et les deux traitent séparément les incertitudes $\lambda_i\mu_i$ introduites par le programme des incertitudes $\psi_j\nu_j$ produites par l'analyse. Exactement comme pour les formes affines, nous ne ferons la distinction entre ces deux types de symboles que lorsque cela sera nécessaire. Dans le cas contraire, nous utiliserons les notations λ_i, μ_i et κ_i^x pour désigner indifféremment les coefficients, symboles et exposants centraux et de perturbations.

Cependant, il y a également deux différences importantes. La première, bien entendu, est que nous utilisons un produit plutôt qu'une somme afin de représenter le comportement de l'erreur relative. La seconde est que, bien que chaque symbole est associé à un coefficient d'une manière analogue à ceux des formes affines, ils sont ici indépendants de l'expression ; leurs valeurs sont donc constantes. Cela est finalement assez naturel. En effet, chaque symbole central représente une erreur élémentaire, dont l'amplitude est calculée au moment de son introduction et est donc constante. Chaque erreur élémentaire impacte l'erreur relative d'un calcul majoritairement au travers de produits, ce qui est représenté par les exposants. Les λ_i restent donc constants au cours de l'analyse. Les coefficients des symboles de perturbation présentent le même comportement : ils sont calculés une fois au moment de l'introduction de l'incertitude et sont ensuite propagés au travers de produits. Notons également que les bornes sur les coefficients garantissent qu'il n'y a aucun zéro dans le produit, ce qui simplifie les définitions et raisonnements qui suivent. Ces bornes ne sont par ailleurs pas très contraignantes. En effet, si à un moment de l'analyse, il nous faut introduire une erreur relative d'une amplitude plus grande que 1, cela revient à dire que la valeur du calcul associé peut varier dans des proportions tellement gigantesques qu'il est raisonnable de conclure que nous ne pouvons pas borner l'erreur relative.

7.1.1 Fonction de concrétisation

La fonction de concrétisation des formes relatives fonctionne d'une manière analogue à celle des formes affines :

$$\gamma_{\mathbb{RF}}(\overset{\star}{x}) = \left\{ c^x \times \prod_{i=1}^{n+m} \left(1 + \lambda_i\mu_i\right)^{\kappa_i^x} - 1 \mid \mu \in [-1\,;\,1]^{n+m} \right\} \tag{7.2}$$

Elle consiste à évaluer la forme relative pour toutes les valuations possibles des symboles de bruit centraux et de perturbations.

Nous définissons également les deux fonctions de conversions $\Phi_{\mathbb{I}}^{\mathbb{RF}} : \mathbb{RF} \mapsto \mathbb{I}$ et $\Phi_{\mathbb{RF}}^{\mathbb{I}} : \mathbb{I} \mapsto \mathbb{RF}$ avec les intervalles. La première est simple à définir, et calcule la

borne supérieure et la borne inférieure de la forme relative :

$$\Phi_{\mathbb{I}}^{\mathbb{RF}}(\overset{\star}{x}) \triangleq c^x \times \left[\prod_{i=1}^{n+m} (1 - |\lambda_i|)^{\kappa_i^x} \; ; \; \prod_{i=1}^{n+m} (1 + |\lambda_i|)^{\kappa_i^x} \right] - 1 \tag{7.3}$$

Intéressons nous maintenant à définir la fonction permettant de convertir tout intervalle $[\underline{x} \; ; \overline{x}]$ satisfaisant l'hypothèse $-1 < \underline{x} < \overline{x} < 1$ vers une forme relative. L'ajout de cette hypothèse est nécessaire pour pouvoir construire une forme relative satisfaisant la définition 7.1. Elle est de plus cohérente. En effet, un intervalle plus large représenterait une erreur relative si grande qu'elle n'aurait plus vraiment de sens.

Nous cherchons donc à calculer les constantes c^x et λ_{n+1} telles que nous ayons à la fois $c^x \times (1 + \lambda_{n+1}) - 1 = \overline{x}$ et $c^x \times (1 - \lambda_{n+1}) - 1 = \underline{x}$. Intuitivement, cela signifie que nous cherchons deux constantes telles que la projection en intervalle de la forme relative $c^x \times (1 + \lambda_{n+1}\mu_{n+1}) - 1$ soit $[\underline{x} \; ; \overline{x}]$. En additionnant les deux conditions, nous obtenons :

$$\begin{gathered} c^x \times (1 + \lambda_{n+1}) - 1 + c^x \times (1 - \lambda_{n+1}) - 1 = \overline{x} + \underline{x} \\ c^x \times (2 + \lambda_{n+1} - \lambda_{n+1}) - 2 = \overline{x} + \underline{x} \\ c^x = \frac{\overline{x} + \underline{x}}{2} + 1 \end{gathered}$$

Et en soustrayant les deux conditions, nous obtenons :

$$\begin{gathered} c^x \times (1 + \lambda_{n+1}) - 1 - c^x \times (1 - \lambda_{n+1}) + 1 = \overline{x} - \underline{x} \\ 2.c^x.\lambda_{n+1} = \overline{x} - \underline{x} \\ \lambda_{n+1} = \frac{\overline{x} - \underline{x}}{2c^x} = \frac{\overline{x} - \underline{x}}{\overline{x} + \underline{x} + 2} \end{gathered}$$

Vérifions maintenant que ces deux constantes satisfont bien les contraintes de la définition 7.1, c'est-à-dire $c^x \geq 0$ et $|\lambda_{n+1}| < 1$:

$$-1 < \underline{x} \leq \overline{x} < 1 \implies \overline{x} + \underline{x} > -2 \iff \frac{\overline{x} + \underline{x}}{2} > -1 \iff \frac{\overline{x} + \underline{x}}{2} + 1 > 0$$

$$\frac{\overline{x} - \underline{x}}{\overline{x} + \underline{x} + 2} < 1 \iff \overline{x} - \underline{x} < \overline{x} + \underline{x} + 2 \iff -2\underline{x} < 2 \iff \underline{x} > -1$$

Les deux constantes c^x et λ_{n+1} satisfont bien les contraintes de la définition 7.1 et permettent de construire une forme relative dont la projection en intervalle est $[\underline{x} \; ; \overline{x}]$. Ainsi, la définition de la fonction de conversion des intervalles vers les formes relatives est donc la suivante :

$$\Phi_{\mathbb{RF}}^{\mathbb{I}}([\underline{x} \; ; \overline{x}]) \triangleq \left(\frac{\overline{x} + \underline{x}}{2} + 1 \right) \times \left(1 + \frac{\overline{x} - \underline{x}}{\overline{x} + \underline{x} + 2} \mu_{n+1} \right) - 1 \tag{7.4}$$

7.1.2 Ordre partiel

Définissons ensuite un ordre partiel sur les formes relatives. Nous profitons ici des grandes similarités entre formes affines et formes relatives pour adapter l'ordre partiel des premières aux secondes. Commençons par étendre $\mathbb{RF}$ avec deux nouvelles valeurs $\bot_{\mathbb{RF}}$ et $\top_{\mathbb{RF}}$, représentant respectivement l'ensemble vide et tout $\mathbb{R}$. Nous dénotons l'ensemble étendu par $\mathbb{RF}^+$.

Une intuition importante à prendre en considération est que nous souhaitons refléter dans notre ordre partiel que, comme pour les formes affines, nous accordons plus d'importance aux symboles centraux qu'à ceux de perturbations. Cette différence de traitement découle du fait que les symboles centraux sont associés aux entrées et aux erreurs d'arrondis du programme analysé alors que les symboles de perturbations ne représentent que les incertitudes liées à l'analyse. La manière dont l'ordre partiel défini ci-après reflète cela peut être compris de la manière suivante : une forme relative $\overset{\star}{x}$ est plus petite qu'une forme relative $\overset{\star}{y}$ si et seulement si les écarts de comportement de leurs parties centrales sont systématiquement compensables en utilisant les symboles de perturbations.

Théorème 7.1.1. La relation $\sqsubseteq_{\mathbb{RF}^+}$ définie comme

$$\begin{aligned} &\forall \overset{\star}{x}, \overset{\star}{y} \in \mathbb{RF}, \overset{\star}{x} \sqsubseteq_{\mathbb{RF}^+} \overset{\star}{y} \iff \\ &\quad \forall \mu \in [-1\,;\,1]^n, \forall \nu^x \in [-1\,;\,1]^n, \exists \nu^y \in [-1\,;\,1]^n, \overset{\star}{x}(\mu, \nu^x) = \overset{\star}{y}(\mu, \nu^y) \end{aligned} \tag{7.5}$$

$$\forall \overset{\star}{x} \in \mathbb{RF}, \bot_{\mathbb{RF}} \sqsubseteq_{\mathbb{RF}^+} \overset{\star}{x} \wedge \overset{\star}{x} \sqsubseteq_{\mathbb{RF}^+} \top_{\mathbb{RF}} \tag{7.6}$$

forme un préordre sur l'ensemble $\mathbb{RF}^+$. De ce préordre découle la relation d'équivalence $\overset{\star}{x} \sim \overset{\star}{y} \iff \overset{\star}{x} \sqsubseteq_{\mathbb{RF}^+} \overset{\star}{y} \wedge \overset{\star}{y} \sqsubseteq_{\mathbb{RF}^+} \overset{\star}{x}$. La relation $\sqsubseteq_{\mathbb{RF}^+} / \sim$, c'est-à-dire la relation qui accepte les mêmes paires que notre préordre et considère les éléments en relation par $\sim$ comme dans la même classe d'équivalence, forme un ordre partiel pour $\mathbb{RF}^+$. Nous le noterons, par abus de langage, également $\sqsubseteq_{\mathbb{RF}^+}$.

Démonstration. Nous ne nous intéressons ici qu'à prouver les propriétés de l'ordre partiel pour des formes relatives. Les cas présentant les valeurs $\bot_{\mathbb{RF}}$ et $\top_{\mathbb{RF}}$ se démontrent en effet trivialement à partir de la définition 7.6.

La réflexivité se démontrant trivialement, et l'antisymétrie étant satisfaite par construction, il nous reste à démontrer la transitivité. Soit $\overset{\star}{x}$, $\overset{\star}{y}$ et $\overset{\star}{z}$ trois formes relatives telles que $\overset{\star}{x} \sqsubseteq_{\mathbb{RF}^+} \overset{\star}{y}$ et $\overset{\star}{y} \sqsubseteq_{\mathbb{RF}^+} \overset{\star}{z}$. Nous ne considérons ici que le cas où $\overset{\star}{x}$, $\overset{\star}{y}$ et $\overset{\star}{z}$ sont toutes les trois différentes de $\top_{\mathbb{RF}}$ et $\bot_{\mathbb{RF}}$ car le traitement de ces différents cas est trivial. Nous avons donc les deux relations suivantes :

$$\forall \mu \in [-1\,;\,1]^n, \forall \nu^x \in [-1\,;\,1]^n, \exists \nu^y \in [-1\,;\,1]^n, \overset{\star}{x}(\mu, \nu^x) = \overset{\star}{y}(\mu, \nu^y)$$

$$\forall \mu \in [-1\,;\,1]^n, \forall \nu^y \in [-1\,;\,1]^n, \exists \nu^z \in [-1\,;\,1]^n, \overset{\star}{y}(\mu, \nu^y) = \overset{\star}{z}(\mu, \nu^z)$$

Pour tout μ et tout ν^x, il existe un ν^y tel que $\overset{\star}{x}(\mu, \nu^x) = \overset{\star}{y}(\mu, \nu^y)$. Or, pour tout μ et tout ν^y, il existe un ν^z tel que $\overset{\star}{y}(\mu, \nu^y) = \overset{\star}{z}(\mu, \nu^z)$. Cela signifie donc que pour tout μ et ν^x, il existe un ν^z tel que $\overset{\star}{x}(\mu, \nu^x) = \overset{\star}{z}(\mu, \nu^z)$, ce qui est équivalent à $\overset{\star}{x} \sqsubseteq_{\mathbb{RF}^+} \overset{\star}{y}$. □

Cet ordre partiel, fortement inspiré de celui des formes affines, permet de prendre en compte les symboles centraux partagés entre les deux opérandes et utilise les symboles de perturbation comme marge de manoeuvre pour compenser de potentiels écarts sur la partie centrale. Intéressons nous maintenant à la monotonie de la fonction de concrétisation des formes relatives.

Théorème 7.1.2. La fonction de concrétisation $\gamma_{\mathbb{RF}^+}$, c'est-à-dire l'extension de $\gamma_{\mathbb{RF}}$ associant l'ensemble vide à $\bot_{\mathbb{RF}}$ et $\mathbb{R}$ tout entier à $\top_{\mathbb{RF}}$, est monotone, c'est-à-dire que si $\overset{\star}{x} \sqsubseteq_{\mathbb{RF}^+} \overset{\star}{y}$, alors $\gamma_{\mathbb{RF}^+}(\overset{\star}{x}) \subseteq \gamma_{\mathbb{RF}^+}(\overset{\star}{y})$.

Démonstration. Nous ne nous intéressons ici qu'à prouver la monotonie pour des formes relatives. Les cas présentant les valeurs $\bot_{\mathbb{RF}}$ et $\top_{\mathbb{RF}}$ se démontrent en effet trivialement. Il nous suffit de montrer que si $\overset{\star}{x} \sqsubseteq_{\mathbb{RF}^+} \overset{\star}{y}$, alors pour tout $v \in \gamma_{\mathbb{RF}^+}(\overset{\star}{x})$, nous avons $v \in \gamma_{\mathbb{RF}^+}(\overset{\star}{y})$. Si v appartient à la concrétisation de $\overset{\star}{x}$, cela signifie qu'il existe une valuation μ des symboles centraux, et une valuation ν^x des symboles de perturbation, tel que $v = \overset{\star}{x}(\mu, \nu^x)$. Or, la définition 7.5 garantit l'existence d'un ν^y tel que $\overset{\star}{x}(\mu, \nu^x) = \overset{\star}{y}(\mu, \nu^y)$. Cela signifie donc qu'il existe une valuation des symboles de bruit tel que $v = \overset{\star}{y}(\mu, \nu^y)$, et donc que v appartient à la concrétisation de $\overset{\star}{y}$. □

7.2 Opérateurs arithmétiques

Nous pouvons maintenant présenter les opérateurs arithmétiques abstraits des formes relatives. Cependant, avant de commencer cette présentation, il nous semble important d'insister sur le fait que les formes relatives ne servent qu'à approximer la sémantique d'erreur relative, et donc que la correction de leurs opérateurs abstraits doit être jugée vis-à-vis de cette sémantique, et non de la sémantique exacte. Nous pouvons désormais commencer notre présentation. Contrairement aux formes affines, ici, ce sont les opérations non linéaires qui sont triviales à définir. Nous allons donc commencer par elles.

Théorème 7.2.1. Soit x et y deux expressions et $\overset{\star}{x}$ et $\overset{\star}{y}$ deux formes relatives approximant respectivement l'erreur relative de x et celle de y correctement dans

un état abstrait $\rho^\sharp$. Les formes relatives suivantes :

$$\overset{\star}{x} \times \overset{\star}{y} \triangleq c^x c^y \times \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\kappa_i^x+\kappa_i^y} \times \left(\Phi^{\mathbb{I}}_{\mathbb{RF}}\left(\Gamma_r^{\phi\sharp}(\widetilde{x}.\widetilde{y})\right)+1\right)-1 \tag{7.7}$$

$$\overset{\star}{x} \div \overset{\star}{y} \triangleq \frac{c^x}{c^y} \times \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\kappa_i^x-\kappa_i^y} \times \left(\Phi^{\mathbb{I}}_{\mathbb{RF}}\left(\Gamma_r^{\phi\sharp}(\widetilde{x} \div \widetilde{y})\right)+1\right)-1 \tag{7.8}$$

$$\overset{\star}{\sqrt{x}} \triangleq \sqrt{c^x} \times \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\frac{\kappa_i^x}{2}} \times \left(\Phi^{\mathbb{I}}_{\mathbb{RF}}\left(\Gamma_r^{\phi\sharp}(\sqrt{\widetilde{x}})\right)+1\right)-1 \tag{7.9}$$

sont, pour tout $\rho \in \gamma(\rho^\sharp)$, des approximations correctes de respectivement $\mathcal{E}_r^\phi[\![x \times y]\!](\rho)$, $\mathcal{E}_r^\phi[\![x \div y]\!](\rho)$ et $\mathcal{E}_r^\phi[\![\mathbf{sqrt}(x)]\!](\rho)$.

Démonstration. Il suffit de remplacer les termes $\mathcal{E}_r^\phi(x)$ et $\mathcal{E}_r^\phi(y)$ dans les définitions 2.20, 2.22 et 2.26 par les formes relatives $\overset{\star}{x}$ et $\overset{\star}{y}$, et d'utiliser l'abstraction des erreurs élémentaires que nous avons présentée au chapitre 5, pour obtenir les définitions ci-dessus. Nous ne fournissons une preuve détaillée que pour la multiplication, les preuves des autres opérateurs étant similaires. Pour la multiplication, la définition 2.20 est :

$$\mathcal{E}_r^\phi[\![x \times y]\!](\rho) = \left(\mathcal{E}_r^\phi(x)+1\right)\left(\mathcal{E}_r^\phi(y)+1\right)\left(\Gamma_r^\phi(\widetilde{x}.\widetilde{y})+1\right)-1$$

En remplaçant chaque occurrence de $\mathcal{E}_r^\phi(x)$ par $\overset{\star}{x}$ et chaque occurrence de $\mathcal{E}_r^\phi(y)$ par $\overset{\star}{y}$, et en abstrayant l'erreur élémentaire grâce aux définitions du chapitre 5 et la fonction de conversion $\Phi^{\mathbb{I}}_{\mathbb{RF}}$, nous obtenons une abstraction correcte de l'équation précédente :

$$\left(c^x \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\kappa_i^x}\right)\left(c^y \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\kappa_i^y}\right)\left(\Phi^{\mathbb{I}}_{\mathbb{RF}}\left(\Gamma_r^{\phi\sharp}(\widetilde{x}.\widetilde{y})\right)+1\right)-1$$

Il suffit de regrouper les différents termes $(1+\lambda_i\mu_i)$ en additionnant leurs exposants pour obtenir la définition 7.7. □

Terminons avec l'addition et la soustraction. La sémantique de l'erreur relative pour ces deux opérations, que nous avons présentée à la définition 2.18, est, pour rappel, la suivante :

$$\mathcal{E}_r^\phi[\![x \pm y]\!](\rho) = \frac{x\left(\mathcal{E}_r^\phi(x)+1\right) \pm y\left(\mathcal{E}_r^\phi(y)+1\right)}{x \pm y} \times \left(\Gamma_r^\phi(\widetilde{x} \pm \widetilde{y})+1\right)-1$$

L'abstraction de cette définition par simple remplacement des termes $\mathcal{E}_r^\phi(x)$ et $\mathcal{E}_r^\phi(y)$, comme nous l'avons fait pour les opérations non linéaires, nous donne

l'équation suivante :

$$\frac{x.c^x \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\kappa_i^x} \pm y.c^y \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\kappa_i^y}}{x \pm y} \times \left(\Gamma_r^{\phi\sharp}(\widetilde{x} \pm \widetilde{y}) + 1\right) - 1 \quad (7.10)$$

Le premier terme de cette équation, dénotons le t, est problématique. En effet, dans le cas général, les additions (ou les soustractions) aux numérateurs et aux dénominateurs ne nous permettent pas de l'exprimer comme une forme relative sans introduire d'incertitude. Une première solution serait de simplement borner t, et d'ajouter un nouveau symbole de perturbation pour représenter cette incertitude. Cependant, ceci nous ferait perdre toute forme de relation entre les erreurs relatives des opérandes et celle du résultat, ce que nous voulons éviter. Conserver certaines des relations est heureusement assez aisé : il suffit en effet de factoriser tous les termes de la forme $(1+\lambda_i\mu_i)$ par le plus petit des exposants des opérandes :

$$t = \frac{x.c^x \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\kappa_i^x - \min(\kappa_i^x,\kappa_i^y)} \pm y.c^y \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\kappa_i^y - \min(\kappa_i^x,\kappa_i^y)}}{x \pm y} \times \prod_{i=1}^{n+m} (1+\lambda_i\mu_i)^{\min(\kappa_i^x,\kappa_i^y)} \quad (7.11)$$

Cette factorisation nous permet de conserver les relations. Cependant, pour pouvoir construire une forme relative approximant correctement, il nous faut pouvoir borner la première partie de t. Notons ici que nous aurions tout à fait pu factoriser en utilisant d'autres exposants, comme le maximum de ceux des opérandes par exemple, afin de conserver plus de relations. Cependant, ce choix impliquerait nécessairement une plus grande incertitude, et donc une analyse potentiellement moins précise.

Pour définir un opérateur abstrait pour l'addition et la soustraction, il nous faut donc pouvoir borner une expression de la forme $(x \times e_x \pm y \times e_y) \div (x \pm y)$ où e_x et e_y sont les termes restants des erreurs relatives de x et y. Un calcul naïf, utilisant l'arithmétique d'intervalles par exemple, nous donnerait des bornes correctes, mais très imprécises à cause de la perte de corrélation entre les différentes occurrences de x et y. Cependant, il est tout à fait possible de calculer des bornes optimales dans les intervalles pour cette expression, comme démontré dans la thèse de Guillaume Melquiond [58]. Nous rappelons ici le théorème permettant de calculer ces bornes optimales. Une preuve de ce dernier est disponible dans la thèse de Guillaume Melquiond [58].

Théorème 7.2.2. Soit $f : \mathbb{R}^4 \mapsto \mathbb{R}$ la fonction suivante :

$$f(x, y, a, b) = \frac{a.x \pm b.y}{x \pm y} \quad (7.12)$$

Soit $[\underline{x}\,;\overline{x}]$, $\left[\underline{y}\,;\overline{y}\right]$, $[\underline{a}\,;\overline{a}]$ et $\left[\underline{b}\,;\overline{b}\right]$ quatre intervalles tels que $[\underline{x}\,;\overline{x}] \pm \left[\underline{y}\,;\overline{y}\right]$ ne contienne pas zéro. Les extremums de f sur le sous-ensemble de $\mathbb{R}^4$ délimité par ces intervalles sont atteints aux sommets de ce sous-ensemble, c'est-à-dire pour tout quadruplet (x, y, a, b) tel que chaque composante soit égale à l'une des bornes de l'intervalle associé.

Il faut cependant faire ici attention sur un point. L'intervalle calculé par le théorème 7.2.2 est optimal dans le sens où il est la meilleure abstraction par intervalle. Cependant, il ne prend pas en compte de possibles relations entre x et y, ainsi qu'entre a et b. Nous rencontrons ici encore les limites du raisonnement par intervalles. La factorisation que nous avons effectuée à l'équation 7.11 garantit l'absence de relation entre les seconds, mais rien ne nous garantit la même propriété pour x et y.

Nous pouvons cependant faire mieux à partir du moment où nous disposons des abstractions par formes affines $\hat{x}$ et $\hat{y}$ pour les opérandes x et y. En effet, f est une fonction col pour tout sous-ensemble de $\mathbb{R}^4$ n'annulant pas son dénominateur, car toutes ses dérivées partielles sont de signes constants. En particulier, si nous fixons les paramètres a et b de f à des constantes arbitraires, alors f devient une fonction col à deux variables, et donc que ces extremums se trouvent sur le bord du sous-ensemble $\hat{S}$ formé par les deux formes affines $\hat{x}$ et $\hat{y}$. Cela signifie que nous pouvons utiliser l'algorithme présenté au chapitre 6 pour borner le résidu non factorisé de l'équation 7.11 en prenant en considération les relations entre les opérandes. Afin de calculer une borne correcte vis-à-vis des intervalles sur a et b, il conviendra ici de calculer les extremums de f pour les quatre affectations $(\underline{a}, \underline{b})$, $(\underline{a}, \overline{b})$, $(\overline{a}, \underline{b})$ et $(\overline{a}, \overline{b})$ et de choisir le maximum des maximums comme borne supérieure, ainsi que le mimimum des minimums comme borne inférieure.

Nous avons donc construit une nouvelle abstraction spécialisée dans la représentation et la propagation des erreurs relatives. Nous avons défini les opérateurs abstraits pour les différents opérateurs arithmétiques de notre langage de manière à correctement abstraire la sémantique de l'erreur relative. Ces opérateurs abstraits sont exacts pour toutes les opérations non linéaires, et nous avons mis en avant une technique pour gérer efficacement et précisément les incertitudes introduites par les opérations linéaires. Nous avons enfin posé les premières bases à la construction d'un domaine abstrait fondé sur cette abstraction en définissant un ordre partiel et une fonction de concrétisation. Il nous reste maintenant à montrer, à travers une évaluation expérimentale, l'intérêt de cette nouvelle abstraction par rapport à une simple utilisation des formes affines pour abstraire la sémantique de l'erreur relative.

Troisième partie

Expérimentations

Chapitre 8

Implémentation

Afin de pouvoir évaluer la précision et les performances des contributions présentées aux chapitres précédents, nous avons travaillé à leur implémentation au sein de l'analyseur par interprétation abstraite *Eva* [9] de la plateforme *Frama-C* [49]. Cette plateforme, écrite en *OCaml*, regroupe une collection d'outils (ou *greffons*) dédiés à l'analyse statique et dynamique de programmes écrits dans le langage C, et propose un langage de spécification commun nommé *ACSL* [5].

Le choix de travailler au sein de *Frama-C* et de son greffon *Eva* a été motivé par les possibilités de collaborations entre analyses qu'offre l'outil. En effet, *Frama-C* présente différents mécanismes permettant de profiter des résultats d'autres greffons, et *Eva* propose des mécanismes génériques permettant l'échange d'informations entre différents domaines abstraits. Cette dernière caractéristique est particulièrement intéressante car elle peut nous permettre de profiter des domaines abstraits déjà existant pour traiter les constructions du C non présentes dans notre langage d'étude, comme l'arithmétique entière ou les pointeurs. Cela nous a donc permis de nous concentrer sur le traitement de l'arithmétique flottante, tout en nous offrant la possibilité de développer une analyse pouvant traiter des programmes présentant d'autres constructions. C'est par exemple grâce à ces mécanismes de communication que notre implémentation est capable de traiter le transtypage de valeurs entières vers les flottants.

Les contributions des chapitres 5 et 6 ont amené à l'implémentation de deux domaines abstraits au sein d'*Eva* : le domaine des zonotopes et *Numerors*, un domaine dédié à l'analyse d'erreurs numériques et reposant sur notre produit réduit entre erreurs absolues et relatives. Les contributions du chapitre 7 ont, quant à elle, mené à l'implémentation d'une preuve de concept séparée, ce afin de réduire la part du stage de Kévin Youyou dédié au travail de programmation. Dans ce chapitre, nous nous concentrons donc sur l'implémentation des contributions des chapitres 5 et 6. Nous commençons la présentation de cette implémentation par une présentation générale de la plateforme *Frama-C*, puis de l'analyseur *Eva*. Ensuite, nous

présentons l'architecture générale de notre implémentation, et en particulier son caractère modulaire, permettant de reconstruire les domaines entiers à partir d'un simple module décrivant comment représenter les réels, les symboles de bruit, et les intervalles. Enfin, nous terminons en présentant plus en détail l'implémentation du domaine des zonotopes, et en particulier ce que nous avons mis en place pour contourner certaines rigidités du fonctionnement d'*Eva* rendant difficile une implémentation dans un paradigme fonctionnel du domaine, et tout particulièrement de sa gestion des conditions.

Notons qu'au moment où nous écrivons ces lignes, l'implémentation n'est pas totalement conforme à l'architecture que nous présentons ici. Ces différences découlent principalement de raisons historiques et d'un manque de temps. Nous comptons faire évoluer notre implémentation dans les prochains mois afin de nous conformer à cette architecture.

8.1 Présentation de *Frama-C*

Frama-C [49] est une plateforme dédiée à l'analyse de programmes écrits en C, regroupant différentes techniques d'analyse au sein d'une même infrastructure, collaborative et extensible. Par exemple, la plateforme propose *Eva* [9], un analyseur par interprétation abstraite, *WP* [4], un outil de vérification déductive fondé sur le calcul de plus faible précondition [28], ou encore *E-ACSL* [79], un outil de vérification à l'exécution.

Tous les analyseurs de la plateforme partagent un même langage de spécification appelé *ACSL* [5] : chaque analyseur peut à la fois prouver la validité (ou l'invalidité) de chaque annotation *ACSL* et générer de nouvelles annotations. Il s'agit de la première approche permettant la *collaboration* entre les différentes analyses. Par exemple, l'outil *RTE* [43] peut générer une annotation pour chaque potentielle erreur à l'exécution, que *Eva*, *WP* ou *E-ACSL* peuvent par la suite essayer de prouver, *Frama-C* se chargeant de combiner ce que chaque outil a pu prouver de manière à calculer ce qu'il reste à traiter[16]. Une autre approche permettant la collaboration est en utilisant les interfaces *OCaml* de chaque analyseur (les *APIs* en anglais). Par exemple, l'analyse permettant de construire le graphe d'appels traite les pointeurs de fonctions en utilisant les résultats obtenus par *Eva* lorsqu'ils sont disponibles.

Frama-C est également un outil extensible : il est possible de développer de nouvelles analyses et de les intégrer à la plateforme sous la forme de *greffons*. Cela est rendu possible grâce au fait que *Frama-C* est un outil *open source*, mais également grâce à son architecture, composée d'un *noyau* autour duquel gravitent les greffons, et spécifiquement conçu pour faciliter le développement et l'intégration de nouveaux greffons. Cette architecture, illustrée par la figure 8.1 provenant du

« manuel pour les développeurs » [78] de *Frama-C*, est composée de nombreuses briques logicielles regroupées en quatre catégories.

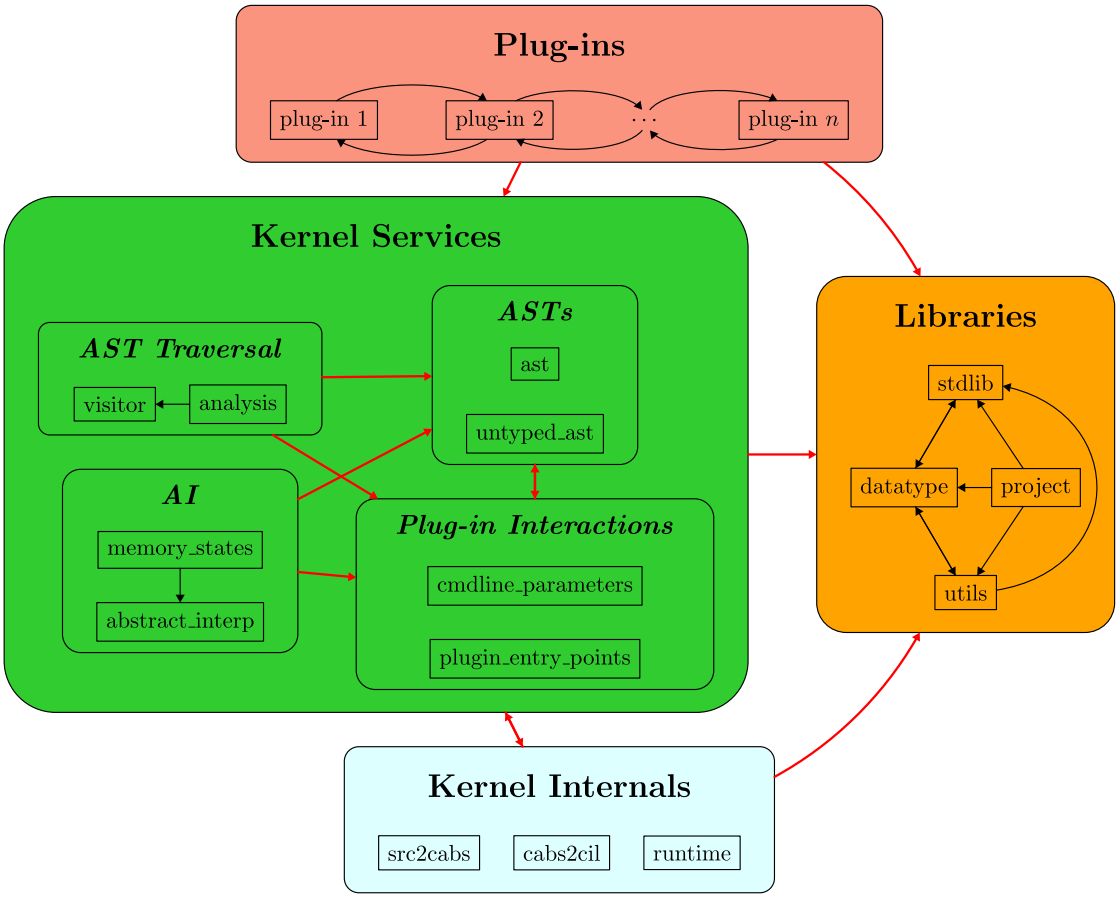


FIGURE 8.1 – Architecture de *Frama-C*.

La première catégorie, illustrée en bleu clair, correspond aux éléments internes du noyau ne concernant pas directement les personnes souhaitant développer de nouvelles analyses. On y trouve le code responsable de l'analyse syntaxique et du typage des programmes C analysés, ainsi que celui responsable du processus de démarrage de *Frama-C*. Il est important de noter que l'analyse syntaxique transforme les programmes analysées en une représentation intermédiaire, appelée *Cil* [49], présentant beaucoup de caractéristiques intéressantes permettant de simplifier la définitions d'analyseurs. En particulier, cette représentation intermédiaire garantie qu'aucune expression ne peut produire d'effet de bords. La deuxième catégorie, illustrée en vert, correspond aux services fournis par le noyau, c'est-à-dire les briques logicielles qui peuvent être utiles aux greffons tout en étant suffisamment générales pour être mutualisées. On y trouve les définitions des arbres de

syntaxes abstraites ainsi que des outils pour en faciliter le parcours et la modification, des analyseurs génériques prédéfinis, les outils permettant aux greffons d'agir sur l'interface homme-machine de *Frama-C* et l'implémentation de certains concepts centraux de l'interprétation abstraite. La troisième catégorie, illustrée en orange, correspond aux bibliothèques sur lesquelles repose *Frama-C*. On y trouve des extensions de la bibliothèque standard *OCaml*, des structures de données génériques et des bibliothèques externes. Enfin, la quatrième et dernière catégorie, illustrée en rouge, correspond aux greffons.

8.2 *Eva*, un analyseur modulaire

Eva [9] est un analyseur par interprétation abstraite modulaire, permettant de combiner plusieurs domaines abstraits afin de former des analyses adaptées aux programmes étudiés. Cette combinaison est plus riche qu'une simple analyse du programme avec différents domaines, car elle permet la mise en place de mécanismes de communications et de réductions inter-domaines. D'autres analyseurs proposent ce type de mécanisme, comme *Astrée* [8] ou *Mopsa* [61]. Là où ces derniers ne présentent pas de mécanisme générique et laissent aux développeurs le choix des méthodes d'échange pour chaque paire de domaines, *Eva* présente un choix de conception offrant une solution de communication générique : chaque domaine repose nécessairement sur une abstraction de valeur, cette dernière étant partageable entre plusieurs domaines. Ainsi, deux domaines partageant la même valeur peuvent échanger de l'information simplement en la faisant porter par les valeurs abstraites. Par exemple, si un domaine abstrait utilisant des intervalles comme abstraction de valeur infère qu'une variable x est positive, il lui suffira de réduire l'intervalle associé à x à sa partie positive pour communiquer l'information à tous les domaines utilisant également des intervalles comme abstraction de valeur. Cette information sera également partagée à tous les domaines utilisant des abstractions de valeur présentant un produit réduit avec les intervalles, permettant d'accroitre plus encore les échanges au sein d'*Eva*.

Le cœur de ce mécanisme de communication réside donc dans la séparation de chaque domaine abstrait en deux parties : une abstraction de valeurs et une abstraction d'états. Les abstractions de valeurs spécifient comment sur-approximer les valeurs possibles des variables, ainsi que les opérateurs abstraits permettant de sur-approximer la sémantique des expressions du C. Par exemple, la notion de valeur abstraite que nous avons présentée au chapitre 3 et formalisée à la définition 3.4.1 correspond à une abstraction de valeurs au sens d'*Eva* (il existe en pratique quelques différences, mais cela ne change pas grand chose dans le cadre de cette présentation). Les abstractions d'états, quant à elles, spécifient comment sur-approximer l'ensemble des états machine pouvant exister en un point de pro-

gramme donné, ainsi que les opérations nécessaires à sur-approximer la sémantique des instructions du C. C'est, par exemple, au niveau des abstractions d'états que sont définis les opérateurs d'élargissement ou l'interprétation des conditions.

Ces abstractions de valeurs et d'états sont combinées dynamiquement par *Eva* en fonction des requêtes de l'utilisateur afin de construire un unique domaine abstrait, qui est ensuite utilisé pour analyser le programme en utilisant une propagation classique. Nous ne présentons pas ici le fonctionnement de cette combinaison. Elle est en effet complexe et repose sur des notions avancées de programmation en *OCaml*. Un lecteur désireux de comprendre en détails le fonctionnement de cette combinaison peut lire la thèse de David Bühler [13]. Ce qu'il est important de comprendre, c'est que cette combinaison permet de mettre en place tous les moyens de communication entre les différentes abstractions et est complètement générique. Cette généricité repose, entre autre, sur le système de modules d'*OCaml* : chaque abstraction, que ce soit de valeurs ou d'états, doit satisfaire une interface définie, qu'il nous est impossible de modifier sans devoir changer en profondeur *Eva*. Cela nous a posé problème lors de l'implémentation du domaine des zonotopes à cause de sa gestion des conditions. Nous présentons cela plus en détails par la suite.

8.3 Architecture de l'implémentation

L'architecture de notre implémentation a été guidée par trois objectifs que nous pensions importants. Le premier est que nous voulions pouvoir, lors des phases de développement et de tests, nous éviter l'utilisation de l'arithmétique flottante dans la représentation des coefficients des formes affines et des bornes des intervalles, tout en minimisant le temps de développement nécessaire à son utilisation future en production. Le choix d'éviter initialement d'utiliser l'arithmétique flottante est motivée par le fait que garantir la correction des opérateurs abstraits, en particulier ceux des formes affines, est plus compliqué avec l'arithmétique flottante, car nous devrions alors prendre en considération les erreurs d'arrondi. Cependant, ce choix implique d'utiliser d'autres représentations pour les coefficients, comme des rationnels par exemple, qui sont beaucoup moins performants. Offrir un mécanisme permettant de reconstruire le plus simplement possible notre analyse en utilisant l'arithmétique flottante est donc nécessaire afin de pouvoir espérer utiliser notre implémentation sur des programmes significativement grands. Le deuxième objectif était de simplifier au maximum l'exploration de nouvelles idées, en particulier pour le choix de la combinaison de valeurs abstraites utilisée pour les erreurs et le choix de l'abstraction pour les symboles de bruit. Le troisième objectif était de pouvoir profiter des mécanismes de communication d'*Eva* afin de pouvoir déléguer la gestion de l'arithmétique entière et de la mémoire aux autres domaines de l'analyseur.

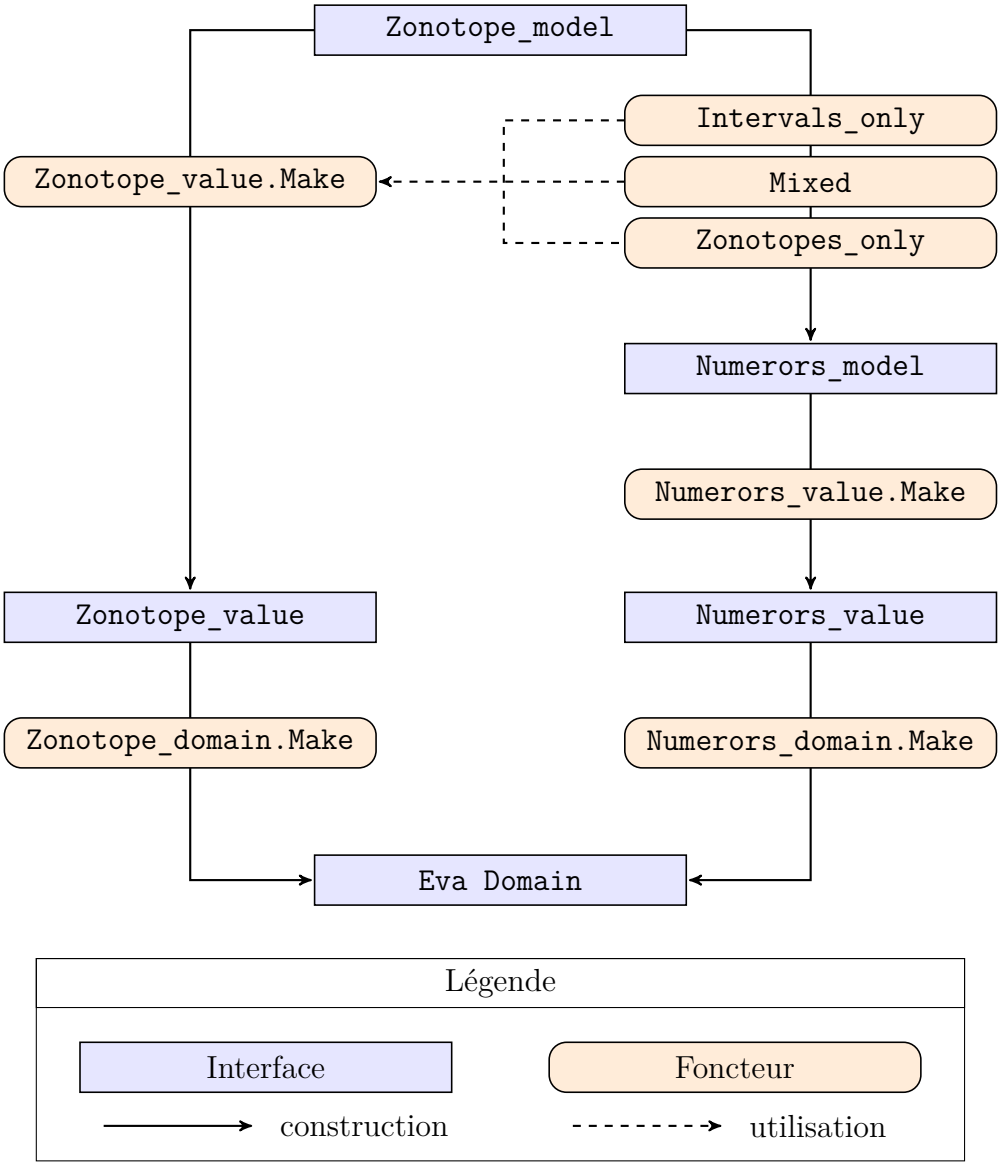


FIGURE 8.2 – Architecture de notre implémentation.

Afin d'atteindre ces objectifs, notre implémentation, qui représente environ 4500 lignes d'*OCaml* (sans compter les commentaires et les lignes vides), utilise le système de modules du langage *OCaml*, et en particulier les foncteurs. La figure 8.2 présente les différents foncteurs, illustrés par les rectangles oranges aux angles arrondis, et les différentes interfaces, illustrées par les rectangles bleus aux angles droits, de notre implémentation, ainsi que l'ordre de construction à respecter pour construire notre analyse. Présentons plus en détails ces différents éléments, en commençant par les interfaces :

— `Eva Domain` : interface utilisée par *Eva* pour les domaines abstraits. Nous y retrouvons, entre autres, tous les opérateurs imposés par la définition 3.3.1.
— `Numerors_value` : interface des valeurs abstraites du domaine *Numerors*. Elle correspond à une extension de l'interface utilisée par *Eva* pour les valeurs abstraites, offrant quelques fonctions en plus pour aider au déboguage par exemple.
— `Zonotope_value` : interface des valeurs abstraites du domaine des zonotopes. Elle correspond également à une extension de l'interface utilisée par *Eva* pour les valeurs abstraites.
— `Numerors_model` : interface spécifiant quelles abstractions utiliser pour les erreurs absolues et les erreurs relatives, ainsi que les opérations nécessaires pour définir le produit réduit.
— `Zonotope_model` : interface spécifiant comment représenter les nombres réels, les intervalles, les symboles de bruit des formes affines et l'abstraction utilisée pour ces symboles afin de traiter les conditions. C'est la seule structure qui ne peut pas être construite par un foncteur. Nous fournissons donc une implémentation par défaut. Cette dernière utilise des rationnels pour représenter les nombres réels, et propose des intervalles simples avec des bornes rationnelles également, une représentation des symboles comme de simples entiers, et une abstraction fondée sur le domaine des inéquations linéaires (ou domaine des polyèdres [20]), que nous simplifions vers des intervalles si nécessaire pour simplifier les opérations coûteuses. Pour améliorer les performances de notre implémentation en vue d'un passage en production, il nous faudra fournir une seconde implémentation de cette interface utilisant l'arithmétique flottante, et remplaçant l'abstraction des symboles par un simple domaine d'intervalles.

Présentons maintenant les différents foncteurs :

— `Intervals_only` : foncteur construisant une représentation des erreurs entièrement fondée sur les intervalles.
— `Mixed` : foncteur construisant une représentation des erreurs utilisant des formes affines pour l'erreur absolue et des intervalles pour l'erreur relative.
— `Zonotopes_only` : foncteur construisant une représentation des erreurs en-

tièrement fondée sur les formes affines.
— `Zonotope_value.Make` : foncteur construisant les valeurs abstraites du domaine des zonotopes à partir d'une représentation des coefficients, des intervalles et des symboles. C'est dans ce foncteur qu'est implémentée l'arithmétique affine, et donc nos contributions du chapitre 6.
— `Numerors_arith.Make` : foncteur construisant les valeurs abstraites du domaine *Numerors* à partir des abstractions à utiliser pour les erreurs. Nos contributions du chapitre 5 sont implémentées dans ce foncteur, mis à part notre contribution sur les erreurs élémentaires, qui est spécifique aux intervalles (bien que réutilisée pour les formes affines), et est donc présente dans l'implémentation fondée sur les rationnels de l'interface `Zonotope_model`.
— `Zonotope_domain.Make` et `Numerors_domain.Make` : foncteurs permettant de construire respectivement le domaine des zonotopes et le domaine *Numerors* pour *Eva*. Ils sont pour l'instant séparés, mais nous envisageons de les fusionner, leurs comportements étant proches car fondés sur le processus de construction d'un domaine abstrait à partir d'une abstraction de valeurs, que nous décrivions au chapitre 3.

Cette architecture répond à nos objectifs. En effet, il suffit de fournir une nouvelle implémentation de l'interface `Zonotope_model` pour pouvoir passer de l'arithmétique rationnelle à l'arithmétique flottante, permettant une mise en production rapide. Ceci nous permet également d'explorer de nouvelles abstractions pour les symboles de bruit. Ensuite, nous disposons déjà des combinaisons d'intervalles et de formes affines intéressantes pour les erreurs. Pour ajouter une nouvelle combinaison d'abstractions, comme par exemple en utilisant nos contributions du chapitre 7, il suffit donc de fournir un nouveau foncteur permettant de construire une représentation des erreurs utilisant les formes relatives. Enfin, cette architecture nous offre la possibilité de mettre en place des communications avec les autres domaines d'*Eva*, car nous respectons la séparation entre domaines et valeurs. Il nous suffit ainsi d'implémenter des mécanismes de réduction avec, par exemple, les intervalles d'*Eva* pour profiter des informations inférées par le reste de l'analyse. Nous avons mis en pratique cette réduction lors des conversions depuis les entiers vers les flottants, et nous comptons également nous en servir pour la gestion de la mémoire dans de prochains efforts de développement.

8.4 Implémentation des conditions dans le domaine des zonotopes

Comme nous l'avions rapidement présenté au chapitre 4, la solution pour interpréter les conditions dans le domaine des zonotopes est de réduire non pas

les formes affines, mais les intervalles des symboles de bruit qui les composent. Cette idée peut être généralisée en considérant que les symboles de bruit n'évoluent plus dans l'ensemble $[-1\,;\,1]^n$ mais dans un ensemble convexe P quelconque, abstrait au moyen d'un domaine abstrait $\Psi^\sharp$ donné, et sur lequel nous reportons l'interprétation des conditions. Cette modification du domaine a des répercussions sur l'ensemble de ses opérateurs, et donc, en particulier, sur les opérateurs des formes affines le composant. Formellement, cela peut être traduit comme suit : nous souhaitons ajouter à chaque opérateur abstrait op : $\mathbb{AF}^{+^n} \mapsto \mathbb{AF}^+$ un nouveau paramètre que nous nommons *contexte*, correspondant à une abstraction des symboles de bruit et prenant donc valeur dans $\Psi^\sharp$. Dit autrement, nous voudrions changer le type de chaque opérateur abstrait pour le type $\Psi^\sharp \mapsto \mathbb{AF}^{+^n} \mapsto \mathbb{AF}^+$.

Cette modification fait que les formes affines ne respectent plus la définition 3.4.1 d'abstraction de valeurs. Ce n'est pas tellement un problème d'un point de vue formel, la notion d'abstraction de valeur n'étant pas la seule approche pour construire le domaine. Cependant, cela représente bien un problème pour notre implémentation. En effet, afin de profiter des mécanismes de communication d'*Eva*, il est nécessaire que notre implémentation respecte le schéma abstraction de valeurs/d'états que nous avons présenté précédemment. Cela implique donc que nous devons fournir une implémentation de nos valeurs abstraites respectant l'interface d'*Eva* correspondante. Or, comme il nous est impossible de modifier cette interface, et qu'elle requiert les mêmes opérateurs que la définition 3.4.1 (en réalité l'interface est légèrement différente de la définition 3.4.1, mais cela ne change pas grand chose dans le cadre de cette présentation), il nous faut trouver une autre solution pour que les opérateurs abstrait aient accès au contexte courant.

Une première solution serait de considérer que chaque opérateur de notre implémentation des formes affines a accès à un pointeur global permettant d'accéder au contexte courant. Cependant, cette solution revient à tricher, en faisant croire que nos opérateurs respectent la définition 3.4.1 alors qu'ils présentent en pratique un paramètre caché correspondant au pointeur global. Cette solution n'est donc pas utilisable car, outre le fait qu'elle repose sur des mécanismes mutables réduisant la confiance que nous pourrions accorder à notre implémentation, elle ne permet pas de satisfaire les contraintes imposées par *Eva*, ce qui remettrait en question la correction de toute analyse qui reposerait sur une implémentation utilisant cette solution.

Une deuxième solution serait de définir nos valeurs abstraites comme des paires appartenant à $\Psi^\sharp \times \mathbb{AF}^+$, et de réécrire tous les opérateurs abstraits pour prendre cette définition en considération. Contrairement à la première solution, nous n'utilisons ici pas de mécanisme mutable, et nous pouvons respecter sans difficulté la définition 3.4.1. Cependant, cette solution présente un autre problème : rien ne garantit que les opérandes d'un même opérateur évoluent dans le même contexte. Si

cette caractéristique peut faire sens pour les opérateurs liés à la structure d'ordre partiel des formes affines, elle est beaucoup plus difficile à accepter pour les opérateurs arithmétiques. En effet, le contexte représente les informations apprises par l'interprétation des conditions du programme. Ces informations ne peuvent pas être modifiées lors de l'interprétation des expressions, ces dernières étant pures, et nous voudrions donc traduire cette garantie dans le fonctionnement des opérateurs arithmétiques de nos formes affines.

La solution que nous avons donc utilisée est de définir nos valeurs abstraites comme appartenant à l'ensemble $\Psi^\sharp \mapsto \mathbb{AF}^+$, c'est-à-dire l'ensemble des fonctions produisant des formes affines à partir de contextes. Cette abstraction permet, tout comme la précédente, d'éviter l'utilisation de mécanisme mutable, mais permet en plus de garantir que l'interprétation des opérandes est faite en utilisant un unique contexte. Il suffit pour cela que la fonction retournée par l'application d'un opérateur commence toujours par évaluer ces opérandes sur le contexte qui lui est fourni.

Exemple 8.4.1. L'addition abstraite pour les fonctions des contextes vers les formes affines est implémentée comme suit :

$$(+) : (\Psi^\sharp \mapsto \mathbb{AF}^+) \mapsto (\Psi^\sharp \mapsto \mathbb{AF}^+) \mapsto (\Psi^\sharp \mapsto \mathbb{AF}^+)$$
$$x + y \triangleq \lambda\psi.\, \hat{x} \leftarrow x(\psi)\ ;\ \hat{y} \leftarrow x(\psi)\ ;\ \hat{x} +_{\mathbb{AF}^+} \hat{y}$$

Notons que le calcul de l'addition de deux formes affines n'est pas affecté par la présence du contexte, permettant ici d'utiliser l'opérateur standard. Ce n'est pas le cas pour tous les opérateurs, comme celui pour l'union par exemple.

Cependant, satisfaire la définition 3.4.1 avec ces abstractions de valeur rend difficile la définition de la fonction de concrétisation et le calcul de la relation d'ordre partiel. En effet, la fonction de concrétisation que nous voudrions associer à ces valeurs abstraites aurait la signature suivante :

$$(\Psi^\sharp \mapsto \mathbb{AF}^+) \mapsto (\Psi^\sharp \mapsto \mathcal{P}(\mathbb{R})) \tag{8.1}$$

ce qui ne satisfait pas la définition 3.4.1. D'un point de vue formel, ceci ne représente en aucun cas une difficulté, il suffit de modifier légèrement le cadre que nous avons défini au chapitre 3 pour définir notre domaine abstrait. D'un point de vue implémentation, *Eva* exige de disposer d'une fonction d'affichage pour les valeurs abstraites, et donc dans notre cas, d'une manière de concrétiser nos fonctions en quelque chose d'affichable. Notre implémentation contourne le problème en fournissant une fonction d'affichage évaluant systématiquement la valeur abstraite dans le contexte où aucun symbole de bruit n'a été réduit par une condition. Ceci ne pose en pratique aucun problème, car nous pouvons toujours demander au domaine

d'afficher les résultats de l'analyse. Ce dernier, disposant forcément du contexte courant, peut évaluer les fonctions associées à chaque variable correctement, et donc présenter des résultats précis.

L'ordre partiel, quant à lui, représente un problème uniquement au niveau de l'implémentation. En effet, sa définition formelle est la suivante :

$$x \sqsubseteq y \iff \forall \psi \in \Psi^\sharp, x(\psi) \sqsubseteq_{\mathbb{AF}^+} y(\psi) \tag{8.2}$$

et introduit donc un quantificateur universel problématique. Cependant, *Eva* n'utilise l'ordre partiel des valeurs abstraites que pour s'assurer de la convergence de certains mécanismes d'amélioration de la précision. Comme ces mécanismes sont tous limités par un nombre d'utilisations borné, il nous suffit de fournir une implémentation de l'ordre partiel retournant toujours faux pour garantir la correction, même si cela peut avoir un impact sur les performances. Notons que ceci n'est vrai qu'au niveau des valeurs abstraites, et qu'il est par contre nécessaire que l'ordre partiel implémenté au niveau du domaine corresponde à celui présenté dans la thèse de Khalil Ghorbal [36].

Pour terminer, notons qu'une implémentation naïve de ces valeurs abstraites causerait de sérieux problèmes de performances. Par exemple, l'interprétation de l'expression $x + x$ dans un contexte ψ quelconque impliquerait de calculer deux fois la forme affine associée à x. Nous recalculerions de multiples fois la même chose. Heureusement, un simple mécanisme de cache permet de faire totalement disparaitre ce problème de performances.

Ainsi, nous disposons d'une implémentation du domaine des zonotopes et du domaine *Numerors* au sein de l'analyseur *Eva*. Cette implémentation est modulaire, simplifiant ainsi l'exploration de nouvelles idées, et respectent les contraintes imposées par *Eva*, permettant ainsi de profiter de ses mécanismes de communication. Nous pouvons désormais présenter les évaluations expérimentales de nos différentes contributions.

Chapitre 9

Evaluations Expérimentales

Dans ce chapitre, nous présentons les évaluations expérimentales que nous avons menées afin de mettre en avant les qualités de nos travaux. Nous avons ici trois objectifs. Le premier est de montrer que l'utilisation du produit réduit entre erreurs absolues et relatives permet d'améliorer significativement la précision de l'analyse à moindre coût, et ce même lorsque l'on utilise un domaine abstrait plus expressif pour représenter les erreurs, comme le domaine des zonotopes. Pour ce faire, nous comparons les bornes estimées par notre domaine abstrait *Numerors* à celles obtenues par différents outils de l'état de l'art que nous pensons former un panel représentatif. Cette comparaison est accompagnée d'une évaluation des différents temps d'exécution. Notre deuxième objectif est de mettre en avant les gains de précision que peuvent apporter les formes relatives par rapport à l'utilisation de formes affines pour l'analyse de l'erreur relative. Nous comparons donc les bornes obtenues en utilisant les formes relatives à celles obtenues par le domaine *Numerors*. La comparaison est effectuée avec les bornes obtenues avec et sans coopération entre les deux erreurs, les formes relatives ne disposant pas, pour le moment, de produit réduit avec les erreurs absolues. Notre troisième objectif est de montrer les gains de précision qu'apporte notre amélioration du produit réduit entre intervalles et formes affines sur l'incertitude introduite par les opérateurs non linéaires. Nous comparons donc les incertitudes introduites par notre méthode avec celles des solutions standards sur un ensemble d'opérandes générées aléatoirement.

9.1 Évaluation du produit réduit entre erreurs absolues et relatives

Nous avons évalué notre domaine abstrait *Numerors* en le comparant à trois outils de l'état de l'art : *Fluctuat* [29], *Daisy* [22, 45] et *FPTaylor* [81]. Nous avons choisi de nous comparer à *Daisy* et *FPTaylor* car ces deux outils représentent

actuellement les deux principales références en terme de calcul de bornes pour l'erreur relative. Quant à *Fluctuat*, c'est l'outil présentant le plus de similarités techniques vis-à-vis de notre approche ; il nous a donc semblé particulièrement intéressant d'évaluer les capacités de notre approche vis-à-vis de *Fluctuat*.

La présentation de cette évaluation est séparée en trois parties. La première présente la suite d'exemples que nous avons utilisée. La deuxième décrit la méthodologie de comparaison que nous avons suivie. En particulier, elle présente les différentes configurations que nous avons utilisées pour chaque outil. La troisième partie est une analyse détaillées des résultats.

9.1.1 Sélection et description des exemples

Les exemples que nous avons utilisés pour évaluer notre approche par rapport à l'état de l'art proviennent principalement de la suite de test *FPBench* [1]. FPBench est un projet collaboratif visant, dans un premier temps, à définir un standard de description de tests autour de l'arithmétique flottante et de la mesure des erreurs qui en découlent, et, dans un second temps, à utiliser ce standard pour proposer un large ensemble de tests. L'objectif est ainsi de permettre de simplifier les comparaisons avec les différents méthodes de l'état de l'art en unifiant les tests utilisés par chacun d'entre eux.

Notre sélection de tests a été guidée par la volonté de garder un ensemble raisonnablement restreint de tests, tout en considérant la plupart des classes d'exemples étudiées par les outils auxquels nous nous comparons, à savoir *Daisy* et *FPTaylor*. Pour restreindre l'ensemble de tests, nous avons exclu les exemples contenant des appels à des fonctions mathématiques comme les fonctions transcendantales, que nous ne traitons pas pour le moment, les variations des mêmes exemples ne montrant pas de comportements différents et les exemples dont les entrées ne sont pas complètement spécifiées car les outils considérés ici ne peuvent pas les traiter. Enfin, nous avons modifié les entrées de certains exemples afin que tous les outils considérés puissent calculer des bornes d'erreurs relatives non triviales.

Nous avons également ajouté quatre exemples afin de mettre en avant différentes caractéristiques qui ne sont pas bien représentées par FPBench, et, en particuliers, les branchements conditionnels. Le test *log_approx* calcule une approximation du logarithme en base deux du carré de son entrée au travers d'une boucle déroulée une quinzaine de fois et d'un développement de Taylor. Le test *conditional_ex* correspond à l'exemple de la section 5.3 où nous expliquions comment tirer parti du produit réduit pour améliorer l'interprétation des conditions. Le test *conditional_ex* est similaire mais avec plus de calculs. Enfin, *sqrt_1* calcule la fonction $\sqrt{2x+3} \div (2\sqrt{x}+3)$. Pour terminer, notons que les autres exemples

1. `http://fpbench.org`

ont déjà utilisé pour évaluer la précision des bornes d'erreurs absolues et relatives calculées par *FPTaylor* [81] et *Daisy* [45].

9.1.2 Méthodologie de comparaison

Pour chaque exemple, notre objectif est de comparer les bornes d'erreurs absolues et relatives calculées par les différents outils considérés, afin d'évaluer la précision de *Numerors* vis-à-vis de l'état de l'art, ainsi que de comparer les temps d'exécution de chaque analyse. Comme les erreurs sont généralement symétriques, nous ne conservons que la valeur absolue de la plus grande erreur possible afin de simplifier la comparaison. Ainsi, si un des outils retourne les bornes $[-2 \times 10^{-12} ; 1 \times 10^{12}]$ pour un des exemples, nous utilisons la valeur 2×10^{-12} pour la comparaison.

Afin de correctement représenter les possibilités des différents outils, nous avons inclus plusieurs configurations pour chacun d'entre eux, correspondant à différents compromis entre performances et précision. *Numerors* en présente ainsi trois :

— *Intervalles*, utilisant des intervalles pour les trois sémantiques ;
— *Mixte*, utilisant le domaine des zonotopes pour la sémantique exacte et celle d'erreur absolue, et des intervalles pour la sémantique d'erreur relative ;
— *Zonotopes*, utilisant le domaine des zonotopes pour les trois sémantiques.

Fluctuat présente une configuration n'utilisant que des intervalles et une n'utilisant que le domaine des zonotopes. Elles sont, comme pour *Numerors*, nommées *Intervalles* et *Zonotopes*. *Daisy* présente également trois configurations. La première, nommée *Dataflow*, utilise une analyse flot de données, similaire au fonctionnement de *Numerors* et *Fluctuat*, ainsi que des intervalles pour les sémantiques exactes et d'erreurs. La deuxième, nommée *Opt*, utilise une analyse fondée sur des techniques d'optimisation, et repose sur des prouveurs SMT pour la sémantique exacte, et des formes affines pour les sémantiques d'erreurs. Enfin, la troisième configuration, nommée *Relative*, repose sur une technique d'optimisation spécialement adaptée à l'analyse d'erreur relative. Pour terminer, *FPTaylor* ne présente qu'une seule configuration, spécifiant d'utiliser le modèle de l'arrondi le plus précis.

Les résultats des analyses calculées par toutes ces configurations sont regroupés dans différents tableaux afin d'en simplifier la compréhension. Le tableau 9.1 présente les temps d'exécution de chaque outil sur chaque exemple. Le tableau 9.2 présente les bornes d'erreurs absolues calculées par les différentes configurations de *Numerors* et de *Fluctuat*. Le tableau 9.3 présente les bornes d'erreurs absolues calculées par la meilleure configuration de *Numerors*, celles calculées par les configurations *Dataflow* et *Opt* de *Daisy* et celles calculées par *FPTaylor*. Enfin, le tableau 9.4 présente les bornes d'erreurs relatives calculées par les différentes configurations de *Numerors*, les configurations *Opt* et *Relative* de *Daisy* et *FPTaylor*.

Exemple	**Numerors** Intervalles	**Fluctuat**	**Daisy** Dataflow	**Numerors** Mixte	**Numerors** Zonotopes	**Fluctuat**	**Daisy** Opt	**Daisy** Relative	**FPTaylor**
log_approx	0.218	0.007	–	0.637	0.790	0.011	–	–	–
conditional_ex	0.191	0.004	–	0.197	0.201	0.005	–	–	0.018
conditional_1	0.193	0.005	–	0.201	0.199	0.005	–	–	0.019
sqrt_1	0.188	0.004	2.122	0.239	0.276	0.005	21.860	3.399	0.024
complex_sqrt	0.193	0.005	2.043	0.246	0.298	0.005	26.797	3.851	0.030
kepler0	0.193	0.006	2.159	0.218	0.225	0.006	23.784	1820.697	16.819
intro_example	0.186	0.004	2.012	0.200	0.185	0.004	7.043	2.920	0.032
sec4_example	0.197	0.005	2.050	0.208	0.227	0.005	20.761	172.829	0.028
test01_sum3	0.192	0.005	2.067	0.193	0.185	0.005	7.848	3.803	0.823
test02_sum8	0.201	0.005	2.164	0.210	0.207	0.005	8.906	887.310	23.361
test03_nonlin2	0.195	0.004	2.075	0.192	0.191	0.005	7.543	2.878	0.026
test04_dqmom9	0.197	0.007	2.487	0.249	0.283	0.008	108.817	14.699	71.617
test05_nonlin1_r4	0.195	0.005	2.026	0.196	0.207	0.005	10.128	32.071	0.015
test05_nonlin1_test2	0.192	0.004	1.993	0.198	0.210	0.005	3.536	13.970	0.013
doppler1	0.190	0.005	2.157	0.212	0.223	0.006	41.369	775.208	13.822
doppler2	0.200	0.005	2.219	0.213	0.225	0.006	42.117	776.996	12.909
doppler3	0.191	0.005	2.162	0.218	0.216	0.005	41.170	774.697	8.816
rigidBody1	0.188	0.005	2.075	0.199	0.181	0.005	12.572	521.092	0.609
rigidBody2	0.196	0.005	2.088	0.214	0.206	0.005	34.469	1168.421	0.403
turbine1	0.182	0.005	2.216	0.240	0.238	0.006	39.368	1469.911	3.419
turbine2	0.190	0.005	2.092	0.221	0.231	0.006	21.144	2998.277	0.627
turbine3	0.191	0.005	2.209	0.219	0.248	0.006	37.988	1710.149	0.659
verhulst	0.192	0.005	2.030	0.198	0.206	0.005	6.363	26.479	0.018
predatorPrey	0.193	0.005	2.102	0.217	0.218	0.005	9.159	45.509	0.027
carbonGas	0.189	0.005	2.101	0.208	0.212	0.005	24.553	68.022	0.032
sine	0.194	0.004	2.101	0.300	0.353	0.005	27.130	141.631	0.035
sqroot	0.194	0.004	2.112	0.206	0.209	0.005	9.816	98.489	0.025
sineOrder3	0.187	0.004	2.059	0.195	0.195	0.004	6.840	40.613	0.021

TABLE 9.1 – Comparaison des temps d'exécutions (en secondes).

9.1.3 Analyse des résultats

Commençons par la comparaison des temps d'exécution des différentes analyses sur chaque exemple, présentée au tableau 9.1. Les résultats sont séparés en deux catégories. La première concerne les trois analyses ne présentant aucun raisonnement relationnel, c'est-à-dire les configurations *Intervalles* de *Numerors* et *Fluctuat* et la configuration *Dataflow* de *Daisy*. La seconde concerne toutes les autres analyses, présentant donc, à divers degrés, des raisonnements relationnels.

Intéressons nous à la première de ces catégories. Si l'on compare les temps d'analyse de *Fluctuat* avec ceux de *Daisy* et *Numerors*, il semble évident que le premier est bien plus rapide que les deux autres. Cependant, il est important de noter que, pour *Numerors* comme pour *Daisy*, il y a un biais introduit par le temps de démarrage de l'outil lui-même. Ce biais est compris entre 0.180 et 0.200 seconde pour *Numerors*, principalement dû aux chargements dynamiques qu'effectue la plateforme *Frama-C* au démarrage, et est d'environ 2 secondes pour *Daisy*. Cela signifie donc qu'il n'y a pas de différences significatives en terme de performance entre ces trois analyses.

Étudions maintenant les temps d'analyse de la seconde catégorie. La première chose que nous pouvons remarquer, c'est que même en prenant en considération le biais de démarrage, les configurations *Opt* et *Relative* de *Daisy* sont bien plus lentes que les autres analyses sur tous les exemples. En particulier, la configuration *Relative* est de loin la plus lente, avec plusieurs ordres de magnitude d'écart sur une grande partie des exemples. À l'inverse, la configuration *Zonotope* de *Fluctuat* présente des temps d'analyse équivalent à ceux de sa configuration *Intervalles* pour tous les exemples. Cela n'est pas étonnant, *Fluctuat* est le plus mature des outils considérés ici. Les configurations *Mixte* et *Zonotopes* de *Numerors* présentent également des temps équivalents à ceux de sa configuration *Intervalles* pour une grande partie des exemples. Cela signifie donc que pour tous ces exemples, les temps d'analyse de la configuration *Zonotopes* de *Numerors* sont équivalents à ceux de *Fluctuat* dans la même configuration lorsque l'on prend en considération le biais de démarrage. Cependant, on remarque tout de même que pour certains exemples, et en particulier pour *log_approx*, les temps d'analyse de *Numerors* augmente significativement lorsque l'on utilise les formes affines. Nous pensons que ces augmentations peuvent cependant être expliqué par l'utilisation de rationnels, particulièrement coûteux lorsqu'ils sont utilisés pour représenter de très petits nombres. *FPTaylor* présente également des temps d'analyse comparables à ceux de *Fluctuat* pour une partie des exemples. Cependant, certains exemples semblent à l'inverse être des cas difficiles à traiter pour *FPTaylor*, avec des temps d'analyse dépassant la seconde pour *kepler0*, *test02_sum8*, *test04_dqmom9*, les trois versions de *doppler* et *turbine1*.

Pour résumer, le tableau 9.1 montre que les temps d'analyse de *Numerors*

sont globalement équivalents à ceux de *Fluctuat*, l'outil le plus mature et le plus rapide de notre sélection, malgré quelques lenteurs probablement dûes à un bug de l'implémentation. *Daisy*, à l'inverse, est bien plus lent dans ces configurations précises. Enfin, *FPTaylor* présente des performances similaires à celles de *Fluctuat* et *Numerors* sur une partie des exemples, mais également des temps d'analyse bien plus longs sur d'autres. Si nous nous comparons à ces deux outils, ce n'est donc pas tellement parce que nous cherchons à être plus précis qu'eux, mais parce que nous souhaitons démontrer que nous sommes capables d'obtenir des bornes d'erreurs généralement proches, et même parfois meilleures, tout en présentant une analyse rapide.

Exemple	**Numerors** Intervalles	**Fluctuat** Intervalles	**Numerors** Mixte	**Numerors** Zonotopes	**Fluctuat** Zonotopes
log_approx	**99.97%**	6.34e-11	**99.97%**	**99.97%**	6.34e-11
conditional_ex	**99.97%**	2.33e-12	**99.97%**	**99.97%**	2.33e-12
conditional_1	**90.34%**	1.68e-11	**90.34%**	**90.34%**	1.68e-11
sqrt_1	**96.53%**	1.44e-13	**98.45%**	**98.45%**	1.24e-13
complex_sqrt	**87.65%**	7.27e-15	**77.72%**	**77.72%**	4.03e-15
kepler0	0.00%	4.89e-13	**2.82%**	**2.82%**	4.89e-13
intro_example	0.00%	1.14e-10	-0.40%	-0.40%	1.13e-10
sec4_example	**94.13%**	2.49e-09	**94.38%**	**94.38%**	2.49e-09
test01_sum3	**18.75%**	5.33e-15	**34.09%**	**34.09%**	4.88e-15
test02_sum8	**11.11%**	7.99e-15	**11.11%**	**11.11%**	7.99e-15
test03_nonlin2	**25.11%**	4.86e-14	**22.22%**	**22.22%**	4.58e-14
test04_dqmom9	**1.14e-04**	∞	**1.14e-04**	**1.14e-04**	∞
test05_nonlin1_r4	**78.57%**	3.89e-06	**78.57%**	**78.57%**	3.89e-06
test05_nonlin1_test2	**20.00%**	1.39e-16	**20.00%**	**20.00%**	1.39e-16
doppler1	**49.42%**	3.90e-13	**49.42%**	**49.42%**	3.90e-13
doppler2	**58.99%**	9.75e-13	**58.99%**	**58.99%**	9.75e-13
doppler3	**35.82%**	1.57e-13	**35.82%**	**35.82%**	1.57e-13
rigidBody1	**8.29%**	3.22e-13	**8.29%**	**8.29%**	3.22e-13
rigidBody2	**1.10%**	3.65e-11	**1.87%**	**1.87%**	3.65e-11
turbine1	**36.71%**	9.76e-14	**32.86%**	**32.86%**	9.20e-14
turbine2	**4.18%**	1.36e-14	**3.74%**	**3.74%**	1.35e-14
turbine3	**36.38%**	7.34e-14	**33.10%**	**33.10%**	6.99e-14
verhulst	**21.18%**	5.04e-16	**13.09%**	**13.09%**	4.24e-16
predatorPrey	**15.69%**	2.11e-16	**14.87%**	**14.87%**	2.09e-16
carbonGas	**41.39%**	4.45e-08	**35.05%**	**35.05%**	4.02e-08
sine	0.00%	7.97e-16	**24.34%**	**24.34%**	7.97e-16
sqroot	**20.05%**	6.83e-16	**22.58%**	**22.58%**	6.83e-16
sineOrder3	**24.16%**	1.15e-15	**43.02%**	**43.02%**	1.14e-15

TABLE 9.2 – Comparaison avec Fluctuat sur les erreurs absolues.

Étudions maintenant les bornes d'erreurs obtenues par les différentes configurations. Sur chaque ligne, les résultats des différentes analyses sont exprimés comme un écart relatif à l'une de bornes, considérées comme référence. L'écart e entre une valeur de référence r et une borne d'erreur b est calculé par la formule $e = (r - b) \div r$ puis exprimé en pourcentage. Si cet écart relatif est positif, alors l'estimation cal-

culée par l'analyse est meilleure que la référence et la valeur correspond au gain de précision. Sinon, l'estimation est moins précise que la référence. De plus, sur chaque ligne, le résultat en gras correspond à l'estimation présentant le plus grand écart relatif, et celui en italique est le deuxième plus grand. Notons qu'une estimation présentant un écart négatif, et donc moins précis que la référence, peut tout de même être présenté en gras. Cela signifie alors que cette estimation est la plus proche de la référence.

Le tableau 9.2 compare les bornes d'erreurs absolues obtenues par les différentes configurations de *Numerors* et de *Fluctuat*. Ces configurations sont séparées en deux catégories. La première correspond aux deux configurations ne reposant que sur des intervalles. La seconde correspond, quant à elle, aux différentes configurations reposant au moins partiellement sur le domaine des zonotopes. Pour ces deux catégories, les références considérées sont les estimations produites par les configurations de *Fluctuat*. Nous remarquons clairement que les différentes configurations de *Numerors* sont toutes plus précises que leurs équivalents pour *Fluctuat* sur tous les exemples, à l'exception de l'exemple *intro_example*, mais l'écart nous semble trop faible pour être significatif. Notons également que pour les trois exemples présentant des conditions, à savoir *log_approx*, *conditional_ex* et *conditional_1*, *Numerors* est significativement plus précis, allant jusqu'à des gains de plusieurs ordres de grandeur pour les deux premiers exemples. Nous remarquons ainsi que l'utilisation du produit réduit avec l'erreur relative permet d'améliorer quasi systématiquement la précision de l'estimation des bornes de l'erreur absolue comparativement à une analyse équivalente mais n'utilisant pas le produit réduit.

Comparons maintenant les meilleures bornes pour l'erreur absolue obtenues par *Numerors* vis-à-vis de *Daisy* et *FPTaylor*, présentées au tableau 9.3. Notons que la configuration *Relative* de *Daisy* n'est pas présente, car dédiée à l'estimation des erreurs relatives. Les références considérées ici sont les estimations de *FPTaylor*. Les valeurs ∞ sont utilisées pour indiquer une estimation largement moins précise que celle de *FPTaylor*, c'est-à-dire présentant un écart relatif inférieur à -1000%, ce qui correspond à une estimation au moins dix fois moins précise que celle obtenue par *FPTaylor*. La première chose que l'on peut remarquer est que les estimations de *FPTaylor* sont quasi systématiquement les plus précises. Les seules exemples pour lesquels une autre analyse fournit la meilleure estimation sont les deux exemples contenant des conditions, et l'exemple *test04_dqmom9*. Pour ces trois exemples, *Numerors* fournit une estimation plus précise de plusieurs ordres de magnitude. Pour les autres exemples, les estimations de *Numerors* sont globalement proches de celles de *FPTaylor* et seuls les quatre exemples *sqrt_1*, *turbine1*, *turbine3* et *carbonGas* présentent des écarts relatifs dépassant l'ordre de grandeur. De plus, nous constatons que notre analyse fournit des estimations plus précises que celles obtenues avec *Daisy* dans tous les autres cas sauf un, et ce peu importe la

Exemple	**Numerors** Zonotopes	**Daisy** Dataflow	**Daisy** Opt	**FPTaylor**
conditional_ex	**99.93%**	–	–	9.09e-13
conditional_1	**92.25%**	–	–	2.09e-11
sqrt_1	**-555.32%**	∞	∞	2.93e-16
complex_sqrt	**-24.35%**	-841.54%	*-824.29%*	7.22e-16
kepler0	**-12.26%**	*-15.52%*	-68.87%	4.23e-13
intro_example	∞	∞	∞	2.22e-16
sec4_example	∞	∞	∞	8.69e-13
test01_sum3	**0.00%**	-65.52%	*-31.06%*	3.22e-15
test02_sum8	**0.00%**	*-12.50%*	-34.37%	7.11e-15
test03_nonlin2	∞	∞	**-238.76%**	4.26e-16
test04_dqmom9	**99.91%**	∞	*99.86%*	1.25e-01
test05_nonlin1_r4	∞	∞	∞	6.58e-09
test05_nonlin1_test2	**0.00%**	*-25.00%*	-75.00%	1.11e-16
doppler1	**-62.13%**	*-258.14%*	-318.33%	1.22e-13
doppler2	**-79.70%**	*-383.04%*	-494.05%	2.23e-13
doppler3	**-51.58%**	*-171.84%*	-222.00%	6.63e-14
rigidBody1	**0.00%**	**0.00%**	-72.25%	2.95e-13
rigidBody2	**0.00%**	*-0.79%*	-77.66%	3.58e-11
turbine1	**-270.07%**	*-434.34%*	-712.73%	1.67e-14
turbine2	**-17.15%**	*-17.72%*	-73.38%	1.11e-14
turbine3	**-388.09%**	*-579.25%*	-935.16%	9.57e-15
verhulst	**-49.07%**	*-64.15%*	-125.12%	2.47e-16
predatorPrey	**-12.13%**	*-12.52%*	-60.23%	1.59e-16
carbonGas	**-342.25%**	-663.99%	*-366.80%*	5.90e-09
sine	**-35.95%**	*-79.67%*	-113.84%	4.43e-16
sqroot	**-5.49%**	*-21.03%*	-74.98%	5.02e-16
sineOrder3	**-9.63%**	-125.70%	*-63.69%*	5.94e-16

TABLE 9.3 – Comparaison avec Daisy et FPTaylor sur les erreurs absolues.

configuration considérée. Ainsi, notre analyse fournit des estimations pour l'erreur absolue proche de celles obtenues par des outils bien moins rapides et reposant sur des techniques bien plus complexes.

Pour terminer, étudions maintenant les bornes pour l'erreur relative. Le tableau 9.4 présente les estimations obtenues par les trois configurations de *Numerors*, ainsi que par *FPTaylor* et par la configuration *Relative* de *Daisy*. Nous n'avons pas inclus les configurations *Dataflow* et *Opt* car elles sont largement moins précises que la troisième sur la majeure partie des exemples. Cependant, la configuration *Relative* ne parvenant pas à borner l'erreur relative pour trois des exemples, nous avons à la place considéré la meilleure estimation obtenue par les deux autres configurations de *Daisy*. Ces bornes sont dénotées par des étoiles dans le tableau 9.4. Nous n'avons pas non plus inclus de configuration pour *Fluctuat*. En effet, le seul calcul d'erreur relative que ce dernier peut fournir est un calcul *a posteriori*, largement moins précis que les estimations présentées ici.

Comme pour les erreurs absolues, les estimations que nous utilisons comme référence sont celles de *FPTaylor*, qui est le plus précis pour la vaste majo-

Exemple	**Numerors** Intervalles	**Numerors** Mixte	**Numerors** Zonotopes	**Daisy** Relative	**FPTaylor**
conditional_ex	**-195.65%**	**-195.65%**	**-195.65%**	–	1.13e-16
conditional_1	**-130.77%**	**-130.77%**	**-130.77%**	–	7.22e-16
sqrt_1	**-33.43%**	**-33.43%**	**-33.43%**	∞	4.52e-16
complex_sqrt	**-2.78%**	**-2.78%**	**-2.78%**	∞	3.20e-16
kepler0	*-31.98%*	**-31.58%**	**-31.58%**	-37.61%	7.50e-16
intro_example	–	–	–	–	∞
sec4_example	*97.67%*	*97.67%*	*97.67%*	**98.49%**	1.55e-11
test01_sum3	∞	**-64.42%**	**-64.42%**	∞	6.53e-16
test02_sum8	**-25.14%**	**-25.14%**	**-25.14%**	*-37.22%*	5.63e-16
test03_nonlin2	–	–	–	–	∞
test04_dqmom9	–	–	–	–	∞
test05_nonlin1_r4	**100.00%**	**100.00%**	**100.00%**	*-62.32%*	1.03e-05
test05_nonlin1_test2	**-11.63%**	**-11.63%**	**-11.63%**	*-32.41%*	2.24e-16
doppler1	*-19.42%*	*-19.42%*	*-19.42%*	**-12.54%**	1.12e-15
doppler2	*-26.54%*	*-26.54%*	*-26.54%*	**-14.75%**	1.19e-15
doppler3	*-33.92%*	*-33.92%*	*-33.92%*	**-32.22%**	8.60e-16
rigidBody1	**-4.86%**	**-4.86%**	**-4.86%**	*-53.59%*	6.35e-16
rigidBody2	*-16.29%*	**-16.11%**	**-16.11%**	-25.61%	9.28e-16
turbine1	∞	∞	∞	**-66.65%**	1.05e-15
turbine2	*-45.93%*	*-45.93%*	*-45.93%*	**-33.83%**	5.17e-16
turbine3	∞	∞	∞	**-75.50%**	3.71e-15
verhulst	**-17.50%**	**-17.50%**	**-17.50%**	*-38.23%*	3.32e-16
predatorPrey	**-3.42%**	**-3.42%**	**-3.42%**	*-18.06%*	5.82e-16
carbonGas	*-19.91%*	*-19.91%*	*-19.91%*	**2.31%**	8.30e-16
sine	-464.52%	*-67.71%*	*-67.71%*	**-41.26%**	4.47e-16
sqroot	-46.89%	**-22.39%**	**-22.39%**	*-27.14%*	4.45e-16
sineOrder3	∞	**-8.51%**	**-8.51%**	*-31.24%*	6.81e-16

TABLE 9.4 – Comparaison avec Daisy et FPTaylor sur les erreurs relatives.

rité des exemples. *Daisy* fournit une estimation plus précise sur les exemples *sec4_example* et *carbonGas*, et *Numerors* est le plus précis uniquement pour l'exemple *test05_nonlin1_r4*. Notons cependant que sur les deux exemples contenant des conditions, *FPTaylor* n'est plus précis que parce que nous n'avons pas réussi à le configurer de manière à prendre en considération les erreurs sur les entrées à travers la condition. Si nous retirons les erreurs des entrées, *Numerors* fournit de meilleurs estimations que *FPTaylor* sur ces deux exemples. Il nous semble important de souligner que sur tous ses exemples, l'estimation de *Numerors* reste très proche de celles calculées par les autres outils. Notre approche fournit en effet la deuxième estimation la plus précise dans tous les autres exemples sauf trois, pour lesquels l'estimation reste dans le même ordre de grandeur.

L'ensemble de ces résultats permet de mettre en avant l'intérêt du produit réduit entre les erreurs. Les différentes configurations de *Numerors* rivalisent en effet avec des outils à l'état de l'art et utilisant des techniques bien plus complexes et coûteuses. Il nous semble cependant nécessaire de noter un dernier point. Le tableau 9.2 et, dans une moindre mesure, le tableau 9.4 permettent de mettre en

avant que l'utilisation du domaine des zonotopes pour les sémantiques exactes et d'erreurs absolues permettent, dans certains cas, d'améliorer la précision de l'analyse. Cependant, ils mettent également en avant que l'utilisation des zonotopes pour l'erreur relative ne permet pas d'améliorer la précision, ni pour l'erreur absolue, ni pour l'erreur relative. En effet, sur tous les exemples, les configurations *Mixte* et *Zonotopes* de *Numerors* présentent exactement les mêmes résultats. Nous avons plusieurs hypothèses pour expliquer ce phénomène. La première est que notre ensemble de tests ne contient pas d'exemples pour lesquels il est nécessaire de disposer de relations sur l'erreur relative. C'est, à notre avis, l'hypothèse la plus probable car nous remarquons que l'utilisation du domaine des zonotopes n'améliore l'estimation des erreurs absolues que dans peu de cas, suggérant que peu de nos exemples nécessite des raisonnements relationnels sur les erreurs. Cependant, notre outil présente des limitations qui rendent difficiles de trouver des exemples intéressants, et nous n'avons ainsi pas encore réussi à inclure de tels exemples à notre jeu de tests. La deuxième hypothèse est que le domaine des zonotopes n'est pas adapté pour représenter les relations de l'erreur relative. C'est pour explorer cette hypothèse que nous avons travaillé à définir l'abstraction présentée au chapitre 7. La troisième et dernière hypothèse, plus pessimiste en termes d'opportunité de recherche, est qu'il ne soit en pratique pas nécessaire d'utiliser des raisonnements relationnelles pour l'erreur relative.

9.2 Évaluation des formes relatives

Intéressons nous maintenant à l'évaluation des formes relatives. Kévin Youyou a implémenté, durant son stage [90], un prototype permettant d'analyser des programmes consistant en une suite d'affectations de variables avec des expressions présentant tous les opérateurs arithmétiques de notre langage. Nous avons ainsi pu traduire les exemples que nous avons utilisés dans la section précédente, et avons regroupé les bornes estimées par ce prototype pour l'erreur relative dans le tableau 9.5. Ce dernier présente deux estimations pour les formes relatives. La première, nommée *Sans optim*, utilise la méthodologie par intervalles pour borner l'incertitude introduite par les opérations linéaires. La seconde, nommée *Avec optim*, utilise, quant à elle, la méthodologie reposant sur les formes affines, plus coûteuse mais plus précise.

Afin de simplifier la lecture, nous n'avons conservé que les exemples pour lesquels la propagation d'une abstraction pour l'erreur relative en dehors du produit réduit avec l'erreur absolue et en utilisant des formes affines pour la sémantique exacte permet de calculer une borne finie. En effet, il est important de noter que notre prototype ne propage pas d'abstraction pour l'erreur absolue, et donc que les estimations présentées au tableau 9.5 ne profitent pas du produit réduit. Afin de

Exemple	Formes relatives		Numerors	
	Sans optim	Avec optim	Sans produit	Avec produit
kepler0	9.91e-16	**9.29e-16**	9.91e-16	*9.87e-16*
sec4_example	**3.61e-13**	**3.61e-13**	*3.62e-13*	*3.62e-13*
test02_sum8	7.05e-16	7.05e-16	7.05e-16	7.05e-16
test05_nonlin1_r4	2.78e-11	2.78e-11	2.78e-11	2.78e-11
test05_nonlin1_test2	2.68e-16	2.68e-16	2.68e-16	**2.50e-16**
rigidBody1	6.66e-16	6.66e-16	6.66e-16	6.66e-16
rigidBody2	*1.06e-15*	**1.03e-15**	1.08e-15	1.08e-15
turbine2	**6.99e-16**	**6.99e-16**	8.66e-16	*7.55e-16*
verhulst	3.90e-16	3.90e-16	3.90e-16	3.90e-16
predatorPrey	6.04e-16	6.04e-16	6.04e-16	**6.02e-16**
carbonGas	1.01e-15	1.01e-15	*9.95e-16*	**7.50e-16**
sine	2.03e-15	*8.81e-16*	2.66e-15	**7.50e-16**
sineOrder3	1.65e-14	*1.12e-15*	2.20e-14	**7.39e-16**
rational_fraction	*7.32e-14*	**6.25e-15**	1.03e-13	8.53e-14

TABLE 9.5 – Comparaison entre les formes affines et les formes relatives pour l'analyse de l'erreur relative.

présenter une évaluation pertinente, nous avons donc comparé ces estimations à la fois avec celles calculées par *Numerors* lorsque l'on désactive le produit réduit, et celles calculées avec le produit réduit. Comme précédemment, la meilleure estimation pour chaque exemple est en gras, et la deuxième meilleure en italique. Notons également que nous avons ajouté un nouvel exemple nommé *rational_fraction* sur lequel nous reviendrons plus en détails par la suite.

Le premier point que met en avant le tableau 9.5 est qu'une partie de nos exemples ne sont tout simplement pas suffisamment complexes, ou ne présentent pas suffisamment de relations, pour qu'un domaine abstrait relationnel soit nécessaire pour l'analyse de l'erreur relative. En effet, pour quatre des exemples, à savoir *test02_sum8*, *test05_nonlin1_r4*, *rigidBody1* et *verhulst*, les estimations des différentes analyses ne présentent aucune différence. Les exemples *test05_nonlin1_test2* et *predatorPrey* sont dans un cas similaire, car seule l'utilisation du produit réduit permet d'améliorer l'estimation. Enfin, pour l'exemple *carbonGas*, l'analyse de *Numerors* n'utilisant pas le produit réduit présente un léger gain par rapport aux formes relatives. Le gain ne nous semble cependant pas significatif, et nous le pensons *a priori* plutôt dû à un bug de l'implémentation. Nous avons cependant prévu d'étudier ce cas plus en détail. L'exemple *sec4_example* présente un léger gain des formes relatives, mais il nous semble trop faible pour être réellement significatif. Ce sont donc sept de nos exemples sur quatorze pour lesquels il ne semble pas nécessaire de disposer de raisonnements relationnels pour calculer une estimation précise de l'erreur relative, favorisant l'hypothèse que nous formulions précédemment, stipulant que notre jeu de tests est trop limité.

Cependant, pour les autres exemples, le tableau 9.5 met en avant des écarts significatifs entre les estimations calculées par les différentes analyses. Ainsi, pour

les exemples *kepler0* et *rigidBody2*, l'utilisation des formes relatives optimisées permet de calculer une estimation légèrement plus précise que celle obtenue par *Numerors* avec le produit réduit. À l'inverse, pour les exemples *sine* et *sineOrder3*, les formes relatives ne permettent pas de calculer une meilleure estimation que celle obtenue grâce au produit réduit. Cependant, par rapport à celles obtenues par *Numerors* sans le produit réduit, les formes relatives permettent d'améliorer la précision de plus d'un ordre de grandeur. Enfin, les formes relatives sont également bien plus précises que *Numerors* pour l'exemple *rational_fraction*, et ce que ce soit avec ou sans l'utilisation du produit réduit.

Étudions donc ce dernier exemple plus en détail. Il correspond à un programme permettant d'évaluer la fonction $f(x)$ définie comme :

$$f(x) = \frac{2x^{12} + x^{11} - 0.4x^9 + 0.6x^4 + 0.2x}{3.1x^{10} - 1.1x^7 + 0.8x^5 - 0.1x} \tag{9.1}$$

pour x compris entre 1.2 et 1.5. Nous avons choisi d'ajouter cet exemple car il présente de bonnes caractéristiques pour évaluer l'intérêt des formes relatives. En effet, il présente beaucoup de relations entre les différentes instances de x et repose sur de nombreuses opérations non linéaires, tout en présentant également des opérations linéaires. Notons que, comme notre langage ne comporte pas d'opérateur arithmétique pour le calcul de puissance, chaque terme en x^n est calculé comme une suite de produits conçue pour minimiser le nombre d'opérations nécessaires. Ainsi, x^{12} est calculé comme le carré de x^6, lui-même calculé comme le carré de x^3, qui est lui calculé par l'expression $(x \times x) \times x$. Les coefficients ont, quant à eux, été choisis de manière à ce que le calcul reste strictement positif à chaque étape, afin que les analyses puissent calculer une borne pour l'erreur relative.

Cet exemple permet de mettre en avant un réel avantage des formes relatives par rapport aux formes affines pour l'estimation de l'erreur relative. En effet, leur utilisation permet de gagner en précision, même sans utiliser l'optimisation pour les opérations linéaires, ce qui montre bien qu'une abstraction spécifiquement conçue pour conserver les relations à travers la sémantique de l'erreur relative est intéressante. Cependant, pour cet exemple de fraction rationnelle, le gain reste modéré par rapport à l'estimation calculée par *Numerors* en utilisant le produit réduit. Le gain est beaucoup plus impressionnant avec l'optimisation pour les opérations linéaires. En effet, l'estimation ainsi obtenue est plus précise d'au moins un ordre de grandeur vis-à-vis de celles obtenues par *Numerors*. Le gain est tel qu'un calcul *a posteriori* de l'erreur absolue permet d'obtenir une estimation plus précise que celle de *Numerors* en utilisant les bornes pour la sémantique exacte obtenues avec les formes affines, à savoir $[0.0677 ; 149.968]$. En effet, l'erreur absolue calculée par *Numerors* est bornée par 1.22×10^{-11} alors que celle *a posteriori* est

$$\left[-6.25 \times 10^{-15} ; 6.25 \times 10^{-15}\right] \times [0.0677 ; 149.968] \leq 9.373 \times 10^{-13}$$

ce qui montre que l'intégration des formes relatives à *Numerors* est une évolution particulièrement intéressante de nos travaux. Cependant, bien qu'elles permettent effectivement d'améliorer les bornes estimées, cela ne permet pas d'atteindre la précision de *FPTaylor*. En effet, à titre de comparaison, la borne pour l'erreur relative calculée par *FPTaylor* pour cet exemple est 2.59×10^{-15}, et celle pour l'erreur absolue est 4.38×10^{-15}.

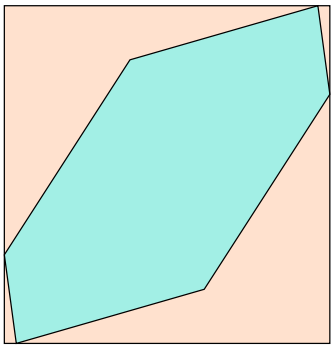
Ces différentes évaluations expérimentales permettent de montrer un intérêt pratique au produit réduit entre les erreurs absolues et relatives, ainsi qu'aux formes relatives. Nous avons en effet mis en avant des exemples pour lesquels l'utilisation de raisonnements relationnels permet d'améliorer significativement la précision pour l'erreur relative, ce qui nous permet donc d'écarter la troisième hypothèse que nous avions formulé à la fin de la section précédente. Cependant, comme nous l'avons souligné à plusieurs reprises, notre ensemble de tests présente des faiblesses et il nous semble donc important de prendre du recul vis-à-vis de ces résultats et de rester prudent. L'un de nos objectifs futurs est ainsi de mener une campagne d'évaluation plus poussée, afin de consolider nos résultats prometteurs. Nous reviendrons sur ces objectifs plus en détails au prochain chapitre.

9.3 Évaluation du calcul de l'incertitude des opérations non linéaires

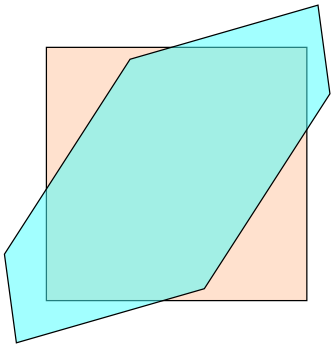
Intéressons nous maintenant à l'évaluation de notre approche de calcul des incertitudes introduites par les opérations non linéaires, présentée au chapitre 6. Notre méthodologie expérimentale consiste à générer aléatoirement des paires de formes affines partageant tout ou partie de leurs symboles de bruit, puis à réduire les intervalles qui leurs sont associés, avant d'évaluer l'incertitude introduite, d'un côté en utilisant l'approche de I. Skalna et M. Hlad´ik [80] ne prenant pas en compte les intervalles, et de l'autre en utilisant notre approche. L'objectif est donc de montrer, d'une part, que notre approche permet de calculer plus précisément l'incertitude dès lors que les intervalles ont été réduits, et d'autre part, que le coût de notre approche reste raisonnable.

Cette évaluation présente une difficulté méthodologique de taille : le choix de réduction des intervalles a un impact énorme sur les résultats, à la fois en termes de précision et de performances. De plus, ce choix doit être cohérent pour toutes les paires d'opérandes pour permettre une comparaison pertinente. Le choix de réduction que nous avons effectué consiste à réduire la surface de la boîte formée par les intervalles des opérandes par un facteur donné, tout en conservant une forme proche de celle de départ. Illustrons cela sur un exemple. La figure 9.1 présente quatre réductions possibles de la boîte associée à un zonotope donné.

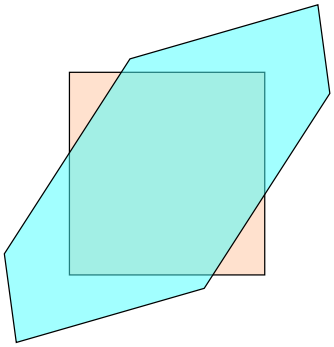
En haut à gauche, la sous figure 9.1a présente une boîte sans réduction. En bas à gauche, la sous figure 9.1b présente une simple mise à l'échelle de 60%. Cette transformation conserve la forme, et est donc acceptable pour notre évaluation. En haut à droite, la sous figure 9.1c présente une transformation conservant une forme proche de celle de départ, mais pas identique : le centre de la boîte n'est pas celui du zonotope. Cette transformation reste suffisamment proche de la forme de départ, et est donc également acceptable pour notre évaluation. Enfin, en bas à droite, la sous figure 9.1d présente une transformation modifiant profondément la forme de la boîte, et qui n'est dont pas acceptable pour notre évaluation. La limite entre déformations acceptables et inacceptables est arbitraire. Nous considérons ici qu'une déformation est acceptable si le ratio entre la largeur et la hauteur de la boîte est équivalent à celui de la boîte de départ avec une marge de 10%.



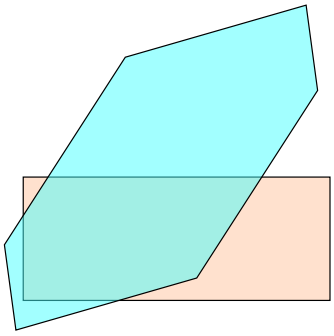
(a) Sans réduction



(c) Légère transformation



(b) Mise à l'échelle



(d) Trop grande transformation

FIGURE 9.1 – Différentes réductions possibles de la boîte.

Le choix de n'utiliser que des réductions conservant une forme proche de celle de départ est motivé par le fait que ce type de réductions permet d'éprouver notre approche pour deux raisons. La première est que, en réduisant la surface tout en conservant la forme, ces réductions maximisent le nombre d'entrées et de sorties de la boîte durant le parcours, ce qui ralentit fortement notre approche. La seconde

est que cela maximise également la surface commune au zonotope et à la boîte, ce qui réduit les gains potentiels en précision. Notre évaluation cherche donc à prendre en défaut notre approche, et permet de montrer que, même dans ce cas de figure, notre approche apporte un gain en précision significatif tout en offrant des performances acceptables.

9.3.1 Protocole expérimental

Pour tester notre approche, nous avons généré des paires d'opérandes pour différentes configurations faisant varier trois paramètres : la réduction de la surface de la boîte, le nombre de symboles de bruit, et le pourcentage de symboles communs aux deux opérandes. Pour le premier paramètre, nous avons utilisé des réductions allant de 10% à 50% par pallier de 10%. Pour le deuxième paramètre, nous avons considéré des formes affines contenant entre 10 et 100 symboles, par pallier de 10, puis entre 100 et 500 symboles, par pallier de 100. Enfin, pour le troisième paramètre, nous avons considéré des opérandes partageant entre 0% et 100% de leurs symboles, par pallier de 20%. Au total, nous avons donc testé notre approche pour 420 configurations différentes. Pour chacune de ces configurations, nous avons généré 1000 paires d'opérandes. Les coefficients de ces paires sont des entiers choisis aléatoirement entre -10 et 10. Pour chaque paire, nous avons également fait varier aléatoirement la forme de la boîte, tout en restant suffisamment proche de la forme originale, comme nous l'expliquions précédemment.

À partir de ces données, l'expérience consiste, pour chaque paire d'opérandes, à calculer l'incertitude introduite par la multiplication en utilisant, d'un côté, l'approche de I. Skalna et M. Hlad´ik, ne prenant pas en compte les intervalles, et d'un autre côté, notre approche. Ensuite, pour chaque configuration, nous calculons quatre métriques : la moyenne et l'écart type du ralentissement causé par notre approche, le nombre de cas pour lesquels l'incertitude calculée par notre approche est plus précise, le nombre de cas pour lesquels elle est égale à l'incertitude calculée par l'approche ne prenant pas en compte les intervalles, et enfin la moyenne et l'écart type du gain en précision apporté par notre approche. Pour réduire de possibles fluctuations sur les temps d'exécution causées par des facteurs extérieurs, chaque calcul est répété 100 fois. L'expérience a été réalisé sur un ordinateur disposant d'un processeur AMD Ryzen Threadripper 1920X, disposant de 24 cœurs de calcul cadencés à 3.5GHz, et de 32GB de mémoire vive. L'exécution complète en profitant du parallélisme a demandé un peu moins de six heures.

9.3.2 Analyse des résultats

Commençons par étudier le ralentissement causé par notre approche. La figure 9.2 présente, pour chaque configuration, le ralentissement, calculé comme

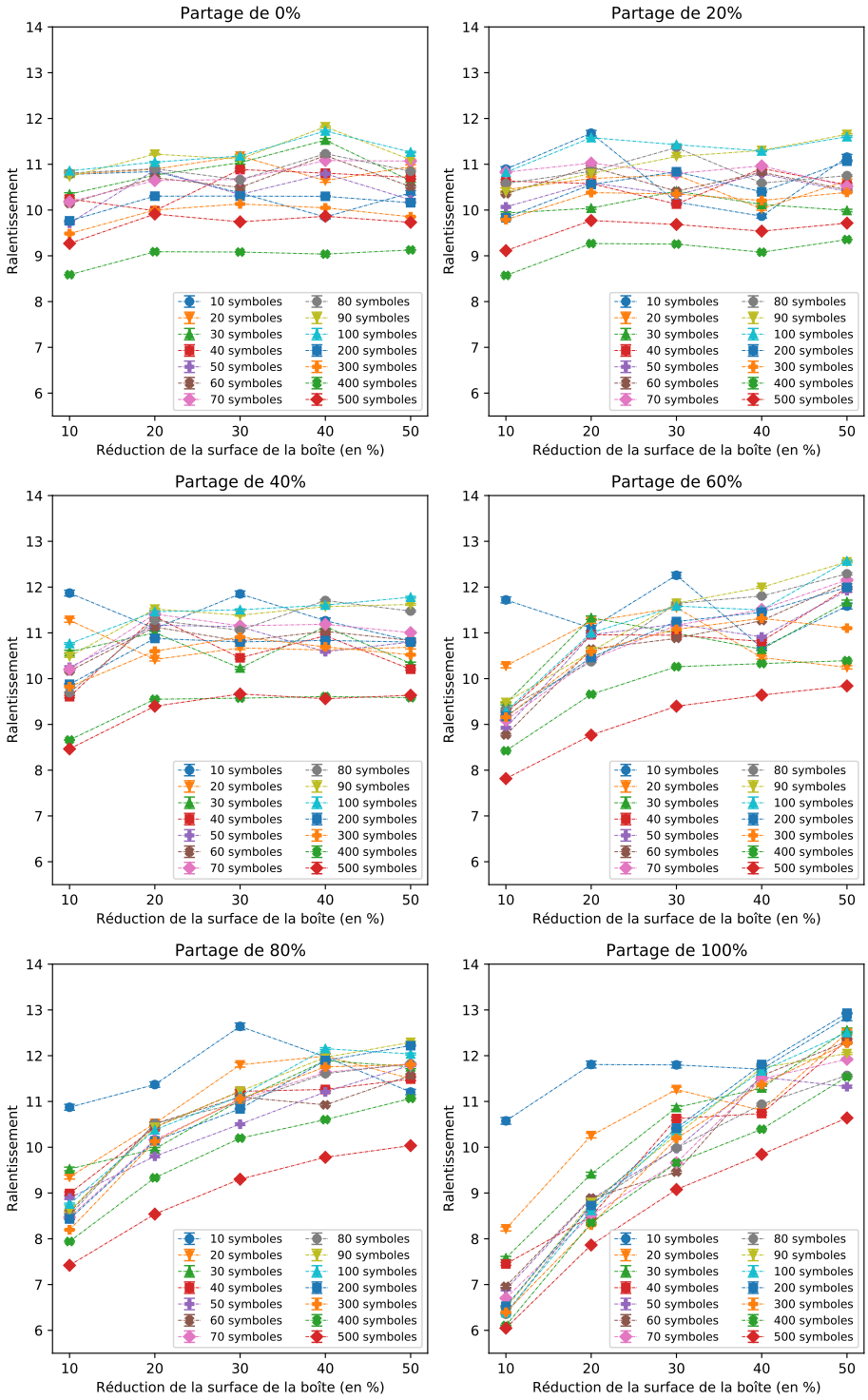


FIGURE 9.2 – Ralentissement en fonction de la réduction de surface pour les différentes configurations de partage des symboles de bruit.

le rapport entre, au numérateur, le temps d'exécution de notre approche et, au dénominateur, le temps d'exécution de l'approche ne prenant pas en compte les intervalles. Afin de faciliter la lecture, les résultats sont séparés en six graphes, un pour chaque configuration du pourcentage de symboles communs, chacun de ces graphes comportant une courbe par configuration du nombre de symboles de bruit.

La première point que l'on peut noter est que les ralentissements sont, pour toutes les configurations, compris entre 6 et 14. De plus, le nombre de symboles de bruit n'impacte pas le ralentissement observé. En effet, pour chaque graphe, les courbes sont toutes très proches les unes des autres. Ce premier point est assez cohérent, les deux approches étant dans la même classe de complexité. Les ralentissements que nous observons ici sont assez élevés. Cependant, il est important de considérer que nous avons délibérément choisi des opérandes coûteuses avec notre approche. Dans la plupart des cas pratiques, le ralentissement sera donc moins important.

La deuxième point à noter est que, plus le partage entre les opérandes est élevé, plus la réduction de la surface de la boîte a un impact sur les performances. Plus précisément, si les opérandes partagent beaucoup de relations, et que leurs intervalles n'ont subit qu'une faible réduction, alors prendre en compte ces intervalles est peu coûteux. Cela peut être expliqué par le fait que, dans ce type de configurations, la plupart des sommets du zonotope appartiennent à la boîte, et donc que construire le bord de l'intersection entre le zonotope et la boîte revient pratiquement à simplement construire le bord du zonotope. À l'inverse, quand on réduit fortement la surface de la boîte, la plupart des sommets du zonotope vont être à l'extérieur. Dans ce cas, notre approche vérifie à chaque étape du parcours s'il existe une intersection avec la boîte, ce qui augmente le coût du calcul.

Étudions maintenant la métrique concernant le nombre de cas où notre approche est plus précise, ainsi que celle comptant le nombre de fois où les deux approches calculent la même incertitude. La figure 9.3 présente, pour chaque configuration, le nombre de fois où notre approche calcule une incertitude strictement plus précise que celle calculée par l'approche de I. Skalna et M. Hlad´ik. La figure 9.4 présente, quant à elle, le nombre de fois où les deux approches calculent la même incertitude. Comme précédemment et pour ces deux figures, les résultats sont séparés en six graphes, un pour chaque configuration du pourcentage de symboles communs. Chacun de ces graphes présente, pour chaque configuration de la réduction de surface, la métrique sous formes de barres, une par configuration du nombre de symboles.

Un premier point important à noter est que l'approche ne prenant pas en compte les intervalles ne permet jamais de calculer une meilleure incertitude que la notre. En effet, en comparant les figures 9.3 et 9.4, nous remarquons que, pour

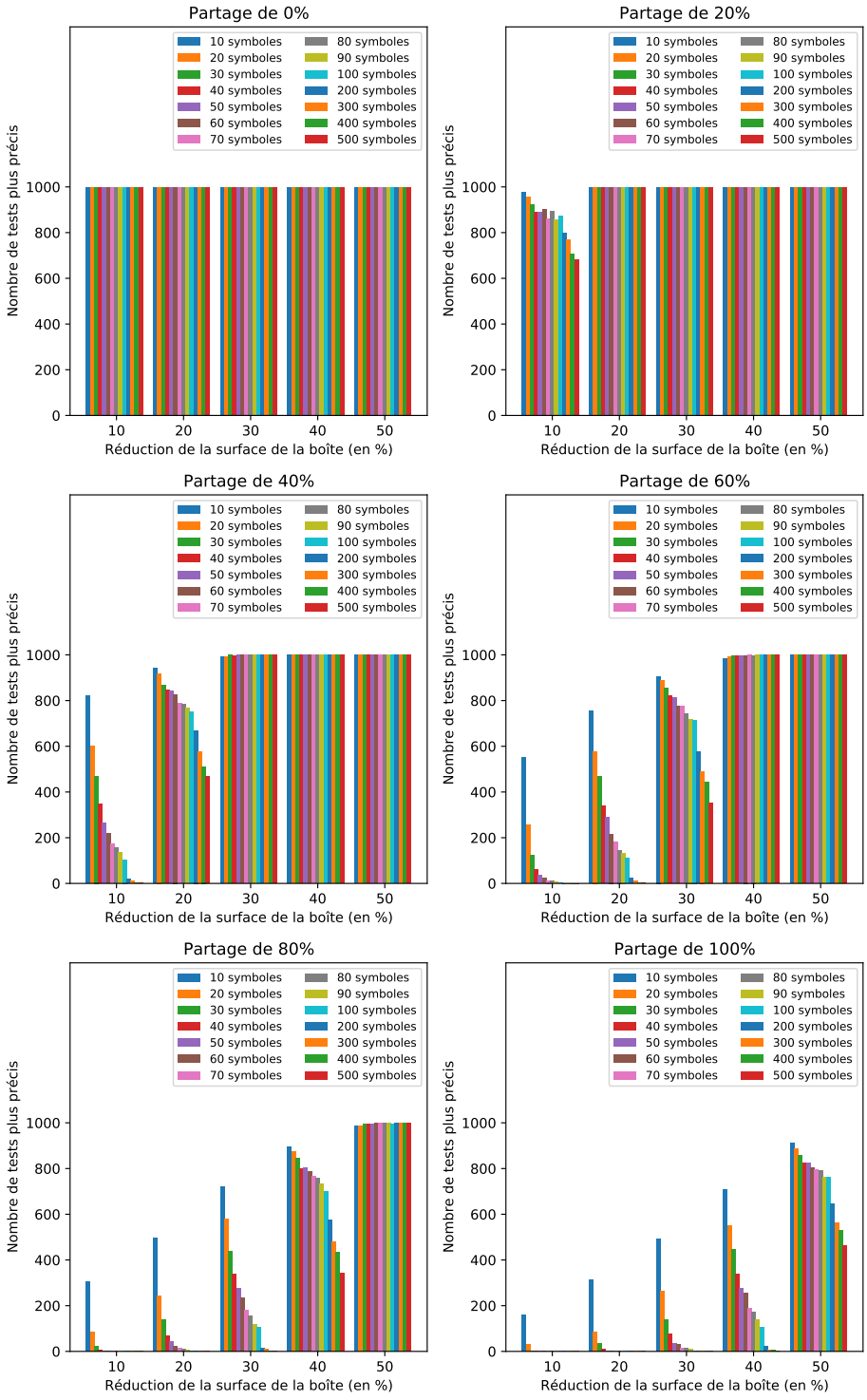


FIGURE 9.3 – Nombre d'incertitudes plus précises par notre approche en fonction de la réduction de surface pour les différentes configurations de partage des symboles de bruit.

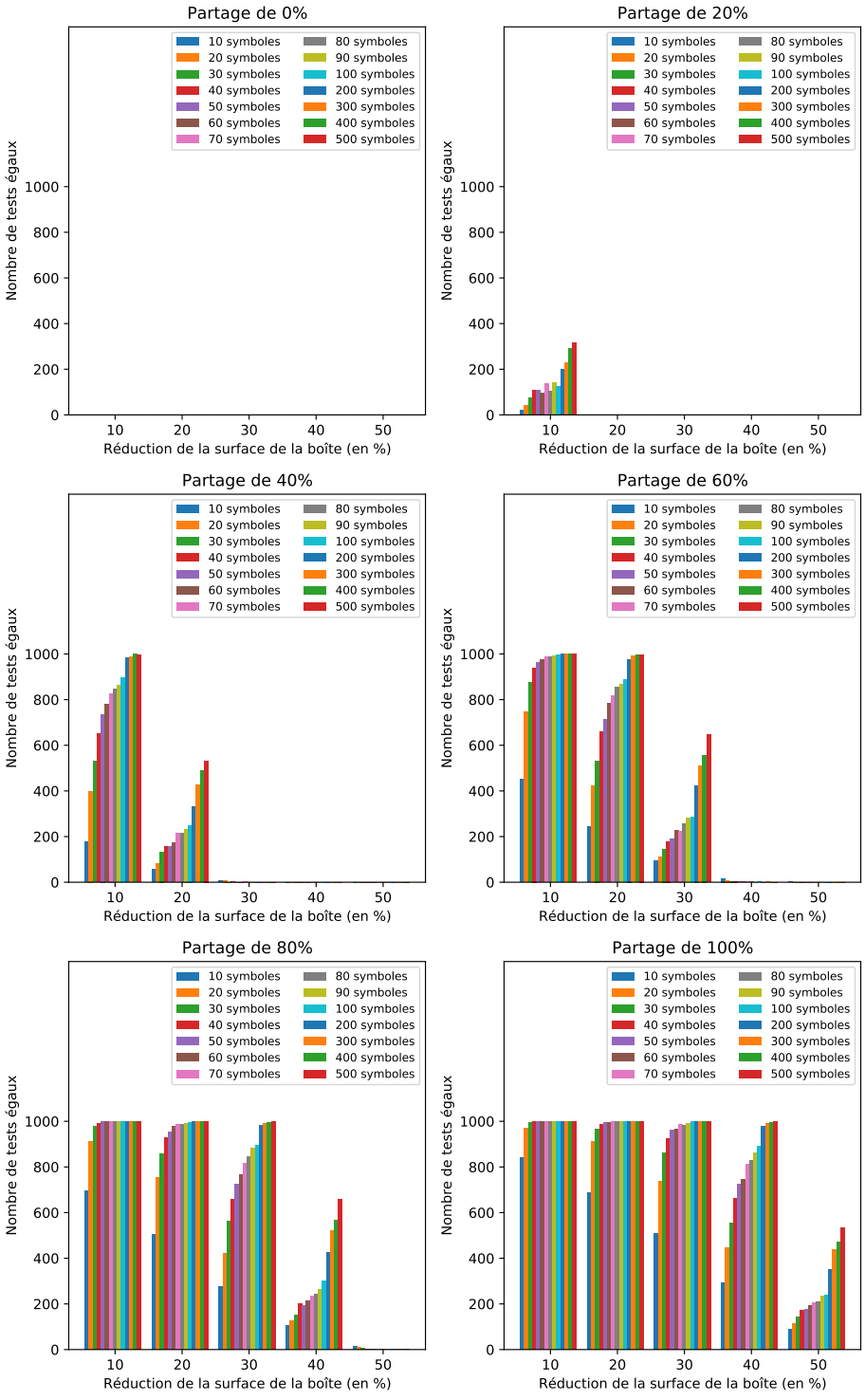


FIGURE 9.4 – Nombre de tests pour lesquels les deux approches calculent la même incertitude en fonction de la réduction de surface pour les différentes configurations de partage des symboles de bruit.

chaque configuration, la somme entre le nombre de tests où notre approche est plus précise et le nombre de tests où les deux approches calculent la même incertitude est toujours égal à 1000. C'est un résultat attendu, notre approche consistant à prendre en considération les intervalles au sein de celle de I. Skalna et M. Hlad´ik. Il nous suffit donc de nous intéresser à la première de ces métriques, la seconde pouvant se calculer directement.

Un deuxième point que l'on peut noter est que, moins les opérandes partagent de relations, plus notre approche permet de calculer une meilleure incertitude. Ce résultat est assez cohérent : moins les opérandes partagent de relations et plus le zonotopes qu'elles forment s'approche de la boîte non réduite, et donc plus l'intersection avec la boîte réduite correspond à la boîte réduite. À l'extrême, quand les opérandes ne partagent aucune relation, l'intersection entre le zonotope et la boîte réduite est strictement équivalente à la boîte réduite, et notre approche calcule toujours une meilleure incertitude. Notons que ce type de cas est en pratique fréquent. Une optimisation intéressante de notre approche pourrait donc consister à commencer par vérifier si les opérandes partagent des relations, et de simplement faire le tour de la boîte si ce n'est pas le cas. Nous obtiendrions ainsi la même précision, et comme vérifier s'il existe des relations est une opération de complexité $O(n)$, nous obtiendrions également de meilleures performances.

Un troisième point à noter est que, plus les opérandes partagent de relations, plus il faut réduire la surface de la boîte pour que notre approche permette d'améliorer l'incertitude. Ceci découle directement du fait que nous avons spécifiquement choisi des réductions conservant la forme, et maximisant donc la surface de l'intersection entre le zonotope et la boîte. Néanmoins, même dans ce scénario défavorable, notre approche permet d'améliorer régulièrement l'incertitude, sauf dans les cas extrêmes avec énormément de relations partagées et une réduction très faible de la surface de la boîte.

Terminons en étudiant les gains en précision apportés par notre approche. La figure 9.5 présente, pour chaque configuration, le gain en précision de notre approche. Pour e_{old}, l'incertitude calculée par l'approche de I. Skalna et M. Hlad´ik, et e_{new}, l'incertitude calculée par notre approche, ce gain est calculé comme $(e_{old} - e_{new}) \div e_{old}$ puis exprimé en pourcentage. Comme précédemment, les résultats sont séparés en six graphes, un pour chaque configuration du pourcentage de symboles communs, chacun de ces graphes comportant une courbe par configuration du nombre de symboles de bruit.

Comme pour la métrique précédente, nous notons que plus les opérandes partagent de relations, moins notre approche a d'impact sur l'incertitude. Nous notons également que le nombre de symboles de bruit des opérandes n'a que très peu d'impact. On ne remarque en effet pratiquement pas de différences entre les courbes de chaque graphe. Une petite différence apparait néanmoins lorsque le ratio de

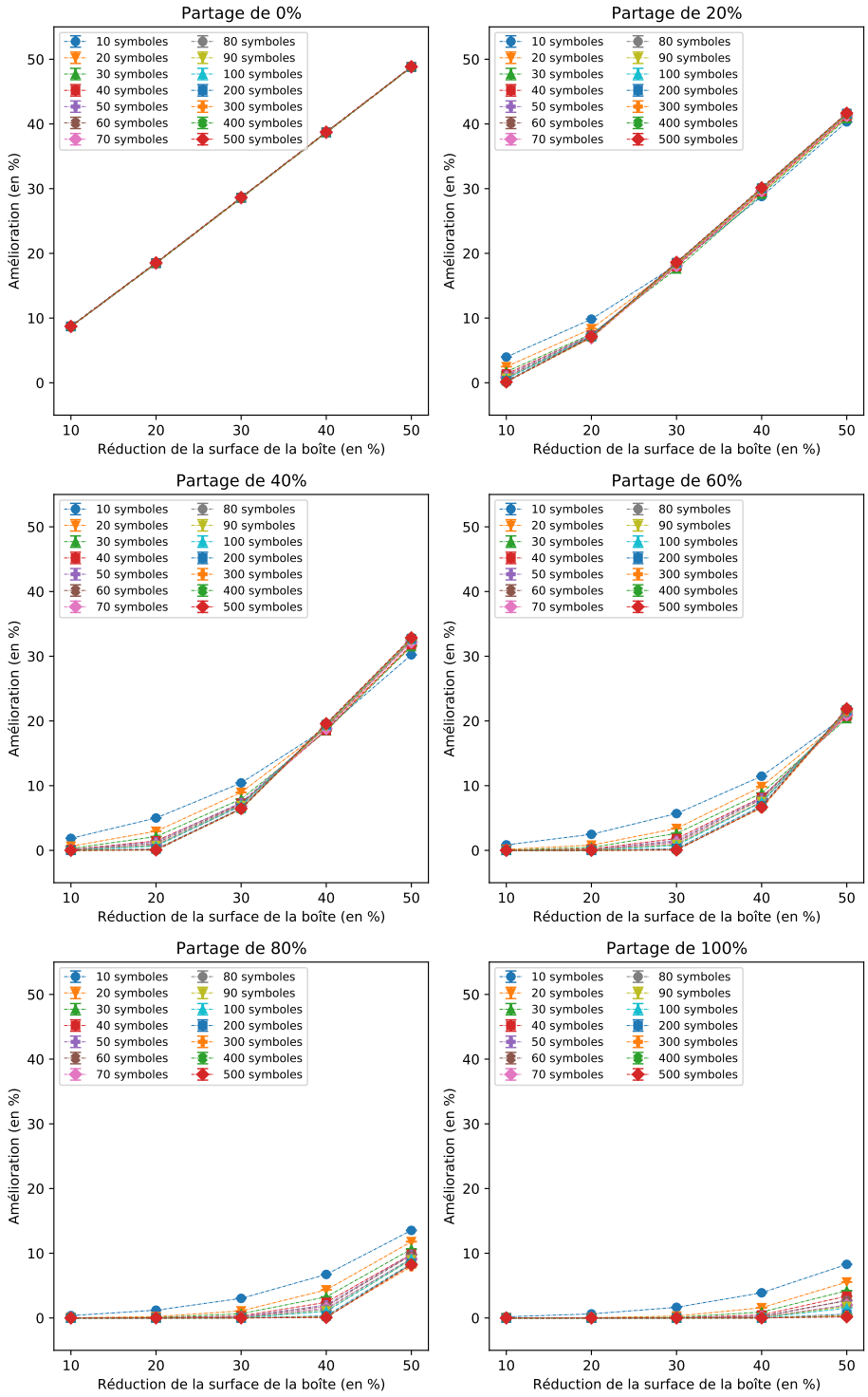


FIGURE 9.5 – Gain de précision en fonction de la réduction de surface pour les différentes configurations de partage des symboles de bruit.

partage augmente, mais elle reste très faible.

Nous pouvons constater que lorsque les opérandes ne partagent aucune relation, notre approche permet d'obtenir un gain de précision quasiment égal à la réduction de surface considérée. Cette amélioration optimale est une raison supplémentaire de considérer l'optimisation que nous expliquions précédemment. Nous constatons également que plus le partage augmente, plus les courbes s'affaissent, et plus il faut réduire fortement la surface de la boîte pour obtenir un gain de précision significatif. Comme précédemment, ce comportement découle de notre choix désavantageux sur la réduction des intervalles, et comme précédemment, notre approche apporte néanmoins régulièrement un gain de précision significatif, sauf dans les configurations les plus extrêmes.

À travers ces différentes expérimentations, nous avons ainsi pu montrer que nos contributions théoriques permettent d'améliorer significativement la précision des analyses de programmes dédiées aux erreurs numériques reposant dessus. De plus, bien que ces gains présentent un coût en termes de performances, ce dernier reste néanmoins très faible comparativement à l'amélioration apportée. Nous avons également montré que, dans certaines configurations, il était possible de mettre en place une optimisation simple permettant d'améliorer significativement les performances de notre approche. Nous pouvons cependant encore améliorer ces travaux.

Chapitre 10

Conclusion, travaux futurs et perspectives

Nous avons maintenant terminé la présentation de nos travaux. Ces derniers avaient pour objectif l'amélioration des techniques d'analyse automatique de programmes pour l'évaluation des erreurs numériques introduites par l'utilisation de l'arithmétique flottante, et ont été concentrés sur l'amélioration des arithmétiques garanties sur lesquelles sont fondées ces analyses automatiques, en cherchant, dans un premier temps, à tirer parti des liens forts entre les notions d'erreurs absolues et d'erreurs relatives, et, dans un second temps, à améliorer les capacités de raisonnements relationnelles de ces arithmétiques. Nous avons également montré, aux travers de l'évaluation expérimentale de notre implémentation au sein de la plateforme *Frama-C*, que ces travaux permettent d'améliorer la précision des analyses d'une manière significative. Il est désormais temps de poser un regard critique sur ces travaux, afin de mettre en avant leurs limites, ainsi que les prochaines étapes de recherche et les prochaines améliorations qu'il nous faudra entreprendre.

10.1 Travaux futurs

Tout au long du manuscrit, nous avons souligné différents éléments que nous considérons comme important de compléter pour combler les manques de nos travaux. Nous en faisons ici un résumé.

Commençons par les aspects spécifiques au développement de nos travaux. Un premier point évident est que le langage sur lequel nous avons fondé notre étude ne représente qu'un petit fragment du langage C que *Frama-C/Eva* analyse. Pour une partie des constructions absentes de notre langage, ce n'est pas vraiment un problème, car nous profitons des mécanismes de communication d'*Eva* pour profiter des informations inférées par les autres domaines. Cependant, ce processus

n'est que partiellement implémenté actuellement, et en particulier, nous ne disposons que d'un support superficiel des manipulations mémoires. De plus, nous ne pouvons compter sur les mécanismes de réduction ni pour le traitement des fonctions transcendantales, ni pour le support du langage de spécification de *Frama-C* par exemple. Il nous faudra donc travailler à étendre notre langage pour pouvoir également analyser ces fonctions.

Un deuxième point, tout aussi évident, est que cette implémentation est un prototype, fruit de trois ans de travail et ayant subi plusieurs transformations profondes au fur et à mesure de nos avancés et des difficultés rencontrées. Cette évolution de la base de code implique plusieurs défauts qu'il nous faudra corriger. Le premier est qu'actuellement, l'implémentation n'est pas tout à fait, comme nous le soulignions au chapitre 8, conforme à l'architecture que nous avons présentée. Le deuxième est que nous ne sommes que partiellement satisfaits de la qualité du code de notre implémentation, et que nous pensons qu'un gros travail de relecture et d'amélioration sera nécessaire avant d'espérer pouvoir utiliser ce code en production. La troisième est que les performances de notre implémentation ne sont pas encore satisfaisantes. Ce dernier défaut découle partiellement du fait que nous utilisons pour le moment des rationnels, ce que nous pourrons corriger simplement en utilisant des nombres flottants, comme nous l'avons précédemment expliqué. Cependant, ce défaut découle également d'une faiblesse de notre implémentation, bien plus coûteuse à corriger en terme de temps de développement : nous ne disposons d'aucun mécanisme permettant de réduire le nombre de symboles de bruit utilisés. Notre implémentation présente ainsi le défaut de faire croître d'une manière non bornée le facteur déterminant de la complexité des opérateurs abstraits, ce qui pose naturellement des problèmes de passage à l'échelle sur des programmes d'une taille significative.

Enfin, un troisième point est que nous pensons nécessaire d'augmenter le nombre de tests que nous utilisons pour évaluer la qualité de l'implémentation. En effet, en l'état, nos tests ne présentent que de petits programmes pour lesquels l'erreur introduite par l'arithmétique flottante reste relativement faible. Il sera nécessaire de mener une campagne d'évaluation plus poussée à l'avenir, incluant des programmes plus complexes et des exemples présentant de grandes erreurs, afin de pouvoir renforcer la confiance que nous avons dans la qualité de nos résultats. Il sera également nécessaire de comparer notre technique pour les opérations non linéaires avec d'autres approches, en particulier celles utilisées dans le domaine abstrait des polyèdres [55].

Un autre axe d'amélioration de notre implémentation concerne la solution que nous avons mise en place pour pouvoir représenter le mécanisme de gestion des conditions du domaine des zonotopes tout en satisfaisant les contraintes imposées par *Eva*. En effet, bien que cette solution fournisse une réponse permettant de

garantir d'une manière élégante que l'évaluation d'une expression ne peut être effectuée que dans un contexte constant, ce choix d'implémentation pose également quelques problèmes pratiques. Déjà, comme nous l'avons présenté, l'ordre partiel sur ces fonctions n'est pas calculable en pratique. Bien que ce ne soit pas un problème actuellement dans l'implémentation, *Eva* n'utilisant l'ordre partiel de ses valeurs abstraites que pour des heuristiques d'optimisation, cela signifie malgré tout que notre implémentation fonctionne par un heureux coup du sort, ce qui est très peu satisfaisant intellectuellement. De plus, l'utilisation de fonctions implique que tous les calculs soient effectués d'une manière paresseuse, c'est-à-dire au moment où l'analyse a absolument besoin de disposer de la forme affine associée à une variable, et non de manière stricte, c'est-à-dire au moment où l'analyse demande d'effectuer le calcul. Cela complique énormément toutes les étapes de recherches de bugs, d'imprécisions, d'incorrections ou de problèmes de performances. *A posteriori*, nous émettons des doutes sur la viabilité de l'utilisation de cette solution dans une plateforme dédiée à l'analyse de programmes industriels et visant à atteindre une maturité technologique élevée. Nous envisageons donc de chercher une solution plus facile à maintenir, quitte à sacrifier l'élégance qu'apporte notre réponse actuelle.

Pour terminer sur les travaux futurs, présentons ce que nous considérons comme importants à compléter sur le plan théorique. Il nous faudra nous occuper de fournir une preuve de correction pour l'algorithme 1 présenté au chapitre 6, et de définir des opérateurs abstraits pour l'union, l'intersection et l'élargissement à l'abstraction relationnelle de l'erreur relative que nous avons présentée au chapitre 7. Notons qu'il nous faudra également intégrer cette abstraction à notre implémentation.

10.2 Perspectives

Les différents travaux futurs que nous avons présentés jusque là ont tous le même objectif : combler les manques de nos travaux, afin de les terminer d'une manière satisfaisante. Cependant, nous pensons également qu'il reste, au delà, des perspectives de recherche intéressantes à explorer. Nous en présentons ici deux, qui nous semble particulièrement intéressantes.

10.2.1 Conserver les relations à travers le produit réduit

La première perspective serait de travailler à améliorer plus encore le produit réduit entre erreurs absolues et erreurs relatives, en particulier afin de pouvoir prendre en compte au maximum les relations entre les erreurs de différentes variables, ainsi que les relations entre les erreurs d'une même variable. Nos travaux

sont un pas dans cette direction. Cependant, nous pensons qu'il reste encore de la marge de progression.

Une première piste découle du fait que, bien que les formes affines soient bien adaptées à représenter l'erreur absolue, le calcul de cette erreur pour les opérateurs non linéaires reste problématique. Par exemple, l'évaluation de la multiplication fait intervenir trois multiplications, soit trois sources d'incertitudes. Notre produit réduit permet de compenser partiellement cette perte de précision, car l'erreur relative d'une multiplication peut être calculée précisément. Mais nous pensons qu'il pourrait être intéressant, pour le cas de la multiplication, de chercher à définir un opérateur abstrait spécifiquement conçu pour évaluer des expressions de la forme $ax + by + ab$ dans les formes affines. Cela reviendrait à déterminer les cinq coefficients p, q, r, s et t tels que l'hyperplan $px+qy+ra+sb+t$ soit une approximation satisfaisante de l'expression $ax+by+ab$, puis à déterminer l'incertitude introduite par cette approximation. La première étape serait assez simple. En effet, l'utilisation des opérateurs de multiplication et d'addition des formes affines fourniraient, en ignorant les incertitudes introduites, une approximation affine satisfaisante de notre expression :

$$\begin{aligned} ax + by + ab &\sim \alpha_0^a x + \alpha_0^x a - \alpha_0^a \alpha_0^x + \alpha_0^b y + \alpha_0^y x - \alpha_0^b \alpha_0^y + \alpha_0^a b + \alpha_0^b a - \alpha_0^a \alpha_0^b \\ &\sim (\alpha_0^x + \alpha_0^b)a + (\alpha_0^y + \alpha_0^a)b + \alpha_0^a x + \alpha_0^b y - \alpha_0^a \alpha_0^x - \alpha_0^b \alpha_0^y - \alpha_0^a \alpha_0^b \end{aligned}$$

Cela reviendrait à chercher une manière de calculer l'incertitude introduite par l'évaluation de l'expression $ax + by + ab$ non pas comme la somme des trois incertitudes introduites par les sous-expressions ax, by et ab, mais comme un tout. Ceci serait malheureusement bien plus complexe que dans les cas que nous avons présenté au chapitre 6. En effet, l'incertitude serait ici une fonction à quatre paramètres, et non deux :

$$\begin{aligned} \Delta(a, b, x, y) = ax + by + ab - (\alpha_0^x + \alpha_0^b)a - (\alpha_0^y + \alpha_0^a)b - \alpha_0^a x - \alpha_0^b y + \\ \alpha_0^a \alpha_0^x + \alpha_0^b \alpha_0^y + \alpha_0^a \alpha_0^b \end{aligned}$$

Cependant, nous pensons qu'il pourrait être malgré tout intéressant de chercher s'il existe des méthodes d'optimisation applicables à ce type de fonctions, et si oui, si elles seraient suffisamment peu coûteuses pour être utilisable en pratique.

Une seconde piste serait de travailler sur la définition d'un produit réduit entre formes affines et formes relatives permettant de prendre en compte les relations entre les erreurs élémentaires absolues et relatives. Prenons un exemple illustratif. L'erreur relative de l'expression $x \times y$ pourrait être calculée en utilisant la définition 2.20, comme suit :

$$\mathcal{E}_r^\phi(x \times y) = \left(\mathcal{E}_r^\phi(x) + 1\right)\left(\mathcal{E}_r^\phi(y) + 1\right)\left(\Gamma_r^\phi(\widetilde{x}\widetilde{y}) + 1\right) - 1 \tag{10.1}$$

Son erreur absolue pourrait, quant à elle, être calculée en utilisant la définition 2.19, avant d'être potentiellement améliorée par le produit réduit en la calculant comme $xy \times \mathcal{E}_r^\phi(x \times y)$. Cependant, un calcul direct de cette formule ne permettrait pas de mettre en avant de relation entre les erreurs, quand bien même nous aurions, par les définitions 1.5 et 1.6 des erreurs élémentaires, la relation $\Gamma_a^\phi(\widetilde{x}\widetilde{y}) = \widetilde{x}\widetilde{y} \times \Gamma_r^\phi(\widetilde{x}\widetilde{y})$. Si les abstractions utilisées pour les erreurs étaient des intervalles, perdre cette relation ne serait pas un problème. Cependant, si nous utilisions des formes affines pour l'erreur absolue et des formes relatives pour l'erreur relative, alors nous aurions tout intérêt à exploiter cette relation car cela ne pourrait qu'améliorer la précision du produit réduit. Nous pensons donc que définir un produit réduit entre ces deux abstractions permettant de conserver ce type relation représenterait une perspective d'amélioration significative de nos travaux.

10.2.2 Améliorer les opérateurs d'élargissement

Nous avons concentré nos travaux sur l'amélioration des arithmétiques garanties sur lesquelles sont fondés les domaines abstraits permettant d'analyser les erreurs numériques d'un programme. En conséquence, nous n'avons absolument pas contribué à améliorer la gestion des boucles pour ces domaines. Cependant, les méthodes standards ne permettent pas de prendre en compte les spécificités de l'analyse des erreurs numériques, et en particulier du produit réduit que nous avons défini. Une perspective d'évolution est donc de définir un opérateur d'élargissement permettant de prendre en compte le produit réduit, idéalement de façon à également prendre en considération les relations que permettent de représenter l'utilisation des formes affines pour l'erreur absolue et des formes relatives pour l'erreur relative.

Une autre perspective que nous pensons intéressante pour l'analyse des boucles est de travailler à améliorer l'opérateur d'élargissement du domaine des zonotopes, indépendamment de son utilisation pour l'analyse des erreurs numériques, et ce afin de le rendre plus adapté à la recherche d'invariants pour certains types de boucles couramment rencontrées dans les programmes critiques : les filtres linéaires. Intuitivement, un filtre est un outil de traitement du signal permettant d'atténuer l'importance de certaines fréquences. Les filtres sont très utilisés dans toutes les applications manipulant des signaux, car ils permettent d'éliminer tout ou partie de ce qui pourrait perturber le fonctionnement de l'application, comme du bruit perturbant le signal par exemple. Un lecteur désireux d'en apprendre plus peut lire le livre de J. S. Chitode sur le traitement numérique du signal [14].

Le domaine des zonotopes a déjà utilisé pour l'analyse de programmes comportant des filtres. Par exemple, *Fluctuat* a été mis en oeuvre par Airbus dans l'analyse de programmes avioniques comportant des filtres d'ordre deux [24]. Cependant, l'analyse de ces filtres a demandé la mise en place d'une procédure spécifique mé-

langeant déroulages et élargissements. De fait, le temps nécessaire à obtenir un invariant précis dépasse la quarantaine de secondes. La raison principale de ces temps d'analyse considérables vient du fait que l'invariant décrivant au mieux le fonctionnement d'un filtre linéaire d'ordre deux est une ellipsoïde, c'est-à-dire un invariant demandant de pouvoir faire des raisonnements quadratiques. Ce genre de raisonnements n'est pas faisable dans le domaine des zonotopes. Cependant, exactement comme un cercle peut être approximé avec un polygone de plus en plus précisément quand on augmente le nombre de faces de ce dernier, il est possible d'approximer de plus en plus précisément une ellipsoïde avec un zonotope en augmentant le nombre de symboles de bruit. La figure 10.1 en fournit une illustration. L'ellipse rouge est approximée par deux zonotopes : celui de gauche utilisant deux symboles de bruit, tandis que celui de droite en utilise trois. Nous voyons clairement que l'approximation de droite est plus précise.

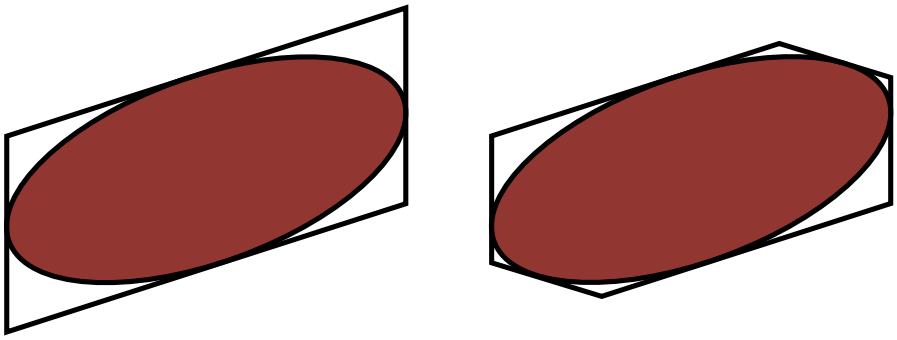


FIGURE 10.1 – Approximations d'une ellipse par des zonotopes.

Dans le cas de l'analyse de filtres d'ordre deux, il est malheureusement nécessaire de disposer d'une approximation très précise, utilisant donc beaucoup de symboles de bruit. En effet, intuitivement, une itération de la boucle implémentant le filtre a deux effets sur l'état abstrait : elle lui fait subir une rotation et en réduit la taille. La figure 10.2 présente une illustration de ce phénomène : l'ellipse verte correspond à la transformation de la grande ellipse blanche après une itération. Nous voyons que l'ellipse verte est incluse dans la grande ellipse blanche, signifiant que nous avons convergé et que l'ellipse blanche est bien une sur-approximation de l'invariant décrivant le fonctionnement du filtre. Cependant, nous ne pouvons pas le prouver avec les zonotopes. En effet, malgré la rotation et la réduction de taille, l'approximation par un zonotope de l'ellipse verte présente deux pointes, illustrées en rouge, qui ne sont pas incluses dans l'approximation de la grande ellipse blanche. Il nous faudrait ici disposer d'une approximation avec plus de symboles de bruit.

La solution standard à ce problème est de considérer que, comme nous cherchons à décrire un invariant quadratique, autant construire un domaine abstrait permettant de représenter spécifiquement ce genre d'objets, c'est-à-dire un do-

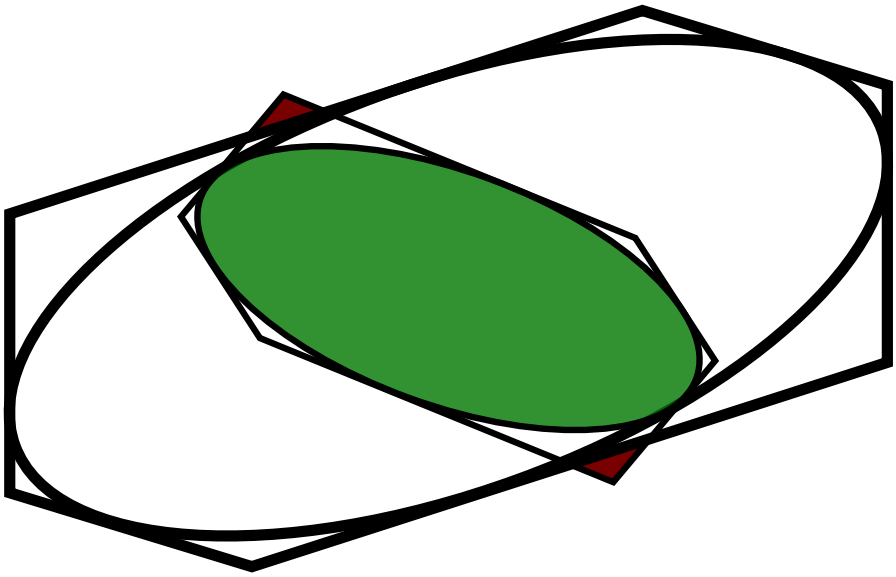


FIGURE 10.2 – Le problème de l'inclusion après une itération. L'invariant est inclue dans l'ellipse de départ, signifiant la convergence de l'analyse, mais l'approximation utilisant un zonotope ne permet pas de le prouver.

maine des ellipsoïdes [31]. Cependant, en pratique, l'utilisation de ce domaine nécessite de fournir en avance certains paramètres afin d'adapter l'analyse à chaque filtre considéré. Cela signifie qu'il nous faut sacrifier le caractère automatique de l'analyse. Cependant, nous pensons qu'il serait possible de faire mieux, en créant un domaine présentant les forces du domaine des zonotopes pour l'analyse de programmes généraux, et celles du domaine des ellipsoïdes pour la description des invariants quadratiques.

Cette perspective découle d'un constat assez simple : zonotopes et ellipsoïdes sont des objets mathématiques assez proches, et les deux peuvent être décrits comme l'application d'une transformation affine à une hypersphère de rayon égal à 1. La seule différence est la norme considérée pour cette hypersphère : norme deux pour les ellipsoïdes et norme infinie pour les zonotopes. Une illustration de ce point est représentée à la figure 10.3 : la même transformation affine y est appliquée à une hypersphère en norme infinie sur la première ligne, produisant un zonotope, et à une hypersphère en norme deux sur la seconde, produisant une ellipse.

Nous pensons que ceci pourrait être exploité afin de définir un domaine abstrait utilisant des zonotopes dont nous pourrions, intuitivement, arrondir les angles afin de représenter les invariants quadratiques précisément. Pour ce faire, une piste qu'il nous semble pertinent d'explorer serait d'utiliser une abstraction permettant de représenter des contraintes quadratiques pour les symboles de bruit, et de définir un opérateur d'élargissements permettant d'introduire ces contraintes. Cela représen-

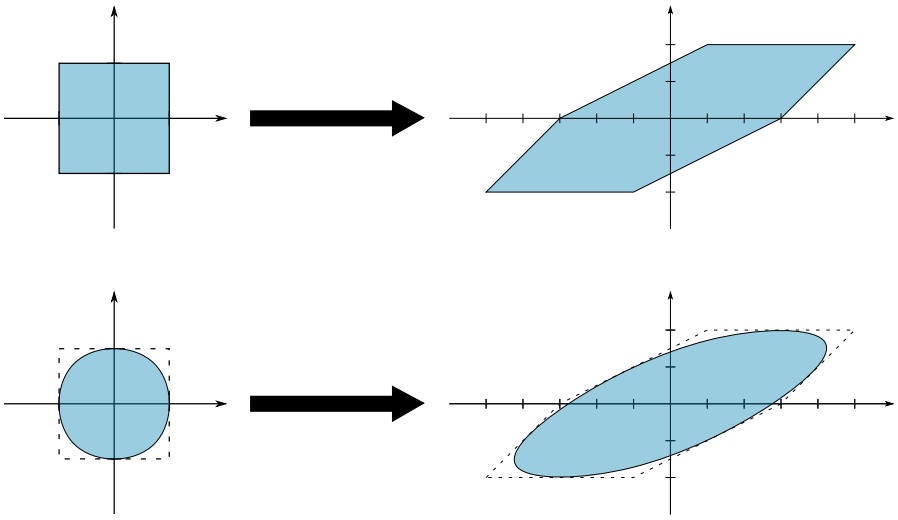


FIGURE 10.3 – Une ellipse et un zonotope construits par la même transformation affine appliquée à des hypersphères dans des normes différentes.

terait l'immense avantage de n'impacter que très peu la manière dont sont évaluées les expressions : nous n'aurions en effet qu'à réutiliser les méthodes décrites dans ce manuscrit car elles resteraient valides indépendamment de l'abstraction utilisée pour les symboles de bruit, à partir du moment où il serait possible de la surapproximer par des intervalles. De la même manière, nous devrions pouvoir réutiliser les opérateur abstraits de l'union et de l'intersection, même s'il serait sans doute plus intéressant d'en définir de nouveaux prenant en compte les contraintes quadratiques. La réelle difficulté est de déterminer comment évaluer l'ordre partiel avec une telle abstraction pour les symboles de bruit, et comment introduire les contraintes quadratiques d'une manière à la fois précise et efficace.

Bibliographie

[1] Sagemath official website. `https://www.sagemath.org`.

[2] IEEE 754. Standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, 22(2), 1985.

[3] S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum. *Handbook of logic in computer science*, volume 3. Oxford University Press, 1995.

[4] P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye. *WP Plug-in Manual*. `http://frama-c.com/download/frama-c-wp-manual.pdf`.

[5] P. Baudin, J. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language*. `http://frama-c.com/acsl.html`.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[7] M. Blair, S. Obenski, and P. Bridickas. Patriot missile defense : Software problem led to system failure at dhahran, saudi arabia. Technical Report of the U.S. General Accounting Office GAO/IMTEC-92-26, United States of America, General Accounting Office, 1992.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, 2002.

[9] S. Blazy, D. Bühler, and B. Yakobowski. Structuring abstract interpreters through state and value abstractions. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, volume 10145 of *Lecture Notes in Computer Science*, 2017.

[10] S. Boldo. Kahan's algorithm for a correct discriminant computation at last formally proven. *IEEE Transactions on Computers*, 58, 2009.

[11] S. Boldo and G. Melquiond. Flocq : A unified library for proving floating-point algorithms in coq. In *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, 2011.

[12] A. Brahmi, M. Carolus, D. Delmas, M. H. Essoussi, P. Lacabanne, V. M. Lamiel, F. Randimbivololona, and J. Sourys. Industrial use of a safe and efficient formal method based software engineering process in avionics. *ERTS*, 2020.

[13] D. Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions : EVA, an Evolved Value Analysis for Frama-C*. PhD thesis, Université Rennes 1, 2017.

[14] J. S. Chitode. *Digital Signal Processing*. Technical Publications ; 1st edition, 2006.

[15] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. *SEBGRAPI'93*, 1993.

[16] L. Correnson and J. Signoles. Combining analyses for C program verification. In *Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*, volume 7437 of *Lecture Notes in Computer Science*. Springer, 2012.

[17] C. Courbet. Nsan : a floating-point numerical sanitizer. In *CC '21 : 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021*. ACM, 2021.

[18] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, 1977.

[19] P. Cousot and R. Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software (LDRS), Raleigh, North Carolina, USA, March 28-30, 1977*. ACM, 1977.

[20] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, 1978.

[21] S. Boldo D. Gallois-Wong and P. Cuoq. Optimal Inverse Projection of Floating-Point Addition. *Numerical Algorithms*, 83, 2020.

[22] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian. Daisy - framework for analysis and optimization of numerical programs (tool paper). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, 2018.

[23] S. de Oliveira. *Finding constancy in linear routines.* PhD thesis, Université Paris-Saclay, 2018.

[24] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*, volume 5825 of *Lecture Notes in Computer Science*, 2009.

[25] N. Demeure. Gestion du compromis entre la performance et la précision de code de calcul, 2021.

[26] C. Denis, P. de Oliveira Castro, and E. Petit. Verificarlo : Checking floating point accuracy through monte carlo arithmetic. In *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, 2016.

[27] E. W. Dijkstra. Notes on structured programming. Technical Report EWD-249 (70-WSK-03), University of Technology, Eindhoven, Pays-Bas, 1969.

[28] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975.

[29] P. Baufreton E. Goubault, S. Putot and J. Gassino. Static analysis of the accuracy in control systems : Principles and experiments. In *Formal Methods for Industrial Critical Systems, 12th International Workshop, FMICS 2007, Berlin, Germany, July 1-2, 2007, Revised Selected Papers*, volume 4916 of *Lecture Notes in Computer Science*, 2007.

[30] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. *Automata, Languages and Programming*, 85/1980, 1980.

[31] J. Feret. Static analysis of digital filters. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as*

Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, volume 2986 of *Lecture Notes in Computer Science*, 2004.

[32] F. Févotte and B. Lathuilière. Studying the numerical quality of an industrial computing code : A case study on code_aster. In *Numerical Software Verification - 10th International Workshop, NSV 2017, Heidelberg, Germany, July 22-23, 2017, Proceedings [collocated with CAV 2017]*, volume 10381 of *Lecture Notes in Computer Science*, 2017.

[33] J. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*. Springer, 2013.

[34] IEEE 754 Standard for Binary Floating-Point Arithmetic. 2008 revision, http ://ieeexplore.ieee.org/servlet/opac ?punumber=4610933, 2008.

[35] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. Mpfr : A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw*, 33, 2007.

[36] K. Ghorbal. *Static Analysis of Numerical Programs : Constrained Affine Sets Abstract Domain. (Analyse Statique de Programmes Numériques : Ensembles Affines Contraints)*. PhD thesis, École Polytechnique, Palaiseau, France, 2011.

[37] A. Gilat. *MATLAB : An Introduction With Applications*. Wiley, 2008.

[38] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*.

[39] E. Goubault and S. Putot. Perturbed affine arithmetic for invariant computation in numerical program analysis. *CoRR*, abs/0807.2961, 2008.

[40] E. Goubault and S. Putot. Static analysis of finite precision computations. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, 2011.

[41] E. Goubault and S. Putot. Robustness analysis of finite precision implementations. In *APLAS*, 2013.

[42] E. Goubault and S. Putot. A zonotopic framework for functional abstractions. *Formal Methods Syst. Des.*, 47, 2015.

[43] P. Herrmann and J. Signoles. *Annotation Generation : Frama-C's RTE pplugin*. `http://frama-c.com/download/frama-c-rte-manual.pdf`.

[44] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12, 1969.

[45] A. Izycheva and E. Darulova. On sound relative error bounds for floating-point arithmetic. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, FMCAD '17, 2017.

[46] F. Jézéquel and J. M. Chesneaux. CADNA : a library for estimating round-off error propagation. *Comput. Phys. Commun.*, 178, 2008.

[47] E. Goubault K. Ghorbal and S. Putot. A logical product approach to zonotope intersection. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, 2010.

[48] B. Kabi, E. Goubault, A. Miné, and S. Putot. Combining zonotope abstraction and constraint programming for synthesizing inductive invariants. In *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*, volume 12549 of *Lecture Notes in Computer Science*. Springer, 2020.

[49] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c : A software analysis perspective. *Formal Aspects Comput.*, 27, 2015.

[50] M. G. Krein and M. A. Rutman. Linear operators leaving invariant a cone in a banach space. *Uspekhi Mat. Nauk*, 3, 1948.

[51] O. Kupriianova and C. Quirin Lauter. Metalibm : A mathematical functions code generator. In *Mathematical Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*, volume 8592 of *Lecture Notes in Computer Science*, 2014.

[52] L. Lakhdar-Chaouch, B. Jeannet, and A. Girault. Widening with thresholds for programs with complex control graphs. In *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*, Lecture Notes in Computer Science, 2011.

[53] C. Lattner and V. S. Adve. LLVM : A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO) 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004.

[54] S. Putot M. Jacquemin and F. Védrine. A reduced product of absolute and relative error bounds for floating-point analysis. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, 2018.

[55] A. Maréchal and M. Périn. Three linearization techniques for multivariate polynomials in static analysis using convex polyhedra. Verimag research report, Verimag, Université Grenoble Alpes, 2014.

[56] M. Martel. Static analysis of the numerical stability of loops. In *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*, volume 2477 of *Lecture Notes in Computer Science*. Springer, 2002.

[57] R. Mayans. The chebyshev equioscillation theorem. *Journal of Online Mathematics and its Applications*, 2006.

[58] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, École Normale Supérieure de Lyon, 2006.

[59] F. Messine. Extentions of affine arithmetic : Application to unconstrained global optimization. *J. Univers. Comput. Sci.*, 8, 2002.

[60] A. Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages (FnTPL)*, 4, 2017.

[61] A. Miné, A. Ouadjaout, and M. Journault. Design of a modular platform for static analysis. In *Proc. of 9h Workshop on Tools for Automatic Program Analysis (TAPAS'18)*, Lecture Notes in Computer Science (LNCS), 2018.

[62] S. Miyajima and M. Kashiwagi. A dividing method utilizing the best multiplication in affine arithmetic. *IEICE Electron. Express*, 1, 2004.

[63] R. E. Moore. *Interval Analysis*. 1966.

[64] R. E. Moore, R. Baker Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. 2009.

[65] J. Muller, N. Brunie, F. de Dinechin, C. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres. *Handbook of Floating-Point Arithmetic, 2nd edition*. Birkhäuser Boston, 2018.

[66] N. Nethercote and J. Seward. Valgrind : A framework for heavyweight dynamic binary instrumentation. In *In Proceedings of the 2007 Programming Language Design and Implementation Conference*, 2007.

[67] A. M. Pitts, G. Winskel, and M. Fiore. *Lecture Notes on Denotational Semantics*. Cambridge University Computer Laboratory, 2020.

[68] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN19, Aarhus University, 1981.

[69] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, volume 137, 1982.

[70] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2), 1953.

[71] X. Rival and K. Yi. *Introduction to Static Analysis : An Abstract Interpretation Perspective*. 2020.

[72] W. Rudin. *Functional Analysis*. McGraw-Hill, 1991.

[73] S. Rump. Error-free transformations and ill-conditioned problems. 01 2009.

[74] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part i : Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1), 2008.

[75] T. Miyata S. Miyajima and M. Kashiwagi. On the best multiplication of the affine arithmetic. *IEICE Trans. Fundamentals*, 2003.

[76] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 2018.

[77] D. S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. *Proceedings Symposium on Computers and Automata*, 21, 1971.

[78] J. Signoles, L. Correnson, M. Lemerre, and V. Prevosto. *Frama-C Plug-in Development Guide*. `http://frama-c.com/download/plugin-developer.pdf`.

[79] J. Signoles, N. Kosmatov, and K. Vorobyov. E-acsl, a runtime verification tool for safety and security of C programs (tool paper). In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, volume 3 of *Kalpa Publications in Computing*, 2017.

[80] I. Skalna and M. Hladík. A new algorithm for chebyshev minimum-error multiplication of reduced affine forms. *Numerical Algorithms*, 2017.

[81] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Trans. Program. Lang. Syst.*, 41, 2019.

[82] J. Stewart. *Multivariate Calculus*. 2004.

[83] L. Titolo, M. A. Feliú, M. M. Moscato, and C. A. Muñoz. An abstract interpretation framework for the round-off error analysis of floating-point programs.

In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer Science*, 2018.

[84] L. Titolo, C. A. Muñoz, M. A. Feliú, and M. M. Moscato. Eliminating unstable tests in floating-point programs. In *Logic-Based Program Synthesis and Transformation - 28th International Symposium, LOPSTR 2018, Frankfurt/Main, Germany, September 4-6, 2018, Revised Selected Papers*, volume 11408 of *Lecture Notes in Computer Science*, 2018.

[85] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematic Society*, 2, 1936.

[86] P. Uewichitrapochana and A. Surarerks. Signed-symmetric function approximation in affine arithmetic. *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2013.

[87] F. Védrine, M. Jacquemin, N. Kosmatov, and J. Signoles. Runtime abstract interpretation for numerical accuracy and robustness. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*. Springer, 2021.

[88] F. Védrine, M. Jacquemin, N. Kosmatov, and J. Signoles. Runtime abstract interpretation for numerical accuracy and robustness. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, 2021.

[89] G. Winskel. *The formal semantics of programming languages : an introduction*. MIT Press, Cambridge, MA, États-Unis, 1993.

[90] Kévin Youyou. Interprétation abstraite relationnelle de précision numérique dédiée aux erreurs relatives. Rapport de stage de master, CEA List, 2020.

[91] D. Zou, M. Zeng, Y. Xiong, Z. Fu, L. Zhang, and Z. Su. Detecting floating-point errors via atomic conditions. *Proc. ACM Program. Lang.*, 4, 2020.

université PARIS-SACLAY | ÉCOLE DOCTORALE Sciences et technologies de l'information et de la communication (STIC)

Titre : Arithmétiques relationnelles pour l'analyse par interprétation abstraite de propriétés de précision numérique

Mots clés : Arithmétique à virgule flottante, Diagnostic de précision, Vérification de logiciels

Résumé : L'arithmétique à virgule flottante est l'approche la plus utilisée pour réaliser des calculs mathématiques reposant sur les nombres réels avec un ordinateur. Cependant, elle souffre d'un défaut : chaque opération peut introduire une erreur, c'est-à-dire une différence avec le résultat que nous aurions obtenu en utilisant des réels. Bien que ces erreurs soient très faibles, elles peuvent s'accumuler et provoquer des bugs ayant parfois des conséquences graves, en particulier dans des domaines critiques comme l'aéronautique ou le nucléaire. Il est donc nécessaire de pouvoir garantir que les erreurs introduites par l'utilisation de l'arithmétique flottante ne causent pas de problème, ou, dit autrement, qu'elles soient suffisamment faibles pour que le programme se comporte comme attendu.

Pour répondre à ce besoin, nous proposons une analyse statique par interprétation abstraite, reposant sur un nouveau domaine abstrait, et calculant une surapproximation des erreurs introduites par l'arithmétique flottante. Cette analyse repose sur l'interaction, au travers d'un produit réduit, entre deux conceptions de la notion d'erreur : l'erreur absolue, intuitive et permettant de mieux comprendre le programme analysé, et l'erreur relative, plus proche du fonctionnement de l'arithmétique flottante.

Notre analyse repose sur la combinaison des arithmétiques affines et d'intervalles, et dispose de capacités de raisonnements relationnels. Cette combinaison a cependant des difficultés à traiter les opérations non linéaires, dont la précision impacte fortement l'évaluation des erreurs relatives. Nous proposons donc deux approches pour répondre à ce problème. La première consiste en différentes améliorations de cette combinaison, permettant d'évaluer plus précisément multiplications et divisions sans impacter significativement les performances. La seconde consiste en la définition d'une nouvelle arithmétique relationnelle, spécifiquement conçue pour représenter l'erreur relative.

Par ailleurs, nous avons implémenté un prototype de notre analyse au sein de l'outil Frama-C/Eva. Les premiers résultats expérimentaux montrent l'intérêt de notre analyse par rapport à l'état de l'art.

Title : Relational Arithmetics for Abstract Interpretation Based Analysis of Numerical Accuracy Properties

Keywords : Floating point arithmetic, Accuracy diagnosis, Software verification

Abstract : Floating point arithmetic is the most used approach to perform mathematical computations using real numbers with a computer. However, this approach has a default : each operation can introduce an error, that is, a difference with the result we would have obtained using real numbers. Even if those errors are very small, they can accumulate et provoke serious bugs, particularly in critical domains like aeronautics or nuclear energy production for example. Thus, we have to be able to guarantee that the errors introduced by the use of floating point arithmetic do not cause problems, in other words, that they are small enough for the program to behave as expected.

To answer this need, we propose an abstract interpretation based static analysis, along with a new abstract domain, that computes an overapproximation of the errors introduced by floating point arithmetic. This analysis is based on the interaction, performed through a reduced product, between two conceptions of the concept of error : absolute error, intuitive and helpful to understand the analyzed program, and relative error, closer of floating point arithmetic functioning.

Our analysis relies on the combination of affine and intervals arithmetics, and thus have relational reasoning capacities. However, this combination has difficulties dealing with non linear operations, whose precision has a huge impact on relative errors evaluations. Thus, we propose two approaches to tackle this problem. The first one consists of several improvements of this combination that help evaluating multiplications and divisions more precisely without impacting performances significantly. The second one consists of the definition of a new relational arithmetic, specifically designed to represent relative errors.

Besides, we have implemented a prototype of our analysis within the Frama-C/Eva tool. The first experimental results enlighten the advantages of our analysis against state of the art tools.

Maison du doctorat de l'Université Paris-Saclay
2e étage de l'aile ouest, École normal supérieure Paris-Saclay
4 avenue des Sciences,
91190 Gif sur Yvette, France