UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES ET DES TECHNIQUES

—

Année 2000

N° attribué par la bibliothèque

Langages et environnements en programmation par contraintes d'intervalles

THÈSE DE DOCTORAT

ÉCOLE DOCTORALE : SCIENCES POUR L'INGÉNIEUR DE NANTES
Spécialité : informatique

présentée et soutenue publiquement par

Frédéric GOUALARD

le 12 juillet 2000
à L'Institut de Recherche en Informatique de Nantes
devant le jury ci-dessous

Président	Pr. Alain COLMERAUER
Rapporteurs	Philippe CODOGNET, professeur • Université de Paris 6
	Pascal VAN HENTENRYCK, professeur • Université catholique de Louvain, Belgique
Examinateurs	Frédéric BENHAMOU, professeur • Université de Nantes
	Alain COLMERAUER, professeur • Université de la Méditerranée, membre de l'Institut Universitaire de France
	Gérard FERRAND, professeur • Université d'Orléans
	Jean-François PUGET, directeur technologies • Ilog S.A.

Directeur de thèse : Pr. Frédéric BENHAMOU

INSTITUT DE RECHERCHE EN INFORMATIQUE DE NANTES
2, rue de la Houssinière • F-44322 NANTES CEDEX 3

N° ED 82-461

LANGAGES ET ENVIRONNEMENTS EN PROGRAMMATION PAR CONTRAINTES D'INTERVALLES

Languages and Environments for Interval Constraint Programming

Frédéric GOUALARD

⋈

favet neptunus eunti

Université de Nantes

Frédéric GOUALARD

Langages et environnements en programmation par contraintes d'intervalles

xxiv+228 p.

Ce document a été préparé avec LaTeX2$_\varepsilon$ et la classe `theseIRIN` version 1.0 écrite par Frédéric GOUALARD, IRIN, Université de Nantes, (`Frederic.Goualard@irin.univ-nantes.fr`). La classe `theseIRIN` est disponible à l'adresse :

`http://www.sciences.univ-nantes.fr/info/perso/permanents/goualard/`

Impression : these-goualard.tex – 1/10/2000 – 19:17

Révision pour la classe : $Id: theseIRIN.cls,v 1.37 2000/10/01 10:08:11 fred Exp

“Here is the man who has made a certain machine,” said the Emperor, “and yet asks us what he has created. He does not know himself. It is only necessary that he create, without knowing why he has done so, or what this thing will do.”

— Ray BRADBURY, The flying machine.

Résumé

La programmation par contraintes d'intervalles est une approche prometteuse pour la résolution de systèmes de contraintes réelles non-linéaires : l'emploi de l'arithmétique d'intervalles garantit la complétude des résultats et l'efficacité des méthodes de filtrage par le calcul de consistances partielles est identique ou (très) supérieure à celle d'algorithmes spécialisés.
Dans cette thèse, nous nous intéressons à l'extension des capacités de la programmation par contraintes d'intervalles suivant trois axes :

1. *Accélération du processus de calcul :* les librairies d'intervalles utilisées par les algorithmes de résolution doivent être à la fois correctes et efficaces. Nous décrivons une librairie C++ paramétrée de calcul sur les intervalles basée sur la notion de *trait* et montrons que la flexibilité obtenue par la paramétrisation du type des bornes n'induit pas de surcoût à l'exécution et autorise une plus grande fiabilité pour la portabilité de la librairie. Nous présentons aussi une extension de la définition de box-consistance autorisant la mise au point d'un algorithme pour son calcul dont l'efficacité est toujours au moins égale à celle des algorithmes utilisés habituellement pour calculer la box-consistance ou la hull-consistance ;
2. *Extension du domaine d'application :* l'emploi des intervalles garantit la complétude des résultats mais pas leur correction alors que cette propriété est cruciale pour certaines applications. Nous décrivons des algorithmes de calcul d'approximations intérieures de relations réelles assurant cette correction ; nous introduisons ensuite des algorithmes autorisant la résolution de systèmes avec des variables quantifiées universellement ;
3. *Définition d'un environnement de programmation adapté :* nous présentons une méthode d'abstraction du store de contraintes permettant sa visualisation à des fins de débogage/optimisation/explication de programmes contenant des contraintes, puis décrivons un outil basé sur cette technique.

Mots-clés : programmation par contraintes, variable quantifiée, positionnement de caméra, débogage, arithmétique d'intervalles, consistance locale

Abstract

Interval constraint programming is a promising approach for solving non-linear real constraint systems: interval arithmetics guarantees the completeness of the results while the efficiency of the filtering methods by partial consistency computation is identical or (very) superior to the one of tailored algorithms.
In this thesis, we focus on the extension of interval constraint programming capabilities through three axis:

1. *Acceleration of the computation process:* interval libraries used by the solving algorithms need be both correct and efficient. We describe a parameterized C++ library based on the traits notion, and we show that the flexibility achieved through parameterization of the type of the bounds does not induce any additional cost on execution, while providing enhanced reliability as for the library portability. We also present an extension of the box consistency definition that allows us devising an algorithm to compute it whose efficiency is always at least equal to the one of the algorithms usually used to compute box consistency or hull consistency ;
2. *Extension of the applicability domain:* the use of intervals ensures completeness but not the correctness of the results, while this last property is crucial for some applications. We describe algorithms to compute inner approximations of real relations ensuring correctness ; we then introduce algorithms that permit solving systems with universally quantified variables ;
3. *Definition of a suitable programming environment:* we present a method for providing an abstraction of the constraint store that allows visualizing it for debugging/optimizing/explaining purposes for constraint programs ; we then describe a tool based on this technique.

Keywords: constraint programming, quantified variable, camera control, debugging, interval arithmetic, local consistency

Classification ACM

Catégories et descripteurs de sujets : D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic*; G.1.0 [**Numerical Analysis**]: General—*Interval arithmetic*; G.1.5 [**Numerical Analysis**]: Roots of Nonlinear Equations—*systems of equations*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems—*Animations*

Termes généraux : Algorithms, Theory, Reliability

Remerciements

Que les professeurs CODOGNET, COLMERAUER, FERRAND et VAN HENTENRYCK, ainsi que M. PUGET, trouvent ici l'expression de ma gratitude pour avoir accepté de faire partie de mon jury de thèse. Je remercie tout spécialement le professeur COLMERAUER pour avoir assumé la responsabilité de président du jury, ainsi que les professeurs CODOGNET et VAN HENTENRYCK qui ont aimablement accepté d'être les rapporteurs de cette thèse. Ils en ont lu avec soin le manuscrit, faisant des remarques fort justes dont j'espère avoir tenu compte le mieux possible.

Après avoir encadré mon stage de DEA à Orléans, le professeur BENHAMOU a été mon directeur de thèse pendant quatre ans. Durant toutes ces années, il a fait preuve d'un intérêt constant à mon égard, m'encourageant à conjuguer travaux théoriques et mise en pratique. J'ai pu être le témoin de la rigueur dont il témoigne lors de la rédaction d'articles scientifiques et souhaite avoir à l'avenir la même appréciation objective et sans concession de mon propre travail que celle dont il faisait preuve lorsque nous rédigions des articles ensemble.

Une partie importante des travaux présentés dans cette thèse a été réalisée en collaboration avec Laurent GRANVILLIERS. En conséquence, de nombreux résultats présentés ici sont aussi les siens. Durant toute ma thèse, Laurent m'a offert son amitié ainsi que l'exemple d'un chercheur infatigable toujours prêt à tester de nouvelles idées, à s'investir dans de nouvelles collaborations. Écrire des articles avec lui a été une expérience enrichissante et un plaisir.

Les travaux concernant la gestion de variables quantifiées ont été motivés par les recherches menées au sein de l'équipe « Images » de l'IRIN. En particulier, Éric LANGUÉNOU et Marc CHRISTIE ont très largement contribué aux résultats obtenus, tant sur le plan théorique que pratique. L'implémentation et les tests des algorithmes présentés dans cette thèse ont été faits par Marc CHRISTIE, que je remercie aussi pour avoir préparé les images qui illustrent le chapitre 12.

Jorge CRUZ, Evgueni PETROV, Marc CHRISTIE et Martine CEBERIO ont eu la tâche ingrate de tester certaines des applications développées durant cette thèse. Tous m'ont fait part de nombreuses suggestions d'amélioration, me signalant les bogues dont elles étaient affligées et endurant les conséquences des diverses mises à jour successives.

Les travaux présentés dans cette thèse ont été faits en partie dans le cadre des projets ESPRIT DiSCiPl n° 22532 et LOCO. Je remercie les autres participants à ces projets avec qui j'ai pu avoir à un moment ou à un autre des échanges enrichissants.

Sommaire

Introduction ..... **xi**
Notice ..... **xv**

I L'arithmétique des intervalles

Introduction **3**
1 L'arithmétique flottante **5**
2 L'arithmétique d'intervalles **15**

II Consistances locales

Introduction **33**
3 Consistances locales **37**
4 Extension de la définition de box-consistance **49**

III Approximations intérieures et variables quantifiées

Introduction **59**
5 Gestion de quantificateurs et calcul d'approximations intérieures **63**
6 Approximations intérieures, variables quantifiées et contraintes d'intervalles **77**

IV Environnements de programmation en PC

Introduction **89**
7 État de l'art des langages de PCI **93**
8 DecLIC *(a Declarative Language with Interval Constraints)* **97**
9 Le débogage en programmation par contraintes : état de l'art **129**
10 Débogage par observation du *store* **135**

V Programmation par contraintes et programmation orientée objets

Introduction **153**
11 Implémentation de l'arithmétique d'intervalles : JAIL **155**
12 Gestion de variables quantifiées : OpAC et WOpAC **169**

Conclusion et perspectives **179**

Bibliographie **183**
Liste des tableaux **197**
Table des figures **199**
Table des exemples **203**
Table des matières **205**
Index **209**
A Annexe à la partie II **223**
B Annexe à la partie III **225**

Introduction

LA RÉSOLUTION DE SYSTÈMES d'équations/inéquations et plus généralement de contraintes non-linéaires réelles par des moyens informatiques est un problème difficile : les méthodes symboliques [88] s'avèrent souvent trop lentes et/ou trop gourmandes en ressources ; quant aux méthodes numériques, l'utilisation des nombres en virgule flottante [116] induit des erreurs de calcul rendant sujet à caution les résultats produits [126, 125, 86]. Se pose aussi le problème de la convergence des algorithmes itératifs tels la méthode de NEWTON-RAPHSON.

Durant les années cinquante, les travaux de MOORE [167] ont mis en évidence l'intérêt de l'*arithmétique des intervalles* pour résoudre le problème de correction des calculs induit par l'emploi d'un sous-ensemble réduit des réels : dans ce formalisme, chaque nombre réel non représentable en machine est manipulé sous la forme d'un intervalle le contenant dont les bornes sont des nombres flottants. De plus, les opérations arithmétiques usuelles sont étendues afin de toujours garantir l'inclusion des résultats réels dans les intervalles calculés. Depuis lors, de nombreux algorithmes numériques classiques (GAUSS-SEIDEL, NEWTON-RAPHSON… [167, 175, 36]) ont été adaptés au cadre de l'arithmétique des intervalles et utilisés avec succès pour calculer les solutions de systèmes réels linéaires ou non-linéaires. Les algorithmes sur les intervalles jouissent de propriétés importantes : ils assurent la complétude des résultats (pas de solution perdue) ; la convergence des algorithmes itératifs est garantie ; et il est possible de prouver l'existence, voire l'unicité, d'une solution dans les pavés calculés. Le recours aux intervalles permet aussi de manipuler des données imparfaitement connues (*e.g.* du fait de l'imprécision des mesures) et de raisonner sur des ensembles de valeurs, ce qui s'avère crucial pour faire de l'optimisation globale [100], par exemple.

Toutes ces qualités ont conduit CLEARY [48] à utiliser l'arithmétique des intervalles pour intégrer dans Prolog une arithmétique relationnelle conforme à la philosophie de ce langage, ce qui apparaît aujourd'hui comme l'un des premiers pas vers la résolution de contraintes réelles par des méthodes d'intervalles. Une autre étape a été franchie lorsque DAVIS [61] s'est avisé d'adapter aux domaines continus les méthodes de résolution de *problèmes de satisfaction de contraintes* (CSPs) sur les domaines discrets utilisées en Intelligence Artificielle : depuis les travaux de WALTZ [245], la résolution de CSPs discrets — problème NP-difficile — se fait de façon approchée par des techniques de filtrage telles que la *consistance d'arc* [152], où, pour chaque contrainte $c(x, y)$, on élimine du domaine de la variable x (resp. y) l'ensemble des valeurs n'ayant pas de support dans le domaine de y (resp. x).

Par la suite, LHOMME [150] et BENHAMOU & OLDER [30] ont affaibli la notion de consistance d'arc [152] pour l'adapter au cas de domaines continus modélisés par des intervalles, obtenant ainsi, respectivement, la *2B-consistance* et la *hull-consistance*, où l'on ne considère plus que les bornes des domaines pour la recherche des valeurs incohérentes. En pratique, le calcul de la hull-consistance requiert une décomposition des contraintes de l'utilisateur en une conjonction de contraintes « primitives ». L'augmentation du nombre de variables induit par cette décomposition limite les capacités de réduction des domaines. Afin de pallier cet inconvénient, BENHAMOU *et al.* [29] ont défini une nouvelle notion de consistance locale, la *box-consistance*, et un algorithme pour la calculer capable de traiter les contraintes sans les décomposer.

Les méthodes de résolution de contraintes réelles basées sur des algorithmes de filtrage par détermination de consistances locales et sur l'arithmétique des intervalles ont montré leur compétitivité par rapport aux algorithmes numériques standards (*cf.* [196], par exemple). On peut cependant identifier un certain nombre de manques et de faiblesses réduisant l'efficacité ou le champ d'application des outils implémentant les techniques de résolution de contraintes d'intervalles :

– les algorithmes de calcul de la hull-consistance et de la box-consistance sont très différents. Ainsi, suivant la forme des contraintes, il est plus avantageux de considérer soit la hull-consistance, soit la box-consistance, car, comme on le verra dans la suite, aucun des algorithmes associés ne s'avère le meilleur choix dans tous les cas. Une première approche — peu satisfaisante — pour pallier ce problème, consisterait à déterminer précisément les classes de contraintes pour lesquelles chacun des algorithmes est optimal. Idéalement, on souhaiterait cependant pouvoir considérer une seule consistance intermédiaire entre la hull-consistance et la

box-consistance dont l'algorithme de calcul serait, pour chaque contrainte, au moins aussi efficace que le meilleur des deux algorithmes associés à ces consistances ;
- les nombreux résolveurs de contraintes d'intervalles actuellement disponibles (clp(BNR) [12, 30, 176], Numerica [235], Prolog IV [32, 22], clp(F) [103, 104], CLIP [105], Newton [234], ICLP($\mathcal{R}$) [147]...) sont des outils efficaces qui garantissent la complétude des résultats grâce au recours à l'arithmétique d'intervalles. Par contre, du fait de la conservativité de cette arithmétique, la correction n'est jamais assurée (des valeurs du résultat peuvent ne pas être solution du système de contraintes initial). Or, cette propriété est déterminante pour de nombreuses applications, que ce soit en simulation de systèmes complexes [16] ou en théorie du contrôle [80], par exemple. Le nombre et l'importance d'un point de vue industriel de ces applications rendent indispensable la définition de nouvelles méthodes de résolution de contraintes d'intervalles garantissant la correction des résultats ;
- enfin, les méthodes de résolution de contraintes d'intervalles sont limitées aux cas de systèmes formés par la conjonction de formules atomiques sans quantification des variables. En toute généralité, on souhaiterait pouvoir résoudre des systèmes de forme quelconque (formules générales du premier ordre avec variables quantifiées). Plus raisonnablement, il semble important d'offrir au moins la possibilité de résoudre des systèmes formés de conjonctions de contraintes où apparaissent des variables universellement quantifiées, tels que ceux obtenus en robotique (problèmes de collision [111]) ou en contrôle et positionnement de caméra [71, 3].

Contributions

Durant notre thèse, nous nous sommes attachés à tenter d'apporter des réponses aux insuffisances décrites ci-dessus :
- nous avons constaté que le nombre d'occurrences des variables dans une contrainte est déterminant pour le choix de l'algorithme de résolution (et partant de la consistance) à utiliser : lorsqu'une variable possède de nombreuses occurrences, l'algorithme BC3 de calcul de la box-consistance est plus efficace que l'algorithme HC3, calculant la hull-consistance ; la situation est inversée pour les variables ne possédant qu'une seule occurrence. Il semblait donc important de considérer les contraintes au niveau de leurs projections sur chacune des variables y apparaissant afin de pouvoir utiliser des algorithmes de réduction des domaines différents suivant le nombre d'occurrences. Pour cela, nous avons étendu la définition de la box-consistance afin de capturer les définitions de la hull-consistance et de la box-consistance originale, ce qui nous a permis de définir le nouvel algorithme BC4 [23] capable de remplacer efficacement HC3 et BC3. Les résultats expérimentaux montrent que BC4 est toujours au moins aussi efficace que le meilleur de ces deux algorithmes ;
- les systèmes de contraintes pour lesquelles la correction des résultats est importante sont constitués d'inéquations. Lorsque ces inéquations sont polynômiales, on a actuellement recours soit à des méthodes symboliques [53, 54, 113, 112, 89, 87], soit à des méthodes numériques d'évaluation/découpage [82, 141]. Nous montrons dans la suite qu'il est possible de résoudre ces systèmes grâce aux méthodes classiques de résolution de contraintes d'intervalles (que les inéquations soient polynômiales ou non) en tirant partie de leur propriété de complétude et en résolvant, non pas les contraintes de l'utilisateur, mais leur négation ; nous proposons des algorithmes généraux corrects et non complets basés sur cette idée. En modifiant légèrement ces algorithmes, on a la possibilité de résoudre des systèmes de contraintes où apparaît une variable universellement quantifiée (cas, par exemple, des systèmes modélisant des problèmes de placement de caméra ou de détection de collision en robotique) ;
- afin de mettre en œuvre et valider nos idées, nous avons défini et développé un certain nombre de systèmes :
  - en nous basant sur clp(fd) [50], le langage logique de résolution de contraintes sur les domaines finis de CODOGNET et DIAZ, nous avons mis au point DecLIC [94, 92], un langage de programmation logique avec contraintes (PLC) capable de résoudre des systèmes mixant des contraintes réelles, entières et booléennes. DecLIC nous a servi de plate-forme d'expérimentation lors de nos recherches d'une consistance intermédiaire entre la hull-consistance et la box-consistance ; ainsi, il offre le choix de la notion de consistance à utiliser pour chaque contrainte. Il a aussi été développé dans le but d'offrir à la communauté un

système universitaire gratuit de programmation avec contraintes d'intervalles ; disponible sur le WEB*, il a été utilisé à l'université de Lisbonne [57] pour résoudre des problèmes de diagnostic de dégénérescence nerveuse des extrémités (bras, jambes),

– au-delà des problèmes d'efficacité, la programmation par contraintes souffre encore de l'absence d'environnements de programmation adaptés. En particulier, les programmes contenant des contraintes ne peuvent pas être débogués avec les outils propres au langage hôte. On a en effet deux niveaux clairement séparés : celui du langage hôte correspondant à la structure de pose des contraintes, et celui du *store* de contraintes. Le *store* est normalement inaccessible au programmeur et le processus de propagation des domaines des variables lui reste donc caché. Or, l'observation du *store* permet de constater les phénomènes de convergence lente, ainsi que d'observer quelles contraintes sont réveillées et quelles modifications des domaines elles induisent. À notre connaissance, les outils de débogage pour la programmation par contrainte actuellement disponibles sont liés à une plate-forme particulière (Oz explorer [207] et Oz, Grace [162] et ECL^{i}PS^{e} ...) et ne s'intéressent pas aux contraintes elles-mêmes mais aux domaines des variables. De fait, l'observation directe du *store* de contraintes est impossible car il s'agit d'un réseau généralement énorme et sans aucune structure. En nous basant sur la notion d'opérateur de contraction [24] capable de modéliser la plupart des résolveurs, nous avons défini la notion de S-box, qui permet d'assimiler un ensemble de contraintes à une seule nouvelle contrainte de haut niveau. Grâce aux S-boxes, il est possible d'organiser le *store* de façon à en offrir une vue structurée arborescente. Nous montrons aussi que, lorsque le langage hôte est un langage logique, l'utilisation des S-boxes permet de retrouver dans le *store* la structure clausale du programme. Dans le cadre du projet ESPRIT DiSCiPl†, nous avons développé un débogueur [93] pour des langages de PLC basé sur la notion de S-box,

– d'un intérêt certain pour le prototypage, les langages de PLC tels que DecLIC apparaissent moins bien adaptés à une intégration dans des applications implémentées dans des langages impératifs. En suivant l'exemple d'Ilog Solver [194], nous avons mis au point OpAC, une librairie C++ de résolution de contraintes. Organisée en modules, OpAC offre de nombreuses possibilités d'extension qui en font un outil de choix pour tester de nouveaux algorithmes de résolution de contraintes, ainsi que des stratégies d'énumération ou de propagation des variations de domaines. OpAC est actuellement utilisée au sein de l'équipe « Contraintes » de l'IRIN et dans le cadre du projet FASE‡ (*Facial Analysis and Synthesis of Expressions*) [201] au *Centrum voor Wiskunde en Informatica* (CWI, Pays-Bas). La librairie OpAC est elle-même basée sur JAIL, une librairie paramétrée C++ pour l'arithmétique d'intervalles. Nous montrons dans la suite que, contrairement à l'idée prévalant parfois [149], la flexibilité offerte par la paramétrisation du type des bornes des intervalles ne dégrade pas les performances de la librairie. Nous mettons aussi en évidence que cette paramétrisation autorise le renforcement de sa fiabilité.

Organisation du document

Ce document est virtuellement découpé en deux grandes parties : les parties I à III présentent l'état de l'art des différents thèmes abordés et décrivent les algorithmes que nous avons mis au point avec les preuves de leur correction ; les parties IV et V sont consacrées à la mise en œuvre de ces algorithmes ainsi qu'à la description des implémentations des systèmes développés durant notre thèse. Nous donnons le détail des contenus de ces cinq parties ci-dessous :

1. Arithmétique d'intervalles :
   (a) présentation de l'arithmétique flottante et des problèmes qu'elle occasionne,
   (b) présentation de l'arithmétique d'intervalles, de ses propriétés et de ses applications ;
2. Consistances locales :
   (a) présentation des différentes notions de consistances locales,

*http://www.sciences.univ-nantes.fr/info/perso/permanents/goualard/Research/software.html
†http://discipl.inria.fr/
‡http://www.stw.nl/projecten/C/cwi4088.html

   (b) présentation de l'extension de la définition de la box-consistance et de l'algorithme pour la calculer ;

3. Approximations intérieures et variables quantifiées :
   (a) description de l'état de l'art en matière de calcul d'approximations intérieures de relations réelles et de gestion de contraintes avec quantificateurs,
   (b) présentation des algorithmes élaborés durant notre thèse pour calculer des approximations intérieures en utilisant le filtrage induit par la box-consistance, ainsi que des méthodes de résolution de contraintes avec une variable quantifiée ;
4. Environnements en programmation logique avec contraintes :
   (a) historique et état de l'art des langages de programmation par contraintes d'intervalles,
   (b) réflexions sur l'impact de la forme des contraintes sur l'efficacité des différents algorithmes de calcul de consistances ; présentation du langage DecLIC, de ses capacités et de son implémentation,
   (c) présentation d'un état de l'art en matière de débogage de programmes contenant des contraintes,
   (d) présentation de la notion de S-box, des possibilités que cette abstraction offre en matière de débogage et d'optimisation en programmation par contraintes, et description d'un prototype de débogueur utilisant les S-boxes ;
5. Programmation par contraintes et programmation orientée objets :
   (a) présentation de JAIL, la librairie C++ paramétrée de calcul sur les intervalles développée durant notre thèse ; étude de l'impact sur les performances de différentes techniques d'implémentation,
   (b) présentation d'OpAC, une librairie C++ extensible pour la résolution de contraintes d'intervalles ; description des résultats obtenus pour la résolution de systèmes de contraintes avec une variable universellement quantifiée par un prototype basé sur une extension d'OpAC ;
6. Rappel des principaux points évoqués dans chaque partie et évocation des perspectives de recherches.

Notice

$(x)_b$ x est un nombre en base b, page 7

$\lhd$ Ordre sur les bornes, page 16

$\langle\, x, \alpha \,\rangle$ borne de valeur x et de bracket α, page 16

$\mathbb{I}^{\mathbb{R}}_{\circ}$ ensemble des intervalles à bornes réelles, page 16

Λ signe d'une borne d'intervalle étendu ($+$ ou $-$), page 73

$\lhd$ ordre sur les indices de DEWEY, page 140

$\mathcal{O}_c$ Ordre partiel sur les contraintes, page 139

$\overline{f}$ extension droite aux bornes de f, page 26

$\prec$ ordre sur les *brackets*, page 15

$\underline{f}$ extension gauche aux bornes de f, page 26

$c[x_j^{(i)}]$ expression de la contrainte c projetée sur la i-ième occurrence de la variable x_j, page 54

$f(\boldsymbol{B})$ domaine de variation de la fonction f sur le pavé $\boldsymbol{B}$, page 19

$x + y$ addition flottante de x et y arrondie au plus proche-pair, page 12

$x - y$ soustraction flottante de x et y arrondie au plus proche-pair, page 12

x/y division flottante de x et y arrondie au plus proche-pair, page 12

$x \times y$ multiplication flottante de x et y arrondie au plus proche-pair, page 12

$\text{apx}_{\mathcal{A}}(\rho)$ fonction d'approximation de $\mathcal{A}$, page 18

$(x)_b$ Nombre x écrit dans la base b, page 7

$\mathbb{F}^{\diamond}$ Ens. des bornes des intervalles flottants, page 25

$\inf_{\circ}(\mathcal{S})$ borne inf. pour un ens. réel, page 16

$\beta|_b$ *bracket* de la borne β, page 16

$\sup_{\circ}(\mathcal{S})$ borne sup. pour un ens. réel, page 16

$\boldsymbol{B}|_{I_j, I'}$ pavé obtenu en remplaçant le j-ième intervalle du pavé $\boldsymbol{B}$ par I', page 18

$\mathbb{R}^{\diamond}$ ens. des bornes des intervalles réels, page 16

$\mathsf{Hull}_{\square}(\rho)$ plus petit intervalle de $\mathbb{I}_{\square}$ contenant ρ, page 26

$\mathbb{I}^{\mathbb{R}}_{\square}$ Ens. des intervalles réels fermés, page 16

$\mathbb{I}_{\square}$ ens. des intervalles à bornes fermées, page 26

$\overline{c}$ négation de la contrainte c, page 80

$N[c]$ opérateur de contraction pour la contrainte c, page 38

$C|_{k,\boldsymbol{B}}$ k-ième projection de la contrainte C par rapport au pavé $\boldsymbol{B}$, page 44

ρ_c relation associée à la contrainte c, page 37

$\mathsf{Union}_{\square}(\rho)$ plus petite union d'intervalles de $\mathbb{U}_{\square}$ contenant ρ, page 26

$\mathbb{U}^{\mathbb{R}}_{\square}$ Ens. des unions d'intervalles réels fermés, page 16

$\mathbb{U}_{\square}$ ens. des unions d'intervalles à bornes fermées, page 26

c_{dec} ens. de décomposition de la contrainte c, page 41

$\mathcal{D}_f$ Domaine de déf. de la fonction f, page 18

$\mathrm{dual}(I)$ dual de l'intervalle I, page 73
$1/_h I$ inverse de l'intervalle dirigé I pour la multiplication, page 73
$-_h I$ inverse de l'intervalle dirigé I pour l'addition, page 73
$\mathrm{prop}(I)$ intervalle propre associé à I, page 73
$\mathcal{D}$ Ens. des intervalles dirigés, page 72
$\sigma(I)$ signe de l'intervalle dirigé I, page 73
$\mathsf{Dom}_{\boldsymbol{B}}(v)$ Domaine de la variable v dans le pavé $\boldsymbol{B}$, page 18
$\mathbb{F}$ Ens. de nombres flottants IEEE 754 sans ∞ ni NaNs, page 9
$\mathbb{F}$ ensemble de nombres flottants de format indéfini, page 12
$\mathbb{F}^\infty$ $\mathbb{F} \cup \{-\infty, +\infty\}$, page 9
$\mathsf{SzComPref}(\mathcal{I}_1, \mathcal{I}_2)$ taille du préfixe commun aux indices $\mathcal{I}_1$ et $\mathcal{I}_2$, page 139
$\mathsf{Hull}_\circ(\rho)$ plus petit intervalle de $\mathbb{I}_\circ^\mathbb{F}$ contenant ρ, page 26
$\mathsf{Inner}_\circ(\rho)$ approximation intérieure forte de ρ, page 78
$\mathcal{S}$ Ens. quelconque, page 18
$\mathsf{Union}_\circ(\rho)$ plus petite union d'intervalles de $\mathbb{U}_\circ^\mathbb{F}$ contenant ρ, page 26
$\mathsf{Hull}(\rho)$ plus petit intervalle contenant ρ, page 18
$\mathbb{I}_\circ^\mathbb{F}$ Ens. des intervalles flottants, page 26
I^- borne gauche de l'intervalle dirigé I, page 72
$\overline{\mathbb{I}}$ ens. des intervalles impropres, page 73
$\mathring{I}$ plus grand intervalle ouvert contenu dans I, page 26
I^+ borne droite de l'intervalle dirigé I, page 72
$\mathbb{I}_\circ$ voir $\mathbb{I}_\circ^\mathbb{F}$, page 26
$\mathcal{D}_f^\mathbb{I}$ Ens. des pavés inclus dans $\mathcal{D}_f$, page 18
$\blacklozenge$ extension aux intervalles de la fonction $\lozenge$, page 19
$\mathbb{I}$ Ens. d'intervalles, page 16
$\rho_c{}^{(k)}(\boldsymbol{D})$ k-ième opérateur de contraction de projection pour la contrainte c, page 41
$\pi_k(\rho)$ k-ième projection de la relation ρ, page 18
$\mathbb{F}^\triangleleft$ Ens. des bornes gauches des intervalles flottants, page 25
$\mathbb{R}^\triangleleft$ ens. des bornes gauches des intervalles réels, page 16
$\underline{I}$ Borne gauche d'un intervalle, page 16
$c_1 \overset{c}{\prec} c_2$ Ordre partiel sur les contraintes, page 139
$+^-$ addition intérieure de MARKOV, page 75
$-^-$ soustraction intérieure de MARKOV, page 75
$/^-$ division intérieure de MARKOV, page 75
$\times^-$ multiplication intérieure de MARKOV, page 75
$\mathsf{mid}(I)$ centre de l'intervalle I, page 19
$\mu_t(v)$ Multiplicité de la variable v dans le terme t, page 18
g^+ successeur du flottant g, page 9
$\boldsymbol{B}|_k$ k-ième projection du pavé $\boldsymbol{B}$, page 18
$\mathcal{P}(\mathcal{S})$ Ensemble des parties de $\mathcal{S}$, page 18
g^- prédécesseur du flottant g, page 9

$\mathbb{F}^{\triangleright}$ Ens. des bornes droites des intervalles flottants, page 25

$\mathbb{R}^{\triangleright}$ ens. des bornes droites des intervalles réels, page 16

$\overline{I}$ Borne gauche d'un intervalle, page 16

$\Diamond$ opération arithmétique réelle, page 19

$\lfloor\!\lfloor \beta \rfloor\!\rfloor$ borne gauche arrondie vers $-\infty$, page 26

$\lfloor x \rfloor$ x arrondi vers $-\infty$, page 10

$[\sqrt{x}]_n$ racine carrée flottante de x arrondie au plus proche-pair, page 12

$[x]_n$ x arrondi au plus proche-pair, page 10

$[x]_0$ x arrondi vers 0, page 10

$\lceil\!\lceil \beta \rceil\!\rceil$ borne droite arrondie vers $+\infty$, page 26

$\lceil x \rceil$ x arrondi vers $+\infty$, page 10

$\mathbb{R}$ Ens. des nombres réels, page 15

$\mathbb{R}^{\infty}$ $\mathbb{R} \cup \{-\infty, +\infty\}$, page 15

$\boldsymbol{B}|_{v,\boldsymbol{D}}$ pavé obtenu en remplaçant le domaine de la variable v dans le pavé $\boldsymbol{B}$ par son domaine dans le pavé $\boldsymbol{D}$, page 18

$\boldsymbol{B}|_{v,I}$ pavé obtenu en remplaçant le domaine de la variable v dans le pavé $\boldsymbol{B}$ par son domaine dans l'intervalle I, page 18

$\tilde{\rho}_c$ voir $\tilde{\rho}_{c,v,I_v}$, page 80

$\mathbb{U}_{\circ}^{\mathbb{F}}$ Ens. des unions d'intervalles flottants, page 26

$\mathsf{Union}(\rho)$ plus petite union d'intervalles contenant la relation ρ, page 18

$\tilde{\rho}_{c,v,I_v}$ relation associée à la contrainte $\forall v \in I_v : c$, page 80

$\mathbb{U}_{\circ}^{\mathbb{R}}$ Ens. des unions d'intervalles réels, page 16

$\mathbb{U}_{\circ}$ voir $\mathbb{U}_{\circ}^{\mathbb{F}}$, page 26

$\beta|_v$ Partie réelle de la borne β, page 16

$\mathsf{Var}(t)$ Ens. des variables du terme t, page 18

$\mathsf{w}(I)$ taille de l'intervalle I, page 19

PREMIÈRE PARTIE

L’arithmétique des intervalles

1. Introduction
2. L’arithmétique flottante
3. L’arithmétique d’intervalles

Introduction

LA MANIPULATION DES NOMBRES a été dès l'origine l'une des tâches principales de l'ordinateur, au point que les premiers d'entre eux s'appelaient des « calculateurs ». Du fait de leur nature finie, les ordinateurs ne peuvent représenter exactement tous les nombres réels et les calculs effectués en machine se trouvent donc être des approximations des calculs réels sur un sous-ensemble fini de nombres rationnels. La limitation des erreurs de calcul induites par ces approximations représente un aspect si important de l'*analyse numérique* que Lloyd TREFETHEN a pu s'indigner qu'une définition communément admise en était [229] :

Numerical analysis is the study of rounding errors.

Cette définition semble d'ailleurs corroborée par le contenu de nombre de livres dédiés à l'analyse numérique, dont les premiers chapitres s'appliquent presque systématiquement à présenter les sources d'erreurs de calculs ainsi que les moyens de les éviter. Certains livres tels que celui de WILKINSON [248] s'attachent même uniquement à l'étude des erreurs de calculs pour certaines classes d'algorithmes.

C'est qu'en vérité, l'influence de l'approximation des réels en machine se fait sentir dans tous les calculs — particulièrement dans le cas de processus de résolution itératifs — et si les erreurs introduites peuvent être de peu d'importance lors du calcul de scènes en image de synthèse, elles peuvent aussi engendrer des morts (on se reportera par exemple au rapport B-247094 [230] expliquant l'échec d'un missile *Patriot* dans l'interception d'un *Scud* ayant entrainé la mort de vingt-huit personnes par une erreur de calcul).

Un grand nombre d'informaticiens ne sont pas sensibilisés au problème des erreurs de calcul malgré leur omniprésence et beaucoup considèrent encore l'extension de la précision des nombres rationnels utilisés comme une panacée, confortés en cela par la « course à la précision » menée par les fondeurs de micro-processeurs : des 32 bits des `float`s, on est passé aux 64 bits des `double`s, puis aux 80 bits utilisés en interne par les processeurs de type Intel Pentium, en attendant la généralisation du format `quad` et ses 128 bits. Or, l'exemple ci-dessous montre que la réduction des erreurs de calculs n'est nullement subordonnée à une augmentation de la précision disponible.

Exemple 1 (Calcul en précision bornée [18]). *Soit à calculer les termes de la suite (u_n) définie par :*

$$\begin{aligned} u_0 &= 2 \\ u_1 &= -4 \\ u_{n+1} &= 111 - 1\,130/u_n + 3\,000/(u_n u_{n-1}) \end{aligned}$$

L'évaluation de quelques termes de la suite en utilisant tant des `float`s que des `double`s semble indiquer que la limite en est 100*. Or, la limite des (u_n) est* 6*. De plus, on peut montrer que la limite calculée de la suite sera* 100 quelle que soit la précision choisie.

Si l'augmentation de la précision disponible n'est pas une solution, il existe cependant de nombreuses techniques tendant à limiter les erreurs de calcul ou à les quantifier de façon à prévenir le programmeur de l'instabilité des opérations effectuées. Citons :

- les algorithmes « sophistiqués » permettant de réaliser une opération particulière avec le maximum de précision (par exemple, les algorithmes de sommation (*distillation*) de PICHAT, KAHAN et MØLLER [183, 108]) ;
- la méthode CESTAC [182, 43] (*Contrôle et Estimation Stochastique des Arrondis de Calculs*) de Jean VIGNES et Michel LA PORTE, consistant à arrondir aléatoirement les calculs par excès ou par défaut. En exécutant plusieurs fois un programme, on obtient un ensemble de résultats à partir desquels on extrait une valeur résultat probable ainsi que le nombre de chiffres significatifs ;
- l'arithmétique de Monte-Carlo [224, 225], qui combine la sélection aléatoire de la direction d'arrondi utilisée par la méthode CESTAC avec un mécanisme de *bornage de la précision* consistant à choisir a priori un nombre de chiffres significatifs n tant pour les entrées que pour les sorties (avec n inférieur à la précision du

format de nombres flottants utilisé), les chiffres non significatifs étant modifiés aléatoirement. L'arithmétique de Monte-Carlo permet à la fois de limiter l'influence des arrondis et de prévenir l'utilisateur lorsqu'un problème est mal conditionné ;
- l'arithmétique réelle « exacte » [73, 187, 165, 189] où l'utilisateur indique au départ la précision souhaitée et où tous les calculs sont faits de façon à respecter cette précision, quitte à augmenter « au vol » la taille de la représentation des nombres manipulés ;
- l'arithmétique des intervalles [167, 10], où tout réel non représentable en machine est remplacé par le plus petit intervalle à bornes flottantes le contenant et où tous les calculs sont effectués sur des intervalles de façon à préserver l'inclusion des opérations réelles correspondantes.

L'étude approfondie de propagation des erreurs nécessaire à la définition d'algorithmes « sophistiqués » en interdit le recours systématique. Quant à la méthode CESTAC, il suffit de lire l'article *The improbability of Probabilistic Error Analyses for Numerical Computations* [122] de William KAHAN — l'un des pères de la norme IEEE 754 [116] concernant la représentation des nombres flottants — pour se convaincre qu'elle ne constitue pas la panacée en ce qui concerne la maîtrise des erreurs de calculs. En particulier, KAHAN réfute l'une des hypothèses de base de la méthode concernant le fait que les erreurs d'arrondi suivent une distribution normale. De plus, les caractéristiques communes de CESTAC et de l'arithmétique de Monte-Carlo les rendent inutilisables dans tous les cas où la reproductibilité des calculs est nécessaire. L'arithmétique réelle « exacte » supprime tous les problèmes d'arrondi mais à un prix qui la rend difficilement utilisable pour faire du calcul intensif. Enfin, l'arithmétique des intervalles, bien qu'implémentable efficacement, est souvent rejetée dans la littérature sous l'accusation de fournir des résultats beaucoup trop pessimistes pour être utilisables [250, p. 566–567].

Cependant, l'arithmétique des intervalles possède par ailleurs de nombreuses qualités puisqu'elle permet à la fois de tenir compte des imprécisions sur les données, de raisonner sur des ensembles de valeurs (ce qui la rend particulièrement intéressante en optimisation globale [100, 145]) et d'encadrer avec certitude les solutions réelles de systèmes d'équations/inéquations — d'où son intérêt en résolution de contraintes.

Dans la suite, nous nous focaliserons donc uniquement sur l'arithmétique des intervalles. Telle qu'implémentée en machine, elle hérite nombre de ces propriétés de l'arithmétique flottante. Nous commencerons donc par discuter la représentation des nombres réels en machine par les nombres flottants, ainsi que les propriétés de l'arithmétique flottante dans le chapitre 1. Puis, le chapitre 2 présentera l'arithmétique des intervalles à bornes tant réelles que flottantes, en s'appuyant sur les résultats du chapitre 1.

CHAPITRE 1

L'arithmétique flottante

Norme IEEE 754 — Formats des nombres en virgule flottante — Arrondis — Propriétés de l'arithmétique flottante — Déficiences de la norme IEEE 754

L'ARITHMÉTIQUE FLOTTANTE a été utilisée presque dès l'origine pour représenter les nombres réels en machine [136, p. 225]. Cependant, si les travaux de BURKS, GOLDSTINE et VON NEUMANN [42] ont prôné dès les années quarante l'utilisation de l'arithmétique binaire, les années soixante et soixante-dix ont vu chaque fabricant d'ordinateur développer sa propre représentation des nombres flottants : si un grand nombre de machines utilisaient une représentation binaire, certaines autres parmi les plus répandues avaient recours à une représentation hexadécimale (IBM 360/370) et d'autres encore (calculateurs HP) à une représentation décimale [180]. Même les conventions de calcul étaient différentes d'une machine à une autre. Ainsi, KAHAN, l'un des auteurs du rapport K-C-S (KAHAN, COONEN, STONE) précurseur de la norme IEEE 754 (standard de représentation et de manipulation des nombres flottants binaires), peut-il évoquer dans ces mémoires [127] certaines des idiosyncrasies des ordinateurs des années soixante-dix qui conduisaient les programmeurs à écrire du code non portable afin de profiter au mieux des capacités de la machine sur laquelle ils travaillaient.

La norme IEEE 754 est désormais un standard de fait supporté par pratiquement toutes les machines modernes à l'exception de certains CRAYs [124] (CRAY X-MP, Y-MP…). Aussi, nous ne présenterons dans ce chapitre que l'arithmétique en nombres flottants telle que définie par cette norme. Nous commencerons par décrire les formats de représentation des nombres, puis nous exposerons les propriétés de l'arithmétique flottante. Enfin, nous discuterons les limites de la norme. En particulier, nous verrons que son respect ne suffit pas à assurer l'identité des calculs d'une machine à une autre.

1.1 Présentation de la norme IEEE 754

La norme IEEE 754 [116] (abrégée en *la norme* dans la suite) définit la représentation de *nombres binaires en virgule flottante* (ou simplement *flottants*) ainsi que certaines opérations associées. En particulier, elle spécifie quatre formats de flottants de précisions différentes :

- le format `single` (`float` en C), devant obligatoirement être présent dans tout système — matériel ou logiciel — conforme à la norme ;
- le format `single extended` ;
- le format `double` (`double` en C), que l'on retrouve dans la plupart des implémentations de la norme ;
- et le format `double extended` (`long double` en C), entièrement optionnel.

1.1.1 Format des nombres en virgule flottante

Quel que soit le format considéré, un nombre flottant g est représenté en mémoire par la donnée de trois valeurs :

- un bit de signe s ;
- un exposant E ;
- et une mantisse m.

On a alors :

$$\mathrm{g} = (-1)^s \times m \cdot 2^E \tag{1.1}$$

Note : Dans la suite, nous utiliserons la notation semilogarithmique de RUTISHAUSER [222, p. 4] pour représenter les nombres en virgule flottante :

$$\mathrm{g} = (-1)^s \times m \cdot 2^E \quad \equiv \quad \mathrm{g} = (-1)^s \times m\,{}_2E$$

La norme IEEE 754 définit un nombre flottant de façon un peu différente en faisant stocker un exposant biaisé $e = E + \textit{biais}$ à la place de E et une partie fractionnaire f définie à partir de m comme on le verra plus loin. Les trois nombres caractéristiques définissant un flottant sont rassemblés en mémoire suivant l'ordre (s, e, f) (*cf.* figure 1.1). L'utilisation d'un biais permet de représenter des exposants négatifs sans recourir à la complémentation à deux (ou à un). L'un des avantages est que la comparaison de deux flottants peut ainsi se faire par une comparaison lexicographique des chaînes de bits les représentant.

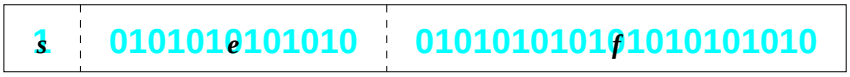


FIG. 1.1: Représentation d'un nombre flottant en mémoire

Les machines des années soixante se partageaient en deux camps en ce qui concerne le traitement de la division d'un nombre positif par 0 : certaines retournaient le plus grand nombre flottant représentable ; d'autres arrêtaient tout calcul en signalant une erreur. Ces deux conventions compliquaient considérablement le travail des programmeurs et les obligeaient à modifier leurs formules de calcul en conséquence. Par exemple, la résistance totale T d'un circuit comportant deux résistances en parallèle R_1 et R_2 (fig. 1.2) s'exprime par la formule :

$$T = \frac{1}{1/R_1 + 1/R_2} \tag{1.2}$$

Si la résistance de R_1 est nulle, la formule (1.2) a quand même un sens et T vaut alors 0. Avec les conventions décrites ci-dessus, on obtient cependant soit une valeur proche de R_2, soit une erreur.

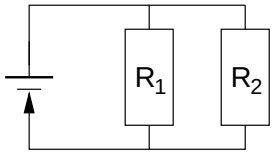


FIG. 1.2: Un circuit avec deux résistances en parallèle

Reprenant certaines idées déjà présentes dans les machines *Z* de Konrad ZUSE [136, p. 225] — l'un des pionniers de l'informatique — la norme IEEE 754 définit un infini positif et un infini négatif, avec les règles : $\mathrm{g}/0 = +\infty$, $-\mathrm{g}/0 = -\infty$ (pour tout nombre flottant strictement positif g). Afin d'obtenir un comportement symétrique pour la division par une quantité infinie, on a introduit le nombre -0, qui est égal à 0, et qui permet d'écrire : $\mathrm{g}/0 = +\infty$ et $\mathrm{g}/-0 = -\infty$.

On peut alors se demander quel est le résultat d'opérations telles que : $0/0$ ou $+\infty/+\infty$. Comme il est primordial de pouvoir rendre un résultat pour chaque opération afin de ne pas perturber le flot de calcul, la norme intègre la notion de *quantité indéfinie* (en anglais *Not a Number*, couramment abrégé en *NaN*) qui est le résultat de toutes les opérations n'ayant pas de sens mathématique, comme la racine carrée d'un nombre négatif par exemple*. On pourra se reporter à la table 1.1 pour la liste des cas d'apparition de *NaN* en ce qui concerne les opérations définies par la norme.

Afin d'illustrer plus simplement nos propos, nous allons utiliser un système de nombres flottants en format « *tiny* » codant l'exposant e sur deux bit, de biais 1 et dont la partie fractionnaire f est représentée sur deux bits également (fig. 1.3).

* Avec un exception notable dans ce cas : la racine carré de -0 est définie comme étant égale à -0

TAB. 1.1: Cas d'apparition d'un NaN pour les opérateurs arithmétiques définis par la norme IEEE 754

$x+y$	*NaN*	$-\infty$	0	$+\infty$
NaN	✓	✓	✓	✓
$-\infty$	✓			✓
0	✓			
$+\infty$	✓	✓		

$x-y$	*NaN*	$-\infty$	0	$+\infty$
NaN	✓	✓	✓	✓
$-\infty$	✓	✓		
0	✓			
$+\infty$	✓			✓

$x \times y$	*NaN*	$-\infty$	0	$+\infty$
NaN	✓	✓	✓	✓
$-\infty$	✓		✓	
0	✓	✓		✓
$+\infty$	✓		✓	

x/y	*NaN*	$-\infty$	0	$+\infty$
NaN	✓	✓	✓	✓
$-\infty$	✓	✓		✓
0	✓		✓	✓
$+\infty$	✓	✓		✓

$\sqrt{x}$	
NaN	✓
$-\infty$	
$x < 0$	✓
0	
$+\infty$	

Notation : Dans la suite, nous indiquerons parfois la base employée pour représenter un nombre par la notation suivante : $(x)_b$ indique que le nombre x est écrit en base b. Cependant, afin d'éviter d'alourdir inutilement le texte, nous n'aurons recours à cette notation qu'en cas d'ambiguïté possible.

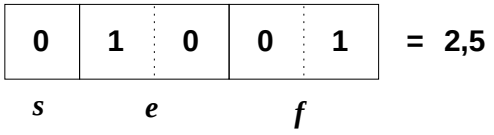


FIG. 1.3: Représentation d'un nombre flottant en format *tiny*

La représentation d'un nombre flottant g en utilisant la formule (1.1) n'est pas unique. En déplaçant la virgule vers la gauche, on peut, par exemple, réécrire le nombre $1{,}01\ _2 1$ sous la forme $0{,}101\ _2 2$ ou $0{,}010\ 1\ _2 3$, etc. Cependant, pour une taille de partie fractionnaire donnée, c'est bien évidemment la première forme qui est la plus précise. On appelle *normalisé* un nombre flottant dont le bit à gauche de la virgule est égal à 1. La norme IEEE 754 impose l'emploi et le stockage des nombres en forme normalisée — sauf dans certains cas, comme on le verra plus loin — afin de garder le maximum de chiffres significatifs dans la partie fractionnaire. Le bit avant la virgule dans la mantisse m étant toujours à 1, on peut alors se passer de le coder explicitement et ne stocker que la partie fractionnaire f (*cf.* la figure 1.3, où le nombre $2{,}5 = (-1)^0 \times 1{,}01\ _2 1$ a été codé dans le format *tiny*). On notera que l'on a l'exposant biaisé e égal à 2, auquel il faut retirer le biais de 1 : le champ e code les exposants biaisés $0, 1, 2, 3$ correspondant aux exposants non biaisés $-1, 0, 1, 2$. Le format *tiny* n'utilise cependant pas les valeurs extrêmes des exposants, réservant $e = 0$ pour coder les deux zéros et les nombres dénormalisés (*cf.* ci-dessous), et $e = 3$ pour coder les infinis ainsi que les *NaN*s. En valeur absolue, le nombre représentable le plus petit (excepté 0) est alors $1{,}00\ _2 0 = (1)_{10}$ et le plus grand $1{,}11\ _2 1 = (3{,}5)_{10}$. Pourquoi réserver l'exposant $e = 0$ pour coder 0, nous privant alors de la possibilité de représenter les nombres $0{,}5 = 1{,}00\ _2 -1$, $0{,}75 = 1{,}10\ _2 -1 \ldots$? Ceci est dû au fait que nous sommes obligés de violer la convention de normalisation pour stocker le nombre 0, puisque le triplet $(s = 0, e = 0, f = 0)$ représente le nombre $(-1)^0\ 1{,}00\ _2 0 = 1$ en format normalisé, et non 0.

Considérons maintenant la répartition des nombres flottants en format *tiny* sur la ligne réelle. La figure 1.4 nous permet de noter deux faits remarquables :

1. l'espace compris entre 0 et le plus petit nombre représentable 1 est beaucoup plus grand qu'entre celui-ci et le nombre flottant suivant ;
2. les nombres flottants ne sont pas répartis uniformément sur la ligne réelle. Plus précisément, l'espace entre deux nombres flottants est égal à $\epsilon \times 2^E$ où ϵ correspond à la différence entre le nombre 1 et le flottant suivant ($\epsilon = 0{,}25$ dans le format *tiny*). C'est ce que l'on appelle l'*epsilon* du format [222] ; on note aussi *ulp* (*unit in the last place*) le bit le plus à droite dans la partie fractionnaire — seul bit à 1 pour ε.

Comme on va le voir en détails plus loin, tout nombre réel non représentable exactement par un flottant est approché par un flottant proche pour être stocké en machine. Le point 1 ci-dessus est donc particulièrement important

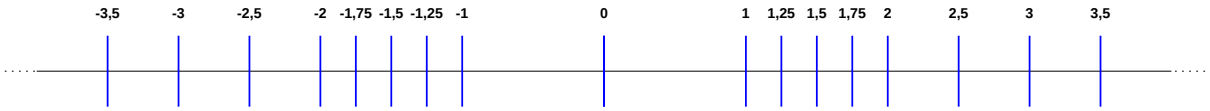


FIG. 1.4: Répartition des flottants normalisés en format *tiny*

puisqu'il suggère que l'erreur due à l'approximation d'un réel proche de 0 par un flottant est très supérieure à celle engendrée par l'approximation d'un nombre légèrement supérieur à 1. Dans les faits, les ordinateurs des années soixante adoptaient souvent la convention d'arrondir à 0 les nombres réels compris entre 0 et le plus petit flottant normalisé, ce qui entraînait nombre d'erreurs. En particulier, la relation :

$$x = y \iff x - y = 0$$

n'est pas vérifiée si les « petits nombres » sont projetés sur 0. En effet, si x et y sont deux flottants très proches, leur différence est très petite et sera remplacée par 0. Cela peut devenir la source de bogues subtiles. Il suffit de considérer le programme [86] :

$$\texttt{if } (x \neq y) \texttt{ then } z = 1/(x - y)$$

où l'on cherche manifestement à se prémunir contre une division par 0. Si les variables x et y contiennent des valeurs différentes mais très proches, l'instruction du `then` sera effectuée et il y aura quand même une division par 0 (le résultat de la soustraction $x - y$ étant assimilé à 0).

Afin de pallier cet inconvénient, la norme IEEE 754 introduit la notion de *gradual underflow* en autorisant l'utilisation de nombres dénormalisés pour « remplir » l'espace entre 0 et le plus petit flottant représentable. Ainsi, si l'on considère notre format *tiny*, on adopte la convention que tout flottant dont l'exposant biaisé e est nul et dont la partie fractionnaire est non nulle est un *nombre dénormalisé* à interpréter comme :

$$(-1)^s \times \mathbf{0}.f\ {}_2e - \text{biais}$$

Ceci est à comparer avec l'interprétation du triplet (s, e, f) pour un nombre normalisé :

$$(-1)^s \times \mathbf{1}.f\ {}_2e - \text{biais}$$

Dans le format *tiny*, on obtient ainsi la possibilité de représenter les nombres $0{,}25 = (-1)^0 \times 0{,}01\ {}_20$, $0{,}5 = (-1)^0 \times 0{,}10\ {}_20$, et $0{,}75 = (-1)^0 \times 0{,}11\ {}_20$, comblant harmonieusement l'espace entre 0 et 1 (*cf.* figure 1.5)

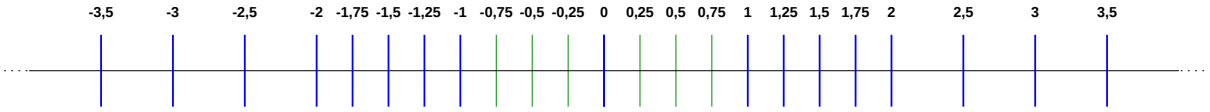


FIG. 1.5: Répartition des flottants en format *tiny* avec les nombres dénormalisés

À titre de référence, le tableau 1.2 présente les caractéristiques des format `single` et `double` définis par la norme et la figure 1.6 rappelle l'interprétation des champs du triplet (s, e, f) pour le format `double`. On peut noter au passage que la norme ne s'intéresse pas à la valeur de la partie fractionnaire en ce qui concerne les *NaN*s. Tout programmeur est donc libre d'utiliser le champ f d'un *NaN* pour coder de l'information utile, telle que le type de l'opération en ayant entraîné l'apparition.

Si le recours à la normalisation permet d'utiliser au mieux l'espace de la partie fractionnaire, il induit aussi deux effets pervers :

– lorsque l'on additionne deux valeurs de magnitudes très différentes, l'opération d'alignement préalable de l'exposant de la plus petite valeur sur l'exposant de la plus grande lui fait perdre des chiffres de sa partie fractionnaire. C'est le phénomène d'*absorption* illustré ci-dessous ;

TAB. 1.2: Caractéristiques des formats `single` et `double`

Paramètre	Format *single*	Format *double*
$E_{\min}$	-126	$-1\,022$
$E_{\max}$	$+127$	$+1\,023$
biais	$+127$	$+1\,023$
f	23 bits	52 bits
taille totale	32 bits	64 bits

– lorsque l'on soustrait deux valeurs très proches, leurs chiffres significatifs s'annulent deux à deux pour ne plus garder que les valeurs les plus à droite dans la partie fractionnaire (les plus bruitées). L'opération d'alignement qui suit ramène ces valeurs vers la gauche, faisant croire à l'utilisateur qu'elles sont correctes. C'est le phénomène de *cancellation*.

Exemple 1.2 (Absorption et cancellation). *On considère le format tiny. Soit à ajouter les valeurs* $2{,}5 = 1{,}01\,_2 1$ *et* $0{,}25 = 0{,}01\,_2 0$*. L'opération d'alignement transforme* $0{,}01\,_2 0$ *en* $0{,}001\,_2 1$*; après élimination des bits surnuméraires de la partie fractionnaire, on fait l'addition de* $0{,}01\,_2 0$ *et* $0{,}00\,_2 1$*. Il vient alors :* $2{,}5 + 0{,}25 = 2{,}5$*. La valeur 2,5 a* absorbé *la valeur 0,25.*
Considérons maintenant un format décimal pour plus de clarté. Soit à faire la soustraction des valeurs $a = 9{,}877\,654\,56 \cdot 10^0$ *et* $b = 9{,}877\,645\,4 \cdot 10^0$*. Il vient :* $a - b = 0{,}000\,000\,2 \cdot 10^0$*, et après normalisation :* $a - b = 2.0\,000\,000 \cdot 10^{-7}$ *(cancellation des chiffres significatifs). L'utilisateur est donc porté à croire qu'il connaît le résultat de la soustraction avec 8 chiffres de précision au lieu d'un seul. De plus, la valeur* 2 *correspond à la soustraction des derniers chiffres — les plus bruités — des parties fractionnaires de* a *et* b*. Par conséquent, la probabilité est grande que ces valeurs soient fausses et que le résultat de la soustraction soit entièrement faux.*

La norme IEEE 754 définit un certain nombre de *fonctions recommandées* dont l'une des plus utiles est la fonction `nextafter()` qui, étant donné un nombre flottant, retourne le flottant suivant ou précédent :

Notations : Soit $\mathbb{F}$ un ensemble de nombres flottants respectant la norme IEEE 754 (sans les infinis ni les *NaN*s). Soit $\mathbb{F}^\infty = \mathbb{F} \cup \{-\infty, +\infty\}$ et soit g un élément de $\mathbb{F}^\infty$. On notera g^+ (resp. g^-) le plus petit flottant appartenant à $\mathbb{F}^\infty$ plus grand que $\mathrm{g} \in \mathbb{F}^\infty$ (resp. le plus grand flottant plus petit que g), avec les cas spéciaux : $(+\infty)^+ = +\infty$, $(-\infty)^- = -\infty$, $(+\infty)^- = \max(\mathbb{F})$, $(-\infty)^+ = \min(\mathbb{F})$.

$$\begin{cases} e = 2\,047, & f \neq 0 & : v = \text{NaN} \\ e = 2\,047, & f = 0 & : v = (-1)^s \times \infty \\ 0 < e < 2\,047 & & : v = (-1)^s \times 1.f\,_2 e - 1\,023 \\ e = 0, & f \neq 0 & : v = (-1)^s \times 0.f\,_2 e - 1\,022 \\ e = 0, & f = 0 & : v = (-1)^s \times 0 \end{cases}$$

FIG. 1.6: Interprétation du triplet (e, s, f) pour le format `double`

1.1.2 L'arrondi

Comme on l'a évoqué précédemment, la majorité des nombres réels doivent être remplacés par des nombres flottants proches pour être manipulés en machine. Ce processus d'arrondi intervient dans trois cas :

1. lorsqu'une donnée d'entrée (tapée au clavier par un utilisateur, ou présente sous forme de constante dans le programme) n'est pas un flottant ;
2. lorsque le résultat d'un calcul (addition de deux flottants, par exemple) n'est pas un flottant ;

3. lorsque une valeur en machine (nombre binaire) ne peut être transformée exactement en nombre décimal pour être affichée.

On appelle *arrondi correct* un processus d'arrondi approchant un nombre réel par le flottant le plus proche par excès ou par défaut. La norme IEEE 754 définit quatre types d'arrondi dont on trouvera une illustration dans la figure 1.7 pour une valeur x réelle et deux nombres flottants consécutifs α et α^+ :

$$\begin{cases} \lfloor x \rfloor & \text{arrondi vers } -\infty \\ \lceil x \rceil & \text{arrondi vers } +\infty \\ [x]_0 & \text{arrondi vers } 0 \\ [x]_n & \text{arrondi au plus proche-pair} \end{cases}$$

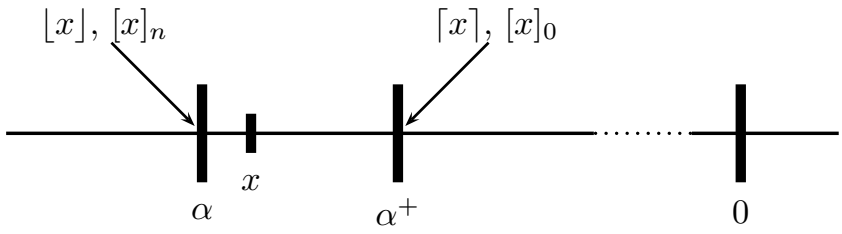


FIG. 1.7: Les différents types d'arrondi

Dans ce document, on utilisera le plus souvent les arrondis vers $-\infty$ et $+\infty$ (ou *arrondi vers le bas* et *arrondi vers le haut*, que l'on peut définir formellement par :

$$\begin{array}{rll} \text{Arrondi vers le bas :} & \lfloor\ \rfloor : & \mathbb{R} \longrightarrow \mathbb{F}^\infty \\ & & x \longmapsto \max\{y \in \mathbb{F}^\infty \mid y \leqslant x\} \\ \text{Arrondi vers le haut :} & \lceil\ \rceil : & \mathbb{R} \longrightarrow \mathbb{F}^\infty \\ & & x \longmapsto \min\{y \in \mathbb{F}^\infty \mid x \leqslant y\} \end{array}$$

Comme le soulignent KAHAN, MULLER [123, 171] et KNUTH [136, p. 237], un arrondi correct est hautement souhaitable car c'est une condition *sine qua non* pour pouvoir faire par la suite des preuves de correction des algorithmes numériques. La norme spécifie donc que les quatres opérations de base (addition, soustraction, multiplication, division) ainsi que la racine carrée et les conversions (nombre binaire de ou vers un nombre décimal) doivent être correctement arrondies.

Un manque notable de la norme IEEE 754 dont nous reparlerons dans les chapitres suivants est l'absence de spécification d'un certain nombre de fonctions de base comme les fonctions transcendantales ($\sin$, $\cos$...), l'exponentielle et le logarithme népérien. Ceci est dû au problème appelé *The Table Maker's Dilemma* (TMD) par GOLDBERG [86] dont l'exemple ci-dessous est une illustration.

Exemple 1.3 (Table maker's dilemma). *Soit à calculer la valeur de $e^{1,626}$ avec cinq chiffres de précision et un arrondi vers $+\infty$. La séquence ci-après montre le calcul de $e^{1,626}$ avec une précision croissante.*

$$\exp(1{,}626) \simeq \boxed{\textbf{\textit{5,083}}\,5} \simeq \boxed{5{,}083\ 50} \simeq \boxed{5{,}083\ 500} \simeq \boxed{5{,}083\ 500\ 0} \simeq \boxed{5{,}083\ 499\ 99}$$

On constate que cinq chiffres de précision sont insuffisants pour arrondir correctement vers $+\infty$ le résultat car on ne peut pas savoir si le dernier chiffre 5 correspond à une approximation par défaut ou par excès de la suite des chiffres non calculés. Il en est de même jusqu'à huit chiffres de précision et c'est seulement avec neuf chiffres que l'on peut enfin décider d'arrondir $e^{1,626}$ à la valeur $5{,}083\ 5$.

Plus généralement, il est impossible de savoir à l'avance le nombre de chiffres à calculer pour pouvoir arrondir correctement les fonctions précitées à une certaine précision. Cependant, comme le soulignent MULLER

et TISSERAND [172], ces fonctions délivrant des résultats non algébriques pour toutes les valeurs algébriques (et donc les nombres flottants) à l'exception de valeurs triviales (e^0, $\ln 1 \ldots$), on est assuré qu'il existe un niveau de précision de calcul tel que le TMD n'apparaît pas. Une stratégie naturelle pour obtenir des fonctions correctement arrondies est celle de ZIV [253], consistant à évaluer avec une précision croissante le terme à arrondir jusqu'à ce qu'il soit possible de déterminer l'arrondi à appliquer. On pourra se reporter aux travaux de LEFEVRE *et al.* [148] pour une présentation approfondie du TMD ainsi que la description d'une méthode permettant d'arrondir correctement la fonction exponentielle pour le format `double`. Notons enfin que, contrariant l'affirmation de POTTS et EDALAT [187] :

> *Current floating-point representations in computers have so many bits that verification of floating point functions cannot be achieved by exhaustively testing every possible input combination.*

le problème du TMD a été résolu pour le format `single` pour les fonctions $\log_2$, $\sin$, $\exp$, $\log$ et $\arctan$ en cherchant exhaustivement le nombre de chiffres à calculer pour pouvoir faire un arrondi correct [208, 172]. Les valeurs trouvées permettent de conjecturer qu'il faut approximativement $2n$ chiffres corrects pour pouvoir arrondir à n chiffres de précision. En pratique, il suffit donc de calculer avec le maximum de précision en format `double` une fonction comme $\sin$ pour pouvoir arrondir correctement à coup sûr le résultat en format `single`.

La recherche exhaustive faite sur le format `single` ne peut pour le moment être étendue au format `double` du fait du grand nombre de valeurs à tester. Cependant, on pourra se reporter à la page WEB du projet *Arénaire* [17] où sont rapportées régulièrement les précisions maximum de calcul à effectuer pour pouvoir arrondir correctement les fonctions trigonométriques usuelles ainsi que l'exponentielle et le logarithme népérien.

1.1.3 Opérateurs non définis par la norme IEEE 754

Dans l'état actuel des connaissances, le TMD interdit d'intégrer les fonctions transcendantales dans la norme IEEE 754 pour tous les formats reconnus afin de ne pas nuire à son implémentation efficace sur toutes les machines. La précision de ces fonctions varie donc d'un ordinateur à un autre et reste parfois inconnue de l'utilisateur. On verra au chapitre 11 une conséquence d'un tel état de fait. Notons cependant qu'il reste possible de faire des calculs « portables » impliquant des fonctions transcendantales en utilisant des librairies telles que `fdlibm` (développée par *Sun Microsystems* et dont les sources sont librement distribuables et utilisables) où chaque fonction a fait l'objet d'un calcul d'erreur permettant d'en certifier la précision — modulo la qualité d'implémentation des opérations de base spécifiées par la norme IEEE 754 (*cf.* la section 1.3). Dans la suite, c'est cette librairie qui nous servira de référence pour le format `double` en ce qui concerne les fonctions non supportées par la norme, tant en ce qui concerne leur précision que leur comportement dans les cas exceptionnels (*cf.* le tableau 1.3).

TAB. 1.3: Cas spéciaux pour les opérateurs arithmétiques non définis par la norme IEEE 754

x	$\exp(x)$	$\log(x)$	$\sin(x)$	$\cos(x)$	$\tan(x)$	$\mathrm{acos}(x)$	$\mathrm{asin}(x)$	$\mathrm{atan}(x)$
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
$-\infty$	0,0	NaN	NaN	NaN	NaN			
$x < -745.13$	0,0							
$x < -1$						NaN	NaN	
$x < 0$		NaN						
0		$-\infty$						
$x > 1$						NaN	NaN	
$x > 709.78$	$+\infty$							
$+\infty$	$+\infty$	$+\infty$	NaN	NaN	NaN			

Source : librairie `fdlibm` sur les `doubles`

Paradoxalement, il semble que l'unité arithmétique des processeurs grand public de type Intel Pentium se trouve mieux documentée que celle de stations de travail orientées vers le calcul scientifique, et l'on trouve dans la documentation technique Intel [56] un exposé des propriétés (comportement vis-à-vis d'entrées exceptionnelles, précision) des algorithmes utilisés pour implémenter le calcul des fonctions transcendantales.

1.2 Propriétés de l'arithmétique flottante

Comme le souligne KNUTH [136, p. 229], bien que le calcul avec les nombres flottants soit intrinsèquement faux, il est primordial de préserver un maximum des propriétés de l'arithmétique réelle afin de pouvoir écrire les algorithmes numériques le plus naturellement possible et en faire des preuves de correction. Nous présentons dans cette section certaines des propriétés fondamentales de l'arithmétique flottante qui nous seront nécessaires dans la suite de notre exposé. Nous ne pourrons évoquer ici que les opérations expressément définies par la norme IEEE 754 puisque, comme nous l'avons vu plus haut, il n'existe pas de garantie d'arrondi correct pour les autres.

Nous considérerons dans la suite de cette section que tous les résultats de calculs flottants sont arrondis *au plus proche-pair* (*i.e.* arrondi au flottant le plus proche du résultat réel avec arrondi sur le flottant pair en cas d'ambiguïté — voir [136, p. 241] pour une justification de cette règle). Les propriétés énoncées étant indépendantes de la précision du format flottant utilisé, nous considérerons un ensemble de nombres flottants de précision finie indéterminée noté $\mathbb{F}$.

Deux des trois grandes lois de l'arithmétique réelle ne sont pas vérifiées par l'arithmétique flottante : la distributivité de la multiplication par rapport à l'addition et l'associativité (on trouvera des contre-exemples pages 229 et 231 du tome 2 de TAOCP [136]). On a donc pour tous flottants a, b et c :

$$\mathsf{a} + \mathsf{b} = \mathsf{b} + \mathsf{a} \qquad \text{(commutativité de l'addition)} \tag{1.3}$$

$$\mathsf{a} \times \mathsf{b} = \mathsf{b} \times \mathsf{a} \qquad \text{(commutativité de la multiplication)} \tag{1.4}$$

$$\mathsf{a} + (\mathsf{b} + \mathsf{c}) \neq (\mathsf{a} + \mathsf{b}) + \mathsf{c} \qquad \text{(non associativité)} \tag{1.5}$$

$$\mathsf{a} \times (\mathsf{b} + \mathsf{c}) \neq (\mathsf{a} \times \mathsf{b}) + (\mathsf{a} \times \mathsf{c}) \qquad \text{(non distributivité)} \tag{1.6}$$

On trouvera d'autres propriétés de l'arithmétique flottante — dont certaines découlent trivialement des propriétés données ici — dans les travaux de GOLDBERG [86], KNUTH [136, p. 229–245] et PRIEST [190].

1.3 Retour sur la norme IEEE 754

Comme on l'a vu précédemment, la généralisation de la norme IEEE 754 a été un grand progrès en ce qui concerne la qualité et la portabilité des programmes ayant recours à l'arithmétique flottante. Cependant, on doit aujourd'hui reconnaitre que la norme n'a pas atteint l'un des objectifs ayant motivé son introduction : l'obtention des même résultats sur tous les ordinateurs. Considérons l'exemple suivant :

Exemple 1.4 (Résultats différents malgré le respect de la norme IEEE 754). *Étant donnés trois nombres flottants en format single* $\mathsf{a} = [11\ 111\ 113]_n$, $\mathsf{b} = [-11\ 111\ 111]_n$ *et* $\mathsf{c} = [7{,}511\ 111\ 1]_n$, *calculons la valeur des expressions* $(\mathsf{a} + \mathsf{b}) + \mathsf{c}$ *et* $\mathsf{a} + (\mathsf{b} + \mathsf{c})$. *Cet exemple est repris du tome 2 de TAOCP [136] et est censé servir de contre-exemple en ce qui concerne l'associativité de l'addition flottante. Sur une Sun UltraSparc 1/167 MHz, on obtient :*

$$\begin{cases} (\mathsf{a} + \mathsf{b}) + \mathsf{c} &= 9{,}511\ 111 \\ \mathsf{a} + (\mathsf{b} + \mathsf{c}) &= 10{,}0 \end{cases}$$

En revanche, sur un ordinateur personnel basé sur un AMD K6/166 MHz, on obtient :

$$\begin{cases} (\mathsf{a} + \mathsf{b}) + \mathsf{c} &= 9{,}511\ 111 \\ \mathsf{a} + (\mathsf{b} + \mathsf{c}) &= 9{,}511\ 111 \end{cases}$$

On constate ainsi des disparités dans les résultats sur deux machines implémentant scrupuleusement la norme IEEE 754. Le problème ici est que la norme spécifie que chaque opération doit être correctement arrondie lors de la copie du résultat dans la variable destination, mais n'impose rien sur le calcul lui-même. Ainsi, sur l'AMD K6, tous les calculs sont faits dans les registres de 80 bits de l'unité arithmétique *et arrondis correctement dans ce format*, puis le résultat est arrondi correctement pour être stocké dans la variable résultat de type `single`, alors que l'UltraSparc évalue les expressions avec des registres de 64 bits. Au final, les erreurs intermédiaires n'étant pas les mêmes, on obtient des résultats différents tout en ayant respecté la norme.

Le même type de problème apparaît avec les langages qui imposent un format par défaut pour tous les calculs en nombres flottants. Ainsi, en C, il y a promotion de tous les calculs flottants au format `double`, ce qui entraîne des comportements étranges comme celui illustré par le programme 1.1 : sur une machine basée sur un processeur AMD K6 166 MHz, le résultat affiché est `"pas égaux"` car la division à la ligne 5 est faite en format `double` avant d'être arrondie au format `single`, puis, la valeur contenue dans `q` est retransformée en `double` à la ligne 6 pour être comparée à la valeur `1,0/10,0` évaluée en `double`.

PROG. 1.1: Promotion et norme IEEE 754

```
1  int
2  main()
3  {
4    float q, a=1.0, b=10.0;
5    q=a/b;
6    if (q==1.0/10.0) {
7      printf("égaux\n");
8    } else {
9      printf("pas égaux\n");
10   }
11 }
```

Enfin, notons avec KAHAN [123] que l'implémentation de la norme sur la majorité des unités arithmétiques modernes ne sert à rien si, comme c'est le cas, les langages n'offrent pas le support de ses fonctionnalités et si les compilateurs se permettent de modifier le code de l'utilisateur sans tenir compte de certaines propriétés des flottants. En particulier, il n'existe pas à notre connaissance de langage reconnaissant au niveau sémantique la modification de la direction de l'arrondi. Ainsi, dans la table 1.4, le programme de gauche peut-il être « optimisé » par le compilateur en le programme de droite. Le compilateur ne connaissant pas l'effet des fonctions de changement de direction d'arrondi `RoundDn()` et `RoundUp()`, il présume qu'elles n'ont pas d'incidence sur les opérations du programme et s'autorise à déplacer les instructions, voire à précalculer le quotient `a/b` à la compilation suivant l'arrondi par défaut, puis à affecter cette constante à `c` et `d`. On remarquera aussi le remplacement du test `c==c` par le code de la première branche du `if`, le compilateur supposant le test trivialement vérifié dans tous les cas. Or, les *NaN*s n'étant pas des nombres, ils sont non ordonnés et retournent faux pour tous les tests de comparaison (on en verra une application au chapitre 11). Donc, si `c` est un *NaN*, le test `c==c` est faux.

On trouvera d'autres exemples d'optimisation abusive dans les articles de GOLDBERG [86], PRIEST [188] et KAHAN [123]. On pourra aussi lire le manuel du langage Borneo [58], une version de Java intégrant entre autres les manipulations des directions d'arrondi au niveau du langage afin de permettre une plus grande maîtrise par le programmeur des opérations flottantes.

TAB. 1.4: Sur-optimisation par le compilateur

```
int
main()
{
   double a=1.0, b=10.0, c, d;
   RoundDn();
   c=a/b;
   RoundUp();
   d=a/b;
   ...
   if (c==c) {
      printf("égal");
   } else {
      printf("différent");
   }
}
```

« optimisé » en

```
int
main()
{
   double a=1.0, b=10.0, c, d;
   RoundDn();
   RoundUp();
   c=a/b;
   d=a/b;
   ...
   printf("égal");
}
```

CHAPITRE 2

L'arithmétique d'intervalles

Arithmétique des intervalles — Extension aux intervalles des fonctions réelles — arithmétique fonctionnelle/relationnelle

L'ARITHMÉTIQUE D'INTERVALLES consiste à remplacer les calculs sur les nombres réels par des calculs sur des ensembles connexes de réels. On considère souvent que l'histoire « moderne » de l'arithmétique des intervalles commence dans les années soixante avec les travaux de MOORE [167] sur le sujet, même si, comme l'ont fait remarquer NEUMAIER [175], KEARFOTT [130] et BLIEK [36], on peut en faire remonter les premières études aux années vingt (BURKILL [41]) et trente (YOUNG [251]), voire, comme n'hésite pas à le faire MOORE lui-même, à ARCHIMÈDE et à sa méthode d'encadrement de π par des polygones inscrits et exinscrits à un cercle.

L'arithmétique d'intervalles pallie les problèmes de représentation des nombres réels en machine en leur substituant des intervalles à bornes flottantes les contenant et en effectuant tous les calculs ultérieurs conservativement. On peut alors définir des méthodes numériques fiables, telles que la recherche de solutions de systèmes d'équations, par des extensions des méthodes sur les réels [167, 36, 218]. Elle autorise aussi le raisonnement sur des ensembles de valeurs, ce qui la rend particulièrement intéressante en optimisation globale [100, 145] et en parallélisation automatique de programmes [38]. On trouvera d'autres exemples d'application de l'arithmétique des intervalles dans les monographies de KEARFOTT [130, 132], le livre de MOORE [168] et le cours de STOLFI *et al.* [223].

Dans ce chapitre, nous présenterons les propriétés de base de l'arithmétique des intervalles à bornes réelles en considérant tant les intervalles fermés ($\{x \in \mathbb{R} \mid a \leqslant x \leqslant b, \quad a \in \mathbb{R}, b \in \mathbb{R}\}$) que les intervalles ouverts (*e.g.* $\{x \in \mathbb{R} \mid a \leqslant x < b, \quad a \in \mathbb{R}, b \in \mathbb{R}\}$) ; nous discuterons de la notion d'extension aux intervalles de fonctions et de relations réelles, puis nous nous focaliserons sur l'arithmétique des intervalles à bornes flottantes. Nous étudierons les problèmes et limitations liées à l'utilisation de l'arithmétique flottante pour représenter les bornes des intervalles ; nous montrerons ensuite que l'utilisation exclusive de l'arithmétique des intervalles flottants à bornes fermées pose certains problèmes dont les répercussions se feront sentir dans la partie III.

Suivant les définitions de base adoptées, on obtient, soit une *arithmétique fonctionnelle*, soit une *arithmétique relationnelle*. Nous présenterons les deux arithmétiques en indiquant leurs avantages et inconvénients respectifs. Les deux types d'arithmétiques seront utilisés dans la partie II. La présentation de l'implantation des opérateurs sur les intervalles sera faite au chapitre 11.

2.1 Les intervalles à bornes réelles

Soit $\mathbb{R}$ l'ensemble des nombres réels et $\mathbb{R}^\infty = \mathbb{R} \cup \{-\infty, +\infty\}$ une compactification de $\mathbb{R}$ [137]. Comme on le définira formellement plus loin, un *intervalle à bornes réelles* correspond à un ensemble connexe de réels et peut être représenté simplement par la donnée d'un couple de réels (les *bornes*). Cependant, l'ensemble peut ne pas admettre de plus petit (resp. plus grand) élément. On a alors un intervalle à bornes ouvertes. Pour tenir compte de cela, nous introduisons la notion de *bracket** à partir de laquelle nous définissons les bornes d'un intervalles réel :

Définition 2.1 (*Bracket*). Soit $\mathcal{L} = \{(, [\}$ (resp. $\mathcal{U} = \{),]\}$) l'ensemble des *brackets de gauche* (resp. *brackets de droite*). On définit l'ensemble $\mathcal{B}$ des *brackets* par $\mathcal{B} = \mathcal{L} \cup \mathcal{U}$ et on le munit de l'ordre $\prec$ [48] défini par : $) \prec [\prec] \prec (.$

*Pour éviter toute ambiguïté nous avons choisi de garder le mot anglais *bracket* plutôt que de le traduire par *parenthésage*.

Dans la suite, on se restreindra souvent à des intervalles fermés. Il est donc avantageux de définir une notation spéciale pour la restriction aux brackets fermées des ensemble ci-dessus : soient $\mathcal{L}_\square = \{]\}$, $\mathcal{U}_\square = \{[\}$ et $\mathcal{B}_\square = \mathcal{L}_\square \cup \mathcal{U}_\square$ ces sous-ensembles privilégiés.

On peut désormais définir une *borne d'un intervalle réel* comme étant la donnée d'un nombre réel et d'une *bracket* :

Définition 2.2 (Borne d'un intervalle réel). L'ensemble $\mathbb{R}^\diamond$ des bornes réelles est défini par :

$$\mathbb{R}^\diamond = \mathbb{R}^\triangleleft \cup \mathbb{R}^\triangleright \quad \text{avec} \quad \begin{cases} \mathbb{R}^\triangleleft & = (\mathbb{R} \times \mathcal{L} \cup \{\langle -\infty, (\,\rangle, \langle +\infty, (\,\rangle\}) \\ \mathbb{R}^\triangleright & = (\mathbb{R} \times \mathcal{U} \cup \{\langle -\infty,)\,\rangle, \langle +\infty,)\,\rangle\}) \end{cases}$$

Étant donnée une borne $\beta = \langle\, x, \alpha \,\rangle$, on définit les deux accesseurs : $\beta|_v = x$ et $\beta|_b = \alpha$. L'ensemble des bornes réelles $\mathbb{R}^\diamond$ peut être muni de l'ordre $\lhd$:

$$\forall \beta_1 = \langle\, \mathsf{r}, \alpha_1 \,\rangle, \beta_2 = \langle\, \mathsf{s}, \alpha_2 \,\rangle \in \mathbb{R}^\diamond : \qquad \beta_1 \lhd \beta_2 \iff (\mathsf{r} < \mathsf{s}) \vee (\mathsf{r} = \mathsf{s} \wedge \alpha_1 \prec \alpha_2) \tag{2.1}$$

Définition 2.3 (Intervalle réel). On appelle *intervalle réel* I tout connexe de $\mathbb{R}$ défini par la donné d'un couple constitué d'une borne réelle gauche et d'une borne réelle droite. On notera :

$$\begin{array}{lllll} a. & \big(\langle\, \mathsf{r}, [\,\rangle, \langle\, \mathsf{s},]\,\rangle\big) & \equiv [\mathsf{r} \,..\, \mathsf{s}] & \equiv \{\mathsf{t} \in \mathbb{R} \mid \mathsf{r} \leqslant \mathsf{t} \leqslant \mathsf{s}\} \\ b. & \big(\langle\, \mathsf{r}, [\,\rangle, \langle\, \mathsf{s},)\,\rangle\big) & \equiv [\mathsf{r} \,..\, \mathsf{s}) & \equiv \{\mathsf{t} \in \mathbb{R} \mid \mathsf{r} \leqslant \mathsf{t} < \mathsf{s}\} \\ c. & \big(\langle\, \mathsf{r}, (\,\rangle, \langle\, \mathsf{s},]\,\rangle\big) & \equiv (\mathsf{r} \,..\, \mathsf{s}] & \equiv \{\mathsf{t} \in \mathbb{R} \mid \mathsf{r} < \mathsf{t} \leqslant \mathsf{s}\} \\ d. & \big(\langle\, \mathsf{r}, (\,\rangle, \langle\, \mathsf{s},)\,\rangle\big) & \equiv (\mathsf{r} \,..\, \mathsf{s}) & \equiv \{\mathsf{t} \in \mathbb{R} \mid \mathsf{r} < \mathsf{t} < \mathsf{s}\} \end{array}$$

Note : Nous avons choisi d'utiliser la représentation des intervalles adoptée par KNUTH [136] plutôt que celle plus généralement employée dans la littérature anglo-saxonne ($[\mathsf{r}, \mathsf{s}]$, $[\mathsf{r}, \mathsf{s})$...) afin d'éviter les cas où le contexte ne permet pas de différencier un intervalle à bornes ouvertes (r, s) d'un couple de réels. De plus, ceci nous permet de parler d'un intervalle en terme de couple de bornes et d'écrire : $I = (\beta_1, \beta_2)$, où β_1 et β_2 sont des éléments de $\mathbb{R}^\diamond$.

Notation : Soient $\mathbb{I}^\mathbb{R}_\circ = \mathbb{R}^\triangleleft \times \mathbb{R}^\triangleright$ l'ensemble des intervalles à bornes réelles. Comme précédemment, on définit la restriction $\mathbb{I}^\mathbb{R}_\square$ de $\mathbb{I}^\mathbb{R}_\circ$ aux intervalles fermés (forme $a.$). Dans la suite de cette section, on simplifiera parfois les notations en écrivant respectivement $\mathbb{I}_\square$ et $\mathbb{I}_\circ$ pour $\mathbb{I}^\mathbb{R}_\square$ et $\mathbb{I}^\mathbb{R}_\circ$. Dans tous les cas où il n'y a pas ambiguïté sur l'ensemble considéré (ou bien lorsque la nature des bornes des intervalles est indifférente), on se permettra l'utilisation de la notation $\mathbb{I}$. De même, étant donné un intervalle $I = (\beta_1, \beta_2) \in \mathbb{I}$, on utilisera les deux raccourcis d'écriture $\underline{I} = \beta_1|_v$ et $\overline{I} = \beta_2|_v$.

L'ensemble $\mathbb{I}^\mathbb{R}_\circ$ constitue un semi-anneau d'ensembles [137, p. 31–36] où l'ensemble vide $\varnothing$ correspond à tout couple de borne $\langle \beta_1, \beta_2 \rangle$ tel que $\beta_2 \lhd \beta_1$. L'ensemble des unions d'intervalles réels correspondant à l'anneau minimal généré par $\mathbb{I}^\mathbb{R}_\circ$ est noté $\mathbb{U}^\mathbb{R}_\circ$. On note $\mathbb{U}^\mathbb{R}_\square$ la restriction de $\mathbb{U}^\mathbb{R}_\circ$ aux unions d'intervalles fermés. Dans la suite, on appellera indifféremment *domaine* D un intervalle I ou une union U d'intervalles.

La représentation d'un ensemble connexe de nombres réels en terme d'intervalle se fait de la façon suivante. Étant donné un ensemble non vide $\mathcal{S} \subseteq \mathbb{R}$, on peut lui associer l'intervalle $\langle \inf_\circ(\mathcal{S}), \sup_\circ(\mathcal{S}) \rangle$ avec $\inf_\circ(\mathcal{S}) \in \mathbb{R}^\triangleleft$ et $\sup_\circ(\mathcal{S}) \in \mathbb{R}^\triangleright$ définis par :

$$\inf_\circ(\mathcal{S}) = \left\{ \begin{array}{ll|l} \langle \inf(\mathcal{S}), [\,\rangle & \text{si } \inf(\mathcal{S}) \in \mathcal{S} & \text{si } \mathcal{S} \text{ a une borne inférieure} \\ \langle \inf(\mathcal{S}), (\,\rangle & \text{sinon} & \\ \langle -\infty, (\,\rangle & & \text{sinon} \end{array} \right.$$

$$\sup_\circ(\mathcal{S}) = \left\{ \begin{array}{ll|l} \langle \sup(\mathcal{S}),]\,\rangle & \text{si } \sup(\mathcal{S}) \in \mathcal{S} & \text{si } \mathcal{S} \text{ a une borne supérieure} \\ \langle \sup(\mathcal{S}),)\,\rangle & \text{sinon} & \\ \langle +\infty,)\,\rangle & & \text{sinon} \end{array} \right.$$

On remarquera que $\inf_\circ(\mathcal{S})$ (resp. $\sup_\circ(\mathcal{S})$) est la borne inférieure de $\mathcal{S} \times \mathcal{L}_\square$ sur $(\mathbb{R}^\triangleleft, \trianglelefteq)$ (resp. la borne supérieure de $\mathcal{S} \times \mathcal{U}_\square$ sur $(\mathbb{R}^\triangleright, \trianglelefteq)$).

2.1.1 Arithmétique des intervalles réels

Dans cette section, nous ne considérerons que l'ensemble des intervalles réels fermés. L'arithmétique des intervalles réels est définie de la façon suivante :

Définition 2.4 (Arithmétique des intervalles réels [167, 175]). Soient I_1 et I_2 deux intervalles de $\mathbb{I}_\square^\mathbb{R}$ et soient $\top$ et φ deux symboles d'opération avec $\top \in \{+, -, \times, /\}$ et $\varphi \in \{\mathrm{abs}, \mathrm{sqr}, \sqrt{}, \exp, \ln, \sin, \cos, \mathrm{atan}\}$. On définit alors les intervalles $I_3 = I_1 \top I_2$ et $I_4 = \varphi(I_1)$ par :

$$I_3 = \{x \top y \mid x \in I_1,\, y \in I_2\} \tag{2.2}$$

$$\text{et} \qquad I_4 = \{\varphi(x) \mid x \in I_1\} \tag{2.3}$$

Note : Dans son livre [167], MOORE définit la division de deux intervalles uniquement dans le cas où le diviseur ne contient pas 0. Cette convention a été souvent reprise dans la littérature. Il est cependant possible de s'affranchir de cette restriction, ce qui est hautement souhaitable dans le cas d'algorithmes étendus aux intervalles tels que la méthode de NEWTON-RAPHSON, où un cas intéressant est justement celui où la dérivée s'annule, et où l'on a donc besoin de pouvoir diviser par un intervalle contenant 0. On trouvera chez RATZ [198] une revue critique des différentes implémentations proposées pour un opérateur de division d'intervalles acceptant que le dénominateur contienne 0, et chez HICKEY *et al.* [106] un exemple d'implémentation efficace et prouvée correcte.

Nous aborderons dans la section 2.2 et au chapitre 11 les détails d'implantation de l'arithmétique des intervalles dont les bornes sont des flottants respectant la norme IEEE 754 [116]. Nous nous restreindrons dans la suite de cette section à présenter les opérateurs d'intervalles sur $\mathbb{I}_\square^\mathbb{R}$ associés aux fonctions de base.

En utilisant les propriétés de monotonie, on montre simplement [167] que les opérations élémentaires peuvent se réécrire en terme des bornes des intervalles par :

$$\begin{aligned}
[\mathsf{a} \,..\, \mathsf{b}] + [\mathsf{c} \,..\, \mathsf{d}] &= [\mathsf{a} + \mathsf{c} \,..\, \mathsf{b} + \mathsf{d}] \\
[\mathsf{a} \,..\, \mathsf{b}] - [\mathsf{c} \,..\, \mathsf{d}] &= [\mathsf{a} - \mathsf{d} \,..\, \mathsf{b} - \mathsf{c}] \\
[\mathsf{a} \,..\, \mathsf{b}] \times [\mathsf{c} \,..\, \mathsf{d}] &= [\min(\mathsf{ac}, \mathsf{ad}, \mathsf{bc}, \mathsf{bd}) \,..\, \max(\mathsf{ac}, \mathsf{ad}, \mathsf{bc}, \mathsf{bd})] \\
[\mathsf{a} \,..\, \mathsf{b}]/[\mathsf{c} \,..\, \mathsf{d}] &= [\min(\mathsf{a/c}, \mathsf{a/d}, \mathsf{b/c}, \mathsf{b/d}) \,..\, \max(\mathsf{a/c}, \mathsf{a/d}, \mathsf{b/c}, \mathsf{b/d})], \quad 0 \notin [c \,..\, d]
\end{aligned} \tag{2.4}$$

À partir des formules ci-dessus, on découvre l'une des principales faiblesses de l'arithmétique qui l'a fait rejeter par nombres de numériciens dont WILKINSON [250] : dans une expression d'intervalles, les différentes occurrences d'une variable sont considérées comme autant de variables différentes. C'est le *problème de dépendance* que l'on peut illustrer par l'exemple simple suivant : soit X une variable dont la valeur est l'intervalle $[4 \,..\, 6]$. En appliquant la définition de la soustraction donnée ci-dessus, on obtient $X - X = [4 \,..\, 6] - [4 \,..\, 6] = [-2 \,..\, 2]$ et non $[0 \,..\, 0]$ comme on le souhaiterait. En fait, comme le font remarquer WALSTER et HANSEN [244], on ne calcule pas

$$X - X = \{\mathsf{r} - \mathsf{r} \mid \mathsf{r} \in X\}$$

mais

$$X - X = \{\mathsf{r} - \mathsf{s} \mid \mathsf{r} \in X \wedge \mathsf{s} \in X\}$$

Les couples $(\mathbb{I}_\circ^\mathbb{R}, +)$ et $(\mathbb{I}_\circ^\mathbb{R}, \times)$ ne sont que des semi-groupes commutatifs. En effet, un intervalle $I = [\mathsf{r}..\mathsf{s}] \in \mathbb{I}_\circ^\mathbb{R}$ n'a pas en général d'inverse pour l'addition et la multiplication (en particulier, si $\mathsf{r} \neq \mathsf{s}$, on a $I + (-I) = [\mathsf{r} - \mathsf{s} \,..\, \mathsf{s} - \mathsf{r}] \neq 0$). Nous verrons dans la section 5.2.2 qu'il est possible de créer des groupes à partir de $(\mathbb{I}_\circ^\mathbb{R}, +)$ et $(\mathbb{I}_\circ^\mathbb{R}, \times)$, obtenant ainsi l'arithmétique d'intervalles de KAUCHER [129] ou celle de MARKOV [156].

D'une manière générale, il est possible de définir une fonction F s'appliquant à des intervalles pour chaque fonction réelle f. On dira que F est une *extension aux intervalles* de f. On verra à la section 2.1.4 que l'arithmétique des intervalles ne possède pas toutes les propriétés de l'arithmétique réelle. En particulier, deux expressions réelles équivalentes ne le sont plus forcément lorsqu'elles sont évaluées dans l'arithmétique des intervalles. On en déduit alors que pour une même expression réelle, il existe une famille d'extensions aux intervalles dont la présentation est l'objet de la section suivante.

2.1.2 Extensions aux intervalles de fonctions réelles

Nous allons introduire quelques notations avant de nous intéresser à la notion fondamentale d'extension aux intervalles d'une fonction réelle.

Notations : Les ensembles quelconques sont représentés par des lettres majuscules calligraphiées (*e.g.* $\mathcal{S}$). L'ensemble des parties d'un ensemble $\mathcal{S}$ est noté $\mathcal{P}(\mathcal{S})$. On note en gras un produit cartésien ou un vecteur. Étant donné un terme t et une variable v, on note $\mathsf{Var}(t)$ l'ensemble des variables de t et $\mu_t(v)$ la multiplicité (nombre d'occurrences) de v dans t. Par abus de notation, soit $\mu(t)$ le produit des multiplicités de l'ensemble des variables apparaissant dans t[†]. On appellera *pavé* $\boldsymbol{B}$ un produit cartésien n-aire d'intervalles : $\boldsymbol{B} = I_1 \times \cdots \times I_n$. Étant donnée une fonction réelle $f\colon \mathbb{R}^n \to \mathbb{R}$, soit $\mathcal{D}_f$ le domaine de définition de f et $\mathcal{D}_f^{\mathbb{I}}$ le sous-ensemble de $\mathbb{I}^n$ défini par : $\mathcal{D}_f^{\mathbb{I}} = \{\boldsymbol{B} \in \mathbb{I}^n \mid \boldsymbol{B} \subseteq \mathcal{D}_f\}$. Étant donnée une relation réelle n-aire ρ et un entier $k \in \{1, \ldots, n\}$, soit $\pi_k(\rho) = \{\mathsf{r}_k \in \mathbb{R} \mid \exists \mathsf{r}_1, \ldots, \exists \mathsf{r}_n \in \mathbb{R}$ tel que $(\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \rho\}$ la k-ième projection de ρ. Étant donnés une variable v, un intervalle I et deux pavés $\boldsymbol{B}$ et $\boldsymbol{D}$, on note $\mathsf{Dom}_{\boldsymbol{B}}(v)$ le domaine de v dans le pavé $\boldsymbol{B}$, $\boldsymbol{B}|_k = I_k$ la k-ième projection du pavé $\boldsymbol{B}$ et $\boldsymbol{B}|_{v,\boldsymbol{D}}$ (resp. $\boldsymbol{B}|_{v,I}$) le pavé obtenu en remplaçant le domaine de v dans le pavé $\boldsymbol{B}$ par son domaine dans le pavé $\boldsymbol{D}$ (resp. par l'intervalle I). Étant donné un pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n$, un entier $j \in \{1, \ldots, n\}$ et un intervalle I', soit $\boldsymbol{B}|_{I_j,I'}$, le pavé $\boldsymbol{B} = I_1 \times \cdots \times I_{j-1} \times I' \times I_{j+1} \times \cdots \times I_n$.

Afin de modéliser par la suite le passage du continu au discret induit par l'utilisation des nombres flottants pour représenter les réels en machine, nous utiliserons la notion de *domaine d'approximation* introduite par BENHAMOU et OLDER :

Définition 2.5 (Domaine d'approximation [30]). Un *domaine d'approximation* $\mathcal{A}$ sur l'ensemble des réels $\mathbb{R}$ est un sous-ensemble de l'ensemble des parties $\mathcal{P}(\mathbb{R})$ de $\mathbb{R}$, pour lequel l'inclusion induit un ordre bien fondé et qui a les propriétés suivantes :

$$\{\mathbb{R}\} \in \mathcal{A}$$
$$\forall S_1, S_2 \in \mathcal{A}\colon S_1 \cap S_2 \in \mathcal{A} \qquad \text{(Clôture par intersection)}$$

La définition ci-dessus est moins générale que la définition originale mais suffira à nos besoins. On peut alors définir l'approximation d'une relation réelle ρ par rapport à un domaine d'approximation $\mathcal{A}$:

Définition 2.6 (Fonction d'approximation). Étant donné un domaine d'approximation $\mathcal{A}$, la *fonction d'approximation de* $\mathcal{A}$, notée $\mathrm{apx}_{\mathcal{A}}(\rho)$, est la fonction qui à toute relation réelle $\rho \subseteq \mathbb{R}$ associe le plus petit élément (au sens de l'inclusion) de $\mathcal{A}$ contenant ρ :

$$\begin{aligned} \mathrm{apx}_{\mathcal{A}}(\rho) \quad &: \mathcal{P}(\mathbb{R}) \to \mathcal{A} \\ &\rho \mapsto \bigcap \{S \in \mathcal{A} \mid \rho \subseteq S\} \end{aligned}$$

Étant donnée une relation réelle ρ, nous utiliserons essentiellement dans ce document deux types d'approximations : l'approximation par la plus petite (au sens de l'inclusion ensembliste) union de pavés $\mathsf{Union}(\rho) = \mathrm{apx}_{\mathbb{U}}(\rho)$ ou par le plus petit pavé $\mathsf{Hull}(\rho) = \mathrm{apx}_{\mathbb{I}}(\rho)$ (*cf.* fig. 2.1).

Étant donnée une fonction réelle $f\colon \mathbb{R}^n \to \mathbb{R}$, on va s'intéresser aux fonctions sur les intervalles bornant le domaine de variation de f. Il en existe bien sûr une infinité, que l'on peut classer suivant un critère qualitatif simple correspondant à la précision avec laquelle elles encadrent le domaine de variation de f. En particulier, du fait de la non-distributivité de l'arithmétique des intervalles, une simple factorisation de l'expression de f peut permettre d'obtenir une fonction d'intervalles plus précise.

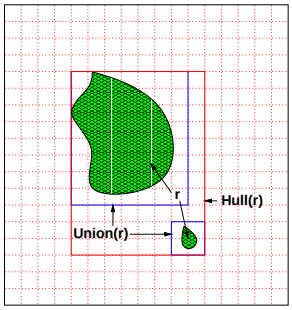


FIG. 2.1: $\mathsf{Hull}(\rho)$ *vs.* $\mathsf{Union}(\rho)$

Notation : Soit $f\colon \mathbb{R}^n \to \mathbb{R}$ une fonction réelle et $\boldsymbol{B} \in \mathbb{I}^n$ un pavé d'intervalles. On note :

$$f(\boldsymbol{B}) = \{f(\mathbf{r}) \mid \mathbf{r} \in \boldsymbol{B}\}$$

[†]Ce raccourci d'écriture nous permettra d'écrire « $\mu(t)$=1 » pour indiquer que toutes les variables de t ont une seule occurrence.

le domaine de variation de f sur $\boldsymbol{B}$. Ce domaine est un intervalle dès lors que f est une fonction continue [218, p. 13].
Étant donné un intervalle $I \in \mathbb{I}$, soit $\mathsf{w}(I) = \overline{I} - \underline{I}$ la taille de I et $\mathsf{mid}(I) = (\underline{I} + \overline{I})/2$ le centre de I. La taille d'un pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n$ est définie comme le maximum des tailles de ses projections :

$$\mathsf{w}(\boldsymbol{B}) = \max_{j \in \{1,\ldots,n\}} \big\{\mathsf{w}(I_j)\big\}$$

et son centre en est le vecteur des centres :

$$\mathsf{mid}(\boldsymbol{B}) = (\mathsf{mid}(I_1), \ldots, \mathsf{mid}(I_n))$$

Dans la suite, nous utiliserons implicitement le morphisme κ de $\mathbb{R}$ dans $\mathbb{I}$ défini par :

$$\begin{aligned} \kappa\colon \quad & \mathbb{R} \to \mathbb{I} \\ & \mathsf{r} \mapsto [\mathsf{r} \,..\, \mathsf{r}] \end{aligned}$$

afin de raccourcir l'écriture de $F(\kappa(\mathsf{r}))$ en $F(\mathsf{r})$.

Définition 2.7 (Extension aux intervalles [175]). Étant donnée une fonction réelle $f\colon \mathbb{R}^n \to \mathbb{R}$, une *extension aux intervalles* $F\colon \mathbb{I}^n \to \mathbb{I}$ de f est une fonction vérifiant :

$$f(\mathbf{r}) = F(\mathbf{r}) \qquad \forall \mathsf{r} \in \mathcal{D}_f \tag{2.5}$$

$$\mathsf{Hull}(f(\boldsymbol{B})) \subseteq F(\boldsymbol{B}) \qquad \forall \boldsymbol{B} \in \mathcal{D}_f^{\mathbb{I}} \tag{2.6}$$

Si la fonction d'intervalles F ne vérifie que la propriété (2.6), on dira que c'est une *extension aux intervalles faible* de f. En pratique, on aura souvent recours à des extensions faibles du fait des calculs en nombres flottants. Les extensions aux intervalles, faibles ou non, sont aussi appelées des *fonctions d'inclusion* [197].

Une mesure qualitative des fonctions d'inclusion est leur *ordre de convergence*, notion introduite par MOORE :

Définition 2.8 (Ordre de convergence d'une fonction d'inclusion [167]). Soit $f\colon \mathcal{D}_f \subseteq \mathbb{R}^n \to \mathbb{R}$ une fonction réelle et soit F une fonction d'inclusion pour f. La fonction F est dite d'ordre de convergence α s'il existe une constante réelle $c \geqslant 0$ telle que :

$$\mathsf{w}(F(\boldsymbol{B})) - \mathsf{w}(\mathsf{Hull}(f(\boldsymbol{B})) \leqslant c \cdot \mathsf{w}(\boldsymbol{B})^{\alpha} \quad \forall \boldsymbol{B} \in \mathcal{D}_f^{\mathbb{I}}$$

Une propriété importante pour les extensions aux intervalles est la *monotonie pour l'inclusion*. On a la définition suivante :

Définition 2.9 (Fonction d'inclusion monotone). La fonction d'intervalles $F\colon \mathbb{I}^n \to \mathbb{I}$ est dite *monotone pour l'inclusion* si elle vérifie la propriété :

$$\forall \boldsymbol{B}, \boldsymbol{D} \in \mathbb{I}^n\colon \quad \boldsymbol{B} \subseteq \boldsymbol{D} \Rightarrow F(\boldsymbol{B}) \subseteq F(\boldsymbol{D}) \tag{2.7}$$

La monotonie pour l'inclusion d'une extension aux intervalles facilite les preuves de convergence pour les algorithmes. Elle assure aussi que la bisection d'un pavé ne peut augmenter la surestimation du domaine de la fonction réelle étudiée. Ainsi, étant données une fonction f, une extension monotone pour l'inclusion F de f et un pavé $\boldsymbol{B}$ découpé en k pavé $\boldsymbol{B_1}$,..., $\boldsymbol{B_k}$, on a :

$$F(\boldsymbol{B_1}) \cup \cdots \cup F(\boldsymbol{B_k}) \subseteq F(\boldsymbol{B})$$

2.1.2.1 Extension naturelle aux intervalles

L'*extension naturelle aux intervalles* d'une opération arithmétique réelle $\lozenge$ est une opération sur les intervalles $\blacklozenge$ telle que pour tous $I_1, \ldots, I_n \in \mathbb{I}$ on a $\blacklozenge(I_1, \ldots, I_n) = \mathsf{Hull}(\{\lozenge(x_1, \ldots, x_n) \mid x_1 \in I_1, \ldots, x_n \in I_n\})$. L'extension naturelle aux intervalles d'une fonction $f\colon \mathbb{R}^n \to \mathbb{R}$ — dite aussi *forme naturelle* — est alors définie comme étant la fonction $F\colon \mathbb{I}^n \to \mathbb{I}$ obtenue à partir de f en remplaçant chaque constante r par $\mathsf{Hull}(\{\mathsf{r}\})$, chaque

variable réelle par une variable d'intervalle et chaque opération arithmétique par son extension naturelle aux intervalles. Les extensions naturelles sont d'ordre 1 pour la convergence. Comme on va le voir, il est aisé de définir des extensions d'ordre supérieur. Cependant, les extensions naturelles méritent d'être étudiées car elles servent de base pour des extensions plus sophistiquées et leur ordre de convergence les rend souvent plus efficaces que les extensions d'ordres supérieur lorsque les domaines des variables sont grands. On notera que l'extension naturelle d'une fonction réelle f est indissociable de la forme textuelle de f. Ainsi, on devrait en toute rigueur parler de l'extension naturelle de f pour une expression donnée de cette fonction (par exemple, la fonction $f(x) = x^2 + x = x(x+1)$ possède au moins deux extensions naturelles : $F_1(X) = X^2 + X$ et $F_2(X) = X(X+1)$).

On a le théorème important suivant :

Théorème 2.1 (Occurrences simples [167]). *Soit $f\colon \mathbb{R}^n \to \mathbb{R}$ une fonction réelle telle que $\mu(f) = 1$ et soit $F\colon \mathbb{I}^n_\circ \to \mathbb{I}_\circ$ l'extension naturelle aux intervalles de f. On a :*

$$F(X_1, \ldots, X_n) = \{f(x_1, \ldots, x_n) \mid x_i \in X_i, i = 1, \ldots, n\}$$

Ainsi, une fonction f ne possédant que des occurrences simples de variables a la propriété très importante que son extension naturelle permet d'obtenir l'évaluation la plus fine possible pour un pavé donné, à savoir le domaine de variation de f sur le pavé.

Chez MOORE, le théorème 2.1 n'est que le corollaire d'un théorème, appelé *théorème fondamental de l'arithmétique des intervalles* :

Théorème 2.2 (Théorème fondamental de l'arithmétique des intervalles [167]). *Soit $F(X_1, \ldots, X_n)$ une expression rationnelle d'intervalles n-aire. On a :*

$$\forall X'_1, \ldots, X'_n \in \mathbb{I}\colon \quad X'_1 \subseteq X_1, \ldots, X'_n \subseteq X_n \quad \Rightarrow \quad F(X'_1, \ldots, X'_n) \subseteq F(X_1, \ldots, X_n)$$

Lors de l'extension aux intervalles de fonctions réelles, il est intéressant de réduire au maximum le nombre d'occurrences multiples des variables par des manipulations symboliques bien choisies (factorisations, mise sous forme de HORNER dans les cas de fonctions polynomiales…). Lorsque cela n'est pas possible, il reste toujours la possibilité d'obtenir des résultats de meilleure qualité en ayant recours à d'autres extensions que l'extension naturelle.

Parmi celles-ci, on peut citer les *formes centrées* [167] telles que les formes de TAYLOR [10], ou les extensions en forme de BERNSTEIN [79]. On trouvera dans la thèse de STAHL [218] une présentation approfondie de ces extensions. Nous nous contenterons donc de donner ici une brève description des formes centrées et des formes imbriquées (*nested form*) ; nous considérerons la forme de BERNSTEIN dans la section 5.2.1.1 de la partie III.

Exemple 2.5 (Différentes extensions aux intervalles). *On souhaite évaluer le domaine de variation de la fonction $f(x) = x(x+1)$ pour $x \in [-1\,..\,1]$. En effectuant quelques manipulations symboliques très simples sur l'expression de f, on obtient les extensions aux intervalles suivantes (*cf. *fig. 2.2) :*

$$\begin{array}{lll} F_1(x) & = x(x+1) & = [-2\,..\,2] \\ F_2(x) & = xx + x & = [-2\,..\,2] \\ F_3(x) & = x^2 + x & = [-1\,..\,2] \\ F_4(x) & = (x+1/2)^2 - 1/4 & = [-1/4\,..\,2] \end{array}$$

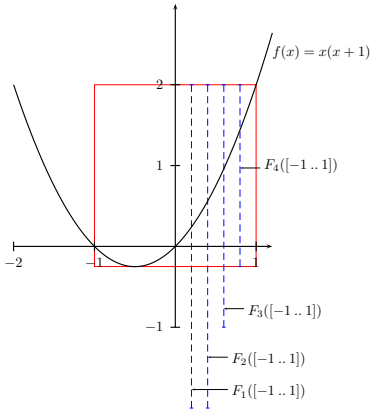


FIG. 2.2: Différentes fonctions d'inclusion pour f

2.1.2.2 Formes centrées et formes de TAYLOR

Nous donnons ci-dessous la définition la plus générale d'une forme centrée, puis nous instancierons cette définition aux cas particuliers des *formes moyennes* (*mean value forms*) et des formes de TAYLOR.

Commençons par rappeler la notion de fonction lipschitzienne appliquée aux fonctions d'intervalles :

Définition 2.10 (Fonction d'intervalles lipschitzienne). Une fonction $F\colon \mathbb{I}^n \to \mathbb{I}$ est dite *lipschitzienne sur le pavé* $\boldsymbol{B} \subseteq \mathbb{I}^n$ s'il existe un nombre réel positif $\mathsf{r}_{\boldsymbol{B}}$ tel que pour tout pavé $\boldsymbol{D} \subseteq \boldsymbol{B}$ on a :

$$\mathsf{w}(F(\boldsymbol{D})) \leqslant \mathsf{r}_{\boldsymbol{B}} \cdot \mathsf{w}(\boldsymbol{D}) \tag{2.8}$$

La fonction F est dite *fonction lipschitzienne* s'il existe un réel positif $\mathsf{r}_{\boldsymbol{B}}$ tel que la relation (2.8) est vérifiée pour tout pavé $\boldsymbol{B}$.

Définition 2.11 (Forme centrée [167, 218]). Soit $f\colon \mathbb{R}^n \to \mathbb{R}$ une fonction réelle continue et soit $\boldsymbol{g}\colon \mathbb{R}^{2n} \to \mathbb{R}^n$ une fonction permettant d'exprimer f sous la forme développée :

$$f(\boldsymbol{x}) = f(\boldsymbol{c}) + {}^t(\boldsymbol{x} - \boldsymbol{c})\boldsymbol{g}(\boldsymbol{x}, \boldsymbol{c}), \quad \forall \boldsymbol{x}, \boldsymbol{c} \in \mathbb{R}^n$$

Définissons aussi une fonction d'intervalles $\boldsymbol{C}\colon \mathbb{I}^n \to \mathbb{R}^n$ telle que $\boldsymbol{C}(\boldsymbol{B}) \in \boldsymbol{B}, \forall \boldsymbol{B} \in \mathbb{I}^n$. La fonction $\boldsymbol{C}$ est un opérateur de sélection retournant un réel appartenant à l'intervalle donné en argument.

Enfin, donnons-nous la fonction d'inclusion $\boldsymbol{G} = (G_1, \ldots, G_n)\colon \mathbb{I}^n \to \mathbb{I}^n$, où pour tout $i \in \{1, \ldots, n\}$, $G_i\colon \mathbb{I} \to \mathbb{I}$ est une fonction lipschitzienne, et posons :

$$\forall \boldsymbol{B} \in \mathbb{I}^n, \forall \mathbf{r} \in \boldsymbol{B}\colon \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{C}(\boldsymbol{B})) \in \boldsymbol{G}(\boldsymbol{B})$$

On appelle alors *forme centrée* de f la fonction d'intervalles $F\colon \mathbb{I}^n \to \mathbb{I}$ définie par :

$$F(\boldsymbol{X}) = f(\boldsymbol{C}(\boldsymbol{X})) + {}^t(\boldsymbol{X} - \boldsymbol{C}(\boldsymbol{X}))\boldsymbol{G}(\boldsymbol{X}) \tag{2.9}$$

Note : Afin de préserver la correction des calculs, on n'évaluera pas en pratique l'expression réelle $f(\boldsymbol{C}(\boldsymbol{X}))$ de l'équation (2.9) mais l'expression d'intervalles $\phi(\boldsymbol{C}(\boldsymbol{X}))$ où ϕ est une fonction d'inclusion (par exemple sa forme naturelle) pour f. On remarquera qu'il est a priori inutile d'utiliser une fonction d'inclusion pour $\boldsymbol{C}$ puisque c'est une fonction de sélection d'un élément de $\boldsymbol{X}$ qui peut se contenter de retourner, par exemple, une des bornes de l'intervalles.

Si l'on parle de formes centrées au pluriel, c'est parce qu'il en existe autant que de façons de choisir un élément de $\boldsymbol{X}$ (choix modélisé ici par celui de la fonction $\boldsymbol{C}$). Un choix évident est d'utiliser le centre de $\boldsymbol{X}$. On trouvera cependant chez BAUMANN [19] l'utilisation d'une fonction $\boldsymbol{C}$ retournant un point optimal pour le développement de f. HANSEN [99] a prouvé que les formes centrées sont d'ordre 2 pour la convergence.

À partir du développement de TAYLOR d'une fonction $f\colon \mathbb{R}^n \to \mathbb{R}$, il est possible de définir une famille d'extension aux intervalles de f dites *formes de* TAYLOR.

Définition 2.12 (Formes de TAYLOR [167]). Soit $f\colon \mathbb{R} \to \mathbb{R}$ une fonction m fois dérivable sur un pavé $\boldsymbol{X}$ et soit $\boldsymbol{C}\colon \mathbb{I}^n \to \mathbb{R}^n$ une fonction définie de la même façon que dans la définition 2.10. On définit les *formes générales de* TAYLOR *de niveau* m de la fonction f par :

$$F(\boldsymbol{X}) = f(\boldsymbol{C}(\boldsymbol{X})) + \sum_{j=1}^{m-1} \frac{1}{j!} f^{[j]}(\boldsymbol{C}(\boldsymbol{X}))\, {}^t(\boldsymbol{X} - \boldsymbol{C}(\boldsymbol{X}))^j + \frac{1}{m!} F^{[m]}(\boldsymbol{X})\, {}^t(\boldsymbol{X} - \boldsymbol{C}(\boldsymbol{X}))^m \tag{2.10}$$

avec les notations :

$$f^{[j]}(\boldsymbol{y}) = \left(\frac{\partial^j f}{\partial {y_1}^j}(\boldsymbol{y}), \ldots, \frac{\partial^j f}{\partial {y_n}^j}(\boldsymbol{y})\right), \quad \forall \boldsymbol{y} = (y_1, \ldots, y_n) \in \mathbb{R}^n$$

et

$$F^{[m]}(\boldsymbol{Y}) = \left(\frac{\partial^m F}{\partial {Y_1}^m}(\boldsymbol{Y}), \ldots, \frac{\partial^m F}{\partial {Y_n}^m}(\boldsymbol{Y})\right), \quad \forall \boldsymbol{Y} = (Y_1, \ldots, Y_n) \in \mathbb{I}^n$$

où $F^{[m]}\colon \mathbb{I}^n \to \mathbb{I}^n$ est une fonction d'inclusion pour la fonction $f^{[m]} = \left(\frac{\partial^m f}{\partial {x_1}^m}, \ldots, \frac{\partial^m f}{\partial {x_n}^m}\right)\colon \mathbb{R}^n \to \mathbb{R}^n$.

On se reportera, entre autres, à la thèse de STAHL [218] pour la preuve que les formes centrées et en particulier les formes de TAYLOR sont bien des fonctions d'inclusion pour f.

La forme de TAYLOR de niveau 1 porte aussi le nom de *forme moyenne* (*mean value form*) car on peut l'obtenir en utilisant le développement de la fonction f en se basant sur le théorème des accroissements finis (*mean value theorem*) :

$$F(\boldsymbol{X}) = f(\boldsymbol{C}(\boldsymbol{X})) + \nabla F(\boldsymbol{X})\ {}^t(\boldsymbol{X} - \boldsymbol{C}(\boldsymbol{X}))$$

où ∇F est une fonction d'inclusion pour le gradient de f.

Le choix de l'extension à utiliser dépend de plusieurs critères dont la forme de la fonction réelle de départ et la taille des domaines des variables. D'après STAHL [218], la forme de BERNSTEIN (*cf.* exemple p. 70) est, dans le cas de fonctions polynomiales, l'extension qui donne les résultats les plus précis. Elle a cependant l'inconvénient d'être relativement coûteuse à calculer. Les formes centrées sont plus précises que l'extension naturelle lorsque les domaines des variables ne sont pas trop grands (RATSCHEK et ROKNE [197] suggèrent de ne pas utiliser les formes centrées lorsque le plus grand domaine des variables de la fonction a une taille supérieure à $1/(2n)$ où n est le nombre de variables). Sinon, c'est l'extension naturelle qui est la plus intéressante. D'un point de vue algorithmique, il semble donc nécessaire d'être capable de changer d'extension au cours du temps afin d'optimiser les calculs. Ainsi, dans DecLIC (*cf.* chap. 8), l'algorithme de NEWTON-RAPHSON de recherche de zéros utilise à la fois une forme de TAYLOR et l'extension naturelle des fonctions considérées.

2.1.2.3 Formes imbriquées

La forme imbriquée [218] pour un polynôme réel p correspond à sa représentation dans la forme la plus factorisée possible. En dimension 1, on retrouve la *forme de* HORNER.

Exemple 2.6 (Forme imbriquée). *Étant donné le polynôme $p(x_1, x_2) = x_1^2x_2^2 + x_1x_2^2 + x_1x_2$, l'une de ses formes imbriquées est le polynôme $P_n(x_1, x_2) = x_1x_2(x_2(x_1 + 1) + 1)$*

On remarque que la forme imbriquée d'un polynôme à plusieurs variables n'est pas unique. On trouvera chez STAHL [218] des heuristiques permettant de décider laquelle choisir. Bien que tirant parti de la propriété de sous-distributivité des intervalles (*cf.* section 2.1.4), l'utilisation de la forme imbriquée n'améliore pas toujours, en pratique, la précision de l'évaluation. En effet, la suite de multiplications qui en résulte induit plus d'erreurs d'arrondi que le calcul des puissances.

2.1.2.4 Comparaison de quelques extensions

Étant donné la fonction $f(x) = (x^6/4 - 1{,}9x^5 + 0{,}25x^4 + 20x^3 - 17x^2 - 55{,}6x + 48)/50$, nous allons montrer graphiquement l'impact du choix de l'extension aux intervalles utilisée pour connaître son domaine de variation pour $x \in [-3\,..\,5.5]$.

La figure 2.3 montre le profil obtenu lorsque l'on évalue f en forme naturelle après avoir découpé le domaine de x en intervalles de taille $\mathsf{w}(I) = 0{,}125$. On peut noter que la fonction f ne possède pas de zéro pour $x \in [-3..0]$. Cependant, il y a un *quasi-zéro* dans l'intervalle $[-3..-2]$: à cet endroit, la fonction prend des valeurs très proches de zéro ; l'évaluation en forme naturelle, même sur des intervalles de petite taille, empêche de différencier ce quasi-zéro d'un vrai zéro. On en verra les conséquences dans la section 3.3 lorsque nous parlerons de box-consistance. On peut remarquer que la précision de l'évaluation est très mauvaise dans l'intervalle $[4..5.5]$ et qu'une « divergence » semble se produire lorsque l'on se rapproche de la borne droite du domaine de x. Un tel comportement est de nature à ralentir très fortement les algorithmes de résolution de contraintes d'intervalles que nous verrons dans la suite, car ils devront découper très finement les intervalles pour avoir des estimations du domaine de variation de bonne qualité.

La figure 2.4 présente l'évaluation de f en forme naturelle pour des intervalles de taille $\mathsf{w}(I) = 0{,}02$. On constate encore une fois la mauvaise qualité de l'évaluation dans la partie droite du domaine (avec cependant l'absence de la divergence notée plus haut). L'évaluation est par contre très bonne aux alentours de son centre.

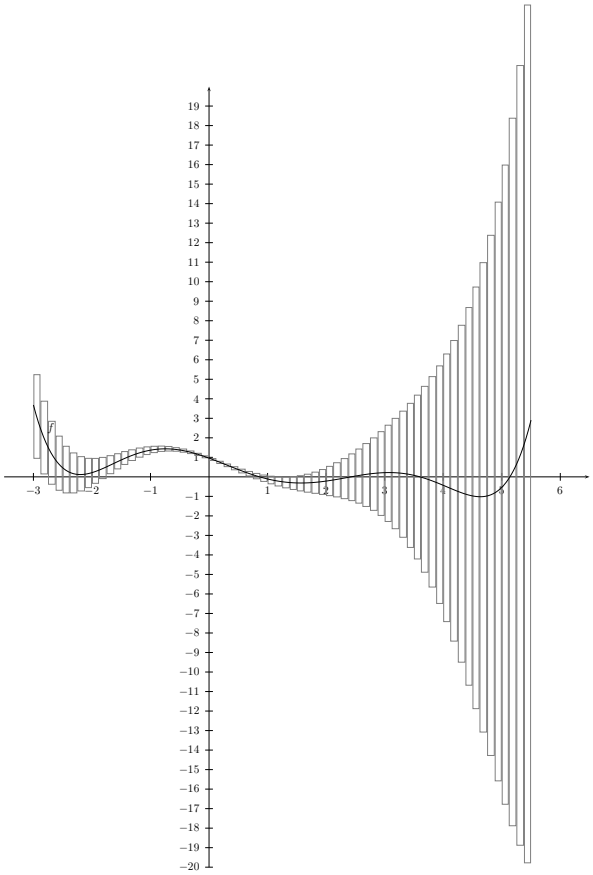


FIG. 2.3: Évaluation de $f(x) = (x^6/4 - 1{,}9x^5 + 0{,}25x^4 + 20x^3 - 17x^2 - 55{,}6x + 48)/50$ en forme naturelle ($\mathsf{w}(I) = 0{,}125$)

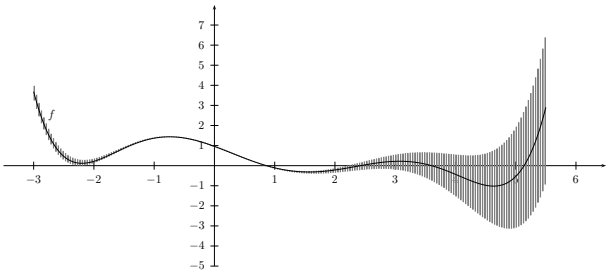


FIG. 2.4: Évaluation de $f(x) = (x^6/4 - 1{,}9x^5 + 0{,}25x^4 + 20x^3 - 17x^2 - 55{,}6x + 48)/50$ en forme naturelle ($\mathsf{w}(I) = 0{,}02$)

La figure 2.5 décrit l'évaluation de f avec la forme de HORNER $H_f(X) = (((((((X/4) - 1{,}9)X + 0{,}25)X + 20)X - 17)X - 55{,}6)X + 48)/50$. Si la précision apparaît toujours mauvaise dans l'intervalle $[4\ ..\ 5.5]$, elle est cependant sans commune mesure avec celle obtenue pour la même taille d'intervalle en forme naturelle. En particulier, on n'a pas ici la dégradation brutale de la précision observée sur la droite du domaine dans la figure 2.3.

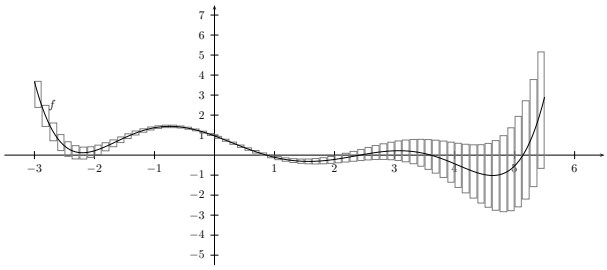


FIG. 2.5: Évaluation de $f(x) = (x^6/4 - 1{,}9x^5 + 0{,}25x^4 + 20x^3 - 17x^2 - 55{,}6x + 48)/50$ en forme de HORNER ($\mathsf{w}(I) = 0{,}125$)

La figure 5.4 p. 71 présente l'évaluation en forme de BERNSTEIN. On discutera du résultat obtenu dans la section 5.2.1.1.

2.1.3 Extension aux intervalles de relations réelles

Les relations réelles admettent, comme les fonctions réelles, des extensions aux intervalles. Suivant l'application visée, différentes extensions sont possibles. Dans la suite, nous utiliserons presque exclusivement la définition suivante :

Définition 2.13 (Extension aux intervalles d'une relation réelle). Étant donnée une relation n-aire réelle $\rho_{\mathbb{R}}$, une extension aux intervalles de $\rho_{\mathbb{R}}$ est un sous-ensemble $\rho_{\mathbb{I}}$ de $\mathbb{I}^n$ défini par :

$$\rho_{\mathbb{I}} = \{(I_1, \ldots, I_n) \in \mathbb{I}^n \mid \exists a_1 \in I_1, \ldots \exists a_n \in I_n \colon (a_1, \ldots, a_n) \in \rho_{\mathbb{R}}\} \tag{2.11}$$

On trouve dans la spécification BIAS (*Basic Interval Arithmetic Specification*) de CHIRIAEV et WALSTER [44] trois types d'extensions de relations :

1. les *extensions ensemblistes* (*set relations*), que nous utiliserons dans la section 5.2 pour le calcul de *solutions algébriques* ;
2. les *extensions certaines* (*certainly relations*) définies de la façon suivante : $\rho_{\mathbb{I}}$ est une extension certaine aux intervalles de la relation $\rho_{\mathbb{R}}$ si :

$$\rho_{\mathbb{I}} = \{(I_1, \ldots, I_n) \in \mathbb{I}^n \mid \forall a_1 \in I_1, \ldots \forall a_n \in I_n \colon (a_1, \ldots, a_n) \in \rho_{\mathbb{R}}\}$$

3. les *extensions possibles* (*possible relations*) de la définition 2.13.

Exemple 2.7 (Différentes extensions de relations). *Nous allons donner l'extension aux intervalles des relations d'égalité ($=$), d'inégalité ($\neq$) et d'infériorité stricte ($<$). Soient I_1 et I_2 deux intervalles non vides de $\mathbb{I}$. On a :*

Relation/Extension	*ensembliste*	*certaine*	*possible*
$I_1 = I_2$	$\underline{I_1} = \underline{I_2} \wedge \overline{I_1} = \overline{I_2}$	$\overline{I_1} \leqslant \underline{I_2} \wedge \underline{I_1} \geqslant \overline{I_2}$	$\underline{I_1} \leqslant \overline{I_2} \wedge \overline{I_1} \geqslant \underline{I_2}$
$I_1 \neq I_2$	$\underline{I_1} \neq \underline{I_2} \vee \overline{I_1} \neq \overline{I_2}$	$\underline{I_1} > \overline{I_2} \vee \underline{I_2} > \overline{I_1}$	$\overline{I_1} > \underline{I_2} \vee \underline{I_1} < \overline{I_2}$
$I_1 < I_2$	$\underline{I_1} < \underline{I_2} \wedge \overline{I_1} < \overline{I_2}$	$\overline{I_1} < \underline{I_2}$	$\underline{I_1} < \overline{I_2}$

2.1.4 Propriétés

L'arithmétique des intervalles réels ne respecte pas toutes les lois de l'arithmétique réelle. En particulier, on n'a pas la distributivité de la multiplication par rapport à l'addition, mais une propriété moins forte appelée *sous-distributivité*. Ainsi, soient I et J et K trois intervalles réels. Il vient [167, 175] :

$$\begin{array}{rll} I + J = J + I & I \times J = J \times I & \text{(commutativité)} \\ (I + J) \pm K = I + (J \pm K) & (I \times J) \times K = I \times (J \times K) & \text{(associativité)} \\ I + [0 \,..\, 0] = [0 \,..\, 0] + I = I & [1 \,..\, 1] \times I = I \times [1 \,..\, 1] = 1 & \text{(élément neutre)} \\ I \times (J \pm K) \subseteq I \times J \pm I \times K & & \text{(sous-distributivité)} \end{array}$$

On verra par la suite que la sous-distributivité est souvent la source de la sur-estimation des domaines de variations des expressions.

En résumé, deux règles générales importantes sont à noter [175] : soient e_1 et e_2 deux expressions arithmétiques réelles, et soient E_1 et E_2 leurs extensions naturelles aux intervalles respectives. On a :

$$(\mu(e_1) = 1 \wedge e_1 = e_2) \Rightarrow E_1 \subseteq E_2 \tag{2.12}$$

$$(\mu(e_1) = 1 \wedge \mu(e_2) = 1 \wedge e_1 = e_2) \Rightarrow E_1 = E_2 \tag{2.13}$$

On notera au passage la relation étroite entre la règle (2.13) et le théorème des occurrences simples de MOORE.

2.2 Les intervalles à bornes flottantes

Dès lors que l'on souhaite manipuler des intervalles en machine, il est impossible d'utiliser des intervalles réels. Comme dans le cas des réels (*cf.* chap. 1), on a recours à une approximation en remplaçant les intervalles réels par des intervalles flottants (*i.e.* des ensembles de *réels* représentés par des intervalles dont les bornes sont des nombres flottants). Il y a cependant une différence importante avec le cas de l'approximation des nombres réels avec des flottants : un intervalle réel est représenté par un intervalle flottant *le contenant*, ce qui garantit la correction des calculs.

Dans cette section, nous allons introduire les notations dont nous aurons besoin par la suite, puis nous décrirons les propriétés vérifiées par l'arithmétique des intervalles flottants. Enfin, nous verrons un aperçu des problèmes liés à l'implantation effective de l'arithmétique des intervalles en machine.

La définition d'un intervalle flottant est pratiquement identique à celle d'un intervalle réel (*cf.* section précédente). Seul change l'ensemble support pour les bornes : on utilise désormais l'ensemble des nombres flottants $\mathbb{F}$ et non plus $\mathbb{R}$. Le format importe peu ; par contre, on suppose que l'ensemble possède les propriétés requises par la norme IEEE 754 [116]).

Définition 2.14 (Borne d'un intervalle flottant). L'ensemble $\mathbb{F}^\diamond$ des bornes flottantes est défini par :

$$\mathbb{F}^\diamond = \mathbb{F}^\triangleleft \cup \mathbb{F}^\triangleright \quad \text{avec} \quad \left\{ \begin{array}{ll} \mathbb{F}^\triangleleft & = (\mathbb{F} \times \mathcal{L} \cup \{\langle -\infty, (\,\rangle, \langle +\infty, (\,\rangle\}) \\ \mathbb{F}^\triangleright & = (\mathbb{F} \times \mathcal{U} \cup \{\langle -\infty,)\,\rangle, \langle +\infty,)\,\rangle\}) \end{array} \right.$$

Insistons encore une fois sur le fait que, bien que défini à partir de $\mathbb{F}$, un intervalle flottant est un ensemble de nombres réels :

Définition 2.15 (Intervalle flottant). On appelle *intervalle flottant* I tout connexe de $\mathbb{R}$ défini par la donnée d'un couple constitué d'une borne flottante gauche et d'une borne flottante droite. On notera :

$$\begin{array}{lll} a. & \big(\langle \mathsf{g}, [\,\rangle, \langle \mathsf{h},]\,\rangle\big) & \equiv [\mathsf{g} \,..\, \mathsf{h}] \quad \equiv \{\mathsf{t} \in \mathbb{R} \mid \mathsf{g} \leqslant \mathsf{t} \leqslant \mathsf{h}\} \\ b. & \big(\langle \mathsf{g}, [\,\rangle, \langle \mathsf{h},)\,\rangle\big) & \equiv [\mathsf{g} \,..\, \mathsf{h}) \quad \equiv \{\mathsf{t} \in \mathbb{R} \mid \mathsf{g} \leqslant \mathsf{t} < \mathsf{h}\} \\ c. & \big(\langle \mathsf{g}, (\,\rangle, \langle \mathsf{h},]\,\rangle\big) & \equiv (\mathsf{g} \,..\, \mathsf{h}] \quad \equiv \{\mathsf{t} \in \mathbb{R} \mid \mathsf{g} < \mathsf{t} \leqslant \mathsf{h}\} \\ d. & \big(\langle \mathsf{g}, (\,\rangle, \langle \mathsf{h},)\,\rangle\big) & \equiv (\mathsf{g} \,..\, \mathsf{h}) \quad \equiv \{\mathsf{t} \in \mathbb{R} \mid \mathsf{g} < \mathsf{t} < \mathsf{h}\} \end{array}$$

On notera $\mathbb{I}_\circ^{\mathbb{F}}$ l'ensemble des intervalles flottants. Comme dans le cas des intervalles réels, soit $\mathbb{U}_\circ^{\mathbb{F}}$ l'ensemble des unions d'intervalles flottants. Dans la suite de ce document, on se référera presque exclusivement aux intervalles et unions d'intervalles flottants. Aussi, à partir de ce point, on utilisera les notations suivantes : $\mathbb{I}_\circ$ (resp. $\mathbb{U}_\circ$) en lieu et place de $\mathbb{I}_\circ^{\mathbb{F}}$ (resp. $\mathbb{U}_\circ^{\mathbb{F}}$) et $\mathbb{I}_\square$ (resp. $\mathbb{U}_\square$) pour désigner la restriction de $\mathbb{I}_\circ^{\mathbb{F}}$ aux intervalles flottants fermés (resp. $\mathbb{U}_\circ^{\mathbb{F}}$ aux unions d'intervalles flottants fermés). De plus, on écrira $\mathbb{I}$ (resp. $\mathbb{U}$) pour désigner indifféremment un intervalle réel ou flottant (resp. une union d'intervalles réels ou flottants).

Le passage des intervalles réels aux intervalles flottants se fait par une opération d'arrondi des bornes. On distingue l'arrondi des bornes gauche et droite :

$$\begin{array}{rll} \text{Arrondi de borne gauche :} & \lfloor\!\downarrow\!\rfloor : & \mathbb{R}^{\triangleleft} \longrightarrow \mathbb{F}^{\triangleleft} \\ & & \beta \longmapsto \max\{\gamma \in \mathbb{F}^{\triangleleft} \mid \gamma \trianglelefteq \beta\} \\ \text{Arrondi de borne droite :} & \lceil\!\uparrow\!\rceil : & \mathbb{R}^{\triangleright} \longrightarrow \mathbb{F}^{\triangleright} \\ & & \beta \longmapsto \min\{\gamma \in \mathbb{F}^{\triangleright} \mid \gamma \trianglerighteq \beta\} \end{array}$$

Ainsi, tout intervalle réel $I_{\mathbb{R}} = (\beta_1, \beta_2)$ est représenté en machine par l'intervalle flottant $I_{\mathbb{F}} = (\lfloor\!\downarrow\!\beta_1\!\rfloor, \lceil\!\uparrow\!\beta_2\!\rceil)$ (avec $I_{\mathbb{F}} \supseteq I_{\mathbb{R}}$). On notera que, dans le cas où $I_{\mathbb{R}}$ est un intervalle réduit à un point réel r, on a :

$$I_{\mathbb{F}} = \begin{cases} (\lfloor \mathsf{r} \rfloor \,..\, \lceil \mathsf{r} \rceil) & \text{si } \mathsf{r} \in \mathbb{R} \setminus \mathbb{F} \\ [\mathsf{r} \,..\, \mathsf{r}] \equiv \mathsf{r} & \text{si } \mathsf{r} \in \mathbb{F} \end{cases}$$

Notations : Étant donnée une relation réelle ρ, il est possible de l'approcher par des approximations dont le domaine est l'ensemble des intervalles (ou unions d'intervalles) flottants. Dans la suite, on utilisera les raccourcis d'écriture suivants : $\mathsf{Union}_\circ(\rho) \equiv \mathrm{apx}_{\mathbb{U}_\circ}(\rho)$, $\mathsf{Union}_\square(\rho) \equiv \mathrm{apx}_{\mathbb{U}_\square}(\rho)$, $\mathsf{Hull}_\circ(\rho) \equiv \mathrm{apx}_{\mathbb{I}_\circ}(\rho)$ et $\mathsf{Hull}_\square(\rho) \equiv \mathrm{apx}_{\mathbb{I}_\square}(\rho)$. Un intervalle flottant non vide $I = (\beta_1, \beta_2)$ est dit *canonique* si $\mathsf{Hull}_\square(I)$ contient au plus deux flottants (*i.e.* $\beta_2|_v \leqslant (\beta_1|_v)^+$). Un pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n$ est dit canonique lorsque tous les intervalles $I_1, \ldots, I_n$ sont canoniques. Soit $\mathring{I}$ l'intérieur de I (*i.e.* $\mathring{I} = \big(\langle \beta_1|_v, (\rangle, \langle \beta_2|_v,)\rangle\big)$).

Nous allons maintenant présenter l'arithmétique des intervalles flottants ainsi que ses propriétés. Nous distinguerons dans les sections 2.2.1 et 2.2.3 une arithmétique *fonctionnelle* et une arithmétique *relationnelle*, ce qui nous permettra en particulier de lever la restriction vue plus haut concernant la définition de la division.

2.2.1 Arithmétique (fonctionnelle) des intervalles flottants

Contrairement à l'approche choisie dans la section 2.1.1 où nous nous étions restreint aux intervalles fermés, nous allons présenter dans cette section les opérateurs sur des intervalles flottants quelconques.

Considérons l'addition de deux intervalles réels telle que définie par l'équation (2.4) transposée telle quelle pour des intervalles flottants :

$$[\mathsf{a} \,..\, \mathsf{b}] + [\mathsf{c} \,..\, \mathsf{d}] = [\mathsf{a} + \mathsf{c} \,..\, \mathsf{b} + \mathsf{d}], \qquad \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\} \subset \mathbb{F}$$

L'ensemble $\mathbb{F}$ n'étant pas fermé pour l'addition, nous n'avons pas la garantie que $\mathsf{a} + \mathsf{c}$ et $\mathsf{b} + \mathsf{d}$ soient des flottants. Il nous faut donc arrondir le résultat de ces additions si nous souhaitons que la somme de deux intervalles flottants soit un intervalle flottant. Pour préserver la correction des calculs, cet arrondi doit être fait de telle façon que l'intervalle flottant résultant contienne l'intervalle réel calculé (*arrondi extérieur*). De plus, si $\mathsf{a}+\mathsf{c}$ n'est pas un flottant, la borne gauche de l'intervalle résultat devra être ouverte car on aura $\mathsf{a} + \mathsf{c} > \lfloor \mathsf{a} + \mathsf{c} \rfloor$ (et symétriquement pour la borne droite). Pour tenir compte de tout cela, on définit la notion d'extension aux bornes d'une fonction réelle :

Définition 2.16 (Extension aux bornes [252]). Étant donnée une fonction réelle n-aire $f : \mathbb{R}^n \to \mathbb{R}$ et les bornes $\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle \in \mathbb{R}^\diamond$, on appelle *extension aux bornes associée à* f les fonctions $\overline{f}$ et $\underline{f}$ définies sur $\mathbb{R}^\diamond$ par :

$$\begin{cases} \overline{f}(\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle) = \begin{cases} \langle f(a_1, \ldots, a_n), \text{`})'\rangle & \text{si } \exists i\colon b_i = \text{`}(' \vee b_i = \text{`})'; \\ \langle f(a_1, \ldots, a_n), \text{`}]'\rangle & \text{sinon.} \end{cases} \\ \\ \underline{f}(\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle) = \begin{cases} \langle f(a_1, \ldots, a_n), \text{`}('\rangle & \text{si } \exists i\colon b_i = \text{`}(' \vee b_i = \text{`})'; \\ \langle f(a_1, \ldots, a_n), \text{`}['\rangle & \text{sinon.} \end{cases} \end{cases}$$

Il vient alors pour l'addition d'intervalles flottants quelconques (en utilisant la notation infixe usuelle pour l'addition) :

$$(\beta_1, \beta_2) + (\gamma_1, \gamma_2) = (\lfloor\!\downarrow \beta_1 \underline{+} \gamma_1 \downarrow\!\rfloor, \lceil\!\uparrow \beta_2 \overline{+} \gamma_2 \uparrow\!\rceil)$$

On souhaiterait pouvoir définir la soustraction, la multiplication et la division de la même manière. Malheureusement, il est impossible de transposer naïvement les formules de la page 17 dès lors que l'on utilise des nombres flottants pour représenter les bornes des intervalles. En effet, considérons par exemple la formule pour la multiplication :

$$[\mathsf{a} \,..\, \mathsf{b}] \times [\mathsf{c} \,..\, \mathsf{d}] = [\min(\mathsf{ac}, \mathsf{ad}, \mathsf{bc}, \mathsf{bd}) \,..\, \max(\mathsf{ac}, \mathsf{ad}, \mathsf{bc}, \mathsf{bd})]$$

Pour simplifier, restreignons-nous au cas d'intervalles fermés et multiplions, par exemple, les intervalles $I_1 = [-\infty \,..\, +\infty]$ et $I_2 = [-1 \,..\, 0]$. Il vient :

$$I_1 \times I_2 = [\min(-\infty \times -1, -\infty \times 0, +\infty \times -1, +\infty \times 0) \,..\, \max(-\infty \times -1, -\infty \times 0, +\infty \times -1, +\infty \times 0)]$$

Or, si l'on regarde le tableau 1.1, p. 7, on constate que les opérations $-\infty \times 0$ et $+\infty \times 0$ entrainent l'apparition de NaNs. Ainsi, suivant la façon dont sont implémentées les fonctions $\min$ et $\max$ on obtiendra au final, soit l'intervalle invalide (ou vide, suivant la convention adoptée) $[NaN \,..\, NaN]$, soit l'intervalle $[-\infty \,..\, +\infty]$; les NaNs étant non ordonnés, les opérations $\min$ et $\max$ ne sont plus nécessairement commutatives.

Afin de garantir la correction des calculs, il est donc impossible d'utiliser les formules standards de l'arithmétique d'intervalles réels. Pour se prémunir contre l'apparition de NaNs, on devra obligatoirement faire des études de cas a priori sur les valeurs des bornes pour ne jamais effectuer d'opérations invalides.

2.2.2 Propriétés

L'arithmétique des intervalles flottants hérite à la fois des limitations de l'arithmétique des nombres flottants et de celles de l'arithmétique des intervalles réels. Ainsi, on peut montrer que l'arithmétique des intervalles flottants a les propriétés de commutativité, sous-distributivité et possède un élément neutre. Elle perd cependant l'associativité, les nombres flottants ne possédant pas eux-mêmes cette propriété (*cf.* chap. 1).

L'implémentation d'une arithmétique d'intervalles gérant tant les intervalles ouverts que les intervalles fermés est facilitée par certaines dispositions de la norme IEEE 754 [116] qui impose en particulier la présence d'un indicateur modifié après chaque résultat[‡] et mis à 1 dès lors que la dernière opération a délivré un résultat ayant été arrondi (*inexact flag*). En testant cet indicateur, il devient aisé de déterminer la forme de la *bracket* correspondante à partir des *brackets* des bornes utilisées pour le calcul. Néanmoins, cela est très coûteux car cela suppose d'accéder au registre de l'unité arithmétique après chaque opération et de remettre l'indicateur à 0, ce qui entraîne sur les architectures modernes une rupture du pipe-line, dégradant les performances. On peut alors se demander s'il y a un intérêt quelconque à gérer les ouverts. On verra au chapitre 3 qu'une opération couramment effectuée lors de la résolution de systèmes de contraintes d'intervalles est le calcul de l'enveloppe de l'intersection d'une relation réelle n-aire ρ et d'un pavé $\boldsymbol{B}$: $\mathsf{Hull}(\rho \cap \boldsymbol{B})$. Or, l'intersection $\rho \cap \boldsymbol{B}$ ne peut être effectuée en machine car le résultat est, en toutes généralités, un sous-ensemble de $\mathbb{R}^n$. Ce que l'on peut calculer est, au mieux, $\mathsf{Hull}(\mathsf{Union}(\rho) \cap \boldsymbol{B})$.

[‡] Plus précisément, l'indicateur est un *sticky bit, i.e.* la modification se fait par un *ou* logique avec sa valeur précédente.

Une question importante est alors de savoir si les deux formes sont strictement équivalentes. Nous allons montrer (prop. 2.1) que ce n'est pas le cas si l'on conduit les calculs en utilisant exclusivement des intervalles fermés, mais que l'on a bien l'équivalence si l'on emploie des intervalles ouverts/fermés :

Lemme 2.1. *Étant donnés une relation n-aire $\rho \subseteq \mathbb{R}$ et un pavé $\boldsymbol{B}$, on a l'équivalence :*

$$\rho \cap \boldsymbol{B} = \varnothing \iff \mathsf{Union}_\circ(\rho) \cap \boldsymbol{B} = \varnothing \tag{2.14}$$

Démonstration. Commençons par remarquer que l'équivalence (2.14) est fausse si l'on considère uniquement des intervalles fermés. Par exemple, étant donnée la relation $\rho = \{x \in \mathbb{R} \mid 1/10 \leqslant x \leqslant 1\}$ et le pavé $\boldsymbol{B}_\square = [0..\lfloor 1/10 \rfloor]$, nous avons $\rho \cap \boldsymbol{B}_\square = \varnothing$ mais $\mathsf{Union}_\square(\rho) = [\lfloor 1/10 \rfloor..1]$. D'où $\mathsf{Union}_\square(\rho) \cap \boldsymbol{B}_\square = [\lfloor 1/10 \rfloor..\lfloor 1/10 \rfloor]$.

L'équation (2.14) est prouvée en raisonnant sur une seule dimension. Le résultat peut alors être généralisé en n dimensions.

L'implication $\Leftarrow$ est triviale car $\mathsf{Union}_\circ(\rho) \supseteq \rho$, par définition de l'opérateur Union.

Prouvons maintenant :

$$\rho \cap \boldsymbol{B} = \varnothing \Rightarrow \mathsf{Union}_\circ(\rho) \cap \boldsymbol{B} = \varnothing \tag{2.15}$$

Les positions relatives possibles pour $\boldsymbol{B}$ et ρ telles que $\rho \cap \boldsymbol{B} = \varnothing$ sont dessinées dans la figure 2.6. Remarquons que si ρ n'a pas de borne inférieure (resp. supérieure), les seules positions possibles sont *Pos. 1b* et *Pos. 2* (resp. *Pos. 1a* et *Pos. 2*).

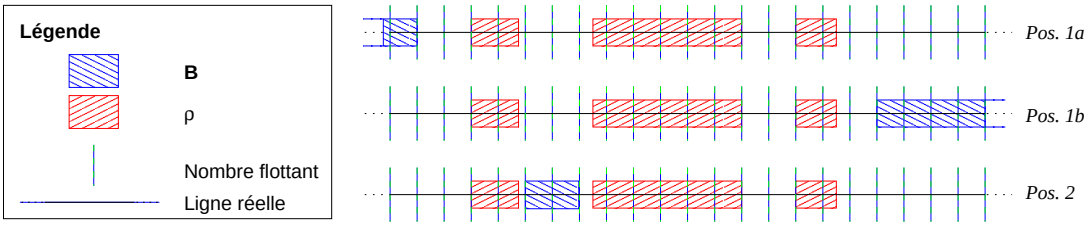


FIG. 2.6: Positions relatives de ρ et $\boldsymbol{B}$ telles que $\rho \cap \boldsymbol{B} = \varnothing$

Cas *Pos. 1a*. Comme $\rho \cap \boldsymbol{B} = \varnothing$, on a $\sup_\circ(\boldsymbol{B}) \lhd \inf_\circ(\rho)$. Montrons alors :

$$\sup_\circ(\boldsymbol{B}) \lhd \inf_\circ(\rho) \Rightarrow \sup_\circ(\boldsymbol{B}) \lhd \lfloor\!\lfloor \inf_\circ(\rho) \rfloor\!\rfloor \tag{2.16}$$

Cette relation sera suffisante pour prouver l'équation (2.15) car $\inf_\circ(\mathsf{Union}_\circ(\rho)) = \lfloor\!\lfloor \inf_\circ(\rho) \rfloor\!\rfloor$. Posons $\sup_\circ(\boldsymbol{B}) = \langle\, \mathsf{g}, \alpha_{1\square} \,\rangle$, $\inf_\circ(\rho) = \langle\, \mathsf{r}, \alpha_2 \,\rangle$ et $\lfloor\!\lfloor \inf_\circ(\rho) \rfloor\!\rfloor = \langle\, \lfloor \mathsf{r} \rfloor, \alpha_3 \,\rangle$. Par définition de l'ordre $\lhd$ (éq. (2.1), p. 16), il vient $\mathsf{g} < \mathsf{r} \vee (\mathsf{g} = \mathsf{r} \wedge \alpha_{1\square} \prec \alpha_2)$. Considérons les deux cas :

1. $\mathsf{g} < \mathsf{r}$: dans ce cas, soit $\mathsf{g} < \lfloor \mathsf{r} \rfloor$ et l'on a directement $\sup_\circ(\boldsymbol{B}) \lhd \lfloor\!\lfloor \inf_\circ(\rho) \rfloor\!\rfloor$, ce qui termine la preuve de l'équation (2.16) ; soit $\mathsf{g} = \lfloor \mathsf{r} \rfloor$, ce qui implique $\alpha_3 = '$ (' puisque nous avons $\lfloor \mathsf{r} \rfloor < \mathsf{r}$ et $\langle\, \lfloor \mathsf{r} \rfloor, (\,\rangle \rhd \langle\, \lfloor \mathsf{r} \rfloor, [\,\rangle$. Une fois de plus, il vient $\sup_\circ(\boldsymbol{B}) \lhd \lfloor\!\lfloor \inf_\circ(\rho) \rfloor\!\rfloor$,
2. $\mathsf{g} = \mathsf{r} \wedge \alpha_{1\square} \prec \alpha_2$: comme nous avons $\mathsf{g} \in \mathbb{F}$, il vient $\mathsf{r} \in \mathbb{F}$. D'où $\lfloor \mathsf{r} \rfloor = \mathsf{r}$. Par conséquent, $\lfloor\!\lfloor \inf_\circ(\rho) \rfloor\!\rfloor = \inf_\circ(\rho)$, ce qui nous donne immédiatement $\sup_\circ(\boldsymbol{B}) \lhd \lfloor\!\lfloor \inf_\circ(\rho) \rfloor\!\rfloor$ et termine la preuve de l'équation (2.15) ;

Cas *Pos. 1b*. La preuve se fait de la même façon que pour le cas *Pos. 1a* ;

Cas *Pos. 2*. La preuve se fait en appliquant deux fois (une fois pour chaque côté) le même raisonnement que celui tenu pour prouver les cas *Pos. 1a* et *Pos. 1b*.

■

Proposition 2.1. *Étant donnés une relation n-aire $\rho \subseteq \mathbb{R}$ et un pavé $\boldsymbol{B}$, on a l'égalité :*

$$\mathsf{Hull}_\circ(\rho \cap \boldsymbol{B}) = \mathsf{Hull}_\circ(\mathsf{Union}_\circ(\rho) \cap \boldsymbol{B}) \tag{2.17}$$

Démonstration. Encore une fois, remarquons que l'équation (2.17) est fausse si l'on utilise uniquement des intervalles fermés puisque $\rho \cap \boldsymbol{B}_\square = \varnothing$ n'implique pas $\mathsf{Union}_\square(\rho) \cap \boldsymbol{B}_\square = \varnothing$ (voir le lemme 2.1).

On peut se contenter de considérer le cas $\rho \cap \boldsymbol{B} \neq \varnothing$ puisque le lemme 2.1 nous assure que l'équation (2.17) est vérifiée quand $\rho \cap \boldsymbol{B} = \varnothing$.

Nous allons faire la preuve en raisonnant une fois de plus sur une seule dimension, le résultat se généralisant au cas de n dimensions. L'équation (2.17) se prouve en raisonnant sur les bornes inférieure et supérieure dans $\mathbb{R}^\diamond$ des ensembles $\rho \cap \boldsymbol{B}$ et $\mathsf{Union}_\circ(\rho) \cap \boldsymbol{B}$.

Nous avons $\inf_\circ(\rho \cap \boldsymbol{B}) = \max(\inf_\circ(\rho), \inf_\circ(\boldsymbol{B}))$, ce qui nous conduit à l'égalité $\inf_\circ(\mathsf{Hull}_\circ(\rho \cap \boldsymbol{B})) = \max(\lfloor\inf_\circ(\rho)\rfloor, \inf_\circ(\boldsymbol{B}))$. De la même manière, $\sup_\circ(\mathsf{Hull}_\circ(\rho \cap \boldsymbol{B})) = \min(\lceil\sup_\circ(\rho)\rceil, \sup_\circ(\boldsymbol{B}))$. De plus, $\inf_\circ(\mathsf{Union}_\circ(\rho)) = \lfloor\inf_\circ(\rho)\rfloor$ et $\sup_\circ(\mathsf{Union}_\circ(\rho)) = \lceil\sup_\circ(\rho)\rceil$. Par conséquent, $\inf_\circ(\mathsf{Union}_\circ(\rho) \cap \boldsymbol{B}) = \max(\lfloor\inf_\circ(\rho)\rfloor, \inf_\circ(\boldsymbol{B}))$ et $\sup_\circ(\mathsf{Union}_\circ(\rho) \cap \boldsymbol{B}) = \min(\lceil\sup_\circ(\rho)\rceil, \sup_\circ(\boldsymbol{B}))$.

Finalement, $\inf_\circ(\mathsf{Hull}_\circ(\mathsf{Union}_\circ(\rho) \cap \boldsymbol{B})) = \max(\lfloor\inf_\circ(\rho)\rfloor, \inf_\circ(\boldsymbol{B}))$ et $\sup_\circ(\mathsf{Hull}_\circ(\mathsf{Union}_\circ(\rho) \cap \boldsymbol{B})) = \min(\lceil\sup_\circ(\rho)\rceil, \sup_\circ(\boldsymbol{B}))$. Les bornes des ensembles $\mathsf{Hull}_\circ(\rho \cap \boldsymbol{B})$ et $\mathsf{Hull}_\circ(\mathsf{Union}_\circ(\rho) \cap \boldsymbol{B})$ sont donc les mêmes. Comme ils sont tous les deux connexes (du fait de l'utilisation de Hull), on en déduit qu'ils sont égaux. ■

2.2.3 L'arithmétique relationnelle des intervalles flottants

La fonction racine carrée étant monotone croissante sur son domaine de définition, il est très facile de déterminer son extension aux intervalles pour le cas d'intervalles fermés comme suit :

```
Sqrt(entrée I1; sortie I2 = √I1)          % Précondition : I1 ⩾ 0
début
    si (I1 = ∅) alors
        retourner (∅)
    sinon
        RoundDn()
        I2 ← √I1
        RoundUp()
        I2 ← √I1
        retourner (I2)
    finsi
fin
```

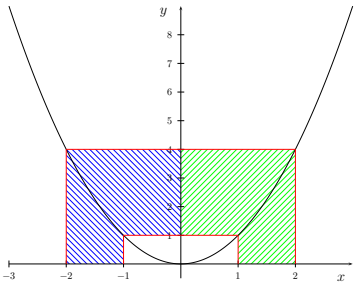


FIG. 2.7: Valeurs possibles pour x

La norme IEEE 754 spécifie que la fonction racine carrée doit être correctement arrondie. L'algorithme ci-dessus calcule donc bien l'extension naturelle aux intervalles de l'opérateur $\sqrt{\ }$, telle qu'elle est définie au début de la section 2.1.2.1.

Étant données deux variables x et y de domaines respectifs I_x et $I_y = [1\,..\,4]$, considérons la relation :

$$y = x^2 \tag{2.18}$$

et cherchons quelles valeurs peut prendre la variable x. Si l'on considère le dessin 2.7, on voit immédiatement que x peut prendre toutes les valeurs du domaine $D_x = [-2..-1] \cup [1..2]$. Cependant, on ne peut pas se contenter d'inverser la relation 2.18 en la relation $x = \sqrt{y}$ pour calculer le domaine de x à partir de celui de y, car la définition de l'extension aux intervalles de la fonction racine carrée nous ferait perdre l'intervalle $[-2\,..\,-1]$. On a donc besoin d'une fonction ${}^\pi\sqrt{\ }$ dont la définition serait :

$$\forall I \in \mathbb{I}\colon {}^\pi\sqrt{I} = \mathsf{Hull}(\{\mathsf{r} \in \mathbb{R} \mid \exists s \in I\colon \mathsf{r}^2 = s\})$$

On notera l'utilisation de l'opérateur Hull pour obtenir au final un seul intervalle. Son utilisation ne se justifie bien sûr que si l'on ne s'autorise pas à manipuler des unions d'intervalles. On peut définir des versions relationnelles des autres opérateurs, tels que la puissance ou la multiplication.

La perte d'information (codage du « trou » induit par la non intervalle-connexité de certains opérateurs) due à l'usage de Hull s'avère en pratique un facteur important de diminution des performances. C'est pourquoi on utilise souvent localement des versions modifiées conservant les trous dans les domaines, en espérant récupérer des intervalles à l'intersection de domaines suivante.

DEUXIÈME PARTIE

Consistances locales

1. Introduction
2. Consistances locales : notions de base
3. Extension de la définition de box-consistance

Introduction

NOMBRE DE PROBLÈMES peuvent se formuler en terme de *problème de satisfaction de contraintes* — nous utiliserons dans la suite le terme *CSP*, acronyme de la traduction anglaise (*Constraint Satisfaction Problem*) — de la façon suivante : étant donnés un ensemble de variables $\mathcal{V} = \{v_1, \ldots, v_n\}$, un produit cartésien de domaines $\boldsymbol{D} = D_1 \times \cdots \times D_n$ (avec $v_i \in D_i, \quad \forall i \in \{1, \ldots, n\}$) et un ensemble de relations $\mathcal{C} = \{c_1, \ldots, c_m\}$ entre les variables (*contraintes*), on cherche toutes les affectations des variables satisfaisant la conjonction $c_1 \wedge \cdots \wedge c_m$.

Suivant le type des domaines des variables, on obtient des CSPs discrets (DCSP) ou continus (CCSP). Les problèmes considérés originellement mettent en jeu des contraintes unaires ou binaires sur des domaines discrets. Dans ce formalisme, on distingue trois types de tests de satisfaction [152, 159] dont les noms font référence à la représentation en graphe (où les nœuds sont étiquetés par les variables et les arcs par les contraintes — *cf.* figure 2.8) communément utilisée pour représenter ces CSPs :

- la *consistance de nœud*, où l'on vérifie que chaque contrainte unaire du système, de la forme $c(v_i)$, est vérifiée pour toutes les valeurs du domaine D_i de v_i ;
- la *consistance d'arc* où, pour chaque contrainte binaire $c(v_i, v_j)$, on vérifie que pour chaque valeur du domaine de v_i (resp. v_j), il existe au moins une valeur dans le domaine de v_j (resp. v_i) telle que la contrainte c est vérifiée. Nous verrons au chapitre 3 que cette notion de consistance peut s'étendre à des contraintes d'arité supérieure à deux ;
- la *consistance de chemin*, généralisation de la consistance d'arc, où, étant donnée une contrainte $c(v_i, v_j)$ vérifiée pour les valeurs a de v_i et b de v_j, on s'assure que pour tout « chemin » de la forme $c_0(a, v_{k_1})$, $c_1(v_{k_1}, v_{k_2}) \ldots$, $c_{u-1}(v_{k_{u-1}}, v_{k_u})$, $c_u(v_{k_u}, b)$, il existe une affectation consistante pour les variables $v_{k_1}, \ldots, v_{k_u}$.

Note : Le bon usage devrait nous faire traduire le terme original « *consistency* » par « cohérence ». L'emploi du terme « consistance » étant cependant avéré dans la littérature française sur le sujet, nous nous permettrons d'utiliser dans la suite cet anglicisme.

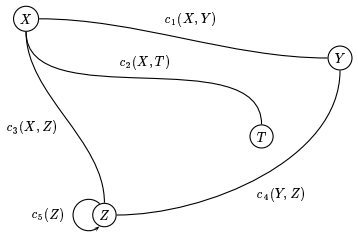


FIG. 2.8: Un réseau de contraintes discrètes

Dans le cas des problèmes faisant intervenir des variables à domaines discrets, l'une des méthodes les plus simples pour trouver les solutions consiste à affecter incrémentalement des valeurs aux différentes variables et à tester la satisfaction des contraintes dont toutes les variables ont été instanciées. Les performances d'un tel algorithme sont en général fort mauvaises car l'affectation d'une valeur invalide à une variable v_j peut n'être remise en cause que très tard, entraînant un phénomène de « *thrashing* » [39] consistant en une exploration de tout l'espace de recherche correspondant au produit cartésien des domaines des variables v_k pour tout $k > j$ avant de reconsidérer le choix fait pour la valeur de v_j.

Comme le souligne MACKWORTH [152], la technique ci-dessus voit son temps de calcul augmenter de façon exponentielle en le nombre de variables, à la fois dans le pire des cas et en moyenne. FIKES [76] fut le premier à faire l'observation suivante : étant données deux variables à domaines discrets $v_1 \in D_1$ et $v_2 \in D_2$ et une contrainte $c(v_1, v_2)$, si une valeur $a \in D_1$ est telle qu'il n'existe pas de valeur $b \in D_2$ telle que $c(a, b)$, on peut retirer a du domaine de v_1. Cette observation est à la base de tous les algorithmes de calcul de la consistance d'arc et en particulier de l'*algorithme de* WALTZ *[245]*, — correspondant à l'une des premières utilisation de la propagation de contraintes —, utilisé par WALTZ pour faire de l'étiquetage d'arêtes de polyhèdres (on trouvera chez MCALLESTER [159] par exemple, une description détaillée du problème de WALTZ). Les nombreux algorithmes de calcul de la consistance d'arc (AC-1, AC-2 [245], AC-3 [152], AC-4 [166], AC-5 [232], AC-6 [35]) servent à

faire un filtrage précoce en éliminant le plus de valeurs possibles des domaines des variables avant d'effectuer une recherche de satisfaction par retour-arrière. On trouvera dans les articles de MACKWORTH et FREUDER [154, 153] une étude de la complexité de ces algorithmes.

Afin de donner un aperçu des idées sous-tendant cette partie, nous allons présenter ci-dessous un exemple de résolution d'un système de contraintes entre des variables entières x et y :

Exemple 2.8 (Résolution par propagation de domaines). *Soient les variables entières x et y et les domaines $D_x = [0 .. 7]$ et $D_y = [0 .. 7]$. On considère les trois contraintes entre x et y du système* (2.19) *:*

$$\begin{cases} xy & = 6 \\ x + y & = 5 \\ x & < y \end{cases} \tag{2.19}$$

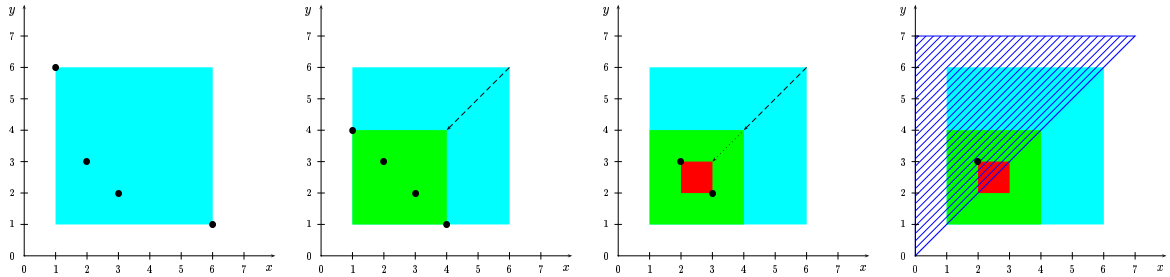
Dans la suite, on ne représente les domaines des variables que par des intervalles. La figure 2.9 décrit pas-à-pas le processus de résolution :

(a). *Considération de la contrainte $x*y = 6$. Les points de la figure (a) correspondent aux seuls couples possibles pour les variables x et y. Afin de préserver la connexité des domaines, on réduit D_x et D_y en $D_x^1 = [1 .. 6]$ et $D_y^1 = [1 .. 6]$;*

(b). *Considération de la contrainte $x + y = 5$. Réduction des domaines de x et y à $D_x^2 = [1 .. 4]$ et $D_y^2 = [1 .. 4]$;*

(c). *Reconsidération de la contrainte $x * y = 6$ (car les domaines de x et y ont été modifiés depuis la dernière invocation de cette contrainte). On obtient alors les nouveaux domaines $D_x^3 = [2 .. 3]$ et $D_y^3 = [2 .. 3]$. Il faut alors réinvoquer la contrainte $x + y = 5$ pour la même raison, ce qui n'induit aucun changement dans les domaines ;*

(d). *Considération de la contrainte $x < y$ qui conduit immédiatement aux domaines finaux $D_x^4 = \{2\}$ et $D_y^4 = \{3\}$.*



(a) (b) (c) (d)

FIG. 2.9: Résolution pas-à-pas du système (2.19)

La transposition de la consistance d'arc pour des variables à domaines continus a été faite en se basant sur la notion d'intervalles (*cf.* partie I) par DAVIS [61], puis par LHOMME [151] (*B-consistance*) et BENHAMOU *et al.* (*hull-consistance* [21], *box-consistance* [29]). Nous donnerons au chapitre suivant les définitions précises de ces différentes consistances, en nous intéressant plus particulièrement à la hull-consistance et à la box-consistance. Durant notre thèse, nous avons étudié le comportement des algorithmes de calcul de ces deux consistances. Pour cela, nous avons défini et implémenté au-dessus de clp(fd) [50] un langage de programmation logique avec contraintes baptisé DecLIC [94, 92] (*Declarative Language with Interval Constraints*) permettant de résoudre des systèmes de contraintes réelles (non-)linéraires. DecLIC autorise le programmeur à choisir pour chaque contrainte l'algorithme de calcul de consistance à utiliser : HC3 pour obtenir la hull-consistance ou BC3 pour la box-consistance. Nous avons ainsi pu noter qu'aucun de ces deux algorithmes n'est un meilleur choix pour toutes les contraintes. Nous avons alors défini une notion de consistance hybride [26] pour tirer partie des points forts des deux algorithmes ; une étude d'une telle consistance hybride apparaît dans la thèse de doctorat de Laurent GRANVILLIERS [95]. Enfin, nous avons modifié légèrement la définition de la box-consistance, ce qui nous a permis de définir un nouvel algorithme pour la calculer qui combine les points forts de HC3 et BC3. Ceci autorise donc la mise au point de

systèmes de résolution de contraintes plus rapides qui, contrairement à DecLIC, ne requièrent pas du programmeur une connaissance des capacités des algorithmes de résolution employés.

Cette partie s'organise comme suit : nous introduisons les notations et les notions de base dans le chapitre 3 ; nous donnons ensuite dans le chapitre 4 une nouvelle définition de la box-consistance étendant celle donnée dans le chapitre 3 qui nous permet de décrire BC4, le nouvel algorithme pour l'obtenir, plus efficace que la méthode décrite dans l'article original de BENHAMOU *et al.* [29].

Contributions

- Analyse de l'efficacité des algorithmes de calcul associés à la hull-consistance et à la box-consistance en fonction de la forme des contraintes ;
- extension de la définition de box-consistance subsumant les notions de box-consistance classique et de hull-consistance, et mise au point d'un algorithme hybrique de calcul de la box-consistance étendue accélérant la résolution de contraintes réelles.

CHAPITRE 3

Consistances locales

Contraintes réelles et contraintes d'intervalles — opérateurs de contraction — consistance d'arc — hull-consistance — box-consistance

DANS CE CHAPITRE, nous définissons formellement la notion de contrainte et introduisons les notations correspondantes. Nous donnons ensuite les définitions de l'arc-consistance ainsi que de ses « affaiblissements » (hull-consistance et box-consistance) pour la résolution des contraintes à variables réelles. Enfin, nous montrons que l'opérateur associé à la box$_\varphi$-consistance [95, 96] est bien un opérateur de contraction.

3.1 Contraintes réelles et contraintes d'intervalles

Soient $\Sigma_1 = (\mathcal{F}_1, \mathcal{R}_1)$ et $\Sigma_2 = (\mathcal{F}_2, \mathcal{R}_2)$ deux *signatures*, avec $\mathcal{F}_1$ et $\mathcal{F}_2$ (resp. $\mathcal{R}_1$ et $\mathcal{R}_2$) des ensembles de symboles de fonctions (resp. de relations). Donnons-nous aussi une Σ_1-structure $\mathcal{D}_1 = \langle\, \mathbb{R}, \Sigma_1 \,\rangle$ et une Σ_2-structure $\mathcal{D}_2 = \langle \mathbb{I}, \Sigma_2 \rangle$, ainsi que deux ensembles infinis dénombrables $V_\mathbb{R} = \{x_1, x_2, \dots\}$ et $V_\mathbb{I} = \{X_1, X_2, \dots\}$ de variables valuées respectivement sur $\mathbb{R}$ et $\mathbb{I}$.

On définit une *contrainte réelle* (resp. une *contrainte d'intervalles*) comme une formule atomique du premier ordre bâtie à partir de $\mathcal{D}_1$ et $V_\mathbb{R}$ (resp. à partir de $\mathcal{D}_2$ et $V_\mathbb{I}$). La définition — fort générale — d'une contrainte donnée ici correspond à celle apparaissant dans l'article de synthèse de JAFFAR et MAHER [118]. Le lecteur devra cependant garder à l'esprit que nous nous intéresserons dans la suite presqu'exclusivement à des contraintes numériques, ce qui réduit en pratique les ensembles $\mathcal{R}_1$ et $\mathcal{R}_2$ aux symboles susceptibles d'être interprétés comme des symboles de relations d'égalité et d'inégalité. Étant donnée une contrainte réelle n-aire c, on notera ρ_c la relation n-aire qui lui est associée.

Exemple 3.9 (Relation associée à une contrainte). *Considérons trois variables réelles $\{x_1, x_2, x_3\} \subseteq \mathcal{V}_\mathbb{R}$ ainsi que la contrainte c: $x_3 = x_1 + x_2$. On a alors :*

$$\rho_c = \{(\mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_3) \in \mathbb{R}^3 \mid \mathsf{r}_1 + \mathsf{r}_2 = \mathsf{r}_3\}$$

De la même façon, on notera ρ_C la relation associée à la contrainte d'intervalles C définie comme suit :

Définition 3.1 (Relation associée à une contrainte d'intervalles [29]). Soit $C(X_1, \dots, X_n)$ une contrainte d'intervalles et soit ρ_C le sous-ensemble de $\mathbb{I}^n$, appelé *relation associée à* C défini par :

$$\rho_C = \{(I_1, \dots, I_n) \in \mathbb{I}^n \mid C(I_1, \dots, I_n)\}$$

Le passage d'une contrainte réelle à une contrainte d'intervalles se fait en suivant la définition d'une extension d'une relation réelle (*cf.* page 24) :

Définition 3.2 (Extension d'une contrainte réelle). Étant donnée une contrainte réelle n-aire c, on appelle *extension aux intervalles* de c la contrainte d'intervalles C définie par :

$$\forall I_1, \dots, \forall I_n \in \mathbb{I}: \quad \exists a_1 \in I_1 \wedge \dots \wedge \exists a_n \in I_n \wedge c(a_1, \dots, a_n) \Rightarrow C(I_1, \dots, I_n)$$

Dans cette partie, nous considérerons souvent des formules sans quantificateur. Ce ne sera plus le cas dans la partie III, où l'on rencontrera des variables universellement quantifiées.

3.2 Contraintes et opérateurs de contraction

Nous présentons dans cette section la modélisation du processus de résolution d'un problème de satisfaction de contraintes (CSP) en nous basant sur le formalisme de BENHAMOU [21, 24].

Considérons un CSP constitué d'un ensemble de contraintes réelles $\mathcal{C} = \{c_1, \ldots, c_m\}$ et d'un produit cartésien $\boldsymbol{D} = D_1 \times \cdots \times D_n$ de domaines des variables $v_1, \ldots, v_n$ (avec $v_i \in D_i, \forall i \in \{1, \ldots, n\}$) ayant au moins une occurrence dans $\mathcal{C}$.

Note : Afin de simplifier la présentation, nous considérerons uniquement dans la suite des CSPs préalablement cylindrifiés [102] (i.e. des CSPs où toutes les contraintes c_i sont d'arité n).

Le but d'un résolveur de contraintes est d'isoler les n-uplets de $\boldsymbol{D}$ satisfaisant la conjonction $c_1 \wedge \cdots \wedge c_m$. Pour cela, on associe à chaque relation n-aire ρ_c d'une contrainte c un *opérateur de contraction* $N[c]$ (ou CNO dans la suite — *Constraint Narrowing Operator* [178, 21]) éliminant d'un pavé donné en entrée *le plus de n-uplets possible* n'appartenant pas à la relation ρ_c. La notion de « le plus de n-uplets possible » peut s'exprimer formellement en ayant recours à la notion de domaine d'approximation (*cf.* déf. 2.5, p. 18). On a :

Définition 3.3 (Opérateur de contraction [24]). Étant donnés un domaine d'approximation $\mathcal{A}$ de $\mathbb{R}$, une relation n-aire réelle ρ et $\boldsymbol{D_1}, \boldsymbol{D_2} \in \mathcal{A}^n$ deux pavés de domaines, on appelle *opérateur de contraction* pour la relation ρ toute fonction $N : \mathcal{A}^n \to \mathcal{A}^n$ ayant les trois propriétés suivantes :

$N(\boldsymbol{D_1})$	$\subseteq \boldsymbol{D_1}$	contractance
$\rho \cap \boldsymbol{D_1}$	$\subseteq N(\boldsymbol{D_1})$	complétude*
$\boldsymbol{D_1} \subseteq \boldsymbol{D_2}$	$\Rightarrow N(\boldsymbol{D_1}) \subseteq N(\boldsymbol{D_2})$	monotonie

BENHAMOU définit aussi une notion plus forte d'*opérateur de contraction optimal* ayant, en plus, la propriété d'*idempotence* ($N(N(\boldsymbol{D})) = N(\boldsymbol{D})$) :

Définition 3.4 (Opérateur de contraction optimal [24]). Étant donné un domaine d'approximation $\mathcal{A}$ de $\mathbb{R}$, un opérateur de contraction $N : \mathcal{A}^n \to \mathcal{A}^n$ pour une relation réelle n-aire ρ est dit *optimal* si et seulement si :

$$\forall \boldsymbol{D} \in \mathcal{A}^n : N(\boldsymbol{D}) = \text{apx}_{\mathcal{A}}(\rho \cap \boldsymbol{D})$$

On trouvera la preuve qu'un opérateur de contraction optimal est idempotent dans [30, 95].

À partir de la notion d'opérateur de contraction, BENHAMOU introduit celle de *CSP étendu* (ECSP) où l'on associe à chaque contrainte un CNO pour la relation sous-jacente. Par exemple, on peut définir pour le système donné au début de cette section le CSP étendu $\mathcal{S} = (\{(c_1, N[c_1]), \ldots, (c_m, N[c_m])\}, \boldsymbol{D})$.

La *sémantique déclarative* $\mathcal{S}^\star$ de $\mathcal{S}$ correspond à l'ensemble des n-uplets de $\boldsymbol{D}$ satisfaisant la conjonction $c_1 \wedge \cdots \wedge c_m$. Le calcul de $\mathcal{S}^\star$ est souvent impossible et l'on doit donc se contenter de calculer une *sémantique approchée* $\overline{\mathcal{S}}$ correspondant au plus grand point-fixe commun des opérateurs $N[c_i]$ contenu dans le pavé $\boldsymbol{D}$ de départ :

$$\overline{\mathcal{S}} = \max(\{u \in \bigcap_{i=1}^{m} \text{point-fixe}(N[c_i]) \mid u \subseteq \boldsymbol{D}\})$$

Ce plus grand point-fixe est obtenu par application de l'algorithme général de filtrage Nar (Alg. 3.1) qui gère une *liste de propagation*† contenant toutes les contraintes dont le CNO doit être réinvoqué : après l'application d'un opérateur de contraction sur un produit cartésien de domaines, on met dans la liste toutes les contraintes possédant une occurrence d'une des variables dont le domaine a été modifié.

L'algorithme Nar possède les propriétés suivantes (on se reportera à l'article d'OLDER et VELLINO [178] pour les preuves) :

*Ou « correction » pour certains auteurs [24].

†Cette liste est, en général, gérée en file (FIFO), mais on peut imaginer d'utiliser d'autres stratégies afin, par exemple, de privilégier les opérateurs semblant réduire le plus efficacement les domaines des variables [25].

ALG. 3.1: Algorithme de filtrage Nar

Nar(**entrée** $\{(c_1, N[c_1]), \ldots, (c_m, N[c_m])\}$; **entrée/sortie** $\boldsymbol{D} = D_1 \times \cdots \times D_n$)
début
  $\mathcal{Q} \leftarrow \{c_1, \ldots, c_m\}$ *% contraintes ajoutées à la liste de propagation*
  $\mathcal{ST} \leftarrow \mathcal{ST} \cup \{c_1 \ldots, c_m\}$ *% contraintes ajoutées au store*
  tant que ($\mathcal{Q} \neq \varnothing$ **et** $\boldsymbol{D} \neq \varnothing$) **faire**
    $c \leftarrow$ `choisir une contrainte` c_i `dans` $\mathcal{Q}$
    $\boldsymbol{D'} \leftarrow N[c](\boldsymbol{D})$
    si ($\boldsymbol{D'} \neq \boldsymbol{D}$) **alors**
      $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{c_j \in \mathcal{ST} \mid \exists x_k \in \mathsf{Var}(c_j) \wedge D'_k \neq D_k\}$
      $\boldsymbol{D} \leftarrow \boldsymbol{D'}$
      si (Idempotent($N[c]$)) **alors**
        $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{c\}$
      finsi
    sinon
      $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{c\}$
    finsi
  ftq
  retourner ($\boldsymbol{D}$)
fin.

Proposition 3.1 (Propriétés de Nar [178, 21]). *Soit* $\mathcal{S} = (\{(c_1, N[c_1]), \ldots, (c_m, N[c_m])\}, \boldsymbol{D})$ *un CSP étendu. On a :*
- *L'algorithme* Nar *est* confluent *(le résultat ne dépend pas de l'ordre d'application des opérateurs de contraction) ;*
- *Si les opérateurs* $N[c_1], \ldots, N[c_m]$ *terminent, l'algorithme* Nar *appliqué à* $\mathcal{S}$ termine *;*
- $\mathsf{Nar}(\mathcal{S})$ *calcule le plus grand point-fixe commun des opérateurs* $N[c_1], \ldots, N[c_m]$ *inclus dans* $\boldsymbol{D}$.

3.3 Consistance d'arc et consistance d'arc faible

Les opérateurs de contraction modélisent l'application d'algorithmes de calcul de consistances. Nous allons définir dans cette section certaines des consistances les plus répandues en commençant par l'arc-consistance, puis nous verrons les affaiblissements rendus nécessaires par la considération de contraintes réelles. Les trois premières consistances que nous allons présenter (à savoir l'*arc-consistance*, la *hull-consistance* et la *union-consistance*) correspondent à des *opérateurs optimaux* sur les domaines d'approximations, respectivement, $\mathcal{P}(\mathbb{R})$, $\mathbb{I}$ et $\mathbb{U}$.

La description informelle de la consistance d'arc dans l'introduction à cette partie la limitait aux contraintes binaires. Il est possible de l'étendre aux contraintes d'arité supérieure, ce que nous faisons ci-dessous.

Définition 3.5 (Consistance d'arc). Étant donnée une contrainte réelle $c(x_1, \ldots, x_n)$, un entier $k \in \{1, \ldots, n\}$ et un produit cartésien de domaines $\boldsymbol{D} = D_1 \times \cdots \times D_n$, on dira que la contrainte c est *arc-consistante par rapport à la variable* x_k si et seulement si :

$$D_k = \pi_k(\rho_c \cap \boldsymbol{D}) \tag{3.1}$$

La contrainte c sera dite *arc-consistante par rapport à* $\boldsymbol{D}$ si la relation (3.1) est vraie pour tout $k \in \{1, \ldots, n\}$.

L'introduction de contraintes d'arité supérieure à 2 conduit à adopter une représentation du réseau de contraintes par un graphe dual de celui présenté dans la figure 2.8 : désormais, les nœuds correspondent aux contraintes et les arcs traduisent les partages de variables entre contraintes (*cf.* l'exemple de la figure 3.1).

Le calcul de la consistance d'arc pour des variables et des contraintes entières n'est « que » coûteux. Il devient impossible en pratique pour des variables réelles. Il suffit de considérer par exemple une contrainte simple telle que

$$\left[\begin{array}{l} D_x = [3\,..\,4] \\ D_y = [4{,}5\,..\,8] \\ D_z = [-7\,..\,9] \\ D_k = [0\,..\,12] \end{array}\right] \qquad \mathcal{S} = \left\{\begin{array}{l} x^2 = y \\ x + y = z \\ z = k + y \\ t = k - 5 \end{array}\right.$$

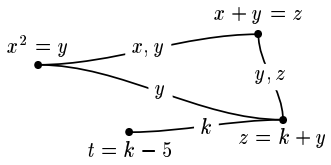


FIG. 3.1: Un système de contraintes et son *store* associé

$c\colon 10x = 1$ où $x \in [0\,..\,1]$. Le nombre $1/10$ n'étant pas représentable en machine (*cf.* chap. 1), il est impossible d'éliminer du domaine de x toutes les valeurs telles que c n'est pas satisfaite. On aura au mieux pour x le domaine $\big(\Downarrow\langle 1/10,]\rangle\Downarrow\,,\ \Uparrow\langle 1/10,]\rangle\Uparrow\big)$, soit : $(\lfloor 1/10 \rfloor\,..\,\lceil 1/10 \rceil)$.

Comme nous l'avons dit plus haut, l'arc-consistance suppose que l'on puisse représenter exactement la relation ρ_c (utilisation de l'approximation $\mathcal{P}(\mathbb{R})$). Comme c'est rarement le cas pour des contraintes réelles, un premier affaiblissement de l'arc-consistance consiste à approcher tous les n-uplets de $\boldsymbol{D}$ satisfaisant la contrainte c par les plus petits pavés représentables, puis à ne conserver que ces pavés. Formellement, on a :

Définition 3.6 (Union-consistance [21]). Étant donnés une contrainte réelle $c(x_1, \ldots, x_n)$, un entier $k \in \{1, \ldots, n\}$ et un produit cartésien de domaines $\boldsymbol{D} = D_1 \times \cdots \times D_n$, on dira que la contrainte c est *union-consistante par rapport à la variable* x_k si et seulement si :

$$D_k = \mathsf{Union}(\pi_k(\rho_c \cap \boldsymbol{D})) \tag{3.2}$$

La contrainte c sera dite *union-consistante par rapport à* $\boldsymbol{D}$ si la relation (3.2) est vraie pour tout $k \in \{1, \ldots, n\}$.

L'union-consistance correspond à la consistance la plus proche de l'arc-consistance qui soit calculable pratiquement en machine dans le cas des contraintes réelles. Les algorithmes de filtrage associés ont cependant l'inconvénient d'être coûteux car ils supposent la manipulation d'unions d'intervalles qui ralentissent énormément les calculs.

On peut alors se contenter d'utiliser une approximation plus grossière et de chercher à calculer le plus petit pavé (au sens de l'inclusion) contenant tous les n-uplets solutions présents dans le pavé de départ. On obtient ainsi la *hull-consistance* :

Définition 3.7 (Hull-consistance [21]). Étant donnés une contrainte réelle $c(x_1, \ldots, x_n)$, un entier $k \in \{1, \ldots, n\}$ et un pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n$, on dira que la contrainte c est *hull-consistante par rapport à la variable* x_k si et seulement si :

$$I_k = \mathsf{Hull}(\pi_k(\rho_c \cap \boldsymbol{B})) \tag{3.3}$$

La contrainte c sera dite *hull-consistante par rapport à* $\boldsymbol{B}$ si la relation (3.3) est vraie pour tout $k \in \{1, \ldots, n\}$.

Nous donnons ci-dessous un exemple d'implantation d'un opérateur de contraction pour la contrainte $x+y=z$ correspondant à la méthode donnée par CLEARY [48] (utilisation de l'arithmétique relationnelle d'intervalles).

Exemple 3.10 (Un CNO pour la contrainte $x+y=z$). *Étant donnée la contrainte $c\colon x+y=z$ et les domaines I_x, I_y et I_z, un opérateur de contraction $N[c]$ associé à la relation ρ_c calculant la hull-consistance est composé des trois* opérateurs de contraction de projection *:*

$$\left\{\begin{array}{l} {N_c}^1(\boldsymbol{B}) = I_x \cap (I_z - I_y) \\ {N_c}^2(\boldsymbol{B}) = I_y \cap (I_z - I_x) \\ {N_c}^3(\boldsymbol{B}) = I_z \cap (I_x + I_y) \end{array}\right.$$

*On exprime chaque variable de la contrainte en terme des autres et l'on fait les calculs en utilisant l'arithmétique des intervalles (*cf. *chap. 2).*

Comme le fait remarquer VAN EMDEN [231], les opérateurs de projection ${N_c}^1$, ${N_c}^2$ et ${N_c}^3$ pour une contrainte de la forme $x \Diamond y = z$ peuvent tous exister même si la fonction $\Diamond$ n'a pas d'inverse. Considérons par exemple le

cas où $\Diamond$ correspond à la multiplication sur le domaine d'approximation $\mathbb{I}$: l'opérateur ${N_c}^1$ calcule le plus petit intervalle contenant l'ensemble $\{\mathsf{r}_x \in I_x \mid \exists \mathsf{r}_y \in I_y, \exists \mathsf{r}_z \in I_z : \mathsf{r}_x \times \mathsf{r}_y = \mathsf{r}_z\}$. Par conséquent, il est défini même dans le cas où 0 appartient à I_y.

Contrairement à la union-consistance, il apparaît possible de calculer efficacement la hull-consistance en utilisant l'arithmétique des intervalles. En pratique, on se heurte cependant à plusieurs problèmes : le calcul de la hull-consistance suppose que l'on soit capable de calculer avec un *arrondi correct* des expressions arbitrairement compliquées, ce que l'on est en général incapable de faire (on se reportera aux travaux de PICHAT [183] et KAHAN [108] pour le cas particulier de la distillation, où de tels algorithmes existent). De plus, il n'est pas toujours aisé d'exprimer chaque variable en terme des autres. Afin d'éliminer ces deux problèmes, CLEARY [48] suggère de décomposer les contraintes complexes en conjonctions de contraintes simples, binaires ou ternaires, qu'il appelle *primitives* (*cf.* l'exemple de la table 3.1). Une contrainte $c \equiv c_1 \wedge \cdots \wedge c_m$ est alors dite hull-consistante par rapport à un pavé $\boldsymbol{D}$ si chaque contrainte primitive $c_i, i \in \{1, \ldots, m\}$ de son *ensemble de décomposition* $c_{\text{dec}} = \{c_1, \cdots, c_m\}$ est elle-même hull-consistante par rapport à $\boldsymbol{D}$.

TAB. 3.1: Décomposition en *contraintes « primitives »*

Système de l'utilisateur	*Système décomposé*	*Primitives utilisées*
$\begin{cases} 1) & x^2 + y^2 = 1 \\ 2) & y + 5 = z \\ 3) & x^2 - y = z \\ 4) & z < 7 \end{cases}$	$\begin{cases} 1) & x^2 = s_1 \\ & y^2 = s_2 \\ & s_1 + s_2 = 1 \\ 2) & y + 5 = z \\ 3) & x^2 = s_3 \\ & z + y = s_3 \\ 4) & z < 7 \end{cases}$	$\begin{cases} x^2 = y \\ x + y = z \\ x < y \end{cases}$

Cependant, la décomposition en *contraintes primitives* binaires ou ternaires d'une contrainte complexe ne suffit pas à assurer la hull-consistance. Pour cela, il faut aussi que l'on soit capable de calculer la hull-consistance pour chacune des primitives, ce qui suppose d'avoir à sa disposition des opérateurs (addition, multiplication, exponentielle, sinus…) correctement arrondis. Or, nous avons vu au chapitre 1 que la norme IEEE 754 [116] ne garantit un tel arrondi que pour cinq opérations ($+, -, \times, \div, \sqrt{}$) alors que la qualité de l'implémentation des autres opérations est laissée à l'appréciation des programmeurs des librairies mathématiques (*cf.* la discussion sur le *Table Maker's Dilemma*, p. 10). Il peut donc s'avérer particulièrement difficile d'implémenter un opérateur de contraction assurant la hull-consistance pour une contrainte aussi simple que $e^x = y$. Aussi, nous définissons ici une notion de contrainte primitive différemment de CLEARY en nous basant sur l'idée d'*extension canonique* de VAN EMDEN :

Définition 3.8 (Extension canonique [231]). Soit c une contrainte réelle n-aire, $\boldsymbol{B} = I_1 \times \cdots \times I_n$ un pavé et $k \in \{1, \ldots, n\}$ un entier. La *k-ième extension canonique de ρ_c par rapport à $\boldsymbol{B}$* est définie par :

$$ {\rho_c}^{(k)}(\boldsymbol{B}) = \{\mathsf{r}_k \in \mathbb{R} \mid \exists \mathsf{r}_1 \in I_1, \ldots, \exists \mathsf{r}_{k-1} \in I_{k-1}, \exists \mathsf{r}_{k+1} \in I_{k+1}, \ldots, \exists \mathsf{r}_n \in I_n \quad \text{t.q.} \quad (\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \rho_c\} $$

On a alors la définition :

Définition 3.9 (Contrainte primitive). Une contrainte réelle n-aire c est dite primitive sur un domaine d'approximation $\mathcal{A}$ si et seulement si on peut trouver n *opérateurs de contraction de projection* ${N_c}^1, \ldots, {N_c}^n$, définis de $\mathcal{A}^n$ vers $\mathcal{A}$, tels que :

$$ \forall \boldsymbol{D} = D_1 \times \cdots \times D_n, \forall k \in \{1, \ldots, n\} : {N_c}^k(\boldsymbol{D}) = \text{apx}_{\mathcal{A}}({\rho_c}^{(k)}(\boldsymbol{D}) \cap D_k) $$

On notera que cette définition diffère de celle adoptée par CLEARY ainsi que de celle donnée par JAFFAR et MAHER [118] (pour qui une contrainte primitive est une formule atomique). En particulier, nous ne mettons aucune restriction a priori sur l'arité ou la forme des contraintes primitives. De plus, le caractère de *contrainte primitive* est lié au domaine d'approximation utilisé.

On verra cependant au chapitre 4 que la notion de primitive au sens de CLEARY (une contrainte primitive met en jeu au plus un opérateur arithmétique) possède une certaine importance dans la pratique. Ainsi, pour un ensemble de variables $\mathcal{V} = \{x_1, x_2, \dots\}$ et un ensemble d'opérateurs arithmétiques $\mathcal{F} = \{\varphi_1, \dots, \varphi_p\}$ (avec α_i l'arité de φ_i, pour $i \in \{1, \dots, p\}$), notons $\mathcal{C}_\mathcal{F}$ l'ensemble des contraintes de la forme $\{\varphi_1(x_1, \dots, x_{\alpha_1}) = x_{\alpha_1+1}, \dots, \varphi_p(x_1, \dots, x_{\alpha_p}) = x_{\alpha_p+1}\}$. On appellera alors *décomposition naturelle par rapport à $\mathcal{C}_\mathcal{F}$ d'une contrainte* c la couverture [77] de la représentation arborescente de c par les représentations arborescentes des contraintes de $\mathcal{C}_\mathcal{F}$ (*cf.* exemple fig. 3.2). En pratique cette couverture n'est pas unique (par exemple, dans la figure 3.2 le sous-arbre pour $2x = \zeta_1$ pourrait être couvert par celui de la contrainte $\alpha\beta = \gamma$ ou par celui de la contrainte $2\alpha = \gamma$ si cette contrainte se trouve dans $\mathcal{C}_\mathcal{F}$). On ne considérera cependant pas ce genre d'ambiguïté dans la suite.

Exemple 3.11 (Décomposition naturelle). *Étant donnés la contrainte c: $2x = z - y^2$, l'ensemble de contraintes primitives $\mathcal{C} = \{\alpha\beta = \gamma, \alpha^2 = \beta, \alpha - \beta = \gamma, \alpha = \beta\}$ et trois nouvelles variables ζ_1, ζ_2 et ζ_3, on a la décomposition naturelle de c :*

$$c \equiv 2x = \zeta_1 \wedge y^2 = \zeta_2 \wedge z - \zeta_2 = \zeta_3 \wedge \zeta_1 = \zeta_3$$

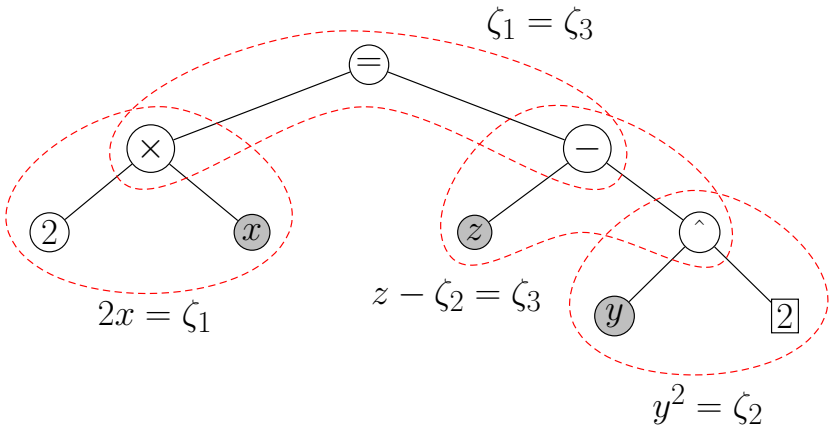


FIG. 3.2: Décomposition naturelle de la contrainte c: $2x = z - y^2$

L'algorithme 3.2 (HC3) présente une légère variation de l'algorithme Nar calculant la hull-consistance pour un ensemble de *contraintes primitives* $c_1, \dots, c_m$. Les CNOs $N[c_i], i \in \{1, \dots, m\}$ sont supposés calculer la hull-consistance pour les contraintes $c_i, i \in \{1, \dots, m\}$; ils sont donc idempotents, ce qui permet de les retirer de la liste de propagation après avoir été considérés. L'algorithme HC3revise (*cf.* alg. 3.3) apparaissant dans l'algorithme HC3 ne fait qu'appliquer les opérateurs de contraction de projection ${N_c}^1, \dots, {N_c}^n$ d'une contrainte n-aire c composant son CNO $N[c]$ (voir l'exemple 3.10).

On peut prouver que l'algorithme HC3revise calcule des pavés hull-consistants pour des contraintes primitives :

Proposition 3.2 (Correction de HC3revise). *Soient un pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n$ et une* contrainte n-aire primitive *c pour laquelle on connaît n opérateurs de contraction de projection sur le domaine d'approximation $\mathbb{I}$. On a :*

$$\begin{aligned} \pi_1(\mathsf{HC3revise}(c, \boldsymbol{B})) &= \mathsf{Hull}(\pi_1(\rho_c \cap \boldsymbol{B})) \\ &\dots \\ \pi_n(\mathsf{HC3revise}(c, \boldsymbol{B})) &= \mathsf{Hull}(\pi_n(\rho_c \cap \boldsymbol{B})) \end{aligned}$$

Démonstration. Soit $\boldsymbol{B'} = I'_1 \times \cdots \times I'_n$ le pavé résultant de l'application de l'algorithme HC3revise sur la contrainte c et le pavé $\boldsymbol{B}$. Au vu de l'algorithme 3.3, on a :

$$\forall j \in \{1, \dots, n\} : \pi_j(\mathsf{HC3revise}(c, \boldsymbol{B})) = {N_c}^j(\boldsymbol{B})$$

ALG. 3.2: Algorithme de filtrage HC3

HC3(**entrée** $\{c_1, \ldots, c_m\}$; **entrée/sortie** $\boldsymbol{B} = I_1 \times \cdots \times I_n$)
début
  $\mathcal{Q} \leftarrow$ PushBack($\mathcal{Q}$, $\{c_1, \ldots, c_m\}$) *% contraintes ajoutées à la liste de propagation*
  $\mathcal{ST} \leftarrow \mathcal{ST} \cup \{c_1 \ldots, c_m\}$ *% contraintes ajoutées au store*
  tant que ($\mathcal{Q} \neq \varnothing$ **et** $\boldsymbol{B} \neq \varnothing$) **faire**
    $c \leftarrow$ Head($\mathcal{Q}$)
    $\boldsymbol{B'} \leftarrow$ HC3revise$(c, \boldsymbol{B})$
    si ($\boldsymbol{B'} \neq \boldsymbol{B}$) **alors**
      $\mathcal{Q} \leftarrow$ PushBack($\mathcal{Q}$, $\{c_j \in \mathcal{ST} \mid \exists x_k \in \mathrm{Var}(c_j) \wedge I'_k \neq I_k\}$)
      $\boldsymbol{B} \leftarrow \boldsymbol{B'}$
    finsi
    PopHead($\mathcal{Q}$)
  ftq
fin.

La liste de propagation $\mathcal{Q}$ est gérée en file (stratégie FIFO). La fonction PushBack( ) *insère chaque élément de l'ensemble passé en deuxième argument dans la file donnée en premier argument ; la fonction* Head( ) *retourne l'élément en tête de la file passée en argument sans le retirer ; la fonction* PopHead( ) *retire le premier élément de la file passée en argument.*

Sous l'hypothèse que la contrainte c est primitive et que le domaine d'approximation considéré est $\mathbb{I}$, on déduit de la définition 3.9 :

$$\forall j \in \{1, \ldots, n\} : \pi_j(\text{HC3revise}(c, \boldsymbol{B})) = \text{Hull}({\rho_c}^{(j)}(\boldsymbol{B}) \cap I_k) \tag{3.4}$$

Le théorème 26[‡] de l'article de HICKEY *et al.* [107] nous donne la relation :

$$\forall j \in \{1, \ldots, n\} : \pi_j(\rho_c \cap \boldsymbol{B}) = {\rho_c}^{(j)}(\boldsymbol{B}) \cap I_j$$

En reportant ce résultat dans l'équation (3.4), on obtient finalement :

$$\forall j \in \{1, \ldots, n\} : \pi_j(\text{HC3revise}(c, \boldsymbol{B})) = \text{Hull}(\pi_j(\rho_c \cap \boldsymbol{B}))$$

ce qui conclut la démonstration. ■

ALG. 3.3: Algorithme HC3revise

HC3revise(**entrée** c : `contrainte réelle` n`-aire`;
  entrée $\boldsymbol{B} = I_1 \times \cdots \times I_n$: `pavé de domaines`; **sortie** $\boldsymbol{B'} = I'_1 \times \cdots \times I'_n$)
début
  $I'_1 \leftarrow {N_c}^1(\boldsymbol{B})$
  ...
  $I'_n \leftarrow {N_c}^n(\boldsymbol{B})$
  retourner ($\boldsymbol{B'}$)
fin.

La décomposition en contraintes primitives de contraintes complexes possède plusieurs inconvénients. En particulier, elle réduit les opportunités de réduction des domaines en détruisant de l'information concernant les dépendances entre variables. Elle induit aussi une augmentation de l'espace mémoire requis pour représenter le *store*.

[‡] On trouvera l'énoncé et la preuve de ce théorème en annexe (théo. A.1), p. 223.

On verra dans le chapitre 10 dévolu aux problèmes de débogage en programmation par contraintes qu'elle entraîne aussi des problèmes de présentation du *store* : le *store* contient des contraintes et des variables issues du processus de décomposition inconnues du programmeur.

Afin de pallier ces inconvénients, BENHAMOU *et al.* [29] ont défini la *box-consistance* dont le calcul ne requiert pas la décomposition des contraintes complexes en primitives. Si l'on réécrit l'équation (3.3) de la hull-consistance d'une contrainte c par rapport à la variable x_k sous la forme équivalente :

$$I_k = \mathsf{Hull}(I_k \cap \{a_k \in \mathbb{R} \mid \forall j \in \{1,\ldots,n\} \setminus \{k\} \colon \exists a_j \in I_j \text{ t.q. } (a_1,\ldots,a_n) \in \rho_c\})$$

on constate que l'un des obstacles à un calcul aisé de la hull-consistance est la présence de la quantification existentielle en ce qui concerne les valeurs des domaines $I_1,\ldots,I_{k-1},I_{k+1},\ldots,I_n$. L'idée de base de la box-consistance est d'éliminer cette quantification en considérant non plus les valeurs des domaines mais les domaines eux-mêmes, ce qui conduit à considérer non plus la contrainte réelle c mais une de ses extension aux intervalles C. On a ainsi la définition suivante :

Définition 3.10 (Box-consistance [29]). Soit c une contrainte réelle n-aire, C une extension aux intervalles de c et $\boldsymbol{B} = I_1 \times \cdots \times I_n$ un pavé de domaines. La contrainte c est dite *box-consistante par rapport à $\boldsymbol{B}$ et à un entier $k \in \{1,\ldots,n\}$* si et seulement si :

$$I_k = \mathsf{Hull}(\{a_k \in I_k \mid C(I_1,\ldots,I_{k-1},\mathsf{Hull}(\{a_k\}),I_{k+1},\ldots,I_n)\}) \tag{3.5}$$

La box-consistance est paramétrée par l'extension aux intervalles choisie pour la contrainte C. Nous verrons au chapitre 4 une extension de la définition 3.10 où l'on s'autorise à choisir une extension différente pour chaque projection de la contrainte.

Opérationnellement, le calcul de la box-consistance pour un système de contraintes se fait en utilisant un algorithme identique à HC3 (*cf.* tab. 3.2, p. 43) où le CNO pour une contrainte est implémenté par l'algorithme BC3revise (*cf.* algo. 3.5) : étant donnés une contrainte n-aire c sur les variables $x_1,\ldots,x_n$, une extension aux intervalles C de c et un pavé de domaines $\boldsymbol{B} = I_1 \times \cdots \times I_n$, on considère pour chaque variable x_k la *projection* $C|_{k,\boldsymbol{B}}$ de C (contrainte unaire obtenue en remplaçant dans l'expression de C toutes les variables sauf x_k par leurs domaines respectifs) et l'on cherche les intervalles canoniques extrêmes vérifiant $C|_{k,\boldsymbol{B}}$. On pourra noter que l'algorithme BC3revise considère implicitement l'extension naturelle aux intervalles des contraintes.

ALG. 3.4: Algorithme de filtrage BC3

```
BC3( entrée {(c_1,V_1),...,(c_m,V_m)} ; entrée/sortie B = I_1 × ··· × I_n )
début
   Q ← PushBack( Q, {(c_1,V_1),...,(c_m,V_m)} )
   ST ← ST ∪ {(c_1,V_1),...,(c_m,V_m)}
   tant que ( Q ≠ ∅ et B ≠ ∅ ) faire
      (c,V) ← Head( Q )
      B' ← BC3revise(c,V,B)
      si ( B' ≠ B ) alors
         Q ← PushBack( Q, {(c_j,V_j) ∈ ST | ∃x_k ∈ V_j ∧ I'_k ≠ I_k} )
         B ← B'
      finsi
      PopHead( Q )
   ftq
   retourner ( B )
fin .
```

On se reportera à l'algorithme 3.2 pour la description des fonctions apparaissant dans l'algorithme BC3.

L'algorithme BC3 tel que présenté ici ne prend plus en entrée un ensemble de contraintes mais un ensemble de couples $(c,\mathcal{V})$ composés d'une contrainte c et d'un ensemble de variables $\mathcal{V}$ ayant au moins une occurrence dans

c. L'algorithme BC3revise ne recherche les intervalles canoniques extrêmes vérifiant une projection $C|_{k,\boldsymbol{B}}$ d'une extension aux intervalles C de la contrainte c que pour les valeurs d'indice k telles que la variable x_k appartient à l'ensemble $\mathcal{V}$. Dans ce chapitre, on considérera toujours que les couples $(c, \mathcal{V})$ sont de la forme $(c, \mathsf{Var}(c))$ (*i.e.* l'algorithme BC3revise considère toutes les projections de la contrainte c). On verra au chapitre 4 une utilisation d'une autre convention.

ALG. 3.5: Algorithme BC3revise

```
BC3revise(entrée c : contrainte réelle n-aire;
      entrée V = {x_1,...,x_p}, μ_c(x_1) > 0,...,μ_c(x_p) > 0,  p ⩽ n ;
      entrée B = I_1 × ··· × I_n : pavé de domaines;
      sortie B' = I'_1 × ··· × I'_n)
début
  B' ← B
  pour chaque x_j ∈ V faire
      p ← C|_{j,B'}   % p : contrainte unaire, projection de C par rapport à la variable x_j et au pavé B'
      I''_j ← RéductionGauche(p, x_j, I'_j)
      si (I''_j ≠ ∅) alors
          I'_j ← RéductionDroite(p, x_j, [I''_j .. Ī'_j])
      sinon
          retourner (∅)
      finsi
  fpc
  retourner (B')
fin.
```

Voir l'algorithme 3.6 pour la description de la fonction RéductionGauche(). *La fonction* RéductionDroite() *s'en déduit simplement par symétrie.*

La figure 3.3 illustre la méthode pour le calcul de la box-consistance dans le cas de la contrainte $f(x) = 0$, où $f(x) = (x^6/4 - 1{,}9x^5 + 0{,}25x^4 + 20x^3 - 17x^2 - 55{,}6x + 40)/50$ et où le domaine initial de x est $I_x = [-3\,..\,6]$. L'algorithme BC3revise utilisé emploie des itérations de la méthode de NEWTON sur les intervalles pour accélérer le calcul des quasi-zéros extrêmes.

Les chiffres cerclés correspondent aux pas de calculs : l'algorithme commence par chercher l'intervalle canonique le plus à gauche dans le domaine de x pour lequel la contrainte $F(X) = 0$ est vérifiée (avec F une extension aux intervalles de f) ; il cherche ensuite l'intervalle canonique le plus à droite vérifiant la même condition. Au final, l'intervalle ⑬ est bien le plus petit intervalle contenant tous les nombres réels du domaine de x vérifiant $F(x) = 0$.

Si l'on compare les domaines calculés suivant la consistance utilisée pour résoudre la contrainte $f(x) = 0$, on obtient le dessin de la figure 3.4. L'arc-consistance apparaît bien comme la notion la plus précise, à condition que l'on puisse représenter exactement en machine chacune des quatre solutions. Sinon, on peut utiliser l'union-consistance, qui approche ces quatre points par des intervalles canoniques. On peut aussi choisir de représenter toutes les solutions réelles avec un seul intervalle (hull-consistance), ou se contenter d'une consistance plus lâche en considérant une extension aux intervalles de la contrainte et non plus la contrainte réelle elle-même (box-consistance). On notera que la box-consistance donne ici un intervalle très grand à cause du quasi-zéro, indiscernable dans le calcul sur les intervalles d'un vrai zéro. On trouvera chez COLLAVIZZA *et al.* [51] une étude comparative théorique des différentes consistances.

Pour le calcul de la box-consistance, la recherche des intervalles canoniques extrêmes se fait en utilisant la méthode de NEWTON étendue aux intervalles [167] couplée avec un processus de découpage des intervalles.

Obtenir la canonicité des intervalles encadrant les (quasi-)zéros extrêmes représente un travail coûteux. On peut alors songer à relâcher l'exigence de canonicité et chercher à encadrer les (quasi-)zéros avec une précision φ. On définit ainsi une box-consistance « relaxée » : la box$_\varphi$-consistance.

ALG. 3.6: Fonction RéductionGauche

```
RéductionGauche(entrée p : contrainte réelle unaire en x;
                entrée x : variable réelle;
                entrée I : domaine (intervalle) de x; sortie I' : intervalle)
début
   si (p([I .. I+])) alors
      retourner (I)
   sinon
      (Il, Ir) ← Découper([I+ .. Ī])    % Découpage de l'intervalle en deux parties
      I' ← RéductionGauche(Il)
      si (I' = ∅) alors
         retourner (RéductionGauche(Ir))
      sinon
         retourner ([I' .. Ī])
      finsi
   finsi
fin.
```

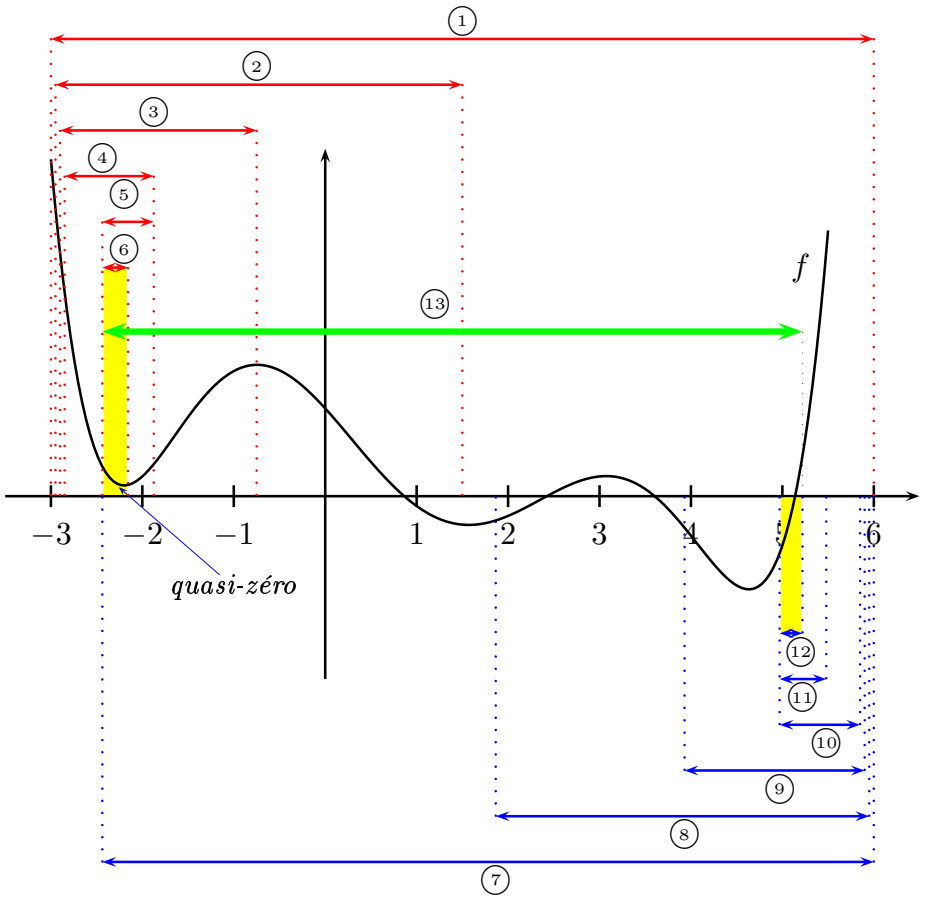


FIG. 3.3: Calcul de la box-consistance pour $f(x) = 0$

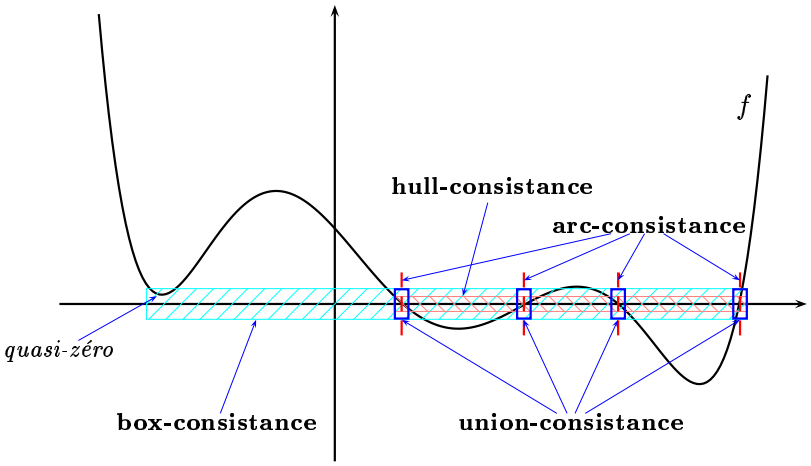


FIG. 3.4: Comparaison des différentes consistances

Définition 3.11 (Box$_\varphi$-consistance [95, 94]). Soit c une contrainte réelle n-aire, C une extension aux intervalles de c, un entier $k \in \{1, \ldots, n\}$, $\boldsymbol{B} = I_1 \times \cdots \times I_n$ un pavé de domaines et φ un nombre flottant positif. La contrainte c est dite box$_\varphi$-consistante par rapport à k, C et $\boldsymbol{B}$ si et seulement si :

$$I_k = \mathsf{Hull}(I_k \cap \{a_k \in \mathbb{R} \mid C(I_1, \ldots, I_{k-1}, \mathsf{Hull}(\{a_k - \varphi, a_k + \varphi\}), I_{k+1}, \ldots, I_n)\})$$

On définit alors une famille d'opérateurs calculant la box$_\varphi$-consistance de la façon suivante :

Définition 3.12 (Opérateur de contraction pour la box$_\varphi$-consistance). Étant donnés un pavé $\boldsymbol{B}$, une contrainte réelle n-aire c, un nombre flottant positif φ, on appelle *opérateur de contraction pour la box$_\varphi$-consistance* un opérateur $N_\varphi^c : \mathbb{I}^n \to \mathbb{I}^n$ défini par : $N_\varphi^c(\boldsymbol{B}) = \boldsymbol{B'}$, où $\boldsymbol{B'}$ est le plus grand pavé inclus dans $\boldsymbol{B}$ tel que c est box$_\varphi$-consistant par rapport à k, C et $\boldsymbol{B'}$, pour tout $k \in \{1, \ldots, n\}$.

Nous donnons ci-dessous la preuve que N_φ^c est un opérateur de contraction malgré l'introduction du paramètre φ.

Proposition 3.3. *Étant donnée une contrainte réelle n-aire c et un nombre flottant φ, l'opérateur N_φ^c de la définition 3.12 est un opérateur de contraction pour ρ_c.*

Démonstration. D'après la définition 3.3, il nous faut prouver les trois propriétés suivantes :

$$\forall \boldsymbol{B}, \boldsymbol{D} \in \mathbb{I}^n :$$
$$N_\varphi^c(\boldsymbol{B}) \subseteq \boldsymbol{B} \tag{3.6}$$
$$\boldsymbol{B} \cap \rho_c \subseteq N_\varphi^c(\boldsymbol{B}) \tag{3.7}$$
$$\boldsymbol{B} \subseteq \boldsymbol{D} \Rightarrow N_\varphi^c(\boldsymbol{B}) \subseteq N_\varphi^c(\boldsymbol{D}) \tag{3.8}$$

Notons tout d'abord que les extensions des relations associées aux contraintes considérées ici sont des *extensions monotones* (*cf.* déf. 2.9). Dans la suite, étant donnés les pavés $\boldsymbol{B} = I_1 \times \cdots \times I_n$ et $\boldsymbol{D} = J_1 \times \cdots \times J_n$, notons $\boldsymbol{B'} = N_\varphi^c(\boldsymbol{B})$ et $\boldsymbol{D'} = N_\varphi^c(\boldsymbol{D})$ et soit $\mathcal{E}_{\boldsymbol{B}}^k$ l'ensemble défini par

$$\mathcal{E}_{\boldsymbol{B}}^k = \{a_k \in \mathbb{R} \mid C(I_1, \ldots, I_{k-1}, \mathsf{Hull}(\{a_k - \varphi, a_k + \varphi\}), I_{k+1}, \ldots, I_n)\}$$

Contractance (3.6)**.** Conséquence immédiate de la définition 3.12 ;

Complétude (3.7)**.** Nous faisons la preuve par contradiction : soit b un élément de $\boldsymbol{B} \cap \rho_c$ avec $b \notin \boldsymbol{B'}$. Considérons le pavé $\boldsymbol{D} \subseteq \boldsymbol{B}$ défini par :

$$\boldsymbol{D} = \mathsf{Hull}(\boldsymbol{B'} \cup \{b\}) \tag{3.9}$$

En utilisant la monotonie de C et le théorème fondamental de l'arithmétique des intervalles (*cf.* théo. 2.2), on sait que

$$C(J_1, \ldots, J_{k-1}, \mathsf{Hull}(\{b_k - \varphi, b_k + \varphi\}), J_{k+1}, \ldots, J_n)$$

est vrai et que $\mathcal{E}^k_{I'} \subseteq \mathcal{E}^k_J$. Par conséquent :

$$\forall k \in \{1, \ldots, n\} :$$
$$J_k = \mathsf{Hull}(J_k \cap \{a_k \in \mathbb{R} \mid \quad C(J_1, \ldots, J_{k-1}, \mathsf{Hull}(\{a_k - \varphi, a_k + \varphi\}), J_{k+1}, \ldots, J_n)\})$$

(Sinon, il existerait un pavé inclus strictement dans $\boldsymbol{D}$ contenant tous les éléments de $\boldsymbol{B'}$ ainsi que $\{b\}$, ce qui contredit le fait que $\boldsymbol{D}$ est le plus petit pavé possédant cette propriété — *cf.* éq. (3.9).) Par conséquent, pour tout $k \in \{1, \ldots, n\}$, la contrainte c est box$_\varphi$-consistante par rapport à k, C et $\boldsymbol{D}$. Comme l'on a $\boldsymbol{D} \supseteq \boldsymbol{B'}$, cela contredit le fait que $\boldsymbol{B'}$ est le plus grand pavé inclus dans $\boldsymbol{B}$ pour lequel c est box$_\varphi$-consistante, ce qui termine la démonstration ;

Monotonie (3.8)**.** Par contractance de N^c_φ, nous avons $\boldsymbol{B'} \subseteq \boldsymbol{D}$. La preuve que $\boldsymbol{B'} \subseteq \boldsymbol{D'}$ découle du fait que $\boldsymbol{D'} \subset \boldsymbol{B'}$ impliquerait que $\boldsymbol{D'}$ n'est pas le plus grand pavé inclus dans $\boldsymbol{D}$ pour lequel la contrainte c est box$_\varphi$-consistante.

■

CHAPITRE 4

Extension de la définition de box-consistance

Calcul de la hull-consistance sans décomposition — liens entre la hull-consistance et la box-consistance — nouvelle définition pour la box-consistance

NOUS AVONS VU AU CHAPITRE 3 deux algorithmes pour réduire les domaines de variables apparaissant dans une contrainte réelle ; leurs qualités et défauts seront vus en détail au chapitre 8 lorsque nous présenterons DecLIC. Nous pouvons cependant établir les faits suivants à partir de ce que nous avons déjà vu : l'algorithme HC3revise (*cf.* alg. 3.3, p. 43) calcule la hull-consistance de façon efficace pourvu que l'on puisse définir les *opérateurs de contraction de projection* (OCPs) idoines. Comme on l'a précédemment évoqué, la détermination de tels opérateurs se heurte à deux difficultés (expression e_x de chaque variable x en termes des autres variables et calcul du plus petit intervalle flottant contenant e_x) qui sont traditionnellement résolues en décomposant les contraintes de l'utilisateur en conjonctions de contraintes pour lesquelles il est aisé de déterminer les OCPs. Une telle décomposition implique un renommage implicite de toutes les variables ayant plus d'une occurrence dans une contrainte, ce qui tend à limiter l'efficacité de HC3revise pour les contraintes où les variables ont une grande multiplicité. D'un autre côté, l'algorithme BC3revise (*cf.* alg. 3.5, p. 45) gère efficacement les variables de grande multiplicité sans avoir à décomposer les contraintes mais n'est pas aussi avantageux (coût des calculs) que HC3revise en ce qui concerne les variables ne possédant qu'une occurrence dans une contrainte.

À partir de ces considérations, nous avons cherché à définir un nouvel algorithme de réduction de domaine possédant les qualités de HC3revise et de BC3revise. L'idée qui s'impose est de faire coopérer sur une même contrainte un algorithme basé sur HC3revise pour réduire les domaines des variables de multiplicité unitaire et un algorithme basé sur BC3revise pour les domaines des variables de multiplicité supérieure à 1.

Le premier pas dans cette direction a été la mise au point d'un algorithme HC4revise [23] possédant les qualités de simplicité et d'efficacité de HC3revise sans nécessiter la décomposition des contraintes de l'utilisateur. On montre dans la suite que l'algorithme HC4 obtenu en remplaçant HC3revise dans HC3 (*cf.* alg. 3.2, p. 43) par HC4revise est équivalent à HC3 en ce qui concerne les domaines finaux calculés pour les variables.

En utilisant HC4revise, nous avons écrit un algorithme BC4 réduisant les domaines des variables apparaissant dans un système de contraintes réelles $\mathcal{C}$ en faisant coopérer en tourniquet les algorithmes HC4revise et BC3. Nous verrons dans la partie dévolue aux résultats expérimentaux que BC4 se révèle plus efficace que BC3 et HC3 sur tous les systèmes de contraintes étudiés.

On montre alors qu'en étendant légèrement la définition de la box-consistance (déf. 3.10, p. 44), on peut prouver sous certaines hypothèses que l'algorithme BC4 calcule cette « nouvelle » box-consistance pour un système de contraintes. Accessoirement, l'extension de la définition nous permet de capturer les différentes définitions données, tant dans l'article original [29] que par COLLAVIZZA *et al.* [51].

4.1 Calcul de la hull-consistance sans décomposition

Considérons la contrainte réelle c: $2x = z - y^2$ ainsi qu'un pavé d'intervalles $\boldsymbol{B} = I_x \times I_y \times I_z$. Les variables x, y et z ne possédant qu'une occurrence dans c, il est possible de réécrire simplement la contrainte de façon à

exprimer chaque variable en terme des autres :

$$\begin{cases} x = (z - y^2)/2 \\ y = \sqrt{z - 2x} \\ z = 2x + y^2 \end{cases}$$

Il est important de noter que les opérations utilisées ici doivent s'entendre dans un contexte relationnel (*cf.* section 2.2.3).

À partir des expressions de c ci-dessus, on peut définir les trois opérateurs :

$$\begin{cases} N_1(\boldsymbol{B}) = I_x \cap (I_z - I_y^2)/2 \\ N_2(\boldsymbol{B}) = I_y \cap \sqrt{I_z - 2I_x} \\ N_3(\boldsymbol{B}) = I_z \cap (2I_x + I_y^2) \end{cases}$$

En utilisant les propriétés de l'arithmétique de l'intervalle, on montre que l'opérateur N_c défini par : $N_c(\boldsymbol{B}) = N_1(\boldsymbol{B}) \times N_2(\boldsymbol{B}) \times N_3(\boldsymbol{B})$ est un opérateur de contraction pour c. On pourra noter que N_c est une généralisation de l'algorithme HC3revise au cas où les opérateurs utilisés ne sont pas des *opérateurs de contraction de projection* (en effet, N_1, N_2 et N_3 ne sont pas a priori des opérateurs de contraction de projection du fait des cumuls d'arrondis consécutifs aux différents calculs flottants)

Il est toujours possible de définir un opérateur de contraction pour une contrainte c de la façon décrite ci-dessus quel que soit le nombre d'occurrences de chaque variable dans c à condition de remplacer chaque occurrence d'une même variable v_1 par une nouvelle variable α_1 de même domaine et d'ajouter au store la contrainte $v_1 = \alpha_1$ (c'est ce qui est fait implicitement lors de la décomposition d'une contrainte utilisateur en contraintes primitives pour le calcul de la hull-consistance).

En utilisant l'arbre représentant une contrainte c, nous allons décrire ci-dessous l'algorithme HC4revise implémentant un opérateur de contraction pour c en effectuant deux passes (une passe montante suivie d'une passe descendante) dans sa représentation arborescente. On verra ensuite que l'intégration de HC4revise dans HC3 permet de ré-obtenir un calcul de hull-consistance pour les contraintes utilisateurs.

Note : Un algorithme ressemblant à HC4revise a été défini sur des DAGs de façon indépendante par Yahia LEBBAH [142].

ALG. 4.1: Algorithme HC4revise

HC4revise (**entrée** $c = r(t_1, \ldots, t_p)$: contrainte réelle ; **entrée/sortie** $\boldsymbol{B}$: pavé ;
entrée $\mathcal{A}$: domaine d'approximation)
début
  $\boldsymbol{D_A} \leftarrow \boldsymbol{B}$
  pour chaque $i \in \{1, \ldots, p\}$ **faire**
    ForwardEvaluation(t_i,$\boldsymbol{D_A}$)
  fpc
  BackwardPropagation(c,$\boldsymbol{D_A}$)
  $\boldsymbol{B} \leftarrow$ Hull$_\square(\boldsymbol{D_A})$
  retourner ($\boldsymbol{B}$)
fin

Soit c: $r(t_1, \ldots, t_p)$ une contrainte réelle où r est un symbole de relation et les t_is des termes. On considérera dans la suite l'*arbre attribué* [7] représentant la contrainte c où chaque nœud t à l'exception de la racine (contenant r) possède un attribut synthétisé t.*fwd* et un attribut hérité t.*bwd*.

À partir de la contrainte c et d'un pavé $\boldsymbol{B}$, l'algorithme HC4revise (*cf.* alg. 4.1) effectue deux passes consécutives dans l'arbre :

ALG. 4.2: Algorithme d'évaluation ascendante

```
ForwardEvaluation (entrée/sortie t : arbre attribué ;
                   entrée D_A = D_1 × ··· × D_n : pavé de domaines)
début
    suivant (t) dans
        ◊(t_1, ..., t_j) :      % un terme
            pour chaque i ∈ {1, ..., j} faire
                ForwardEvaluation(t_i, D_A)
            fpc
            t.fwd ← ⧫(t_1.fwd, ..., t_j.fwd)
        a:                      % une constante
            t.fwd ← apx_A({a})
        x_k:                    % une variable
            t.fwd ← D_k
    finsuivant
fin
```

1. une *passe ascendante d'évaluation* (*cf.* l'algorithme 4.2 et la figure 4.1) correspondant à une simple évaluation de la valeur de l'extension naturelle aux intervalles de chaque sous-terme et à son stockage dans l'attribut synthétisé du nœud correspondant ;
2. une *passe de propagation descendante* (*cf.* l'algorithme 4.3 et la figure 4.2) d'évaluation des attributs hérités de la racine vers les feuilles en utilisant les attributs synthétisés de la première passe. Pour chaque nœud t, on utilise un opérateur de contraction de projection (identique à ceux employés par l'algorithme HC3revise) pour calculer $t.bwd$. On peut distinguer deux cas :
   – pour le nœud racine $r(t_1, \ldots, t_p)$, on calcule les attributs $t_k.bwd$ ($k \in \{1, \ldots, p\}$) grâce au k-ième opérateur de contraction de projection ${N_{c'}}^k$ de la contrainte c' : $r(x_1, \ldots, x_p)$, où $x_1, \ldots, x_p$ sont des nouvelles variables de domaines respectifs $t_1.fwd, \ldots, t_p.fwd$,
   – pour un nœud t_{h+1} de la forme $\Diamond(t_1, \ldots, t_h)$, on calcule les attributs $t_k.bwd$ ($k \in \{1, \ldots, h\}$) grâce au k-ième opérateur de contraction de projection ${N_{c''}}^k$ de la contrainte c'' : $x_{h+1} = \Diamond(x_1, \ldots, x_h)$, où $x_1, \ldots, x_h$ sont de nouvelles variables de domaines respectifs $t_1.fwd, \ldots, t_h.fwd$ et où x_{h+1} est une nouvelle variable de domaine $t_{h+1}.bwd$. On pourra remarquer que les variables x_i correspondent aux variables introduites lors du processus de décomposition habituel.

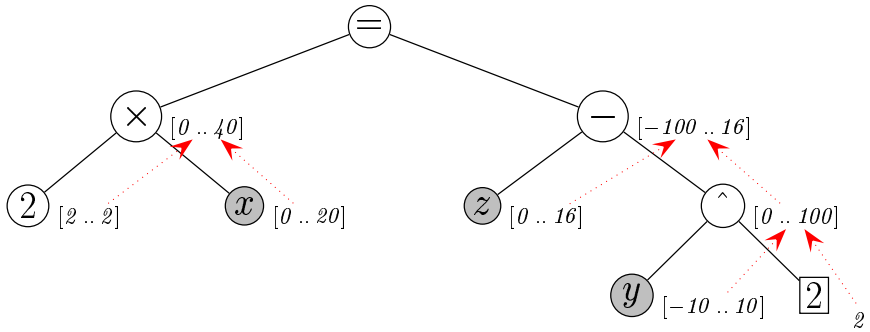


FIG. 4.1: Arbre annoté pour l'évaluation ascendante de la contrainte $2x = z - y^2$

Lors de la phase descendante, le calcul d'un domaine vide pour un attribut hérité implique que la contrainte c est inconsistante avec les domaines de départ. À chaque feuille variable (nœuds en grisé dans les figures 4.1 et

4.2), on intersecte le domaine de la variable x_i s'y trouvant avec le domaine de son attribut hérité $x_i.bwd$.

On pourra noter au passage la forte similitude entre le processus de propagation descendante de l'algorithme HC4revise et la méthode de différenciation automatique en mode inversé [97, 220] pour le calcul des dérivés partielles d'une fonction : après l'évaluation ascendante, on évalue dans un cas les opérateurs de contraction de projection et l'on calcule les dérivés partielles dans l'autre ; à la fin de la double traversée de l'arbre, on intersecte les domaines des variables dans un cas et l'on ajoute les dérivées dans l'autre.

ALG. 4.3: Algorithme de propagation descendante

```
BackwardPropagation (entrée/sortie t : arbre attribué ;
                     entrée/sortie D_A = D_1 × ··· × D_n : produit cartésien de domaines)
début
    suivant (t) dans
        r(t_1, ..., t_m):                    % la racine
            c ← (r(x_1, ..., x_m))
            D'_A ← (t_1.fwd, ..., t_m.fwd)
            pour chaque i ∈ {1, ..., m} faire
                t_i.bwd ← π_i(apx_A(ρ_c ∩ D'_A))
                BackwardPropagation(t_i, D_A)
            fpc
        ◊(t_1, ..., t_h):                    % un terme sous la racine
            c ← (◊(x_1, ..., x_h) = x_{h+1})
            D'_A ← (t_1.fwd, ..., t_h.fwd, t.bwd)
            pour chaque i ∈ {1, ..., h} faire
                t_i.bwd ← π_i(apx_A(ρ_c ∩ D'_A))
                BackwardPropagation(t_i, D_A)
            fpc
        x_j:                                 % une variable
            D_j ← D_j ∩ t.bwd                % Intersection des domaines pour gérer
                                             % le cas d'occurrences multiples de x_j
    finsuivant
fin
```

Exemple 4.12 (Un opérateur de contraction implémenté par HC4revise). *Soit la contrainte réelle* $c\colon 2x = z - y^2$ *et les domaines* $I_x = [0\,..\,20]$, $I_y = [-10\,..\,10]$, $I_z = [0\,..\,16]$. *Les figures 4.1 et 4.2 illustrent l'exécution de l'algorithme* HC4revise *pour réduire les domaines des variables* x, y, *et* z.
La première phase (fig. 4.1) correspond à l'évaluation des sous-termes en utilisant l'arithmétique des intervalles. Lors de la deuxième phase (fig. 4.2), on part de la racine et la première opération effectuée est l'intersection $[0\,..\,40] \cap [-100\,..\,16]$ *des attributs synthétisés des fils droit et gauche (correspondant à l'interprétation «* possible *» de l'extension aux intervalles de la notion d'égalité —* cf. *section 2.1.3). Le résultat de cette intersection est placé dans les attributs hérités des nœuds fils. Au nœud contenant l'opération* $\times$, *la propagation descendante calcule la valeur de l'attribut hérité du nœud contenant la variable* x *en utilisant l'opérateur de projection* ${N_{c'}}^x$ *de la contrainte* $c'\colon 2 \times x = \alpha_1$ *où le domaine de* x *est la valeur de son attribut synthétisé et le domaine de* α_1 *est l'intervalle* $[0\,..\,16]$ *(valeur de l'attribut hérité du nœud contenant* $\times$*). L'algorithme procède de même pour tous les autres nœuds de l'arbre jusqu'à obtenir les domaines finaux* $I_x = [0\,..\,8]$, $I_y = [-4\,..\,4]$ *et* $I_z = [0\,..\,16]$.

Afin de pouvoir utiliser l'algorithme HC4revise dans un algorithme tel que HC3 en préservant les propriétés de celui-ci (*cf.* prop. 3.1), il nous faut montrer qu'il implémente bien un opérateur de contraction.

Proposition 4.1. *Étant donnés une contrainte réelle* c, *un pavé* $\boldsymbol{B}$ *et un domaine d'approximation* $\mathcal{A}$, *l'algorithme* HC4revise *implémente un opérateur de contraction pour* c.

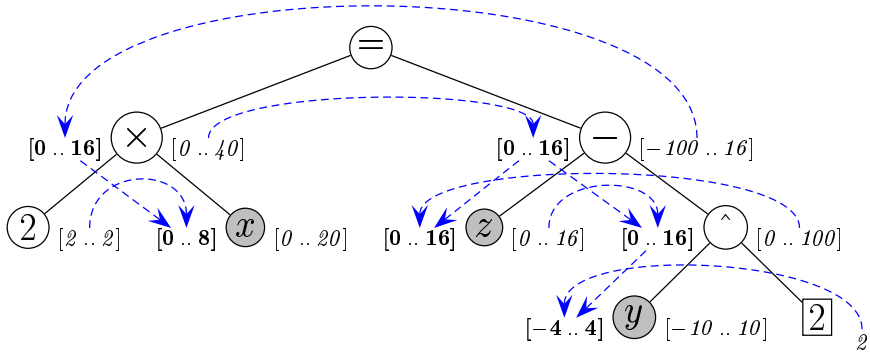


FIG. 4.2: Arbre annoté pour la propagation descendante de la contrainte $2x = z - y^2$

Démonstration. Il suffit de noter que l'algorithme HC4revise consiste essentiellement en une composition d'applications d'opérateurs de contraction. Sachant qu'une composition d'opérateurs de contraction constitue un opérateur de contraction [30], on déduit le résultat ci-dessus. ■

Comme évoqué plus haut, l'algorithme HC4revise ne calcule pas la hull-consistance pour une contrainte quelconque c. Cependant, si on l'intègre dans l'algorithme HC3 à la place de l'algorithme HC3revise, obtenant ainsi HC4 (*cf.* alg. 4.4), on a le résultat suivant :

ALG. 4.4: Algorithme de filtrage HC4

```
HC4(entrée {c_1, ..., c_m} ; entrée/sortie B = I_1 × ··· × I_n)
début
  Q ← PushBack(Q, {c_1, ..., c_m})   % contraintes ajoutées à la liste de propagation
  ST ← ST ∪ {c_1 ..., c_m}   % contraintes ajoutées au store
  tant que (Q ≠ ∅ et B ≠ ∅) faire
    c ← Head(Q)
    B' ← HC4Revise(c, B, 𝕀_□)
    si (B' ≠ B) alors
      Q ← PushBack(Q, {c_j ∈ ST | ∃x_k ∈ Var(c_j) ∧ I'_k ≠ I_k})
      B ← B'
    sinon % HC4revise pas idempotent : prochain appel sur c peut réduire encore les domaines
      PopHead(Q)
    finsi
  ftq
fin.
```

Proposition 4.2. *Étant donnés un ensemble de contraintes réelles $\mathcal{C}$ et un pavé d'intervalles $\boldsymbol{B}$, l'algorithme* HC4 *calcule le même pavé de domaines final pour les contraintes de $\mathcal{C}$ que l'algorithme* HC3 *appliqué à l'ensemble des contraintes décomposées en primitives $\mathcal{C}_{dec} = \bigcup_{c \in \mathcal{C}} c_{dec}$, où c_{dec} correspond à la* décomposition naturelle *(cf. p. 42) de la contrainte c :*

$$\mathsf{HC4}(\mathcal{C}, \boldsymbol{B}) = \mathsf{HC3}(\mathcal{C}_{dec}, \boldsymbol{B})$$

Démonstration. Pour prouver ce résultat, on note que HC4 et HC3 utilisent les mêmes opérateurs de contraction et que les attributs des nœuds de l'arbre dans HC4revise correspondent aux nouvelles variables de HC3. Le fait que l'ordre d'application des opérateurs de contraction soit indifférent permet alors d'en déduire le résultat. ■

De plus, soit HC4revise* l'algorithme HC4revise où les domaines des attributs synthétisés de chaque nœud ne sont pas intersectés avec le domaine de l'attribut hérité correspondant. On a alors la proposition suivante :

Proposition 4.3. *Étant donnés une contrainte n-aire c, un pavé $\boldsymbol{B}$ et une variable x_k n'ayant qu'une occurrence dans c, on a les inclusions :*

$$\mathsf{Hull}_\square(\rho_c \cap \boldsymbol{B}) \subseteq \mathsf{HC4revise}(c, \boldsymbol{B}, \mathbb{U}_\square) \subseteq \mathsf{HC4revise}^\star(c, \boldsymbol{B}, \mathbb{U}_\square)$$

Démonstration. L'inclusion $\mathsf{Hull}_\square(\rho_c \cap \boldsymbol{B}) \subseteq \mathsf{HC4revise}(c, \boldsymbol{B}, \mathbb{U}_\square)$ se déduit du fait que l'algorithme HC4revise est un opérateur de contraction (donc complet). L'autre inclusion est trivialement vraie puisque HC4revise correspond à l'algorithme HC4revise* où l'on effectue des intersections de domaines en plus. ■

4.2 Une nouvelle définition pour la box-consistance

La définition original de la box-consistance [29] (*cf.* déf. 3.10, p. 44) est basée sur le seul domaine d'approximation $\mathbb{I}_\square$ et sur l'extension naturelle aux intervalles des contraintes considérées. Nous proposons ci-dessous une généralisation de cette définition autorisant le recours à différentes extensions pour chacune des projections d'une contrainte, ainsi que l'utilisation de deux domaines d'approximation différents. Cette extension de la définition nous permet, entre autres, de capturer les différentes variations de définition de la box-consistance existant dans la littérature, telle que celle donnée par COLLAVIZZA *et al.* [51] où les quasi-zéros sont encadrés par des intervalles ouverts (domaine d'approximation $\mathbb{I}_\circ$) tandis que le domaine final est toujours représenté par un intervalle fermé (domaine d'approximation $\mathbb{I}_\square$).

Définition 4.1 (Box$_{\mathcal{A}_1,\mathcal{A}_2,\Gamma}$- consistance [23]). Étant donnés deux domaines d'approximation $\mathcal{A}_1$ et $\mathcal{A}_2$, soit c une contrainte réelle n-aire, $\Gamma = \{C_1, \ldots, C_n\}$ un ensemble de n extensions aux intervalles de c, $k \in \{1, \ldots, n\}$ un entier et $\boldsymbol{D} = D_1 \times \cdots \times D_n$ un produit cartésien de domaines. La contrainte c est dite *box$_{\mathcal{A}_1,\mathcal{A}_2,\Gamma}$-consistante* par rapport au pavé $\boldsymbol{D}$, à une variable x_k de c et à C_k si et seulement si :

$$D_k = \mathrm{apx}_{\mathcal{A}_1}(D_k \cap \{\mathsf{r}_k \in \mathbb{R} \mid C_k(D_1, \ldots, D_{k-1}, \mathrm{apx}_{\mathcal{A}_2}(\{\mathsf{r}_k\}), D_{k+1}, \ldots, D_n)\}) \tag{4.1}$$

On dira que la contrainte c est box-consistante par rapport à Γ et $\boldsymbol{D}$ si et seulement si la relation (4.1) est vraie pour tout entier k dans l'ensemble $\{1, \ldots, n\}$. De même, un ensemble de contraintes $\mathcal{C}$ est dit box-consistant par rapport à un pavé de domaines $\boldsymbol{D}$ si chaque contrainte c de $\mathcal{C}$ est box-consistante par rapport à $\boldsymbol{D}$ et à un ensemble Γ de n extensions aux intervalles de c.

On pourra alors noter que la box$_{\mathbb{I}_\square,\mathbb{I}_\circ,\{C,\ldots,C\}}$-consistance — où C est l'extension naturelle aux intervalles de la contrainte considérée — correspond à la définition de la box-consistance énoncée par COLLAVIZZA *et al.* [51], alors que la box$_{\mathbb{I}_\square,\mathbb{I}_\square,\{C,\ldots,C\}}$-consistance correspond à la définition originale [29]. Afin de simplifier l'exposé, on notera *box$_o$-consistance* la box$_{\mathbb{I}_\square,\mathbb{I}_\square,\{C,\ldots,C\}}$-consistance.

Dans la suite, on va utiliser une extension aux intervalles particulière afin de tirer partie de la définition étendue de la box-consistance : étant données une contrainte c: $f_1(x_1, \ldots, x_n) \diamond 0$ ($\diamond \in \{=, \leqslant, \geqslant\}$) et la variable x_j ($1 \leqslant j \leqslant n$) ayant k occurrences dans c ($k \geqslant 1$), soit $c[x_j^{(i)}]$ ($1 \leqslant i \leqslant k$) l'expression de la contrainte c obtenue en remplaçant toutes les occurrences de x_j sauf la i-ième par des nouvelles variables de même domaine que x_j et en exprimant la relation résultante en terme de la i-ième occurrence de x_j ($c[x_j^{(i)}]$: $x_j^{(i)} \diamond' f_2(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n, y_1, \ldots, y_{k-1})$). Enfin, notons $C_{[x_j^{(i)}]}$ l'extension naturelle aux intervalles de la contrainte $c[x_j^{(i)}]$.

Comme déjà évoqué plus haut, l'algorithme de calcul de la box$_o$-consistance (*cf.* alg. 3.5, p. 45) par une recherche de quasi-zéros extrêmes permet de gérer efficacement la réduction des domaines de variables ayant une grande multiplicité dans une contrainte c en évitant de considérer séparément chacune des occurrences. Par contre, il s'avère relativement inintéressant, car coûteux, pour réduire le domaine de variables n'ayant qu'une occurrence dans c. Or, c'est justement le type de variables que l'algorithme HC4revise peut gérer très efficacement.

La proposition ci-dessous permet d'établir un lien entre l'algorithme HC4revise et la box-consistance que nous exploiterons dans la section suivante pour définir un nouvel algorithme de calcul de la box-consistance.

Proposition 4.4. *Soit c une contrainte n-aire, C son extension naturelle aux intervalles, x_k une variable n'ayant qu'une occurrence dans c, $\Gamma = \{C, \ldots, C_{[x_k^{(1)}]}, \ldots, C\}$ un ensemble de n extensions aux intervalles de c et $\boldsymbol{B}$ un pavé d'intervalles n-aire. De plus, soit $\boldsymbol{B'}$ le pavé obtenu par l'application de* HC4revise$^\star(c, \boldsymbol{B}, \mathbb{U}_\Box)$. *On a alors : la contrainte c est box$_{\mathbb{I}_\Box, \mathbb{I}_\Box, \Gamma}$-consistante par rapport à $\boldsymbol{B'}$, x_k et $C_{[x_k^{(1)}]}$.*

Démonstration. On fait la preuve par induction sur la représentation arborescente de c que, étant donné $\boldsymbol{B'}$ le résultat de HC4revise$^\star(c, \boldsymbol{B}, \mathbb{U}_\Box)$, l'intervalle I'_k est égal à Hull$_\Box({\rho_c}^{(k)}(\boldsymbol{B}) \cap I_k)$. De la définition 4.1 s'ensuit le résultat. ■

4.3 Un nouvel algorithme pour la box-consistance

Les considérations déjà plusieurs fois évoquées dans ce qui précède en ce qui concerne les efficacités relatives de HC4revise et BC3revise nous ont conduit à définir l'algorithme BC4 (*cf.* Alg. 4.5) de calcul de la box-consistance les utilisant tous les deux. Pour cela, on doit considérer les contraintes du système à résoudre au niveau de leurs projections sur les variables et non plus seulement globalement. Ainsi, il devient possible d'appliquer HC4revise pour réduire spécifiquement les domaines des variables n'ayant qu'une occurrence dans une contrainte et d'employer BC3revise pour les variables possédant plusieurs occurrences.

ALG. 4.5: Algorithme BC4

BC4(**entrée** $\mathcal{C} = \{c_1, \ldots, c_m\}$: ensemble de contraintes ; **entrée/sortie** $\boldsymbol{B}$: pavé d'intervalles)
début
  répéter
    $\boldsymbol{B'} \leftarrow \boldsymbol{B}$
    faire
      PasFini ← **faux**
      pour chaque $c \in \mathcal{C}$ **faire** *% Calcul du pt. fixe de HC4revise pour chaque variable*
                % ayant une seule occurrence dans c
        $\mathcal{V}_c^1 \leftarrow \{x \in \mathsf{Var}(c) \mid \mu_c(x) = 1\}$ *% $\mathcal{V}_c^1$: ens. des vars. avec une seule occurrence dans c*
        $\boldsymbol{B''} \leftarrow \boldsymbol{B}$
        HC4revise(c,$\boldsymbol{B}$,$\mathbb{U}_\Box$)
        PasFini ← $(\boldsymbol{B} \neq \varnothing \wedge ((\exists I_k \neq I''_k \wedge x_k \in \mathcal{V}_c^1) \vee$ *PasFini*$))$
      fpc
    tant que (*PasFini*)
    si $(\boldsymbol{B} \neq \varnothing)$ **alors**
      $\boldsymbol{B} \leftarrow$ BC3$(\mathcal{C}_p^+, \boldsymbol{B})$ *% L'ensemble $\mathcal{C}_p^+$ est l'ensemble $\{(c_1, \mathcal{V}_{c_1}^+), \ldots, (c_m, \mathcal{V}_{c_m}^+)\}$ où $\mathcal{V}_c^+$ est*
        % l'ensemble des variables dont la multiplicité dans c est supérieure à 1.
    finsi
  jusqu'à $(\boldsymbol{B} = \varnothing$ **ou** $\boldsymbol{B'} = \boldsymbol{B})$
fin

Comme plus haut, on appellera BC4* l'algorithme BC4 où l'appel à HC4revise est remplacé par un appel à HC4revise*.

On a alors la proposition suivante :

Proposition 4.5. *Soit un ensemble de contraintes $\mathcal{C}$ et soient $\mathcal{C}_p^1$ et $\mathcal{C}_p^+$ deux ensembles de contraintes définis à partir de $\mathcal{C}$ par $\mathcal{C}_p^1 = \{C_{[x^{(1)}]} \mid \exists c \in \mathcal{C} : x \in \mathsf{Var}(c) \wedge \mu_c(x) = 1\}$ et $\mathcal{C}_p^+ = \{C_x \mid \exists c \in \mathcal{C} : x \in \mathsf{Var}(c) \wedge \mu_c(x) > 1\}$. Soit $\boldsymbol{B}$ et $\boldsymbol{B'}$ deux pavés d'intervalles, avec $\boldsymbol{B'} =$* BC4$^\star(\mathcal{C}, \boldsymbol{B})$. *On a : l'algorithme* BC4* *termine, est contractant et confluent. De plus, l'ensemble $\mathcal{C}$ est box$_{\mathbb{I}_\Box, \mathbb{I}_\Box, \mathcal{C}_p^1 \cup \mathcal{C}_p^+}$-consistant par rapport à $\boldsymbol{B'}$.*

Démonstration. La confluence, la correction et la terminaison se prouvent de la même façon que pour l'algorithme HC4. En ce qui concerne la box-consistance, il suffit de remarquer que HC4revise* calcule la box$_{\mathbb{I}_\Box, \mathbb{I}_\Box, \mathcal{C}_p^1}$-

consistance pour les variables ayant plus d'une occurrence. En conséquence, le calcul du point-fixe commun conduit au résultat. ■

En utilisant la proposition 4.3, on peut alors prouver que l'algorithme BC4 est complet et inclus dans celui de BC4*.

4.4 Résultats expérimentaux

Les algorithmes HC3, BC3 et BC4 ont été implémentés dans Cosinus, le logiciel de résolution de contraintes de Laurent GRANVILLIERS.

Le tableau 4.1 présente les résultats obtenus sur une SUN UltraSparc 2/167 MHz pour les algorithmes BC3 et BC4, et sur un AMD K6/166 MHz pour HC3. Les résultats de l'AMD ont ensuite été mis à l'échelle de l'UltraSparc. Le seuil de découpage était fixé à $1.0 \cdot 10^{-8}$ pour tous les domaines et nous n'avons utilisé ni facteur d'amélioration, ni affaiblissement de la box-consistance (*cf.* la définition de la box$_\varphi$-consistance, p. 47). Les temps de calcul dépassant une heure ont été remplacés par des points d'interrogation (?).

TAB. 4.1: Résultats expérimentaux

Benchmark	BC4 *(s.)*	BC3 *(s.)*	BC3/BC4	HC3 *(s.)*†	HC3/BC4
Cosnard 10	0,15	2,45	**15**	4,10	**27**
Cosnard 20	0,95	16,50	**17**	172,75	**182**
Cosnard 40	2,90	123,30	**42**	?	↗
Cosnard 80	13,70	916,95	**67**	?	↗
Broyden 10	2,40	2,35	**1**	?	↗
Broyden 160	86,65	86,55	**1**	?	↗
Kearfott 10	0,50	1,40	**3**	0,75	**1,5**
Kearfott 11	0,70	6,45	**9**	0,70	**1**
i4	32,65	160,80	**5**	?	↗
bifurcation	2,85	31,10	**11**	3,55	**1,2**
DC circuit	0,40	12,15	**30**	0,65	**1,6**
pentagon	1,10	12,95	**12**	0,70	**0,6**
pentagon all	44,60	2 340,00	**52**	124,20	**2,8**

† *Temps sur un AMD K6/166 MHz mis à l'échelle d'une SUN UltraSparc 2/167 MHz.*

Les problèmes *Cosnard* x et *broyden-banded* x correspondent au problème de MORÉ et COSNARD pour x variables et x contraintes et au problème de Broyden-banded avec x variables (*cf.* p. 103). *Kearfott 10*, *Kearfott 11* et *bifurcation* sont des problèmes où le nombre d'occurrences simples et d'occurrences multiples de variables dans les contraintes est à peu près égal ; les variables apparaissant dans i_4, *DC circuit*, *pentagon* et *pentagon all* ont des occurrences simples.

Le tableau 4.1 nous permet de retrouver le comportement bien connu des algorithmes HC3 et BC3 : HC3 est incapable de calculer les solutions de *Broyden-banded* car la décomposition des contraintes amplifie le problème de dépendance ; HC3 est lent pour le problème de MORÉ-COSNARD du fait du grand nombre de contraintes primitives générées et des variables ajoutées. Par contre, BC3 se comporte très bien sur ces problèmes. L'algorithme HC3 se montre plus efficace sur les problèmes où les contraintes ont des occurrences simples de variables.

On constate que l'utilisation de HC4revise avant d'appliquer BC3 dans l'algorithme BC4 accélère énormément les calculs. L'algorithme BC4 se révèle plus efficace que HC3 sur la plupart des benchmarks, y compris ceux qui étaient mal gérés par BC3.

TROISIÈME PARTIE

Approximations intérieures et variables quantifiées

1. **Introduction**
2. **Quantificateurs et approximations intérieures**
3. **Contraintes d’intervalles et variables quantifiées**

Introduction

Interprétations d'un pavé solution — correction vs. *complétude des pavés solutions — variables universellement quantifiées — exemples d'applications*

LES TECHNIQUES de résolution présentées dans la partie II s'appliquent implicitement à des contraintes dont les variables ne sont pas quantifiées. Ainsi, la propriété vérifiée, dans le meilleur des cas, par chacun des pavés retournés par les solveurs basés sur ces techniques est celle de contenir — au mieux — au moins un élément solution du système de contraintes considéré.

Certaines applications attendent d'autres propriétés de la part de ces pavés solutions. SHARY [213] considère ainsi deux interprétations possibles pour un pavé :

1. il contient au moins un élément solution ;
2. tous les éléments qu'il contient sont solution.

Quant à WARD *et al.* [246], ils distinguent quatre interprétations possibles pour un pavé et un système de contraintes $\mathcal{S}$ (*cf.* fig. 4.3) :

only. Toutes les solutions de $\mathcal{S}$ sont contenues dans le pavé ;

every. Chaque élément du pavé est une solution de $\mathcal{S}$;

some. Il existe au moins une solution de $\mathcal{S}$ dans le pavé ;

none. Le pavé ne contient aucune solution de $\mathcal{S}$.

L'interprétation considérée dans la partie II correspond donc à la notion **some** de WARD *et al.* avec cependant la réserve que l'on n'a pas toujours l'assurance que la boîte considérée contienne effectivement une solution du système. Les résolveurs de contraintes présentés précédemment possèdent ainsi la propriété de *complétude* (on ne perd jamais de solution car l'union des pavés calculés contient l'ensemble des éléments de la relation considérée inclus dans le pavé de départ), mais pas celle de *correction* (les intervalles retournés contiennent des éléments qui ne sont pas solution du système de contraintes).

Or, la correction peut être cruciale pour certaines applications. Considérons, par exemple, le problème de la détermination de certains paramètres dans le cadre de l'élaboration d'une surface portante pour un plancher [202]. Si tous les points des pavés calculés par le solveur ne sont pas solution du système de contraintes modélisant le problème, comment l'ingénieur chargé de la conception peut-il décider des valeurs à affecter à chacun des paramètres ? Choisir aléatoirement un point d'un pavé peut en effet conduire à la construction d'une surface portante ne respectant pas certaines contraintes physiques. Calculer des pavés suffisamment petits pour que l'on puisse les assimiler à des points eu égard au degré de précision dans la réalisation — que représente un intervalle de taille 10^{-16} lorsque l'on travaille à l'échelle du centimètre ? — ne résoud pas le problème. En effet, on n'a pas toujours l'assurance que chaque pavé contient effectivement une solution.

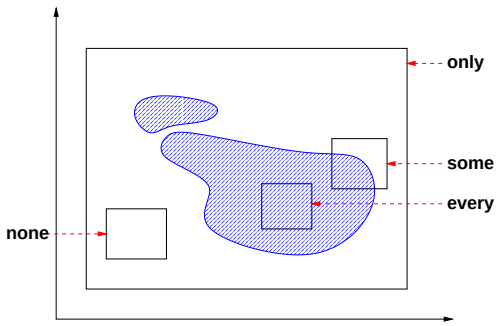


FIG. 4.3: Classification de WARD *et al.*

Par ailleurs, certains problèmes en Robotique (*sensor planning* [3], *pathplanning* [160]), en *control design* [11], ou en positionnement de caméra [71] font intervenir des contraintes où certaines variables sont universellement quantifiées. C'est par exemple le cas pour les deux problèmes ci-dessous correspondant respectivement à la détermination des conditions de stabilité d'un système de régulation par rétroaction (*control design*) et à la recherche d'un ensemble de trajectoires satisfaisant certaines propriétés (*collision problem* en robotique).

Exemple 4.13 (Un système contrôlé par rétroaction [1]). *Considérons un système de régulation de la température d'une pièce (*cf. *fig. 4.4) où l'installation est modélisée par une fonction de transfert* $G(s,p) = p_1/(1-s/p2)$*, avec* $0{,}8 \leqslant p_1, p_2 \leqslant 1{,}25$*, et le compensateur par une fonction de rétroaction* $C(s,q) = q_1$*. Les conditions de stabilité pour la sortie sont déterminées par le système suivant :*

$$\forall p_1 \in [0{,}8 \,..\, 1{,}25], \forall p_2 \in [0{,}8 \,..\, 1{,}25], \forall w_1 \in [0 \,..\, 10]: \quad \begin{cases} p_2(1+p_1q_1) < 0, \\ 99w_1^2 + p_2^2(100(1+p_1q_1)^2 - 1) > 0, \\ (400 - q_1^2)w_2^2 + p_2^2(400(1+p_1q_1)^2 - q_1^2) > 0 \end{cases} \tag{4.2}$$

On recherche donc les domaines des variables q_1*,* q_2 *et* w_2 *tels que les trois contraintes ci-dessus sont satisfaites pour* toutes *les valeurs de* p_1 *et* p_2 *comprises entre 0,8 et 1,25.*

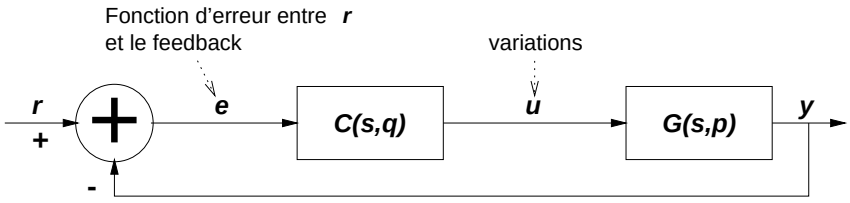


FIG. 4.4: Un système contrôlé par rétroaction

Exemple 4.14 (Problème de collision). *Un point* A *se déplace sur une trajectoire circulaire de rayon* r*. On souhaite connaître l'ensemble des points* B *du plan se trouvant à tout moment à une distance au moins égale à* e *du point* A (cf. *fig. 4.5). Ce problème se modélise par la contrainte :*

$$\forall \theta: \sqrt{(r\cos\theta - x_B)^2 + (r\sin\theta - y_B)^2} \geqslant e$$

L'ensemble des solutions correspond à tous les points se trouvant à l'extérieur de la bande torique colorée apparaissant dans la figure 4.5. On retrouvera une instance de ce problème étendue au cas de deux points mobiles A_1 *et* A_2 *dans le chapitre 12.*

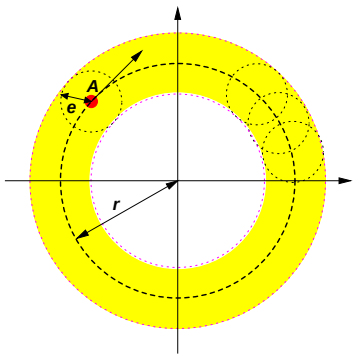


FIG. 4.5: Un problème de collision

Dans le cas du problème de positionnement de caméra, évoqué au chapitre 6, les systèmes à résoudre contiennent des *contraintes temporelles*, c'est-à-dire des contraintes où le temps intervient sous la forme d'une variable universellement quantifiée.

Les algorithmes de résolution de contraintes décrits dans la partie II ne peuvent être utilisés pour résoudre de tels problèmes car ils chercheraient à découper les domaines des variables quantifiées et ne peuvent garantir la correction.

La méthode la plus utilisée à notre connaissance pour résoudre des systèmes de contraintes possédant des occurrences de variables universellement quantifiées consiste à déterminer un système de contraintes équivalent sans quantificateurs (*élimination de quantificateurs*) grâce la *décomposition algébrique cylindrique* (CAD), méthode inventée par COLLINS [53] et perfectionnée par HONG [110]. La méthode CAD ne s'applique que sur des systèmes de contraintes polynômiales à coefficients rationnels. Elle a cependant été étendue en 1999 par PAU et SCHICHO [181] au cas de contraintes

faisant intervenir des fonctions trigonométriques (*décomposition trigonométrique cylindrique*) de forme particulière : les *polynômes trigonométriques*, qui sont des polynômes où peuvent apparaître les fonctions sinus et cosinus appliquées à des variables particulières distinctes des variables algébriques.

Pour le système (4.2), on obtiendrait ainsi le système équivalent sans quantificateur [2] :

$$p_a = q_1 + 20 \wedge p_b = q_1 + 2 \wedge p_c = 8q_1 + 11 \wedge (p_a \geqslant 0 \wedge p_b \leqslant 0) \vee (p_c < 0 \wedge p_b \geqslant 0)$$

Comme le note SHARY [212], la résolution d'un système de contraintes avec quantificateurs universels peut se ramener au calcul d'un sous-ensemble de la relation réelle sous-jacente (*approximation intérieure*). Ce dernier problème a été étudié indépendamment par de nombreuses personnes, mais nous pouvons dégager deux méthodes différentes : l'utilisation de l'arithmétique de KAUCHER [129] (travaux de SHARY [211], MARKOV [156] et POPOVA) — ou de l'*arithmétique d'intervalles modaux* (travaux de GARDEÑES et MIELGO [78], Armengol *et al.* [14]...) et un processus général d'évaluation/découpage dont nous reparlerons dans la suite.

Le reste de cette partie s'organise comme suit : nous présentons rapidement dans le chapitre 5 certaines des méthodes utilisées pour résoudre des contraintes avec quantificateurs et pour déterminer des approximations intérieures de relations réelles ; nous introduirons alors au chapitre 6 les algorithmes que nous avons définis pour gérer des systèmes de contraintes temporelles en décrivant au passage les travaux précédents initiés par Éric LANGUÉNOU et Franck JARDILLIER [120] dans le cadre du *problème du cameraman virtuel* (Un prototype de modeleur déclaratif de mouvements de caméra, WOpAC, a été implémenté par Marc CHRISTIE durant son DEA [46] au-dessus de la librairie C++ de résolution de contraintes OpAC développée par nos soins. Nous présenterons rapidement cette librairie au chapitre 12 et nous montrerons les résultats obtenus par WOpAC sur un ensemble de jeux d'essais).

Contributions

- Définition d'algorithmes de calcul d'approximations intérieures de relation réelles ;
- définition d'algorithmes de résolution de systèmes de contraintes réelles non-linéaires avec variables universellement quantifiées.

CHAPITRE 5

Gestion de quantificateurs et calcul d'approximations intérieures

Élimination de quantificateurs et décomposition algébrique cylindrique — décomposition trigonométrique cylindrique — quantificateurs et approximations intérieures — calcul d'approximations intérieures par évaluation/découpage — bases de BERNSTEIN *— arithmétique de* KAUCHER, MARKOV *et arithmétique d'intervalles modaux*

NOUS PRÉSENTERONS DANS CE CHAPITRE un état de l'art succinct des différentes méthodes permettant la résolution de contraintes avec quantificateurs. Nous nous pencherons dans un premier temps sur des méthodes générales, capables de gérer des contraintes avec variables tant universellement qu'existentiellement quantifiées. Les deux méthodes abordées (décomposition algébrique cylindrique et décomposition trigonométrique cylindrique) effectuent une élimination des quantificateurs des formules données en entrée et calculent des formules équivalentes non quantifiées. Nous nous contenterons ici d'une présentation informelle des principes de la *décomposition algébrique cylindrique* de COLLINS [53], renvoyant le lecteur intéressé à l'excellente introduction de JIRSTRAND [121], ainsi qu'aux notes de RODA [199] sur le cours de COLLINS pour une description plus détaillée. Comme son nom l'indique, la décomposition algébrique cylindrique ne s'intéresse qu'à des *ensembles semi-algébriques, i.e.* des ensembles définis par des polynômes. Nous évoquerons une variation de la décomposition algébrique cylindrique due à PAU et SCHICHO [181] s'appliquant à des *polynômes trigonométriques*, c'est à dire des expressions définies à partir de polynômes et des fonctions sinus et cosinus.

Dans le cadre de l'application considérée durant notre thèse, nous nous intéressons plus particulièrement aux contraintes avec quantificateurs universels. Nous verrons que la gestion de formules universellement quantifiées peut se ramener en pratique au calcul d'une *approximation intérieure* de relations réelles. Nous présenterons donc formellement cette notion en évoquant les différentes définitions possibles rencontrées dans la littérature, puis nous décrirons quatre méthodes de calcul d'approximation intérieure, en commençant par la méthode de GARLOFF et GRAF [82] basée sur l'expansion de polynômes en polynômes de BERNSTEIN, qui permet de calculer une approximation intérieure dans le cas d'une conjonction d'inégalités polynômiales. Cette méthode n'est qu'une des nombreuses instances d'une procédure plus générale de calcul d'approximation intérieure par évaluation/découpage. Nous évoquerons deux autres techniques du même ordre : celle de KUTSIA et SCHICHO [141], faisant appel à un critère de non-existence de racines d'un polynôme dans un pavé pour calculer une approximation intérieure pour des disjonctions/conjonctions d'inégalités polynômiales strictes ; et la méthode de JARDILLIER et LANGUÉNOU [120] basée sur l'évaluation en arithmétique d'intervalles de contraintes d'inégalités (polynômiales ou non). Cette dernière méthode a été mise au point dans le cadre d'un problème de positionnement de caméra (*virtual cameraman problem*) exprimé par un système de contraintes temporelles. Nous verrons dans le chapitre 6 les algorithmes développés durant notre thèse pour résoudre plus efficacement ce type de problème.

Une autre méthode pour calculer des approximations intérieures fait appel à l'arithmétique de KAUCHER [129] ou à l'arithmétique d'intervalles modaux [78]. Ces arithmétiques seront présentées dans la section 5.2.2 en insistant sur leurs applications dans le cadre du *control design*.

Dans la suite, les notions mathématiques marquées d'une croix de Malte (✠) sont rappelées dans l'annexe B, page 225.

5.1 Élimination de quantificateurs

Contrairement aux méthodes que nous décrirons dans la section 5.2, celles présentées ci-dessous sont capables de gérer des conjonctions et/ou disjonctions de contraintes (trigonométriques) polynômiales avec quantificateurs universels et/ou existentiels. Partant d'une formule quantifiée, elles permettent de dériver symboliquement une formule équivalente non quantifiée. La première méthode (décomposition algébrique cylindrique) s'applique uniquement à des polynômes à coefficients rationnels ; la deuxième méthode (décomposition trigonométrique cylindrique) est capable de gérer aussi les polynômes trigonométriques (nous donnerons une définition précise de cette forme dans la section 5.1.2).

5.1.1 Décomposition algébrique cylindrique

L'algorithme de *décomposition algébrique cylindrique* (CAD) a été mis au point par COLLINS [53] en 1973 pour résoudre le problème de l'*élimination de quantificateurs* pour le cas des *corps réels clos* : étant donnée une formule F d'une théorie du premier ordre sur un corps clos T, on recherche une formule F' de T sans quantificateur, équivalente à F et possèdant les mêmes variables libres.

Étant donné un ensemble de polynômes $\mathcal{P} \subset \mathbb{Q}[x_1, \ldots, x_n]$, l'idée de base de la méthode CAD est de décomposer $\mathbb{R}^n$ en un ensemble de régions $\Gamma = \{\gamma_1, \ldots, \gamma_p\}$ sur lesquelles chaque polynôme $p_i \in \mathcal{P}$ a un signe constant. On dira qu'un polynôme $p \in \mathcal{F}$ est *invariant* sur une région γ si l'une des trois relations ci-dessous est vérifiée :

$$\forall \boldsymbol{x} \in \gamma \colon p(\boldsymbol{x}) > 0$$
$$\forall \boldsymbol{x} \in \gamma \colon p(\boldsymbol{x}) = 0$$
$$\forall \boldsymbol{x} \in \gamma \colon p(\boldsymbol{x}) < 0$$

Plus généralement, on dira que la région γ est $\mathcal{P}$-invariante si chaque polynôme $p \in \mathcal{P}$ est invariant sur γ. Comme on va le voir dans l'exemple 5.16, on peut déterminer à partir d'une décomposition $\mathcal{P}$-invariante de $\mathbb{R}^n$ la valeur de vérité de formules closes basées sur les polynômes de $\mathcal{P}$, voire isoler les régions solution d'un système de contraintes.

Soit une formule du premier ordre F (disjonctions/conjonctions de contraintes polynômiales avec quantificateurs) que l'on souhaite traiter par l'algorithme CAD. On commence par mettre F en *forme prénexe* (tous les quantificateurs en tête), puis l'on extrait l'ensemble $\mathcal{P} \subset \mathbb{Q}[x_1, \ldots, x_n]$ de polynômes apparaissant dans F. La méthode CAD travaille alors en trois étapes :

1. *Étape de projection.* À partir de $\mathcal{P}_0 = \mathcal{P}$, on calcule une suite d'ensembles de polynômes $\mathcal{P}_1 \subset \mathbb{Q}[x_1, \ldots, x_{n-1}]$, $\mathcal{P}_2 \subset \mathbb{Q}[x_1, \ldots, x_{n-2}]$, …, $\mathcal{P}_{n-1} \subset \mathbb{Q}[x_1]$, où l'ensemble solution de $\mathcal{P}_i$ correspond à la projection sur $\mathbb{R}^{n-i}$ des « points remarquables » (tangentes, points isolés…) de l'ensemble solution de $\mathcal{P}_{i-1}$. On détermine ainsi à l'étape i une caractérisation par l'ensemble des polynômes $\mathcal{P}_i$ des connexes de $\mathbb{R}^{n-i}$ étant $\mathcal{P}_{i-1}$-*délinéables*[✠] (*cf.* exemple 5.15) ;
2. *Étape de base.* On détermine l'ensemble des racines $\alpha_1, \ldots, \alpha_s$ de $\mathcal{P}_{n-1} \subset \mathbb{Q}[x_1]$, ce qui permet d'isoler $2s+1$ intervalles : $(-\infty \mathinner{..} \alpha_1), [\alpha_1 \mathinner{..} \alpha_1], (\alpha_1 \mathinner{..} \alpha_2), \ldots, \ldots, [\alpha_s \mathinner{..} \alpha_s], (\alpha_s \mathinner{..} +\infty)$. De ces $2s+1$ intervalles, constituant par construction une décomposition $\mathcal{P}_{n-1}$-invariante de $\mathbb{R}$, on extrait $2s+1$ points, les *échantillons* (par exemple, chacun des α_i ainsi que le centre des intervalles $(\alpha_i \mathinner{..} \alpha_{i+1})$), qui seront utilisés dans la dernière étape ;
3. *Étape d'extension.* Dans cette étape, on cherche à étendre une décomposition $\mathcal{P}_i$-invariante de $\mathbb{R}^{n-i}$ en une décomposition $\mathcal{P}_{i-1}$-invariante de $\mathbb{R}^{n-i+1}$. Partant de $\mathbb{R}$, on sait, par construction, que $\mathcal{P}_{n-2}$ est délinéable sur chacun des $2s+1$ intervalles I_i de l'étape de base. On peut alors évaluer pour chaque intervalle I_i chaque polynôme $p \in \mathbb{Q}[x_1, x_2]$ de $\mathcal{P}_{n-2}$, obtenant ainsi des polynômes en la seule variable x_2 (la variable x_1 étant remplacée par le point échantillon de I_i), sur lesquels on réapplique l'étape de base pour déterminer de nouveaux échantillons qui seront utilisés à l'étape suivante (extension de la décomposition de $\mathbb{R}^{n-i+1}$ à une décomposition de $\mathbb{R}^{n-i+2}$).

Au final, on obtient une *décomposition* à la fois *algébrique*✠ et *cylindrique*✠ de $\mathbb{R}^n$, où chaque région de $\mathbb{R}^n$ est $\mathcal{P}$-invariante.

Exemple 5.15 (Étape de projection [121]). *Soit le polynôme* $p = (x_1-2)^2+(x_2-2)^2+(x_3-2)^2-1$ *représentant une sphère de rayon* 1 *centrée en* $(2,2,2)$ *(*cf. *fig. 5.1). La projection des racines de* p *sur* $\mathbb{R}^2$ *correspond au cercle de rayon* 1 *centré en* $(2,2)$ *représenté par le polynôme* $p' = (x_1-2)^2+(x_2-2)^2-1$ *et la projection des racines de* p' *sur* $\mathbb{R}$ *est constituée des deux points* $\alpha_1 = 1$ *et* $\alpha_1 = 3$.

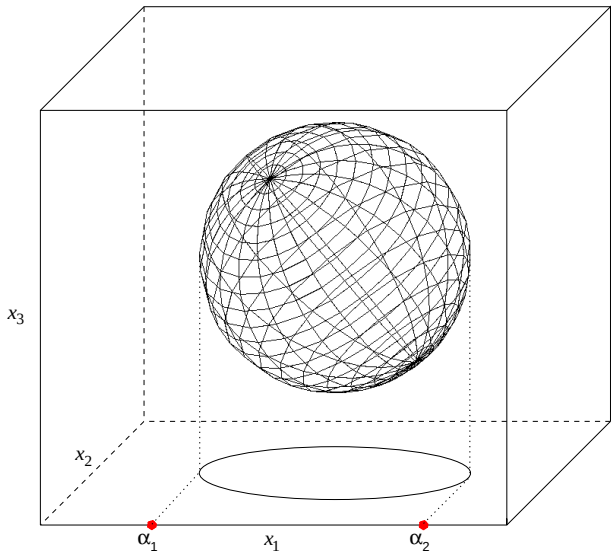


FIG. 5.1: Étape de projection pour $p = (x_1-2)^2+(x_2-2)^2+(x_3-2)^2-1$

Exemple 5.16 (Résolution par CAD de contraintes avec quantificateurs). *Soit à résoudre le système de contraintes :*

$$\phi\colon \forall y\colon (x^2+y^2-1>0 \wedge x^3-y^2<0) \tag{5.1}$$

La figure 5.2 représente les courbes correspondant aux polynômes x^2+y^2-1 *et* x^3-y^2.

Étape de projection. *La formule* (5.2) *est déjà en forme prénexe. On commence par en extraire les polynômes pour former l'ensemble* $\mathcal{P}_0$. *Il vient :*

$$\mathcal{P}_0 = \{x^2+y^2-1, x^3-y^2\}$$

L'ensemble $\mathcal{A}_0$ *des points remarquables de* $\mathcal{P}_0$ *est constitué des points de* $\mathbb{R}$ *:* $-1, 0, \alpha, 1$, *avec* $\alpha \approx 0{,}754\ 8$ *la solution réelle de l'équation* $x^3+x^2-1=0$. *L'ensemble des zéros des polynômes de* $\mathcal{P}_1$ *doit être égal à* $\mathcal{A}_0$. *Il vient** *:*

$$\mathcal{P}_1 = \{x^4+(2-\alpha)x^3+(1-2\alpha)x^2-\alpha x\}$$

Étape de base. *On cherche les zéros du polynôme de* $\mathcal{P}_1$, *à partir desquels on détermine une partition de* $\mathbb{R}$. *Il vient :*

$$\begin{array}{llllll} I_1 = & (-\infty\,..\,-1) & I_2 = & [-1\,..\,-1] & I_3 = & (-1\,..\,0) \\ I_4 = & [0\,..\,0] & I_5 = & (0\,..\,\alpha) & I_6 = & [\alpha\,..\,\alpha] \\ I_7 = & (\alpha\,..\,1) & I_8 = & [1\,..\,1] & I_9 = & (1\,..\,+\infty) \end{array}$$

*Pour des raisons de clarté, l'ensemble $\mathcal{P}_1$ donné ici ne contient que les polynômes prenant part aux calculs ultérieurs.

À chaque intervalle de la partition, on peut associer un échantillon ainsi qu'une représentation par une formule non quantifiée. Pour l'intervalle I_3, par exemple, on peut choisir le point $p_{I_3} = -1/2$ et l'on a la représentation $\phi_{I_3} = x > -1 \wedge x < 0$. Pour la décomposition ci-dessus, nous pouvons choisir les points-échantillons suivants :

$$\psi_1 = \{-2, -1, -\frac{1}{2}, 0, \frac{1}{3}, \alpha, \frac{4}{5}, 1, 2\}$$

Étape d'extension. *On détermine l'ensemble des polynômes de $\mathbb{Q}[y]$ obtenus en instanciant la variable x dans les polynômes de $\mathcal{P}_0$ avec chacune des valeurs de ψ_1. Il vient :*

$$\mathcal{P}'_0 = \{y^2+3, y^2, y^2-\frac{3}{4}, y^2-1, y^2-\frac{8}{9}, y^2+\alpha^2-1, y^2-\frac{9}{25}, -8-y^2, -1-y^2, -\frac{1}{8}-y^2, \\ -y^2, \frac{1}{27}-y^2, \alpha^3-y^2, \frac{64}{125}-y^2, 1-y^2, 8-y^2\}$$

Comme dans l'étape de base, on détermine les zéros de chaque polynôme de $\mathcal{P}'_0$; on en déduit une partition de l'axe des ordonnées, ainsi qu'une partition de $\mathbb{R}^2$ et l'on détermine des points-échantillons qui nous permettent de connaître le signe des polynôme de $\mathcal{P}_0$ sur chaque connexe de la partition de $\mathbb{R}^2$. À partir de ces informations, il est aisé de déterminer une représentation par une formule non quantifiée de chaque connexe. Par exemple, le connexe γ de la figure 5.2 a pour représentation :

$$\phi_\gamma = x > -1 \wedge x < 0 \wedge x^2 + y^2 - 1 < 0$$

Grâce à la partition de $\mathbb{R}^2$ et à la représentation de chaque connexe par une formule non quantifiée, on peut remplacer la formule quantifiée (5.1) *par la formule non quantifiée équivalente :*

$$\phi' : x < -1$$

Étant donnée une formule F de la forme :

$$\Xi_1 x_1 \ldots \Xi_k x_k : \phi(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n), \quad \text{avec} \quad \Xi_i \in \{\forall, \exists\}, i \in \{1, \ldots, k\}$$

on peut noter que la méthode CAD ne décompose pas $\mathbb{R}^k$, mais $\mathbb{R}^n$. Ainsi, c'est le nombre total de variables et non pas seulement le nombre de variables quantifiées qui détermine la complexité de la méthode.

5.1.2 Décomposition trigonométrique cylindrique

La méthode CAD n'est applicable qu'à des ensembles de polynômes à coefficients rationnels. Cependant, nombre de problèmes s'expriment à l'aide de contraintes faisant intervenir des fonctions trigonométriques (à commencer par l'application de contrôle de caméra que nous considérerons au chapitre suivant). Une approche naïve consiste à algébriser le problème en remplaçant par exemple $\sin(\theta)$ par x, $\cos(\theta)$ par y et en ajoutant la contrainte : $x^2 + y^2 = 1$. PAU et SCHICHO [181] ont montré qu'une telle approche se révèle en pratique inutilisable du fait du trop grand nombre de variables introduites.

Ces auteurs ont donc mis au point une variante de la méthode CAD s'appliquant à des formules contenant des occurrences des fonctions sinus et cosinus : les *polynômes trigonométriques*. Un polynôme trigonométrique est de la forme :

$$p(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n) = \sum_{i=1}^{m} a_i t_i$$

avec :

$$\begin{cases} a_i \in \mathbb{Q}[x_1, \ldots, x_k], \\ t_i = \prod_{l=1}^{s_i} \sin(x_{i_l})^{e_{1_l}} \cos(x_{i_l})^{e_{2_l}}, \quad e_{1_l} \in \mathbb{N}, e_{2_l} \in \mathbb{N},\ i_l \in \{k+1, \ldots, n\} \end{cases}$$

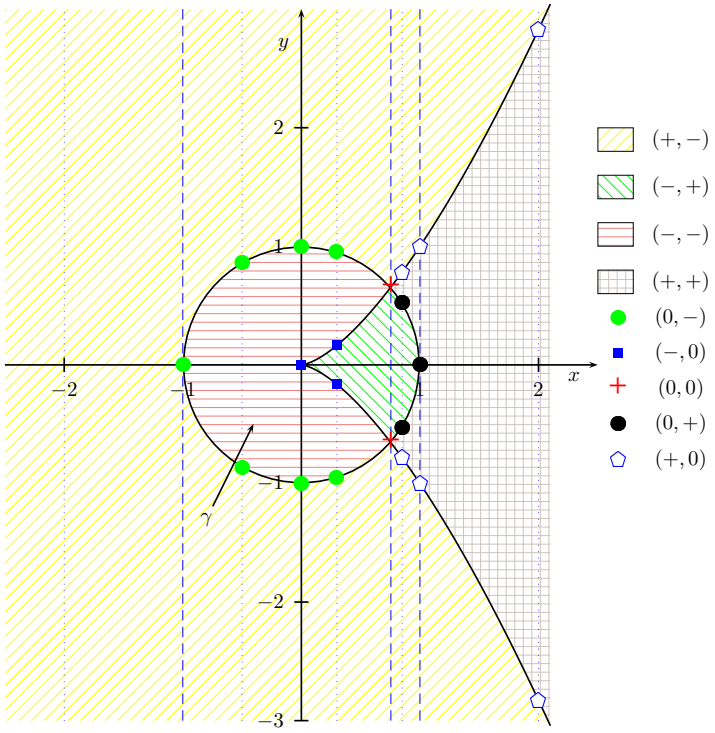


FIG. 5.2: Résolution de $\forall y\colon (x^2 + y^2 - 1 > 0 \wedge x^3 - y^2 < 0)$ par CAD

Dans son essence, la méthode proposée par PAU et SCHICHO est identique à la méthode CAD. La décomposition de $\mathbb{R}^n$ obtenue n'est cependant plus algébrique car les connexes calculés sont délimités par les racines de polynômes trigonométriques. On parle alors de *décomposition trigonométrique cylindrique*.

Bien que moins restrictive que CAD, cette méthode impose cependant des restrictions fortes sur la forme des contraintes manipulées :

– une séparation totale entre les variables algébriques et les variables trigonométriques ;
– les fonctions sinus et cosinus ne prennent en argument que des variables, pas des expressions.

5.2 Quantificateurs universels et approximations intérieures

Nous avons vu dans l'introduction à cette partie qu'il est possible d'associer plusieurs sens à un intervalle. SHARY [211] et KUPRIYANOVA [138] se sont intéressés en particulier au sens d'une équation de la forme :

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{5.2}$$

où $\boldsymbol{A}$ est une matrice d'intervalles et où $\boldsymbol{x}$ et $\boldsymbol{b}$ sont des vecteurs d'intervalles.

Ils ont ainsi isolé quatre définitions possibles de l'« ensemble solution » de l'équation (5.2) :

1. l'**ensemble unifié** (*united solution set*) :

$$\Sigma_{\exists\exists}(\boldsymbol{A}, \boldsymbol{b}) = \{x \in \mathbb{R}^n \mid \exists A \in \boldsymbol{A}, \exists b \in \boldsymbol{b}\colon Ax = b\}$$

où A est une matrice réelle, et x et b des vecteur réels.

2. l'**ensemble tolérable** (*tolerable solution set*) :

$$\Sigma_{\forall\exists}(\boldsymbol{A}, \boldsymbol{b}) = \{x \in \mathbb{R}^n \mid \forall A \in \boldsymbol{A}, \exists b \in \boldsymbol{b}\colon Ax = b\}$$

3. l'**ensemble contrôlable** (*controllable solution set*) :

$$\Sigma_{\exists\forall}(\boldsymbol{A},\boldsymbol{b}) = \{x \in \mathbb{R}^n \mid \forall b \in \boldsymbol{b}, \exists A \in \boldsymbol{A}, : Ax = b\}$$

4. l'**ensemble algébrique** (*algebraic solution*) :

$$\Sigma(\boldsymbol{A},\boldsymbol{b}) = \{\boldsymbol{x} \in \mathbb{I}^n_{\square} \mid \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}\} \tag{5.3}$$

avec l'égalité de deux intervalles prise au sens ensembliste ($I = J \iff \underline{I} = \underline{J} \wedge \overline{I} = \overline{J}$).

SHARY [211] remarque que le calcul de $\Sigma_{\exists\exists}$, $\Sigma_{\forall\exists}$ et $\Sigma_{\exists\forall}$ est coûteux et suggère de se contenter d'une *approximation intérieure* — c'est à dire, d'un sous-ensemble — de ces ensembles.

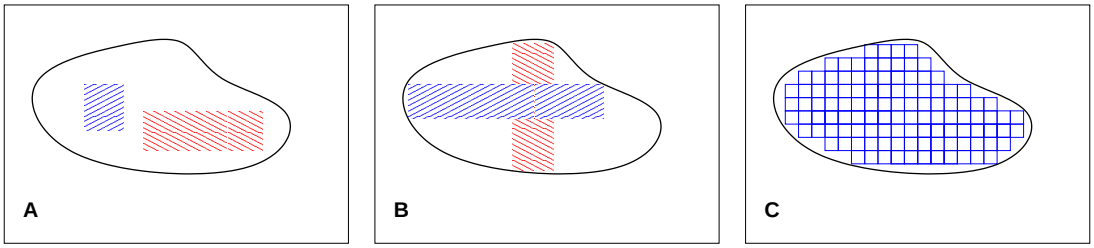


FIG. 5.3: Différentes approximations intérieures d'une relation réelle

Suivant l'application visée, il est possible de définir la notion d'approximation intérieure de différentes façons. Nous nous contenterons ici des trois définitions possibles suivantes, illustrées par la figure 5.3 :

Définition 5.1 (approximation intérieure faible — fig. 5.3-A). Étant donnée une relation réelle $\rho \subseteq \mathbb{R}^n$, on appellera *approximation intérieure faible* $\mathsf{Inner}_w(\rho)$ de ρ l'ensemble défini par :

$$\mathsf{Inner}_w(\rho) = \boldsymbol{B_1}, \quad \text{avec} \quad \boldsymbol{B_1} \in \{\boldsymbol{B} \in \mathbb{I}^n_{\circ} \mid \boldsymbol{B} \subseteq \rho\} \tag{5.4}$$

Une approximation intérieure faible de ρ est définie par toute boîte incluse dans ρ [156, 15].

On a là une définition très faible de l'approximation intérieure, qui présente cependant l'avantage d'être parfois « raisonnablement » facile à calculer. SHARY [213] utilise une définition plus forte de l'approximation intérieure, imposant une certaine notion de maximalité :

Définition 5.2 (Approximation intérieure faible maximale — fig. 5.3-B). Étant donnée une relation réelle $\rho \subseteq \mathbb{R}^n$, soit $\mathcal{B} = \{\boldsymbol{B} = I_1 \times \cdots \times I_n \mid \boldsymbol{B} \subseteq \rho \wedge \forall j \in \{1,\ldots,n\}, \forall I' \supseteq I_j : \boldsymbol{B}|_{I_j,I'} \subseteq \rho \Rightarrow I' = I_j\}$. On appellera *approximation intérieure faible maximale* $\mathsf{Inner}_m(\rho)$ tout pavé de la forme :

$$\mathsf{Inner}_m(\rho) = \boldsymbol{B_1}, \quad \text{avec} \quad \boldsymbol{B_1} \in \mathcal{B}$$

SHARY définit donc l'approximation intérieure de la relation ρ comme tout pavé maximalement inclus dans ρ (*i.e.* on ne peut étendre le pavé suivant aucune de ses dimensions sans inclure des éléments n'appartenant pas à ρ).

Les deux définitions précédentes apparaissent raisonnables lorsque la relation ρ considérée est connexe et que l'on ne s'intéresse qu'à obtenir un ensemble de solutions sûres sans contrainte sur la représentativité de l'espace des solutions par cet ensemble. Or, si dans le cadre qui nous intéresse (résolution de contraintes temporelles pour le placement automatique de caméras), l'espace des solutions se révèle souvent connexe, nous avons aussi le besoin d'offrir à l'utilisateur un ensemble de solutions le plus représentatif possible. C'est pourquoi nous considérerons une définition plus forte de l'approximation intérieure :

Définition 5.3 (Approximation intérieure forte — fig. 5.3-C). Étant donnée une relation réelle $\rho \subseteq \mathbb{R}^n$, on appellera *approximation intérieure forte* $\mathsf{Inner}_s(\rho)$ l'ensemble défini par :

$$\mathsf{Inner}_s(\rho) = \{\mathbf{r} \in \mathbb{R}^n \mid \mathsf{Hull}_{\circ}(\{\mathbf{r}\}) \subseteq \rho\} \tag{5.5}$$

Pour nous, une approximation intérieure d'une relation ρ est l'ensemble des éléments de ρ contenus dans un pavé flottant inclus dans ρ.

Comme le remarque SHARY, le calcul d'une approximation intérieure assure la validité des relations associées aux ensembles $\Sigma_{\exists\exists}$, $\Sigma_{\forall\exists}$ et $\Sigma_{\exists\forall}$. Plus généralement, de nombreux travaux ont été menés indépendamment pour résoudre « correctement » des systèmes de contraintes, ou pour gérer des contraintes avec variables quantifiées, en calculant une approximation intérieure des relations manipulées. Nous présenterons dans la section suivante une approche de calcul d'approximation intérieure par un processus d'évaluation/découpage. Nous décrirons deux instances particulières de cette méthode, renvoyant au chapitre 6 la présentation d'une troisième méthode développée par JARDILLIER et LANGUÉNOU pour des *contraintes temporelles* (contraintes avec le temps comme unique variable universellement quantifiée).

5.2.1 Calcul d'approximation intérieure par évaluation/découpage

Étant donnée une contrainte c et la relation réelle associée ρ_c, une méthode simple de calcul de l'approximation intérieure de ρ_c incluse dans un pavé $\boldsymbol{B}$ — au sens ici de sous-ensemble non nécessairement maximal — est de tester (par une méthode GlobSat() restant à définir) si la contrainte c est vérifiée pour tous les points de $\boldsymbol{B}$. On obtient alors un résultat en logique trivaluée : soit c est vérifiée par tous les points de $\boldsymbol{B}$; soit on peut prouver qu'il existe des éléments de $\boldsymbol{B}$ infirmant la contrainte ; soit enfin, on ne peut rien conclure. Dans ce dernier cas, on va découper $\boldsymbol{B}$ et retester la satisfiabilité de c sur chaque sous-pavé séparément. On a là une méthode générale de calcul d'approximation intérieure (*cf.* l'algorithme 5.1) pour laquelle il est nécessaire d'instancier la méthode GlobSat().

ALG. 5.1: Schéma général de calcul d'approximation intérieure par évaluation/découpage

```
1  CalculInner(in: c, B ∈ 𝕀^n ; out: 𝒰 ∈ 𝒫(𝕀^n))
2  début
3      sat ← GlobSat(c, B)
4      suivant (sat) dans
5          vrai:
6              retourner ({B})
7          faux:
8              retourner (∅)
9          inconnu:
10             si (Arrêt(B)) alors
11                 retourner (∅)
12             sinon
13                 (B_1, ..., B_k) ← Split(B)
14                 retourner (⋃_{j=1}^{k} CalculInner(c, B_k))
15             finsi
16     finsuivant
17 fin
```

La fonction GlobSat() teste si la contrainte c est vérifiée pour tous les points du pavé $\boldsymbol{B}$. La fonction Arrêt() teste s'il n'est plus nécessaire/possible de découper un pavé (test de canonicité, par exemple). La fonction Split() découpe un pavé en k sous-pavés suivant une stratégie à préciser (une instanciation fréquente consiste à découper le pavé initial en 2 suivant une de ses dimensions non-canoniques).

Dans le cas de contraintes de la forme $f(\boldsymbol{x})\diamond 0$ (avec $\diamond \in \{=, <, >, \leqslant, \geqslant\}$), il est possible de définir simplement une méthode GlobSat en utilisant l'arithmétique des intervalles pour évaluer le domaine de variation de f sur un pavé $\boldsymbol{B}$. Nous en verrons un exemple dans la section 5.2.1.1 décrivant succinctement les travaux de GARLOFF et GRAF [82] sur l'utilisation de l'extension de BERNSTEIN.

D'autres implémentations de la méthode GlobSat() sont possibles. Nous présenterons en particulier dans la section 5.2.1.2 [141] les travaux de KUTSIA et SCHICHO qui utilisent des critères *ad hoc* de détermination d'inexistence de racines dans un pavé pour des polynômes afin de résoudre des systèmes d'inégalités polynômiales.

5.2.1.1 Base de BERNSTEIN

De nombreux problèmes de *control design* tels que la détermination des conditions de stabilité d'un régulateur de température comme celui présenté dans l'exemple 4.13 peuvent se traduirent par un système d'inégalités polynômiales. Résoudre ce type de problème revient à déterminer des valeurs « sûres » pour toutes les variables impliquées, ce qui suppose donc la détermination d'une approximation intérieure de la relation réelle sous-jacente. ABDALLAH *et al.* ont montré [2, 1] que les méthodes symboliques d'élimination de quantificateurs tels que la méthode CAD permettent de résoudre des instances de petite taille de ces problèmes.

Afin de pouvoir traiter des problèmes plus gros, GARLOFF et GRAF [82, 80, 81, 83] proposent d'instancier le schéma général de calcul d'approximations intérieures décrit par l'algorithme 5.1 en utilisant une expansion en polynômes de BERNSTEIN des polynômes apparaissant dans les contraintes. Comme le rappellent BERCHTOLD *et al.* [33] en citant les travaux de FAROUKI et RAJAN [74, 75], les polynômes exprimés dans la base des polynômes de BERNSTEIN sont numériquement plus stables que lorsqu'ils sont exprimés dans la base des puissances. Aussi, l'extension naturelle aux intervalles de polynômes exprimés dans la base de BERNSTEIN s'avère plus précise (*i.e.* elle surestime moins le domaine de variation) que l'extension naturelle de ces mêmes polynômes dans la base des puissances. On pourra se reporter à la thèse de STAHL [218] pour une étude approfondie de la forme de BERNSTEIN, ainsi qu'à l'article de HONG et STAHL [114] prouvant que cette forme est monotone pour l'inclusion (*cf.* page 19).

Dans la suite, nous allons rappeler certaines notions sur la base de BERNSTEIN. Nous indiquerons en particulier la propriété fondamentale sur les coefficients d'un polynôme exprimé dans cette base en ce qui concerne la détermination de son domaine de variation sur un pavé donné. Nous indiquerons ensuite de quelle façon GARLOFF et GRAF utilisent l'expression d'un polynôme dans la base de BERNSTEIN pour calculer des approximations intérieures. Afin de simplifier l'exposé, nous nous restreindrons ici au cas de polynômes à une seule variable. Nous renvoyons le lecteur aux articles de GARLOFF et GRAF [82] ou à ceux de BERCHTOLD *et al.* [33] pour une généralisation de ces résultats au cas de polynômes à plusieurs variables.

Définition 5.4 (Polynôme de BERNSTEIN d'ordre n). Étant donnés un entier $n \in \mathbb{N}$ et un intervalle réel I, on appelle *k-ième polynôme de* BERNSTEIN *d'ordre n sur I* le polynôme $B^I_{n,k}(x)$ défini par :

$$B^I_{n,k}(x) = \binom{n}{k} \frac{(x - \underline{I})^k (\overline{I} - x)^{n-k}}{(\overline{I} - \underline{I})^n}$$

L'ensemble des $n+1$ polynômes de BERNSTEIN formant une base de l'espace des polynômes de degré inférieur ou égal à n, il est possible d'exprimer ces polynômes indifféremment dans la base des puissances ou dans la base de BERNSTEIN. On a ainsi :

$$\forall I \in \mathbb{I}, \forall x \in I : p(x) = \sum_{k=0}^{n} a_k x^k = \sum_{k=0}^{n} b^I_{n,k} B^I_{n,k}(x)$$

où $b^I_{n,k}$ est le k-ième *coefficient de* BERNSTEIN d'ordre n de p sur l'intervalle I défini par [114] :

$$\begin{aligned} b^I_{n,k} &= \sum_{i=0}^{n} \sum_{j=0}^{\min(k,i)} \frac{\binom{k}{j}\binom{i}{j}}{\binom{n}{j}} \mathsf{w}(I)^j \underline{I}^{i-j} a_i \\ &= \sum_{i=0}^{n} \sum_{j=0}^{\min(n-k,i)} \frac{\binom{n-k}{j}\binom{i}{j}}{\binom{n}{j}} \mathsf{w}(I)^j \overline{I}^{i-j} a_i \end{aligned}$$

Exemple 5.17 (Forme de BERNSTEIN). *Soit le polynôme* $f(x) = (x^6/4 - 1{,}9x^5 + 0{,}25x^4 + 20x^3 - 17x^2 - 55{,}6x + 48)/50$ *introduit à la section 2.1.2.4. Son expression en forme de* BERNSTEIN *sur l'intervalle* $I = [-3..5{,}5]$

est :

$$\begin{aligned} B_f(x) &= 3{,}72B^I_{6,0} - 12{,}56B^I_{6,1} + 32{,}16B^I_{6,2} - 39{,}59B^I_{6,3} + 29{,}70B^I_{6,4} - 14{,}02B^I_{6,5} + 2{,}84B^I_{6,6} \\ &= 7{,}53 \cdot 10^{-6}(x+3)^6 + 9{,}86 \cdot 10^{-6}(-x+5.5)^6 - 2{,}0 \cdot 10^{-4}(x+3)(-x+5.5)^5 - \\ &\quad 2{,}23 \cdot 10^{-4}(x+3)^5(-x+5.5) + 1{,}28 \cdot 10^{-3}(x+3)^2(-x+5.5)^4 - \\ &\quad 2{,}10 \cdot 10^{-3}(x+3)^3(-x+5.5)^3 + 1{,}18 \cdot 10^{-3}(x+3)^4(-x+5.5)^2 \end{aligned}$$

La figure 5.4 — à comparer avec la figure 2.4, p. 23 — présente le résultat de l'évaluation de f pour des intervalles de taille 0,02 sur le domaine $I = [-3\,..\,5.5]$. On notera que la relative surestimation du domaine de variation de f est largement imputable au fait que les coefficients de BERNSTEIN *ont été calculés pour le « grand » domaine $[-3\,..\,5{,}5]$ de taille $\mathsf{w} = 8{,}5$.*

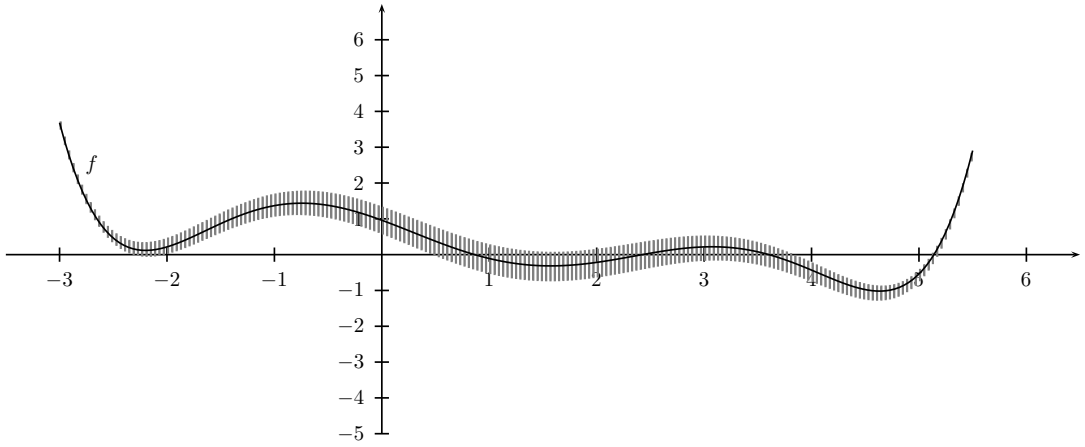


FIG. 5.4: Évaluation de $f(x)$ en forme de BERNSTEIN ($\mathsf{w}(I_j) = 0{,}02$)

On a le théorème important :

Théorème 5.1 (Encadrement du domaine de variation [200]). *Soit $p(x) \in \mathbb{R}[x]$ un polynôme de degré n, et I un intervalle réel. Il vient :*

$$\forall \mathsf{r} \in I : \min_{k=0,\ldots,n} b^I_{n,k} \leqslant p(\mathsf{r}) \leqslant \max_{k=0,\ldots,n} b^I_{n,k}$$

Ainsi les coefficients de BERNSTEIN permettent de borner le domaine de variation du polynôme p sur l'intervalle I.

GARLOFF et GRAF utilisent une généralisation du théorème 5.1 au cas de polynômes à plusieurs variables pour calculer une approximation intérieure pour un système d'inégalités polynômiales de la façon suivante : étant donnés un pavé $\boldsymbol{B}$ et un polynôme $p \in \mathbb{R}[x_1, \ldots, x_n]$ de degré m, on évalue les coefficients de BERNSTEIN de p sur $\boldsymbol{B}$. Il vient alors :

$$\begin{aligned} \min_{\boldsymbol{k}} b^I_{m,\boldsymbol{k}} > 0 \Rightarrow p(\mathbf{r}) > 0, \quad \forall \mathbf{r} \in \boldsymbol{B} \\ \max_{\boldsymbol{k}} b^I_{m,\boldsymbol{k}} \leqslant 0 \Rightarrow p(\mathbf{r}) \leqslant 0, \quad \forall \mathbf{r} \in \boldsymbol{B} \end{aligned}$$

où $\boldsymbol{k}$ est un multi-index de la forme $\boldsymbol{k} = (k_1, \ldots, k_l)$.

Les relations ci-dessus fournissent une implémentation possible pour la méthode GlobSat() : par exemple, si l'on cherche une approximation intérieure pour la relation associée à la contrainte $p(\boldsymbol{x}) > 0$, il suffit de calculer les coefficients de BERNSTEIN et de s'assurer qu'ils sont tous positifs.

Un inconvénient de la méthode de GARLOFF et GRAF est qu'elle est limitée, comme la méthode CAD, au cas de contraintes polynômiales. De même, l'expression d'un polynôme dans la base de BERNSTEIN s'avère beaucoup plus coûteuse à évaluer (comparer $f(x)$ et $B_f(x)$ dans l'exemple 5.17).

5.2.1.2 Utilisation d'un critère d'inexistence de racines

Une autre instance de la méthode d'évaluation/découpage, due à KUTSIA et SCHICHO [141], s'appliquant à des conjonctions/disjonctions d'inégalités polynômiales strictes, calcule un sous-ensemble de l'approximation intérieure de la relation correspondante qu'ils appellent *ε-solution set*, où ε correspond au niveau de relaxation autorisé pour les contraintes (*e.g.* une contrainte de la forme $f(\boldsymbol{x}) > 0$ est identifiée à la contrainte $f(\boldsymbol{x}) > \varepsilon$).

En utilisant le critère d'inexistence de racine d'un polynôme dans un pavé donné par la proposition 5.1 ci-dessous, KUTSIA et SCHICHO implémentent GlobSat() de la façon suivante : pour un polynôme $p(x_1, \ldots, x_n)$, une contrainte c: $p \diamond 0$ avec $\diamond \in \{<, >\}$ et un pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n$:

- si $p(x_1, \ldots, x_n)$ n'a aucune racine dans $\boldsymbol{B}$ et que le point $(\underline{I_1}, \ldots, \underline{I_n})$ satisfait c, alors GlobSat$(c, \boldsymbol{B})$ vaut **vrai** ;
- si $p(x_1, \ldots, x_n)$ n'a aucune racine dans $\boldsymbol{B}$ et que le point $(\underline{I_1}, \ldots, \underline{I_n})$ ne satisfait pas c, alors GlobSat$(c, \boldsymbol{B})$ vaut **faux** ;
- si $p(x_1, \ldots, x_n)$ a une ou plusieurs racines dans $\boldsymbol{B}$, alors GlobSat$(c, \boldsymbol{B})$ vaut **inconnu**.

Proposition 5.1 (Inexistence de racine dans un pavé [141]). *Étant donnés un pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n$ et un polynôme réel $p(x_1, \ldots, x_n)$, soit $\eta_c^{\boldsymbol{B}}$ le nombre de racines de p sur $\boldsymbol{B}$. Pour tout entier $i \in \{1, \ldots, n\}$, notons ∂_p^i la dérivée partielle de l'extension naturelle aux intervalles de $p(x_1, \ldots, x_n)$ par rapport à la variable x_i. On a la relation :*

$$|p(\underline{I_1}, \ldots, \underline{I_n})| > \sum_{i=1}^{n} \mathsf{w}(I_i)\overline{\partial_p^i} \quad \Rightarrow \quad \eta_c^{\boldsymbol{B}} = 0$$

La proposition 5.1 ne donne qu'une condition suffisante d'inexistence de racine. Il n'est donc pas toujours possible de déterminer exactement s'il existe ou non une racine. Dans ce cas, les auteurs utilisent le paramètre ε pour relâcher la contrainte c. Le processus de détermination d'inexistence de racines sur un pavé est entrelacé avec des étapes de découpage « intelligente » lorsque GlobSat() retourne « inconnu » : les auteurs choisissent de découper le pavé initial en 2^n pavés en découpant chaque intervalle en un point tel que les pavés obtenus ont une valeur de vérité décidable pour GlobSat().

5.2.2 Arithmétiques de KAUCHER/MARKOV et arithmétique modale

Note : Dans la suite, nous ne considérerons que les intervalles réels à bornes fermées. Afin d'alléger les notations, nous identifierons donc l'ensemble $\mathbb{I}$ à l'ensemble $\mathbb{I}_{\square}^{\mathbb{R}}$.

Nous avons vu dans la section 2.1.1 de la partie I que l'arithmétique d'intervalles ne possède pas d'inverse pour les opérations $+$ et $\times$.
En remarquant la forte analogie entre les deux faits suivants :

- une contrainte sur la variable x de la forme $a + x = b$ (avec $\{x, a, b\} \subset \mathbb{R}$) n'a pas de solution dans $\mathbb{R}^+$ lorsque $a > b$,
- une contrainte sur la variable X de la forme $A + X = B$ (avec $\{X, A, B\} \subset \mathbb{I}$) n'a pas de solution dans $\mathbb{I}$ lorsque $\mathsf{w}(A) > \mathsf{w}(B)$,

KAUCHER [129] a eu l'idée de remplacer l'ensemble $\mathbb{I} = \{[\mathsf{r}\,..\,\mathsf{s}] | \mathsf{r}, \mathsf{s} \in \mathbb{R}, \mathsf{r} \leqslant \mathsf{s}\}$ par l'ensemble $\mathcal{D} = \{[\mathsf{r}\,..\,\mathsf{s}] | \mathsf{r}, \mathsf{s} \in \mathbb{R}\}$ où la borne gauche peut être strictement supérieure à la borne droite (à rapprocher de l'introduction des nombres réels négatifs). L'intérêt d'une telle extension réside en ceci : alors que $(\mathbb{I}, +)$ et $(\mathbb{I}, \times)$ ne sont que des monoïdes, il suffit d'étendre naturellement la définition des lois de composition internes $+$ et $\times$ sur $\mathcal{D}$ pour que $(\mathcal{D}, +)$ et $(\mathcal{D} \setminus \mathcal{T}, \times)$ soient des groupes (avec $\mathcal{T}$ défini ci-dessous). Dans la littérature, les éléments de $\mathcal{D}$ sont appelés *intervalles étendus* ou *intervalles dirigés* ; c'est cette dernière dénomination que nous utiliserons dans la suite.

En suivant les notation de MARKOV [156, 157, 158], nous allons présenter rapidement l'arithmétique de KAUCHER $\mathbb{K} = (\mathcal{D}, +, \times)$ puis nous discuterons son utilisation pour le calcul d'approximations intérieures. La notation employée nous permettra alors de présenter dans un cadre unifié l'arithmétique de MARKOV, une autre extension de l'arithmétique des intervalles où l'on a conservé l'ensemble $\mathbb{I}$ mais où l'on a ajouté de nouveaux opérateurs.

Notations : Étant donné un intervalle dirigé I, soit I^- sa borne gauche et I^+ sa borne droite. On dira que I est un *intervalle propre* (*proper interval*) si $I^- \leqslant I^+$ et un *intervalle impropre* (*improper interval*) si

$I^- > I^+$. Soit $\lambda \in \Lambda, \Lambda = \{-, +\}$, le signe distinguant une borne gauche d'une borne droite. On définit les relations suivantes pour $\mu, \nu \in \Lambda$: $++ = -- = +$ et $+- = -+ = -$.
Pour $I = [I^- .. I^+]$, soit $\mathrm{dual}(I) = [I^+ .. I^-]$ le *dual* de I et $\mathrm{prop}(I)$ l'*intervalle propre associé à* I défini par :

$$\mathrm{prop}(I) = \begin{cases} [I^- .. I^+] & \text{si } I^- \leqslant I^+ \\ [I^+ .. I^-] & \text{si } I^- > I^+ \end{cases}$$

Pour $\mathbb{I}$ l'ensemble des intervalles propres, soit $\overline{\mathbb{I}}$ l'ensemble des intervalles impropres. On a donc : $\mathcal{D} = \mathbb{I} \cup \overline{\mathbb{I}}$. Définissons les ensembles suivants :

$$\begin{array}{lll} Z & = \{I \in \mathbb{I} \mid I^- \leqslant 0 \leqslant I^+\} & \qquad Z^* & = \{I \in \mathbb{I} \mid I^- < 0 < I^+\} \\ \overline{Z} & = \{I \in \overline{\mathbb{I}} \mid I^+ \leqslant 0 \leqslant I^-\} & \qquad \overline{Z}^* & = \{I \in \overline{\mathbb{I}} \mid I^- \leqslant I^+\} \\ \mathcal{T} & = Z \cup \overline{Z} & \qquad \mathcal{T}^* & = Z^* \cup \overline{Z}^* \end{array}$$

On définit alors le *signe* $\sigma(I)$ *d'un intervalle* $I \in \mathcal{D}^* = \mathcal{D} \setminus \mathcal{T}^*$ par :

$$\sigma(I) = \begin{cases} + & \text{si } I^- \geqslant 0 \wedge I^+ \geqslant 0 \\ - & \text{si } I^- \leqslant 0 \wedge I^+ \leqslant 0 \end{cases}$$

Les opérations arithmétiques sur $\mathcal{D}$ correspondent à une extension du domaine de définition des opérations sur $\mathbb{I}$ [156, 157, 158, 185] (*cf.* table 5.1).

Nous avons dit plus haut que $(\mathcal{D}, +)$ et $(\mathcal{D}, \times)$ étaient des groupes. On a donc un élément neutre pour l'addition ($[0 .. 0]$) et un élément neutre pour la multiplication ($[1 .. 1]$) ; à la différence des éléments de $\mathbb{I}$, chaque élément de $\mathcal{D}$ (resp. $\mathcal{D} \setminus \mathcal{T}$) possède aussi un inverse pour l'addition (resp. la multiplication). On pose :

$$\begin{array}{lll} \forall I \in \mathcal{D}, & -_h I & = [-I^- .. -I^+] \\ \forall I \in \mathcal{D} \setminus \mathcal{T}, & 1/_h I & = [1/I^- .. 1/I^+] \end{array}$$

Il est alors aisé de vérifier que pour tout intervalle dirigé I, on a $I + (-_h I) = [0 .. 0]$ et pour tout $I \in \mathcal{D} \setminus \mathcal{T}$, $I \times (1/_h I) = [1 .. 1]$.

TAB. 5.1: Arithmétique de KAUCHER

$$I + J = [I^- + J^- .. I^+ + J^+] \qquad I, J \in \mathcal{D}$$

$$I - J = [I^- - J^+ .. I^+ - J^-] \qquad I, J \in \mathcal{D}$$

$$I \times J = \begin{cases} [I^{-\sigma(J)} J^{-\sigma(I)} .. I^{\sigma(J)} J^{\sigma(I)}] & I, J \in \mathcal{D}^* \\ [I^{\sigma(I)} J^{-\sigma(I)} .. I^{\sigma(I)} J^{\sigma(I)}] & I \in \mathcal{D}^*, J \in \mathcal{T}^* \\ [I^{-\sigma(J)} J^{\sigma(J)} .. I^{\sigma(J)} J^{\sigma(J)}] & I \in \mathcal{T}^*, J \in \mathcal{D}^* \\ [\min\{I^- J^+, I^+ J^-\} .. \max\{I^- J^-, I^+ J^+\}] & I, J \in Z^* \\ [\max\{I^- J^-, I^+ J^+\} .. \min\{I^- J^+, I^+ J^-\}] & I, J \in \overline{Z}^* \\ 0 & (I \in Z^*, J \in \overline{Z}^*) \vee (I \in \overline{Z}^*, J \in Z^*) \end{cases}$$

$$I/J = \begin{cases} [I^{-\sigma(J)}/J^{\sigma(I)} .. I^{\sigma(J)}/J^{-\sigma(I)}] & I \in \mathcal{D}^*, J \in \mathcal{D} \setminus \mathcal{T} \\ [I^{-\sigma(J)}/J^{-\sigma(J)} .. I^{\sigma(J)}/J^{-\sigma(J)}] & I \in \mathcal{T}^*, J \in \mathcal{D} \setminus \mathcal{T} \end{cases}$$

Malgré ses bonnes propriétés, l'arithmétique de KAUCHER souffre d'un inconvénient majeur, relevé par MARKOV [158] : quelle interprétation donner au résultat d'un calcul lorsque c'est un intervalle impropre ?

Certaines interprétations existent dans des cas particuliers, comme par exemple la résolution de systèmes d'équations linéaires (voir le début de la section 5.2) :

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{5.6}$$

avec $\boldsymbol{A}$ une matrice $n \times n$ d'intervalles et $\boldsymbol{x}$ et $\boldsymbol{b}$ des vecteurs de taille n d'intervalles.

La solution algébrique (*cf.* éq. (5.3), p. 68) joue dans ce cas un rôle très important. En effet, SHARY a montré [211] les propriétés suivantes pour des calculs effectués avec l'arithmétique de KAUCHER :

- si le vecteur $\boldsymbol{x}_a$ est une solution algébrique à l'équation $\boldsymbol{A}\boldsymbol{x} = \mathrm{dual}(\boldsymbol{b})$ tel que tous les intervalles le composant sont impropres, alors son dual est une approximation intérieure de l'ensemble $\Sigma_{\exists\exists}$;
- si le vecteur $\boldsymbol{x}_a$ est une solution algébrique à l'équation (5.6), alors :
  - si tous les intervalles le composant sont impropres, son dual est une approximation intérieure de $\Sigma_{\exists\forall}$,
  - si tous les intervalles le composant sont propres, c'est une approximation intérieure de l'ensemble $\Sigma_{\forall\exists}$.

De même, KUPRIYANOVA [140, 138, 139] a prouvé que la solution algébrique à l'équation $\mathrm{dual}(\boldsymbol{A})\boldsymbol{x} = \boldsymbol{b}$ est une approximation intérieure maximale (au sens de l'inclusion) de l'ensemble $\Sigma_{\exists\exists}$ pour l'équation (5.6).

L'utilisation de l'arithmétique de KAUCHER revêt ici une importance capitale puisqu'elle seule semble permettre de trouver une solution algébrique au problème (5.6). POPOVA et ULLRICH décrivent succinctement [185] un algorithme dû à ZYUZIN [254] permettant de calculer par une méthode itérative une solution algébrique à l'équation (5.6).

Une interprétation plus systématique est celle de l'*arithmétique d'intervalles modaux* [78, 13, 14, 98] (ou arithmétique modale) où l'on associe aux intervalles une *modalité* (quantificateur) suivant qu'ils sont propres ou impropres. On a ainsi :

$$\begin{array}{lll} I = [3\,..\,6] & = ([3\,..\,6], \exists) & \text{Interprétation : « il existe une valeur dans } I \text{ »} \\ I = [6\,..\,3] & = ([3\,..\,6], \forall) & \text{Interprétation : « pour toute valeur dans } \mathrm{dual}(I) \text{ »} \end{array}$$

Pour une relation sur $n+1$ variables $x_1, \ldots, x_n, y$ de la forme :

$$y = f(x_1, \ldots, x_n) \tag{5.7}$$

où f est une fonction réelle continue, soient $(i_1, \ldots, i_k)$ et $(j_1, \ldots, j_l)$ (avec $k + l = n$) deux vecteurs d'indices tels que les domaines des variables $x_{i_1}, \ldots, x_{i_k}$ sont des intervalles propres et ceux des variables $x_{j_1}, \ldots, x_{j_l}$ des intervalles impropres. Soit I_m le domaine de la variable x_m et soit Ξ la modalité du domaine de y. L'arithmétique modale interprète alors l'équation 5.7 comme :

$$\forall x_{i_1} \in I_{i_1} \ldots, \forall x_{i_k} \in I_{i_k}, \Xi y \in I_y, \exists x_{j_1} \in I_{j_1} \ldots, \exists x_{j_l} \in I_{j_l} : y = f(x_1, \ldots, x_n)$$

Le résultat ci-dessus reste vrai dans le cas de calculs avec des intervalles flottants si l'on prend la précaution d'arrondir tous les résultats vers l'extérieur, comme on le fait dans le cas de l'arithmétique d'intervalles classique. On trouvera chez ARMENGOL *et al.* [15, 237] une interprétation duale lorsque l'on utilise un arrondi vers l'intérieur.

Exemple 5.18 (Interprétation pour l'arithmétique modale [98]). *Soient trois variables x, y, z, avec les domaines $I_x = [1\,..\,3]$ et $I_y = [4\,..\,8]$. Suivant que l'on considère les domaines I_x et I_y ou leurs duaux, on a les interprétations :*

$$\begin{array}{lll} I_x + I_y = [1\,..\,3] + [4\,..\,8] = [5\,..\,11] & \iff & \forall x \in I_x, \forall y \in I_y, \exists z \in [5\,..\,11] : z = x + y \\ I_x + \mathrm{dual}(I_y) = [1\,..\,3] + [8\,..\,4] = [9\,..\,7] & \iff & \forall x \in I_x, \forall z \in [7\,..\,9], \exists y \in I_y : z = x + y \\ \mathrm{dual}(I_x) + I_y = [3\,..\,1] + [4\,..\,8] = [7\,..\,9] & \iff & \forall y \in I_y, \exists z \in [7\,..\,9], \exists x \in I_x : z = x + y \end{array}$$

En utilisant les deux interprétations possibles en arithmétique modale, ARMENGOL *et al.* [15] ont défini un algorithme permettant de d'encadrer l'espace d'états possibles d'un simulateur pour un système complexe par une approximation intérieure et une approximation extérieure. Leur méthode s'applique cependant uniquement à des systèmes modélisables par des fonctions rationnelles.

Une autre façon d'interpréter le résultat de calculs faits avec l'arithmétique de KAUCHER a été proposée par MARKOV [156] : indépendamment des travaux de KAUCHER, GARDEÑES, ARMENGOL *et al.* , il a défini une extension de l'arithmétique des intervalles, que nous appellerons dans la suite *arithmétique de* MARKOV. À la différence de KAUCHER, MARKOV a gardé l'ensemble des intervalles $\mathbb{I}$ en ajoutant de nouveaux opérateurs, dits *opérateurs arithmétiques intérieurs*, notés $(+^-, -^-, \times^-, /^-)$ (*cf.* table 5.2).

Notation : Soit $\mathbb{I}_\circ^{\mathbb{R}^*} = \mathbb{I}_\circ^{\mathbb{R}} \setminus Z^*$. On définit l'opérateur $\chi \colon \mathbb{I}_\circ^{\mathbb{R}^*} \setminus \{0\} \to [0\,..\,1]$ par :

$$\forall I \in \mathbb{I}_\circ^{\mathbb{R}^*} \setminus \{0\} \colon\ \chi(I) = \begin{cases} I^-/I^+ & \text{si } \sigma(I) = + \\ I^+/I^- & \text{si } \sigma(I) = - \end{cases}$$

TAB. 5.2: Opérations arithmétiques intérieures de MARKOV

$$I +^- J = \begin{cases} [I^- + J^+ \,..\, I^+ + J^-] & I, J \in \mathcal{D}, \mathsf{w}(I) \geqslant \mathsf{w}(J) \\ [I^+ + J^- \,..\, I^- + J^+] & I, J \in \mathcal{D}, \mathsf{w}(I) < \mathsf{w}(J) \end{cases}$$

$$I -^- J = \begin{cases} [I^- - J^- \,..\, I^+ - J^+] & I, J \in \mathcal{D}, \mathsf{w}(I) \geqslant \mathsf{w}(J) \\ [I^+ - J^+ \,..\, I^- - J^-] & I, J \in \mathcal{D}, \mathsf{w}(I) < \mathsf{w}(J) \end{cases}$$

$$I \times^- J = \begin{cases} [I^{-\sigma(J)} J^{\sigma(I)} \,..\, I^{\sigma(J)} J^{-\sigma(I)}] & \text{si } \chi(J) \geqslant \chi(I) \\ [I^{\sigma(J)} J^{-\sigma(I)} \,..\, I^{-\sigma(J)} J^{\sigma(I)}] & \text{si } \chi(J) < \chi(I) \end{cases}$$

$$I /^- J = \begin{cases} [I^{-\sigma(J)} / J^{-\sigma(I)} \,..\, I^{\sigma(J)} J^{\sigma(I)}] & \text{si } \chi(J) \geqslant \chi(I) \\ [I^{\sigma(J)} J^{\sigma(I)} \,..\, I^{-\sigma(J)} J^{-\sigma(I)}] & \text{si } \chi(J) < \chi(I) \end{cases}$$

Une propriété fondamentale des opérateurs intérieurs est que l'on a : $I \top^- J \subseteq I \top J$ pour $\top \in \{+^-, -^-, \times^-, /^-\}$ et $I, J \in \mathbb{I}$.

Ce résultat peut se généraliser de la façon suivante :

Proposition 5.2 ([156]). *Soient f et g deux fonctions continues sur un connexe D de $\mathbb{R}$. Pour tout opérateur $\top \in \{+, -, \times, /\}$ et pour tout domaine connexe $I \subseteq D$, on a :*

$$f(I) \top^- g(I) \subseteq (f \top g)(I) \subseteq f(I) \top g(I) \tag{5.8}$$

La relation (5.8) nous montre que l'on peut utiliser les opérateurs intérieurs pour calculer des approximations intérieures et les opérateurs usuels pour calculer des approximations extérieures.

Un des grands résultats de MARKOV a été de montrer [156] qu'il est possible de passer de l'arithmétique de KAUCHER $(\mathcal{D}, +\times)$ à l'arithmétique de MARKOV $(\mathbb{I}, +, +^-, \times, \times^-)$ et vice-versa. Ainsi, on peut effectuer les calculs dans l'arithmétique de KAUCHER en profitant de ses bonnes propriétés, et interpréter au final le résultat dans l'arithmétique de MARKOV afin d'obtenir des intervalles propres.

Les différentes arithmétiques présentées ci-dessus présentent de grandes qualités en ce qu'elles permettent de déterminer mécaniquement à la fois des approximations intérieures et des approximations extérieures. Elles souffrent cependant pour le moment de certaines limitations (restriction aux fonctions rationnelles). L'arithmétique modale permet de calculer des approximations fines (intérieures et/ou extérieures) du domaine de variations de fonctions rationnelles à condition de calculer au préalable leurs dérivées par rapport à chacune des variables y apparaissant afin d'obtenir des informations de monotonie. Elle semble cependant être utilisée avec succès dans le cas de la détermination de la stabilité de certains systèmes en *control design*. Il serait donc intéressant de comparer cette approche avec celles présentées plus haut (méthode CAD, expansion en polynômes de BERNSTEIN...) qui s'intéressent au même type de problèmes.

CHAPITRE 6

Approximations intérieures, variables quantifiées et contraintes d'intervalles

Approximations intérieures — contraintes d'intervalles et variables universellement quantifiées — calcul d'approximation intérieure par évaluation/découpage — calcul d'approximation intérieure par box-consistance sur la négation — comparaison des approches

NOS TRAVAUX SUR LA RÉSOLUTION de contraintes réelles non-linéaires avec variables universellement quantifiées ont été motivés par le problème suivant, dit *problème du cameraman virtuel* [120] : étant donnée une « scène » de durée $d = [t_i..t_f]$ décrite par l'ensemble des trajectoires des objets la composant, on recherche la position et l'orientation à tout instant $t \in [t_i\,..\,t_f]$ d'une caméra filmant la scène en respectant un certain nombre de contraintes exprimées dans un langage cinématographique (*e.g.* « le personnage 1 doit toujours se trouver à la gauche du personnage 2 », « la voiture doit être vue de face durant toute la scène et occuper le tiers haut gauche de l'écran »…).

Le *contrôle de caméra* est un problème important que l'on retrouve dans des domaines aussi variés que la robotique [3], la modélisation d'environnements virtuels [37], ou le cinéma [60]. Dans tous les cas, on cherche à offrir à un utilisateur une vue adéquate de certains centres d'intérêt dans une scène pendant un temps prédéfini. Un problème classique en robotique [227], par exemple, est de déterminer la position d'une caméra permettant d'observer à tout moment la pièce usinée par un robot.

Une approche communément rencontrée pour résoudre ce type de problème consiste à laisser l'utilisateur interagir directement avec les différents paramètres de la caméra et à sélectionner une seule solution par un processus d'optimisation d'un certain critère. Un inconvénient majeur de cette approche est qu'elle requiert une bonne maîtrise par l'utilisateur des conséquences des manipulations de chaque paramètre.

D'autres travaux [37, 85, 70] ont essayé de déterminer les mouvements d'une caméra à partir des propriétés désirées pour la scène à observer. Ils ne s'intéressent cependant pas au problème général du déplacement de la caméra dans une scène animée et se contentent, pour la plupart, d'une seule solution.

Suivant les travaux de SNYDER [217], JARDILLIER et LANGUÉNOU [120] ont, quant à eux, choisis de déterminer les mouvements d'une caméra en utilisant des méthodes de résolution de contraintes d'intervalles : les trajectoires de tous les objets d'une scène, ainsi que le déplacement et l'orientation de la caméra sont définis grâce à des contraintes où le temps apparaît comme une variable universellement quantifiée. La résolution du système obtenu se fait par une instance du schéma général d'évaluation/découpage décrit par l'algorithme 5.1 de la page 69 en utilisant l'arithmétique d'intervalles.

En nous basant sur leurs travaux, nous avons défini de nouveaux algorithmes de résolution de systèmes de contraintes avec variables universellement quantifiées utilisant des consistances locales — telles que la box-consistance — et la propagation de domaines. Nous verrons au chapitre 12 que ces algorithmes permettent de gagner plusieurs ordres de grandeur pour la résolution de problèmes pratiques.

En nous basant sur la définition 5.3 d'une approximation intérieure, nous commencerons par définir un opérateur pour son calcul et formulerons certaines de ses propriétés. Nous décrirons ensuite la méthode de JARDILLIER et LANGUÉNOU pour gérer des contraintes de la forme $\forall t \in I_t : c_1 \wedge \cdots \wedge c_m$ (avec $c_i : f(x_1, \ldots, x_n) \diamond 0, \quad i \in \{1, \ldots, m\}, \diamond \in \{\leqslant, \geqslant\}$) en la formalisant et en donnant une preuve de sa correction qui manquait à l'article original. Enfin, nous présenterons les algorithmes mis au point durant notre thèse pour résoudre les contraintes temporelles rencontrées lors de la modélisation de mouvements de caméra. Leur validation et la description des

jeux de tests utilisés fait l'objet du chapitre 12.

6.1 Approximation intérieure

Symétriquement à la définition de l'opérateur d'approximation extérieure Hull (voir p. 18), on peut définir un opérateur d'approximation intérieure (forte, *cf.* déf.5.3) de la façon suivante :

Définition 6.1 (Opérateur d'approximation intérieure). Étant donnée une relation réelle n-aire ρ, un *opérateur d'approximation intérieure* $\mathsf{Inner}_\circ : \mathbb{R}^n \to \mathbb{R}^n$ *pour* ρ est défini par :

$$\mathsf{Inner}_\circ(\rho) = \{\mathbf{r} \in \mathbb{R}^n \mid \mathsf{Hull}_\circ(\{\mathbf{r}\}) \subseteq \rho\}$$

Un opérateur d'approximation intérieure jouit des propriétés suivantes, qui nous serviront dans la suite pour établir certaines preuves de correction :

Proposition 6.1 (Propriétés de l'opérateur d'approximation Inner). *L'opérateur* Inner *est contractant, monotone, idempotent et distributif par rapport à l'union et l'intersection de sous-ensembles de* $\mathbb{R}^n$.

Démonstration. Étant donnés $\mathcal{A}$ et $\mathcal{B}$ deux sous-ensembles de $\mathbb{R}^n$ tels que $\mathcal{A} \subseteq \mathcal{B}$, soit $\mathbf{r}$ un élément de $\mathbb{R}^n$ et $\boldsymbol{D} = \mathsf{Hull}_\circ(\{\mathbf{r}\})$.

Contractance. Conséquence immédiate de la définition de $\mathsf{Inner}_\circ$ puisque $\{\mathbf{r}\} \subseteq \mathsf{Hull}_\circ(\{\mathbf{r}\})$;

Monotonie. Comme $\mathbf{r} \in \mathsf{Inner}_\circ(\mathcal{A})$ entraine $\boldsymbol{D} \subseteq \mathcal{A}$, on a $\boldsymbol{D} \subseteq \mathcal{B}$, et donc, par définition de Inner, $\mathbf{r} \in \mathsf{Inner}_\circ(\mathcal{B})$;

Idempotence. Commençons par montrer l'inclusion $\mathsf{Inner}_\circ(\mathsf{Inner}_\circ(\mathcal{A})) \subseteq \mathsf{Inner}_\circ(\mathcal{A})$:
pour $\mathbf{r} \in \mathsf{Inner}_\circ(\mathsf{Inner}_\circ(\mathcal{A}))$, la définition de Inner implique que $\boldsymbol{D} \subseteq \mathsf{Inner}_\circ(\mathcal{A})$. Comme $\mathsf{Inner}_\circ(\mathcal{A}) \subseteq \mathcal{A}$ (par contractance de Inner), nous avons $\boldsymbol{D} \subseteq \mathcal{A}$, et donc $\mathbf{r} \in \mathsf{Inner}_\circ(\mathcal{A})$.
Établissons maintenant l'inclusion $\mathsf{Inner}_\circ(\mathsf{Inner}_\circ(\mathcal{A})) \supseteq \mathsf{Inner}_\circ(\mathcal{A})$: pour $\mathbf{r} \in \mathsf{Inner}_\circ(\mathcal{A})$, nous avons $\boldsymbol{D} \subseteq \mathcal{A}$, par définition de Inner. Par conséquent, tous les éléments de $\boldsymbol{D}$ sont aussi dans $\mathsf{Inner}_\circ(\mathcal{A})$. Donc $\boldsymbol{D} \subseteq \mathsf{Inner}_\circ(\mathcal{A})$ et $\mathbf{r} \in \mathsf{Inner}_\circ(\mathsf{Inner}_\circ(\mathcal{A}))$;

Distributivité par rapport à l'union. En utilisant une fois de plus la définition de Inner, il vient $\mathbf{r} \in \mathsf{Inner}_\circ(\mathcal{A} \cup \mathcal{B})$ si et seulement si $\boldsymbol{D} \subseteq \mathcal{A} \cup \mathcal{B}$, ce qui est équivalent à $\boldsymbol{D} \subseteq \mathcal{A} \vee \boldsymbol{D} \subseteq \mathcal{B}$, et donc $\mathbf{r} \in \mathsf{Inner}_\circ(\mathcal{A}) \vee \mathbf{r} \in \mathsf{Inner}_\circ(\mathcal{B})$, c'est à dire $\mathbf{r} \in \mathsf{Inner}_\circ(\mathcal{A}) \cup \mathsf{Inner}_\circ(\mathcal{B})$;

Distributivité par rapport à l'intersection. La preuve est identique au cas de l'union.

■

6.1.1 Opérateurs de contraction intérieurs

À partir de la définition d'opérateur d'approximation intérieure, on peut introduire une notion « duale » de celle d'opérateur de contraction (sous-entendu, extérieur) telle que donnée par la définition 3.3, page 38 :

Définition 6.2 (Opérateur de contraction intérieur (OCI)). Soit c une contrainte réelle n-aire. Un *opérateur de contraction intérieur* pour c est une fonction $\mathsf{IC}_c : \mathbb{I}_\circ^n \to \mathcal{P}(\mathbb{I}_\circ^n)$ vérifiant :

$$\forall \boldsymbol{B} \in \mathbb{I}_\circ^n : \bigcup \mathsf{IC}_c(\boldsymbol{B}) \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \rho_c) \tag{6.1}$$

où $\bigcup \mathsf{IC}_c(\boldsymbol{B})$ correspond au sous-ensemble de $\mathbb{R}^n$ défini par l'union de tous les pavés de l'ensemble $\mathsf{IC}_c(\boldsymbol{B})$. Dans la suite, on omettra le symbole $\bigcup$ pour alléger les notations.

Note : Afin de rester compatible avec les autres parties de ce document, nous identifierons dans la suite « opérateur de contraction » (sans précision sur le fait qu'il s'agisse d'un opérateur intérieur ou extérieur) avec « opérateur de contraction extérieur ».

On peut ainsi associer à chaque contrainte un opérateur de contraction intérieur calculant à partir d'un pavé initial $\boldsymbol{B}$ un ensemble de pavés inclus dans l'intersection de la relation ρ sous-jacente et de $\boldsymbol{B}$. Contrairement à un CNO (déf. 3.3), un OCI ne garantit pas la complétude (tous les éléments de l'intersection $\rho \cap \boldsymbol{B}$ ne pouvant être inclus dans un pavé n'appartiennent pas à $\mathsf{Inner}_\circ(\boldsymbol{B} \cap \rho_c)$). On a par contre une garantie de correction :

Proposition 6.2 (Correction de IC). *Étant donnés une contrainte réelle c et un opérateur de contraction intérieur IC_c pour c, on a la relation :*

$$\forall \boldsymbol{B} \in \mathbb{I}_\circ^n : \mathsf{IC}_c(\boldsymbol{B}) \subseteq (\boldsymbol{B} \cap \rho_c)$$

Démonstration. Conséquence immédiate des définitions de Inner et de Hull. ■

Pour une contrainte réelle c, un opérateur de contraction intérieur IC_c pour c, un opérateur de contraction extérieur OC_c pour c et un pavé $\boldsymbol{B}$, on a les relations suivantes, dérivant de la définition 3.3 et de la proposition 6.2 :

$$\mathsf{IC}_c(\boldsymbol{B}) \subseteq (\boldsymbol{B} \cap \rho_c) \subseteq \mathsf{OC}_c(\boldsymbol{B}) \tag{6.2}$$

Il est possible de définir des opérateurs de contraction intérieurs avec une propriété plus forte que celle énoncée par la relation (6.2). De tels opérateurs sont dits *optimaux* au sens ci-dessous :

Définition 6.3 (Opérateur de contraction intérieur optimal). Soit c une contrainte n-aire et IC_c un opérateur de contraction intérieur pour c. L'opérateur IC_c est dit *optimal* si et seulement si la relation suivante est vérifiée pour tout pavé $\boldsymbol{B}$:

$$\mathsf{IC}_c(\boldsymbol{B}) = \mathsf{Inner}_\circ(\boldsymbol{B} \cap \rho_c) \tag{6.3}$$

On peut remarquer que la définition d'opérateurs de contraction intérieurs à partir de l'arithmétique des intervalles n'est pas aussi facile que celle d'opérateurs de contraction extérieurs (si l'on n'envisage pas l'utilisation d'une arithmétique « non-standard » telle que l'arithmétique de KAUCHER ou celle de MARKOV). En effet, les consistances que l'on a définies au chapitre 3 ne sont que partielles. Ainsi, on a l'assurance que les valeurs rejetées ne sont pas solutions des contraintes ; par contre, on n'a pas en général d'information sur les valeurs qui ont été gardées. Nous allons montrer dans la suite qu'il est possible d'utiliser les opérateurs de contraction extérieurs pour calculer des approximations intérieures à condition de les appliquer sur la négation des contraintes considérées.

Avant cela, nous allons présenter la méthode d'évaluation/découpage développée par JARDILLIER et LANGUÉNOU pour résoudre les systèmes de contraintes avec variable universellement quantifiée apparaissant dans le cadre du problème du cameraman virtuel.

6.2 Résolution de contraintes avec quantificateurs universels

Dans le reste de ce chapitre, nous ne considérerons que le cas de contraintes où n'apparaît qu'une seule variable universellement quantifiée. On a vu plus haut que cela correspond en pratique au type de contraintes obtenues lors de la résolution du problème du cameraman virtuel. Considérons une telle contrainte :

$$\forall v \in I_v : c(x_1, \ldots, x_n, v) \tag{6.4}$$

En pratique, les pavés solutions obtenus devront être inclus dans l'approximation intérieure de $\rho_c \cap I_1 \times \cdots \times I_n \times I_v$ pour que l'on puisse prendre des valeurs arbitraires dans les domaines des variables x_1,...,x_n telles que la contrainte c est vérifiée. Ainsi, résoudre la contrainte (6.4) correspond au problème : étant donné un intervalle I_v, trouver les intervalles $I'_1, \ldots, I'_n$ tels que la relation suivante est vérifiée :

$$\forall x_1 \ldots \forall x_n \forall v : v \in I_v \wedge x_1 \in I'_1 \wedge \cdots \wedge x_n \in I'_n \Rightarrow c(x_1, \ldots, x_n, v)$$

Étant donnés une contrainte réelle $(n+1)$-aire $c(x_1, \ldots, x_n, x_v)$ et un pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n \times I_v$, l'application d'un opérateur de contraction intérieur IC_c sur $\boldsymbol{B}$ nous fournit un ensemble de pavés $U = \{\boldsymbol{B'_1}, \ldots, \boldsymbol{B'_p}\}$ où chaque pavé $\boldsymbol{B'_j} = D_1 \times \cdots \times D_n \times D_v$ est un sous-pavé de $\boldsymbol{B}$ vérifiant : $\forall \mathsf{r}_1 \in D_1, \ldots, \forall \mathsf{r}_n \in D_n, \forall \mathsf{r}_v \in D_v : c(\mathsf{r}_1, \ldots, \mathsf{r}_n, \mathsf{r}_v)$.

Par conséquent, on peut envisager la résolution d'une contrainte de la forme $\forall v \in I_v \colon c(x_1, \ldots, x_n, v)$ comme une opération de filtrage où l'on ne retient de U que les pavés $\boldsymbol{B'} = D_1 \times \cdots \times D_n \times D_v$ vérifiant $D_v = I_v$.

Notation Dans la suite, étant données une contrainte c et une variable v apparaissant dans c, on notera $\tilde{\rho}_{c,v,I_v}$ la relation associée à la contrainte $\forall v \in I_v \colon c$. Afin d'alléger les notations, on se permettra de remplacer $\tilde{\rho}_{c,v,I_v}$ par $\tilde{\rho}_c$ lorsqu'il n'y aura pas d'ambiguïté sur la variable et son domaine.

6.2.1 Résolution par évaluation/découpage

Nous allons décrire dans cette section la méthode utilisée par JARDILLIER et LANGUÉNOU [120] pour gérer des contraintes de la forme $\forall v \in I_v \colon c_1 \wedge \cdots \wedge c_m$.

Partant d'un pavé $\boldsymbol{B}$ de domaines, JARDILLIER et LANGUÉNOU calculent une approximation intérieure de la relation associée à la contrainte $\forall v \in I_v \colon c_1 \wedge \cdots \wedge c_m$ en décomposant le domaine initial I_v de la variable v en intervalles canoniques I_v^1,...,I_v^p (avec $I_v^1 \cup \cdots \cup I_v^p = I_v$) et en testant si la contrainte $c_1 \wedge \cdots \wedge c_m$ est vérifiée pour chacun des pavés $I_1 \times \cdots \times I_n \times I_v^1$,..., $I_1 \times \cdots \times I_n \times I_v^p$. Ces évaluations retournent un résultat à interpréter dans une logique trivaluée (*vrai, faux, inconnu*). Les pavés étiquetés *vrai* ne contiennent que des solutions, ceux étiquetés *faux* ne contiennent aucune solution ; quant à ceux étiquetés *inconnu*, ils sont découpés récursivement jusqu'à ce que les sous-pavés résultants puissent être classés en *vrai* ou en *faux*, ou bien qu'ils ne puissent plus être découpés (pavés canoniques). Les pavés conservés au final sont ceux vérifiant :

$$\forall j \in \{1, \ldots, p\} \colon \mathsf{eval}_{\{c_1 \wedge \cdots \wedge c_m\}}(I_1 \times \cdots \times I_n \times I_v^j) = \textit{vrai} \tag{6.5}$$

Dans ce document, l'algorithme de JARDILLIER et LANGUÉNOU est noté EIA4 (alg. 6.1). On peut rapprocher le processus initié par EIA4 des travaux de SAM-HAROUD et FALTINGS [203, 202] où les boîtes étiquetées *vrai*, *faux* et *inconnu* sont organisées en 2^k-arbres afin de faciliter le calcul de consistances globales.

L'algorithme EIA4 est présenté ici dans sa version originale, c'est-à-dire pour des intervalles fermés uniquement. Nous donnons ci-dessous une preuve de sa correction non présente dans l'article original.

Proposition 6.3 (Correction de EIA4). *Soit v une variable, $\boldsymbol{B}$ un pavé, $\mathcal{S} = \{c_1, \ldots, c_m\}$ un ensemble de contraintes et I_v le domaine de v dans le pavé $\boldsymbol{B}$. Alors, l'algorithme* EIA4 *implémente un opérateur de contraction intérieur pour la contrainte c: $\forall v \in I_v \colon (c_1 \wedge \cdots \wedge c_m)$, c'est-à-dire :*

$$\mathsf{EIA4}(\mathcal{S}, \boldsymbol{B}, v, \inf(I_v)) \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \rho_c)$$

Démonstration. Indexons tous les identificateurs apparaissant dans l'algorithme EIA4 par la profondeur de récursion, avec 0 l'indice de l'entrée initiale. Soit $\underline{\mathsf{g}} = \inf(I_v)$ et $\overline{\mathsf{g}} = \sup(I_v)$. Nous allons prouver par contradiction que si $\mathsf{sat_j}$ est *vrai*, on a $\boldsymbol{B_j}|_{v,[\underline{\mathsf{g}}..\mathsf{g}_j^+]} \subseteq \mathsf{Inner}_\circ(\boldsymbol{B_0} \cap \rho_c)$. Ceci suffit à prouver la correction d'EIA4 puisque les seuls pavés conservés sont ceux tels que $\mathsf{sat_j}$ est vrai et $\mathsf{g}_j^+ = \overline{\mathsf{g}}$.

Supposons que $\boldsymbol{B_j}|_{v,[\underline{\mathsf{g}}..\mathsf{g}_j^+]} \not\subseteq \mathsf{Inner}_\circ(\boldsymbol{B_0} \cap \rho_c)$. Alors, il existe une étape i, avec $i < j$, telle que $\boldsymbol{B_i}|_{v,[\mathsf{g}_i..\mathsf{g}_i^+]} \not\subseteq \mathsf{Inner}_\circ(\boldsymbol{B_0} \cap \rho_c)$ et $\boldsymbol{B_j} \subseteq \boldsymbol{B_i}$. Cependant, $\mathsf{sat_i}$ aurait alors été faux, et donc EIA4 n'aurait pas poursuivi la récursion, ce qui contredit le fait que l'étape j a été atteinte et termine la preuve. ■

L'équation (6.5) implique que chaque pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n \times I_v$ retenu est inclus dans l'approximation intérieure de ρ_c. En conséquence, la propriété vérifiée par chacun d'entre eux est :

$$\forall x_1 \ldots \forall x_n \forall v \colon (v \in I_v \wedge x_1 \in I_1 \wedge \cdots \wedge x_n \in I_n \Rightarrow c_1 \wedge \cdots \wedge c_m)$$

6.2.2 Calcul d'approximation intérieure par négation

Nous allons maintenant présenter dans cette section des algorithmes résolvant des contraintes réelles avec une variable universellement quantifiée en raisonnant sur leur négation. Ces algorithmes implémentent des opérateurs de contraction intérieurs pour chaque contrainte c en utilisant l'opérateur de contraction extérieur $\mathsf{OC}_{\overline{c}}$, où $\overline{c}$ est la négation de c (*i.e.* on a : $\rho_{\overline{c}} = \mathbb{R}^n \setminus \rho_c$). Comme les valeurs rejetées par l'opérateur sont obligatoirement non solution de $\overline{c}$ — par complétude de OC (*cf.* déf. 3.3) — elles sont des solutions garanties pour la contrainte c.

Plus formellement, une contrainte de la forme $\forall x_v \in I_v : c(x_1, \ldots, x_n, x_v)$ peut être remplacée par $\neg\exists x_v : x_v \in I_v \wedge \neg c(x_1, \ldots, x_n, x_v)$ où la contrainte $\exists x_v : x_v \in I_v \wedge \neg c(x_1, \ldots, x_n, x_v)$ peut être résolue par l'opérateur OC. Plus généralement, un système de contraintes de la forme $\forall x_v \in I_v^1 : c_1(x_1, \ldots, x_n, x_v) \wedge \cdots \wedge \forall x_v \in I_v^m : c_m(x_1, \ldots, x_n, x_v)$ peut être remplacé par le système $[\neg\exists x_v : x_v \in I_v^1 \wedge \neg c_1(x_1, \ldots, x_n, x_v)] \wedge \cdots \wedge [\neg\exists x_v : x_v \in I_v^m \wedge \neg c_m(x_1, \ldots, x_n, x_v)]$ où les conjonctions de niveau le plus élevé sont préservées.

ALG. 6.1: EIA4 – Algorithme d'évaluation/découpage pour la contrainte $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v) : (c_1 \wedge \cdots \wedge c_m)$

```
1  EIA4(in: S = {c_1, ..., c_m}, B ∈ I_□^n, v ∈ V_ℝ, g ∈ 𝔽;  out: U ∈ P(I_□^n))
2  début
4      B' ← B|_{v,[g..g+]}
5      sat ← GlobSat(c_1, B') ∧ ··· ∧ GlobSat(c_m, B')
6      suivant (sat) dans
7          vrai:
8              si (g+ = sup(Dom_B(v))) alors
9                  retourner ({B})
10             sinon
11                 retourner (EIA4(S, B, v, g+))
12             finsi
13         faux:
14             retourner (∅)
15         inconnu:
16             si (Canonical_v(B)) alors
17                 retourner (∅)
18             sinon
19                 (D, D') ← Split_v(B)
20                 retourner (EIA4(S, D, v, g) ∪ EIA4(S, D', v, g))
21             finsi
22     finsuivant
23 fin
```

La fonction Split_v découpe en deux intervalles l'un des domaines non canoniques de $\boldsymbol{D}$. Étant donnés une contrainte n-aire c et un pavé $\boldsymbol{B}$, GlobSat(c, $\boldsymbol{B}$) vérifie que la contrainte c est vérifiée pour chaque valeur $\boldsymbol{r}$ de $\boldsymbol{B}$. L'opérateur $\mathsf{Canonical}_v(\boldsymbol{B})$ teste si le pavé $\boldsymbol{B}$ est canonique en toutes ses dimensions sauf celle du domaine de la variable v.

Note : La méthode décrite ci-dessus nous interdit de définir des opérateurs de contraction intérieurs optimaux. En effet, il suffit de considérer par exemple le cas d'une contrainte n-aire c et d'un pavé canonique clos $\boldsymbol{B}_\square$ tel que ses sommets g_i ($i \in \{1, \ldots, 8\}$) sont dans $\rho_{\overline{c}}$ alors que $\mathring{\boldsymbol{B}} \subseteq \rho_c$ (où $\mathring{\boldsymbol{B}}$ est le plus grand connexe ouvert contenu dans $\boldsymbol{B}_\square$). Du fait de sa propriété de complétude, l'opérateur $\mathsf{OC}_{\overline{c}}$ appliqué au pavé $\boldsymbol{B}_\square$ doit le retourner intact. Par conséquent, tous les éléments de $\mathring{\boldsymbol{B}}$ ne seront pas retenus par l'opérateur de contraction intérieur alors qu'ils appartiennent à l'approximation intérieure de ρ_c.

Dans la suite, nous commencerons par présenter deux algorithmes utilisant des opérateurs OC basés sur la hull-consistance : le premier résout des contraintes de la forme $\forall v \in I_v : c$ et le second des contraintes de la forme $\forall v \in I_v (c_1 \wedge \cdots \wedge c_m)$. Les bonnes propriétés de l'approximation Hull faciliteront les preuves de correction des algorithmes. Nous introduirons ensuite trois algorithmes utilisant la box-consistance afin d'accélérer les calculs, puis nous comparerons ces derniers algorithmes à EIA4.

6.2.2.1 Utilisation de la hull-consistance

On suppose dans la suite que l'on est capable de calculer le plus petit pavé englobant une relation réelle quelconque. L'algorithme ICA3 (alg. 6.2) implémente un opérateur de contraction intérieur pour une contrainte avec une variable universellement quantifiée.

ALG. 6.2: $\mathsf{ICA3}_c$ – Algorithme de résolution de $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v)\colon c$

```
1  ICA3_c(in: B ∈ 𝕀ₙ°, v ∈ 𝒱_ℝ; out: 𝒲 ∈ 𝒫(𝕀ₙ°))
2  début
3        D ← OC_c̄(B)
5        𝒲 ← B \ D|_{v,B}
6        si (D ≠ ∅ et ¬Canonical_v(D)) alors
7              (D₁, D₂) ← Split_v(D|_{v,B})
8              𝒲 ← 𝒲 ∪ ICA3_c(D₁, v) ∪ ICA3_c(D₂, v)
9        finsi
10       retourner (𝒲)
11 fin
```

La fonction Split_v découpe en deux intervalles l'un des domaines non canonique de $\boldsymbol{D}$. Le domaine de v $\mathsf{Dom}_{\boldsymbol{D}}(v)$ n'est jamais considéré pour le découpage. De la même façon, $\mathsf{Canonical}_v$ teste la canonicité pour toutes les dimensions du pavé passé en paramètre, sauf pour celle correspondant au domaine de la variable v.

Proposition 6.4 (Correction de ICA3). *Soit c une contrainte réelle, v une variable, $\boldsymbol{B} = I_1 \times \cdots \times I_n$ un pavé, et I_v le domaine de v dans $\boldsymbol{B}$. Alors l'algorithme* ICA3 *implémente un opérateur de contraction intérieur pour la contrainte $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v)\colon c$.*

Démonstration. Indexons tous les identificateurs apparaissant dans ICA3 par la profondeur de récursion, avec 0 l'indice pour l'entrée initiale. Nous allons prouver que $\mathsf{ICA3}_c(\boldsymbol{B}, v) \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_c)$.

Commençons par noter qu'à chaque étape i, on a $\boldsymbol{B_i} \subseteq \boldsymbol{B_0}$ car $\boldsymbol{B_i}$ est obtenu en découpant $\boldsymbol{D_{i-1}}$ en deux, où $\boldsymbol{D_{i-1}} \subseteq \boldsymbol{B_{i-1}}$ (par contractance de OC). Prouver la correction revient alors à montrer que pour chaque ensemble calculé $\mathcal{W}_i$, on a :

$$\mathcal{W}_i \subseteq \mathsf{Inner}_\circ(\boldsymbol{B_0} \cap \tilde{\rho}_c) \tag{6.6}$$

puisque le résultat final de ICA3 est l'union des $\mathcal{W}_i$. Pour cela, nous allons montrer ci-dessous qu'à chaque étape i,on a $\mathcal{W}_i \subseteq \mathsf{Inner}_\circ(\boldsymbol{B_i} \cap \tilde{\rho}_c)$. La monotonie et la distributivité de Inner par rapport à l'intersection nous permettront alors d'en déduire l'inclusion (6.6).

À chaque étape i, nous avons : $\boldsymbol{D_i}|_{v,\boldsymbol{B_i}} = \{(\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \mathbb{R}^n \mid \exists \mathsf{s}_v \in I_v\colon (\mathsf{r}_1, \ldots, \mathsf{s}_v, \ldots, \mathsf{r}_n) \in \boldsymbol{D_i}\}$. Par conséquent, $\boldsymbol{B_i} \setminus \boldsymbol{D_i}|_{v,\boldsymbol{B_i}}$ est l'ensemble des éléments définis par $\boldsymbol{B_i} \setminus \boldsymbol{D_i}|_{v,\boldsymbol{B_i}} = \{\mathbf{r} = (\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \mathbb{R}^n \mid \mathbf{r} \in \boldsymbol{B_i} \wedge \forall \mathsf{s}_v \in I_v\colon (\mathsf{r}_1, \ldots, \mathsf{s}_v, \ldots, \mathsf{r}_n) \notin \boldsymbol{D_i}\}$.

Par complétude de OC chaque élément ne se trouvant pas dans $\boldsymbol{D_i}$ n'est pas dans $\boldsymbol{B_i} \cap \rho_{\overline{c}}$. Par conséquent, un élément $\mathbf{r} = (\mathsf{r}_1, \ldots, \mathsf{r}_n)$ de $\boldsymbol{B_i} \setminus \boldsymbol{D_i}|_{v,\boldsymbol{B_i}}$ est tel que : $\forall \mathsf{s}_v \in I_v\colon (\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \boldsymbol{B_i} \cap \rho_c$, et donc $\mathbf{r} \in \boldsymbol{B_i} \cap \tilde{\rho}_c$. Par ailleurs, nous avons aussi $\mathsf{Hull}_\circ(\{\mathbf{r}\}) \subseteq \boldsymbol{B_i} \cap \tilde{\rho}_c$ puisque $\boldsymbol{B_i} \setminus \boldsymbol{D_i}|_{v,\boldsymbol{B_i}}$ est une différence de pavés. Il vient alors $\mathbf{r} \in \mathsf{Inner}_\circ(\boldsymbol{B_i} \cap \tilde{\rho}_c)$, et finalement $\mathcal{W}_i \subseteq \mathsf{Inner}_\circ(\boldsymbol{B_i} \cap \tilde{\rho}_c)$. ■

Note : On remarquera que la ligne 7 dans l'algorithme 6.2 peut être avantageusement remplacée par :

$$(\boldsymbol{D_1}, \boldsymbol{D_2}) \leftarrow \mathsf{Split}_v(\boldsymbol{D})$$

à condition que le domaine initial I_v^0 de la variable v soit passé en paramètre à ICA3 ; la ligne 5 deviendrait alors :

$$\mathcal{W} \leftarrow \boldsymbol{B} \setminus \boldsymbol{D}|_{v,I_v^0}$$

La gestion de systèmes de contraintes est faite par l'algorithme ICA4 (*cf.* alg. 6.3) de la façon suivante : la satisfiabilité de chaque contrainte du système par rapport aux domaines vérifiant les contraintes précédentes est considérée tour à tour. En particulier, il est important de noter que, contrairement à l'algorithme Nar, p. 39, *chaque contrainte n'est prise en compte qu'une seule fois* (pas de calcul de point-fixe). En effet, après avoir considéré une contrainte c, le produit cartésien des domaines des variables ne contient que des points solution de c. Par conséquent, une réduction de ces domaines par la suite n'invalide nullement la satisfiabilité de c.

ALG. 6.3: ICA4 – Algorithme de résolution de $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v)\colon (c_1 \wedge \cdots \wedge c_m)$

```
1  ICA4(in: S = {c_1, ..., c_m}, A ∈ P(I_o^n), v ∈ V_R; out: W ∈ P(I_o^n))
2  début
3      si (S ≠ ∅) alors
4          B ← ∅
5          pour chaque D ∈ A faire
6              B ← B ∪ ICA3_{c_1}(D, v)
7          fpc
8          retourner (ICA4(S \ {c_1}, B, v))
9      sinon
10         retourner (A)
11     finsi
12 fin
```

Proposition 6.5 (Correction de ICA4). *Soit v une variable, $\mathcal{S} = \{c_1, \ldots, c_m\}$ un ensemble de contraintes, $\boldsymbol{B}$ un pavé, et I_v le domaine de v dans $\boldsymbol{B}$. On a la relation :*

$$\mathsf{ICA4}(\mathcal{S}, \{\boldsymbol{B}\}, v) \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_m) \tag{6.7}$$

avec $\forall \boldsymbol{D} \in \mathsf{ICA4}(\mathcal{S}, \{\boldsymbol{B}\}, v)\colon \boldsymbol{D}|_v = I_v$.

Démonstration. Indexons les identificateurs apparaissant dans ICA4 par l'indice dans $\mathcal{S}$ de la contrainte considérée à l'étape courante.

Nous allons prouver la relation (6.7) par induction. Soit $\mathcal{B}_i$ l'ensemble calculé lorsque $\mathcal{S} = \{c_i, \ldots, c_m\}$. Nous cherchons à montrer :

$$\forall i \in \{1, \ldots, m\}\colon \mathcal{B}_i \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_i) \tag{6.8}$$

Cela suffit à prouver l'équation (6.7) car, pour $i = m + 1$, l'algorithme ICA4 retourne son entrée, à savoir ici $\boldsymbol{B}_m \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_m)$.

La relation (6.8) est vraie pour $i = 1$ du fait de la correction de ICA3 (prop. 6.4) puisque $\mathcal{A} = \{\boldsymbol{B}\}$.

Supposons que la relation (6.8) est vraie pour toutes les étapes $i \in \{1, \ldots, j-1\}, j \leqslant m$. Il s'ensuit que la relation suivante est vérifiée : $\mathcal{B}_{j-1} \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_{j-1})$

Soit $\mathcal{A}_j = \{\boldsymbol{D_1}, \ldots, \boldsymbol{D_p}\} = \mathcal{B}_{j-1}$ l'entrée de l'algorithme quand $\mathcal{S} = \{c_j, \ldots, c_m\}$. L'ensemble $\mathcal{B}_j$ tel qu'il est calculé par la boucle « **pour chaque** » vaut : $\mathcal{B}_j = \boldsymbol{D'_1} \cup \cdots \cup \boldsymbol{D'_p}$, avec $\forall k \in \{1, \ldots, p\}\colon \boldsymbol{D'_k} \subseteq \mathsf{Inner}_\circ(\boldsymbol{D_k} \cap \tilde{\rho}_j)$. Par conséquent

$$\mathcal{B}_j \subseteq \mathsf{Inner}_\circ(\boldsymbol{D_1} \cap \tilde{\rho}_j) \cup \cdots \cup \mathsf{Inner}_\circ(\boldsymbol{D_p} \cap \tilde{\rho}_j) \tag{6.9}$$

La distributivité de Inner par rapport à l'union nous autorise à réécrire l'équation (6.9) en :

$$\mathcal{B}_j \subseteq \mathsf{Inner}_\circ((\boldsymbol{D_1} \cup \cdots \cup \boldsymbol{D_p}) \cap \tilde{\rho}_j) \tag{6.10}$$

Comme $\mathcal{B}_{j-1} = \{\boldsymbol{D_1}, \ldots, \boldsymbol{D_p}\}$ est inclus dans $\mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_{j-1})$, l'équation (6.10) devient : $\mathcal{B}_j \subseteq \mathsf{Inner}_\circ(\mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_{j-1}) \cap \tilde{\rho}_j)$

Par idempotence et distributivité de Inner par rapport à l'intersection, nous obtenons finalement $\mathcal{B}_j \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_j)$, ce qui termine la preuve. ■

En dépit de ses propriétés fortes, l'utilisation de la hull-consistance est insuffisante à assurer la complétude des algorithmes présentés ci-avant. Par conséquent, rien ne nous empêche d'utiliser une consistance moins forte et plus rapide à calculer afin d'accélérer la résolution. Dans la suite, nous allons exprimer les algorithmes ICA3 et ICA4 en ayant recours à la box-consistance plutôt qu'à la hull-consistance. Nous présenterons aussi un algorithme généralisant ICA4 pour la résolution de contraintes de la forme : $(\forall v \in I^1\colon c_1) \wedge \cdots \wedge (\forall v \in I^m\colon c_m)$

6.2.2.2 Utilisation de la box-consistance

À partir de l'algorithme ICA3, nous dérivons l'algorithme ICAb3 utilisant la box-consistance, ce qui nous permet de gérer des contraintes complexes sans avoir à les décomposer.

ALG. 6.4: $\mathsf{ICAb3}_c$ – Algorithm de résolution de la contrainte $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v)\colon c$

```
1  ICAb3_c(in: B ∈ 𝕀°_n, v ∈ 𝒱_ℝ; out: 𝒲 ∈ 𝒫(𝕀°_n))
2  début
3      B' ← OCb_c(B)
4      si (Dom_B'(v) = Dom_B(v)) alors
5          D ← OCb_c̄(B')
6          𝒲 ← B' \ D|_{v,B'}
7          si (D ≠ ∅ et ¬Canonical_v(D)) alors
8              (D_1, D_2) ← Split_v(D|_{v,B'})
9              𝒲 ← 𝒲 ∪ ICAb3_c(D_1, v) ∪ ICAb3_c(D_2, v)
10         finsi
11         retourner (𝒲)
12     sinon
13         retourner (∅)
14 fin
```

La fonction Split_v découpe en deux intervalles un des domaines non canonique de $\boldsymbol{D}$. Le domaine $\mathsf{Dom}_{\boldsymbol{D}}(v)$ n'est jamais considéré pour le découpage. De la même façon, $\mathsf{Canonical}_v$ teste la canonicité pour tous les domaines sauf pour celui de v.

Proposition 6.6 (Correction de ICAb3). *Soient c une contrainte, ρ_c la relation associée, v une variable et $\boldsymbol{B}$ un pavé. L'algorithme* ICAb3 *implémente un opérateur de contraction intérieur pour la contrainte* $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v) : c$.

Démonstration. Par complétude de OCb, l'application de OCb_c sur le pavé $\boldsymbol{B}$ n'élimine aucune des valeurs de $\boldsymbol{B}$ incluses dans $\mathsf{Inner}_\circ(\boldsymbol{B} \cap \rho_c)$. De plus, comme nous souhaitons que les pavés obtenus au final dans l'ensemble $\mathcal{W}$ vérifient $\forall \boldsymbol{D} \in \mathcal{W}\colon \boldsymbol{D}|_k = \boldsymbol{B}|_k$, le fait de contracter le pavé d'entrée et de tester que le domaine de v n'a pas été réduit ne peut qu'accélérer les calculs sans influer sur la correction du résultat. Ainsi, la correction de ICAb3 découle de celle de ICA3 puisque $\forall \boldsymbol{B}\colon \mathsf{OCb}_c(\boldsymbol{B}) \supseteq \mathsf{OC}_c(\boldsymbol{B})$, et donc $\boldsymbol{B'} \setminus \mathsf{OCb}_{\overline{c}}(\boldsymbol{B'})|_{v,\boldsymbol{B'}} \subseteq \boldsymbol{B'} \setminus \mathsf{OC}_{\overline{c}}(\boldsymbol{B'})|_{v,\boldsymbol{B'}}$. ■

6.2.2.3 Comparaison des approches évaluation/découpage *vs.* négation

Soit ICAb4 l'algorithme ICA4 où l'appel à IC3 a été remplacé par un appel à ICAb3. L'algorithme ICAb4 peut facilement être prouvé correct (*i.e.* $\mathsf{ICAb4}(\mathcal{S}, \{\boldsymbol{B}\}, v) \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_m)$) en suivant le même raisonnement que pour ICA4. De plus, on a le résultat suivant :

Proposition 6.7 (relation entre EIA4 et ICAb4). *Soit $\mathcal{S} = \{c_1 \cap \cdots \cap c_m\}$ un ensemble de contraintes, v une variable, $\boldsymbol{B}$ un pavé, I_v, le domaine de v dans $\boldsymbol{B}$ et* $\mathsf{g} = \inf(I_v)$. *On a la relation :*

$$\mathsf{EIA4}(\mathcal{S}, \boldsymbol{B}, v, \mathsf{g}) \subseteq \mathsf{ICAb4}(\mathcal{S}, \{\boldsymbol{B}\}, v) \tag{6.11}$$

Démonstration. On va prouver la relation (6.11) en montrant que tout élément de $\mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_m)$ qui n'est pas retenu par ICAb4 ne peut être retenu par EIA4. La preuve est basée sur la définition de la box-consistance. Soit $\mathsf{Hull}_\square(I_v) = [\mathsf{g} \,..\, \mathsf{h}]$. Considérons $\mathbf{r} = (\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \mathbb{R}^n$ avec $\mathbf{r} \in \mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_m)$ et $\mathbf{r} \notin \mathsf{ICAb4}(\mathcal{S}, \{\boldsymbol{B}\}, v)$. Alors, il doit exister une constraint c_α telle que $\mathbf{r}$ n'est pas inclus dans l'ensemble retourné par $\mathsf{ICAb3}_{c_\alpha}$, c'est à dire que $\mathbf{r}$ a été retenu dans un pavé $\boldsymbol{D_i}|_{v,\boldsymbol{B'_i}}$ canonique suivant toutes ses dimensions sauf celle du domaine de v ; par conséquent, $\mathsf{Outer}_\square(\boldsymbol{D_i}|_{v,\boldsymbol{B'_i}}) = J_1 \times \cdots \times I_v \times \cdots \times J_n$ avec $\forall J_j\colon \mathsf{Outer}_\square(J_j) =$

$\mathsf{Outer}_\square(\{\mathsf{r}_j\})$. Comme $\boldsymbol{D_i} = \mathsf{OCb}_{\overline{c_\alpha}}(\boldsymbol{B'_i})$, la définition de OCb suppose qu'il existe deux intervalles canoniques J_v^a et J_v^b tels que $\overline{C_\alpha}(J_1, \ldots, J_v^a, \ldots, J_n)$ et $\overline{C_\alpha}(J_1, \ldots, J_v^b, \ldots, J_n)$ sont vérifiés. Par conséquent, étant donné $\boldsymbol{J^a}_\square = \mathsf{Outer}_\square(J_1) \times \cdots \times \mathsf{Outer}_\square(J_v^a) \times \cdots \times \mathsf{Outer}_\square(J_n)$ et $\boldsymbol{J^b}_\square = \mathsf{Outer}_\square(J_1) \times \cdots \times \mathsf{Outer}_\square(J_v^b) \times \cdots \times \mathsf{Outer}_\square(J_n)$, nous avons $\overline{C_\alpha}(\boldsymbol{J^a}_\square)$ et $\overline{C_\alpha}(\boldsymbol{J^b}_\square)$ aussi. Il s'ensuit que $\mathsf{GlobSat}(c_\alpha, \boldsymbol{J^a}_\square)$ et $\mathsf{GlobSat}(c_\alpha, \boldsymbol{J^b}_\square)$ ne peuvent être vrais. Donc sat est "inconnu" pour les pavés $\boldsymbol{J^a}_\square$ et $\boldsymbol{J^b}_\square$ dans $\mathsf{EIA4}(\mathcal{S}, \boldsymbol{B}, b, \mathrm{g})$. Comme ces pavés sont canoniques, **r** ne sera pas retenu par EIA4 non plus. ■

La proposition 6.7 est importante car elle nous assure que la décomposition du domaine de la variable quantifiée avec la plus grande précision possible n'améliore pas la qualité de l'approximation intérieure calculée.

À partir de l'algorithme ICAb4, il est aisé de définir un algorithme pour résoudre un système de contraintes de la forme :

$$(\forall v \in I^1 : c_1) \wedge \cdots \wedge (\forall v \in I^m : c_m)$$

Soit ICAb5 un tel algorithme (*cf.* alg. 6.5). Comme pour ICAb4 et ICA4, chaque contrainte n'a besoin d'être considérée qu'une seule fois.

ALG. 6.5: ICAb5 – Algorithme de résolution pour la contrainte $\forall v \in I^1 : c_1 \wedge \cdots \wedge \forall v \in I^m : c_m$

```
1  ICAb5(in: S = {(c_1, I^1), ..., (c_m, I^m)}, A ∈ P(I^n_o), v ∈ V_R;  out: W ∈ P(I^n_o))
2  début
3      si (S ≠ ∅) alors
4          B ← ∅
5          pour chaque D ∈ A faire
7              B ← B ∪ ICAb3_{c_1}(D|_{v,I^1}, v,)
9          fpc
10         si (B = ∅) alors
11             retourner (∅)
12         sinon
13             retourner (ICAb5(S \ {(c_1, I^1)}, B, v))
14         finsi
15     sinon
16         retourner (A)
17     finsi
18 fin
```

Proposition 6.8 (Correction de ICAb5). *Soit $\mathcal{S} = \{(c_1, I^1), \ldots, (c_m, I^m)\}$ un ensemble de paires constituées d'une contrainte et d'un domaine. Étant donnés un pavé $\boldsymbol{B}$ et une variable v, on a la relation :*

$$\mathsf{ICAb5}(\mathcal{S}, \{\boldsymbol{B}\}, v) \subseteq \mathsf{Inner}_\circ(\boldsymbol{B} \cap \tilde{\rho}_1 \cap \cdots \cap \tilde{\rho}_m)$$

Soit ξ_j le nombre de flottants dans l'intervalle I_j, et $\xi = \max_j \xi_j$. Pour un système de m contraintes n-aires de la forme $\forall v \in I_v : (c_1 \wedge \cdots \wedge c_m)$, le nombre d'appels à la fonction GlobSat dans EIA4 est dans le pire des cas :

$$\Gamma = m^n \prod_{i=1}^{n} (2\xi_i - 3) = O((m\xi)^n)$$

Dans le pire des cas, le nombre d'appels à l'algorithme OCb dans ICAb4 est aussi en $O((m\xi)^n)$. Cependant, cette évaluation s'avère très pessimiste et ne reflète que très imparfaitement la situation en pratique : comme on le verra au chapitre 12, le filtrage induit par l'algorithme ICAb4 lorsqu'il considère chaque contrainte à son tour réduit énormément le nombre de pavés à considérer dans la suite.

Restriction du cadre général Bien que certains des algorithmes décrits plus haut, ainsi que le cadre général, n'imposent pas de restriction particulière sur le type des contraintes manipulées, nous ne considérons en pratique que des contraintes non-linéaires réelles de la forme $f(x_1, \ldots, x_n) \diamond 0$, avec $\diamond \in \{\leqslant, \geqslant\}$. Quoique restreinte, la classe de ces contraintes couvre nos besoins en ce qui concerne le problème du cameraman virtuel ainsi que bien d'autres applications (pour s'en persuader, on se rappellera que la plupart des méthodes présentées au chapitre précédent ne géraient aussi que des inégalités). De plus, il est facile de déterminer la négation de ce type de contraintes. Dans ce cadre, l'opérateur GlobSat() utilisé dans l'algorithme EIA4 est implémenté par une simple procédure d'évaluation en arithmétique d'intervalles de $f(x_1, \ldots, x_n)$ afin de déterminer si le résultat est supérieur ou égal (resp. inférieur ou égal) à 0. On notera que le problème de la résolution *correcte* de contraintes de la forme $f(x_1, \ldots, x_n) = 0$ ne se pose pas en toute généralité du fait de l'impossibilité potentielle de représenter les solutions réelles avec des intervalles flottants.

QUATRIÈME PARTIE

Environnements de programmation en programmation par contraintes

1. **Introduction**
2. **Langages de programmation par contraintes d'intervalles: état de l'art**
3. **Présentation de DecLIC**
4. **Débogage en PLC : état de l'art**
5. **Débogage par observation du *store***

Introduction

NOUS ALLONS PRÉSENTER DANS CETTE PARTIE les environnements de programmation logique avec contraintes développés durant notre thèse. Nous commencerons par donner un état de l'art des langages de programmation par contraintes d'intervalles; nous replacerons les différents langages actuellement existants dans leur cadre historique, en indiquant pour chacun d'eux ces capacités et ses limites. Nous décrirons ensuite DecLIC, le langage de PLC basé sur clp(fd) [50] intégrant les algorithmes de réduction de domaines pour différentes consistances. Des exemples d'utilisation du langage seront donnés en insistant sur l'impact de la forme des contraintes sur l'efficacité des algorithmes de résolution ; puis l'implémentation du système sera présentée en détail, en exposant en particulier l'extension de la WAM (*Warren Abstract Machine*) rendue nécessaire par les fonctionnalités de DecLIC. Nous nous focaliserons ensuite sur le problème du débogage en programmation par contraintes, en analysant les manques en terme d'environnements dédiés.

Le principe de programmation par contraintes, indépendant de tout langage hôte, permet de résoudre un problème en spécifiant simplement les propriétés que doivent vérifier les solutions. Cette déclarativité, associée à une efficacité comparable — et parfois supérieure — à celle d'algorithmes spécialisés, ont su séduire de nombreux chercheurs qui l'ont introduit dans divers langages, impératifs (REF-ARF [76]), à objets (ILOG Solver [194]), logiques (CHIP [5], Prolog IV [32], clp(fd) [50], clp(BNR) [30]), ou fonctionnels (HOPE [59]).

Cependant, la programmation par contraintes ne s'est pas aussi répandue dans le milieu industriel que son potentiel le lui permettrait. L'absence d'environnement de programmation évolué permettant de déboguer et d'optimiser aisément les programmes n'est sans doute pas étrangère à cette situation. En effet, si l'expressivité de la programmation par contraintes permet d'écrire des programmes concis, clairs et efficaces, elle interdit l'utilisation de débogueurs traditionnels du type pas-à-pas car l'essentiel du travail de résolution de contraintes n'apparaît pas dans le code source et reste caché à l'utilisateur. Comme le soulignent KEIROUTZ *et al.* [133] ainsi que SANNELLA [204] à propos de la conception assistée par ordinateur (*CAD tools*), les outils basés sur une approche de programmation par contraintes se révèlent souvent plus flexibles et plus puissants que les autres, mais sont aussi moins faciles à utiliser car il est difficile de relier un processus de conception particulier à l'ensemble des contraintes obtenues par l'outil de CAO :

> *« People typically develop their own mental models of the plan of how constraints are satisfied by a constraint system. However, these models often do not reflect the true solution plan. »*
>
> — KEIROUZ *et al.* [133]

De plus, s'il est aisé de comprendre le comportement d'une contrainte isolée, plusieurs contraintes peuvent interagir de façon relativement inattendue.

Dans le cadre de la programmation par contraintes elle-même, le principal problème est que le code source d'un programme ne reflète pas le processus de résolution et que le véritable travail est fait dans le *store* à un niveau auquel le programmeur n'a normalement pas accès. Cet état de fait empêche une bonne compréhension de l'exécution du programme et, partant, la possibilité de le déboguer facilement. Considérons par exemple le problème de YOSHIGAHARA décrit ci-dessous et sa résolution par deux méthodes différentes.

Exemple 6.19 (Problème de YOSHIGAHARA [215]). *Trouver les valeurs des variables entières $A\ldots, I$ en utilisant les chiffres de 1 à 9 une seule fois chacun de manière à vérifier la formule :*

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} = 1 \tag{6.12}$$

où BC, EF et HI correspondent aux nombres obtenus en concaténant les valeurs des variables.

Une résolution *naïve* du problème consiste à générer les 9^9 affectations possibles aux variables A–I, suivies par le test que la relation (6.12) est vérifiée. C'est l'approche *generate and test* dont une mise en œuvre en Java est donnée par le programme 6.1.

PROG. 6.1: Le problème de Yoshigahara résolu en Java

```
class yoshigahara {
  private static boolean alldifferent(int a, int b, ...) {
     return (a!=b && a!=c && a!=d && ...
             && f!=i && g!=h && g!=i && h!=i);
  }
  public static void main(String[] args) {
     for (int A=1;A<=9;++A)                                          // Generate
        ...
          for (int I=1;I<=9;++I) {
             if (alldifferent(A,B,C,D,E,F,G,H,I)) {                  // Test
               float res=A/(10.f*B + C) + D/(10.f*E + F)+
                         G/(10.f*H + I);
                if (res == 1.0)
                System.out.println(A+"/"+B+C+" + "+D+"/"+E+F+
                                   " + "+G+"/"+H+I+" = 1");
             }
          }
  }
}
```

Le problème de YOSHIGAHARA ne possède qu'une seule solution entière ($5/34 + 7/68 + 9/12 = 1$) parmi de nombreuses solutions réelles. Cependant, son calcul ne peut se faire en utilisant uniquement des entiers, sauf à réécrire la contrainte différemment. Le programme 6.2 présente la résolution du problème en DecLIC. Pour les problèmes en nombres entiers, on utilise des procédures d'instanciation affectant par *backtracking* à chaque variable chacune des valeurs de son domaine. C'est ce que font les prédicats `labeling/1` et `labelingff/1` hérités de clp(fd). Le prédicat `labelingff/1` apparaissant en ligne 6 du programme 6.2 utilise en plus un ordre particulier de considération des variables, dit *first-fail*, consistant à instancier les variables dans l'ordre croissant de la taille des domaines.

PROG. 6.2: Le problème de YOSHIGAHARA [215] résolu en DecLIC

```
:- L=[A,B,C,D,E,F,G,H,I],
   is_fd(L),
   alldifferent(L),
   domain(L,1,9),
   A/(10*B+C)+D/(10*E+F)+G/(10*H+I) $= 1,
   labelingff(L).                          %  Generate (plus petit domaine d'abord)
```

La résolution du problème en programmation par contraintes consiste à spécifier au départ les relations liant les variables A–I, puis à générer les différentes affectations possibles (approche *test and generate*). De cette façon, de nombreuses affectations ne seront pas considérées. Par exemple, si la pose des contraintes élimine du domaine de A la valeur 6, ce sont tous les 9-uplets de la forme $(6, \_, \_, \_, \_, \_, \_, \_, \_)$ qui sont éliminés de la phase de test.

Le débogage du programme 6.1 se fait aisément en utilisant un débogueur de type pas-à-pas car la totalité des instructions exécutées par la machine pour résoudre le problème apparaît dans le code source. Ce n'est pas le cas du programme 6.2, où tout le véritable travail de résolution est caché au programmeur. Par exemple, si nous utilisons un traceur Prolog pour exécuter le programme 6.2 pas-à-pas de la ligne 4 à la ligne 6, nous ne pourrons que constater les modifications de domaines induites par l'introduction dans le *store* de la contrainte apparaissant à la ligne 5. Tout le travail intermédiaire de propagation de la réduction des domaines nous est inaccessible. En particulier, nous ne disposons d'aucune information en cas d'échec.

En programmation par contraintes, on a deux niveaux indépendants :

1. le niveau des instructions du programme lui-même (débogable classiquement avec des outils de type « pas-à-pas ») ;

2. le niveau du *store* pour lequel les débogueurs habituels sont inutilisables.

Il semble donc nécessaire d'introduire un nouvel outil spécialisé capable de donner accès au *store* sous une forme utilisable par le programmeur. Dans le chapitre 9, nous présenterons certaines solutions adoptées par les quelques débogueurs de langages de programmation par contraintes pour donner un accès (souvent partiel) au *store*. Comme on le verra, la plupart de ces débogueurs se focalisent plutôt sur la présentation des variables et des modifications de leurs domaines que sur les relations entre les contraintes. Enfin, nous décrirons dans le chapitre 10, la technique de débogage définie durant notre thèse, se basant sur une observation directe d'un *store* hiérarchisé ; nous présenterons l'outil de débogage implémenté pour valider nos idées et indiquerons les applications possibles de notre méthode dans le cadre de l'optimisation de la propagation des domaines.

Le reste de cette partie s'organise de la façon suivante : nous donnons un état de l'art des langages de programmation par contraintes d'intervalles dans le chapitre 7 ; nous présentons les possibilités du langage DecLIC ainsi que son implémentation dans le chapitre 8 ; nous exposons ensuite dans le chapitre 9 un état de l'art succinct en ce qui concerne les environnements existants pour le débogage et l'optimisation de programmes en programmation (logique) avec contraintes ; enfin, nous exposons dans le chapitre 10 une nouvelle méthode de débogage : nous décrivons les fondements théoriques de la méthode, puis nous présentons un prototype de débogueur utilisant cette méthode ; nous terminons le chapitre en détaillant son implémentation.

Contributions

- Définition et mise au point de DecLIC [94], un langage pour la programmation logique avec contraintes entières/booléennes/réelles intégrant plusieurs notions de consistances ;
- définition d'une méthode de débogage de programmes avec contraintes par affichage du *store* de contraintes. Implémentation d'un débogueur basé sur la méthode.

CHAPITRE 7

Langages de programmation par contraintes d'intervalles : un état de l'art

NOUS DÉCRIRONS au chapitre 8 le langage de programmation par contraintes d'intervalles DecLIC défini et implémenté durant notre thèse. Dans ce chapitre, nous allons présenter les différents langages de programmation par contraintes d'intervalles existant afin de pouvoir mettre en perspective notre système avec les travaux précédents. Le nombre de ces langages est trop important pour que nous puissions prétendre les décrire tous. Nous nous contenterons donc d'en présenter un échantillon représentatif en privilégiant ceux intégrant des fonctionnalités originales ou ayant marqué une date dans l'histoire de la programmation par contraintes d'intervalles.

Afin d'offrir une *arithmétique logique* respectant la philosophie du langage, CLEARY [48, 47] intègre en 1987 dans Prolog une *arithmétique relationnelle d'intervalles*, créant ainsi le premier langage de résolution de contraintes d'intervalles. Son système possède déjà les principales caractéristiques des langages à venir : un algorithme de propagation de domaines ressemblant à HC3 (*cf.* p. 43) avec une stratégie FIFO pour la reconsidération des contraintes, ainsi que plusieurs méthodes de découpage d'intervalles pour séparer les solutions ou perturber un réseau de contraintes en état de quiescence précoce. Les contraintes de l'utilisateur sont toutes décomposées en primitives (`add/3`, `mult/3`...) comme cela est fait pour HC3. Pour gérer les contraintes non intervalles-convexes induisant des trous dans les domaines des variables, CLEARY suggère d'utiliser systématiquement le mécanisme de *backtracking* de Prolog.

En 1988, reprenant les idées de CLEARY, OLDER et VELLINO mettent au point BNR Prolog [178, 177]. Contrairement à leur prédécesseur, leur système ne gère que les intervalles à bornes fermées. Pour le reste, les fonctionnalités offertes sont identiques à celles du système de CLEARY. Dans les faits, on peut cependant considérer que BNR Prolog est le premier langage logique de résolution de contraintes d'intervalles implémenté, documenté et diffusé.

De façon indépendante, HYVÖNEN [115] définit en 1989 un système de résolution de contraintes d'intervalles (non basé sur Prolog). La principale particularité de son système est que les algorithmes de réduction de domaines associés aux contraintes primitives sont codés explicitement sous une forme fonctionnelle : une contrainte n-aires est manipulée comme un ensemble de n affectations de la forme $x_1 = f_1(x_2, \ldots, x_n), \ldots, x_n = f_n(x_1, \ldots, x_{n-1})$ et non pas comme une relation entre les n variables.

En 1992, BENHAMOU, OLDER et VELLINO généralisent les méthodes de résolution de contraintes par propagation de domaines aux cas de contraintes sur des variables entières et booléennes. Ces idées sont implémentées dans clp(BNR) [12, 176, 30, 31], le successeur de BNR Prolog. Contrairement à BNR Prolog, dont les capacités de résolution de contraintes étaient limitées aux contraintes réelles, clp(BNR) est capable de résoudre des systèmes combinant des contraintes sur des variables entières, booléennes ou réelles. Les méthodes n'ont cependant pas fondamentalement évolué : décomposition des contraintes de l'utilisateur en primitives, calcul de consistances locales et propagation de la variation des domaines dans le réseau de contraintes. On peut noter que, probablement dans un souci d'efficacité, le langage fait une entorse aux principes de Prolog en requérant un typage statique des variables.

Le langage Interlog II a été mis au point par LHOMME dans le cadre de sa thèse de doctorat [151]. Si ses fonctionnalités sont relativement proches de celles offertes par BNR Prolog (résolution de contraintes réelles), les méthodes employées sont, elles, sensiblement différentes. Interlog II utilise en particulier les notions de consis-

tances supérieures (filtrage par k-B-consistance) définies par LHOMME. Afin de pallier la relative inefficacité des méthodes de filtrage sur les contraintes linéaires, il intègre une procédure d'élimination de GAUSS simplifiant les contraintes linéaires avant de les ajouter au réseau global. La gestion de contraintes disjonctives conduisant à des unions d'intervalles (non intervalle-convexité des opérateurs) peut se faire de deux façons : soit en posant des points de choix et en considérant chaque intervalle de l'union séparément (méthode de CLEARY), soit en approchant l'union par un seul intervalle.

L'année 1994 voit apparaître un grand nombre de systèmes de résolution de contraintes d'intervalles. Parmi ceux-ci, on peut distinguer les langages suivants :

- CIAL [45, 128], un système basé sur CLP($\mathcal{R}$) [101] dont la particularité principale est de faire coopérer deux mécanismes de résolution de contraintes : une méthode analogue à celle proposée dans BNR Prolog pour gérer les contraintes non-linéaires, et des méthodes de GAUSS et GAUSS-SEIDEL incrémentales pour les contraintes linéaires ;
- Ilog Solver [194, 195] introduit une approche originale : la *programmation impérative avec contraintes*. Il s'agit d'une librairie C++ extensible possédant la notion de variable logique et de *backtracking*. Les méthodes implémentées pour la résolution de contraintes réelles apparaissent similaires à celles présentes dans clp(BNR). Il est donc aussi possible de traiter de façon uniforme des contraintes sur les réels, les booléens, ou les entiers ;
- Prolog IV [32, 22], un langage de programmation logique permettant de résoudre au sein d'un même système des contraintes sur les arbres, les listes, les réels, les rationnels, les entiers et les booléens. Prolog IV propose de nombreux algorithmes de résolution (algorithme du simplexe, élimination de GAUSS...) en fonction du type des domaines des variables. Les contraintes sur les variables réelles, booléennes et entières peuvent être gérées par des méthodes similaires à celles de clp(BNR). Il y a donc toujours décomposition des contraintes de l'utilisateur en primitives et utilisation de la hull-consistance ;
- enfin, le langage Newton [29, 234], basé sur Prolog, marque un tournant dans l'histoire des langages de résolution de contraintes d'intervalles en étant le premier à ne pas décomposer les contraintes de l'utilisateur en primitives grâce à l'emploi de la box-consistance [29]. Newton se révèle ainsi bien plus efficace que ses prédécesseurs pour résoudre des systèmes complexes non-linéaires. Le langage offre aussi la possibilité de faire de l'optimisation sans ou sous contraintes.

En 1997, Numerica [235], le successeur de Newton (et aussi d'Helios [233], que l'on peut considérer comme une version préliminaire de Numerica), introduit un nouveau standard pour la résolution de contraintes d'intervalles. Numerica est un langage de modélisation (non basé sur Prolog) dont la syntaxe permet de poser des systèmes de contraintes en s'approchant très près de leur formulation mathématique. Il offre la possibilité de résoudre des systèmes de contraintes réelles ainsi que des problèmes d'optimisation avec et sans contraintes. Les méthodes utilisées sont analogues à celles présentes dans Newton. Contrairement à beaucoup des langages présentées plus haut, Numerica est, avec Ilog Solver, Prolog IV et clp(BNR), l'un des rares langages à être sorti des laboratoires pour être utilisé dans un contexte industriel.

Poursuivant dans la voie des langages de modélisation, GRANVILLIERS a défini en 1998 Cosinus, un langage de résolution de contraintes réelles basé sur la box-consistance. Cosinus [95] offre la possibilité de faire coopérer des méthodes numériques (filtrage avec utilisation de l'arithmétique d'intervalles) et symboliques (calcul de bases de GRÖBNER pour accélérer les calculs par génération de contraintes redondantes [28, 27]).

LEBBAH [142] a mis au point en 1999 un langage de résolution de contraintes écrit en C++ utilisant un algorithme générique de consistance forte ($kB(w)$-consistance) et un opérateur de NEWTON sur les intervalles. Son système implémente aussi des techniques d'accélération de la convergence par extrapolation et suppression de cycles d'application d'opérateurs de contraction [143, 144].

Enfin, HICKEY a étendu GNU Prolog [68] pour créer CLIP, un langage de résolution de contraintes d'intervalles implémentant à la fois les méthodes présentes dans clp(BNR) (décomposition de contraintes en primitives et calcul de la hull-consistance) et des opérateurs de contraction plus élaborés (opérateur de Taylor). L'une des particularités du langage est que les algorithmes utilisés sont écrits dans le langage hôte et non dans un langage impératif.

La figure 7.1 présente une vue chronologique partielle de l'histoire des langages de résolution de contraintes d'intervalles.

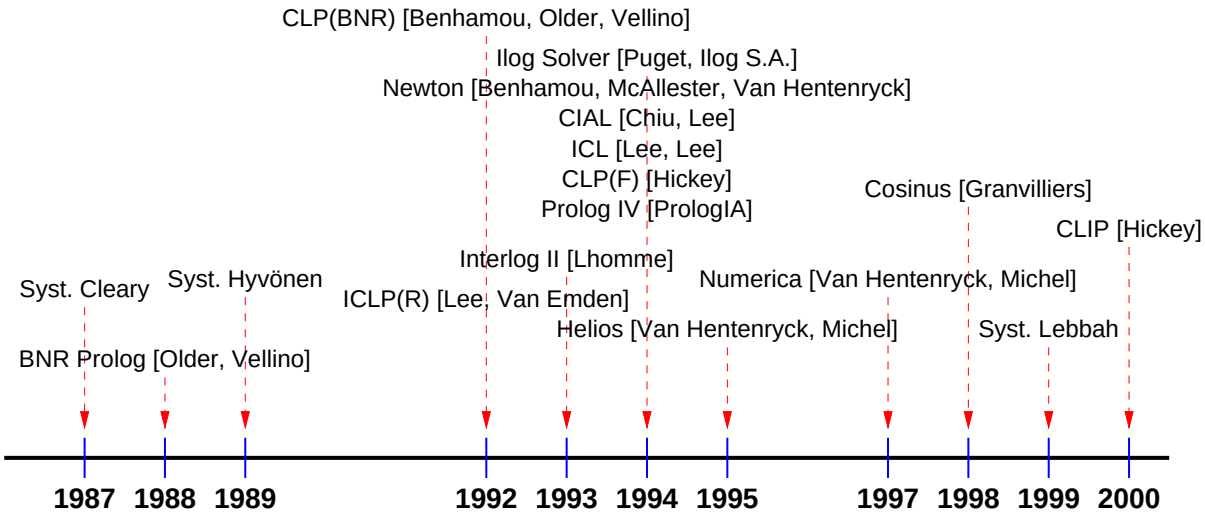


FIG. 7.1: Chronologie pour les langages de programmation par contraintes d'intervalles

CHAPITRE 8

DecLIC *(a Declarative Language with Interval Constraints)*

Présentation du langage — utilisation de la hull-consistance — utilisation de la box-consistance — choix de la consistance — présentation de la WAM — clp(FD) — implémentation de DecLIC

AFIN D'ÉTUDIER L'IMPACT DU CHOIX DES CONSISTANCES sur la résolution des systèmes de contraintes réelles, nous avons défini et implémenté DecLIC, un langage de résolution de contraintes réelles, entières et booléennes autorisant le programmeur à choisir la consistance — hull-consistance ou box$_\varphi$-consistance (*cf.* déf. 3.7 p. 40 et déf. 3.11, p. 47) — à appliquer aux contraintes du système. La détermination de la consistance utilisée se fait contrainte par contrainte de façon syntaxique. Cela nous a permis de constater qu'aucune des deux consistances pré-citées n'est le meilleur choix pour toutes les contraintes. Nous dégageons dans ce chapitre un certain nombre de « règles » pour guider le programmeur dans le choix de la « meilleure » consistance à utiliser en fonction de la forme de ses contraintes. Cependant, se décharger sur le programmeur du choix de la consistance détruit le caractère déclaratif de la programmation par contraintes. Aussi, en nous basant sur les résultats des expérimentations autorisées par DecLIC, nous avons généralisé la notion de box-consistance, ce qui nous a permis de définir un nouvel algorithme de réduction des domaines des variables combinant les avantages des algorithmes associés traditionnellement au calcul de la hull-consistance et de la box-consistance. La présentation et l'étude de ce nouvel algorithme ont fait l'objet du chapitre 4.

Dans la section 8.1, nous présentons le langage DecLIC en illustrant ses capacités « originales » par des exemples ; ceci nous permettra d'exposer le comportement des algorithmes utilisés pour le calcul de la hull-consistance et de la box-consistance. À partir des résultats expérimentaux présentés dans la section 8.2, nous tenterons d'extraire dans la section 8.3 certaines règles quant à la sélection de la consistance à appliquer en fonction de la forme des contraintes.

Nous avons utilisé clp(fd) [50] de CODOGNET et DIAZ, comme base pour l'implémentation de DecLIC ; clp(fd) est un langage étendant Prolog [55] pour permettre la résolution de contraintes entières et booléennes. Les programmes écrits dans ce langage peuvent être indifféremment interprétés ou compilés en exécutables via une extension de la *machine abstraite de* WARREN [247, 8] (WAM). Suivant l'approche de CODOGNET et DIAZ [50], nous avons étendu une fois de plus le jeu d'instructions de la WAM pour permettre la compilation de programmes DecLIC. Dans la section 8.4, nous présenterons les structures de données ainsi que les instructions de la WAM telle qu'implémentées dans clp(fd) ; nous décrirons ensuite dans la section 8.5 les modifications et extensions apportées pour implémenter DecLIC.

8.1 Présentation du langage

Cette section ne présente qu'un bref aperçu des possibilités de DecLIC en se focalisant sur les aspects qui en font la spécificité. Nous renvoyons le lecteur au manuel d'utilisation de DecLIC [92] pour une présentation approfondie du langage ainsi qu'une description exhaustive des prédicats prédéfinis.

DecLIC est un logiciel disponible sur le WEB* ; actuellement porté sur deux plate-formes, SUN UltraSparc/Solaris et ix86/Linux, il a été utilisé par Jorge CRUZ, doctorant à l'*universidade nova de Lisboa* pour résoudre

*`http://www.sciences.univ-nantes.fr/info/perso/permanents/goualard/Research/software.html`

des problèmes de diagnostic de dégénérescence nerveuse des extrémités [57] (bras, jambes). DecLIC n'est ni le premier système de résolution de contraintes basé sur Prolog, ni même le plus complet. Parmi les systèmes de résolution de contraintes réelles proches de DecLIC, on peut citer ICL [146], CIAL [45] (spécialisé dans la résolution de contraintes linéaires), BNR-Prolog [177], clp(BNR) [30] et Prolog IV [32]. On pourra aussi se reporter au chapitre 7 pour une description détaillée des différents langages de résolution de contraintes d'intervalles.

Ce qui différencie DecLIC de ces systèmes, c'est en premier lieu la possibilité d'utiliser différentes consistances en fonction de la forme des contraintes à résoudre.

DecLIC a été implémenté au-dessus de clp(fd) avec l'objectif d'assurer la plus grande compatibilité possible avec ce langage. Ainsi, la plupart des programmes clp(fd) sont aussi des programmes DecLIC valides. Cependant, les performances de DecLIC sont beaucoup moins bonnes que celles de clp(fd) en ce qui concerne les problèmes faisant intervenir uniquement des variables entières car DecLIC représente les domaines uniquement avec des intervalles à bornes flottantes, alors que clp(fd) est capable de représenter des domaines par des unions d'intervalles entiers.

D'un point de vue sémantique, DecLIC est une instance des langages de type CLP($\mathcal{X}$) [117] sur les réels. Un programme DecLIC est avant tout un programme Prolog[†] où certains prédicats servent à poser des contraintes. Il est possible de mélanger des contraintes booléennes, entières et réelles. Les variables apparaissant dans un programme DecLIC peuvent être séparées en trois catégories :

1. les *p-variables* ou variables Prolog pures ;
2. les *fd-variables*, contraintes à prendre des valeurs entières ou booléennes ;
3. les *i-variables*, contraintes à prendre des valeurs réelles.

Dans notre terminologie, les deux derniers types sont des *c-variables*, c'est à dire des variables apparaissant dans des contraintes.

Note : À la différence de clp(BNR) [30, 12], le type d'une c-variable (entier *vs.* réel) n'est pas défini statiquement par une instruction de typage, mais dynamiquement grâce aux contraintes `is_integer/1` et `is_fd/1` (*cf.* prog. 6.2).

En plus des symboles relationnels définis par Prolog [66] et clp(fd) [65], DecLIC utilise douze symboles supplémentaires : $\$\diamond, \diamond \in \{=, \backslash =, <=, >=, <, >\}$ pour écrire les contraintes non-linéaires décomposées en conjonctions de contraintes primitives (hull-consistance) et $\$\$\diamond, \diamond \in \{=, \backslash =, <=, >=, <, >\}$ pour les contraintes non-linéaires traitées sans décomposition (box-consistance).

Nous avons choisi de gérer les intervalles ouverts et fermés, comme cela est fait dans Prolog IV [32] afin d'avoir une plus grande correction dans les calculs. En particulier, cela nous permet de détecter des incohérences que nous ne pourrions pas détecter avec les seuls intervalles fermés.

Dans les sections suivantes, nous allons présenter les caractéristiques principales de DecLIC. Nous commencerons par illustrer la possibilité de choisir la consistance à utiliser pour résoudre des contraintes en donnant deux exemples de programmes : un programme posant des contraintes « quasi-primitives » (petits termes[‡] contenant des variables de très faible multiplicité) utilisant uniquement la hull-consistance, et un programme posant des contraintes « complexes » (termes composés de variables à occurrences multiples et de nombreux opérateurs arithmétiques). Les choix des consistances en fonction des exemples seront justifiés plus loin.

8.1.1 Utilisation de la hull-consistance

Notre premier exemple correspond à la recherche des coordonnées des points définissant un pentagone régulier [151] : ayant fixé le point p_1 en $(1, 0)$, on recherche les valeurs des abscisses et ordonnées des points p_2, p_3, p_4 et p_5 telles que $\langle p_1, p_2, p_3, p_4, p_5 \rangle$ soit un pentagone régulier (*cf.* fig. 8.1). On a donc les contraintes :

[†] DecLIC partage avec clp(fd) la totalité des prédicats Prolog. La syntaxe utilisée est celle définie par la norme ISO [192, 63] avec cependant quelques écarts au standard en ce qui concerne le comportement des prédicats.

[‡] Rappelons que notre définition de *contrainte primitive* ne fait pas intervenir la taille de la contrainte. D'un point de vue pratique cependant, on peut considérer que les seules contraintes primitives sont celles ne faisant pas intervenir plus d'un opérateur arithmétique.

$$
\begin{array}{ll}
\forall i \in \{1,\ldots,5\}: & \\
\qquad x_i^2 + y_i^2 = 1 & \text{Pentagone inscrit dans un cercle de rayon 1} \\
\qquad (x_i - x_{(i \bmod 5)+1})^2 + (y_i - y_{(i \bmod 5)+1})^2 = D^2 & \text{Points répartis régulièrement} \\
y_2 < y1, x_3 < x_2, y_4 > y_3, x_5 > x_4, y_5 > y_1 & \text{Figure convexe (pas une étoile)} \\
(x_1, y_1) = (1, 0) & \text{Un point fixé : } p_1
\end{array}
\tag{8.1}
$$

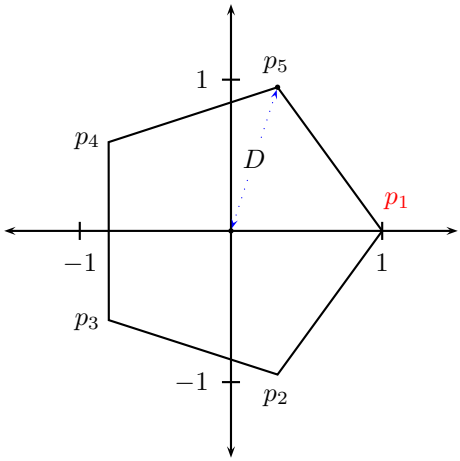


FIG. 8.1: Problème du pentagone régulier

Ce système de contraintes réelles se traduit en le programme DecLIC donné par le programme 8.1. Le code est séparable en deux parties :

1. Des lignes 1 à 19, on ajoute les contraintes au *store* ;
2. la ligne 20 commence une phase de découpage des domaines (stratégie *round-robin* de considération des variables) entraînant de nouvelles propagations dans le *store*, afin de pallier l'incomplétude des algorithmes utilisés et séparer les différentes solutions.

Les contraintes du système (8.1) sont de « petite taille » et sont décomposables facilement en ensembles restreints de primitives. De plus, chaque variable ne possède qu'une occurrence dans chacune d'elles. Comme on le verra plus loin, de telles contraintes sont de bonnes candidates pour être résolues en utilisant la hull-consistance car la décomposition étant ici limitée, on n'a pas trop de perte d'information du fait de l'introduction de nouvelles variables. Par ailleurs, les contraintes primitives étant codées une fois pour toutes dans le solveur, leurs opérateurs de contraction sont particulièrement rapides.

8.1.2 Utilisation de la box-consistance

Nous allons maintenant considérer le cas de la box-consistance en étudiant la résolution d'un système constitué des fonctions dites de *Broyden-Banded* [170] :

$$f_i(x_1,\ldots,x_n) = x_i(2+5x_i^2) + 1 - \sum_{j \in J_i} x_j(1+x_j) \qquad (1 \leqslant i \leqslant n)$$

avec

$$J_i = \{j \mid j \neq i \wedge \max(1, i-5) \leqslant j \leqslant \min(n, i+1)\}$$

PROG. 8.1: Programme DecLIC pour le problème du pentagone régulier [151]

```
pentagon(p(X1,Y1),p(X2,Y2),p(X3,Y3),p(X4,Y4),p(X5,Y5),D):-
     norm(p(X1,Y1),1),
     norm(p(X2,Y2),1),
     norm(p(X3,Y3),1),
     norm(p(X4,Y4),1),
     norm(p(X5,Y5),1),
     distance(p(X1,Y1),p(X2,Y2),D),
     distance(p(X2,Y2),p(X3,Y3),D),
     distance(p(X3,Y3),p(X4,Y4),D),
     distance(p(X4,Y4),p(X5,Y5),D),
     distance(p(X5,Y5),p(X1,Y1),D),
     X1 in {1.0}, Y1 in {0.0},                                    %  X1 ∈ [1.0 .. 1.0]
     Y2 #< Y1,                                  % #< : relation entre deux termes linéaires
     X3 #< X2, Y4 #> Y3, X5 #> X4, Y5 #> Y1,
     D in o(0.3)..o(10000.0).                                     %  D ∈ (0.3 .. 10000)
norm(p(X,Y),N):-
     X**2+Y**2 $= N**2.
distance(p(X1,Y1),p(X2,Y2),D):-
     (X1-X2)**2+(Y1-Y2)**2 $= D**2.
:- pentagon(p(X1,Y1),p(X2,Y2),p(X3,Y3),p(X4,Y4),p(X5,Y5),D),
   solve([X2,X3,X4,X5,Y2,Y3,Y4,Y5,D]).                  %  Découpage à 1.0e-8 (défaut)
```

Le programme 8.2 décrit le programme DecLIC résolvant le système d'équations

$$\forall i \in \{1, \ldots, n\} : f_i(x_1, \ldots, x_n) = 0$$

pour $n = 4$.

Notons que, contrairement aux contraintes du programme 8.1, celles du programme 8.2 sont « complexes » dans le sens où, si l'on considère que les primitives dont on dispose sont au plus ternaires (un seul opérateur arithmétique), leur décomposition entraîne l'introduction de nombreuses variables. C'est pourquoi l'utilisation de la box-consistance semble « intuitivement » s'imposer puisque l'algorithme employé est capable de manipuler les contraintes sans les décomposer.

Nous avons aussi pu nous rendre compte qu'il est parfois plus efficace d'introduire toutes les contraintes d'un système « régulier » comme celui ci-dessus dans le *store* avant de faire la propagation que de les y ajouter incrémentalement. Aussi, DecLIC définit le connecteur and permettant de relier plusieurs contraintes entre elles afin de spécifier que l'on souhaite les ajouter ensemble au *store*. Le programme 8.2 est, comme le programme 8.1, décomposé en deux parties : une de pose des contraintes et une de découpage des domaines des variables.

PROG. 8.2: Le programme DecLIC pour le système de fonctions *Broyden-Banded* [170]

```
:- domain([X1,X2,X3,X4],-1,1),                          %  X_i = [−1 .. 1],  ∀ i ∈ {1, . . . , 4}
   X1*(2+5*X1**2) + 1 - (X1*(1+X1) + X2*(1+X2))          $$= 0 and
   X2*(2+5*X2**2) + 1 - (X1*(1+X1) + X3*(1+X3))          $$= 0 and
   X3*(2+5*X3**2) + 1 - (X1*(1+X1) + X2*(1+X2) + X4*(1+X4)) $$= 0 and
   X4*(2+5*X4**2) + 1 - (X1*(1+X1) + X2*(1+X2) + X3*(1+X3)) $$= 0,
   solve([X1,X2,X3,X4]).
```

Nous allons maintenant présenter des exemples de programmes mélangeant soit des contraintes sur des variables de domaines différents, soit des contraintes résolues en utilisant des consistances différentes.

8.1.3 Variables entières et variables réelles

Le type d'une variable (booléenne, entière, réelle) n'est pas déterminé statiquement en DecLIC par une instruction de typage. Il l'est, soit implicitement par l'utilisation de certaines contraintes (*e.g.* and(X,Y,Z) —

$X \wedge Y = Z$ — contraint les variables X, Y et Z à être de type booléen), soit explicitement en ayant recours aux contraintes `is_boolean/1`, `is_integer/1` et `is_fd/1` qui contraignent la variable passée en argument à être, respectivement, de type booléen, entier et entier positif.

Note : Pour DecLIC, comme pour clp(fd), les variables booléennes sont en fait des variables entières dont le domaine est restreint aux deux valeurs 0 et 1.

Grâce à la contrainte `is_fd/1`, il est possible de chercher uniquement les solutions entières positives à des systèmes de contraintes faisant intervenir des calculs réels (*cf.* prog. 6.2).

8.1.4 Hull-consistance et Box-consistance

Considérons une contrainte c faisant intervenir des variables avec de grandes multiplicités. Si l'on décompose c afin d'utiliser la hull-consistance, on va perdre beaucoup d'informations car on ne reliera plus les différentes occurrences d'une même variable les unes aux autres. Il est donc plus intéressant de calculer la box-consistance pour c, ce qui nous évite de la décomposer. Mais, si l'on considère maintenant une contrainte c' de grande arité et dont les variables ont des multiplicités faibles, il ne devient plus aussi intéressant d'utiliser la box-consistance. En effet, l'algorithme appliqué va alors devoir invoquer une procédure coûteuse de recherche des quasi-zéros extrêmes (généralement basée sur la méthode de NEWTON, quoiqu'une recherche dichotomique soit aussi possible) pour chacune des — nombreuses — projections de c'. Or, si l'on décompose c' en primitives, on perdra peu d'informations du fait de la faible multiplicité des variables. On verra dans le chapitre 4 que l'on peut même envisager de ne pas décomposer du tout c' tout en calculant la hull-consistance (ce qui nous permet, lorsque toutes les variables de c' n'ont qu'une seule occurrence de tirer parti du théorème des occurrences simples — *cf.* théo. 2.1 — pour calculer aisément la hull-consistance).
Soit le système de contraintes P_1 suivant :

$$x^{10} - 2x^9 + 6x^8 - 10x^7 + 17x^6 - 24x^5 + 52x^4 - 80x^3 + 72x^2 - 64x + 32 = 0 \tag{8.2}$$

$$x \prod_{i=1}^{10} y_i = 0 \tag{8.3}$$

$$\sum_{i=1}^{10} y_i^2 = 0 \tag{8.4}$$

En tirant parti des observations ci-dessus, on va choisir de résoudre la contrainte (8.2) par un calcul de box-consistance et les contraintes (8.3) et (8.4) par des calculs de hull-consistance, ce qui donne le code DecLIC du programme 8.3

PROG. 8.3: Utilisation conjointe de consistances

```
1  product([],1).
2  product([X|L],P):- product(L,Q), P $= X*Q.
3  sumSquare([],0).
4  sumSquare([X|L],S):- sumSquare(L,T), S $= X**2+T.
5  :- length(L,10),                                          % L = [Y1, Y2, Y3, ..., Y10]
6     X in -1.0e8..1.0e8, L2=[X|L],
7     X**10 + 17*X**6 + 6*X**8 + 52*X**4 + 72*X**2 - 2*X**9 - 24*X**5 -
8     10*X**7 - 80*X**3 - 64*X + 32 $$= 0,
9     product(L2,0),                                          % X*Y1*Y2*Y3*...*Y10 = 0
10    sumSquare(L,0),                                % Y1**2 + Y2**2 + ... + Y10**2 = 0
11    solve(L2).
```

On trouvera dans la section 8.2 une comparaison des temps de calcul nécessaires à la résolution des exemples donnés ici en fonction de la consistance utilisée.

8.2 Résultats expérimentaux

Nous allons maintenant présenter certains problèmes de test pour illustrer l'importance du choix de la consistance à utiliser pour chaque contrainte. Nous essayerons ensuite dans la section 8.3 de tirer de nos résultats des « règles » permettant de guider un tel choix.

8.2.1 Présentation des problèmes de test

Nous avons sélectionné onze problèmes de la communauté *Intervalles* dont la structure permet des comparaisons intéressantes en ce qui concerne le choix des consistances utilisées. Les temps de résolution sur une SUN UltraSparc 1/167 MHz pour ces problèmes sont rassemblés dans les tableaux 8.1 et 8.2. Nous avons comparé les temps pour le calcul de la hull consistance, de la box-consistance et de la box$_\varphi$-consistance avec un φ fixé à 0,1. Les domaines finaux des variables ont tous une taille inférieure à 10^{-8} (paramètre d'arrêt dans la procédure de découpage initiée par le prédicat `Solve/1,2`). De plus, nous avons utilisé un *facteur d'amélioration*[§] de $10\,\%$.

Le tableau 8.1 présente les temps obtenus lorsque l'on utilise une seule consistance pour toutes les contraintes ; le tableau 8.2 compare les temps de résolution pour certaines contraintes lorsque l'on s'autorise à choisir la consistance à utiliser en fonction de la forme de la contrainte.

Dans la suite, une marque « ✓ » identifie les contraintes gérées par la box-consistance de celles gérées par la hull-consistance (tab. 8.2).

`Bifurcation` ***Problème de bifurcation [131]***

Description :

$$5z_1^9 - 6z_1^5 z_2 + z_1 z_2^4 + 2z_1 z_3 = 0 \tag{8.5}$$
$$-2z_1^6 z_2 + 2z_1^2 z_2^3 + 2z_2 z_3 = 0 \tag{8.6}$$
$$z_1^2 + z_2^2 - 0{,}265\,625 = 0 \tag{8.7}$$

`Chemical Eq.` ***Équilibre chimique [164]***

Description : Étant donnée l'équation stœchiométrique de la combustion du propane dans l'air :

$$\mathrm{C_3H_8} + \frac{\mathrm{R}}{2}(\mathrm{O_2} + 4\mathrm{N_2}) \longrightarrow (\mathrm{CO_2}, \mathrm{H_2O}, \mathrm{N_2}, \mathrm{CO}, \mathrm{H_2}, \mathrm{H}, \mathrm{OH}, \mathrm{O}, \mathrm{NO}, \mathrm{O_2})$$

on cherche le nombre de moles de produits formés par mole de propane consommé.

On a le système d'équations suivant :

$$y_1 = x_1 x_2$$
$$y_2 = 3{,}448 \cdot 10^{-3}/\sqrt{40} x_2 x_3 \checkmark$$
$$y_3 = 2{,}155 \cdot 10^{-4}/\sqrt{40} x_2 x_4 \checkmark$$
$$y_4 = 3{,}846 \cdot 10^{-5}/40 x_2^2 \checkmark$$
$$y_5 = 1{,}93 \cdot 10^{-1} x_3^2$$
$$y_6 = x_4^2$$
$$y_7 = x_2 x_3^2$$
$$z_1 = 1{,}799 \cdot 10^{-5}/40 x_2$$
$$2{,}597 \cdot 10^{-3}/\sqrt{40} x_3 = z_2$$
$$10 x_5 = z_3$$
$$x_1 + y_7 + z_1 + y_2 + y_3 = z_4 \checkmark$$
$$y_1 + x_1 - 3x_5 = 0 \checkmark$$
$$2y_1 + z_4 - z_3 + 2y_4 = 0 \checkmark$$
$$2y_7 + 2y_5 - 8x_5 + z_2 + y_2 = 0 \checkmark$$
$$y_3 + 2y_6 - 4z_3 = 0 \checkmark$$
$$y_1 + y_5 + y_6 - 1 + y_4 + z_2 + z_4 = 0 \checkmark$$

[§] Après avoir réduit le domaine d'une variable, il est parfois trop coûteux de propager son nouveau domaine dans le *store* si la réduction est faible. En introduisant un *facteur d'amélioration* ζ, on s'autorise à ne pas faire de propagation si la taille de la variable a été diminué de moins de $\zeta\,\%$.

i_1

Description :

$\forall i \in \{1, \ldots, 10\} : x_i \in [-2 \,..\, 2]$

$$
\begin{array}{lllll}
x_1 & - & 0{,}254\,287\,22 & - & 0{,}183\,247\,57 x_4 x_3 x_9 = 0 \\
x_2 & - & 0{,}378\,421\,97 & - & 0{,}162\,754\,49 x_1 x_{10} x_6 = 0 \\
x_3 & - & 0{,}271\,625\,77 & - & 0{,}169\,550\,71 x_1 x_2 x_{10} = 0 \\
x_4 & - & 0{,}198\,079\,14 & - & 0{,}155\,853\,16 x_7 x_1 x_6 = 0 \\
x_5 & - & 0{,}441\,667\,28 & - & 0{,}199\,509\,20 x_7 x_6 x_3 = 0 \\
x_6 & - & 0{,}146\,541\,13 & - & 0{,}189\,227\,93 x_8 x_5 x_{10} = 0 \\
x_7 & - & 0{,}429\,371\,61 & - & 0{,}211\,804\,86 x_2 x_5 x_8 = 0 \\
x_8 & - & 0{,}070\,564\,38 & - & 0{,}170\,812\,08 x_1 x_7 x_6 = 0 \\
x_9 & - & 0{,}345\,049\,06 & - & 0{,}196\,127\,40 x_{10} x_6 x_8 = 0 \\
x_{10} & - & 0{,}426\,511\,02 & - & 0{,}214\,665\,44 x_4 x_8 x_1 = 0
\end{array}
$$

i_2

Description :

$\forall i \in \{1, \ldots, 10\} : x_i \in [-1 \,..\, 2]$

$$
\begin{array}{lllll}
x_1 & - & 0{,}254\,287\,22 & - & 0{,}183\,247\,57 x_4 x_3 x_9 = 0 \\
x_2 & - & 0{,}378\,421\,97 & - & 0{,}162\,754\,49 x_1 x_{10} x_6 = 0 \\
x_3 & - & 0{,}271\,625\,77 & - & 0{,}169\,550\,71 x_1 x_2 x_{10} = 0 \\
x_4 & - & 0{,}198\,079\,14 & - & 0{,}155\,853\,16 x_7 x_1 x_6 = 0 \\
x_5 & - & 0{,}441\,667\,28 & - & 0{,}199\,509\,20 x_7 x_6 x_3 = 0 \\
x_6 & - & 0{,}146\,541\,13 & - & 0{,}189\,227\,93 x_8 x_5 x_{10} = 0 \\
x_7 & - & 0{,}429\,371\,61 & - & 0{,}211\,804\,86 x_2 x_5 x_8 = 0 \\
x_8 & - & 0{,}070\,564\,38 & - & 0{,}170\,812\,08 x_1 x_7 x_6 = 0 \\
x_9 & - & 0{,}345\,049\,06 & - & 0{,}196\,127\,40 x_{10} x_6 x_8 = 0 \\
x_{10} & - & 0{,}426\,511\,02 & - & 0{,}214\,665\,44 x_4 x_8 x_1 = 0
\end{array}
$$

i_4

Description :

$\forall i \in \{1, \ldots, 10\} : x_i \in [-1 \,..\, 1]$

$$
\begin{array}{lllll}
x_1^2 & - & 0{,}254\,287\,22 & - & 0{,}183\,247\,57 (x_4 x_3 x_9)^2 = 0 \\
x_2^2 & - & 0{,}378\,421\,97 & - & 0{,}162\,754\,49 (x_1 x_{10} x_6)^2 = 0 \\
x_3^2 & - & 0{,}271\,625\,77 & - & 0{,}169\,550\,71 (x_1 x_2 x_{10})^2 = 0 \\
x_4^2 & - & 0{,}198\,079\,14 & - & 0{,}155\,853\,16 (x_7 x_1 x_6)^2 = 0 \\
x_5^2 & - & 0{,}441\,667\,28 & - & 0{,}199\,509\,20 (x_7 x_6 x_3)^2 = 0 \\
x_6^2 & - & 0{,}146\,541\,13 & - & 0{,}189\,227\,93 (x_8 x_5 x_{10})^2 = 0 \\
x_7^2 & - & 0{,}429\,371\,61 & - & 0{,}211\,804\,86 (x_2 x_5 x_8)^2 = 0 \\
x_8^2 & - & 0{,}070\,564\,38 & - & 0{,}170\,812\,08 (x_1 x_7 x_6)^2 = 0 \\
x_9^2 & - & 0{,}345\,049\,06 & - & 0{,}196\,127\,40 (x_{10} x_6 x_8)^2 = 0 \\
x_{10}^2 & - & 0{,}426\,511\,02 & - & 0{,}214\,665\,44 (x_4 x_8 x_1)^2 = 0
\end{array}
$$

Description : On cherche à calculer par discrétisation l'intégrale

$$\int_0^1 H(s,t)(u(s)+s+1)^3\,\mathrm{d}s = 0$$

avec

$$H(s,t) = \begin{cases} s(1-t), s \leqslant t, \\ t(1-s), s > t \end{cases}$$

On obtient le système d'équations non-linéaire ci-dessous — pour un paramètre de discrétisation m fixé ici à 20 :

$\forall k \in \{1,\ldots,m\}$:

$$x_k \in [-4\,..\,5], \quad h = \frac{1}{m+1}$$

$$x_k + \frac{1}{2(m+1)}[(1-kh)\sum_{j=1}^{k} jh(x_j+jh+1)^3 + kh\sum_{j=k+1}^{m}(1-jh)(x_j+jh+1)^3] = 0$$

`Broyden-Banded 10` — *Problème de Broyden-Banded 10 variables*

Description :

$$f_i(x_1,\ldots,x_n) = x_i(2+5x_i^2)+1-\sum_{j\in J_i} x_j(1+x_j) \qquad (1 \leqslant i \leqslant n)$$

avec $J_i = \{j \mid j \neq i \wedge \max(1,i-5) \leqslant j \leqslant \min(n,i+1)\}$.

`Wilkinson` — *Polynôme de* WILKINSON *[249]*

Description : Considérons le polynôme $\prod_{i=1}^{20}(x+i)$. Trivialement, l'ensemble de ces racines est $S = \{-20,-19,\ldots,-1\}$. Il suffit cependant de lui ajouter le petit terme perturbateur $2^{-23}x^{19}$ pour que le nombre de racines se réduise à 10 :

$$P(x) = \prod_{i=1}^{20}(x+i) + \varepsilon\, x^{19}, \text{ avec} \quad \varepsilon = 2^{-23}$$

P_1 — *P_1 [94]*

Description :

$$\begin{cases} x^{10}-2x^9+6x^8-10x^7+17x^6-24x^5+52x^4-80x^3+72x^2-64x+32 & = 0 \checkmark \\ x\prod_{i=1}^{10} y_i & = 0 \\ \sum_{i=1}^{10} y_i^2 & = 0 \end{cases}$$

P_2 — *P_2 [94]*

Description :

$$\begin{cases} x_1^5-28x_1^4+288x_1^3-1\,358x_1^2+2\,927x_1-2\,310 & = & 0 \checkmark \\ x_2^4-10x_2^3+35x_2^2-50x_2+24 & = & 0 \checkmark \\ x_3^4-4x_3^3-53x_3^2+60x_3+108 & = & 0 \checkmark \\ x_4^5-23x_4^4+155x_4^3-241x_4^2-420x_4 & = & 0 \checkmark \\ x_5^5-12{,}5x_5^4-28{,}5x_5^3+341x_5^2+848x_5+336 & = & 0 \checkmark \\ x_6^7-28x_6^6+322x_6^5-1\,960x_6^4+6\,769x_6^3-13\,132x_6^2+13\,068x_6-5\,040 & = & 0 \checkmark \\ \sum_{i=1}^{6} x_i & = & 9 \\ \sum_{i=1}^{6} ix_i & = & 46 \\ \prod_{i=1}^{6} x_i & = & 0 \end{cases}$$

TAB. 8.1: Résultats expérimentaux : une seule consistance

Problèmes	***Hull***	***Box***	***Box*$_\varphi$**	***Box*$_\varphi$/*Hull***
Bifurcation	5 624	14 195	2 478	0,44
Pentagon	1 058	5 652	2 426	2,29
Chemical Eq.	5 556	14 621	4 993	0,90
i_1	413	224	117	0,28
i_2	467	228	115	0,25
i_4	?	?	130 500	?
Cosnard 20	?	62 701	28 910	?
Broyden-Banded 10	?	3 416	917	?
Wilkinson	3 049	539	437	0,14
P_1	2 510	?	10 100	4,02
P_2	?	181 871	28 955	?

Temps en millisecondes sur une UltraSparc 1/167 MHz.

TAB. 8.2: Résultats expérimentaux : plusieurs consistances

Problèmes	***Hull***	***Box*$_\varphi$**	***Hull*+*Box*$_\varphi$**
Pentagon	1 058	2 426	2 148
Chemical Eq.	5 556	4 993	5 783
P_1	2 510	10 100	1 070
P_2	?	28 955	19 976

Temps en millisecondes sur une UltraSparc 1/167 MHz.

8.3 Du choix de la consistance

Comme on l'a vu dans les sections précédentes, le choix de la consistance à utiliser pour traiter chaque contrainte est laissé à l'appréciation du concepteur d'un programme DecLIC. Ceci va à l'encontre de la déclarativité des langages de programmation par contraintes et cela requiert de plus de la part de l'utilisateur une bonne connaissance des méthodes de résolution implémentées. Il est donc souhaitable que ce soit le système qui décide lui-même de la consistance à appliquer. Une telle automatisation du choix semble cependant difficile. Des résultats expérimentaux de la section 8.2, on pourrait tirer les deux « règles » suivantes :

- si une contrainte contient peu de variables (chacune d'entre elles ayant une grande multiplicité), l'utilisation de la box-consistance est préférable ;
- si une contrainte contient beaucoup de variables (chacune de faible multiplicité), utiliser la hull-consistance.

Ces deux règles apparaissent cependant très floues. Que doit-on faire, par exemple, lorsqu'une contrainte contient beaucoup de variables ayant une grande multiplicité, ou peu de variables avec une faible multiplicité ? Les notions de « beaucoup » et « peu » sont elles-mêmes sujettes à de multiples interprétations. Il semble difficile de définir des règles plus précises si l'on se restreint à la vision « une consistance et un algorithme de résolution par contrainte utilisateur ». Nous verrons dans le chapitre 4 qu'il est possible de traiter efficacement chaque contrainte avec une seule consistance en étendant légèrement la définition de la box-consistance et en travaillant, non plus au niveau de la contrainte, mais au niveau de chacune de ses projections ; on peut alors utiliser des algorithmes différents en fonction de la multiplicité de chacune des variables ayant une occurrence dans la contrainte.

8.4 La Machine Abstraite de WARREN étendue de clp(fd)

Nous allons présenter dans cette section l'architecture et les instructions composant la *Warren Abstract Machine* [247] (WAM). Nous décrirons cependant ici une variante de la WAM originale, utilisée par CODOGNET et

DIAZ pour implémenter wamcc [49] et clp(fd) [50], et renvoyons le lecteur au rapport de WARREN [247], au livre de Aït-Kaci [9], ou au rapport de JANSSES *et al.* [119] pour une présentation approfondie de la WAM originale.

Les compilateurs pour Prolog ont longtemps eu la réputation d'être difficiles à implémenter du fait du haut niveau de ce langage. En particulier, contrairement aux langages impératifs classiques, l'analyse lexicographique et syntaxique d'un programme Prolog est un problème en soi du fait de la possibilité de définir de nouveaux opérateurs et de changer la priorité de ceux existant[¶]. Le passage direct d'un ensemble de clauses à du code machine pour un ordinateur étant malaisé, WARREN [247] eut l'idée en 1983 d'introduire une étape intermédiaire en définissant une *machine virtuelle*, la *Machine Abstraite de Warren* (*Warren Abstract Machine*, ou WAM) dont le jeu d'instructions permet une traduction aisée des mécanismes de Prolog et est par ailleurs compilable de façon efficace en code machine ou en un programme dans un langage impératif tel que le C.

La compilation d'un programme Prolog se fait alors en deux étapes :

1. Traduction du programme Prolog en une suite d'instructions de la WAM ;
2. Compilation (directe ou indirecte) du code WAM en une forme exécutable sur une machine d'architecture classique.

Outre la simplification du processus de compilation, l'introduction d'une *forme intermédiaire* permet de découper les compilateurs Prolog en une *partie frontale* et une *partie finale* (*cf.* fig. 8.2), offrant ainsi [7] la possibilité d'un reciblage plus aisé (pour obtenir un compilateur Prolog pour différentes machines, il suffit de ne réécrire que la partie finale) ainsi que l'opportunité d'effectuer des optimisations sur le code intermédiaire indépendantes du matériel.

Notons au passage que le point 2 n'est nullement obligatoire. Comme le souligne WARREN lui-même [247], on peut envisager d'utiliser du code WAM de trois façons différentes :

1. Exécution directe sur une vraie machine Prolog (*e.g.* la HPM (*High-speed Prolog Machine*) [174]) dont le jeu d'instructions et l'architecture correspondent à ceux de la WAM ;
2. Émulation du code WAM avec un émulateur écrit dans un langage impératif. Un inconvénient majeur de cette solution est la lenteur d'exécution caractéristique des émulateurs ;
3. Enfin, compilation du code WAM en code machine.

La troisième solution permet d'obtenir une bonne rapidité d'exécution, mais suppose l'implémentation d'un compilateur, travail lourd relevant du génie logiciel. Une approche médiane, adoptée par CODOGNET et DIAZ [49, 50], est de compiler le code WAM en un langage impératif comme le C et de se reposer sur les compilateurs de ce langage pour la génération du code machine.

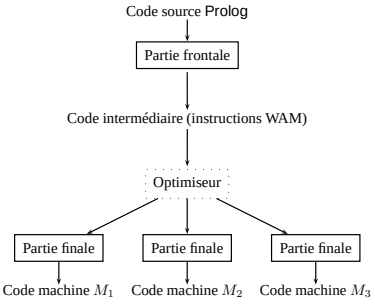


FIG. 8.2: Un compilateur Prolog en deux parties

La machine abstraite de WARREN se compose d'un ensemble de structures de données et d'un jeu d'instructions manipulant ces structures. Dans la section 8.4.1 nous présenterons brièvement l'architecture mémoire de la variante de la WAM de CODOGNET et DIAZ utilisée dans clp(fd) ; les sections 8.4.2 à 8.4.3 introduirons les instructions WAM en les groupant par fonctionnalités ; la section 8.4.5 donnera un exemple de compilation en code WAM d'un programme Prolog ; enfin, nous décrirons dans la section 8.4.6 les structures de données ainsi que les instructions ajoutées à la WAM pour gérer les contraintes sur les domaines finis dans clp(fd). Le lecteur trouvera une présentation détaillée de cette machine abstraite dans la thèse de DIAZ [67].

8.4.1 Gestion de la mémoire

L'espace mémoire géré par la WAM est divisé en quatre parties (*cf.* fig. 8.3) :

– la *zone de code* : c'est la portion de mémoire contenant les clauses du programme Prolog à exécuter. Nous la représentons à part dans la figure 8.3, car cette zone n'est pas réellement gérée par la variante de la WAM qui nous intéresse (les clauses étant transformées en fonctions C lors de la compilation) ;

[¶] Depuis les travaux de BOSSCHERE [62] datant de 1994, on peut cependant considérer que l'analyse d'un programme Prolog n'est plus un problème, celui-ci ayant mis dans le domaine public un analyseur lexicographique/syntaxique écrit en C que l'on peut librement réutiliser.

– le *trail* (ou *pile de restauration* [67]) : lors de l'exécution d'un programme Prolog suite à une requête, un certain nombre de liaisons de variables se produisent qu'il est nécessaire de détruire en cas de *backtracking*. Dans la WAM originale [247], le *trail* permet d'enregistrer les adresses des variables à délier lors de la reconsidération d'un point de choix. Dans la WAM de CODOGNET et DIAZ, le trail possède plusieurs types d'entrées permettant de :
  – délier une seule variable,
  – réinitialiser un ou plusieurs mots avec une valeur sauvegardée,
  – appeler une fonction C lors des réinitialisation.
  On se reportera à la thèse de DIAZ [67, p. 28–29] pour plus de détails ;
– le *tas* (ou *pile globale*) : malgré son nom, cette zone est gérée en pile (cette appellation est due à la possibilité de lier les mots le constituant tant du haut vers le bas que du bas vers le haut. C'est sur le tas que la WAM construit les structures (termes, listes) utilisées lors de l'exécution du programme ;
– la *pile locale* : de façon analogue aux programmes écrits dans des langages impératifs classique, la WAM utilise une pile pour sauvegarder certaines informations lors du passage du contrôle d'une clause à une autre (le corps d'une clause est considéré comme un ensemble d'appels de procédures). La pile sert aussi à enregistrer des *points de choix* permettant de gérer les multiples définitions d'un même prédicat.

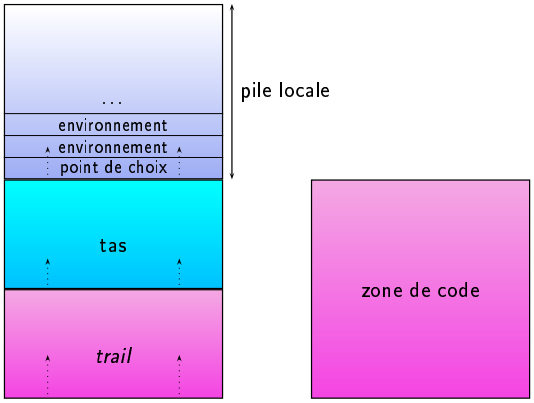


FIG. 8.3: Gestion de la mémoire par la WAM

Note : La WAM originale utilise une cinquième zone mémoire, la *PDL* (*Push-Down List*) sur laquelle sont empilées les différentes structures à unifier lors du passage de paramètres (appel de clause). Dans wamcc et clp(fd), la PDL a été avantageusement remplacée par un pointeur d'unification.

8.4.2 Instructions de contrôle

Considérons le programme Prolog ci-dessous :

```
1)    concatenate([],L,L).
2)    concatenate([X|L1],L2,[X|L3]):-
                concatenate(L1,L2,L3).
```

et la requête :

```
3)    ?- concatenate([1,2,3],[4,5,6],L).
```

En adoptant une vision procédurale et en considérant un ensemble de registres de passage $A_i, i \in \{1, \ldots, m\}$, l'exécution de `3)` se fait de la façon suivante :

1. Chargement de A_1 avec `[1,2,3]` ;

2. Chargement de A_2 avec `[4,5,6]` ;
3. Chargement de A_3 avec `L` ;
4. Appel de `concatenate/3` ;
5. Clause `1)` : unification de A_1, A_2, A_3 avec les paramètres attendus par `1)`, puis continuation en séquence ou *backtracking* ;
6. Clause `2)` :
   (a) Unification de A_1, A_2, A_3 avec les paramètres attendus par `2)` ;
   (b) Chargement de A_1 avec `L1`,
   (c) Chargement de A_2 avec `L2`,
   (d) Chargement de A_3 avec `L3`,
   (e) Appel de `concatenate/3`,
   (f) Continuation en séquence ;
7. Continuation en séquence.

Lorsqu'une clause est composée de plus d'un but, il faut aussi sauvegarder certains registres afin que leur contenu ne soit pas écrasé entre deux appels.

Toutes les clauses d'un programme Prolog sont regroupées par paquets de même *tête* (*i.e.* même nom de prédicat de tête et même arité) dans la zone de code. La WAM possède un registre `P` indiquant à tout instant l'instruction actuellement exécutée, ainsi qu'un registre `CP` pointant sur l'instruction à exécuter au retour d'un appel (pointeur de continuation).

Lors de l'appel d'un prédicat d'arité n, l'appelant charge n registres $(A_1, \ldots, A_n)$ avec les paramètres adéquats, puis effectue un branchement au point d'entrée de la zone de code correspondant au prédicat appelé, après avoir sauvé dans un *environnement* (*cf.* 8.4) créé sur la pile locale les valeurs de ses variables locales ainsi que le registre `CP` (qui risque d'être modifié par l'appelé si celui-ci passe à son tour le contrôle à d'autres sous-buts). Outre ces renseignements, l'environnement courant contient un pointeur vers l'environnement précédent (liste chaînée) afin de pouvoir restaurer la pile lors de la sortie de la clause.

CP	**Pointeur de continuation**
E	**Pointeur sur env. préc.**
Y[0]	**Var. permanente**
...	
Y[m]	**Var. permanente**

FIG. 8.4: Environnement de clause

On notera qu'une *règle de chaînage* (clause dont le corps ne contient qu'un seul but) n'a pas besoin d'environnement puisqu'aucun paramètre ne risque d'être écrasé. De même, il n'est pas nécessaire de sauver toutes les variables dans l'environnement : on distingue les *variables temporaires* (celles qui n'apparaissent que dans un seul but — la tête de la clause étant comptée avec le premier but) des variables permanentes (toutes les variables non temporaires [247]). Seules les variables permanentes doivent être sauvées dans l'environnement.

Les principales instructions WAM de contrôle sont :

`proceed` : branchement à l'instruction pointée par `CP` ;

`call P` : branchement au début de la zone de code correspondant au prédicat `P` avec positionnement du pointeur de continuation `CP` ;

`execute P` : identique à `call`, mais utilisée exclusivement pour l'appel du dernier but d'une clause : le positionnement de `CP` ainsi que la gestion de l'environnement sont superflus, ce dernier étant détruit au retour de la clause courante ;

`allocate` : création d'un environnement sur la pile locale en assurant le chaînage avec l'environnement précédent ;

`deallocate` : destruction du dernier environnement créé.

On peut alors dégager trois schémas d'exécution suivant la forme d'une clause :

`P.`	`P:-Q.`	`P:-Q,R1,...,Rm,S.`
Unification des arguments de `P`.	Unification des arguments de `P`.	`allocate`
`proceed`	Chargement des arguments de `Q`.	Unification des arguments de `P`.
	`execute Q`	Chargement des arguments de `Q`.
		`call Q`
		Chargement des arguments de `R1`
		`call R1`
		...
		Chargement des arguments de `Rm`
		`call Rm`
		Chargement des arguments de `S`
		`deallocate`
		`execute S`

On peut remarquer que, dans le cas du schéma de droite, on peut éliminer l'environnement avant l'appel du dernier but car, à ce moment, les paramètres de ce but sont déjà chargés dans les registres de passage.

Lorsqu'un prédicat possède plus d'une définition (clauses), il y a création d'un *point de choix* (*cf.* fig. 8.5) sur la pile locale. Un point de choix est une structure permettant la sauvegarde des informations qu'il est nécessaire de restaurer lors d'un *backtracking*. Il contient entre autres les valeurs courantes des pointeurs de pile et de tas (de façon à ce que l'on puisse éliminer tout ce qui a été créé lors de la tentative précédente), les arguments lors du premier appel, un pointeur sur la prochaine clause à essayer (mis à jour par chaque clause) et la valeur courante du pointeur de *trail* : en cas de *backtracking*, on libère toutes les variables apparaissant dans le *trail* jusqu'à ce point. Afin de limiter le nombre de points de choix, les clauses d'un paquet correspondant à la définition d'un prédicat sont regroupées en sous-blocs de telle sorte qu'il est possible de faire un premier filtrage des clauses dont la tête est susceptible de s'unifier avec l'appelant. On se reportera aux références données ci-dessus pour les détails.

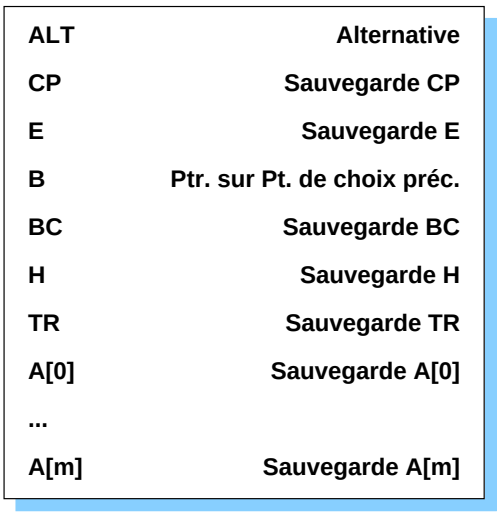


FIG. 8.5: Structure d'un point de choix

8.4.3 Représentation des termes

Les termes sont représentés en mémoire par des mots étiquetés indiquant leur type :
- une variable possède l'étiquette `REF`. Son champ valeur indique l'adresse de ce qu'elle contient. Une variable libre contient sa propre adresse ;
- une structure possède l'étiquette `STC`. Son champ valeur indique l'adresse d'un mot contenant le foncteur et l'arité n. Les n mots suivants correspondent aux sous-termes ;
- une constante entière a l'étiquette `INT`. Son champ valeur contient simplement sa valeur ;
- une constante flottante[‖] a l'étiquette `FLT`. Son champ valeur pointe sur le premier des deux mots consécutifs du tas représentant le nombre flottant ;
- une liste a l'étiquette `LST`. Les mots suivants contiennent les représentations de la tête et de la queue.

On trouvera dans la figure 8.6 des exemple de représentation pour chacun des types.

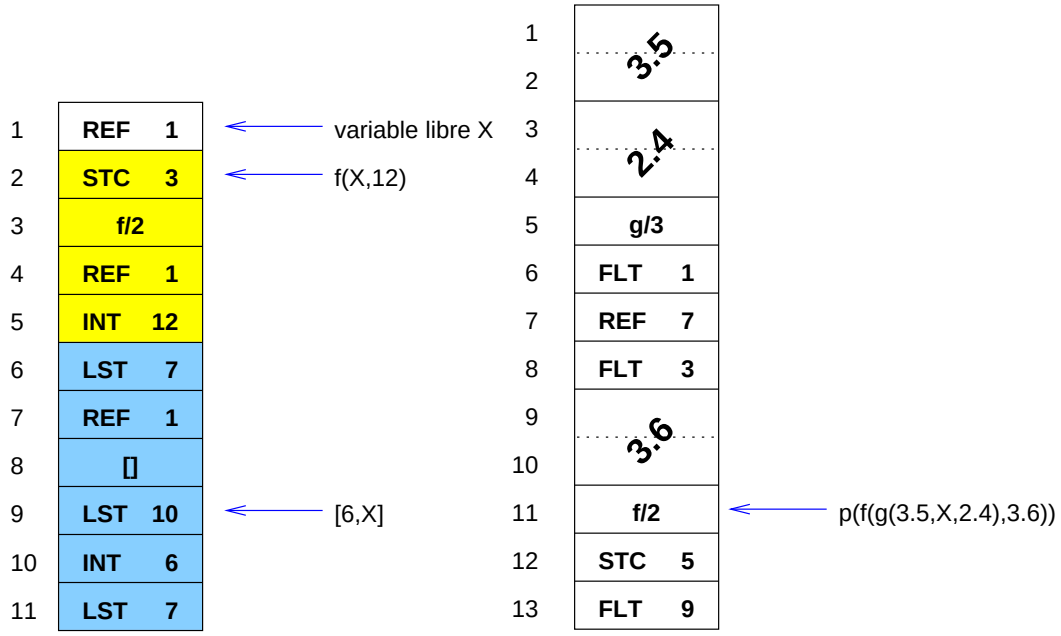


FIG. 8.6: Exemples de représentation de termes

8.4.4 Chargement des registres et unification

Le chargement des registres de passage se fait grâce à un ensemble d'instructions spécialisées correspondant aux différents types possibles de paramètres :
- les paramètres structurés (listes, termes avec foncteurs) ainsi que les constantes numériques flottantes sont construits sur le tas ; le registre contient l'adresse du premier mot de cette représentation ;
- les constantes numériques entières (ainsi que la constante `nil` — liste vide `[]`) sont directement chargées dans un registre. Les autres constantes sont stockées dans une table d'adressage dispersé et sont référencées par leur adresse dans cette table ;
- en ce qui concerne les variables en paramètre (*i.e.* n'apparaissant pas dans une structure), on charge le registre approprié avec leur adresse. Cette adresse est normalement localisée dans l'environnement de la clause courante, avec cependant deux exceptions :

[‖] La WAM de CODOGNET et `Diaz` [49, 50] ne permettait pas l'utilisation de nombres flottants. Leur introduction a été l'une des tâches effectués durant notre stage de DEA [91].

– si une variable a sa première occurrence dans le dernier but, sa place est allouée sur le tas. Ceci est dû au fait que l'on détruit l'environnement d'une clause avant l'appel du dernier but. Or, une variable dont la première occurrence est dans le dernier but est forcément libre avant l'appel (*i.e.* elle ne fait référence ni au tas, ni à un environnement précédent) et l'on risquerait de lier par la suite une variable à celle-ci, alors même que son emplacement mémoire a été libéré (référence fantôme),
– si à la dernière occurrence d'une *variable dangereuse*[**] celle-ci n'est pas encore liée, on lui alloue une place sur le tas (*globalisation*) et c'est ce nouvel emplacement mémoire qui est désormais considéré. Cette globalisation est faite pour les mêmes motifs que ceux évoqués au point précédent.

On a les (principales) instructions de chargement :

put_variable V_i**,** A_j**:** création d'une variable libre dans l'environnement. On fait pointer les registres A_j et V_i dessus ;

put_value V_i**,** A_j**:** le registre A_j reçoit la valeur de la variable référencée par V_i ;

put_unsafe_value Y_i**,** A_j**:** instruction identique à `put_value` avec globalisation si nécessaire ;

put_constant C**,** A_i**:** le registre A_i reçoit un mot taggé `CST` avec l'adresse de la constante dans la table d'adressage ;

put_integer I**,** A_i**:** A_i reçoit un mot taggé `INT` avec la valeur entière I ;

put_float N**,** A_i**:** A_i reçoit un mot taggé `FLT` avec l'adresse du nombre flottant N créé sur le tas ;

put_structure F**,** n**,** A_i**:** A_i reçoit un mot taggé `STC` avec l'adresse du début de la structure de foncteur F et d'arité N créée sur le tas ;

put_list A_i**:** spécialisation de `put_structure` pour les listes (pas de foncteur).

L'unification des paramètres d'un but se fait par les instructions spécialisées de la forme `get_XXX`.
On a les comportements suivants, selon la nature du i[e] paramètre de l'appelé :
– une variable libre : on recopie la valeur du registre A_i dans une variable locale ;
– une variable liée V_j : on unifie V_j avec A_i ;
– une constante : si A_i référence une variable libre, on lie cette variable à la constante ; sinon, on teste si ce que référence A_i est bien la constante attendue ;
– une liste ou une structure : les instructions `get_list` et `get_structure` possèdent deux modes : *READ* et *WRITE*. Si le registre A_i pointe sur une liste ou le foncteur attendu, on passe en mode READ et l'on cherche à unifier les sous-termes grâce aux instructions `unify_XXX` ; si A_i pointe sur une variable libre, on passe en mode WRITE et l'unification des sous-termes correspond alors à leur construction sur le tas ; sinon, il y a échec d'unification. Ainsi, les instructions `unify_XXX` ont un rôle de construction ou d'appariement suivant que le mode est READ ou WRITE.

On a les instructions :

get_variable V_i**,** A_j**:** affectation du contenu de A_j au registre V_i ;

get_value V_i**,** A_j**:** unification des contenus de V_i et A_j ;

get_integer N**,** A_i**:** unification de l'entier N et du contenu du registre A_i ;

get_float N**,** A_i**:** unification du nombre flottant N et du contenu du registre A_i ;

get_structure F**,** n**,** A_i**:** voir le point sur les listes/structures ci-dessus ;

get_list A_i**:** voir le point sur les listes/structures ci-dessus ;

unify_void N**:** crée ou passe N cellules correspondant à des arguments anonymes ;

unify_variable V_i**:** charge le registre V_i avec le contenu de la cellule courante (dans le processus d'unification), ou crée une variable libre et fait pointer le registre dessus (dépend du mode READ/WRITE courant) ;

unify_value V_i**:** unifie le contenu du registre V_i avec l'argument courant, ou met sur le tas le contenu du registre ;

unify_local_value V_i**:** instruction identique à `unify_value` avec globalisation si nécessaire ;

[**] Une variable dangereuse est une variable dont la première occurrence est en paramètre d'un but du corps d'une clause.

`unify_integer` N **:** unifie le contenu de la cellule courante avec l'entier N (*i.e.* compare le contenu de la cellule avec N et pousse N sur le tas si nécessaire) ;

`unify_float` N **:** unifie le contenu de la cellule courante avec le nombre flottant N ;

`unify_constant` C **:** unifie le contenu de la cellule courante avec la constante C.

8.4.5 Exemples de compilation

Afin d'illustrer les notions décrites dans les sections précédentes, nous allons donner ici deux exemples de compilation de programmes Prolog en code WAM empruntés au rapport de WARREN [247].

```
concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]):- concatenate(L1,L2,L3).

concatenate/3: switch_on_term C1a,C1,C2,fail
                          %    ^  ^   ^  ^--- si 1er parametre = structure
                          %    |  |   +------ si 1er parametre = liste
                          %    |  +---------- si 1er parametre = constante
                          %    +------------- si 1er parametre = variable

C1a: try_me_else C2a       % concatenate(
C1:  get_nil A1            %    [],
     get_value A2,A3       %    L,L
     proceed               % ).

C2a: trust_me_else fail    % concatenate(
C2:  get_list A1           %    [
     unify_variable X4     %       X|
     unify_variable A1     %       L1], L2,
     get_list A3           %    [
     unify_value X4        %       X|
     unify_variable A3     %       L3]):-
     execute concatenate/3 % concatenate(L1,L2,L3).
```

On remarquera l'absence de création d'environnement, due au fait qu'aucune des deux clauses ne possède plus d'un but dans son corps. On notera également que les paramètres qui ne changent pas de « place » d'un appel de `concatenate` à l'autre ne font l'objet d'aucune manipulation puisqu'ils sont déjà placés dans les bons registres ; c'est par exemple le cas de `L2`.

Considérons maintenant un programme rendant nécessaire l'utilisation d'un environnement :

```
qsort([],R,R).
qsort([X|L],R0,R):-
        split(L,X,L1,L2), qsort(L1,R0,[X|R1]), qsort(L2,R1,R).

qsort/3: switch_on_term C1a,C1,C2,fail

C1a: try_me_else C2a          % qsort(
C1:  get_nil A1               %    []
     get_value A2,A3          %    R,R
     proceed                  % ).

C2a: trust_me_else fail       % qsort(
C2:  allocate                 %
     get_list A1              %    [
```

```
unify_variable Y6        %       X|
unify_variable A1        %       L],
get_variable Y5,A2       %    R0,
get_variable Y3,A3       %    R):-
put_value Y6,A2          % split(L,X,
put_variable Y4,A3       %    L1,
put_variable Y1,A4       %    L2
call split/4,6           % ),
put_unsafe_value Y4,A1   % qsort(L1,
pu_value Y5,A2           %    R0,
put_list A3              %    [
unify_value Y6           %       X|
unify_variable Y2        %       R1]
call qsort/3,3           % ),
put_unsafe_value Y1,A1   % qsort(L2,
put_value Y2,A2          %    R1,
put_value Y3,A3          %    R
deallocate               %
execute qsort/3          % ).
```

8.4.6 Extensions de la WAM pour clp(fd)

Toutes les contraintes gérées par clp(fd) sont définies à partir d'une unique contrainte primitive : `X in R` (*cf.* la table 8.3 pour la grammaire de cette contrainte).

`X` est une variable représentant un entier ou un booléen (*variable DF*). Il lui est associé un domaine fini de valeurs possibles. La contrainte `X in R` impose à `X` d'appartenir au domaine `R`. Dans clp(fd), un domaine peut être un intervalle ou une union d'intervalles.

Le domaine dénoté par `R` peut dépendre du domaine d'autres variables grâce aux *indexicaux* `min()`, `max()` et `dom()`.

Exemple 8.20 (Utilisation d'indexicaux). *Soient X et Y des variables DF, avec :*

$$\begin{cases} X \in \{1,2,3,4,5\} \\ Y \in \{2,3\} \cup \{5,6\} \end{cases}$$

La contrainte `X in min(Y)..max(Y)` *contraint* `X` *à appartenir à l'intervalle* $[2\,..\,6]$*, alors que* `X in dom(Y)` *la contraint à appartenir à l'union d'intervalles* $[2\,..\,3] \cup [5\,..\,6]$.

Il existe aussi un indexical permettant de gérer les contraintes *anti-monotones* (*i.e.* les contraintes qui font augmenter la taille d'un domaine au lieu de le réduire) telles que la complémentation : `X in -dom(Y)`. On utilise l'indexical `val()` pour « geler » une contrainte jusqu'à ce que son argument soit lié à une seule valeur. Ceci permet d'écrire, par exemple, la contrainte de diséquation :

```
'x<>y'(X,Y):-
      X in -{val(Y)},
      Y in -{val(X)}.
```

L'expression des contraintes « primitives » (addition, multiplication…) se fait de façon naturelle en terme de conjonctions de contraintes `X in R`. On a par exemple :

```
'x+y=z'(X,Y,Z):-
      X in (min(Z)-max(Y))..(max(Z)-min(Y)),
      Y in (min(Z)-max(X))..(max(Z)-min(X)),
      Z in (min(X)+min(Y))..(max(X)+max(Y)).
```

TAB. 8.3: Grammaire de la contrainte X *in* R dans clp(fd)

$c ::=$	X *in* R	(contrainte)
$r ::=$	t_1 .. t_2	(intervalle)
	{t}	(singleton)
	R	(paramètre domaine)
	dom(Y)	(domaine indexical)
	r_1 : r_2	(union)
	r_1 & r_2	(intersection)
	-r	(complémentation)
	r + t	(addition domaine/terme)
	r - t	(soustraction domaine/terme)
	r * t	(multiplication domaine/terme)
	r / t	(division domaine/terme)
	r mod t	(modulo domaine/terme)
	r_1 + r_2	(addition domaine/domaine)
	r_1 - r_2	(soustraction domaine/domaine)
	r_1 * r_2	(multiplication domaine/domaine)
	r_1 / r_2	(division domaine/domaine)
	r_1 mod r_2	(modulo domaine/domaine)
	f_r(a_1,...,a_k)	(fonction utilisateur)
$a ::=$	$r \mid t$	(argument de fonction utilisateur)
$t ::=$	min(Y)	(indexical min)
	max(Y)	(indexical max)
	val(Y)	(valeur de Y lié)
	t_1 mod t_2	
	ct | t_1+t_2 | t_1-t_2 | t_1*t_2 | t_1/<t_2 | t_1/>t_2	
	f_t(a_1,...,a_k)	
$ct ::=$	C	(terme)
	n | infinity | ct_1+ct_2 | ct_1-ct_2 | ct_1*ct_2 | ct_1/<ct_2 | ct_1/>ct_2	
	ct_1 mod ct_2	

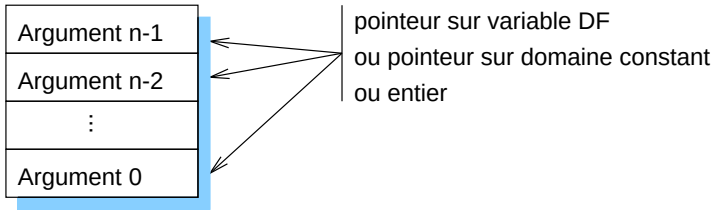


FIG. 8.7: Structure pour les arguments

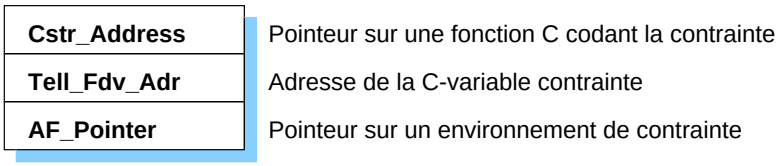


FIG. 8.8: Structure pour une contrainte

L'introduction des variables DF dans wamcc pour créer clp(fd) a conduit CODOGNET et DIAZ à ajouter à la WAM de nouvelles structures de données ainsi que de nouvelles instructions décrites ci-après.

Les règles d'unification pour une variable DF X sont les suivantes :
- avec une variable Prolog Y : Y est lié à X ;
- avec un entier n : ajout de la contrainte X *in* $n..n$;
- avec une variable DF Y : ajout des contraintes X *in* dom(Y) et Y *in* dom(X).

Les nouvelles structures de données pour gérer les contraintes sont décrites ci-dessous.

Les environnements Un *environnement* (*cf.* fig. 8.7) est une structure partagée par toutes les contraintes d'une clause. Elle contient l'adresse de toutes les variables DF, constantes entières et domaines apparaissant dans ces contraintes.

Exemple 8.21 (Environnement pour contraintes). *Étant donnée la clause :*

```
'x=y+c'(X,Y,C):- X in min(Y)+C..max(Y)+C,
                 Y in min(X)-C..max(X)-C.
```

l'environnement associé contiendra un pointeur sur X et sur Y, ainsi que l'entier C.

Représentation des contraintes À chaque contrainte X *in* R est associée une structure (*cf.* fig. 8.8) stockant :
- un pointeur sur l'environnement d'évaluation de la contrainte ;
- un pointeur sur la variable contrainte X ;
- l'adresse d'une fonction C évaluant le domaine R et mettant à jour le domaine de X.

Représentation des domaines Les domaines admettent deux représentation suivant qu'il s'agit d'intervalles ou d'unions d'intervalles :
- un couple (min, max) de valeurs entières pour les intervalles ;
- un vecteur de bits codant en extension le domaine pour les unions d'intervalles.

Représentation des variables DF Une variable DF V (*cf.* figure 8.9) est caractérisée par :
- son domaine ;
- les contraintes qui dépendent d'elle (*i.e.* celles où elle apparaît dans la partie R de X *in* R). Pour une plus grande efficacité, on met dans des listes séparées les contraintes dépendant uniquement de la borne gauche de V, de sa borne droite... ce qui évite de réexécuter inutilement des contraintes.

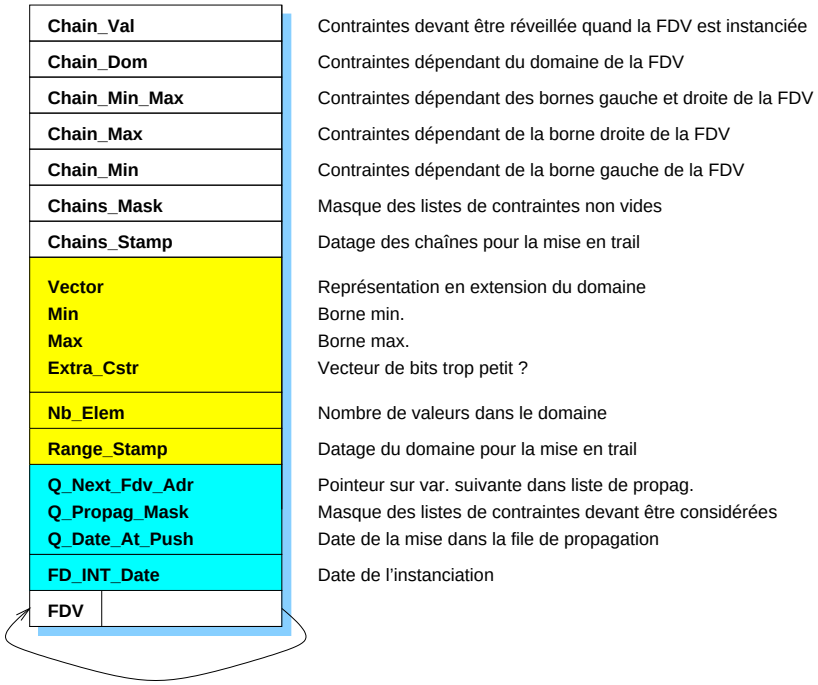


FIG. 8.9: Représentation d'une variable DF dans clp(fd)

Contrairement à l'algorithme Nar (*cf.* alg. 3.1, page 39), l'algorithme de propagation de domaines de clp(fd) ne chaîne pas les contraintes à réinvoquer mais les variables dont le domaine a été modifié, en mettant à jour un masque indiquant les listes de contraintes à réexécuter.

Trois types d'instructions sont ajoutées à la WAM pour gérer les contraintes :

1. des instructions d'interfaçage avec Prolog : création de l'environnement d'une clause ;
2. des instructions d'installation des contraintes : création de la structure de représentation d'une contrainte et initialisation des listes de dépendances ;
3. des instructions d'exécution des contraintes, *i.e.* :
   – chargement des paramètres nécessaires à l'évaluation de R dans des registres,
   – évaluation du domaine R : l'évaluation se fait par un parcours de l'arbre syntaxique de R,
   – mise à jour de la variable X.

Instructions d'interfaçage

`fd_set_AF(nb_arg,`V_i`)`
réserve l'espace dans le tas pour un environnement de `nb_arg` arguments. Le registre AF et la variable V_i pointent sur cet environnement ;

`fd_variable_in_A_frame(`V_i`)`
lie V_i à une variable DF créée sur le tas (de domaine $0..+\infty$) et range son adresse dans le mot pointé par AF. Le registre AF est alors incrémenté ;

`fd_value_in_A_frame(`V_i`)`
si la valeur déréférencée w de V_i est :
– une variable libre : similaire à `fd_variable_in_A_frame(`w`)`,
– un entier : il est empilé sur le tas sous forme de variable DF et son adresse est rangée dans le mot pointé par AF. Le registre AF est alors incrémenté ;
– une variable DF : son adresse est rangée dans le mot pointé par AF, qui est alors incrémenté.

`fd_range_parameter_in_A_frame(`V_i`)`
Le registre V_j doit être lié à une liste d'entiers et le domaine correspondant est créé sur le tas. L'adresse de ce domaine est copiée dans le mot pointé par AF. Le registre AF est alors incrémenté ;

`fd_term_parameter_in_A_frame(`V_i`)`
Le registre V_i doit être un entier, qui est rangé dans le mot pointé par AF. Le registre AF est alors incrémenté ;

`fd_install_constraint(install_proc,`V_i`)`
réinitialise AF avec le contenu de V_i et CC avec l'instruction suivante avant de donner le contrôle au code d'adresse `install_proc` ;

`fd_call_constraint`
initialise CC avec l'instruction suivante et donne le contrôle au code d'exécution de la contrainte pointée par CF.

Instructions d'installation des contraintes

`fd_create_C_frame(constraint_proc,tell_fv)`
crée sur le tas une structure pour la contrainte dont le code d'exécution se trouve à l'adresse indiquée par `constraint_proc` et dont la variable contrainte est la `tell_fv`^{e}. Le registre CF pointe sur cette structure ;

`fd_install_`$\left\{\begin{array}{l}\texttt{ind\_min}\\ \texttt{ind\_max}\\ \texttt{ind\_min\_max}\\ \texttt{ind\_dom}\\ \texttt{ind\_val}\end{array}\right\}$**`(fv)`**

Ces instructions sont utilisées quand la contrainte pointée par CF utilise le $\min$ (ou le $\max$...) de la `fv`^{e} variable. Un nouvel élément est ajouté à la liste correspondant à la `fv`^{e} variable ;

`fd_proceed`
rend le contrôle à l'adresse pointée par CC.

Instructions d'exécution des contraintes

`fd_range_parameter(R[r],fp)`
charge le domaine pointé par le `fp`^{e} paramètre dans le registre `R[r]` ;

`fd_term_parameter(T[t],fp)`
charge la valeur du `fp`^{e} paramètre dans `T[t]` ;

`fd_ind_`$\left\{\begin{array}{l}\texttt{min}\\ \texttt{max}\end{array}\right\}$**`(T[t],fp)`**
charge le min (ou le max) de la `fv`^{e} variable dans `T[t]` ;

`fd_ind_min_max(T[tmin],T[tmax],fv)`
charge le min et max de la `fv`^{e} variable dans `T[min]` et `T[tmax]` ;

`fd_ind_dom(R[r],fv)`
charge le domaine de la `fv`^{e} variable dans `R[r]` ;

`fd_dly_val(T[t],fv,lab_else)`
si la `fv`^{e} variable est un entier, sa valeur est copiée dans `T[t]`, sinon, le contrôle est donné au code d'étiquette `lab_else`.

`fd_interval_range(R[r],T[tmin],T[tmax])` exécute `R[r]` $\leftarrow$ `T[tmin]..T[tmax]` ;

`fd_`$\left\{\begin{array}{l}\texttt{union}\\ \texttt{inter}\end{array}\right\}$**`(R[r],R[r1])`** exécute `R[r]` $\leftarrow$ `R[r]` $\left\{\begin{array}{l}\cup\\ \cap\end{array}\right\}$ `R[r1]` ;

`fd_compl(R[r])` exécute `R[r]` $\leftarrow 0..\infty \setminus$ `R[r]` ;

`fd_compl_of_singleton(R[r],T[t]))` exécute `R[r]` $\leftarrow 0..\infty \setminus\{$`T[t]`$\}$;

fd_range_{add, sub, mul, div}_term(R[r],T[t]) exécute R[r] ← R[r] {+, −, ×, /} T[t] ;

fd_range_{add, sub, mul, div}_range(R[r1],R[r2]) exécute R[r1] ← R[r1] {+, −, ×, /} R[r2] ;

fd_range_copy(R[r],R[r1]) exécute R[r] ← R[r1] ;

fd_integer(T[t],n) exécute T[t] ← n ;

fd_term_{add, sub, mul, div, floor_div, ceil_div}_term(T[t],T[t1]) exécute T[t] ← T[t] {+, −, ×, /, ⌊/⌋, ⌈/⌉} T[t1] ;

fd_term_copy(T[t],T[t1]) exécute T[t] ← T[t1] ;

fd_tell_range(R[r]) ajoute la contrainte X *in* R quand R n'est pas un intervalle ;

fd_tell_interval(T[tmin],T[tmax]) ajoute la contrainte X *in* T[tmin]..T[tmax].

Nous donnons ci-dessous un exemple [67] de compilation pour la clause :

```
'x=y+c'(X,Y,C):-
        X in min(Y) + C .. max(Y) + C,
        Y in min(X) - C .. max(X) - C.
```

On a :

```
'x=y+c':  fd_set_AF(3,X(3))                        % 3 elements :  X,Y,C
          fd_value_in_A_frame(X(0))                % X est fv(0)
          fd_value_in_A_frame(X(1))                % Y est fv(1)
          fd_term_parameter_in_A_frame(X(2))       % C est fp(2)
          fd_install_constraint(inst_1,X(3))       % installe cstr_1
          fd_call_constraint                       % execute cstr_1
          fd_install_constraint(inst_2,X(3))       % installe cstr_2
          fd_call_constraint                       % execute cstr_2
          proceed                                  % retour Prolog

inst_1:   fd_create_C_frame(cstr_1,0)
          fd_install_ind_min_max(fv(1))            % utilise min/max(Y)
          fd_proceed                               % retour d'installation

cstr_1:   fd_ind_min_max(T(0),T(1),fv(1))          % min(Y) et max(Y)
          fd_term_parameter(T(2),fp(2))            % C
          fd_term_add_term(T(0),T(2))              % min(Y)+C
          fd_term_add_term(T(1),T(2))              % max(Y)+C
          fd_tell_interval(T(0),T(1))              % X in min(Y)+C..max(Y)+C
          fd_proceed                               % retour d'execution

inst_2:   (...)
```

8.5 Implémentation de DecLIC

Ainsi qu'il a été évoqué précédemment, toutes les contraintes « primitives » de clp(fd) sont exprimées en terme de conjonction de contraintes de la forme `X in R`. Comme le soulignent CODOGNET et DIAZ, l'un des avantages de cette approche est que toute optimisation du traitement de la contrainte `X in R` profite à toutes les autres contraintes. De plus, le fait de n'avoir à traiter qu'une seule forme de contrainte au niveau du noyau du résolveur permet de définir une méthode de propagation très spécialisée, donc efficace. L'un des inconvénients de la méthode est qu'il est peut être difficile d'ajouter des contraintes n'étant pas traduites en termes de primitives, comme c'est le cas pour les contraintes gérées par l'algorithme de calcul de la box-consistance. Cela ne s'avère cependant pas un problème dans ce cas précis. En effet, les contraintes pour lesquelles on calcule la box-consistance sont « décomposées » en contraintes unaires (les projections sur chaque variable) qui sont aisément traduisibles dans le schéma `X in R`. Par conséquent, l'utilisation de la seule contrainte primitive `X in R` s'avère un bon choix pour intégrer les algorithmes de calcul de la hull-consistance et de la box-consistance dans un même cadre de travail.

Nous avons donc choisi de conserver l'*approche RISC* [50] de clp(fd) pour DecLIC. Nous avons cependant été amenés à restreindre la grammaire de la contrainte `X in R` de clp(fd) (*cf.* table 8.3) aux deux formes suivantes :

`X in g..h`	avec $\mathsf{g}, \mathsf{h} \in \mathbb{F}$
`X in f(Y1,...,Yn)`	avec $f : \mathbb{I}^n \to \mathbb{I}$

Pour des raisons d'efficacité, nous avons décidé de restreindre les domaines des variables à des intervalles (à bornes ouvertes ou fermées) et de ne plus gérer les unions d'intervalles. De plus, il n'est pas possible d'autoriser le programmeur à définir un intervalle par des expressions pour chaque borne dès lors que l'on travaille avec des nombres flottants, car il est nécessaire de garantir que l'intervalle flottant résultant est un sur-ensemble de l'intervalle réel correspondant.

Exemple 8.22 (Calcul d'un intervalle par des expressions pour les bornes). *Considérons une contrainte de la forme `X in 4*min(Y)-max(Z)..3*A-9/B`. Pour être correct, nous devons contraindre la variable `X` à prendre sa valeur dans l'intervalle* $\left(\lfloor\!\lfloor\langle 4 * min(Y) - max(Z), [\,\rangle\rfloor\!\rfloor, \lceil\!\lceil\langle 3 * A - 9/B,]\,\rangle\rceil\!\rceil\right)$ *où les quantités arrondies sont calculées en précision infinie. Dans la pratique, on calculera les expressions `4*min(Y)-max(Z)` et `3*A-9/B` en précision finie et il n'est alors pas suffisant de positionner le sens d'arrondi une seule fois avant chaque calcul pour obtenir un arrondi dans le bon sens*[††].

8.5.1 Le modèle de calcul

Dans la suite, nous appellerons *D-contrainte* une contrainte décomposée en contraintes primitives (addition, multiplication) pour lesquelles on cherche à obtenir la hull-consistance, et *N-contrainte* une contrainte pour laquelle on calcule la box-consistance.

L'algorithme de propagation de modification de domaines de DecLIC est une variante de l'algorithme HC3 (*cf.* page 43) gérant trois ensembles distincts de contraintes :

1. un ensemble S_1 contenant toutes les contraintes `is_integer` et `is_fd` ;
2. un ensemble S_2 contenant toutes les contraintes pour lesquelles on calcule la box-consistance ;
3. un ensemble S_3 contenant toutes les contraintes (en formes décomposées) pour lesquelles on calcule la hull-consistance ;

Exemple 8.23 (Un *store* dans DecLIC). *Étant donné le programme :*

```
:- is_integer(X), is_fd(Y),
   X+Y**2 $= Z, Z $< 6,
   3*X**3-4*X**2+2*X-7 $$= Y.
```

on a le store de la figure 8.10.

[††] Une solution à ce problème est de calculer les deux expressions en utilisant l'arithmétique des intervalles puis de prendre la borne gauche de l'intervalle résultant du calcul de l'expression `4*min(Y)-max(Z)` et la borne droite de l'évaluation de `3*A-9/B` pour créer un intervalle englobant avec certitude l'intervalle ci-dessus.

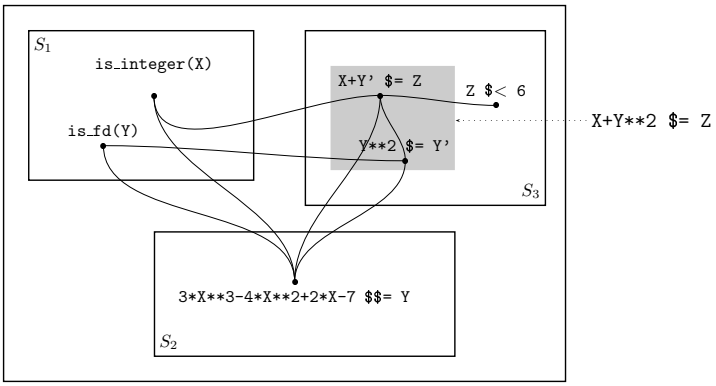


FIG. 8.10: *Store* pour le programme de l'exemple 8.23

L'addition d'une nouvelle contrainte dans le *store* est faite de la façon décrite par l'algorithme 8.1. On notera que la stratégie adoptée réinvoque les contraintes `is_integer` et `is_fd` le plus tôt possible et que les contraintes non-décomposées sont reconsidérées avant les contraintes décomposées sur lesquelles on calcule la hull-consistance.

8.5.2 Les structures de données

Nous allons maintenant décrire les structures de données utilisées par DecLIC pour gérer les contraintes. Nous ne présenterons ici que les structures provenant de clp(fd) mais ayant été modifiées, ainsi que les structures propres à DecLIC. Nous renvoyons le lecteur à la section 8.4.6 pour la description des structures communes à clp(fd) et DecLIC (structure d'arguments — fig. 8.7 — et structure pour une D-contrainte — fig. 8.8).

Listes de propagation L'algorithme IncNar (*cf.* alg. 8.1) utilise trois ensembles, CS_1, CS_2, CS_3, pour stocker les contraintes à réinvoquer. Comme dans la plupart des systèmes de résolution de contraintes, DecLIC gère ces ensembles comme des queues (stratégie FIFO). Dans la pratique, DecLIC utilise seulement deux listes — l'ensemble CS_1 n'est pas représenté car les contraintes `is_integer` et `is_fd` sont réinvoquées immédiatement après qu'une variable a été modifiée :

la liste de propagation globale (GPL) est une queue de propagation chaînant les structures des variables dont le domaine a été réduit ;

la liste de propagation Newton (NPL) est une queue chaînant les structures des N-contraintes devant être reconsidérées.

Des pointeurs sur la tête et la queue de chacune de ces contraintes sont conservées dans des registres spécifiques (*cf.* table 8.4).

Structures pour les contraintes Comme dans clp(fd), on associe à chaque contrainte une structure rassemblant les informations nécessaires lors de sa réinvocation. DecLIC manipule deux types de structures pour les contraintes : les structures pour les D-contraintes, analogues à celles utilisées par clp(fd) ; et les structures pour les N-contraintes (*cf.* fig. 8.11). La différence fondamentale entre ces deux types de structures est due au fait qu'à chaque contrainte X *in* R correspond une fonction C spécifique, alors que toutes les N-contraintes sont gérées par la même fonction C intégrée dans le code de DecLIC. Les structures de N-contraintes contiennent les informations requises par cette fonction.

Structure de représentation d'une variable On crée une structure de la forme décrite par la figure 8.12 sur le tas pour chaque variable apparaissant dans une contrainte. Cette structure est une version légèrement modifiée de celle utilisée par clp(fd) (*cf.* fig. 8.9, p. 116). Elle se divise en trois parties :

ALG. 8.1: Algorithme de propagation de DecLIC

IncNar(**entrée**: (c, N) *% Contrainte à ajouter*
entrée/sortie: S_1, S_2, S_3 *% Les 3 sous-stores*
entrée/sortie: $\boldsymbol{B} = I_1 \times \cdots \times I_n$) *% Domaines des variables*
début
pour chaque $i \in \{1, 2, 3\}$ **faire**
$CS_i \leftarrow \varnothing$ *% Ens. des contraintes de S_i à réinvoquer*
fpc
$\alpha \leftarrow$ typeof(c) *% α = 1, 2 ou 3 suivant le « type »*
% de c : `is_integer()`, `f$..` ou `f$$..`
$CS_\alpha \leftarrow \{(c, N)\}$ *% c doit être invoquée dans le store S_α*
$S_\alpha \leftarrow S_\alpha \cup \{(c, N)\}$ *% c est ajoutée dans le store correspondant à son type*
tant que $((\exists i \in \{1, 2, 3\} \mid CS_i \neq \varnothing)$ **et** $(\forall i' \in \{1, \ldots, n\} : I_{i'} \neq \varnothing))$ **faire**
% On boucle tant qu'il existe des contraintes à reconsidérer dans au moins l'un des sous-stores
$j \leftarrow \min(\{i \in \{1, 2, 3\} \mid CS_i \neq \varnothing\})$
Choisir un (c_l, N_l) dans CS_j
% Choix d'une contrainte à réinvoquer parmi celles
% du sous-store de plus petit index (plus grande priorité)
$I' \leftarrow N_l(I)$
si $(I' \neq I)$ **alors**
pour chaque $k \in \{1, 2, 3\}$ **faire**
$CS_k \leftarrow CS_k \cup \{(c_m, N_m) \in S_k \mid \exists x_o \in \mathsf{Var}(()c_m) \wedge I'_o \neq I_o\}$
% Toutes les contraintes ayant une occurrence de
% variable dont le domaine a été réduit sont mises
% dans la queue correspondant à leur type
fpc
$I \leftarrow I'$
finsi
$CS_j \leftarrow CS_j \backslash \{(c_l, N_l)\}$
ftq
fin

TAB. 8.4: Registres de DecLIC

Nom	*Usage*
AF	pointeur sur l'environnement de contraintes courant
BP	pointeur sur la tête de la GPL (liste de propagation globale)
TP	pointeur sur la queue de la GPL
CF	pointeur sur la structure de contrainte courante
BNP	pointeur sur la tête de la NPL (Liste de propagation Newton)
ENP	pointeur sur la queue de la NPL

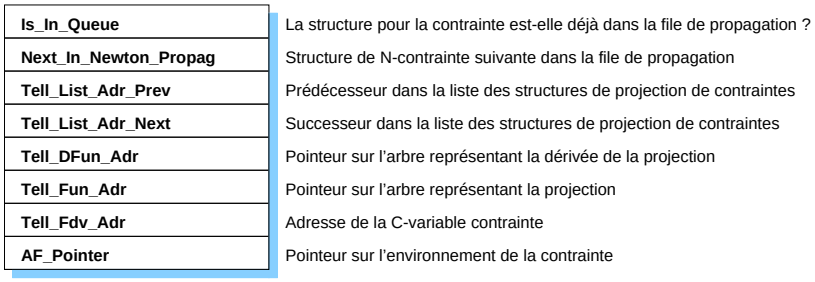


FIG. 8.11: Structure pour une N-contrainte

1. une partie dévolue à la gestion de la propagation (effectuée par mise en queue des structures des variables dont le domaine a été réduit) ;
2. une partie contenant les informations sur le domaine : les bornes gauche et droite de l'intervalle (représentées par des `double`s IEEE [116]) et un mot codant la forme des *brackets* gauche et droite, ainsi que le type de la variable (i-variable ou fd-variable). Cette dernière information est consultée à chaque fois que le domaine de la variable est modifié. S'il s'agit d'une fd-variable, son domaine est arrondi « vers l'intérieur » aux entiers les plus proches avant que la phase de propagation n'ait lieu. Comme dans clp(fd), une étiquette (`Range_Stamp`) est utilisée pour éviter la mise en *trail* du domaine à chacune de ses modifications). La partie de la structure codant les informations liées au domaine est mise en *trail* au plus une fois par point de choix (technique due à AGGOUN et BELDICEANU [4]) : un compteur global est incrémenté à la création de chaque point de choix et décrémenté lors de sa destruction. La mise en *trail* est faite uniquement lorsque `Range_Stamp` (valeur du compteur globale lors de la dernière mise en *trail*) est différent de la valeur courante du compteur global ;
3. une dernière partie contient les chaînes de dépendance liant les structures des contraintes à réinvoquer lors d'une modification du domaine de la variable. Cette partie possède son propre compteur de mise en *trail* utilisant la technique décrite au point précédent.

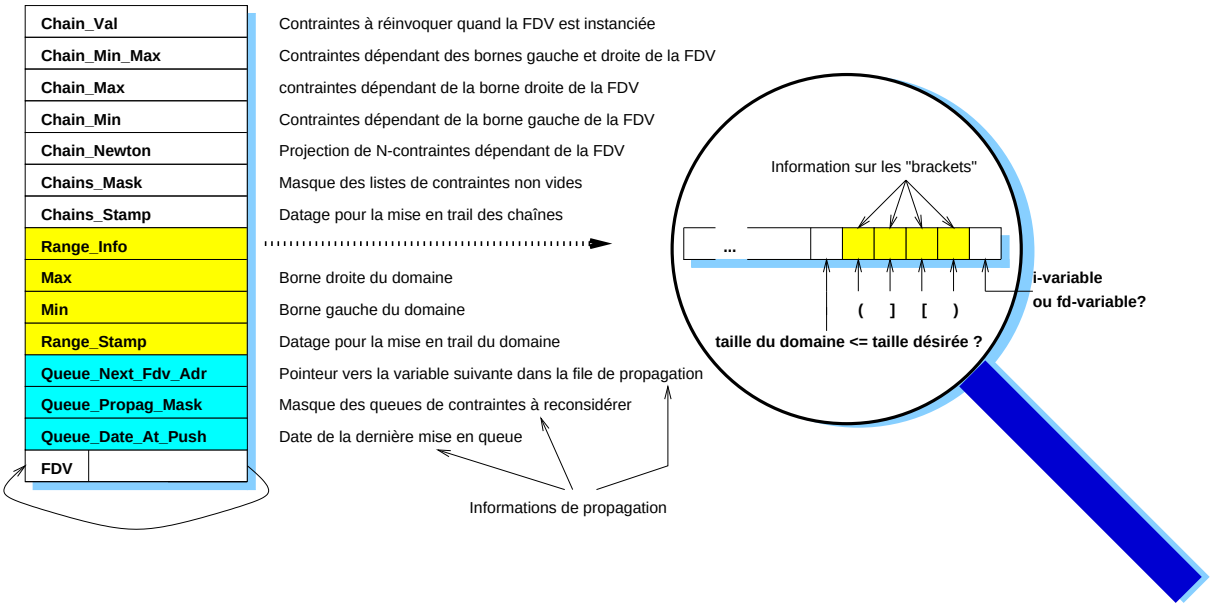


FIG. 8.12: Structure pour une variable

8.5.3 Compilation de la contrainte `X in R`

Comme on l'a dit précédemment, chaque contrainte « primitive » de DecLIC est définie en terme de conjonctions de contraintes de la forme `X in R`, où l'utilisation des indexicaux est restreinte à `dom()` — qui correspond ici à des domaines connexes uniquement. On choisit d'abandonner l'utilisation des indexicaux `min()` et `max()` puisqu'il devient impossible de créer des domaines à partir d'expressions arithmétiques définissant les bornes[‡‡].

La partie `R` d'une contrainte `X in R` est compilée en une fonction contenant une suite d'instructions WAM réinvoquée à chaque fois que le domaine d'une des variables ayant une occurrence dans `R` est modifié.

Exemple 8.24 (Compilation de la contrainte `X in R`). *La contrainte `X in dom(Y)+(o(3.4)..7*3)` (% $X \in$ `dom(Y)` $\cup(3{,}4\ ..\ 21]$) est compilée en la suite d'instructions :*

```
Begin_Fd_Constraint(1)
      fd_ind_dom(0,1)  % R(0) <- dom(1ère var.) dans structure argument (Y)
      fd_rounding_down % arrondi vers -infini
      fd_float(2,3.4)  % T(2) <- 3.4
      fd_rounding_up   % arrondi vers +infini
      fd_integer(1,7)  % T(1) <- 7
      fd_integer(3,3)  % T(3) <- 3
      fd_term_mul_term(1,3) % T(1) <- T(1)*T(3)
      fd_interval_range(1,2,1,16,8) % R(1) <- o(T2)..T(1)
      fd_range_add_range(0,1) %^-^-- 16=ouvert à gauche, 8=fermé à droite
      fd_tell_range(0) % X <- dom(X) intersecté avec R(0) + propagation
      fd_proceed
End_Fd_Constraint
```

L'instruction `fd_tell_range` intersecte le domaine de la variable courante (`Tell_Fdv_Adr` dans la structure de contrainte pointée par `CF` ; voir la table 8.4) avec le domaine contenu dans le registre passé en paramètre. Elle pousse aussi la structure de la variable dans la queue de propagation si son domaine a été modifié.

8.5.4 Gestion des contraintes lors de la compilation

Lors de la compilation d'un programme DecLIC, chaque D-contrainte est décomposée en une suite d'appels aux primitives de la librairie de DecLIC.

Exemple 8.25 (compilation d'une D-contrainte). *Le système de contrainte :*

```
:- X**2 + Y**2 $= 1,
   X**2 - Y    $= 0.
```

est compilé en la suite d'instructions WAM donnée dans la table 8.5. On remarquera que la décomposition se fait localement à chaque contrainte, ce qui implique que les sous-expressions communes ne sont pas partagées. L'instruction `call(a,b)` appelle une fonction C `b`-aire contenant le code chargé de réduire le domaine d'une variable pour que la contrainte `a` soit vérifiée.

TAB. 8.5: Décomposition et compilation d'une D-contrainte

```
put_y_variable(4,0) % A(0) <- X
put_y_variable(8,1)
call("xx=y",2)      % Y(8) $= X**2
put_y_variable(3,0) % A(1) <- Y
put_y_variable(7,1)
call("xx=y",2)      % Y(7) $= Y**2
put_y_value(8,0)    % A(0) <- X**2
put_y_value(7,1)    % A(1) <- Y**2
put_integer(1,2)    % A(2) <- 1
```

```
call("x+y=z",3)   % X**2 + Y**2 = 1
put_y_value(4,0)  % A(0) <- X
put_y_variable(2,1)
call("xx=y",2)    % Y(2) $= X**2
put_y_value(2,0)  % A(0) <- X**2
put_integer(0,1)  % A(1) <- 0
put_y_value(3,2)  % A(2) <- Y
call("x+y=z",3)   % X**2 + 0 $= Y
```

[‡‡] En fait, la librairie de DecLIC fait un grand usage des indexicaux `min()` et `max()` mais restreint leur utilisation aux cas simples ne pouvant entraîner des problèmes liés aux arrondis.

Lors de la compilation, une N-contrainte de la forme `f$$◇0` (avec $\diamond \in \{=, <=, \dots\}$) est traduite en une chaîne de caractère représentant `f` en forme préfixée. Par exemple, la contrainte `p(X,Y):- X**2+Y**2$$=1.` est traduite en

```
"-(+(^(#(0),i(2)),^(#(1),i(2))),i(1))"
```

où `^` correspond à la puissance, `i(n)` code l'entier `n` et `#(m)` représente la $\texttt{m}^{\text{e}}$ variable dans la structure d'argument de la clause.

On génère une *fonction d'installation* (*cf.* section 8.5.5) pour chaque N-contrainte du programme. Les N-contraintes reliées entre elles par un connecteur `and` (*e.g.* prog. 8.2) partagent la même fonction d'installation. La conséquence de ce partage est que ces N-contraintes sont mises en une seule opération dans le *store* et l'algorithme de propagation est invoqué une seule fois, ce qui permet dans certains cas d'accélérer le processus de résolution (on atteint expérimentalement des accélérations d'un facteur 10 pour certains problèmes).

Exemple 8.26 (Compilation des N-contraintes). *Le système composé des deux N-contraintes :*

```
:- X**2 + Y**2 $$= 1 and
   X**2 - Y    $$= 0.
```

est compilé en :

```
Begin_Private_Pred
 fd_set(2,0)     % Création d'une structure d'arguments pour X et Y
 init_newton_system   % Newton propag list <- vide
 fd_install_newton_constraint(1,0) % appelle Fd_install_Newton(1) pour la
                                   % 0ième variable (i.e. X)
 fd_call_newton_constraints % propagation Newton  + propagation générale
 ...
End_Pred

Begin_Fd_Install_Newton(1)
 fd_install_newton("-(+(^(#(0),i(2)),^(#(1),i(2))),i(1))")
 fd_install_newton("-(^(#(0),i(2)),#(1))")
 fd_proceed
End_Fd_Install_Newton
```

La figure 8.13 présente un exemple de réseau de contraintes créé à l'exécution pour la résolution du système ci-dessus.

8.5.5 Gestion des contraintes à l'exécution

À l'exécution, l'installation des N-contraintes par l'instruction `fd_install_newton` se fait de la façon suivante. Soient n N-contraintes $c_1, \dots, c_n$ définies par :

$$\begin{cases} c_1 : & f_1(x_{1_1}, \dots, x_{m_1}) & = 0 \\ & \ddots & \\ c_n : & f_n(x_{1_n}, \dots, x_{m_n}) & = 0 \end{cases}$$

Considérons la N-contrainte $c_i : f_i(x_{1_i}, \dots, x_{m_i}) = 0$. Pour chaque projection de c_i par rapport à la variable x_{j_i} :

- on crée une structure de N-contrainte $NF_{i,j}$ (cette structure sera utilisée pour représenter l'opérateur associé à la contrainte c_i réduisant le domaine de x_{j_i}) ;
- on calcule symboliquement la dérivée partielle $\partial f_i / \partial x_j$ et on insère un pointeur sur son expression dans la structure $NF_{i,j}$;
- on chaîne la structure $NF_{i,j}$ dans la liste `Chain_Newton` des structures des variables $x_{1_i}, \dots, x_{m_i}$ (car le calcul de la projection associée à $NF_{i,j}$ dépend des domaines de ces variables) ;
- on ajoute $NF_{i,j}$ dans la NPL.

Après installation des N-contraintes, on lance l'étape de propagation décrite par l'algorithme 8.2 en invoquant l'instruction `fd_call_newton_constraints`.

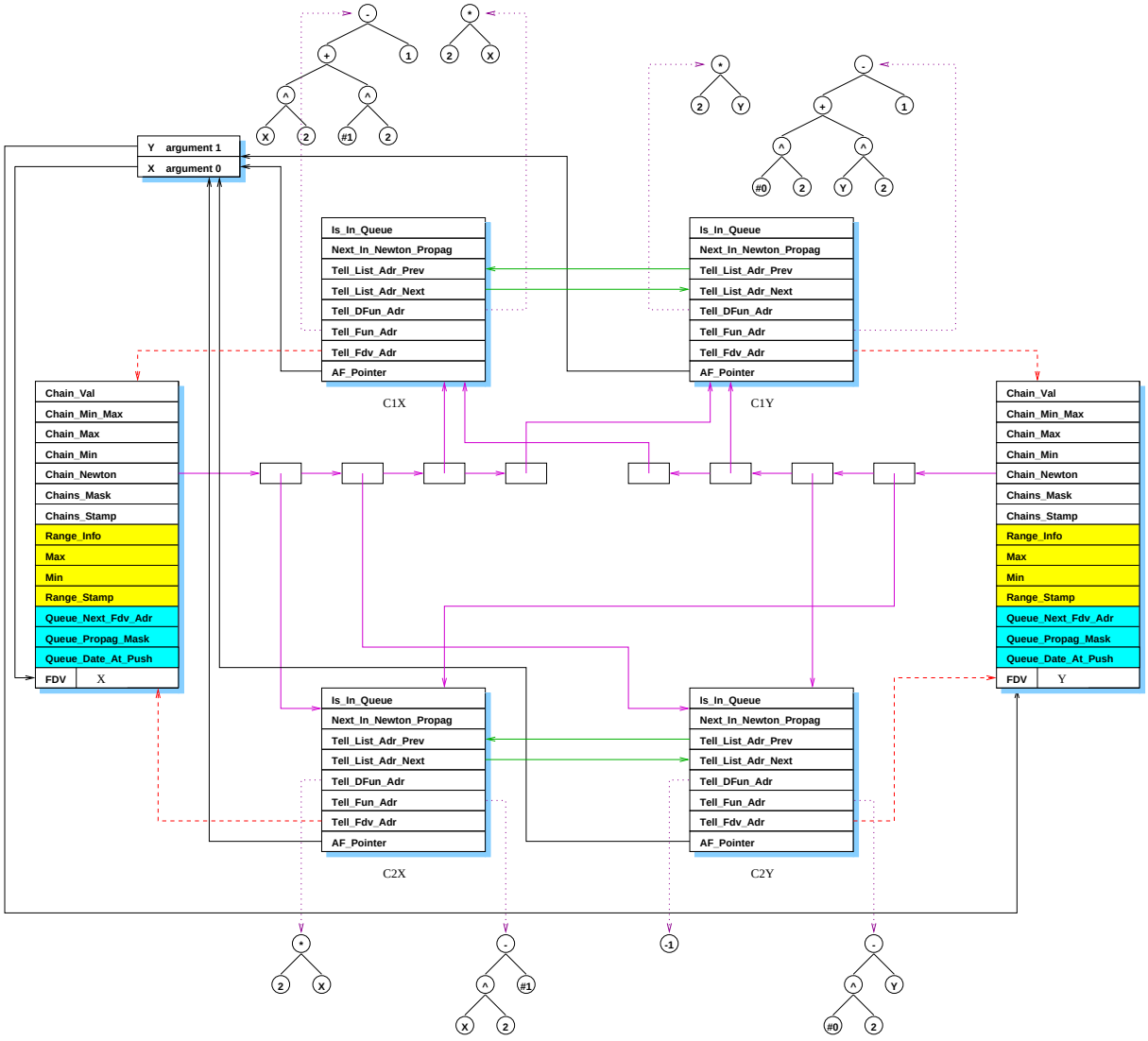


FIG. 8.13: Réseau de contraintes pour l'exemple 8.26

ALG. 8.2: Propagation dans la NPL

```
NewtonNar(entrée: NPL   % Liste chaînée des structures de N-contraintes à réinvoquer
          sortie: échec ou succès)
  début
    c ← pop(NPL)
    répéter
      x ← c.Tell_Fdv_Adr
      I'_x ← NewtonNarrow(c, x)
      si (I'_x = ∅) alors
        Détruire NPL
        retourner (échec)
      sinon
        si (I'_x ≠ I_x) alors
          PushInGPL(x.frame)
          PushInNPL({N-contraintes dépendant de x})
        finsi
      finsi
      c ← pop(NPL)
    jusqu'à (NPL = ∅)
    retourner (succès)
  fin
```

8.5.6 Interactions entre les algorithmes de résolution

Les algorithmes de résolution calculant la hull-consistance et la box-consistance interagissent par l'intermédiaire des domaines des variables partagées par les N-contraintes et les D-contraintes. Comme on l'a dit précédemment, la propagation se fait en considérant les N-contraintes avant les D-contraintes. La modification du domaine d'une variable x à la suite de l'application de l'opérateur de contraction d'une contrainte c entraîne :

- l'ajout de la structure associée à la variable à la fin de la GPL ;
- la mise à jour du masque de propagation `Queue_Propag_Mask` de x en fonction du type de modification du domaine (changement de valeur de la borne gauche et/ou droite). On notera que si la contrainte c est une N-contrainte, on ne marque pas `Chain_Newton` comme devant être reconsidéré, puisque la propagation dans la NPL est faite immédiatement. De plus, si x était déjà dans la GPL, on interdit la réinvocation des contraintes chaînée dans `Chain_Newton` en modifiant le masque `Queue_Propag_Mask` ;
- on effectue la propagation de la GPL en commençant par extraire la première structure vf de variable de cette liste, puis :
  1. on chaîne toutes les structures de N-contraintes se trouvant dans `Chain_Newton` dans la NPL et on applique l'algorithme 8.2,
  2. on réinvoque les contraintes se trouvant dans les autres listes de dépendances de vf (ce qui peut ajouter de nouvelles variables dans la GPL),
  3. on extrait la structure de variable suivante de la GPL et on recommence tout le processus à partir du point 1 jusqu'à atteindre un état stable (échec ou GPL vide).

8.5.7 Extension de la WAM pour DecLIC

La compilation d'un programme DecLIC supposait la définition de nouvelles instructions WAM. Nous les décrivons succinctement ci-dessous.

`fd_rounding_`$\left\{ \begin{array}{l} \textbf{up} \\ \textbf{down} \end{array} \right\}$**.** Change le mode d'arrondi courant dans la compilation d'une contrainte `X` *`in`* `R`

(*cf.* exemple 8.24) ;

`fd_float(T,n).` Affecte la valeur `n` au registre `T`. Utilisé lors du calcul d'un domaine dans une contrainte `X` *in* `R` ;

`fd_infinity(T).` Affecte la valeur $+\infty$ (au sens IEEE 754) au registre `T` ;

`init_newton_system.` Fait pointer le registre `ENP` sur le début de la NPL matérialisé par une structure de N-contrainte vide ;

`fd_install_newton_constraint(nb,X).` Initialise le pointeur de structure d'arguments à `X` puis appelle la `nb`^{e} fonction d'installation de contrainte ;

`fd_install_newton(str).` Installe la N-contrainte représentée par la chaîne de caractères `str` : crée l'arbre représentant la contrainte codée par `str` (dérive aussi symboliquement la contrainte par rapport à chaque variable). Crée une structure de N-contrainte pour chaque projection et les ajoute dans la liste `Chain_Newton` de la variable pointée par le registre `AF` ainsi que dans la NPL ;

`fd_call_newton_constraints.` Reconsidère tour à tour chaque contrainte se trouvant dans la NPL, puis lance la propagation dans la GPL.

CHAPITRE 9

Le débogage en programmation par contraintes : état de l'art

Présentation de Grace — Oz explorer — le débogueur de CHIP — le débogueur CNV

LA NOTION DE DÉBOGAGE est indissociable de celle d'*erreur*. On peut considérer comme erroné un programme dont les résultats sont différents qualitativement (solutions différentes de celles attendues, solutions fausses), ou quantitativement (absence de certaines solutions, plus de résultats qu'attendu), de ceux attendus (*débogage de correction*) ; on peut aussi s'intéresser à un programme dont le temps de calcul est jugé trop long (*débogage de performance*). Les débogueurs que nous allons présenter ci-dessous s'intéressent aux deux aspects du débogage et sont tous des outils de visualisation des structures liées à la partie « purement contraintes » des programmes. Il existe bien sûr d'autres outils permettant, soit le débogage au niveau des instructions du langage hôte (par exemple, gdb [219] pour un langages impératif de type C et Opium [72] pour un langage logique comme Prolog), soit un débogage des contraintes par des méthodes non visuelles (ajout d'assertions [193], diagnostic déclaratif [228]...). Cependant, ce chapitre ayant pour but de mettre en perspective notre propre système de débogage par visualisation présenté au chapitre 10, nous n'aborderons pas ces méthodes et renvoyons pour cela le lecteur au livre issu du projet ESPRIT *DiSCiPl* [64].

Nous avons choisi de présenter ici quatre outils de visualisation pour le débogage en programmation par contraintes. Chacun d'eux offre à l'utilisateur une vision différente du processus de résolution :

- Grace [162, 163], le débogueur graphique d'ECL^{i}PS^e [6], affiche les domaines des variables et permet de choisir les stratégies d'énumération ;
- Oz Explorer [207], le débogueur de Oz [216], se focalise sur l'*arbre de recherche* (affichage et contrôle) ;
- le débogueur [214] inclus dans CHIP [69] combine les deux méthodes précédentes ;
- CNV [204, chap. 10], le débogueur pour SkyBlue [205] et Multi-Garnet [206] affiche le *store* de contraintes sous forme d'un graphe.

Comme on va le voir, le choix de la méthode utilisée est en grande partie dicté par le types de contraintes gérées dans le langage sous-jacent.

9.1 Le débogueur Grace

Grace est un débogueur écrit en Tcl/Tk pour le système ECL^{i}PS^e destiné à permettre le débogage de performance de programmes logiques utilisant des contraintes dont les variables ont des valeurs discrètes.

Cet outil se concentre sur la visualisation des domaines des variables et sur l'étape d'énumération : l'utilisateur voit en temps réel les modifications de domaines induites par les réinvocations de contraintes (avec la possibilité d'arrêter la propagation ou de la faire s'effectuer pas-à-pas) ; il dispose aussi d'un historique des états qui lui permet de voir l'impact d'une stratégie d'énumération sur la réduction des domaines (*cf.* figure 9.1) ; enfin, il a la possibilité de modifier interactivement le domaine d'une variable pour voir la propagation qui en résulte.

Le point fort de Grace est la possibilité de choisir parmi de nombreuses stratégies d'énumération (tant pour le choix de la variable à instancier que pour celui de la valeur du domaine à utiliser pour cette instanciation). Par contre, le *store* n'est pas directement visualisable en tant que réseau de contraintes. Il est cependant possible d'afficher la liste des contraintes contenant une occurrence d'une variable particulière (*cf.* figure 9.2).

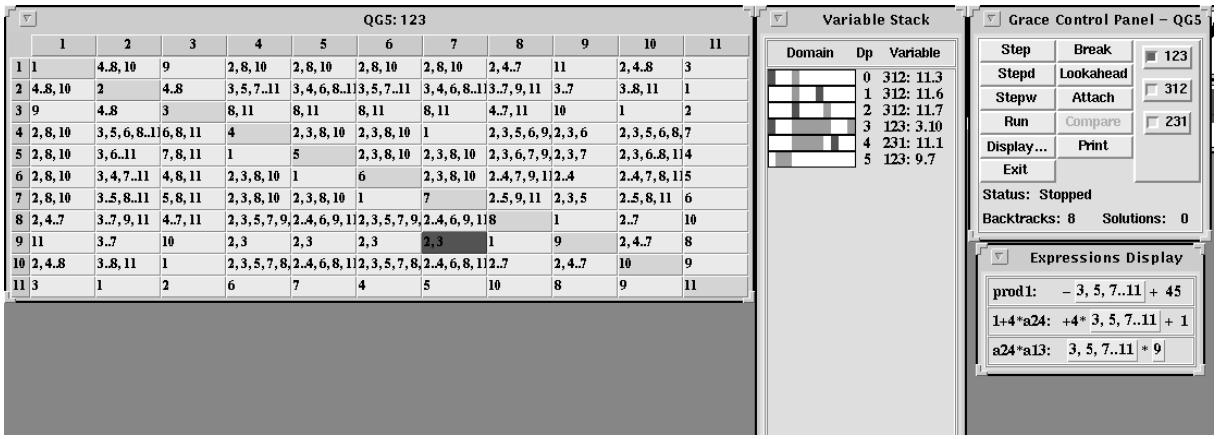


FIG. 9.1: Interface graphique de Grace

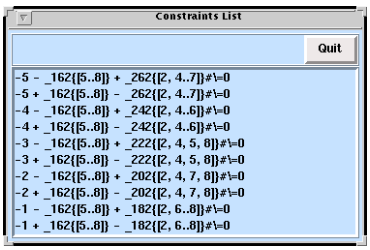


FIG. 9.2: Représentation du *store* dans Grace

Notons que, si l'utilisation de Grace n'est a priori pas limitée à ECL^{i}PS^e, son orientation quasi-exclusive vers la maîtrise de l'énumération le cantonne à des résolveurs de contraintes sur les domaines finis.

9.2 Oz Explorer

Oz Explorer est un outils de débogage fourni avec le système Oz entièrement basé sur la présentation de l'arbre de recherche. Comme Grace, il se destine principalement au débogage de performance. L'arbre de recherche est visualisé graphiquement sous une forme naturelle (*cf.* figure 9.3), ce qui permet d'en observer la forme et d'en déduire la distribution des solutions, ainsi que l'impact d'une stratégie d'énumération sur le nombre de nœuds créés avant de trouver une solution.

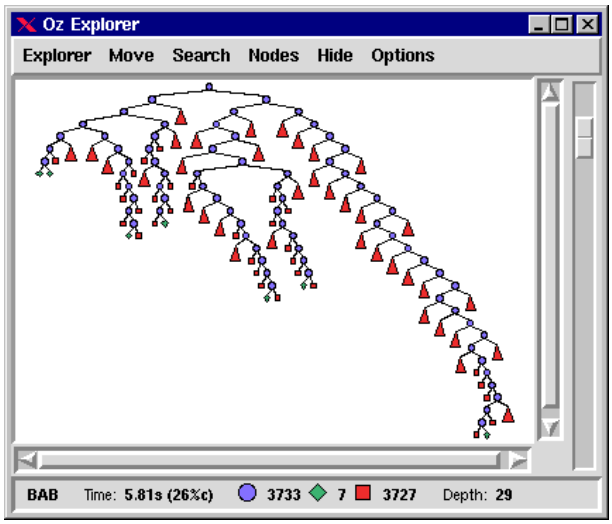


FIG. 9.3: Interface de *Oz Explorer*

L'utilisateur a la possibilité de se déplacer à sa guise dans l'arborescence et d'observer à chaque nœud l'état du *store* (il y a recalcul de son état à partir de nœuds clés, ce qui permet d'éviter de sauvegarder la totalité des informations de propagation).

La visualisation directe de l'arbre de recherche semble rendre plus facile a priori l'observation de l'impact des heuristiques d'énumération sur la vitesse de résolution d'un problème.

Comme son nom l'indique, Oz Explorer est particulièrement lié aux facilités offertes par Oz. Sa portabilité semble donc nulle ou sujette à une réécriture d'une grande partie du système.

9.3 Le débogueur de CHIP

Le débogueur présent dans CHIP réutilise les approches des deux débogueurs vus ci-dessus : comme Oz Explorer, il permet de voir l'arbre de recherche sous une forme arborescente (*cf.* figure 9.5) ; comme Grace, il permet d'afficher les domaines des variables ainsi qu'un historique des modifications des domaines le long d'une branche de l'arbre de recherche.

Grace offrait la possibilité d'afficher la liste des contraintes contenant une certaine variable ; le débogueur de CHIP permet, lui, d'afficher une matrice d'incidence variables/contraintes qui donne de façon synthétique cette information pour toutes les variables et toutes les contraintes. Il s'agit donc d'une représentation compacte du *store* ou le réseau est codé matriciellement. Il est possible d'observer la propagation des modifications de domaines dans cette matrice et donc de déduire l'impact d'une stratégie d'énumération sur la vitesse de résolution et le travail efficace effectué.

Comme Grace et Oz Explorer, le débogueur de CHIP s'intéresse exclusivement à la phase d'énumération dans la résolution d'un problème faisant intervenir des variables à valeur entière.

9.4 Le débogueur CNV

CNV est un débogueur pour les langages Sky-Blue [204] et Multi-Garnet [206]. Il s'applique à des problèmes de génération d'interfaces utilisateur à l'aide de contraintes hiérarchiques [40]. Le débogage s'effectue à partir d'une représentation graphique du *store* sous forme d'un réseau de contraintes (*cf.* figure 9.4). Le réseau contient des boîtes noires (les contraintes de placement et de « force » — *medium*, *weak*...), des boîtes blanches (les variables avec leur valeur) et des liens reliant variables et contraintes. Un lien dirigé d'une contrainte c vers une variable v indique que la valeur actuelle de v est due à l'application de c.

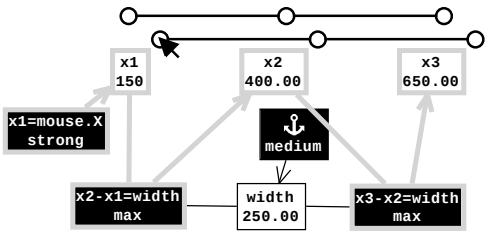


FIG. 9.4: Un store de contraintes dans *Multi-Garnet*

Notons que le modèle de résolution de contraintes sous-jacent n'est pas un modèle de raffinement (réduction de domaines de variables) mais un modèle perturbationnel (à chaque instant, les variables ont une valeur fixée). Aussi, la notion centrale d'arbre de recherche présente dans les trois cas précédents, ainsi que celle de domaine de variable n'a pas de sens ici. La méthodologie de débogage est donc complètement différente de celles vues plus haut. En particulier, on s'intéresse désormais aux propriétés du réseau de contraintes (connexité, présence de cycles...). L'affichage des liens orientés (contrainte → variable) traduit l'information de propagation que l'on obtenait auparavant en observant l'historique des modifications des domaines des variables.

L'affichage d'un réseau de contraintes/variables ne permet pas une représentation aussi compacte que la matrice d'incidence utilisée dans le débogueur de CHIP. Afin de permettre la visualisation de grands graphes, CNV permet de masquer temporairement des contraintes et des variables que l'on ne souhaite pas étudier à un instant donné. On verra au chapitre 10 une traduction de cette idée dans notre débogueur.

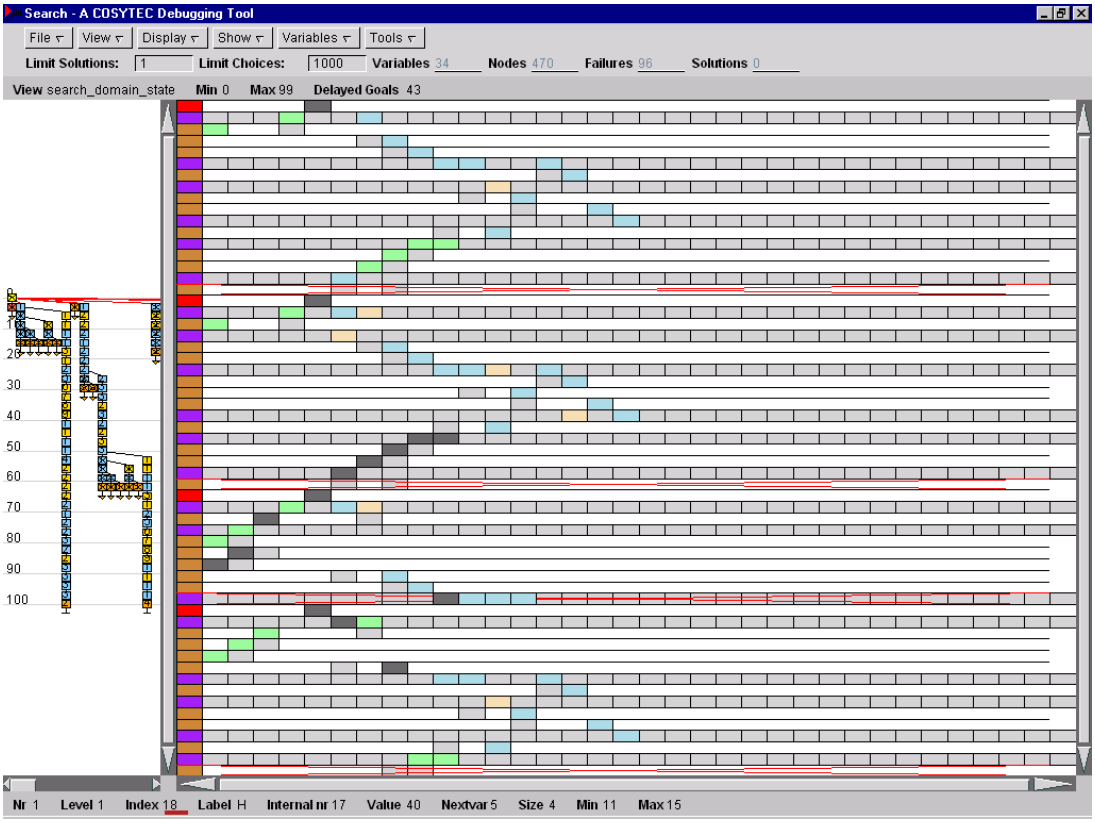


FIG. 9.5: Arbre de recherche et propagation dans CHIP

CHAPITRE 10

Débogage par observation du *store*

Structuration du store guidée par les clauses — les S-boxes — présentation du prototype de débogueur — implémentation du débogueur

Les outils de débogage intégrés dans ECL^{i}PS^e [6] et Oz [216] ne permettent pas de visualiser directement le *store* mais s'intéressent plus particulièrement à l'affichage des domaines des variables en temps réel [162] et à l'aspect de l'arbre de recherche [207]. Ces fonctionnalités sont importantes et utiles. Cependant, nous pensons qu'une description plus directe des relations entre les contraintes du *store* est indispensable pour déboguer des programmes en programmation logique avec contraintes, particulièrement lorsque les variables peuvent prendre des valeurs réelles — cas où les variations de domaines n'apportent que peu d'informations.

On souhaite donc afficher une représentation graphique du *store* sous la forme d'un réseau tel que celui de la figure 3.1, p. 40. Cependant, dessiner le *store* tel qu'il est manipulé par le résolveur de contraintes n'apporterait aucune aide à l'utilisateur. En effet, un *store* est constitué d'un ensemble souvent énorme de contraintes sans aucune structuration, et cela même pour de petits problèmes (*cf.* celui affiché dans la figure 10.7). De plus, les contraintes se trouvant dans le *store* ne sont pas toujours celles données par l'utilisateur (par exemple, un système tel que clp(BNR) [12] décompose les contraintes en primitives en introduisant de nouvelles variables). Il faut donc permettre au programmeur de contrôler la taille et la complexité du *store*.

10.1 Structuration du *store* : exemple

En programmation logique avec contraintes, l'utilisateur structure son programme en le découpant en clauses. Cette structure est perdue lorsque les contraintes sont ajoutées au *store* alors qu'elle véhicule souvent une information utile. Par exemple, considérons le programme P (Prog. 10.1) décrivant un problème de recherche de point de collision entre un mur et une balle.

PROG. 10.1: Le programme P

```
object_A(X,Y,Z):-                                          % Forme du mur
        X ⩽ 0,  Y ⩽ 0,  Z ⩽ 0.
object_B(XCx,YCy,ZCz):-                                    % Forme de la balle
        XCx^2 + YCy^2 + ZCz^2 = 1.
center_B(T,Cx,Cy,Cz):-                                     % Pos. du centre de la balle à T
        T^2 − Cx = 10,
        2T − Cy = 10,
        Cz − T^2 + 7T = 10.
object_B_moving(T,X,Y,Z):-                                 % Pos. du point (X,Y,Z) dans la balle à T
        center_B(T,Cx,Cy,Cz),
        XCx = X − Cx,
        YCy = Y − Cy,
        ZCz = Z − Cz,
        object_B(XCx,YCy,ZCz).
:- T ⩾ 0,  object_A(X,Y,Z),
   object_B_moving(T,X,Y,Z).
```

La clause `center_B(`T`,`C_x`,`C_y`,`C_z`)`, constituée de trois contraintes, définit la position du centre de la balle à un instant T. On pourrait la considérer comme une nouvelle « contrainte globale » de base dont le sens est intuitif. Cependant, l'introduction dans le *store* des contraintes qui composent `center_B()` élimine tout lien privilégié entre elles. La figure 10.1 présente le *store* pour le programme P. Il s'agit déjà d'une abstraction du « vrai » *store* car n'y apparaissent que les contraintes de l'utilisateur. Pourtant, l'ensemble est déjà trop compliqué pour que l'on puisse en tirer quelque chose.

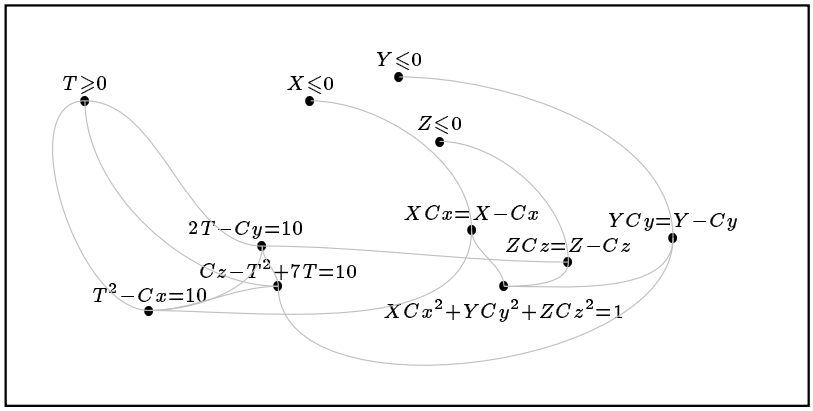


FIG. 10.1: *Store* de P avec les contraintes de l'utilisateur

Une première approche permettant de diminuer la complexité du *store* est d'y réintroduire la structure induite par les clauses du programme. La figure 10.2 montre le *store* de P où chaque clause est traduite par une boîte (plus précisément, le comportement à l'exécution est le suivant : une boîte est créée à chaque fois que le programme « entre » dans une clause ; les contraintes qui sont ajoutées au *store* sont mises dans cette boîte jusqu'à ce que l'on sorte de la clause).

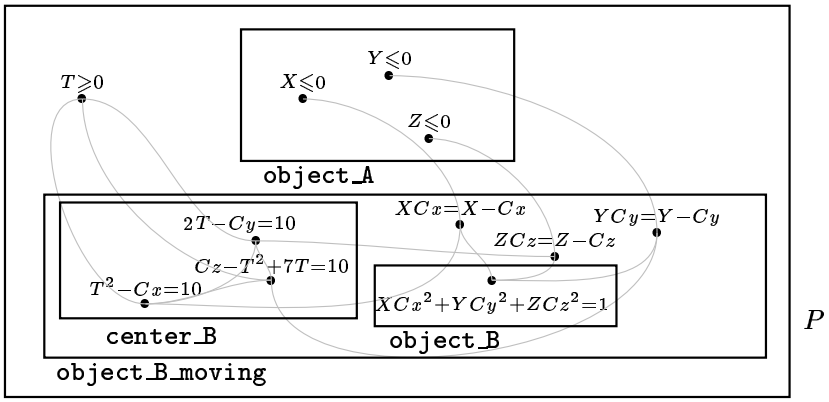


FIG. 10.2: Structure du *store* pour le programme P (tous les buts marqués)

Cette structuration du *store* permet d'en montrer des abstractions en masquant par exemple l'ensemble des contraintes se trouvant dans les boîtes d'un certain niveau. Par exemple, le *store* de la figure 10.2 peut être abstrait en un *store* équivalent ne contenant plus que trois contraintes — $T \geqslant 0$, `object_A(`X,Y,Z`)` et `object_B_moving(`T,X,Y,Z`)` — (figure 10.3). Il est donc plus facile d'ajouter des « gardes » sur les liens entre ces contraintes pour prévenir l'utilisateur de la violation d'une propriété attachée à une contrainte globale définie par une clause. Par exemple, si les domaines des variables en sortie de la boîte `object_B_moving` ne correspondent pas à ce

que l'on attendait compte tenu des domaines en entrée, on peut autoriser l'utilisateur à se focaliser sur cette boîte en « descendant » dans la hiérarchie de boîtes pour obtenir le *store* de la figure 10.4.

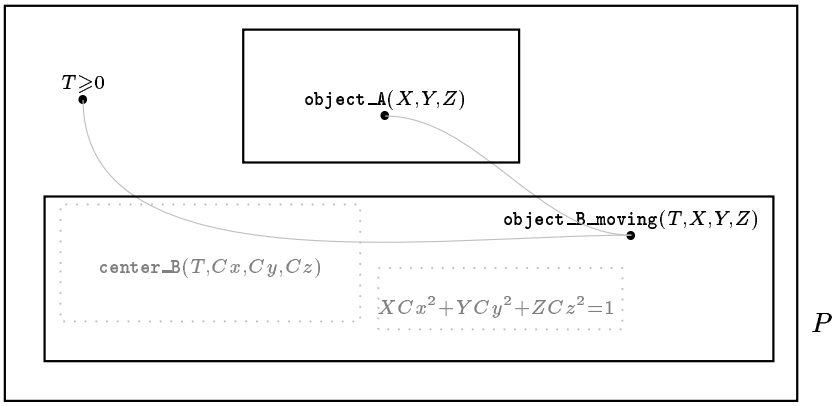


FIG. 10.3: Abstraction du *store* pour le programme P

L'action de focalisation sur une boîte restreint momentanément le *store* à l'ensemble des contraintes contenues directement ou indirectement dans cette boîte. Cela suppose que la propagation des variations de domaines ne se fasse que dans la boîte qui a le *focus* (sinon, on ne peut avoir la certitude que l'erreur se trouve bien parmi les contraintes de la boîte) et que le *focus* soit rendu à la boîte globale à un moment ou à un autre afin que toutes les contraintes du programme aient une chance d'être réinvoquées et que la sémantique du programme soit préservée.

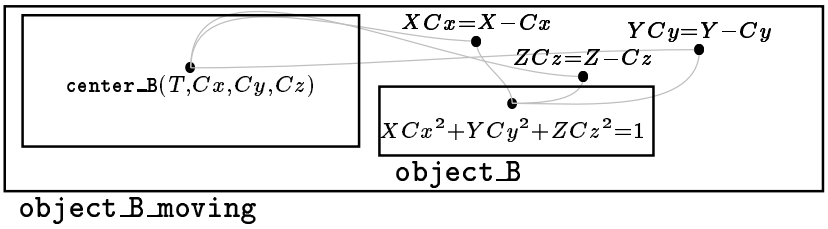


FIG. 10.4: Le nouveau *store* après focus sur `object_B_moving`(T, X, Y, Z)

Notons qu'une plus grande souplesse peut être obtenue si l'on autorise l'utilisateur à créer des boîtes qui ne sont pas directement liées à la structure clausale de son programme, ce qui peut se faire par un mécanisme de marquage des contraintes à mettre dans une même boîte (*cf.* le programme 10.1 où les marques correspondent aux buts du programme).

10.2 Les *S-boxes*

La notion de « boîte » abordée dans la section précédente est définie plus formellement dans cette section. Considérons le système de contraintes $\mathcal{S} = c_1, \ldots, c_m$ ainsi que le produit cartésien des domaines des variables apparaissant dans $\mathcal{S}$, $\mathbf{D} = D_1 \times \cdots \times D_n$. Rappelons (*cf.* section 3.2) que la sémantique déclarative $\mathcal{S}^\star$ de $\mathcal{S}$ correspond à l'ensemble des n-uplets de $\mathbf{D}$ satisfaisant la conjonction des contraintes $c_1 \wedge \cdots \wedge c_m$ et que, dans le cas général, on doit se contenter d'une approximation $\overline{\mathcal{S}}$ de cette sémantique (cas de variables à valeurs dans l'ensemble des réels, par exemple). Cette approximation correspond au plus grand point-fixe commun des $N[c_i]$ inclus dans $\mathbf{D}$ [24] :

$$\overline{\mathcal{S}} = \max(\{u \in \bigcap_{i=1}^{m} \text{point-fixe}(N[c_i]) \mid u \subseteq \mathbf{D}\}) \tag{10.1}$$

Définissons la contrainte $C = c_1 \wedge \cdots \wedge c_m$ dont l'opérateur de contraction $N[C]$ calcule $\overline{\mathcal{S}}$. On peut remplacer l'ensemble des contraintes $\{c_1, \ldots, c_m\}$ par la contrainte C sans changer la sémantique du programme.

La contrainte C peut à son tour être intégrée de la même façon dans une contrainte C'. On peut ainsi hiérarchiser un ensemble important de contraintes en en préservant la sémantique. Cette idée est à la base de la notion de *S-box* :

Définition 10.1 (*S-box* [93]). Soit $\mathcal{C}$ un ensemble de contraintes sur les variables $v_1, \ldots, v_n$ dont les domaines possibles appartiennent à l'ensemble $\mathbb{D}$. Une *S-box* sur $\mathcal{C}$ est un ensemble non-vide $\sigma = \{a_1, \ldots, a_m\}$ où a_i est une contrainte de $\mathcal{C}$ ou une *S-box*. L'*opérateur de contraction* $N[\sigma]$ associé à la *S-box* σ est défini comme suit :

$$\begin{array}{rl} N[\sigma] : & \mathbb{D}^n \longrightarrow \mathbb{D}^n \\ & \boldsymbol{D} \mapsto N[\sigma](\boldsymbol{D}) = \max(\{u \in \bigcap_{i=1}^{m} \text{point-fixe}(N[a_i]) \mid u \subseteq \boldsymbol{D}\}) \end{array}$$

L'opérateur associé à l'algorithme Nar a les propriétés suivantes [178] : il est *contractant*, *correct*, *confluent* et *monotone*. C'est donc un opérateur de contraction. Comme on a aussi, $\text{Nar}(\{(c_1, N[c_1]), \ldots, (c_m, N[c_m])\}, \{D_1 \times \cdots \times D_n\}) = \overline{\mathcal{S}}$ pour $\mathcal{S} = \{c_1, \ldots, c_m\}$, on en déduit que l'on peut en pratique utiliser Nar comme implémentation de l'opérateur de contraction $N[\sigma]$ d'une *S-box* σ.

Cependant, pour que la notion de *S-box* corresponde à celle de « boîte » de la section précédente, il faut que l'opérateur de contraction $N[\sigma]$ soit obtenu de façon atomique, c'est-à-dire que le point-fixe de la propagation des variations de domaines soit atteint dans chaque *S-box* σ d'une hiérarchie de *S-boxes* sans réinvoquer les contraintes en dehors de σ. Considérons par exemple la hiérarchie de *S-boxes* correspondant au *store* de la figure 10.2 (voir l'arbre de la figure 10.5).

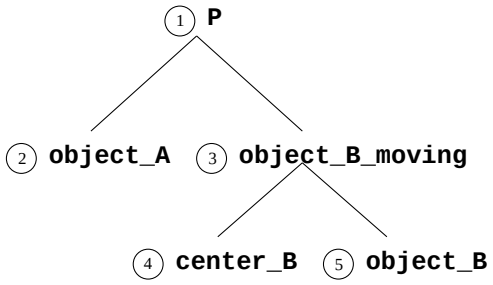


FIG. 10.5: Représentation en arbre de la hiérarchie de boîtes/*S-boxes* de la figure 10.2

Opérationnellement, si l'on considère que chaque *S-box* gère sa propre file de propagation pour l'ensemble des contraintes qu'elle contient, on ne peut réinvoquer une contrainte se trouvant dans une liste appartenant à une *S-box* « au-dessus » ou au même niveau que la *S-box* courante σ (celle contenant la dernière contrainte réinvoquée) si la liste de σ ou celle de l'une des *S-boxes* « sous » σ n'est pas vide.

La difficulté réside ici dans le fait que, lors de la propagation, une contrainte peut être ajoutée dans n'importe quelle liste quel que soit le nœud courant. Par exemple, supposons que la *S-box* courante soit celle du nœud ③. Il est alors possible que la réinvocation d'une contrainte se trouvant dans sa liste de propagation ajoute des contraintes dans la liste des nœuds ④ et ⑤. Il faudra donc reconsidérer ces listes avant d'aller voir celles des nœuds ① et ②. Cependant, réinvoquer une contrainte de la liste de ⑤ après avoir vidé la liste de ④ peut ré-introduire des contraintes dans ④. Il faudra donc retourner en ④ avant de remonter. Au final, on est sûr que cela terminera (on travaille avec des opérateurs contractants sur un ensemble fini de nombres), mais il faudra parcourir l'arbre en suivant un chemin compliqué avant d'obtenir la quiescence globale.

Pour résoudre ce problème, nous proposons de n'utiliser qu'une seule file de propagation $\mathcal{Q}$ et de se reposer sur les propriétés de la *notation de* DEWEY [135]. On associe à chaque contrainte l'indice de DEWEY dans l'arbre de la hiérarchie de *S-boxes* de la *S-box* qui la contient (*cf.* fig. 10.6) ; pendant la propagation, chaque contrainte c' à (re-)considérer est ajoutée dans la liste de propagation à une position telle que la liste soit triée par rapport à l'ordre $\mathcal{O}_c$ défini ci-dessous (avec c la dernière contrainte réinvoquée avant l'ajout de c').

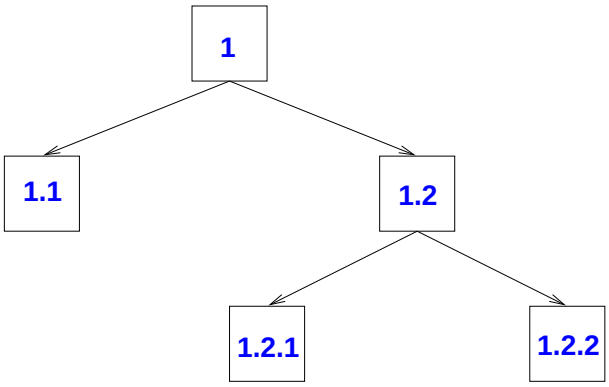


FIG. 10.6: Hiérarchie de *S-boxes* de la figure 10.5 avec la notation de DEWEY

Notations : Soit P un programme, $\mathcal{H}$ une hiérarchie de s-boxes sur P et σ une *S-box* contenant une contrainte c de P. L'indice $\mathcal{I}_c$ de c correspond à l'indice de DEWEY de σ dans $\mathcal{H}$. Soit SzComPref$(\mathcal{I}_1, \mathcal{I}_2)$ la fonction retournant la « taille » (nombre de signes) du préfixe commun aux indices $\mathcal{I}_1$ et $\mathcal{I}_2$.

Définition 10.2 (Ordre partiel $\mathcal{O}_c$ sur les contraintes). Étant données trois contraintes c_1, c_2 et c, la contrainte c_1 est dite *plus petite que* c_2 par rapport à c ($c_1 \overset{c}{\prec} c_2$) si et seulement si :

$$c_1 \overset{c}{\prec} c_2 \iff \begin{cases} 1. \quad \text{SzComPref}(\mathcal{I}_{c_1}, \mathcal{I}_c) > \text{SzComPref}(\mathcal{I}_{c_2}, \mathcal{I}_c) \\ 2. \quad \text{ou SzComPref}(\mathcal{I}_{c_1}, \mathcal{I}_c) = \text{SzComPref}(\mathcal{I}_{c_2}, \mathcal{I}_c) \\ \quad \text{et } \mathcal{I}_{c_2} >^{\text{lexe}} \mathcal{I}_{c_1} \\ \quad (>^{\text{lexe}} \text{: extension lexicographique de } > \text{ sur les naturels}) \end{cases}$$

Exemple 10.27 (Ordre sur les contraintes). *Étant données les contraintes c_1, c_2, c_3, c_4 d'indices $\mathcal{I}_{c_1} = 1.1.5$, $\mathcal{I}_{c_2} = 1.1.4$, $\mathcal{I}_{c_3} = 1.1.5.6$ et $\mathcal{I}_{c_4} = 3.1.4$, on a les inégalités :*

$$\begin{cases} c_1 \overset{c_3}{\prec} c_2 \\ c_2 \overset{c_4}{\prec} c_1 \end{cases}$$

On peut alors prouver que la stratégie de propagation basée sur l'ordonnancement des contraintes donné ci-dessus permet d'assurer l'atomicité du calcul du point-fixe dans les *S-boxes* en remarquant que la règle 1 assure que les contraintes des *S-boxes* les plus proches de la *S-box* σ de c (dernière contrainte réinvoquée) seront mises en premier dans la file de propagation (principe de localité) et que la règle 2 trie les contraintes se trouvant dans des *S-boxes* « au-dessus » ou au même niveau que σ de façon à ce que le principe de localité soit déjà respecté pour elles lorsque ce sera leur tour d'être considérées. L'algorithme Nar est modifié afin de prendre en compte la nouvelle stratégie de propagation. L'algorithme 10.1 en donne une version incrémentale : RevIncNar.

Soient $c_1, \ldots, c_m$ les contraintes apparaissant dans la liste de propagation $\mathcal{Q}$ (dans cet ordre). Les fonctions utilisées par l'algorithme 10.1 sont :

SBox(c) **:** retourne la *S-box* contenant c ;

ALG. 10.1: RevIncNar : Nar revisité pour gérer les *S-boxes*

```
RevIncNar(entrée: (c_1, N[c_1]) ; entrée/sortie: D = D_1 × ··· × D_n)
début
   Q ← Insert_ι(Q, c_1)   % Contrainte ajoutée à la liste de propagation
   ST ← ST ∪ {c_1}   % Contrainte ajoutée au store
   tant que (non(Empty(Q)) et UnderFocus(SBox(Top(Q))) et D ≠ ∅) faire
         (Q, c) ← PopHead(Q)
         ι ← Dewey(c)
         D' ← N[c](D)
         si (D' ≠ D) alors
               pour tout {c_j ∈ ST | ∃x_k ∈ Var(c_j) ∧ D'_k ≠ D_k} faire
                     Q ← Insert_ι(Q, c_j)
               fpt
               D ← D'
         finsi
   ftq
fin
```

Dewey(c) : retourne l'indice de DEWEY de c ;

Insert$_\iota(\mathcal{Q}, c)$: insère la contrainte c dans la file $\mathcal{Q}$ juste avant la contrainte c_i vérifiant :

$$\begin{cases} \forall j \in \{1, \ldots, i-1\} : \mathcal{I}_{c_j} \overset{\iota}{\lhd} \mathcal{I}_c \\ \forall j \in \{i, \ldots, m\} : \mathcal{I}_{c_j} \overset{\iota}{\trianglerighteq} \mathcal{I}_c \end{cases}$$

où $\lhd$ est l'ordre sur les indices de DEWEY correspondant à l'ordre $\prec$ sur les contraintes ;

UnderFocus(σ) : retourne vrai si l'indice de DEWEY de la *S-box* σ_f ayant le *focus* est un préfixe de l'indice de DEWEY de σ (*i.e.* σ est un « descendant » de σ_f dans la hiérarchie de *S-boxes*) ;

Top$(\mathcal{Q})$: retourne c_1 sans la retirer de $\mathcal{Q}$;

PopHead$(\mathcal{Q})$: retourne c_1 et l'élimine de $\mathcal{Q}$.

On remarquera que l'indice ι doit être initialisé à 1 (où « 1 » correspond à l'indice de DEWEY de la *S-box* contenant toutes les contraintes) quelque part en dehors de RevIncNar avant le premier appel à l'algorithme. De plus, $\mathcal{Q}$ peut ne pas être vide lors de l'entrée ou de la sortie de RevIncNar. C'est par exemple le cas lorsque le focus porte sur la *S-box* σ_f et qu'il existe des contraintes à reconsidérer dans la *S-box* σ se trouvant « au-dessus » de σ_f.

10.3 Présentation du débogueur

Nous avons implémenté un prototype de débogueur au-dessus de clp(FD) [50], une extension de Prolog par CODOGNET et DIAZ pour la programmation par contraintes avec des variables à valeur dans des domaines finis. Nous en présentons brièvement les fonctionnalités ci-dessous.

La figure 10.7 montre une copie d'écran effectuée lors d'une tentative de traçage de la résolution de `cars` (*cf.* prog. 10.2), un problème d'ordonnancement fourni avec clp(FD). La figure 10.8 montre la même séance de débogage où l'on a inclus les différentes clauses dans des S-boxes.

La fenêtre principale est constituée en deux parties :

1. un éditeur de texte contenant le code source à interpréter ;
2. une console où l'on peut entrer des buts à faire évaluer.

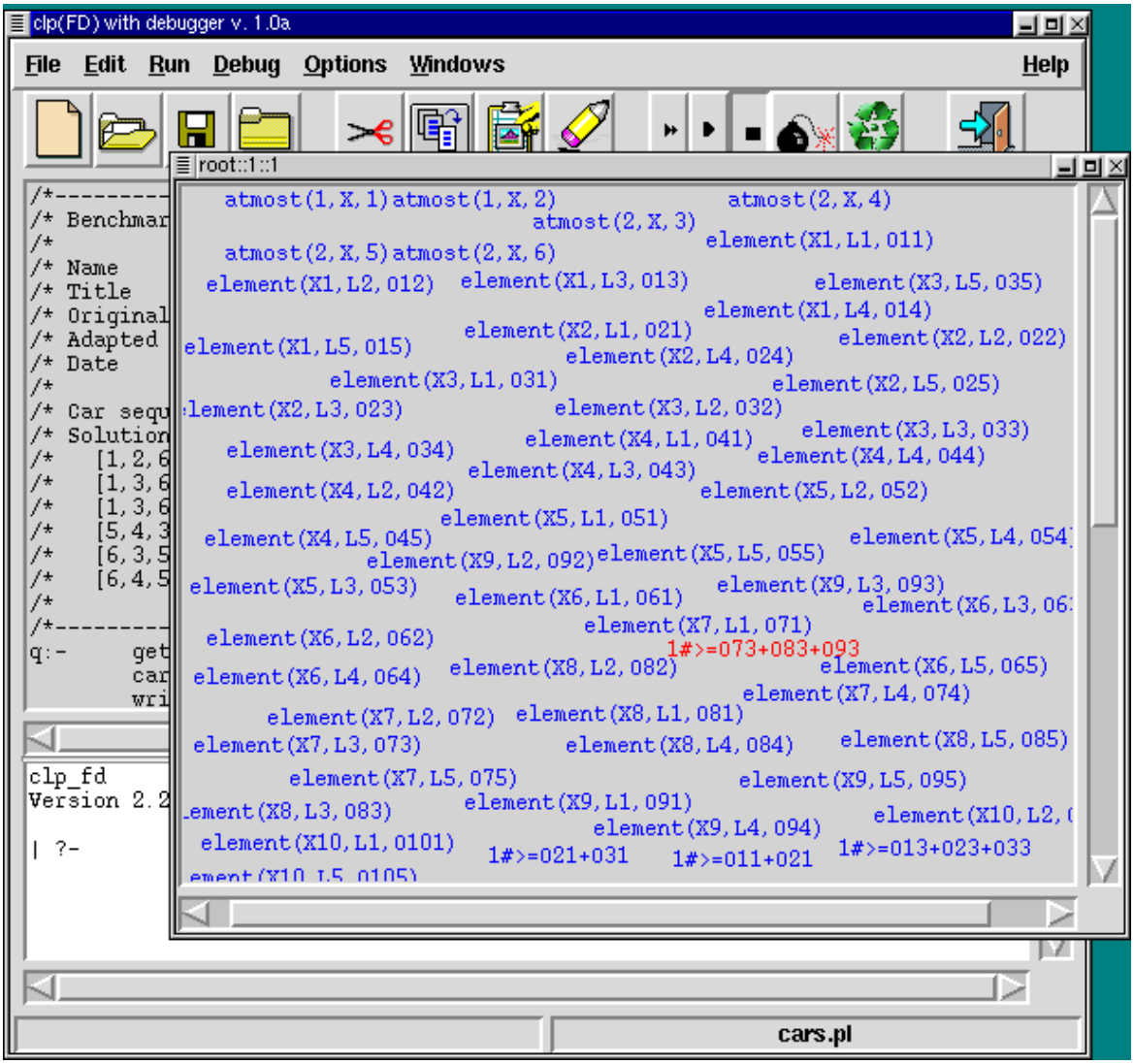


FIG. 10.7: Utilisation du débogueur pour `cars.pl` sans *S-boxes*

La création des S-boxes peut se faire dans l'éditeur en sélectionnant à la souris des ensembles de contraintes et/ou de buts et en leur donnant un nom (*cf.* fig. 10.9).

Le débogueur permet de visualiser graphiquement et en temps réel les modifications des domaines des variables (*cf.* fig. 10.10). Les contraintes et les variables apparaissent dans les S-boxes avec le même aspect que dans le code source (en particulier, le nom des variables est préservé, même si l'on utilise aussi leur représentation interne Prolog comme identifiant).

PROG. 10.2: Le code clp(FD) pour `cars`

```
:- cars(L).
cars(X):- X=[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10],
        Y=[O11,O12,O13,O14,O15,O21,O22,O23,O24,O25,O31,O32,O33,O34,O35,O41,
           O42,O43,O44,O45,O51,O52,O53,O54,O55,O61,O62,O63,O64,O65,O71,O72,
           O73,O74,O75,O81,O82,O83,O84,O85,O91,O92,O93,O94,O95,O101,O102,
           O103,O104,O105],
        L1=[1,0,0,0,1,1], L2=[0,0,1,1,0,1], L3=[1,0,0,0,1,0],
        L4=[1,1,0,1,0,0], L5=[0,0,1,0,0,0],
        domain(Y,0,1), domain(X,1,6),
        atmost(1,X,1), atmost(1,X,2), atmost(2,X,3), atmost(2,X,4),
        atmost(2,X,5),atmost(2,X,6),
        element(X1,L1,O11), element(X1,L2,O12), element(X1,L3,O13), element(X1,L4,O14),
        element(X1,L5,O15), element(X2,L1,O21), element(X2,L2,O22), element(X2,L3,O23),
        element(X2,L4,O24), element(X2,L5,O25), element(X3,L1,O31), element(X3,L2,O32),
        element(X3,L3,O33), element(X3,L4,O34), element(X3,L5,O35), element(X4,L1,O41),
        element(X4,L2,O42), element(X4,L3,O43), element(X4,L4,O44), element(X4,L5,O45),
        element(X5,L1,O51), element(X5,L2,O52), element(X5,L3,O53), element(X5,L4,O54),
        element(X5,L5,O55), element(X6,L1,O61), element(X6,L2,O62), element(X6,L3,O63),
        element(X6,L4,O64), element(X6,L5,O65), element(X7,L1,O71), element(X7,L2,O72),
        element(X7,L3,O73), element(X7,L4,O74), element(X7,L5,O75), element(X8,L1,O81),
        element(X8,L2,O82), element(X8,L3,O83), element(X8,L4,O84), element(X8,L5,O85),
        element(X9,L1,O91), element(X9,L2,O92), element(X9,L3,O93), element(X9,L4,O94),
        element(X9,L5,O95), element(X10,L1,O101), element(X10,L2,O102),
        element(X10,L3,O103), element(X10,L4,O104), element(X10,L5,O105),
        1 #>= O11+O21, 1 #>= O21+O31, 1 #>= O31+O41, 1 #>= O41+O51, 1 #>= O51+O61,
        1 #>= O61+O71, 1 #>= O71+O81, 1 #>= O81+O91, 1 #>= O91+O101,
        2 #>= O12+O22+O32, 2 #>= O22+O32+O42,
        2 #>= O32+O42+O52, 2 #>= O42+O52+O62, 2 #>= O52+O62+O72, 2 #>= O62+O72+O82,
        2 #>= O72+O82+O92, 2 #>= O82+O92+O102, 1 #>= O13+O23+O33, 1 #>= O23+O33+O43,
        1 #>= O33+O43+O53, 1 #>= O43+O53+O63, 1 #>= O53+O63+O73, 1 #>= O63+O73+O83,
        1 #>= O73+O83+O93, 1 #>= O83+O93+O103, 2 #>= O14+O24+O34+O44+O54,
        2 #>= O24+O34+O44+O54+O64, 2 #>= O34+O44+O54+O64+O74,
        2 #>= O44+O54+O64+O74+O84, 2 #>= O54+O64+O74+O84+O94,
        2 #>= O64+O74+O84+O94+O104, 1 #>= O15+O25+O35+O45+O55,
        1 #>= O25+O35+O45+O55+O65, 1 #>= O35+O45+O55+O65+O75,
        1 #>= O45+O55+O65+O75+O85, 1 #>= O55+O65+O75+O85+O95,
        1 #>= O65+O75+O85+O95+O105,
        O11+O21+O31+O41+O51+O61+O71+O81 #>= 4, O11+O21+O31+O41+O51+O61 #>= 3,
        O11+O21+O31+O41                 #>= 2, O11+O21                 #>= 1,

        O12+O22+O32+O42+O52+O62+O72 #>= 4, O12+O22+O32+O42             #>= 2,
        O12                         #>= 0, O13+O23+O33+O43+O53+O63+O73 #>= 2,
        O13+O23+O33+O43             #>= 1, O13                         #>= 0,
        O14+O24+O34+O44+O54         #>= 2, O15+O25+O35+O45+O55         #>= 1.
```

Une fenêtre est associée à chaque S-box (*cf.* fig. 10.8). Chacune peut être masquée ou affichée selon le souhait de l'utilisateur. Chaque S-box apparaît comme une contrainte ordinaire dans la fenêtre de son ancêtre dans la hiérarchie de S-boxes.

Il est possible d'associer des points d'arrêts aux variables (arrêt lorsque son domaine est égal ou différent à un autre domaine, lorsqu'il est réduit à une seule valeur, etc.), aux contraintes (arrêt en cas de reinvocation) et aux S-boxes (arrêt lorsque l'on rentre ou sort d'un S-box, pour pouvoir inspecter les domaines des variables). D'autres

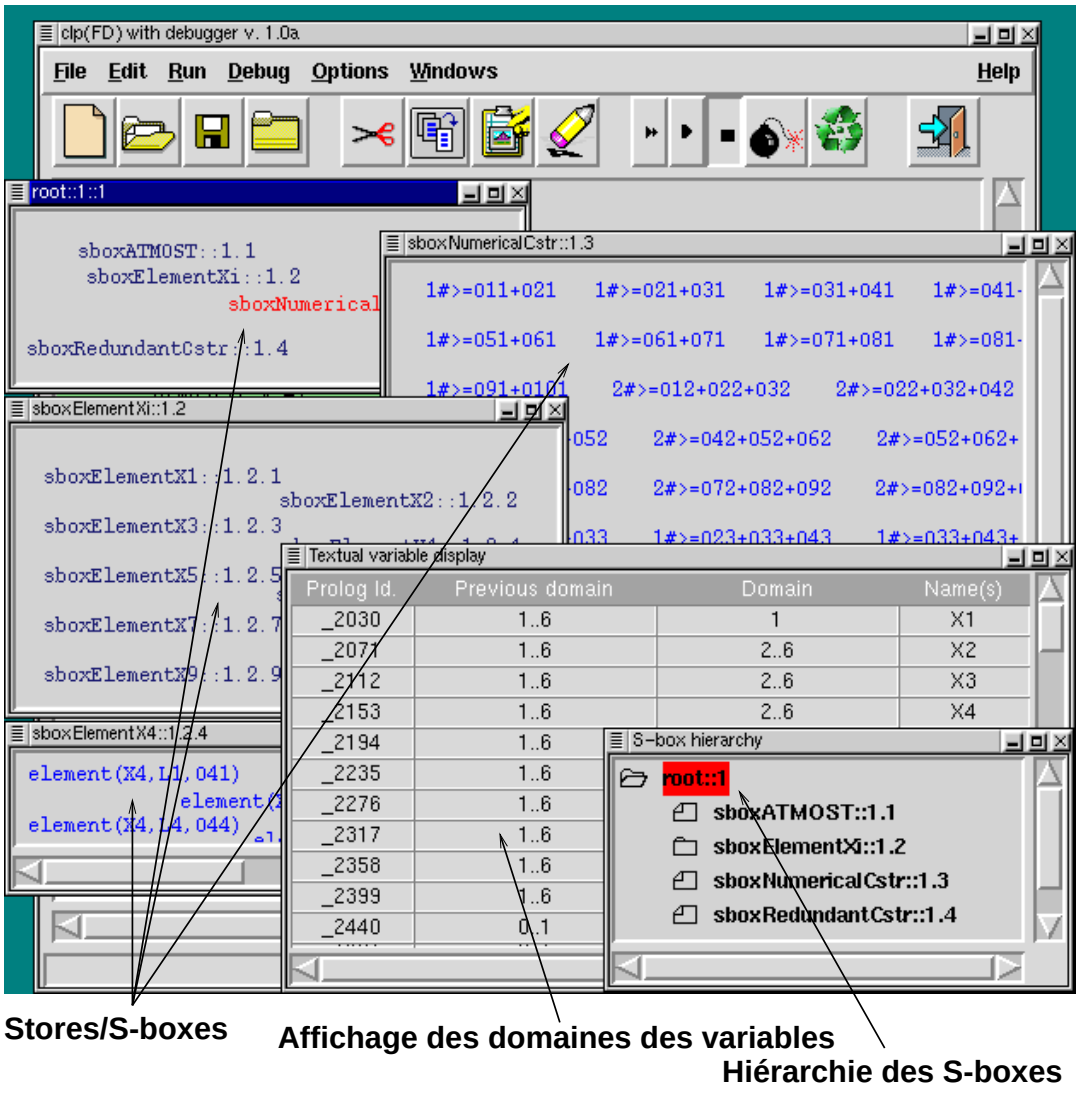


FIG. 10.8: Utilisation du débogueur pour `cars.pl` avec *S-boxes*

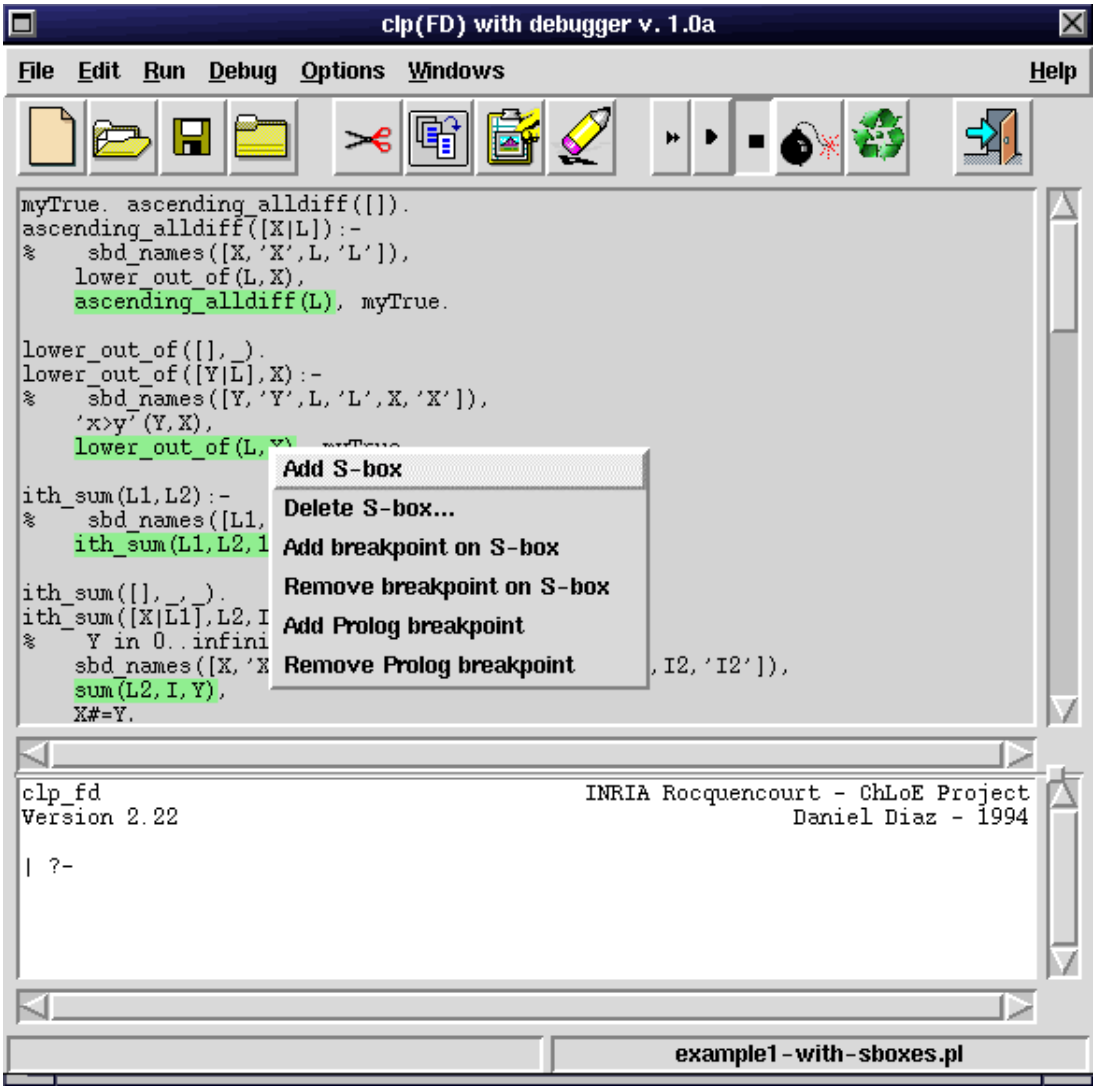


FIG. 10.9: Ajout de S-boxes dans le texte par marquage de buts

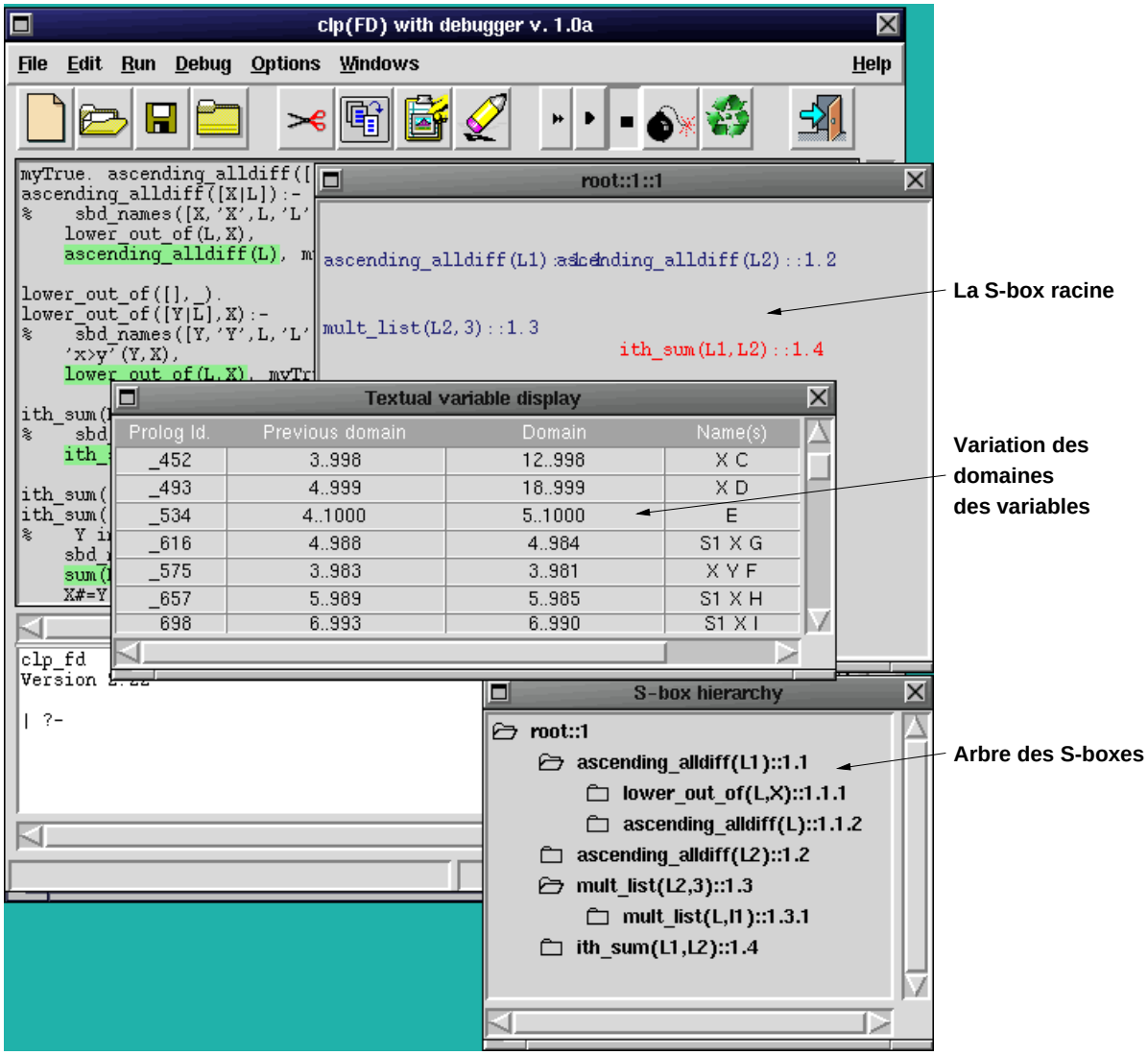


FIG. 10.10: Cession de débogage

évènements tels que l'échec ou l'entrée dans la phase d'énumération peuvent aussi être prétexte à un arrêt. Le processus de propagation peut être exécuté pas-à-pas, ce qui permet à l'utilisateur de comprendre les relations existant entre différents ensembles de contraintes.

10.4 Réalisation pratique

Nous allons présenter dans cette section les détails d'implantation du prototype de débogueur basé sur les *S-boxes* dont les fonctionnalités principales ont été présentées dans la section précédente. Nous ne discuterons pas ici les structures propres à clp(fd), et renvoyons le lecteur aux références [49, 50, 67] ainsi qu'au chapitre 8 de cette thèse pour une présentation détaillée des mécanismes internes du solveur lui-même.

Bien que l'efficacité ne soit pas un objectif primordial pour un débogueur, il est important que la gestion de l'interaction avec l'utilisateur soit suffisamment rapide pour en rendre la manipulation aisée. On est donc tenté d'intégrer très étroitement le débogueur au solveur afin de réduire au maximum les surcoûts de communication entre ces deux modules. D'un autre côté, une intégration totale nuit à la portabilité du débogueur et oblige à le réécrire presque entièrement pour pouvoir l'utiliser sur une autre plate-forme.

Afin de préserver la portabilité tout en garantissant une efficacité raisonnable, nous avons choisi une approche à mi-chemin entre une séparation complète et une intégration totale du débogueur et du solveur en élaborant un système en deux parties (voir figure 10.11) :

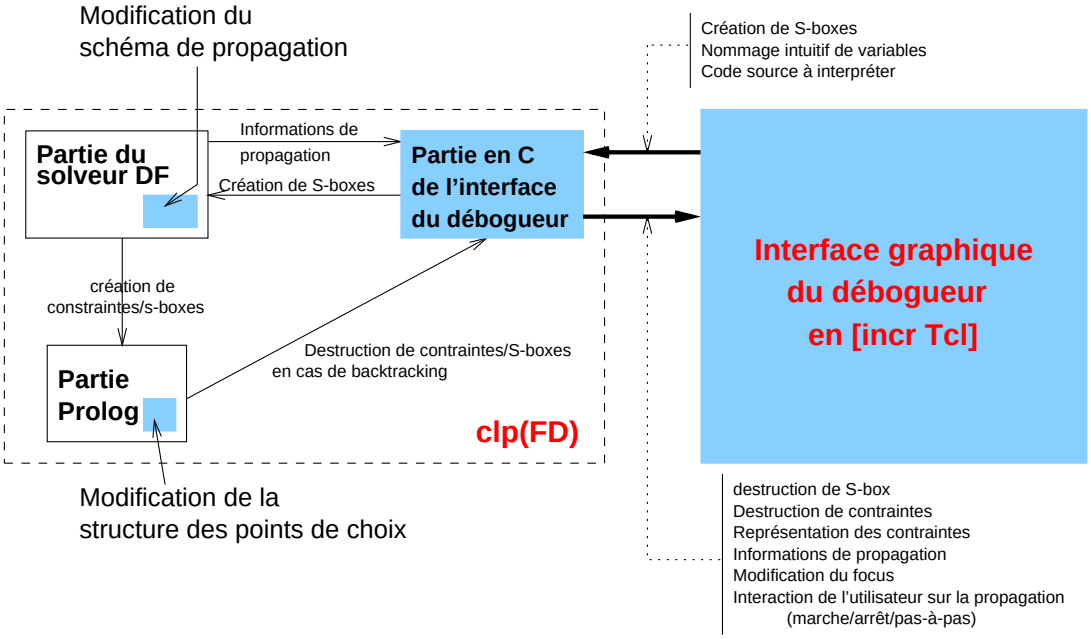


FIG. 10.11: Architecture du débogueur

1. une partie écrite en C étroitement liée au solveur, qui implémente certaines primitives de bas niveau telles que les méthodes décrivant la façon d'afficher une contrainte ainsi que certaines structures ayant besoin d'un accès privilégié aux données gérées par le solveur (*e.g.* les structures liées à la représentation des *S-boxes* en terme d'ensemble de contraintes) ;
2. et une partie écrite en [incr Tcl] [161] (une variante orientée objet du langage de script Tcl/Tk [179]) pour implémenter tout ce qui concerne l'interface utilisateur. Cette partie possède sa propre représentation interne des structures de données relatives au *store* ainsi qu'aux *S-boxes* (avec duplication éventuelle de certaines portions déjà présentes dans la partie ci-dessus) afin de minimiser les échanges de messages fort coûteux entre les morceaux en C et ceux en [incr Tcl].

Les cadres grisés de la figure 10.11 correspondent aux parties qui ont été ajoutées ou modifiées dans clp(fd). Comme on peut le constater, la séparation entre le débogueur et le solveur n'est pas aussi nette que l'on pourrait le souhaiter :

– nous avons dû modifier les structures de stockage de points de choix pour pouvoir garder une trace des contraintes et des *S-boxes* ajoutées au *store*. Cela nous permet de les effacer des fenêtres de l'interface graphique lors d'un *backtracking*. La figure 10.12 montre la nouvelle structure pour les points de choix ;
– l'algorithme de propagation de clp(fd) a été remplacé par l'algorithme 10.1 (RevIncNar).

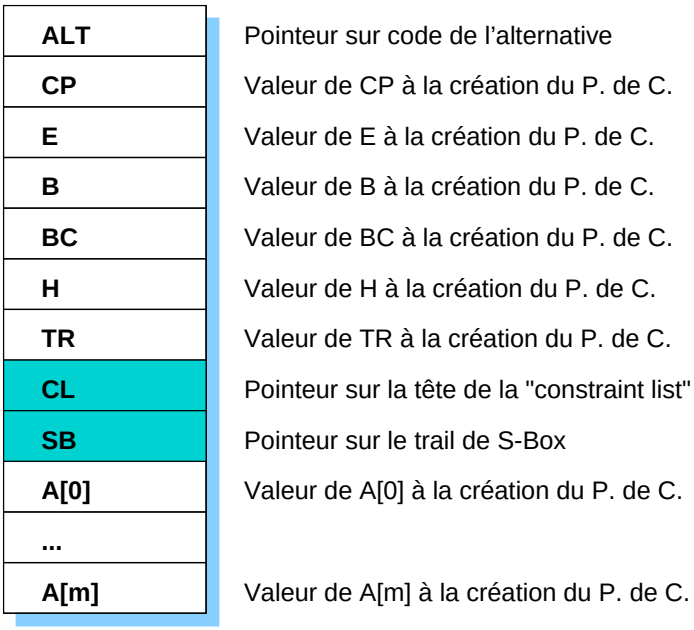


FIG. 10.12: Structure pour la représentation des points de choix

Il serait en fait difficile de faire un débogueur basé sur les *S-boxes* sans faire ces modifications au niveau du noyau du résolveur. La seule alternative serait de pouvoir intégrer le débogueur dans un système possédant déjà les fonctionnalités nécessaires.

La table 10.1 présente tous les registres ajoutés à ceux présents dans clp(fd). Leur utilisation est décrite dans les sections suivantes.

TAB. 10.1: Registres ajoutés à clp(fd)

Nom	Sens	Description
SBF	*S-Box Focus*	pointeur sur la *S-box* qui a le *focus*
SBC	*S-Box Current*	pointeur sur la *S-box* courante *i.e.* la dernière créée et non finie
PLH	*Propagation List Head*	Pointeur sur la tête de la liste de propagation

10.4.1 Modification du processus de *backtracking*

Une structure de *trail* (voir figure 10.13) est créée chaque fois que le moteur Prolog ajoute un point de choix. En toute généralité, on ne peut en effet mettre les informations relatives à une *S-box* sur le trail Prolog (*cf.* fig. 8.3) car ces dernières peuvent ne pas suivre la structure clausale du programme (une *S-box* peut être créée dans une clause et fermée dans une autre — voir l'exemple 10.28).

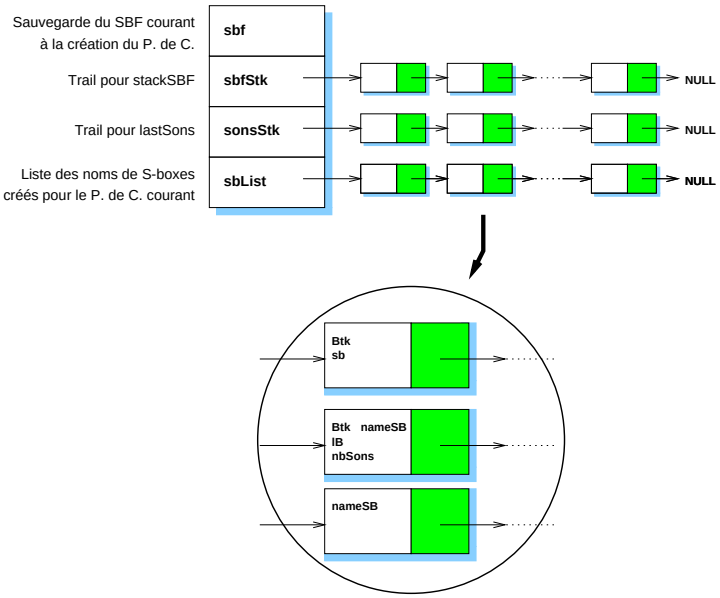


FIG. 10.13: Structure du trail

Exemple 10.28 (Création de *S-boxes* et structure clausale).

```
1  p:- begin_sbox, q, r.          % S-box 1.1 créée
2  q:- t, end_sbox, s.            % S-box 1.1 fermée
3  t:- begin_sbox, s, end_sbox.   % S-box 1.1.1 créée et fermée
```

Par conséquent, toutes les structures de données liées à la gestion des *S-boxes* qu'il est nécessaire de réinitialiser en cas de *backtracking* sont sauvées sur un *trail* local spécial dont l'adresse est gardée dans la structure de point de choix.

10.4.2 Modification de la méthode de propagation

Comme on l'a déjà vu au chapitre 8, la propagation des variations de domaines des variables se fait de la façon suivante dans clp(fd) : à chaque fois que le domaine d'une variable v est modifié, la structure de données correspondant à la variable v est mise en queue de la liste de propagation. La propagation se fait ensuite en retirant la structure de variable se trouvant en tête de la liste et en réinvoquant les contraintes dont l'adresse se trouve dans une des chaînes de la structure (il s'agit en effet des contraintes possédant une occurrence de la variable). Ces réinvocations peuvent à leur tour ajouter de nouvelles structures de variables à la fin de la liste de propagation. La quiescence est atteinte lorsque la queue est vide. Ainsi, on peut dire que la queue de propagation est « orientée variables » car il s'agit d'une liste de variables.

Or, l'algorithme 10.1 de la page 140 est, lui, « orienté contraintes » car sa queue de propagation contient des contraintes. Par conséquent, la gestion des *S-boxes* implique une modification en profondeur du mécanisme de propagation.

Pour ce faire, quatre nouveaux champs ont été ajoutés à la structure de contrainte originale (voir la structure résultant dessinée dans la figure 10.14) : la queue de propagation est implémentée par un chaînage des structures

associées aux contraintes à réinvoquer en utilisant le champ `Next_In_Queue`. La tête de la queue de propagation est pointée en permanence par le registre `PLH`. On pourra remarquer qu'il suffit désormais de garder un pointeur sur la tête de la liste (et non plus un pointeur sur la tête et un sur la queue comme dans la version originale) puisque l'insertion d'une structure de contrainte dans la queue se fait par une recherche séquentielle à partir de la tête de la position préservant l'ordre $\mathcal{O}$. Le champ `Already_In_Queue` nous permet d'éviter de réinsérer une contrainte se trouvant déjà dans la queue.

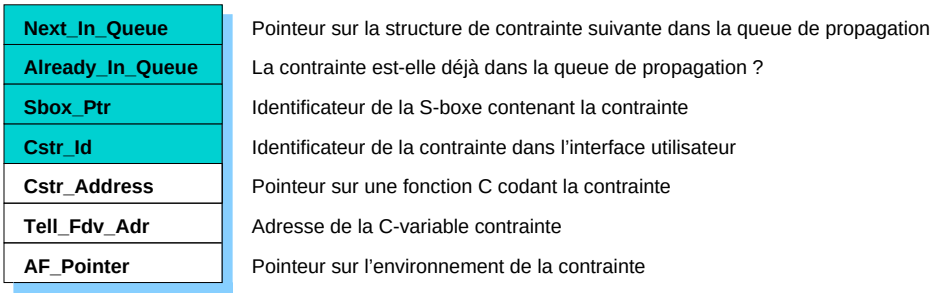


FIG. 10.14: Structure pour la représentation des contraintes

Le registre `SBC` contient le nom de la *S-box* courante. Cela permet de déterminer aisément à quelle *S-box* appartient une contrainte nouvellement ajoutée au *store* : à la création de la structure de la contrainte, on copie le contenu de `SBC` dans son champ `Sbox_Ptr`.

Le dernier champ ajouté dans la structure associée à une contrainte, `Cstr_Id` est un identifiant de la contrainte utilisé lors des communications avec l'interface graphique en [incr Tcl]. Cela permet de demander à l'interface de distinguer (par un changement de couleur) la contrainte active (*i.e.* actuellement réinvoquée).

La structure associée à une variable n'a été que peu modifiée : les seules modifications apportées correspondent à l'élimination des champs non utilisés (*i.e.* tous ceux liés à la gestion de l'ancienne file de propagation par chaînage des variables). La structure modifiée apparaît dans la figure 10.15.

10.4.3 Gestion des *S-boxes*

Comme on l'a vu dans la section 10.3 présentant notre prototype, l'une des méthodes pour créer les *S-boxes* est de marquer certains buts dans l'éditeur de code source du débogueur. Avant d'être interprété, le code contenu dans l'éditeur est fusionné avec ces marques en un nouveau programme de la manière suivante : pour chaque ensemble de buts/contraintes $g_1, \ldots, g_m$, les instructions `begin_sbox`* et `end_sbox` sont ajoutées, respectivement, avant g_1 et après g_m. Ce sont elles qui vont gérer la création et la « fermeture » de la *S-box* correspondante à l'exécution.

Exemple 10.29 (Traduction d'un programme marqué).

```
p:- a, b, c, d.        ⟶        p:- a, begin_sbox, b, c, end_sbox, d.
```

Chaque *S-box* est identifiée par son indice de DEWEY (*cf.* section 10.2). Une chaîne de caractères est une représentation naturelle pour cet identifiant. Une *S-box* a donc un nom de la forme `"1.2.3"`. Un avantage de cette représentation, outre sa simplicité, est que l'implémentation de l'ordre $\mathcal{O}$ devient une comparaison pure et simple de chaînes de caractères.

L'identifiant d'une *S-box* devant être créée est déterminé de la façon suivante : on utilise une pile des « derniers fils » (voir figure 10.16(a)) contenant le nombre de *S-boxes* filles de chaque *S-box* ainsi que certaines informations utilisées lors d'un *backtracking* ; le nom d'une nouvelle *S-box* est alors la concaténation du champ `nameSB` en haut de la pile (correspondant à la *S-box* courante) et du nombre de fils incrémenté de 1.

Lors de la création d'une nouvelle `S-box` σ, le *focus* est passé à σ car la propagation ne peut être faite en dehors de σ tant qu'elle n'a pas été refermée (sinon, on ne pourrait considérer σ comme une contrainte primitive).

*En fait, l'instruction `begin_sbox` accepte un paramètre qui est le nom symbolique de la *S-box* choisi par l'utilisateur. On l'omettra ici pour plus de clarté

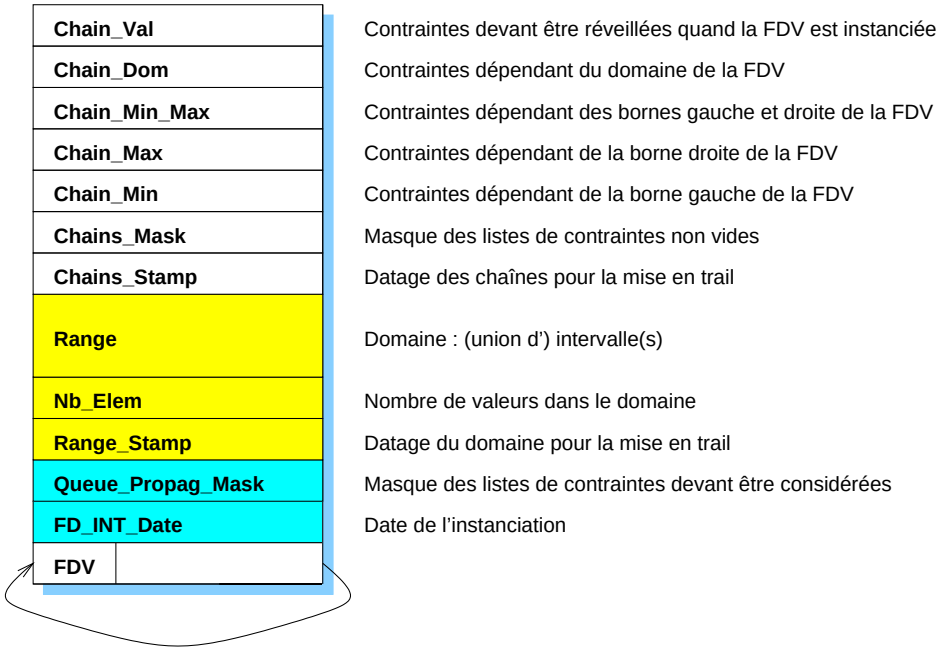
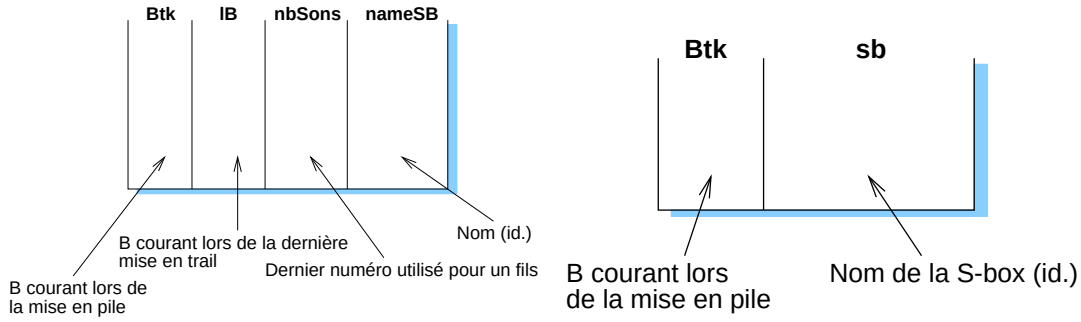


FIG. 10.15: Structure pour la représentation des variables

Il est donc nécessaire de garder une pile des noms des *S-boxes* ayant eu le *focus* de façon à pouvoir le réinitialiser à sa valeur précédente lorsqu'une *S-box* est fermée (pile des SBF illustrée par la figure 10.16(b)).



(a) Pile des « *derniers fils* »

(b) ile de SBF

FIG. 10.16: Les piles internes du débogueur

Notons que l'on ne duplique jamais les identificateurs de *S-boxes* : on se contente de stocker un pointeur sur eux. Il est ainsi facile de libérer la mémoire utilisée pour représenter les chaînes lors d'un *backtracking* en utilisant le champ `sbList` du *trail* de `S-box`.

La première version de notre outil, n'intégrant pas le mécanisme des S-boxes, était écrite en Tcl/Tk. L'ajout des S-boxes s'est révélé difficilement réalisable avec Tcl/Tk du fait que ce langage ne possède pas de mécanisme évolué de limitation de portée des variables. À l'inverse, l'utilisation de [incr Tcl] nous a permis de traduire chaque S-box par un objet, facilitant ainsi leur manipulation et les communications entre elles.

CINQUIÈME PARTIE

Programmation par contraintes et programmation orientée objets

1. **Introduction**
2. **Présentation de JAIL**
3. **Validation des algorithmes de calcul d'approximations intérieures et de gestion de variables quantifiées : OpAC et WOpAC**

Introduction

LES SYSTÈMES DE PROGRAMMATION LOGIQUE AVEC CONTRAINTES tels que celui présenté dans la partie IV sont bien adaptés au prototypage ou à l'expérimentation ; ils sont cependant difficilement intégrables dans des applications implémentées dans des langages impératifs. En particulier, la communication entre les différents modules requiert souvent la connaissance d'un langage et/ou d'un protocole d'interfaçage spécifique. La facilité de pose des contraintes dans un langage de haut niveau tel que Prolog apparaît à l'évidence inutile pour de telles utilisations et l'on souhaiterait plutôt disposer d'une librairie offrant des facilités de résolution de contraintes (*cf.* ILOG Solver [195, 194], par exemple). C'est pourquoi nous avons défini durant notre thèse la librairie C++ OpAC pour la résolution de contraintes. Afin de garantir la plus grande flexibilité possible, OpAC a été implémentée sous forme de modules, le premier d'entre eux, correspondant aux méthodes pour les calculs en arithmétique d'intervalles, constituant une librairie à part : JAIL. OpAC peut être très facilement étendue grâce à l'emploi de l'approche « objets », puisqu'il suffit de dériver une des classes de base pour implémenter de nouvelles méthodes de résolution, d'énumération, voires de nouveaux types de contraintes sur de nouveaux domaines. Deux extensions d'OpAC existent actuellement :

- pour les besoins du projet FASE au CWI (Pays-Bas), nous avons développé un ensemble d'opérateurs associés à des contraintes de faible arité apparaissant lors de la modélisation d'expressions du visage humain par le déplacement de points-clés ;
- afin de valider les algorithmes présentés au chapitre 6, Marc CHRISTIE, doctorant à l'Institut de Recherche en Informatique de Nantes, a implémenté WOpAC, une extension d'OpAC autorisant la résolution de contraintes avec variables universellement quantifiées. Il a aussi intégré OpAC dans un prototype de modeleur déclaratif de mouvement de caméra adapté à la cinématographie.

Le plan de cette partie est le suivant : nous décrivons au chapitre 11 l'implémentation de JAIL, notre librairie de calcul en arithmétique d'intervalles en indiquant les difficultés rencontrées lors de la mise au point de ce type de librairie et en présentant les solutions adoptées, puis nous analysons l'impact de différentes techniques d'implémentation s'offrant à nous et nous comparons les performances de JAIL avec celles de librairies d'intervalles actuellement disponibles. Enfin, nous présentons l'architecture d'OpAC au chapitre 12 et décrivons les résultats obtenus avec WOpAC en ce qui concerne la résolution de contraintes avec variables universellement quantifiées.

Contributions

- Définition et mise au point de JAIL, une librairie C++ paramétrée efficace pour l'arithmétique d'intervalles et analyse de différentes implémentations ;
- définition d'OpAC, une librairie C++ ouverte facilement extensible pour la programmation par contraintes ; description d'une extension d'OpAC pour la gestion de systèmes de contraintes réelles non-linéaires avec variables universellement quantifiées.

CHAPITRE 11

Implémentation de l'arithmétique d'intervalles : *JAIL (Just Another Interval Library)*

Patrons de classes C++ — traits en C++ — architecture de JAIL — implémentation des opérateurs — évaluation de JAIL

COMME ON L'A VU dans la partie II, l'arithmétique des intervalles est utilisée par certains résolveurs de contraintes pour calculer efficacement et rigoureusement les solutions de systèmes de contraintes linéaires et non-linéaires réelles. L'implantation de ces outils requiert des librairies de calcul sur les intervalles qui soient elles-mêmes efficaces et correctes. Comme le remarque PRIEST [191] en ce qui concerne la correction, ce n'est malheureusement pas le cas de plusieurs librairies parmi les plus utilisées. Ceci semble plus sûrement dû aux subtilités inhérentes au calcul avec des nombres flottants et à la diversité des implantations de la norme IEEE 754 (*cf.* la section 1.3, p. 12–13) qu'à des algorithmes incorrects pour le calcul des opérateurs sur les intervalles.

Un tel état de fait est de nature à rendre méfiants les utilisateurs vis-à-vis des outils basés sur l'arithmétique des intervalles. Il est donc indispensable de disposer de librairies sûres, rapides et le plus portable possible de calcul sur les intervalles. Certaines tentatives de normalisation et de spécification détaillée de telles librairies ont déjà été faites : [44, 241, 243, 184]. Il n'existe cependant pas encore de standard implémenté universellement reconnu dont la correction soit totale et l'efficacité raisonnable. De plus, comme on l'a évoqué dans la section 1.3, l'existence de librairies d'intervalles correctes repose en partie sur une reconnaissance explicite par les langages de programmation des spécificités et des propriétés de l'arithmétique flottante. Ce n'est pour le moment, à notre connaissance, le cas d'aucun langage. On peut cependant espérer qu'il n'en sera bientôt plus de même : les fonctionnalités nécessaires à l'arithmétique d'intervalles font partie de la proposition de standard pour Fortran 2000 ; elles sont aussi intégrées dans Borneo [58], le dialecte Java développé à Berkeley par DARCY et KAHAN.

Durant notre thèse de doctorat, nous avons défini et développé une librairie de calcul sur les intervalles qui se veut un pas vers la réalisation de ce standard pour le langage C++ : *JAIL*. Afin d'offrir la plus grande flexibilité possible et de respecter l'esprit des librairies C++ actuellement disponibles, telles que la STL [221] (*Standard Template Library*), JAIL a été conçue comme une librairie entièrement paramétrée acceptant n'importe quel format de flottant comme type de base (il suffit que le format utilisé respecte un certain nombre de propriétés énoncées dans la suite). Il est ainsi possible d'instancier indifféremment JAIL en une librairie classique de calcul sur des intervalles à bornes `double`s ou sur des intervalles dont les bornes sont des nombres flottants en précision arbitraire. De même, JAIL définit les opérateurs dans leur forme fonctionnelle (*cf.* sect. 2.2.1) et relationnelle (*cf.* sect. 2.2.3).

Ce chapitre est organisé comme suit : la section 11.1 présente brièvement les techniques de programmation C++ utilisées pour implémenter JAIL qui en font la spécificité (le lecteur est renvoyé aux ouvrages suivants pour une présentation approfondie des concepts C++ manipulés : [226, 238, 239, 240]) ; la section 11.2 décrit ses fonctionnalités et son implémentation tant du point de vue de son architecture en terme de classe C++, qu'en ce qui concerne l'implantation effective des opérateurs sur les intervalles ; puis, la section 11.3 présente ses performances : nous montrons dans un premier temps les résultats de tests effectués sur différentes versions de JAIL afin de justifier nos choix finaux d'implantation, puis nous comparons les performances et la correction de notre librairie avec celles de quatre autres librairies de calcul sur les intervalles citées dans la littérature : Profil [134], RVInterval, le module d'arithmétique d'intervalles du logiciel d'optimisation globale VerGO [236], fi_lib [109] et fi_lib++ [149], une librairie C++ bientôt disponible. On notera qu'aucune de ces librairies n'offre la possibilité de choisir le type

des bornes des intervalles.

11.1 Présentation des techniques C++

La technique C++ des *traits* [173] est à la base de l'architecture de JAIL. Avant de la présenter et d'en montrer l'intérêt pour une bibliothèque de calcul sur les intervalles, nous allons rappeler la notion de *patron de classe*.

11.1.1 Les classes paramétrées

En C++ [226], il est possible de définir des classes ou des fonctions paramétrées (*templated class/function*) par des types primitifs ou d'autres classes, qui peuvent être elles-mêmes paramétrées. Ces classes et ces fonctions ne sont que des patrons qu'il est nécessaire d'*instancier* pour avoir des classes et des fonctions exploitables. La librairie standard C++ (STL) est ainsi entièrement constituée d'algorithmes et de classes paramétrées.

Exemple 11.30 (La classe `list` de la STL). *La classe `list` de la STL définit une liste doublement chaînée. C'est une classe paramétrée qu'il faut instancier avec le type des éléments à insérer en liste. Par exemple, les instructions :*

```
list<int> maListedEntiers;
list<string&> maListeDeChaines;

maListedEntiers.push_front(34);
maListeDeChaines.push_front("ceci est une chaîne");
```

déclarent et utilisent une liste d'entiers et une liste de références sur des chaînes de caractères. Sans instanciation par le type des éléments, la classe `list` n'est pas utilisable. Ce n'est qu'un patron pour une classe.

Un inconvénient inhérent aux patrons de classes est qu'ils ne peuvent être compilés puisque ce ne sont pas de vrais types. De plus, la définition d'un patron devant être entièrement visible pour son instanciation, on est obligé de mettre tout son code dans un fichier d'en-tête (*header file*). En conséquence, l'utilisation d'une librairie employant intensivement les classes paramétrées (comme la STL, par exemple) allonge énormément le temps de compilation des programmes puisque tout le code de la librairie doit être compilé lui aussi à chaque fois (alors qu'une librairie sans patrons peut être compilée une fois pour toutes et liée au programme qui l'utilise).

La définition d'une classe paramétrée se fait de façon analogue à celle d'une classe non paramétrée, en indiquant au préalable la liste des identificateurs qu'il faudra remplacer à l'instanciation.

Exemple 11.31 (La classe paramétrée `Nombre`). *On souhaite évaluer par la pratique la complexité d'algorithmes. Pour cela, on définit une classe `Nombre` redéfinissant les opérations arithmétiques pour un type de base numérique laissé au choix de l'utilisateur de façon à comptabiliser chaque opération effectuée. La classe `Nombre` doit donc être paramétrée par ce type. Le programme 11.1 présente le code de la classe. Le paramètre `T` déclaré à la ligne 4 correspond au type de base donné à l'instanciation. Le programme principal (lignes 54 à 67) définit deux instances de la classe `Nombre` pour le type `double` et le type `int`. Si l'on exécute le programme 11.1, on obtient le résultat suivant :*

```
5 + 6 = 11
5 / 6 = 0.833333
cos(5) = 0.283662
5 + 6 = 11
5 / 6 = 0
cos(5) = 0
```

L'instanciation de la classe `Nombre` pour le type `double` donne bien les résultats escomptés. Par contre, les résultats pour le type `int`, bien que parfaitement corrects, peuvent paraître déroutants. Afin de comprendre ce qui se passe, considérons le cas de la surcharge de l'opérateur de division (lignes 36 à 40). Le paramètre `T` a la valeur `int` ; donc la division `val/v.val` à la ligne 39 est une division de deux entiers retournant, en accord avec le standard C++, le quotient entier, utilisé pour la construction du résultat. Le quotient de $5/6$ valant bien 0,

le résultat retourné est correct. Il en est de même pour l'opération `cos()`*. En l'absence de toute spécification, la fonction* `cos()` *appelée à la ligne* 45 *est celle définie dans la librairie mathématique standard, prenant en entrée un* `double` *et retournant un* `double`*. Lors de son appel, la valeur de* `v.val` *est promue en* `double` *sans perte d'information. Par contre, le résultat* `double` *est donné en entrée du constructeur* `Nombre<int>` *qui le caste en* `int`*. La valeur 0,283 662 est alors transformée en* 0 *pour créer le résultat.*

PROG. 11.1: La classe `Nombre` sans *trait*

```
1  #include <iostream.h>
2  #include <math.h>
3
4  template<class T>
5  class Nombre {
6  public:
7    Nombre(const T& v);
8
9    Nombre operator+(const Nombre& x);
10    Nombre operator/(const Nombre& v);
11    friend Nombre cos<>(const Nombre& v);
12
13    friend ostream& operator<< <>(ostream& os,
14                                  const Nombre& v);
15 private:
16    T val;
17    static unsigned int nbAdd;
18    static unsigned int nbDiv;
19    static unsigned int nbCos;
20 };
21
22 template<class T> unsigned int Nombre<T>::nbAdd=0;
23 template<class T> unsigned int Nombre<T>::nbDiv=0;
24 template<class T> unsigned int Nombre<T>::nbCos=0;
25
26 template<class T>
27 Nombre<T>::Nombre(const T& x) {
28   val=x;
29 }
30
31 template<class T> Nombre<T>
32 Nombre<T>::operator+(const Nombre<T>& v) {
33   ++nbAdd;
34   return Nombre(val+v.val);
35 }
36
37 template<class T> Nombre<T>
38 Nombre<T>::operator/(const Nombre<T>& v) {
39   ++nbDiv;
40   return Nombre(val/v.val);
41 }
42
43 template<class T> Nombre<T>
44 cos(const Nombre<T>& v) {
45   ++Nombre<T>::nbCos;
46   return Nombre<T>(::cos(v.val));
47 }
48
49 template<class T> ostream&
50 operator<<(ostream& os, const Nombre<T>& v) {
51   os << v.val;
52   return os;
53 }
54
55 int
56 main() {
57   Nombre<double> x(5.0), y(6.0);
58
59   cout << x << "+" << y << "=" << x+y << endl;
60   cout << x << "/" << y << "=" << x/y << endl;
61   cout << "cos(" << x << ")=" << cos(x) << endl;
62
63   Nombre<int> z(5), t(6);
64
65   cout << z << "+" << t << "=" << z+t << endl;
66   cout << z << "/" << t << "=" << z/t << endl;
67   cout << "cos(" << z << ")=" << cos(z) << endl;
68 }
69
```

Dans le cas présent, le comportement de la classe `Nombre<int>` est peut-être celui attendu par l'utilisateur. Mais comment spécifier dans le cas général que le type de retour d'une certaine opération doit être différent du type `T` utilisé pour l'instanciation ? Comment indiquer la fonction à utiliser suivant le type `T` ?

Pour pallier ce problème, Nathan MYERS [173] a mis au point une méthode élégante dite *méthode des traits*, désormais utilisée dans les librairies standard de C++ et décrite ci-dessous.

11.1.2 Les *traits*

Un *trait* est une classe paramétrée collectant des types et des méthodes, comme le fait une *interface* en Java [90] ou une *signature* [20] dans le formalisme de BAUMGARTNER et RUSSO. Mais, à la différence des *interfaces* et des *signatures*, un *trait* sert à exporter des noms et non des profils. Considérons l'exemple suivant :

Exemple 11.32 (Caractéristiques de types flottants [173]). *On souhaite écrire une classe paramétrée* `Matrice` *de nombres flottants. Pour les besoins de certaines méthodes, la classe a besoin de faire référence à l'exposant maximum du type passé à l'instanciation, ainsi qu'à la valeur de son* epsilon*. *Le programme 11.2 montre la solution à ce problème avec des traits : on définit la classe* `float_traits` *que l'on spécialise pour chaque type flottant que l'on souhaite employer. La classe matrice utilise la fonction* `epsilon()` *et la*

*L'epsilon d'un type flottant est le plus petit nombre positif ε du type tel que $1 + \varepsilon > 1$ [222, p. 8].

constante `max_exponent` *de manière uniforme quel que soit le type de flottant utilisé pour son instanciation. On remarquera que la signature de la fonction* `epsilon()` *dépend du type instanciant le trait. Ainsi, la classe* `float_traits` *exporte uniquement le nom* `epsilon()` *de la méthode ; cette caractéristique fait tout l'intérêt de l'emploi des traits.*

PROG. 11.2: Un exemple d'utilisation de *traits*

```
1  template<class T>
2  struct float_traits {
3  };
4
5  // Spécialisation pour "float"
6  struct float_traits<float> {
7    typedef float float_type;
8    enum {
9      max_exponent = FLT_MAX_EXP
10   };
11   static inline float_type epsilon() {
12     return FLT_EPSILON;
13   }
14 };
15
16 // Spécialisation pour "double"
17 struct float_traits<double> {
18   typedef double float_type;
19   enum {
20     max_exponent = DBL_MAX_EXP
21   };
22   static inline float_type epsilon() {
23     return DBL_EPSILON;
24   }
25 };
26
27 template<class T>
28 class Matrice {
29 public:
30   ...
31   void methode() {
32     ...
33     T x=float_traits<T>::epsilon()+
34        float_traits<T>::max_exponent;
35     ...
36   }
37   ...
38 };
39
```

Le programme 11.3 décrit une version modifiée du programme 11.1 où l'on a introduit les *traits* afin de permettre à l'utilisateur de spécifier librement le type de retour de chaque opérateur ainsi que les fonctions arithmétiques à employer. On notera la présence de trois instanciation du *trait* `nb_traits` pour les types `short int`, `int` et `double` (lignes 13 à 46).

Si l'on exécute le programme 11.3, on obtient le résultat :

```
5 + 6 = 11
5 / 6 = 0.833333
cos(5) = 0.283662
5 + 6 = 11
5 / 6 = 0.833333
cos(5) = 0.283662
```

En spécialisant le *trait* `nb_traits` pour le type `int`, on a spécifié qu'il fallait utiliser le type `double` pour toutes les méthodes de la classe `Nombre` retournant un `TypeReel`. On notera aussi la présence de la constante `initialisateur` déclarée par le *trait* et utilisée pour initialiser par défaut un objet `Nombre`. Enfin, il est important de remarquer que chaque type primitif indique dans sa spécialisation de `nb_traits` quelle est la fonction `cos()` à utiliser. Ainsi, c'est la fonction de la librairie mathématique standard `double cos(double)` qui sera utilisée pour les types `int` et `double`, alors que l'on appellera une fonction `float cos(short int)` pour le type `short int`. On voit ainsi que l'emploi des *traits* permet d'optimiser finement le code en autorisant l'appel de fonctions spécialisées tirant partie des caractéristiques de chaque type, tout en présentant une interface uniforme aux classes qui s'en servent.

11.2 Fonctionnalités et implémentation de JAIL

JAIL offre à l'utilisateur les principales fonctions définies dans la spécification de CHIRIAEV et WALSTER [44] (*Basic Interval Arithmetic Specification*, BIAS) :

- opérateurs de base : $+, -, \times, \div, \sqrt{\ }$;
- opérateurs trigonométriques : $\sin, \cos, \tan, \mathrm{acos}, \ldots$;
- opérateurs hyperboliques : $\sinh, \cosh, \mathrm{acosh}, \ldots$;

PROG. 11.3: La classe `Nombre` avec *traits*

```
1  #include <iostream.h>
2  #include <math.h>
3
4  template<class T>
5  struct nb_traits {
6    typedef T TypeDeBase;
7    typedef T TypeReel;
8    static T cos(T x);
9    static const T initialiseur;
10 };
11
12 // short int
13 template<>
14 struct nb_traits<short int> {
15   typedef short int TypeDeBase;
16   typedef float TypeReel;
17   static float cos(short int x) {
18     return ::cos(float(x));
19   }
20   static const short int initialiseur;
21 };
22 template<> const short int
23 nb_traits<short int>::initialiseur=0;
24
25 // int
26 template<>
27 struct nb_traits<int> {
28   typedef int TypeDeBase;
29   typedef double TypeReel;
30   static double cos(int x) {
31     return ::cos(double(x));
32   }
33   static const int initialiseur;
34 };
35 template<> const int
36 nb_traits<int>::initialiseur=0;
37
38 // double
39 template<>
40 struct nb_traits<double> {
41   typedef double TypeDeBase;
42   typedef double TypeReel;
43   static double cos(double x) {
44     return ::cos(x);
45   }
46   static const double initialiseur;
47 };
48 template<> const double
49 nb_traits<double>::initialiseur=0.0;
50
51 template<class T>
52 class Nombre {
53 public:
54   typedef typename nb_traits<T>::TypeDeBase
55                                   TypeDeBase;
56   typedef typename nb_traits<T>::TypeReel Reel;
57   Nombre(const T& v);
58   Nombre();
59   T getVal() const;
60   Nombre<T> operator+(const Nombre<T>& x);
61   Nombre<Reel> operator/(const Nombre<T>& v);
62   friend Nombre<Reel> cos<>(const Nombre<T>& v);
63   friend ostream& operator<< <>(ostream& os,
64                                 const Nombre& v);
65 private:
66   T val;
67   static unsigned int nbAdd;
68   static unsigned int nbDiv;
69   static unsigned int nbCos;
70 };
71
72 template<class T> unsigned int Nombre<T>::nbAdd=0;
73 template<class T> unsigned int Nombre<T>::nbDiv=0;
74 template<class T> unsigned int Nombre<T>::nbCos=0;
75
76 template<class T>
77 Nombre<T>::Nombre(const T& x) {
78   val=x;
79 }
80
81 template<class T>
82 Nombre<T>::Nombre() {
83   val=nb_traits<T>::initialiseur;
84 }
85
86 template<class T> T
87 Nombre<T>::getVal() const {
88   return val;
89 }
90
91 template<class T> Nombre<T>
92 Nombre<T>::operator+(const Nombre<T>& v) {
93   ++nbAdd;
94   return Nombre(val+v.val);
95 }
96
97 template<class T>
98 Nombre<typename nb_traits<T>::TypeReel>
99 Nombre<T>::operator/(const Nombre<T>& v) {
100   typedef typename
101     nb_traits<T>::TypeReel TypeReel;
102   ++nbDiv;
103   return Nombre<TypeReel>(TypeReel(val)/
104                           TypeReel(v.val));
105 }
106
107 template<class T>
108 Nombre<typename nb_traits<T>::TypeReel>
109 cos(const Nombre<T>& v) {
110   typedef typename
111     nb_traits<T>::TypeReel TypeReel;
112   ++Nombre<T>::nbCos;
113   return
114    Nombre<TypeReel>(nb_traits<T>::cos(v.val));
115 }
116
117 template<class T> ostream&
118 operator<<(ostream& os, const Nombre<T>& v) {
119   os << v.val;
120   return os;
121 }
122
123 int
124 main() {
125   Nombre<double> x(5.0), y(6.0);
126   cout << x << "+" << y << "=" << x+y << endl;
127   cout << x << "/" << y << "=" << x/y << endl;
128   cout << "cos(" << x << ")=" << cos(x) << endl;
129
130   Nombre<int> z(5), t(6), u;
131   cout << z << "+" << t << "=" << z+t << endl;
132   cout << x << "/" << y << "=" << z/t << endl;
133   cout << "cos(" << z << ")=" << cos(z) << endl;
134 }
135
```

– autres opérateurs transcendantaux : $\exp, \log, \ldots$

La librairie ne contient pour l'instant que l'ensemble des *possibly relations* de la BIAS, correspondant aux relations utilisées en programmation par contraintes d'intervalles, la première application de JAIL. Elle contient en plus les versions relationnelles de certains opérateurs (utilisées au chapitre 4 et pour calculer la hull-consistance de contraintes primitives — *cf.* chap. 3) :

– sqrInv : $\mathsf{sqrInv}(I, J) = \mathsf{Hull}(\{\mathsf{r} \in J \mid \exists \mathsf{s} \in I : \mathsf{r}^2 = \mathsf{s}\})$;
– div (division relationnelle) : $\mathsf{div}(I, J, K) = \{k \in K \mid \exists i \in I, j \in J : i = kj\}$;
– ...

11.2.1 Architecture générale

Le cœur de JAIL est la classe paramétrée `Interval`. Une instanciation pour le type de base `double` (`Interval<double>`) est incluse dans la librairie. L'utilisateur peut à sa guise décider d'instancier la classe avec un autre type de nombre flottant. Il lui faut pour cela fournir une spécialisation du *trait* `JailRealTraits` (le programme 11.4 présente une portion d'une telle spécialisation pour le type `double`). Nous avons choisi d'utiliser la méthode des *traits* présentée ci-dessus car un *traits* permet de rendre explicite les propriétés et les méthodes que doit posséder le type de base. Ce n'est pas le cas des macros, par exemple. En particulier, nous obligeons ainsi l'utilisateur à fournir pour son type des versions de chaque fonction arithmétique arrondies par défaut et par excès (*e.g.* les fonctions `cosDn` et `cosUp` dans le programme 11.4). À charge pour lui d'implémenter comme il le souhaite ces fonctions ; en ce qui concerne l'instanciation pour les `double`s fournie avec la librairie, nous nous sommes reposés sur le standard IEEE 754 pour les opérations de base $+, -, \times, \div, \sqrt{}$. Quant aux fonctions non prises en compte par le standard, nous avons utilisé une procédure d'*epsilon-inflation* : on retranche ou ajoute quelques ulps (*unit in the last place*, *cf.* p. 7) au résultat en fonction du sens de l'arrondi souhaité. Cela suppose de connaître une borne sur la précision des opérateurs présents dans la librairie. Pour les machines basées sur une architecture ix86, nous nous sommes référés aux valeurs données par Intel [56] ; ne possédant pas les informations nécessaires pour les autres processeurs, nous avons choisi d'employer la librairie gratuite `fdlibm` de SUN Microsystems, où la précision de chaque fonction est indiquée.

PROG. 11.4: Instanciation du trait de JAIL pour le type `double`

```
1  struct JailRealTraits<double> {
2    typedef double JailReal;
3    static inline double NextFloat(const double& x);
4    static inline double PrevFloat(const double& x);
5    static const double EPSILON;
6    static const double MINREAL;
7    static const double MAXREAL;
8    static const double PInf;
9    static const double MInf;
10   static const double TwoPi;
11   static const double Pi;
12   static const double PiHalf;
13   static const double PiDn;
14   static const double PiUp;
15   static const double HalfPiDn;
16   static const double HalfPiUp;
17   static const double QNaN;
18   static inline double RoundUp(const double& x);
19   static inline double RoundDn(const double& x);
20   static inline bool isNotFinite(const double& x);
21   static inline bool isFinite(const double& x);
22   static inline bool isNaN(const double& x);
23   static inline bool isMinusZero(const double& x);
24   static inline bool isPlusZero(const double& x);
25   static inline bool isSigned(const double& x);
26   static inline void strToReal(const char *const str,
27                                double* x);
28   static inline double floor(const double& x);
29   static inline double ceil(const double& x);
30   static inline double sqrt(const double& x);
31   static inline double expDn(const double& x);
32   static inline double logDn(const double& x);
33   static inline double cosDn(const double& x);
34   static inline double sinDn(const double& x);
35   static inline double tanDn(const double& x);
36   static inline double powDn(const double& x,
37                              unsigned int a);
38   static inline double expUp(const double& x);
39   static inline double logUp(const double& x);
40   static inline double cosUp(const double& x);
41   static inline double sinUp(const double& x);
42   static inline double tanUp(const double& x);
43   static inline double powUp(const double& x,
44                              unsigned int a);
45   static inline const double& MIN(const double& a,
46                                   const double& b);
47   static inline const double& MAX(const double& a,
48                                   const double& b);
49   static inline const double& ABS(const double& a);
50   ...
51 };
52
```

L'utilisateur doit aussi donner une représentation de chacune des constantes dont nous pouvons avoir besoin dans l'implémentation des fonctions de l'arithmétique d'intervalles ($\pi, \pi/2, \mathsf{NaN}, \ldots$), ainsi que les fonctions

d'affichage. De cette façon, les méthodes de la classe `Interval` ne font jamais directement appel à des fonctions ou des constantes implicitement déclarées (au risque de se tromper de versions, du fait de promotions de types involontaires), mais font explicitement référence aux fonctionnalités du type de base d'instanciation déclarées dans le *trait*. Nous verrons dans la section 11.3 qu'une telle indirection systématique ne diminue pas les performances globales.

Le programme 11.5 présente un exemple d'utilisation de la librairie JAIL (nous reparlerons du *trust rounding* dans la section suivante). On notera l'emploi du constructeur `Interval("0.9")` pour créer un intervalle canonique contenant la valeur 0,9 non représentable en machine (le passage par une chaîne de caractères est indispensable pour éviter que le nombre ne soit arrondi suivant l'arrondi par défaut à la compilation). Comme on le voit sur l'exemple, on peut librement mixer valeurs rationnelles et intervalles. La librairie contient des versions spécialisées des différents opérateurs pour tirer partie quand cela est possible des occurrences de constantes rationnelles. L'ensemble des classes, fonctions et constantes de la librairie est inclus dans l'espace de nom `Jail` afin d'éviter des conflits de noms avec d'autres librairie. C'est pourquoi l'on exporte le type `Jail::interval<double>` en le nommant `Interval` (ligne 5) pour simplifier l'écriture.

JAIL est actuellement portée sur les plate-formes suivantes :
- ix86 sous Linux ;
- Sparc/UltraSparc sous SunOs 4.x et Solaris 2.5 ;
- SGI sous IRIX.

Elle a été développée de façon à être aisément portée sur d'autres architectures sous Unix grâce à l'utilisation des outils GNU (autoconf).

PROG. 11.5: Évaluation de la fonction de `Makino/Berz` [155]

```
1  #include <iostream.h>
2  #include "jail"
3
4  template class Jail::Interval<double>;
5  typedef Jail::Interval<double> Interval;
6
7  int
8  main() {
9    Jail::RoundUpward(); // Trust rounding
10
11   Interval
12     x1("1.95","2.05"),
13     x2("0.95","1.05"),
14     x3("0.95","1.05");
15
16   cout << (4*tan(3*x2))/(3*x1+
17                          x1*sqrt(6*x1/(-7*(x1-8))))
18     -120-2*x1-7*x3*(1+2*x2)
19     -sinh(0.5+(6*x2)/(8*x2+7))
20     +sqr(3*x2+13)/(3*x3)-20*x3*(2*x3-5)
21     +(5*x1*tanh(Interval("0.9")*x3))/
22     sqrt(5*x2)-20*x2*sin(3*x3) << endl;
23 }
24
```

11.2.2 Implémentation des opérateurs

Les unités pour l'arithmétique flottante (FPUs) modernes travaillent en *flux continu* (architecture pipe-linée) : à tout instant, le FPU contient plusieurs instructions à différents stades de leur décodage. Pour changer le sens de l'arrondi, le FPU doit se vider de toutes les instructions qu'il contient, puis modifier un *flag*, avant de réadmettre de nouvelles instructions. Cette rupture de pipe-line est la source d'une dégradation des performances pour le calcul flottant. Afin de limiter ce facteur, nous avons choisi d'implémenter JAIL de la façon suivante : les opérations $+, -, \times, \div$ sont toutes faites dans le mode « arrondi vers $+\infty$ » en utilisant l'équivalence :

$$\lfloor x \top y \rfloor = -\lceil (-x) \top y \rceil, \quad \text{pour } \top \in \{+, -, \times, \div\}$$

Pour la racine carrée, on ne peut bien évidemment pas utiliser la formule ci-dessus. Afin de ne pas avoir à gérer spécialement ce cas-là, nous avons utilisé la proposition suivante :

Proposition 11.1. *Pour tout nombre flottant négatif x, on a la relation :*

$$-\left\lceil \frac{x}{\lceil \sqrt{-x} \rceil} \right\rceil \leqslant \lfloor \sqrt{-x} \rfloor \tag{11.1}$$

Démonstration. La relation (11.1) se prouve simplement comme suit :

$$\begin{array}{lll} & \lceil\sqrt{-x}\rceil \geqslant \sqrt{-x} & \text{par propriété de } \lceil\ \rceil \\ \text{puis :} & \frac{x}{\sqrt{-x}} \leqslant \frac{x}{\lceil\sqrt{-x}\rceil} & \text{car } x \text{ est négatif} \\ \text{et :} & \frac{x}{\sqrt{-x}} \leqslant \frac{x}{\lceil\sqrt{-x}\rceil} \leqslant \left\lceil \frac{x}{\lceil\sqrt{-x}\rceil} \right\rceil & \text{par propriété de } \lceil\ \rceil \\ \text{enfin :} & -\left\lceil \frac{x}{\lceil\sqrt{-x}\rceil} \right\rceil \leqslant -\frac{x}{\sqrt{-x}} & \\ \textit{i.e.} \text{ :} & -\left\lceil \frac{x}{\lceil\sqrt{-x}\rceil} \right\rceil \leqslant \sqrt{-x} & \end{array}$$

La relation (11.1) découle alors du fait que $\lfloor\sqrt{-x}\rfloor$ est le plus grand flottant inférieur ou égal à $\sqrt{-x}$. ■

La proposition 11.1 nous offre le moyen de calculer la borne gauche de la racine carrée d'un intervalle sans avoir à modifier le sens de l'arrondi.

Par commodité et pour nous éviter de faire toujours deux changements de signe, on choisit de représenter la borne gauche par son opposé. Ainsi, l'intervalle $[-6\,..\,4]$ est stocké sous la forme $\langle 6, 4\rangle$.

On obtient ainsi le code suivant pour l'addition de deux intervalles :

```
Addition(entrée I1, I2; sortie I3 = I1 + I2)
début
    RoundUp()
    I3 ← −(I1 + I2)
    I3 ← I1 + I2
fin
```

Avec cette technique, on divise environ par deux le nombre de changements de sens de l'arrondi à effectuer. Mais il est possible de faire mieux si l'on s'arrange pour que l'arrondi par excès soit le mode par défaut à la place de l'arrondi au plus proche-pair. Pour cela, il suffit de positionner l'arrondi sur ce mode avant tout calcul et de faire en sorte que toutes les méthodes qui ont absolument besoin de modifier le sens de l'arrondi (typiquement les fonctions d'entrée/sortie) le repositionnent vers $+\infty$ à la fin de leur traitement. Avec cette optimisation, on peut faire en sorte que tous les calculs soient effectués avec un seul changement de sens de l'arrondi. C'est ce que nous appelons dans la suite le *trust rounding*, car la librairie fait confiance aux méthodes pour préserver le sens de l'arrondi. JAIL offre la possibilité de choisir entre le *trust rounding* (*trust on*) et le changement explicite à l'entrée de chaque fonction (*trust off*) lors de l'initialisation préliminaire à la compilation de la librairie.

Comme les NaNs se propagent dans les calculs et qu'une opération arithmétique dont un opérande est vide doit retourner un intervalle vide, nous avons jugé intéressant de représenter l'intervalle vide par $[NaN\,..\,NaN]$. Plus généralement, un intervalle vide dans JAIL est un intervalle dont l'une des bornes au moins est un NaN ou bien dont la borne gauche est strictement supérieure à la borne droite. Une telle représentation suppose d'écrire soigneusement les prédicats sur les intervalles tels que `isEmpty()` : les NaNs étant incomparables, il nous faut nier les tests. On a par exemple le code suivant :

```
isEmpty(entrée: I1; sortie: I1 == ∅)
début
    retourner (¬(−I1 ⩽ I1))
fin
```

Si l'une des bornes de I_1 est un NaN, le test $(-\underline{I_1} \leqslant \overline{I_1})$ sera automatiquement faux et l'on retournera donc bien **vrai**.

Un autre avantage à la gestion explicite des NaNs de par leur utilisation pour représenter un intervalle vide est que l'on ne risque pas de faire disparaître un NaN lors d'un calcul (ce qui pourrait faire croire que tout c'est bien passé alors qu'une erreur a eu lieu) si l'on prend la précaution de tester au début de chaque fonction si l'un des opérandes est vide. On pourrait aussi envisager d'utiliser les bits inutilisés dans la représentation des NaNs pour stocker de l'information permettant d'expliquer a posteriori pourquoi un intervalle est vide.

11.3 Évaluation de JAIL

Nous présentons dans la suite les résultats de tests effectués pour tester la validité des choix d'implémentation et pour comparer JAIL avec les autres librairies C++ disponibles tant du point de vue de l'efficacité que de la correction. Nous avons utilisé pour cela deux ordinateurs d'architectures différentes respectant la norme IEEE 754 afin de mettre à jour d'éventuelles différences de comportement :

- une SUN UltraSparc 1/167 MHz avec 64 Mo de RAM, sous Solaris 2.5 ;
- un DELL PowerEdge 4300 avec deux processeurs Pentium III 500 MHz et 512 Mo de RAM, sous Red Hat Linux 6.1.

Les tableaux 11.3, 11.4, 11.5 et 11.6 utilisent le jeu de tests suivant : afin de tester intensivement la totalité du code des différentes fonctions d'intervalles de base, nous avons isolé un certain nombre de valeurs $t_i, i \in \{1, \ldots, 71\}$, données dans la table 11.1, à partir desquelles nous avons construit les intervalles $I_k = [t_i \,..\, t_j]$ pour $i \in \{1, 70\}, j \in \{i+1, 71\}$.

TAB. 11.1: Valeurs utilisées pour les tests

$-\infty$	-MAX_DOUBLE	-4.54535e16	-4.54534e16	-4.54533e16
-4.54532e16	-1.0e8	-10003	-10002	-10001
-823.5	-563	-562.5	-562	-560
-345.8	-123.125	-29	-28.7	-12.0
-PiUp	-PiDn	-2.0	-HalfPiUp	-HalfPiDn
-1.5	-1.4	-1.3	-1.2	-1.1
-1.0	-1.0e-8	-EPSILON	-MINREAL	-0.0
0.0	MINREAL	EPSILON	1.0e-8	1.0
1.1	1.2	1.3	1.4	1.5
HalfPiDn	HalfPiUp	2.0	PiDn	PiUp
12.0	28.7	29	123.125	345.8
560	561.5	562	562.5	563
823.5	10001	10002	10003	1.0e8
4.54532e16	4.54533e16	4.54534e16	4.54535e16	MAXREAL
$+\infty$				

Nous avons ensuite testé chaque fonction en affectant à chaque opérande toutes les valeurs de I_k. Chacun de ces tests a été relancé un certain nombre fois pour chaque opérateur. Le tableau 11.2 indique le nombre total d'opérations effectuées pour chaque opérateur testé.

11.3.1 Impact des choix d'implémentation

Pour valider nos choix d'implémentation, nous avons utilisé cinq versions de JAIL :

- une version paramétrée (utilisation de *traits* et d'un patron de classe `Interval` à instancier) en mode *trust on* (le mode « arrondi vers le haut » est considéré comme le mode par défaut) ;
- une version paramétrée en mode *trust off* (mode « arrondi vers le haut » positionné au début de chaque fonction) ;
- une version non paramétrée (`Interval` est une vraie classe dont les bornes sont obligatoirement des `doubles`) en mode *trust on* ;

TAB. 11.2: Nombre d'opérations effectuées par opérateur testé

op.	Nb. essais	Nb. opérations
$x + y$	7	43 226 575
$x - y$	7	43 226 575
$x \times y$	4	24 700 900
$x \div y$	2	12 350 450
$\sqrt{x}$	6 000	14 910 000
x^2	6 000	14 910 000
$x^{-10} \dots x^{10}$	60	3 131 100
e^x	2 000	4 970 000
$\log x$	2 000	4 970 000
$\cos x$	1 000	2 485 000
$\sin x$	1 000	2 485 000
$\tan x$	500	1 242 500

– une version non paramétrée en mode *trust off* ;
– une version non paramétrée en mode *RndDn/RndUp* (changement du sens de l'arrondi en fonction de la borne à calculer).

Les tableaux 11.3 et 11.4 synthétisent les résultats des tests pour nos deux machines.

TAB. 11.3: Impact des choix d'implantation de JAIL (version paramétrée)

	UltraSparc 1/167 MHz		Bi-Pentium III 500 MHz	
	`trust on`	`trust off`	`trust on`	`trust off`
$x + y$	19 204	24 175	9 470	10 930
$x - y$	17 526	24 812	9 510	11 230
$x \times y$	17 220	22 559	9 970	10 480
$x \div y$	33 105	33 563	4 880	4 900
$\sqrt{x}$	12 829	14 451	5 500	5 670
x^2	7 309	9 378	3 780	4 260
$x^{-10} \dots x^{10}$	13 141	13 913	2 000	2 110
e^x	23 915	23 182	7 490	7 330
$\log x$	18 404	17 934	3 980	3 760
$\cos x$	19 218	19 577	2 140	2 160
$\sin x$	21 289	21 628	2 270	2 270
$\tan x$	14 807	14 970	950	980
compilation	11 200	13 300	4 000	4 010

Temps en millisecondes

On peut noter au préalable que les fonctions transcendantales étant arrondies par *epsilon-inflation*, l'impact des différents modes d'arrondi se révèle faible pour elles.

Au vu des résultats, on constate certaines disparités suivants les machines : globalement, la version paramétrée s'avère plus rapide que la version non paramétrée, et le mode *trust on* plus efficace que tous les autres modes. Cependant, il apparaît que la version paramétrée n'entraîne qu'une faible accélération pour le Pentium par rapport à la version non paramétrée alors que l'on constate un gain supérieur à $70\,\%$ pour les opérations de base sur l'UltraSparc. Par ailleurs, les temps de compilation pour la version paramétrée sont environ 2,7 fois plus importants pour la version paramétrée, quelle que soit la machine considérée. Après étude des fichiers générés par les différentes versions de JAIL (avec/sans *traits*), il semble que l'efficacité de la version paramétrée soit due au fait que

TAB. 11.4: Impact des choix d'implantation de JAIL (version non paramétrée)

	UltraSparc 1/167 MHz			Bi-Pentium III 500 MHz		
	`trust on`	`trust off`	`RndDn/RndUp`	`trust on`	`trust off`	`RndDn/RndUp`
$x + y$	34 445	41 281	49 280	9 950	11 630	12 150
$x - y$	34 543	41 311	42 840	9 730	11 180	10 620
$x \times y$	29 709	35 383	38 121	8 570	9 130	9 590
$x \div y$	39 561	40 634	41 792	4 520	4 550	4 790
$\sqrt{x}$	21 412	22 942	26 693	5 950	6 160	5 700
x^2	13 827	15 778	18 071	3 480	4 050	4 240
$x^{-10} \dots x^{10}$	15 997	16 572	11 632	2 460	2 550	2 540
e^x	26 262	24 810	24 545	8 710	8 550	8 540
$\log x$	21 069	20 769	20 860	3 990	3 840	3 700
$\cos x$	19 951	20 421	20 766	2 270	2 290	2 360
$\sin x$	22 220	22 581	32 663	2 440	2 450	3 910
$\tan x$	15 660	15 608	16 013	940	930	1 010
compilation	4 200	4 100	4 000	1 480	1 450	1 460

Temps en millisecondes

l'utilisation des *traits* permet une plus grande expansion du code « en ligne » et donc une suppression du surcoût lié à l'appel des méthodes C++. De plus, la librairie étant compilée en même temps que l'application qui l'utilise, le compilateur a une vision plus globale du programme et peut appliquer des optimisations plus agressives. Il faut noter que cela se fait au détriment du temps de compilation et de la taille du code créé (le fichier objet est trois fois plus gros avec les *traits*). On peut cependant espérer que la généralisation des compilateurs C++ utilisant des fichiers d'en-tête (*header files*) précompilés permettra de réduire ces inconvénients.

Au final, on peut considérer que pour des applications ou le temps de compilation est moins crucial que le temps d'exécution (ce qui devrait être le cas de la majorité d'entre elles…), la version paramétrée de JAIL avec le mode *trust on* s'avère le meilleur choix. C'est aussi celle qui offre le plus de souplesse en autorisant l'utilisateur à choisir le type des bornes de ses intervalles.

11.3.2 Comparaison avec d'autres librairies

Nous avons testé la correction des librairies JAIL, Profil [134], RVInterval [236], fi_lib [109] et fi_lib++ [149] en évaluant cinq fonctions tirées d'un article de PRIEST [191]. Comme on pourra le constater dans les tableaux 11.5 et 11.6, les librairies RVInterval et fi_lib retournent des résultats incorrects pour plusieurs de ces fonctions et ont des comportements différents suivant la machine considérée.

Rappelons que, contrairement à JAIL, aucune de ces librairies n'offre la possibilité de choisir le type des bornes des intervalles. Pour toutes, le type `double` est le défaut.

Dans les tableaux 11.5 et 11.6, la valeur « — » indique que le résultat correspondant n'a pu être calculé car la librairie ne disposait pas de la fonction nécessaire ; le mot « CORE » signale que le programme a été brutalement interrompu par une erreur fatale.

Au vu des résultats, on s'aperçoit que le plus haut niveau de fonctionnalités de JAIL ne le pénalise nullement par rapport aux autres librairies (avec cependant une certaine faiblesse au niveau des fonctions trigonométriques). Notons que le bon résultat obtenu par Profil pour la puissance est obtenu au détriment de la qualité des valeurs calculées, cette librairie implémentant x^n par une adaption à l'exponentiation de la *méthode du paysan russe* [136, p. 462] (itération de multiplications).

Nous n'avons pas testé Profil sur le Pentium car nous n'avons pu réussir à compiler cette librairie sur cette machine.

Afin de pouvoir comparer JAIL à fi_lib++ alors que cette librairie n'est pas encore disponible, nous avons repris le test décrit par LERCH et WOLFF VON GUDENBERG [149] consistant à appliquer une des quatres opérations de

TAB. 11.5: Comparaisons de librairies pour l'arithmétique d'intervalles sur une UltraSparc 1/167 MHz

	Profil	RVInterval	fi_lib	JAIL†
$x + y$	26 362	53 400	48 479	19 204
$x - y$	26 317	53 102	47 637	17 526
$x \times y$	25 951	40 274	35 758	17 220
$x \div y$	35 521	38 245	39 538	33 105
$\sqrt{x}$	321 568	12 382	23 853	12 829
x^2	16 426	26 819	14 133	7 309
$x^{-10} \ldots x^{10}$	6 698	25 747	—	13 141
e^x	201 797	23 599	11 605	23 915
$\log x$	119 980	20 195	10 940	18 404
$\cos x$	109 844	7 963	6 295	19 218
$\sin x$	91 883	16 982	5 953	21 289
$\tan x$	41 657	—	4 117	14 807
$ab((1/a)^2 - (1/b)^2)$ $a = 10^{-175}, b = 2 \cdot 10^{-175}$	$[-\infty \,..\, +\infty]$	$[-\infty \,..\, +\infty]$	$[-9{,}88 \cdot 10^{-324} \,..\, 2{,}47 \cdot 10^{-149}]$	$[-\infty \,..\, +\infty]$
$y(1/(-\sqrt{1/(x-1)}) + 1)$ $x \in [10^{-310} \,..\, 1], y \in [-1 \,..\, 1]$	$[-\infty \,..\, +\infty]$	$[-1{,}0 \cdot 10^{308} \,..\, 1{,}0 \cdot 10^{308}]$	$[+\infty \,..\, -\infty]$	$[-\infty \,..\, +\infty]$
$1/(s^2t^2 + 1)$ $s \in [10^{-200} \,..\, 1], t \in [1 \,..\, 10^{200}]$	$[0 \,..\, 1]$	$[0 \,..\, 1]$	$[9{,}999 \cdot 10^{-201} \,..\, 1{,}001\ 1]$	$[0 \,..\, 1]$
$\sum_{i=1}^{6} \sin(ix)$ $x \in [9 \,..\, 10]$	$[-5{,}295 \,..\, 5{,}281\ 5]$	$[-5{,}295 \,..\, 5{,}281\ 5]$	$[-5{,}295 \,..\, 5{,}281\ 5]$	$[-5{,}295 \,..\, 5{,}281\ 5]$
$(1 + x^2)/\sqrt{1 + x^3}$ $x = 10^{300}$	$[0 \,..\, +\infty]$	$[0 \,..\, +\infty]$	$[-9{,}88 \cdot 10^{-324} \,..\, 1{,}48 \cdot 10^{-323}]$	$[0 \,..\, +\infty]$
compilation	7600	6800	8900	11200

Résultats en millisecondes avec egcs 2.91.66
†*Version paramétrée avec* `trust on`

TAB. 11.6: Comparaisons de librairies pour l'arithmétique d'intervalles sur un Bi-Pentium III 500 MHz

	RVInterval	fi_lib	JAIL†
$x + y$	23 770	24 610	9 470
$x - y$	23 820	23 650	9 510
$x \times y$	20 640	17 250	9 970
$x \div y$	7 410	6 690	4 880
$\sqrt{x}$	7 000	12 060	5 500
x^2	9 030	7 620	3 780
$x^{-10} \ldots x^{10}$	14 270	—	2 000
e^x	8 270	5 990	7 490
$\log x$	7 300	CORE	3 980
$\cos x$	3 280	1 680	2 140
$\sin x$	2 260	1 720	2 270
$\tan x$	—	1 330	950
$ab((1/a)^2 - (1/b)^2)$ $a = 10^{-175}, b = 2 \cdot 10^{-175}$	$[-\infty \,..\, \infty]$	$[1{,}061 \cdot 10^{-139} \,..\, 1{,}061 \cdot 10^{-139}]$	$[-\infty \,..\, +\infty]$
$y(1/(-\sqrt{1/(x-1)}) + 1)$ $x \in [10^{-310} \,..\, 1], y \in [-1 \,..\, 1]$	$[-\infty \,..\, +\infty]$	$[0{,}997 \,..\, -0.\ 997]$	$[-\infty \,..\, +\infty]$
$1/(s^2t^2 + 1)$ $s \in [10^{-200} \,..\, 1], t \in [1 \,..\, 10^{200}]$	$[0 \,..\, 1]$	$[9{,}999\ 99 \cdot 10^{-201} \,..\, 1{,}000\ 1]$	$[0 \,..\, 1]$
$\sum_{i=1}^{6} \sin(ix)$ $x \in [9 \,..\, 10]$	$[-5{,}295 \,..\, 5{,}281\ 5]$	$[-5{,}295 \,..\, 5{,}281\ 5]$	$[-5{,}295 \,..\, 5{,}281\ 5]$
$(1 + x^2)/\sqrt{1 + x^3}$ $x = 10^{300}$	$[0 \,..\, +\infty]$	$[NaN \,..\, 7{,}458 \cdot 10^{145}]$	$[0 \,..\, +\infty]$

Résultats en millisecondes avec egcs 2.91.66
†*Version paramétrée avec* `trust on`

base sur des vecteurs d'intervalles remplis aléatoirement et à déterminer le nombre d'opérations sur les intervalles effectuées par secondes. Le tableau 11.7 reprend le tableau donné dans [149] en y ajoutant les deux colonnes de droite. N'ayant pas accès au protocole de test exact utilisé par LERCH et WOLFF VON GUDENBERG, nous avons testé RVInterval, JAIL et fi_lib sur le Pentium III 500 MHz, puis nous avons utilisé le rapport des performances obtenues par fi_lib sur les deux machines pour mettre à l'échelle nos résultats.

TAB. 11.7: Performances en millions d'opérations-intervalles par seconde

op.	fi_lib++⋆	fi_lib⋆	Profil⋆	RVInterval†	JAIL† ‡
+	6,13	3,06	5,58	2,69	15,48
−	5,85	3,09	5,76	2,81	16,32
∗	4,18	3,10	4,67	2,14	8,68
/	3,80	3,11	3,91	2,43	4,45

⋆ *Résultats sur un bi-Pentium III/500 MHz*
† *Résultats sur un bi-Pentium III/500 MHz mis à l'échelle*
‡ *Version paramétrée avec* `trust on`

On constate encore une fois que JAIL est très compétitif par rapport aux autres librairies et qu'elle est en moyenne deux fois plus rapide que fi_lib++.

CHAPITRE 12

Gestion de variables quantifiées : OpAC et WOpAC

Description d'OpAC — évaluation de nos algorithmes — comparaison EIA4 vs. ICAb4

LE BUT DE CE CHAPITRE est de présenter les résultats de tests des algorithmes décrits dans le chapitre précédent. Afin de les évaluer, un prototype de modeleur déclaratif de mouvement de caméra a été implémenté par Marc CHRISTIE au-dessus de WOpAC, une extension d'OpAC — une librairie C++ « ouverte » développée par nos soins durant notre thèse — intégrant les différents algorithmes de gestion de contraintes avec variables universellement quantifiées présentés précédemment.

Nous allons commencer par présenter succinctement l'architecture d'OpAC, puis nous décrirons les jeux de tests utilisés pour valider les algorithmes du chapitre 6 et nous donnerons les temps obtenus pour leur résolution avec la librairie WOpAC de Marc CHRISTIE.

12.1 Présentation d'OpAC

OpAC (*an Open Architecture for Constraints*) est une librairie C++ de résolution de contraintes développée par nos soins afin d'offrir une plate-forme pour diverses expérimentations. Pour cela, elle a été créée dès l'origine pour être aisément extensible et intégrable dans tout logiciel nécessitant la résolution de contraintes. Nous l'avons organisée en six modules correspondant à six classes de base indispensables à la résolution de contraintes par propagation de domaines :

1. la classe `nombre`, définissant le type des valeurs que peut prendre une variable ;
2. la classe `domaine` ;
3. la classe `variable` ;
4. la classe `contrainte`, définissant la forme et la sémantique des contraintes utilisées ;
5. la classe `solveur` chargée de gérer les variables du système de contraintes et de découper leurs domaines jusqu'à ce que l'on obtienne la précision souhaitée ;
6. la classe `propagateur` gérant la réinvocation des contraintes dont certaines des variables ont été modifiées.

Ces classes peuvent être dérivées par l'utilisateur au gré de ses besoins. Il peut ainsi utiliser des variables dont le domaine est représenté par des nombres flottants en précision infinie ou par des `double`s classiques, changer la stratégie de propagation des modifications de domaines…

La figure 12.1 montre les classes dérivées actuellement implémentées en standard dans OpAC ainsi que celles qui le seront à court terme.

Afin de donner un aperçu des possibilités d'OpAC, nous présentons avec le programme 12.1 un exemple utilisant OpAC pour résoudre le problème suivant : trouver les points d'intersection du cercle d'équation $x^2+y^2=1$ et de la parabole d'équation $x^2=y$.

Comme on le voit dans l'exemple décrit par le programme 12.1, il est très simple d'agir sur la résolution d'un problème en changeant simplement le type d'instance utilisée. Il suffit par exemple de remplacer l'objet de la classe `solveRoundRobin` par un objet de la classe `solveLargestFirst` pour que la stratégie FIFO de *splitting*

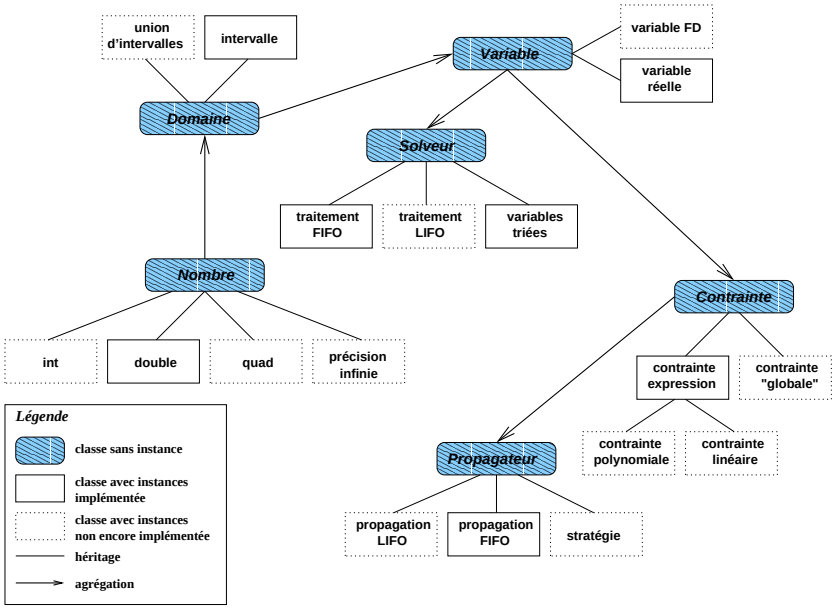


FIG. 12.1: La hiérarchie de classes dans OpAC

soit remplacée par une stratégie ou l'on découpe d'abord les variables dont le domaine est le plus grand. En dérivant une nouvelle classe à partir de la classe `solveur`, l'utilisateur a toute latitude pour définir de nouvelles stratégies de *splitting*. Il en est de même pour l'ordre de réinvocation des contraintes : la classe `propagatorDEQ` propose une stratégie FIFO que l'utilisateur peut remplacer par toute autre stratégie de son choix (réinvocation privilégiée des contraintes « les plus efficaces », des « moins coûteuses »...). On peut aussi définir de nouveaux types de contraintes ou bien spécifier des opérateurs de contraction particuliers pour certaines classes de contraintes (par exemple, la classe `cstrExpression` est à même de gérer des contraintes polynômiales. Il s'avère cependant plus efficace de définir une classe spécialisée `cstrPolynomial` dérivant de `cstrExpression` utilisant l'information supplémentaire pour appliquer des algorithmes spécifiques de résolution plus rapides).

PROG. 12.1: Le problème *circle-parabola* résolu avec OpAC

```
1  int main() {
2    varReal // Déclaration des variables :
3     x(-1.0e8,1.0e8), // Domaine: intervalles à bornes double
4     y(-1.0e8,1.0e8);
5    cstrExpression
6      circle( sqr(x)+sqr(y) == 1), // Cercle de rayon 1
7     parabola(sqr(x) == y);
8    propagatorDEQ store; // Propagateur à gestion en file
9   varDeque vd;
10   solveRoundRobin s; // Solve à gestion en file
11
12   // Addition des contraintes dans le store
13  store.add(circle);
14   store.add(parabola);
15   store.relax(); // On choisit ici de propager
16                  // les domaines après avoir
17                  // ajouté toutes les contraintes
18
19   // Addition des variables dans la liste de solve
20  vd.push_front(&y);
21   vd.push_front(&x);
22   s.consider(store,vd);
23
24   // Résolution par bisection
25  if (s.firstSolution()) {
26     cout << "x = " << x << endl;
27     cout << "y = " << y << endl << endl;
28   }
29   while (s.nextSolution()) {
30     cout << "x = " << x << endl;
31     cout << "y = " << y << endl << endl;
32   }
33 }
34
```

OpAC a pour le moment été étendue par Marc CHRISTIE pour intégrer les algorithmes de gestion de contraintes avec quantificateurs présentés ci-avant (librairie WOpAC). Elle est aussi utilisée en optimisation globale et locale dans le cadre de la thèse de Martine CEBERIO à l'Institut de Recherche en Informatique de Nantes et des travaux

d'Evgueni PETROV (Institut ERSHOV, Novossibirsk) ; une version étendue par nos soins est aussi employée dans le cadre du projet FASE* [201] au CWI (Pays-Bas).

12.2 Évaluation des algorithmes

Nous allons commencer par décrire les différents jeux d'essais utilisés ; nous présenterons certaines méthodes utilisées pour accélérer les calculs, puis nous comparerons les algorithmes EIA4 et ICAb4 en ce qui concerne le temps de résolution.

Dans la suite, chaque *benchmark* est paramétré par le nombre de variables (non compris la variable universellement quantifiée) et le nombre de contraintes à résoudre.

Le problème *School Problem*$_{3,1}$ a été introduit par JARDILLIER et LANGUÉNOU et consiste à trouver toutes les paraboles au-dessus d'une ligne :

$$\forall t \in [0,2] : at^2 + bt + c \geqslant 2t + 1 \quad \text{avec} \quad a \in [0\,..\,1], b \in [0\,..\,1], c \in [0\,..\,1]$$

Le problème *School Problem*$_{3,2}$ en est une variante inconsistante :

$$\forall t \in [0,2] : \begin{cases} at^2 + bt + c \geqslant 2t + 1 \\ at^2 + bt + c \leqslant 2t \end{cases}$$

avec les mêmes domaines pour a, b et c.

Le problème *Flying Saucer*$_{4,1}$ consiste à trouver toutes les paires de points telles que la distance entre la soucoupe volante et la ligne reliant les deux points est supérieure à un seuil d à tout instant t dans un intervalle donné :

$$\sqrt{(x_1 + u(x_2 - x_1) - x_3^t)^2 + (y_1 + u(y_2 - y_1) - y_3^t)^2} \geqslant d$$

avec

$$u = \frac{(x_3^t - x_1)(x_2 - x_1) + (y_3^t - y_1)(y_2 - y_1)}{|P_2 - P_1|^2}$$

où $P_1 = (x_1, y_1)$ et $P_2 = (x_2, y_2)$ sont les inconnues, $P_3^t = (x_3^t, y_3^t)$ les coordonnées de la soucoupe volante à l'instant t, et d la distance minimale à respecter entre la ligne (P_1, P_2) et la soucoupe.

Le problème *Simple Circle* correspond à l'exemple 4.14 : étant donné un point B se déplaçant suivant une trajectoire circulaire, on recherche la position de tous les points A tels que la distance entre A et B est toujours supérieure à une valeur donnée. Les problèmes *Simple Circle*$_{2,2}$ et *Simple Circle*$_{2,3}$ sont des instances du même problème avec respectivement 2 et 3 points se déplaçant suivant des trajectoires circulaires. Pour un seul point, on a :

$$\forall t \in [-\pi\,..\,\pi] : \sqrt{(r_1 \sin t - x)^2 + (r_1 \cos t - y)^2} \geqslant d_1 \qquad \begin{cases} x \in [-5\,..\,5] \\ y \in [-5\,..\,5] \\ d_1 = 0{,}5 \end{cases}$$

où d_1 est la distance minimale requise entre les points A et B et r_1 le rayon de la trajectoire de B.

Le problème *Projection*$_{3,4}$ vérifie si un objet mobile se projette à tout instant dans un *frame* particulier. La caméra considérée possède trois degrés de liberté : x_t^c, y_t^c et θ_t^c. Pour x_t^o, y_t^o, z_t^o les coordonnées à l'instant t de l'objet, et x_t^c, y_t^c, z_t^c, ϕ_t^c, θ_t^c, ψ_t^c, γ_t^c les paramètres de la caméra, on a le système :

$$\begin{aligned}
x'_t &= -(x_t^o - x_t^c)\sin\theta_t^c + (y_t^o - y_t^c)\cos\theta_t^c \\
y'_t &= -(x_t^o - x_t^c)\cos\theta_t^c \sin\phi_t^c + (y_t^o - y_t^c)\sin\phi_t^c \sin\theta_t^c + (z_t^o - z_t^c)\cos\phi_t^c \\
z'_t &= -(x_t^o - x_t^c)\cos\theta_t^c \cos\phi_t^c + (y_t^o - y_t^c)\sin\theta_t^c \cos\phi_t^c + (z_t^o - z_t^c)\sin\phi_t^c \\
{}^{\blacktriangledown}x_t^f &\leqslant x'_t/(z'_t/\gamma_t^c) \qquad {}^{\blacktriangle}x_t^f \geqslant x'_t/(z'_t/\gamma_t^c) \\
{}^{\blacktriangledown}y_t^f &\leqslant y'_t/(z'_t/\gamma_t^c) \qquad {}^{\blacktriangle}y_t^f \geqslant y'_t/(z'_t/\gamma_t^c)
\end{aligned}$$

*http://www-it.et.tudelft.nl/FASE/

avec $t \in [0\,..\,20]$, $x_c \in [-3\,..\,3]$, $y_c \in [-3\,..\,3]$, $z_c = 2$, $\phi_c \in [-0{,}5\,..\,0{,}5]$, $\theta_c = 0$, et $\gamma_c^t = 0{,}8$ et où ${}^{\blacktriangledown}x_t^f$ correspond à l'abscisse de la borne gauche au temps t du *frame* f.

Le problème *Collision*$_{4,1}$ consiste à rechercher toutes les positions (x, y, z) pour une caméra C telles que la distance de la caméra à un objet O se déplaçant suivant la trajectoire :

$$\begin{cases} x_o(t) = 3\sin t\cos t(\sin t - \cos t) \\ y_o(t) = 3\sin t\cos t(\sin t + \cos t) \end{cases}$$

soit toujours supérieure ou égale à 0,5. On prend $x \in [-3\,..\,3]$, $y \in [-3\,..\,3]$, $z \in [0\,..\,3]$. La figure 12.2 présente une interprétation graphique de l'espace des solutions.

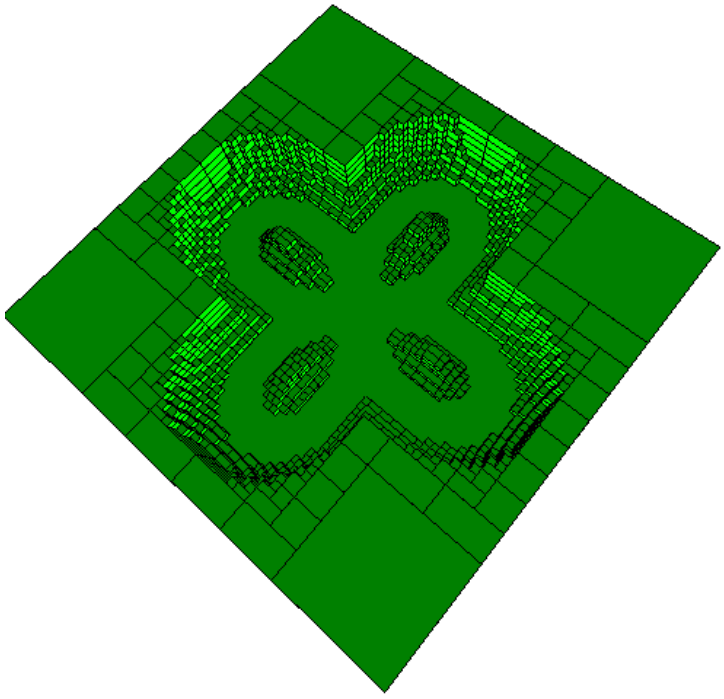


FIG. 12.2: Résolution avec WOpAC du problème *Collision*$_{4,1}$

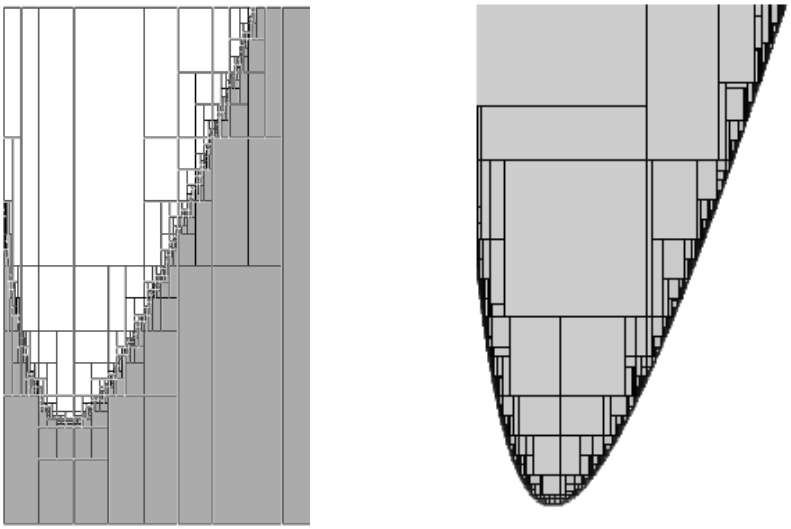
Exemple 12.33 (Premier problème de GARLOFF & GRAF [82]). *On cherche à déterminer la valeur des paramètres v et w pour lesquels le polynôme :*

$$p(s) = s^3 + vs^2 + (w - 5v - 13)s + w$$

est stable. En utilisant un critère dit critère de LIÉNARD-CHIPART, *on transforme ce problème en le système équivalent :*

$$\begin{aligned} v, w &> 0 \\ -5v^2 - 13v + vw - w &> 0 \end{aligned}$$

Suivant GARLOFF *et* GRAF, *nous cherchons une approximation intérieure de la relation $-5v^2 - 13v + vw - w > 0$ pour $v \in [2\,..\,10]$ et $w \in [40\,..\,50]$. La figure 12.3 compare graphiquement les résultats obtenus par* GARLOFF *et* GRAF *en utilisant une expansion de* BERNSTEIN *et ceux obtenus avec WOpAC par l'algorithme* ICAb4.



(a) Résolution par expansion de BERNSTEIN

(b) Résolution avec WOpAC

FIG. 12.3: Premier problème de GARLOFF & GRAF [82]

Exemple 12.34 (Deuxième problème de GARLOFF & GRAF [82]). *Une fois de plus, on recherche la valeur de paramètres garantissant la stabilité d'un système. Le problème à résoudre ici est de trouver les valeurs des variables A, B, D telles que :*

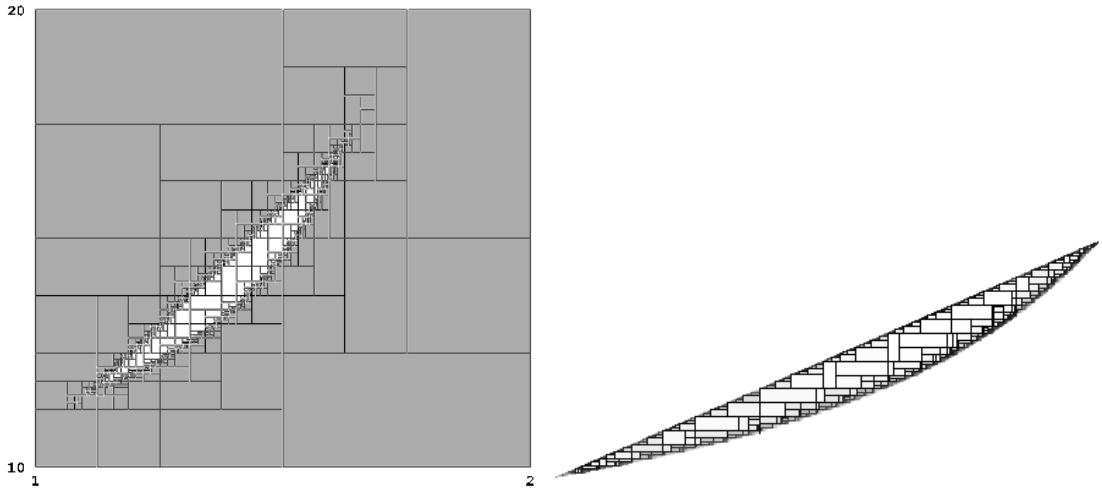
$$\begin{aligned} A, B, D &> 0 \\ AB^2 - D^2 &> 0 \\ -AB + A + D^2 - D - 1 &> 0 \\ AB - AD - 2A + D^3 + 4D^2 + 4D &> 0 \\ AB^3 - AB^2D - 4AB^2 + 2ABD + 4AB + 2BD^3 + 5BD^2 + 2BD - D^3 - 4D^2 - 4D &> 0 \\ AB - 2A - BD^2 - 4BD - 4B - 2D^2 + 3D - 2 &> 0 \end{aligned}$$

Comme GARLOFF *et* GRAF*, nous avons choisi les domaines de départ $A \in [100\,..\,120]$, $B \in [0\,..\,2]$, $D \in [10\,..\,20]$. La figure 12.4 compare l'approche de* GARLOFF *et* GRAF *avec la nôtre lorsque A est fixé à la valeur 110. D'après* ABDALLAH et al. *[2], le logiciel QEPCAD de* HONG*, basé sur la méthode CAD, a besoin de deux heures de calcul*[†] *pour prouver l'existence d'une solution au problème. Pour notre part, nous trouvons en 337 ms. un premier pavé intérieur et en moins de 3 mn. une approximation intérieure à 10^{-2} près, sur une SUN UltraSparc 1/167 MHz.*

12.3 Comparaison EIA4 *vs.* ICAb4

L'un des inconvénients de l'algorithme EIA4 est que des solutions trouvées successivement sont relativement proches. Or, il est important de pouvoir fournir à l'utilisateur un échantillon de solutions le plus représentatif possible dans un temps très court. La gestion de ce problème dans ICAb4 a été faite de la façon suivante : étant

[†] La machine utilisé n'est cependant pas préisé !



(a) Résolution par expansion de BERNSTEIN (b) Résolution avec WOpAC

FIG. 12.4: Deuxième problème de GARLOFF & GRAF [82]

donnée une contrainte de la forme $\forall t\colon (c_1 \wedge \cdots \wedge c_m)$ et un pavé $\boldsymbol{B} = I_1 \times \cdots \times I_n$, le processus de résolution possède deux degrés de liberté, à savoir : la sélection de la prochaine contrainte à considérer et la sélection de la variable suivante à découper. La figure 12.5 compare les différences en ce qui concerne l'ordre de génération des solutions pour le problème *Simple Circle*$_{2,2}$ pour deux stratégies sur le choix de la variable à découper : *depth-first* où chaque contrainte est considérée à son tour et où chaque domaine est découpé jusqu'à une limite fixée ; et *semi-depth-first* où chaque contrainte est considérée à son tour, mais où chaque variable n'est découpée qu'une seule fois et mise en queue dans une file de domaines.

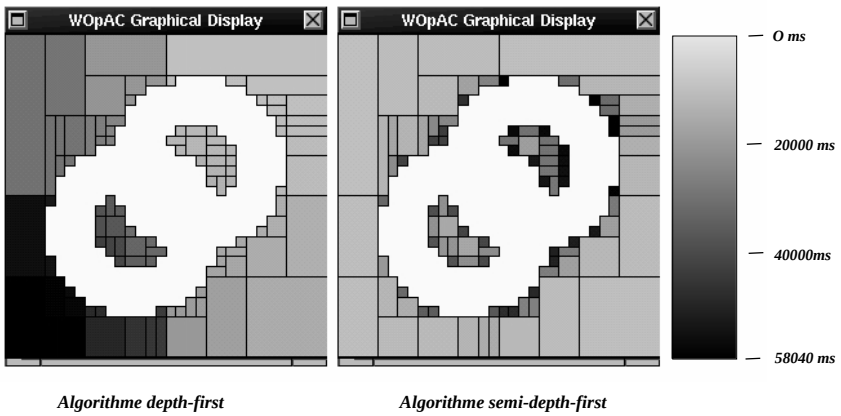


FIG. 12.5: *Depth-first vs. semi-depth first*

Comme on peut aisément le remarquer, l'algorithme *semi-depth-first* calcule des solutions consécutives réparties sur tout l'espace alors que l'algorithme *depth-first* calcule les solutions de haut en bas et de la droite vers la gauche.

L'algorithme EIA4 implémenté par JARDILLIER et LANGUÉNOU diffère de l'algorithme idéal que nous avons présenté en ce qu'ils ne cherchent pas à atteindre la canonicité des intervalles, mais se contentent de domaines de

taille inférieure à un ω donné. De même, lors du découpage des domaines, l'algorithme ICAb4 considère un seuil ε pour le processus de *splitting*. Les tables 12.1 et 12.2 présentent les temps de résolution (exprimés en secondes) des problèmes présentés plus haut sur une SUN UltraSparc 1/167 MHz sous Solaris 2.5.

TAB. 12.1: EIA4(ω) *vs.* ICAb4 — première solution

Benchmark	EIA4(ω)	ICAb4	EIA4(ω)/ICAb4
Projection$_{3,8}(\varepsilon = 10^{-1})$	0,20	0,17	1,18
Projection$_{3,8}(\varepsilon = 10^{-2})$	38,59	0,16	241,19
Projection$_{3,8}(\varepsilon = 10^{-3})$	>600	0,16	>3 750,00
Projection$_{3,4}(\varepsilon = 10^{-2})$	53,12	0,12	442,67
Projection$_{3,4}(\varepsilon = 10^{-3})$	>600	0,12	>5 000,00
School Problem$_{3,1}(\varepsilon = 10^{-2})$	0,02	0,09	0,22
School Problem$_{3,1}(\varepsilon = 10^{-3})$	1,58	0,09	17,56
Simple Circle$_{2,2}(\varepsilon = 10^{-2})$	0,99	0,05	19,80
Simple Circle$_{2,2}(\varepsilon = 10^{-3})$	20,86	0,05	417,20
GARLOFF/GRAF 1 $(\varepsilon = 10^{-2})$	—	0,04	—
GARLOFF/GRAF 2 $(\varepsilon = 10^{-2})$	—	0,57	—
Collision$_{4,1}$ $(\varepsilon = 10^{-1})$	—	0,00	—

On constate que l'algorithme ICAb4 permet de gagner plusieurs ordres de grandeur par rapport à l'algorithme EIA4 pour la résolution des jeux de tests considérés ; le gain augmente très fortement avec la précision demandée (de 400 fois plus rapide pour Projection$_{3,4}$ lorsque ε vaut 10^{-2}, on passe à un facteur supérieur à 5000 pour un ε de 10^{-3}). Contrairement à EIA4, le calcul d'un premier pavé solution avec ICAb4 reste de l'ordre du dixième de seconde pour tous les problèmes testés. Dans la conclusion de cette thèse, on verra qu'il est possible de tirer partie de ce fait pour utiliser ICAb4 pour obtenir rapidement un pavé inclus dans une relation pouvant servir de noyau de départ pour d'autres algorithmes de calcul d'approximation intérieure.

TAB. 12.2: EIA4(ω) *vs.* ICAb4 — toutes les solutions

Benchmark	EIA4	ICAb4	EIA4/ICAb4
Projection$_{3,5}$	783,03	68,83	11,38
Projection$_{3,10}$	>9 000	3 634	>2,40
Projection$_{5,5}$	>9 000	3 612	>2,40
School Problem$_{3,1}$	156,02	12,72	12,27
Flying Saucer$_{4,1}$	1 459,01	1 078,03	1,35
Simple Circle$_{2,1}$	12 789,03	651,59	19,63
Simple Circle$_{2,2}$	1 579,05	55,95	28,22
GARLOFF/GRAF 1	—	631,23	—
GARLOFF/GRAF 2	—	94,19	—
Collision$_{4,1}$	—	263,21	—

Même pour un ε de l'ordre de 10^{-2}, le temps de calcul de toutes les solutions apparaît très important. Heureusement, les applications visées peuvent souvent se contenter d'un ensemble représentatif restreint de pavés dans lequel l'utilisateur pourra choisir à loisir des valeurs pour ses variables.

Conclusion et perspectives

Conclusion et perspectives

NOUS ALLONS RAPPELER dans la suite les principaux faits énoncés dans cette thèse, pointer les contributions de notre travail ainsi que ses limites, puis indiquer des directions pour de futures recherches.

Arithmétique d'intervalles

L'arithmétique d'intervalles constitue la base indispensable pour la résolution sûre de contraintes non-linéaires réelles. Il existe actuellement des librairies de bonne qualité en Fortran [44] développées par les numériciens. Par contre, le langage C++ ne possède pas encore de librairie reconnue comme un standard et nombre des librairies C++ actuellement disponibles ont été développées de manière *ad hoc* pour des besoins ponctuels de leurs auteurs, ou bien portées directement du langage C (cas de Profil [134], par exemple).

Nous avons présenté dans cette partie une librairie utilisant les ressources propres au langage C++ : surcharge des opérateurs et patrons de classes. La librairie JAIL est ainsi plus souple et offre plus de possibilités d'extensions que celles que nous avons pu tester. En particulier, l'utilisateur peut choisir le type des bornes de ses intervalles. Nous avons montré expérimentalement que, contrairement à l'opinion parfois admise [149], cette plus grande souplesse ne pénalise pas JAIL en ce qui concerne les performances de calcul. Le recours à la technique des *traits* [173] permet d'implémenter les fonctions d'intervalles de façon plus sûre en obligeant explicitement les utilisateurs à fournir des versions correctement arrondies des opérateurs arithmétiques pour leur type de base.

Originellement développée pour être utilisée par OpAC, notre librairie de résolution de contraintes par propagation d'intervalles, JAIL ne possède pas encore toutes les fonctionnalités décrites dans la spécification BIAS de CHIRIAEV et WALSTER [44], qui se veut un standard pour les librairies d'intervalles. Un de nos objectifs est d'implémenter à terme la totalité de la *Basic Interval Arithmetic Specification* afin de pouvoir offrir JAIL à la communauté des utilisateurs de l'arithmétique des intervalles programmant en C++.

Lors de nos recherches sur les calculs d'approximations intérieures de relations réelles, nous avons rencontré des variations de l'arithmétique d'intervalles : les arithmétiques modales [78], de KAUCHER [129], et de MARKOV [156]. Présentant de meilleures propriétés que l'arithmétique d'intervalles, elles autorisent en particulier le calcul d'approximations intérieures des domaines de variations de fonctions réelles ainsi que des simplifications d'expressions par manipulations symboliques [186], ce qui peut se révéler intéressant dans un contexte de coopération de calculs symboliques et numériques. Actuellement utilisées marginalement dans quelques centres de recherches, nous pensons que les qualités de ces arithmétiques trouveront bientôt des applications importantes pour la résolution de problèmes industriels importants, par exemple en *control design* ou en robotique (gestion de contraintes quantifiées par calcul d'approximations intérieures). Il n'existe pas encore à notre connaissance de librairie disponible pour ces arithmétiques à l'exception d'un paquetage Mathematica [185]. Par contre, une extension de la spécification BIAS les prenant en compte a déjà été définie [184]. La disponibilité de librairies efficaces pour des langages de programmation tels que C++ étant une condition *sine qua non* à la mise en œuvre d'applications basées sur ces arithmétiques, nous projetons d'étendre JAIL afin d'offrir les fonctionnalités proposées par l'extension de BIAS de POPOVA [184].

Consistances partielles

Nous avons présenté la hull-consistance et la box-consistance, deux notions de consistances majeures pour la résolution de contraintes réelles. En intégrant les algorithmes HC3 et BC3 calculant ces deux consistances dans DecLIC, un langage de résolution de contraintes étendant clp(FD), nous avons montré que le choix de la consistance à utiliser est dépendant de la forme des contraintes. En étendant la définition de la box-consistance, nous avons pu capturer les différentes variantes de la définition originale [29] et proposer BC4, un nouvel algorithme

hybride reprenant les points forts des algorithmes HC3 et BC3 capable de gérer efficacement les contraintes d'intervalles quelles que soient leurs formes. Nous avons aussi prouvé que l'opérateur associé à la box$_\varphi$-consistance, un affaiblissement de la box-consistance destiné à accélérer les calculs est un opérateur de contraction, ce qui nous permet de l'intégrer dans les algorithmes de propagation de domaines sans perdre leurs propriétés.

L'algorithme BC4 considère actuellement chaque contrainte séparément au niveau de ses projections. Une extension susceptible d'améliorer son efficacité serait de considérer la conjonction des contraintes d'un système comme une seule contrainte. La vision globale du système ainsi obtenue devrait permettre de détecter plus précocement les inconsistances.

Débogage en programmation par contraintes

La programmation par contraintes induit l'existence de deux niveaux clairement séparés : le niveau du langage hôte et celui du *store* de contraintes. Le premier niveau est en général aisément accessible, que le langage soit impératif ou logique, en utilisant des débogueurs de type « traçage pas-à-pas ». Par contre, le deuxième niveau reste complètement caché aux yeux du programmeur. L'outil de débogage que nous avons élaboré et présenté offre une présentation structurée du *store* de contraintes et autorise le programmeur à considérer un ensemble de contraintes comme une nouvelle contrainte avec une sémantique clairement définie grâce à l'abstraction de S-box. Notre présentation a été faite dans le cadre de la programmation logique avec contraintes, mais le concept de S-box peut être utilisé pour tous les langages de programmation par contraintes qui peuvent se modéliser comme l'application d'opérateurs de contraction [24]. On peut cependant noter que l'utilisation des S-boxes en PLC permet de retrouver dans le *store* la structure induite par les clauses du programme, ce qui permettrait peut-être de réutiliser certaines techniques de débogage Prolog se basant sur la structure clausale.

Nous avons vu que l'intégration des *S-boxes* dans le processus de propagation peut se faire en affectant un numéro d'ordre (indice de DEWEY) à chaque contrainte en fonction de la *S-box* à laquelle elle appartient. Il est alors tentant de rapprocher cette technique de celles présentées par BORNING *et al.* [40] et WALLACE & FREUDER [242]. BORNING propose de séparer l'ensemble de contraintes en différentes classes suivant la priorité que l'on souhaite leur accorder. On peut par exemple décider d'avoir une classe de contraintes qui doivent absolument être satisfaites (priorité élevée) et une classe de contraintes que l'on aimerait voir satisfaites. Cette dichotomie se fait par l'affectation d'une étiquette à chaque contrainte. On pourrait utiliser la notion de *S-box* pour traduire les classes de priorités. Il faudrait alors modifier une nouvelle fois l'algorithme de propagation pour qu'il réinvoque en priorité les contraintes importantes et qu'il ne considère pas comme un échec le fait qu'une contrainte des classes non-prioritaires ne soit pas satisfaite. De la même façon, WALLACE et FREUDER montrent qu'un ordonnancement précis des contraintes dans la liste de propagation permet d'atteindre la quiescence plus vite. C'est par exemple le cas lorsque l'on réinvoque d'abord les contraintes des variables les plus connectées (possédant le plus d'occurrences dans des contraintes différentes). Une stratégie d'ordonnancement pourrait aussi se traduire par la notion de *S-box*. On peut imaginer un système utilisant les heuristiques décrites dans [242] pour affecter les contraintes d'un programme à différentes *S-boxes* de façon à optimiser la propagation.

Enfin, les fonctionnalités de traçage pas-à-pas de la propagation de domaines de notre débogueur pourraient être grandement étoffées si l'on était capable de faire de la rétraction efficace de contraintes, car l'on pourrait offrir des possibilités de « marche arrière » qui, actuellement obligeraient à stocker de nombreuses informations sur les domaines des variables. Les travaux de BERLANDIER [34] et GEORGET [84] sur la rétraction de contraintes pourraient éventuellement servir de point de départ.

Résolution de contraintes avec variables quantifiées

Après avoir montré que de nombreux problèmes requièrent la correction des résultats alors que les solveurs de contraintes actuellement disponibles n'assurent que la complétude, nous avons décrit des algorithmes de calcul d'approximations intérieures de relations réelles permettant d'obtenir la propriété de correction, et ce, pour des systèmes de contraintes numériques de n'importe quelle forme, contrairement à la plupart des autres méthodes existantes qui se restreignent au cas de polynômes. Basés sur les notions classiques de consistances locales, ils peuvent ainsi bénéficier des recherches actives menées dans ce domaine. Nous avons aussi proposé un algorithme

gérant des systèmes de contraintes possédant une occurrence de variable universellement quantifiée, ce qui correspond au type de contraintes que l'on rencontre dans de nombreuses applications en robotique (détection de collision, placement de caméra), la variable quantifiée correspondant alors au temps. Dans le cadre d'une application de placement de caméra dans des scènes animées, les algorithmes proposés se sont révélés plus efficace d'au moins un ordre de grandeur que ceux utilisés précédemment.

Cependant, le temps de calcul augmente très vite avec le nombre de variables des problèmes et la résolution de systèmes comportant plus de quelques variables est encore hors de portée (ce qui est aussi le cas pour les autres techniques présentées dans l'état de l'art). Une amélioration possible semble être le recours à des extensions aux intervalles plus précises que celle actuellement employée (extension naturelle). Nous projetons en particulier de tester l'impact du recours à l'extension de BERNSTEIN ; cette extension est déjà utilisée par GARLOFF et GRAF pour calculer des approximations intérieures par évaluation/découpage. L'utilisation conjointe de la propagation de domaines et des notions de consistances locales devrait vraisemblablement accélérer les calculs.

Une piste prometteuse pour le calcul d'approximations intérieures, et partant, pour la gestion de contraintes avec variables quantifiées, semble être celle de l'arithmétique modale, où il est possible de spécifier le sens ($\forall, \exists$) associé à chaque domaine de variable. Cette arithmétique ainsi que celle de KAUCHER très proche, ont déjà été utilisées avec succès dans un cadre très restreint pour des contraintes polynômiales ainsi que pour des systèmes linéaires [13, 211]. Nous pensons qu'une utilisation plus large de ces arithmétiques est possible pour résoudre des systèmes de contraintes non-linéaires réelles quelconques en offrant la propriété de correction du résultat, à condition de les étendre convenablement pour n'être pas limité aux seules fonctions rationnelles satisfaisant certaines conditions de monotonie. On pourrait alors s'attaquer à des systèmes de contraintes contenant des variables quantifiées universellement ou existentiellement. L'étape suivante serait alors de définir de nouveaux algorithmes permettant de prendre en compte tant des conjonctions de contraintes que des disjonctions, afin de pouvoir résoudre les systèmes de forme la plus générale possible.

COLLAVIZZA *et al.* [52] ont mis récemment au point une méthode de calcul d'approximation intérieure d'une relation réelle : partant d'un « noyau » inclus dans la relation, ils expansent les domaines des variables jusqu'à obtenir un sous-ensemble « maximal » de l'approximation intérieure (où la notion de maximalité doit s'entendre au sens de SHARY [210, 209]). Un inconvénient de leur méthode est qu'elle n'offre aucune aide pour calculer le noyau de départ nécessaire. Il semblerait donc intéressant d'essayer de la combiner avec celle que nous avons décrite au chapitre 6, où l'on calculerait les noyaux pour chaque sous-ensemble connexe de la relation avec notre méthode, laissant à leur algorithme la tâche de les expanser pour obtenir une approximation intérieure maximale.

Bibliographie

[1] Chaouki ABDALLAH, Peter DORATO, Wei YANG, Richard LISKA et Stanly STEINBERG. Applications of quantifier elimination theory to control theory. In *Proceedings of the 4th IEEE Mediterranean Symposium on Control and Automation*, pages 340–345, Malene, Crete, Greece, 1996.

[2] Chaouki T. ABDALLAH, Peter DORATO, Richard LISKA, Stanly STEINBERG et Wei YANG. Applications of quantifier elimination theory to control theory. Technical report EECE95-007, Department of Electrical and Computer Engineering, School of Engineering, University of New Mexico, septembre 1995.

[3] Steven ABRAMS, Peter K. ALLEN et Konstantinos TARABANIS. Computing camera viewpoints in an active robot work-cell. Manuscript,

[4] Abderrahmane AGGOUN et Nicolas BELDICEANU. Time stamps techniques for the trailed data in CLP systems. In *Actes du Séminaire 1990 – Programmation en Logique*, Trégastel, France, 1990. CNET.

[5] Abderrahmane AGGOUN et Nicolas BELDICEANU. Overview of the CHIP compiler system. In Frédéric BENHAMOU et Alain Marie Albert COLMERAUER, réds., *Constraint Logic Programming: Selected Research*, pages 421–435. The MIT Press, 1993.

[6] Abderrahmane AGGOUN, D. CHAN, P. DUFRESNE, E. FALVEY, H. GRANT, A. HEROLD, G. MACARTNEY, Micha MEIER, D. MILLER, Shyam MUDAMBI, B. PEREZ, E. VAN ROSSUM, J. SCHIMPF, P. A. TSAHAGEAS et D. H. de VILLENEUVE. *ECL^{i}PS^e 3.5*. European Computer Industry Research Centre (ECRC), Munich, *Deutchland*.

[7] Alfred Vaino AHO, Ravi SETHI et Jeffrey David ULLMAN. *Compilateurs : Principes, techniques et outils*. InterÉditions, 1989.

[8] Hassan AÏT-KACI. *The WAM : A (Real) Tutorial*. DEC, Paris Research Laboratory, janvier 1990.

[9] Hassan AÏT-KACI. *Warren's Abstract Machine, a tutorial Reconstruction*. The MIT Press, 1991.

[10] Götz ALEFELD et Jürgen HERZBERGER. *Introduction to Interval Computations*. Academic Press Inc., New York, USA, 1983. Traduit par Jon ROKNE de l'original Allemand *'Einführung In Die Intervallrechnung'*,

[11] Brian D. O. ANDERSON, N. K. BOSE et E. I. JURY. Output feedback stabilization and related problems—solution via decision methods. *IEEE Transactions on Automatic Control*, AC-20(1), 1975.

[12] Applied Logic Systems, Inc. *CLP(BNR) user guide and reference*, 1996. Disponible à l'adresse `http://www.als.com`.

[13] Joaquim ARMENGOL, Louise TRAVÉ-MASSUYÈS, Josep Lluís de la ROSA et Josep VEHÍ. Envelope generation for interval systems. In Miguel TORO, réd., *Actas del seminario sobre técnicas cualitativas. VII Congreso de la Asociación Española para la Inteligencia Artificial (CAEPIA 97)*, pages 33–48, Málaga, 1997.

[14] Joaquim ARMENGOL, Louise TRAVÉ-MASSUYÈS, Josep VEHÍ et Miguel Ángel SÁINZ. Envelope generation using modal interval analysis. Research report R/1998/0005, Departament d'Electrònica, Informàtica i Automàtica (EIA), University of Girona, 1998.

[15] Joaquim ARMENGOL, Louise TRAVÉ-MASSUYÈS, Josep VEHÍ et Miguel Ángel SÁINZ. Modal interval analysis for error-bounded semiqualitative simulation. In *1r Congrés Català d'Intelligència Artificial*, pages 223–231, 1998.

[16] Joaquim ARMENGOL, Josep VEHÍ, Josep Lluís de la ROSA et Louise TRAVÉ-MASSUYÈS. On modal interval analysis for envelope determination within the ca˜en qualitative simulator. In *Proceedings of the 7th Information Processing and Management of Uncertainty in Knowledge-Based Systems Conference (IPMU 98)*, pages 110–117, La Sorbonne, Paris, France, 1998.

[17] Projet ARÉNAIRE. Some worst cases for the table maker's dilemma. Page WEB accessible à `http://www.inria.fr/ Equipes/ ARENAIRE-fra.html`.

[18] Jean-Claude BAJARD, Dominique MICHELUCCI, Jean-Michel MOREAU et Jean-Michel MULLER. Introduction to the special issue on "real numbers and computers". *Journal of Universal Computer Science*, 1(7):436–438, 1995.

[19] Eckart BAUMANN. Optimal centered forms. *BIT*, 28(1):80–87, 1988.

[20] Gerald BAUMGARTNER et Vincent F. RUSSO. Signatures: a language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice & Experience*, 25(8):863–889, août 1995.

[21] Frédéric BENHAMOU. Interval constraint logic programming. In Andreas PODELSKI, réd., *Constraint programming: basics and trends: 1994 Châtillon Spring School, Châtillon-sur-Seine, France, May 16–20, 1994*, volume 910 of *Lecture notes in computer science*, pages 1–21. Springer-Verlag, 1995.

[22] Frédéric BENHAMOU, Pascal BOUVIER, Alain COLMERAUER, H. GARETTA, B. GILETTA, Jean-Louis MASSAT, G. A. NARBONI, Stéphane N'DONG, Robert PASERO, Jean-François PIQUE, TOURAÏVANE, Michel VAN CANEGHEM et Éric VÉTILLARD. *Le manuel de Prolog IV*. PrologIA, Marseille, juin 1996.

[23] Frédéric BENHAMOU, Frédéric GOUALARD, Laurent GRANVILLIERS et Jean-François PUGET. Revising hull and box consistency. In *Proceedings of the sixteenth International Conference on Logic Programming (ICLP'99)*, pages 230–244, Las Cruces, USA, 1999. The MIT Press. ISBN 0-262-54104-1,

[24] Frédéric BENHAMOU. Heterogeneous constraint solving. In *Proceedings of the fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139*, pages 62–76, Aachen, Germany, 1996. Springer-Verlag.

[25] Frédéric BENHAMOU, Frédéric GOUALARD et Laurent GRANVILLIERS. Optimiser la propagation d'intervalles par des méthodes de recherche locale. Rapport technique, IRISA-Rennes, novembre 1997. Journées du PRC-GDR "Programmation",

[26] Frédéric BENHAMOU, Frédéric GOUALARD et Laurent GRANVILLIERS. A hybrid consistency for real constraint solving. In *Proceedings of INTERVAL'98*, Nanjing, China, 1998.

[27] Frédéric BENHAMOU et Laurent GRANVILLIERS. Combining local consistency, symbolic rewriting and interval methods. In *Proceedings of AISMC-3*, volume 1138 of *LNCS*, pages 144–159, Steyr, Austria, 1996. Springer-Verlag.

[28] Frédéric BENHAMOU et Laurent GRANVILLIERS. Automatic generation of numerical redundancies for non-linear constraint solving. *Reliable Computing*, 3(3):335–344, 1997.

[29] Frédéric BENHAMOU, David MCALLESTER et Pascal VAN HENTENRYCK. CLP(Intervals) revisited. In *Proceedings of the International Symposium on Logic Programming (ILPS'94)*, pages 124–138, Ithaca, NY, novembre 1994. The MIT Press.

[30] Frédéric BENHAMOU et William J. OLDER. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.

[31] Frédéric BENHAMOU, William J. OLDER et André VELLINO. Constraint logic programming on boolean, integer and real intervals. *Journal of Symbolic Computation*, 1995. Submitted,

[32] Frédéric BENHAMOU et TOURAÏVANE. Prolog IV: langage et algorithmes. In *JFPL'95: IVèmes Journées Francophones de Programmation en Logique*, pages 51–65, Dijon, France, 1995. Teknea.

[33] Jakob BERCHTOLD, Irina VOICULESCU et Adrian BOWYER. Multivariate Bernstein-form polynomials. Technical report 31/98, School of Mechanical Engineering, University of Bath, 1998.

[34] Pierre BERLANDIER. Deux variations sur le thème de la consistance d'arcs : maintien et renforcement. Rapport de recherche 2426, INRIA Sophia-Antipolis, décembre 1994.

[35] Christian BESSIÈRE et Marie-Odile CORDIER. Arc-consistency and arc-consistency again. In *Proceedings of AAAI Conference*, pages 108–113, Washington, USA, 1993.

[36] Christian BLIEK. *Computer Methods for Design Automation*. Thèse de doctorat, Department of Ocean Engineering, Massachusetts Institute of Technology, MA et Katholieke Universiteit Leuven, Belgique, août 1992.

[37] Jim BLINN. Where am I? what am I looking at? *IEEE Computer Graphics and Applications*, 8(4):76–81, juillet 1988.

[38] William BLUME et Rudolf EIGENMANN. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, avril 1995.

[39] Daniel G. BOBROW et Bertram RAPHAEL. New programming languages for artificial intelligence. *ACMCS*, 6(3):155–174, septembre 1974.

[40] Alan BORNING, Bjørn FREEMAN-BENSON et Molly WILSON. Constraint hierarchies. *LISP and symbolic computation*, 5:223–270, 1992. Kluwer Academic Publishers,

[41] J. C. BURKILL. Functions of intervals. *Proceedings of the London Mathematical Society*, 22:375–446, 1924.

[42] Arthur Walter BURKS, Herman Heine GOLDSTINE et John Von NEUMANN. Preliminary discussion of the logic design of an electronic computing instrument. Rapport technique, Institute for advanced study, princeton, 1946. reprinted in C. G. Bell, Computer structures, readings and examples, Mc Graw-Hill, New York, 1971,

[43] Jean-Marie CHESNEAUX, Stéphane GUILAIN et Jean VIGNES. *La bibliothèque CADNA : présentation et utilisation*, novembre 1996.

[44] Dmitri CHIRIAEV et G. William WALSTER. Interval arithmetic specifications. Manuscript J3/97-199 for ANSI X3J3, juillet 1997. Disponible à l'adresse `http://www.mscs.mu.edu/ globsol/Papers/spec.ps`.

[45] Chong-Kan CHIU. Interval linear constraint solving in constraint logic programming. Technical Report CS-TR-94-13, Chinese University of Hong Kong, décembre 1994. Disponible à l'adresse `ftp://ftp.cs.cuhk.hk/pub/techreports/./94/tr-94-13.ps.gz`.

[46] Marc CHRISTIE. Spécification de mouvements de caméra et variables universellement quantifiées. Rapport de DEA, Institut de Recherche en Informatique de Nantes, septembre 1998.

[47] John G. CLEARY. Proving the existence of solutions in logical arithmetic. Manuscript,

[48] John G. CLEARY. Logical arithmetic. *Future Generation Computing Systems*, 2(2):125–149, 1987.

[49] Philippe CODOGNET et Daniel DIAZ. `wamcc:` compiling Prolog to C. In *International Conference on Logic Programming*, pages 317–331, 1995.

[50] Philippe CODOGNET et Daniel DIAZ. Compiling constraints in `clp(fd)`. *Journal of Logic Programming*, 27(3):185–226, 1996.

[51] Hélène COLLAVIZZA, François DELOBEL et Michel RUEHER. Comparing partial consistencies. *Reliable Computing*, 5:1–16, 1999.

[52] Hélène COLLAVIZZA, François DELOBEL et Michel RUEHER. Extending consistent domains of numeric CSP. In *Proceedings of the 16th IJCAI*, volume 1, pages 406–411, Stockholm, Sweden, juillet 1999.

[53] George E. COLLINS. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proceedings of the Second GI Conf. Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183, Kaiserslauten, 1975. Springer.

[54] George E. COLLINS et Hoon HONG. Partial cylindrical decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12:299–328, 1991.

[55] Alain Marie Albert COLMERAUER, Henri KANOUI, Robert PASERO et Philippe ROUSSEL. Un système de communication homme-machine en Français. Rapport, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, 1973.

[56] Intel CORPORATION. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, technical report n° 243190 Chapter 7: Floating-Point Unit. Intel Corporation, 1997.

[57] Jorge CRUZ, Pedro BARAHONA et Frédéric BENHAMOU. Integrating deep biomedical models into medical decision support systems: an interval constraint approach. In *Proceedings of the Seventh biennial meetings*

of the AIME and ESMDM societies (AIMDM'99), Joint European Conference on Artificial Intelligence in Medicine and Medical Decision Making, Lecture Notes in Artificial Intelligence, Aalborg, Denmark, juin 1999. Springer Verlag.

[58] Joseph D. DARCY. *Borneo 1.0.2: Adding IEEE 754 floating point support to Java*. University of California, Berkeley, mai 1998.

[59] John DARLINGTON et Yi-Ke GUO. A new perspective on integrating functions and logic languages. In *3rd International Conference on Fifth Generation Computer Systems*, pages 682–693, Tokyo, Japan, 1992.

[60] Glorianna DAVENPORT, Thomas Aguierre SMITH et Natalio PINCEVER. Cinematic primitives for multimedia. *IEEE Computer Graphics and Applications*, 11(4):67–74, juillet 1991.

[61] E. DAVIS. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[62] Kœn de BOSSCHERE. An operator precedence parser for standard Prolog text. ELIS Technical Report PARIS 94-09, Vakgrœp Elektronica en Informatiesystemen, Universiteit Gent, septembre 1994.

[63] Pierre DERANSART, AbdelAli ED-DBALI et Laurent CERVONI. *Prolog: The standard*. Springer-Verlag, 1996.

[64] Pierre DERANSART, Manuel HERMENEGILDO et Jan MAŁUSZINSKI, réds. *Analysis and Visualization Tools for Constraint Programming*. Springer Verlag, 1999. À paraître,

[65] Daniel DIAZ. `clp(FD) 2.21` *User's Manual*. INRIA-Rocquencourt, Le Chesnay, France, juillet 1994.

[66] Daniel DIAZ. `wamcc 2.21` *User's Manual*. INRIA-Rocquencourt, Le Chesnay, France, juillet 1994.

[67] Daniel DIAZ. *Étude de la compilation des langages logiques de programmation par contraintes sur les domaines finis : le système* `CLP(FD)` . Thèse de doctorat, Université d'Orléans, 1995.

[68] Daniel DIAZ. *GNU Prolog: a Native Prolog Compiler with Constraint Solving over Finite Domains*. GNU, 1.1 édition, juin 1999.

[69] M. DINCBAS, Pascal VAN HENTENRYCK, Helmut SIMONIS, Abderrahmane AGGOUN, T. GRAF et F. BERTHIER. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 683–702, Tokyo, Japan, 1988.

[70] Steven M. DRUCKER et David ZELTZER. Intelligent camera control in a virtual environment. In *Proceedings of Graphics Interface '94*, pages 190–199, Banff, Alberta, Canada, mai 1994. Canadian Information Processing Society.

[71] Steven Mark DRUCKER. *Intelligent camera control for graphical environments*. Thèse de doctorat, Massachusetts Institute of Technology, Program in Media Arts and Sciences, School of Architecture and Planning, juin 1994.

[72] Mireille DUCASSÉ. Opium – an advanced debugging system. In G. COMYN et N. FUCHS, réds., *Proceedings of the Second Logic Programming Summer School*, number 636 in Lecture Notes in Artificial Intelligence. Springer-Verlag, septembre 1992.

[73] Abbas EDALAT. Domains for computation in mathematics, physics and exact real arithmetic. Manuscript to appear in *Bulletin of Symbolic Logic*,

[74] Rida T. FAROUKI et V. T. RAJAN. On the numerical condition of polynomials in Bernstein form. *Computer Aided Geometric Design*, 4:191–216, 1987.

[75] Rida T. FAROUKI et V. T. RAJAN. Algorithms for polynomials in Bernstein form. *Computer Aided Geometric Design*, 5:1–26, 1988.

[76] Richard E. FIKES. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[77] Christopher W. FRASER, David R. HANSON et Todd A. PROEBSTING. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, septembre 1992. Disponible à l'adresse `http://www.cs.Princeton.EDU:80/software/iburg/iburg.ps`.

[78] E. H. GARDEÑES et H. MIELGO. Modal intervals: reasons and ground semantics. In *Interval Mathematics*, number 212 in Lecture Notes in Computer Science. Springer-Verlag, 1986.

[79] Jürgen GARLOFF. Convergent bounds for the range of multivariate polynomials. In K. NICKEL, réd., *Interval Mathematics*, pages 37–56. Springer Verlag, 1985.

[80] Jürgen GARLOFF. Applications of Bernstein expansion to the solution of control problems. *Reliable Computing*, 6(3):303–320, 2000. Special Issue on Applications to Control, Signals and Systems. Jürgen Garloff and Éric Walter, eds.,

[81] Jürgen GARLOFF et Birgit GRAF. Robust schur stability of polynomial parameter dependency. *Multidimensional Systems and Signal Processing*, 10:189–199, 1999.

[82] Jürgen GARLOFF et Birgit GRAF. Solving strict polynomial inequalities by Bernstein expansion. In N. MUNRO, réd., *The Use of Symbolic Methods in Control System Analysis and Design*, pages 339–352. The Institution of Electrical Engineers, London, England, 1999.

[83] Jürgen GARLOFF, Birgit GRAF et Michael ZETTLER. Speeding up an algorithm for checking robust stability of polynomials. In *Proceedings of the 2nd IFAC Symposium on Robust Control Design*, pages 183–188, Oxford, 1998. Elsevier Science.

[84] Yan GEORGET. *Extensions réactives de la Programmation par Contraintes*. Phd thesis, École Polytechnique, octobre 1999.

[85] Michael GLEICHER et Andrew WITKIN. Through-the-lens camera control. In *Proceedings of SIGGRAPH'92*, volume 26 (2) of *Computer Graphics*, pages 331–340, juillet 1992.

[86] David Marc GOLDBERG. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, mars 1991.

[87] Laureano GONZALEZ-VEGA. Some examples of problem solving by using the symbolic viewpoint when dealing with polynomial systems of equations. In F. W. Hehl J. FLEISCHER, J. GRABMEIER et W. KÜCHLIN, réds., *Computer Algebra in Science and Engineering*, pages 102–116. World Scientific Publishing, 1995.

[88] Laureano GONZALEZ-VEGA, T. RECIO, H. LOMBARDI et M.-F. ROY. Sturm-habicht sequences, determinants and real roots of univariate polynomials. In *special volume of the series Texts and Monographs in Symbolic Computation entitled 25 years of Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer-Verlag, 1996.

[89] Laureano GONZALEZ-VEGA et Guadalupe TRUJILLO. Symbolic recipes for polynomial system solving: Real solutions. Lecture notes of the tutorial presented at the International Symposium on Symbolic and Algebraic Computation ISSAC–96 held in Zurich (Switzerland) (1996). Universidad de Cantabria, Spain,

[90] James GOSLING, Bill JOY et Guy STEELE. *The Java Language Specification*. Addison-Wesley, first édition, août 1996.

[91] Frédéric GOUALARD. IConS, compilation d'un langage pour la résolution de contraintes par propagation d'intervalles et coopération de résolveurs dédiés. Mémoire de DEA informatique, Laboratoire d'Informatique d'Orléans, septembre 1995.

[92] Frédéric GOUALARD. *DecLIC: a Declarative Language with Interval Constraints V. 1.0a, User's Guide and Reference Manual*. LIFO, Orléans, 1999.

[93] Frédéric GOUALARD et Frédéric BENHAMOU. A visualization tool for constraint program debugging. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pages 110–117, Cocoa Beach, Florida, 1999. IEEE Computer Society.

[94] Frédéric GOUALARD, Frédéric BENHAMOU et Laurent GRANVILLIERS. An extension of the WAM for hybrid interval solvers. *The Journal of Functional and Logic Programming. The MIT Press*, 1999(4), avril 1999. Special issue on Parallelism and Implementation Technology for Constraint Logic Programming, Vítor Santos Costa, Enrico Pontelli and Gopal Gupta (eds),

[95] Laurent GRANVILLIERS. *Consistances locales et transformations symboliques de contraintes d'intervalles*. Thèse de doctorat, Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans, décembre 1998.

[96] Laurent GRANVILLIERS, Frédéric GOUALARD et Frédéric BENHAMOU. Box consistency through weak box consistency. In *Proceedings of the eleventh IEEE International Conference on Tools with Artificial Intelligence*, pages 373–380, Chicago, IL, USA, 1999. IEEE Computer Society. ISBN 0-7695-0456-6,

[97] Andreas GRIEWANK. On automatic differentiation. Preprint ANL/MCS-P10-1088, novembre 1988,

[98] SIGLA/X GROUP. Modal interval arithmetic: an introduction. Available at the SIGLA/X group homepage,

[99] Eldon Robert HANSEN. On the centered form. In Eldon Robert HANSEN, réd., *Topics in Interval Analysis*, pages 102–106. Clarendon Press, Oxford, 1969.

[100] Eldon Robert HANSEN. *Global Optimization Using Interval Analysis*. Pure and Applied Mathematics. Marcel Dekker Inc., 1992.

[101] N. HEINTZE, Joxan JAFFAR, S. MICHAYLOV, P. STUCKEY et Roland YAP. *The CLP($\Re$) Programmer's Manual Version 1.1*, novembre 1991.

[102] Leon HENKIN, J. Donald MONK et Alfred TARSKI. *Cylindric Algebras (Part I)*. North-Holland, 1971.

[103] Timothy HICKEY et Qun JU. Efficient implementation of interval arithmetic narrowing using IEEE arithmetic. Submitted to ILPS'97,

[104] Timothy J. HICKEY. Analytic constraint solving and interval arithmetic. In *Proceedings of POPL'00*, 2000.

[105] Timothy J. HICKEY. Clip: a CLP(Intervals) dialect for metalevel constraint solving. In *Procs. of PADL'00*, volume 1753 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[106] Timothy J. HICKEY, Qun JU et Maarten Herman VAN EMDEN. Interval arithmetic: from principles to implementation. Submitted to JACM,

[107] Timothy J. HICKEY, Maarten Herman VAN EMDEN et H. WU. A unified framework for interval constraints and interval arithmetic. In Michael MAHER et Jean-Francois PUGET, réds., *Principles and Practice of Constraint Programming—CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 250–264. Springer-Verlag, avril 1998.

[108] Nicholas John HIGHAM. The accuracy of floating-point summation. *SIAM J. Sci. Stat. Comput.*, juillet 1992. À paraître,

[109] W. HOFSCHUSTER et W. KRAEMER. A fast public domain interval library in ANSI C. In Achim SYDOW, réd., *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 2 of *Numerical Mathematics*, pages 395–400, Berlin, 1997.

[110] Hoon HONG. An improvement of the projection operator in cylindrical algebraic decomposition. In Shunro WATANABE et Morio NAGATA, réds., *ISSAC '90: proceedings of the International Symposium on Symbolic and Algebraic Computation: August 20–24, 1990, Tokyo, Japan*, pages 261–264, New York, NY 10036, USA and Reading, MA, USA, 1990. ACM Press and Addison-Wesley. Disponible à l'adresse `http://www.acm.org:80/pubs/citations/proceedings/issac/96877/p261-hong/`.

[111] Hoon HONG. Collision problems by an improved cad-based quantifier elimination algorithm. Technical Report 91-05, RISC-Linz, Johannes Kepler University, Linz, Austria, 1991. Disponible à l'adresse `http://info.risc.uni-linz.ac.at/archive/reports/1991/91-05.html`.

[112] Hoon HONG. Bounds for absolute positiveness of multivariate polynomials. Rapport technique, Research Institute for Symbolic Computation, mars 1997.

[113] Hoon HONG et Dalibor JAKUŠ. Testing positiveness of polynomials. Technical Report 96-02, RISC-Linz, Johannes Kepler University, Linz, Austria, 1996. Disponible à l'adresse `http://info.risc.uni-linz.ac.at/archive/reports/1996/96-02.html`.

[114] Hoon HONG et Volker STAHL. Bernstein form is inclusion monotone. *Computing*, 55:43–53, 1995.

[115] Eero HYVÖNEN. Constraint reasoning based on interval arithmetic. In *Proceedings of IJCAI'89*, 1989.

[116] IEEE. IEEE standard for binary floating-point arithmetic. Rapport technique IEEE Std 754-1985, Institute of Electrical and Electronics Engineers, 1985. Reaffirmed 1990,

[117] Joxan JAFFAR et Jean-Louis LASSEZ. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Programming Languages*, pages 111–119, Munich, janvier 1987.

[118] Joxan JAFFAR et Michael J. MAHER. Constraint logic programming: a survey. *Journal of Logic Programming*, 19(20), 1994.

[119] G. JANSSENS, Bart DEMŒN et Y. WILLEMS. Execution mechanisms for Prolog. Rapport technique CW 53, Department of Computer Science, K. U. Leuven, avril 1987.

[120] Frank JARDILLIER et Éric LANGUÉNOU. Screen-space constraints for camera movements: the virtual cameraman. In N. FERREIRA et M. GÖBEL, réds., *Eurographics'98 proceedings*, volume 17, pages 175–186. Blackwell Publishers, 1998.

[121] Mats JIRSTRAND. Cylindrical algebraic decomposition—an introduction. Technical Report 1995-10-18, Department of Electrical Engineering, Linköping University, Sweden, 1995.

[122] William Morton KAHAN. The improbability of probabilistic error analyses for numerical computations. Lecture notes prepared for the UC Berkeley Statistics Colloquium (February 1996), subsequently revised (March 1996). Earlier version presented at the third ICIAM Congress (July 1995). Disponible à l'adresse `http://http.cs.berkeley.edu/~wkahan/improber.ps`.

[123] William Morton KAHAN. Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. Manuscript, mai 1996, Disponible à l'adresse `http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps`.

[124] William Morton KAHAN. What can you learn about floating-point arithmetic in one hour? Slides for CS 267, Berkeley, University of Californy, février 1996.

[125] William Morton KAHAN. Miscalculating area and angles of a needle-like triangle. septembre 1997, Disponible à l'adresse `http://http.cs.berkeley.edu/~wkahan/Triangle.ps`.

[126] William Morton KAHAN et Melody Y. IVORY. Roundoff degrades an idealized cantilever. juillet 1997, Disponible à l'adresse `http://http.cs.berkeley.edu/~wkahan/Cantilever.ps`.

[127] William Morton KAHAN et Charles SEVERANCE. An interview with the old man of floating-point. *IEEE Computer*, mars 1998.

[128] Chong kan CHIU et Jimmy Ho-Man LEE. Efficient interval linear equality solving in constraint logic programming. Manuscript,

[129] Edgar W. KAUCHER. Interval analysis in the extended interval space IR. *Computing. Supplementum*, 2:33–49, 1980.

[130] Ralph Baker KEARFOTT. Interval computations: Introduction, uses, and resources. Disponible à l'adresse `http://interval.usl.edu/euromath.html`.

[131] Ralph Baker KEARFOTT. On a general technique for finding directions proceedings from bifurcation points. In T. KÜPPER, H. D. MITTELMANN, et H. WEBER, réds., *Numerical Methods for Bifurcation Problems*, International Series of Numerical Mathematics, pages 210–218. Birkhäuser, Boston, 1984.

[132] Ralph Baker KEARFOTT et Vladik KREINOVICH, réds. *Applications of Interval computations*, Applied Optimization. Kluwer Academic Publishers, 1996.

[133] Walid T. KEIROUZ, Glenn A. KRAMER et Jahir PABON. Exploiting constraint dependency information for debugging and explanation. In *In proceedings of "Principles and practice of constraint programming; The Newport Papers."*, Cambridge, Mass., avril 1993. The MIT Press.

[134] Olaf KNÜPPEL. PROFIL/BIAS—a fast interval library. *Computing*, 53:277–287, 1996.

[135] Donald Ervin KNUTH. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, troisième édition, 1997.

[136] Donald Ervin KNUTH. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, troisième édition, 1997.

[137] Andrei Nikolaevich KOLMOGOROV et Sergey Vladimirovich FOMIN. *Introductory Real Analysis*. Dover Publications, Inc., New York, 1975. Translated and edited from Russian by Richard A. Silverman,

[138] L. KUPRIYANOVA. Inner estimation of the united solution set of interval linear algebraic system. *Reliable Computing*, 1:15–31, 1995.

[139] L. KUPRIYANOVA. Inner estimation of the united solution set of interval linear algebraic system. In *Proceedings of APIC'95*, El Paso, 1995. Extended abstract,

[140] L. KUPRIYANOVA. A method of square root for solving interval linear algebraic system. *Reliable Computing*, 1(1):15–31, 1995.

[141] Teimuraz KUTSIA et Josef SCHICHO. Numerical solving of constraints of multivariate polynomial strict inequalities. Manuscript, octobre 1999,

[142] Yahia LEBBAH. *Contribution à la résolution de contraintes par consistance forte*. Thèse de doctorat, École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes, 1999.

[143] Yahia LEBBAH et Olivier LHOMME. Acceleration methods for numeric CSPs. In *AAAI-98*, pages 19–25. The MIT Press, juillet 1998.

[144] Yahia LEBBAH et Olivier LHOMME. Prediction by extrapolation for interval tightening methods. In *Procs. of SCAN'98*, pages 159–166. Kluwer, septembre 1999.

[145] Anthony P. LECLERC. *Efficient and Reliable Global Optimization*. Thèse de doctorat, Graduate School of the Ohio State University, 1992.

[146] Jimmy Ho-Man LEE et Tak-Wai LEE. A WAM-based abstract machine for interval constraint logic programming. In *Sixth IEEE International Conference on Tools with Artificial Intelligence*, pages 122–128, New-Orleans, USA, 1994.

[147] Jimmy Ho-Man LEE et Maarten Herman VAN EMDEN. Adapting CLP($\mathcal{R}$) to floating-point arithmetic. In *Procs. of the International Conference on Fifth Generation Computer Systems*, volume 16, pages 996–1003, Tokyo, Japan, 1992.

[148] Vincent LEFÈVRE, Jean-Michel MULLER et Arnaud TISSERAND. The table maker's dilemma. *IEEE Transactions on Computers*, 47(11), novembre 1998.

[149] Michael LERCH et Jürgen Wolff von GUDENBERG. `fi_lib++`: Specification, implementation and test of a library for extended interval arithmetic. Manuscript, mars 2000,

[150] Olivier LHOMME. Consistency techniques for numeric CSPs. In R. BAJCSY, réd., *Proceedings of the 13th IJCAI*, pages 232–238, Chambéry, France, 1993. IEEE Computer Society Press.

[151] Olivier LHOMME. *Contribution à la résolution de contraintes sur les réels par propagation d'intervalles*. Thèse de doctorat, Université de Nice-Sophia Antipolis, juillet 1994.

[152] Alan K. MACKWORTH. Consistency in networks of relations. *Artificial Intelligence*, 1(8):99–118, 1977.

[153] Alan K. MACKWORTH et Eugene C. FREUDER. The complexity of constraint satisfaction revisited. Manuscript,

[154] Alan K. MACKWORTH et Eugene C. FREUDER. The complexity of some polynomial network consistency algorithms for constraint satisfaction problem. *Artificial Intelligence*, 25(1):65–74, 1985.

[155] Kyoko MAKINO et Martin BERZ. Efficient control of the dependency problem based on Taylor model methods. *Reliable Computing*, 5:3–12, 1999.

[156] Svetoslav M. MARKOV. On directed interval arithmetic and its applications. *Journal of Universal Computer Science*, 1(7):514–526, 1995.

[157] Svetoslav M. MARKOV, Evgenija D. POPOVA et Christian P. ULLRICH. On the solution of linear algebraic equations involving interval coefficients. In P. Vassilevski (Eds.) S. MARGENOV, réd., *Iterative Methods in Linear Algebra, II*, volume 3 of *IMACS Series in Computational and Applied Mathematics*, pages 216–225. IMACS, 1996.

[158] Svetoslav M. MARKOV et Christian P. ULLRICH. On algebraic relations and symbolic manipulations in interval arithmetic with application to linear systems. Technical report 96-1, University of Basel, Department of computer Science, janvier 1996.

[159] David MCALLESTER. Constraint satisfaction search. Lecture Notes for 6.824, Artificial Intelligence, octobre 1992,

[160] Scott MCCALLUM. Partial solution to a path finding problem using the CAD method. Manuscript, juin 1995,

[161] Michael J. MCLENNAN. *Object-Oriented Programming with [Incr Tcl]*. Lucent Technologies, 1996.

[162] Micha MEIER. Debugging constraint programs. In Ugo MONTANARI et Francesca ROSSI, réds., *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, volume 976, pages 204–221. Springer Verlag, septembre 1995.

[163] Micha MEIER. *Grace 1.0 User Manual*. ECRC, juillet 1996.

[164] Keith MEINTJES et Alexander P. MORGAN. Chemical equilibrium systems as numerical test problems. *ACM Transactions on Mathematical Software*, 16(2):143–151, juin 1990.

[165] Valérie MÉNISSIER-MORAIN. Arbitrary precision real arithmetic: design and algorithms. Soumis au Journal of Symbolic Computation, septembre 1996.

[166] Roger MOHR et Thomas C. HENDERSON. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[167] Ramon Edgar MOORE. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N. J., 1966.

[168] Ramon Edgar MOORE. *Methods and Applications of Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979.

[169] Jorge J. MORÉ et Michel Yves COSNARD. Numerical solutions of nonlinear equations. *ACM Transactions on Mathematical Software*, 5:64–85, 1979.

[170] Jorge J. MORÉ, Burton S. GARBOW et Kenneth E. HILLSTROM. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, mars 1981.

[171] Jean-Michel MULLER. Vers des primitives propres en arithmétique des ordinateurs. *Techniques et sciences informatiques*, 19:1–5, 2000.

[172] Jean-Michel MULLER et Arnaud TISSERAND. Towards exact rounding of the elementary functions. Rapport technique RR 95-33, Laboratoire d'Informatique du Parallélisme, École Normale Supérieure de Lyon, novembre 1995.

[173] Nathan C. MYERS. A new and useful technique: "traits". *C++ Report*, 7(5):32–35, juin 1995.

[174] Ryosei NAKAZAKI, Akihiko KONAGAYA, Shinichi HABATA, Hideo SHIMAZU, Mamoru UMEMURA, Masahiro YAMAMOTO, Minoru YOKOTA et Takashi CHIKAYAMA. Design of a high-speed Prolog machine (hpm). In *Proceedings of the 12th Annual Symposium on Computer Architecture*, pages 191–197, Boston, MA, USA, juin 1985.

[175] Arnold NEUMAIER. *Interval methods for systems of equations*, volume 37 of *Encyclopedia of Mathematics and its Applications*. Cambridge Unversity Press, 1990.

[176] William J. OLDER et Frédéric BENHAMOU. Programming in `clp(bnr)`. In *Proceedings of PPCP'93*, 1993.

[177] William J. OLDER et André VELLINO. Extending Prolog with constraint arithmetic on real intervals. In *Proc. of IEEE Canadian conference on Electrical and Computer Engineering*, New York, 1990. IEEE Computer Society Press.

[178] William J. OLDER et André VELLINO. Constraint arithmetic on real intervals. In Frédéric BENHAMOU et Alain Marie Albert COLMERAUER, réds., *Constraint Logic Programming: Selected Papers*. The MIT Press, 1993.

[179] John K. OUSTERHOUT. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, 1994. Seventh Printing,

[180] Michael L. OVERTON. Floating point representation. Unpublished note, 1996,

[181] Petru PAU et Josef SCHICHO. Quantifier elimination for trigonometric polynomials by cylindrical trigonometric decomposition. Technical report, Research Institute for Symbolic Computation, mars 1999.

[182] Michèle PICHAT et Jean VIGNES. *Ingénierie du contrôle de la précision des calculs sur ordinateur*. Collection Informatique. Éditions TECHNIP, 1993.

[183] Michèle PICHAT. Correction d'une somme en arithmétique à virgule flottante. *Numerische Mathematik*, 19:400–406, 1972.

[184] Evgenija D. POPOVA et Christian P. ULLRICH. A generalization of bias specification. Technical report 95-8, URZ+IFI, 1995.

[185] Evgenija D. POPOVA et Christian P. ULLRICH. Directed interval arithmetic in Mathematica: Implementation and applications. Research report 96-3, University of Basel, Department of Computer Science, janvier 1996.

[186] Evgenija D. POPOVA et Christian P. ULLRICH. Simplification of symbolic-numerical interval expressions. In *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*, pages 207–214. ACM Press, 1998. Also available as Research Report 97-1 from the University of Basel,

[187] Peter John POTTS et Abbas EDALAT. Exact real computer arithmetic. Manuscript,

[188] Douglas M. PRIEST. Differences among IEEE 754 implementations. Addendum to Appendix D of David Goldberg's "What Every Computer Scientist should Know About Floating-Point Arithmetic" in Sun's "Numerical Computation Guide".

[189] Douglas M. PRIEST. Algorithms for arbitrary precision floating point arithmetic. In Peter KORNERUP et David W. MATULA, réds., *Proceedings: 10th IEEE Symposium on Computer Arithmetic: June 26–28, 1991, Grenoble, France*, pages 132–143, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1991. IEEE Computer Society Press.

[190] Douglas M. PRIEST. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Thèse de doctorat, Department of Computer Science, University of California, Berkeley, Berkeley, CA, USA, décembre 1992. Disponible à l'adresse `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

[191] Douglas M. PRIEST. Handling IEEE 754 invalid operation exceptions in real interval arithmetic. Manuscript, mai 1997,

[192] ISO PROLOG. ISO standard for the Prolog language. Prolog: Part1, general core. Rapport technique ISO/IEC JTC1 SC22 WG17 N110, The University of Georgia, Athens, Georgia, USA, 1994.

[193] Germán PUEBLA, Francisco BUENO et Manuel HERMENEGILDO. A framework for assertion-based debugging in constraint logic programming. In *Proceedings of the ninth International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99)*, septembre 1999.

[194] Jean-François PUGET. A C++ implementation of CLP. In *Proceedings of the Singapore Conference on Intelligent Systems (SPICIS'94)*, Singapore, 1994.

[195] Jean-François PUGET et Michel LECONTE. Beyond the glass box: Constraints as objects. In *Proceedings of the 1995 International Logic Programming Symposium (ILPS'95)*, 1995.

[196] Jean-François PUGET et Pascal VAN HENTENRYCK. A constraint satisfaction approach to a circuit design problem. *Journal of Global Optimization*, 13:75–93, 1998.

[197] Helmut RATSCHEK et Jon ROKNE. Interval methods. In *Handbook of Global Optimization*, pages 751–828. Kluwer Academic, 1995.

[198] Dietmar RATZ. Inclusion isotone extended interval arithmetic. Technical Report 5/1996, Institut für Angewandte Mathematik, Universität Karlsruhe, 1996.

[199] G. RODA. Quantifier elimination. Lecture notes based on a course by George E. Collins, juillet 1996. RISC – Summer Semester '96,

[200] Jon G. ROKNE. Bounds for an interval polynomial. *Computing*, 18:225–240, 1977.

[201] Zsofia RUTTKAY et Han NOOT. Solution strategies to produce facial animations. In *Procs. of the ERCIM CompulogNet workshop*. ERCIM, juin 2000.

[202] Jamila SAM. *Constraint Consistency Techniques for Continuous Domains.* Thèse de doctorat, École polytechnique fédérale de Lausanne, 1995.

[203] Jamila SAM-HAROUD et Boi V. FALTINGS. Consistency techniques for continuous constraints. *Constraints*, 1:85–118, 1996.

[204] Michael SANNELLA. Constraint satisfaction and debugging for interactive user interfaces. Technical report 94-09-10, Department of Computer Science and Engineering, University of Washington, septembre 1994. Revised version of the Ph.D. dissertation,

[205] Michael SANNELLA. The skyblue constraint solver and its applications. In Vijay A. SARASWAT et Pascal VAN HENTENRYCK, réds., *Proceedings of the 1993 Workshop on Principles and Practice of Constraint Programming*. The MIT Press, 1994.

[206] Michael SANNELLA et Alan BORNING. Multi-garnet: Integrating multi-way constraints with garnet. Technical Report TR 92-07-01, Department of Computer Science and Engineering, University of Washington, septembre 1992.

[207] Christian SCHULTE. Oz explorer: A visual constraint programming tool. In Lee NAISH, réd., *Proceedings of the 14th International Conference on Logic Programming*, pages 286–300, Cambridge, juillet 1997. The MIT Press.

[208] Michael J. SCHULTE et Earl E. SWARTZLANDER. Exact rounding of certain elementary functions. In M. J. IRWIN, Earl E. SWARTZLANDER et G. JULLIEN, réds., *Proceedings of the 11th Symposium on Computer Arithmetic*, pages 138–145, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[209] Irene A. SHARAYA. On maximal inner estimation of the solution sets of linear systems with interval parameters. *Computational Technologies*, 3(2):55–66, 1998.

[210] Sergey Petrovich SHARY. Algebraic solutions to interval linear equations and their applications. In Götz ALEFELD et Jürgen HERZBERGER, réds., *Numerical Methods and Error Bounds, proceedings of the IMACS-GAMM International Symposium on Numerical Methods and Error Bounds*, pages 224–233, Oldenburg, Germany, juillet 1995. Akademie Verlag.

[211] Sergey Petrovich SHARY. Algebraic approach to the interval linear static identification, tolerance, and control problems, or one more application of Kaucher arithmetic. *Reliable Computing*, 2(1):3–33, 1996.

[212] Sergey Petrovich SHARY. A new approach to the analysis of static systems under interval uncertainty. In Götz ALEFELD, Andreas FROMMER et B. LANG, réds., *Scientific Computing and Validated Numerics*, pages 118–132. Akademie Verlag, Berlin, 1996.

[213] Sergey Petrovich SHARY. Interval Gauss-Seidel method for generalized solution sets to interval linear systems. In *MISC'99—Workshop on Applications of Interval Analysis to Systems and Control*, pages 51–65, Girona, Spain, février 1999.

[214] Helmut SIMONIS et Abderrahmane AGGOUN. Search tree debugging. In François FAGES, réd., *Actes des huitièmes journées francophones de programmation logique et programmation par contraintes*, pages 265–280. HERMES, juin 1999.

[215] David SINGMASTER. Quelques divertissements numériques. *La Recherche*, 26(278):818–823, Juillet/Août 1995. Numéro Spécial Nombres,

[216] Gert SMOLKA. The definition of Kernel Oz. In Andreas PODELSKI, réd., *Constraint Programming: Basics and Trends*, volume 910 of *Châtillon Spring School, Lecture Notes in Computer Science*, pages 251–292. Springer-Verlag, 1994.

[217] John M. SNYDER. Interval analysis for computer graphics. In Edwin E. CATMULL, réd., *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26-2, pages 121–130, juillet 1992.

[218] Volker STAHL. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations.* Thèse de doctorat, Johannes Kepler Universität, Linz, septembre 1995.

[219] Richard M. STALLMAN et Roland H. PESCH. *Debugging with GDB, The GNU Source-Level Debugger*. Free Software Foundation, Inc., Boston, MA, fifth édition, avril 1998.

[220] Ole STAUNING. *Automatic Validation of Numerical Solutions*. Thèse de doctorat, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, août 1997. IMM-PHD-1997-36,

[221] Alexander STEPANOV et Meng LEE. *The Standard Template Library*. Silicon Graphics Inc. and Hewlett-Packard Laboratories, octobre 1995.

[222] Josef STOER et Roland BULIRSCH. *Introduction to Numerical Analysis*. Number 12 in Texts in Applied Mathematics. Springer, second édition, 1993.

[223] Jorge STOLFI et Luiz Henrique de FIGUEIREDO. Self-validated numerical methods and applications. Courses notes of the 21st Brazilian Mathematics Colloquium, juillet 1997.

[224] Douglas STOTT PARKER, JR.. Monte Carlo arithmetic: exploiting randomness in floating-point arithmetic. Technical report CSD-970002, Computer Science Department, University of California, mars 1997.

[225] Douglas STOTT PARKER, JR., Brad PIERCE et Paul R. EGGERT. Monte Carlo arithmetic: a framework for the statistical analysis of roundoff error. Technical report CSD-970014, Computer Science Department, University of California, juin 1999. À paraître in *IEEE Computation in Science and Engineering*,

[226] Bjarne STROUSTRUP. *The C++ programming language*. Addison Wesley, troisième édition, 1997.

[227] Konstantinos TARABANIS. Automated sensor planning and modeling for robotic vision tasks. Rapport technique CUCS-045-90, University of Columbia, 1990.

[228] Alexandre TESSIER. Declarative debugging in constraint logic programming. In Joxan JAFFAR et Roland H. C. YAP, réds., *Asian Computing Science Conference*, volume 1179 of *Lecture Notes in Computer Science*, pages 64–73. Springer-Verlag, 1996. (Also available as Technical Report 96/09, LIFO, University of Orléans,),

[229] Lloyd Nicholas TREFETHEN. The definition of numerical analysis. *SIAM News*, 25, novembre 1992. Reprinted in the *Bulletin* of the Institute for Mathematics and its Applications, 1993.,

[230] UNITED STATES GENERAL ACCOUNTING OFFICE. Patriot missile defense—software problem led to system failure at Dahran, Saudia Arabia. Rapport technique GAO/IMTEC-92-26, Information Management and Technology Division, Washington, District of Columbia, 1992.

[231] Maarten Herman VAN EMDEN. Canonical extensions as common basis for interval constraints and interval arithmetic. In Frédéric BENHAMOU, réd., *Actes des 6e journées francophones de programmation logique et programmation par contraintes*, pages 71–83. Hermès, 1997.

[232] Pascal VAN HENTENRYCK, Yves DEVILLE et Choh-Man TENG. A generic arc consistency algorithm and its specializations. Rapport technique R.R. 91-22, Université Catholique de Louvain, Faculté des Sciences Appliquées, décembre 1991.

[233] Pascal VAN HENTENRYCK et Laurent MICHEL. Helios: A modeling language for nonlinear constraint solving and global optimization using interval analysis. Rapport technique CS-95-33, Brown University, novembre 1995.

[234] Pascal VAN HENTENRYCK, Laurent MICHEL et Frédéric BENHAMOU. `Newton`: Constraint programming over nonlinear constraints. *Science of Computer Programming*, 30(1–2):83–118, janvier 1998.

[235] Pascal VAN HENTENRYCK, Laurent MICHEL et Yves DEVILLE. *Numerica: A Modeling Language for Global Optimization*. The MIT Press, 1997.

[236] Ronald J. VAN IWAARDEN. Global optimization using VerGO. Manuscript, Disponible à l'adresse `http://www.cs.hope.edu/~rvaniwaa/VerGO/VerGO.html`.

[237] Josep VEHÍ, Joaquim ARMENGOL et Miguel Ángel SÁINZ. Robust stability of interval plants with polynomial coefficients using modal interval arithmetic. Research report 96/09-EIA, Departament d'Electrònica, Informàtica i Automàtica (EIA), University of Girona, 1996.

[238] Todd L. VELDHUIZEN. Algorithm specialization in C++. Early version of an article to appear in the May 1995 issue of C++ report,

[239] Todd L. VELDHUIZEN. C++ templates as partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, 1999.

[240] Todd L. VELDHUIZEN. Techniques for scientific C++. Manuscript, août 1999, Disponible à l'adresse `http://extreme.indiana.edu/ tveldhui/papers/techniques/`.

[241] Jürgen Freiherr Wolff von GUDENBERG. Hardware support for interval arithmetic. Rapport technique 125, Lehrstuhl für Informatik II, Universität Würzburg, octobre 1995. Extended Version,

[242] Richard J. WALLACE et Eugene C. FREUDER. Ordering heuristics for arc consistency algorithms. In *Proceedings of CAI'92*, 1992.

[243] G. William WALSTER. The extended real interval system. Manuscript, mars 1998,

[244] G. William WALSTER et Eldon Robert HANSEN. Interval algebra, composite functions and dependence in compilers. Manuscript, avril 1998, Disponible à l'adresse `http://www.mscs.mu.edu/ globsol/walster-papers.html`.

[245] David I. WALTZ. Generating semantic descriptions from drawings of scenes with shadows. In P. H. WINSTON, réd., *The Psychology of Computer Vision*, chapter 3. McGraw-Hill, New York, 1975.

[246] Allen C. WARD, Tomás LOZANO-PÉREZ et Warren P. SEERING. Extending the constraint propagation of intervals. In *Proceedings of IJCAI'89*, pages 1453–1458, 1989.

[247] David H. D. WARREN. An abstract Prolog instruction set. Rapport technique 309, SRI International, octobre 1983.

[248] James Hardy WILKINSON. *Rounding errors in algebraic processes.* Prentice-Hall, 1963. Réédité chez Dover Publications (1994),

[249] James Hardy WILKINSON. *The algebraic Eigenvalue problem*. Oxford University Press, 1965.

[250] James Hardy WILKINSON. Modern error analysis. *SIAM Review*, 13(4):548–568, octobre 1971.

[251] R. C. YOUNG. The algebra of many-valued quantities. *Math. Annalen*, 104:260–290, 1931.

[252] Jianyang ZHOU. Approximate solving of $y = \cos(x)$ and other real constraints by cartesian product of intervals. Rapport technique, LIM, Université de Marseille, 1994.

[253] Abraham ZIV. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, septembre 1991.

[254] V. S. ZYUZIN. On a way of finding two-sided approximation to the solution of system of linear interval equations. In *Differential Equations and the Theory of Functions*, pages 28–32. Saratov State University, 1987. (En russe),

Liste des tableaux

Partie I — L'arithmétique des intervalles

1.1 Cas d'apparition d'un NaN pour les opérateurs arithmétiques définis par la norme IEEE 754 . . . 7
1.2 Caractéristiques des formats `single` et `double` . . . 9
1.3 Cas spéciaux pour les opérateurs arithmétiques non définis par la norme IEEE 754 . . . 11
1.1 Promotion et norme IEEE 754 . . . 13
1.4 Sur-optimisation par le compilateur . . . 14

Partie II — Consistances locales

3.1 Algorithme de filtrage Nar . . . 39
3.1 Décomposition en *contraintes « primitives »* . . . 41
3.2 Algorithme de filtrage HC3 . . . 43
3.3 Algorithme HC3revise . . . 43
3.4 Algorithme de filtrage BC3 . . . 44
3.5 Algorithme BC3revise . . . 45
3.6 Fonction RéductionGauche . . . 46

4.1 Algorithme HC4revise . . . 50
4.2 Algorithme d'évaluation ascendante . . . 51
4.3 Algorithme de propagation descendante . . . 52
4.4 Algorithme de filtrage HC4 . . . 53
4.5 Algorithme BC4 . . . 55
4.1 Résultats expérimentaux . . . 56

Partie III — Approximations intérieures et variables quantifiées

5.1 Schéma général de calcul d'approximation intérieure par évaluation/découpage . . . 69
5.1 Arithmétique de KAUCHER . . . 73
5.2 Opérations arithmétiques intérieures de MARKOV . . . 75

6.1 EIA4 – Algorithme d'évaluation/découpage pour la contrainte $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v)\colon (c_1 \wedge \cdots \wedge c_m)$ . . . 81
6.2 ICA3$_c$ – Algorithme de résolution de $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v)\colon c$ . . . 82
6.3 ICA4 – Algorithme de résolution de $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v)\colon (c_1 \wedge \cdots \wedge c_m)$ . . . 83
6.4 ICAb3$_c$ – Algorithm de résolution de la contrainte $\forall v \in \mathsf{Dom}_{\boldsymbol{B}}(v)\colon c$ . . . 84
6.5 ICAb5 – Algorithme de résolution pour la contrainte $\forall v \in I^1\colon c_1 \wedge \cdots \wedge \forall v \in I^m\colon c_m$ . . . 85

Partie IV — Environnements de programmation en PC

6.1 Le problème de Yoshigahara résolu en Java . . . 90
6.2 Le problème de YOSHIGAHARA [215] résolu en DecLIC . . . 90

8.1 Programme DecLIC pour le problème du pentagone régulier [151] . . . 100

8.2 Le programme DecLIC pour le système de fonctions *Broyden-Banded* [170] 100
8.3 Utilisation conjointe de consistances 101
8.1 Résultats expérimentaux : une seule consistance 105
8.2 Résultats expérimentaux : plusieurs consistances 105
8.3 Grammaire de la contrainte X *in* R dans clp(fd) 114
8.1 Algorithme de propagation de DecLIC 121
8.4 Registres de DecLIC 121
8.5 Décomposition et compilation d'une D-contrainte 123
8.2 Propagation dans la NPL 126

10.1 Le programme P 135
10.1 RevIncNar : Nar revisité pour gérer les *S-boxes* 140
10.2 Le code clp(FD) pour `cars` 142
10.1 Registres ajoutés à clp(fd) 147

Partie V — Programmation par contraintes et programmation orientée objets

11.1 La classe `Nombre` sans *trait* 157
11.2 Un exemple d'utilisation de *traits* 158
11.3 La classe `Nombre` avec *traits* 159
11.4 Instanciation du trait de JAIL pour le type `double` 160
11.5 Évaluation de la fonction de `Makino/Berz` [155] 161
11.1 Valeurs utilisées pour les tests 163
11.2 Nombre d'opérations effectuées par opérateur testé 164
11.3 Impact des choix d'implantation de JAIL (version paramétrée) 164
11.4 Impact des choix d'implantation de JAIL (version non paramétrée) 165
11.5 Comparaisons de librairies pour l'arithmétique d'intervalles sur une UltraSparc 1/167 MHz 166
11.6 Comparaisons de librairies pour l'arithmétique d'intervalles sur un Bi-Pentium III 500 MHz . . . 166
11.7 Performances en millions d'opérations-intervalles par seconde 167

12.1 Le problème *circle-parabola* résolu avec OpAC 170
12.1 EIA4(ω) *vs.* ICAb4 — première solution 175
12.2 EIA4(ω) *vs.* ICAb4 — toutes les solutions 175

Table des figures

Partie I — L'arithmétique des intervalles

1.1 Représentation d'un nombre flottant en mémoire . . . 6
1.2 Un circuit avec deux résistances en parallèle . . . 6
1.3 Représentation d'un nombre flottant en format *tiny* . . . 7
1.4 Répartition des flottants normalisés en format *tiny* . . . 8
1.5 Répartition des flottants en format *tiny* avec les nombres dénormalisés . . . 8
1.6 Interprétation du triplet (e, s, f) pour le format `double` . . . 9
1.7 Les différents types d'arrondi . . . 10

2.1 $\mathsf{Hull}(\rho)$ *vs.* $\mathsf{Union}(\rho)$ . . . 18
2.2 Différentes fonctions d'inclusion pour f . . . 20
2.3 Évaluation de $f(x) = (x^6/4 - 1.9x^5 + 0.25x^4 + 20x^3 - 17x^2 - 55.6x + 48)/50$ en forme naturelle ($\mathsf{w}(I) = 0.125$) . . . 23
2.4 Évaluation de $f(x) = (x^6/4 - 1.9x^5 + 0.25x^4 + 20x^3 - 17x^2 - 55.6x + 48)/50$ en forme naturelle ($\mathsf{w}(I) = 0.02$) . . . 23
2.5 Évaluation de $f(x) = (x^6/4 - 1.9x^5 + 0.25x^4 + 20x^3 - 17x^2 - 55.6x + 48)/50$ en forme de HORNER ($\mathsf{w}(I) = 0.125$) . . . 24
2.6 Positions relatives de ρ et $\boldsymbol{B}$ telles que $\rho \cap \boldsymbol{B} = \varnothing$ . . . 28
2.7 Valeurs possibles pour x . . . 29

Partie II — Consistances locales

2.8 Un réseau de contraintes discrètes . . . 33
2.9 Résolution pas-à-pas du système (2.19) . . . 34

3.1 Un système de contraintes et son *store* associé . . . 40
3.2 Décomposition naturelle de la contrainte c: $2x = z - y^2$ . . . 42
3.3 Calcul de la box-consistance pour $f(x) = 0$ . . . 46
3.4 Comparaison des différentes consistances . . . 47

4.1 Arbre annoté pour l'évaluation ascendante de la contrainte $2x = z - y^2$ . . . 51
4.2 Arbre annoté pour la propagation descendante de la contrainte $2x = z - y^2$ . . . 53

Partie III — Approximations intérieures et variables quantifiées

4.3 Classification de WARD *et al.* . . . 59
4.4 Un système contrôlé par rétroaction . . . 60
4.5 Un problème de collision . . . 60

5.1 Étape de projection pour $p = (x_1 - 2)^2 + (x_2 - 2)^2 + (x_3 - 2)^2 - 1$ . . . 65
5.2 Résolution de $\forall y$: $(x^2 + y^2 - 1 > 0 \wedge x^3 - y^2 < 0)$ par CAD . . . 67
5.3 Différentes approximations intérieures d'une relation réelle . . . 68
5.4 Évaluation de $f(x)$ en forme de BERNSTEIN ($\mathsf{w}(I_j) = 0.02$) . . . 71

Partie IV — Environnements de programmation en PC

7.1 Chronologie pour les langages de programmation par contraintes d'intervalles 95

8.1 Problème du pentagone régulier 99
8.2 Un compilateur Prolog en deux parties 106
8.3 Gestion de la mémoire par la WAM 107
8.4 Environnement de clause 108
8.5 Structure d'un point de choix 109
8.6 Exemples de représentation de termes 110
8.7 Structure pour les arguments 115
8.8 Structure pour une contrainte 115
8.9 Représentation d'une variable DF dans clp(fd) 116
8.10 *Store* pour le programme de l'exemple 8.23 120
8.11 Structure pour une N-contrainte 122
8.12 Structure pour une variable 122
8.13 Réseau de contraintes pour l'exemple 8.26 125

9.1 Interface graphique de Grace 130
9.2 Représentation du *store* dans Grace 130
9.3 Interface de *Oz Explorer* 131
9.4 Un store de contraintes dans *Multi-Garnet* 132
9.5 Arbre de recherche et propagation dans CHIP 133

10.1 *Store* de P avec les contraintes de l'utilisateur 136
10.2 Structure du *store* pour le programme P (tous les buts marqués) 136
10.3 Abstraction du *store* pour le programme P 137
10.4 Le nouveau *store* après focus sur `object_ B_ moving`(T, X, Y, Z) 137
10.5 Représentation en arbre de la hiérarchie de boîtes/*S-boxes* de la figure 10.2 138
10.6 Hiérarchie de *S-boxes* de la figure 10.5 avec la notation de DEWEY 139
10.7 Utilisation du débogueur pour `cars.pl` sans *S-boxes* 141
10.8 Utilisation du débogueur pour `cars.pl` avec *S-boxes* 143
10.9 Ajout de S-boxes dans le texte par marquage de buts 144
10.10 Cession de débogage 145
10.11 Architecture du débogueur 146
10.12 Structure pour la représentation des points de choix 147
10.13 Structure du trail 148
10.14 Structure pour la représentation des contraintes 149
10.15 Structure pour la représentation des variables 150
10.16 Les piles internes du débogueur 150

Partie V — Programmation par contraintes et programmation orientée objets

12.1 La hiérarchie de classes dans OpAC 170
12.2 Résolution avec WOpAC du problème *Collision*$_{4,1}$ 172
12.3 Premier problème de GARLOFF & GRAF [82] 173
12.4 Deuxième problème de GARLOFF & GRAF [82] 174
12.5 *Depth-first vs. semi-depth first* 174

Crédit photo

Les figure 12.3(a) et 12.4(a) sont extraites d'un article de Jürgen GARLOFF et Birgit GRAF [82] et sont reproduites ici avec l'aimable autorisation des auteurs.
La figure 9.3 est extraite d'un article de Christian SCHULTE [207] et est reproduite ici avec l'aimable autorisation de l'auteur.
Les dessins 12.2, 12.3(b), 12.4(b) et 12.5 ont été réalisés avec WOpAC et USV par Marc CHRISTIE.

Table des exemples

1 Calcul en précision bornée [18] . . . 3
1.2 Absorption et cancellation . . . 9
1.3 Table maker's dilemma . . . 10
1.4 Résultats différents malgré le respect de la norme IEEE 754 . . . 12
2.5 Différentes extensions aux intervalles . . . 20
2.6 Forme imbriquée . . . 22
2.7 Différentes extensions de relations . . . 24
2.8 Résolution par propagation de domaines . . . 34
3.9 Relation associée à une contrainte . . . 37
3.10 Un CNO pour la contrainte $x + y = z$ . . . 40
3.11 Décomposition naturelle . . . 42
4.12 Un opérateur de contraction implémenté par HC4revise . . . 52
4.13 Un système contrôlé par rétroaction [1] . . . 59
4.14 Problème de collision . . . 60
5.15 Étape de projection [121] . . . 64
5.16 Résolution par CAD de contraintes avec quantificateurs . . . 65
5.17 Forme de BERNSTEIN . . . 70
5.18 Interprétation pour l'arithmétique modale [98] . . . 74
6.19 Problème de YOSHIGAHARA [215] . . . 89
8.20 Utilisation d'indexicaux . . . 113
8.21 Environnement pour contraintes . . . 115
8.22 Calcul d'un intervalle par des expressions pour les bornes . . . 119
8.23 Un *store* dans DecLIC . . . 119
8.24 Compilation de la contrainte `X in R` . . . 123
8.25 compilation d'une D-contrainte . . . 123
8.26 Compilation des N-contraintes . . . 124
10.27 Ordre sur les contraintes . . . 139
10.28 Création de *S-boxes* et structure clausale . . . 148
10.29 Traduction d'un programme marqué . . . 149
11.30 La classe `list` de la STL . . . 156
11.31 La classe paramétrée `Nombre` . . . 156
11.32 Caractéristiques de types flottants [173] . . . 157
12.33 Premier problème de GARLOFF & GRAF [82] . . . 172
12.34 Deuxième problème de GARLOFF & GRAF [82] . . . 172

Table des matières

Introduction **xi**

Notice **xv**

Partie I — L'arithmétique des intervalles

Introduction **3**

1 L'arithmétique flottante **5**
1.1 Présentation de la norme IEEE 754 5
1.1.1 Format des nombres en virgule flottante 5
1.1.2 L'arrondi 9
1.1.3 Opérateurs non définis par la norme IEEE 754 11
1.2 Propriétés de l'arithmétique flottante 12
1.3 Retour sur la norme IEEE 754 12

2 L'arithmétique d'intervalles **15**
2.1 Les intervalles à bornes réelles 15
2.1.1 Arithmétique des intervalles réels 17
2.1.2 Extensions aux intervalles de fonctions réelles 18
2.1.3 Extension aux intervalles de relations réelles 24
2.1.4 Propriétés 25
2.2 Les intervalles à bornes flottantes 25
2.2.1 Arithmétique (fonctionnelle) des intervalles flottants 26
2.2.2 Propriétés 27
2.2.3 L'arithmétique relationnelle des intervalles flottants 29

Partie II — Consistances locales

Introduction **33**

3 Consistances locales **37**
3.1 Contraintes réelles et contraintes d'intervalles 37
3.2 Contraintes et opérateurs de contraction 38
3.3 Consistance d'arc et consistance d'arc faible 39

4 Extension de la définition de box-consistance **49**
4.1 Calcul de la hull-consistance sans décomposition 49
4.2 Une nouvelle définition pour la box-consistance 54
4.3 Un nouvel algorithme pour la box-consistance 55
4.4 Résultats expérimentaux 56

Partie III — Approximations intérieures et variables quantifiées

Introduction **59**

5 Gestion de quantificateurs et calcul d'approximations intérieures **63**
5.1 Élimination de quantificateurs . . . 64
5.1.1 Décomposition algébrique cylindrique . . . 64
5.1.2 Décomposition trigonométrique cylindrique . . . 66
5.2 Quantificateurs universels et approximations intérieures . . . 67
5.2.1 Calcul d'approximation intérieure par évaluation/découpage . . . 69
5.2.2 Arithmétiques de KAUCHER/MARKOV et arithmétique modale . . . 72

6 Approximations intérieures, variables quantifiées et contraintes d'intervalles **77**
6.1 Approximation intérieure . . . 78
6.1.1 Opérateurs de contraction intérieurs . . . 78
6.2 Résolution de contraintes avec quantificateurs universels . . . 79
6.2.1 Résolution par évaluation/découpage . . . 80
6.2.2 Calcul d'approximation intérieure par négation . . . 80

Partie IV — Environnements de programmation en PC

Introduction **89**

7 État de l'art des langages de PCI **93**

8 DecLIC *(a Declarative Language with Interval Constraints)* **97**
8.1 Présentation du langage . . . 97
8.1.1 Utilisation de la hull-consistance . . . 98
8.1.2 Utilisation de la box-consistance . . . 99
8.1.3 Variables entières et variables réelles . . . 100
8.1.4 Hull-consistance et Box-consistance . . . 101
8.2 Résultats expérimentaux . . . 102
8.2.1 Présentation des problèmes de test . . . 102
8.3 Du choix de la consistance . . . 105
8.4 La Machine Abstraite de WARREN étendue de clp(fd) . . . 105
8.4.1 Gestion de la mémoire . . . 106
8.4.2 Instructions de contrôle . . . 107
8.4.3 Représentation des termes . . . 110
8.4.4 Chargement des registres et unification . . . 110
8.4.5 Exemples de compilation . . . 112
8.4.6 Extensions de la WAM pour clp(fd) . . . 113
8.5 Implémentation de DecLIC . . . 119
8.5.1 Le modèle de calcul . . . 119
8.5.2 Les structures de données . . . 120
8.5.3 Compilation de la contrainte X *in* R . . . 123
8.5.4 Gestion des contraintes lors de la compilation . . . 123
8.5.5 Gestion des contraintes à l'exécution . . . 124
8.5.6 Interactions entre les algorithmes de résolution . . . 126
8.5.7 Extension de la WAM pour DecLIC . . . 126

9 Le débogage en programmation par contraintes : état de l'art 129
9.1 Le débogueur Grace . . . 129
9.2 Oz Explorer . . . 131
9.3 Le débogueur de CHIP . . . 131
9.4 Le débogueur CNV . . . 132

10 Débogage par observation du *store* 135
10.1 Structuration du *store* : exemple . . . 135
10.2 Les *S-boxes* . . . 137
10.3 Présentation du débogueur . . . 140
10.4 Réalisation pratique . . . 146
10.4.1 Modification du processus de *backtracking* . . . 148
10.4.2 Modification de la méthode de propagation . . . 148
10.4.3 Gestion des *S-boxes* . . . 149

Partie V — Programmation par contraintes et programmation orientée objets

Introduction 153

11 Implémentation de l'arithmétique d'intervalles : JAIL 155
11.1 Présentation des techniques C++ . . . 156
11.1.1 Les classes paramétrées . . . 156
11.1.2 Les *traits* . . . 157
11.2 Fonctionnalités et implémentation de JAIL . . . 158
11.2.1 Architecture générale . . . 160
11.2.2 Implémentation des opérateurs . . . 161
11.3 Évaluation de JAIL . . . 163
11.3.1 Impact des choix d'implémentation . . . 163
11.3.2 Comparaison avec d'autres librairies . . . 165

12 Gestion de variables quantifiées : OpAC et WOpAC 169
12.1 Présentation d'OpAC . . . 169
12.2 Évaluation des algorithmes . . . 171
12.3 Comparaison EIA4 *vs.* ICAb4 . . . 173

Conclusion et perspectives 179

Bibliographie 183

Liste des tableaux 197

Table des figures 199

Table des exemples 203

Table des matières 205

Index 209

A Annexe à la partie II 223

B Annexe à la partie III **225**

Index

“If you don’t find it in the index, look very carefully through the entire catalogue.”

— Catalogue Sears & Rœbuck, 1897.

Index

P_1
problème, 104
P_2
problème, 104
i_1
problème, 102
i_2
problème, 103
i_4
problème, 103
[incr Tcl], 146, 149, 150

1994, 94

ABDALLAH, 70, 173
absorption, 8
accesseur
de borne, 16
accroissements finis
théorème des ~, 22
AGGOUN, 122
algebraic solution, 68
algébrique
ensemble, 68
algébrisation
viabilité, 66
aléatoire
arithmétique ~, voir CESTAC
and
connecteur DecLIC, 100
approximation
intérieure
faible, 68
faible maximale, 68
forte, 68
arbre
attribué, 50
de recherche, 129, 131
ARCHIMÈDE, 15
arithmétique
aléatoire, voir CESTAC
d'intervalles flottants
propriétés, 27
de KAUCHER, 72, 73
de MARKOV, 75
de Monte-Carlo, 4
modale, 74
relationnelle, 29
réelle exacte, 4
arithmétique modale
interprétation, 74
ARMENGOL, 74, 75
arrondi
au plus proche-pair, 12
cas d'~, 9
correct, 10, 13, 41
de borne, 26
différents types, 10
extérieur, 26
Arénaire
projet, 11
attribut
hérité, 50
synthétisé, 50
attribué
arbre, 50

backtracking, 107, 147
BackwardPropagation()
algorithme, 52
base
de GRÖBNER, 94
BAUMANN, 21
BAUMGARTNER, 157
BC3
algorithme, 44
BC3revise
algorithme, 45
BC4
algorithme, 55
BELDICEANU, 122
BENHAMOU, vii, xi, 1, 18, 34, 35, 38, 44, 93
BERCHTOLD, 70
BERLANDIER, 180
BERNSTEIN
forme de ~, 70
polynôme de ~, 70
BERNSTEIN, 20, 22, 24, 63, 69–71, 75, 172–174, 181, 193
biais
d'un exposant, 6
BIAS
Basic Interval Arithmetic Specification, 24, 158
Bifurcation
problème, 102
bit
de signe, 5
BLIEK, 15
BNR Prolog, 93, 94
BNR-Prolog, 98
borne

d'un intervalle flottant, 25
d'un intervalle réel, 16
extension, voir extension
Borneo, 13
Borneo, 13, 155
BORNING, 180
BOSSCHERE, 106
box-consistance, 44, 94
box$_{\mathcal{A}_1,\mathcal{A}_2,\Gamma}$-consistance, 54
box$_o$-consistance, 54
box$_\varphi$-consistance, 47
est un CNO, 47
opérateur de contraction, 47
box$_\varphi$-consistance, 102
bracket, 15
BRADBURY, Ray, 5
BRADBURY, 5
Broyden-Banded
problème de ~, 99
Broyden-Banded 10
problème, 104
BURKILL, 15

C
et promotion, 13
C, 5, 13, 106, 107, 115, 120, 123, 129, 146, 179
CAD
cylindrical algebraic decomposition, 64
cameraman virtuel, 77
cancellation, 9
canonique, voir intervalle, voir extension
CEBERIO, vii, 170
centrée
forme, 20, 21
CESTAC
méthode, 3
chargement
de registre, 110
Chemical Eq.
problème, 102
χ, 75
CHIP, 89
CHIP, 89, 129, 131–133
CHIPART, 172
CHIRIAEV, 24, 158, 179
choix
de consistance, 105
CHRISTIE, vii, 61, 153, 169, 170, 191
chronologique
vue, 94
CIAL, 94, 98
CLEARY, xi, 40–42, 93, 94
CLIP, xii, 94
CLP($\mathcal{R}$), 94
clp(BNR), xii, 89, 93, 94, 98, 135
clp(F), xii
clp(fd), 97
clp(FD), 140, 142, 179
clp(fd), xii, 34, 89, 90, 97, 98, 101, 105–107, 113–116, 119, 120, 122, 146–148
CNV, 129, 132
CODOGNET, vii, xii, 1, 97, 105–107, 110, 115, 119, 140
COLLAVIZZA, 45, 49, 54, 181
COLLINS, 60, 63, 64
COLLINS, 63
Collision$_{4,1}$, 172
COLMERAUER, vii, 1
compactification
de $\mathbb{R}$, 15
compensateur, 60
compilation
de X *in* R, 123
complémentation à deux, 6
consistance
choix, 105
d'arc, 33, 39
de chemin, 33
de nœud, 33
forte, 94
hull-~, 40
union-~, 40
contraction, voir opérateur
contrainte
d'intervalles, 37
D-~, 119
de placement, 132
extension, 37
force, 132
installation
instructions, 117
N-~, 119
non intervalle-convexe, 93
primitive, *41*
redondante, 94
relation associée, 37
représentation, 115
réelle, 37
structure, 120
temporelle, 69
controllable solution set, 68
contrôlable
ensemble, 68
Contrôle
et Estimation Stochastique des Arrondis de Calculs, voir CESTAC
contrôle
instruction de ~, 107
convergence
accélération, 94
ordre de ~, 19
correction
des résultats, 59
Cosinus, 56, 94

COSNARD, 56
Cosnard 20
    problème, 103
CRAY, 5
critère
    de LIÉNARD-CHIPART, 172
CRUZ, vii, 97
CSP
    Constraint Satisfaction Problem, 33
    cylindrifié, 38
    étendu, 38
CWI
    Centrum voor Wiskunde en Informatica, 153
cycle
    suppression, 94
cylindrification, 38
cylindrique
    décomposition trigonométrique ~, 66

D-contrainte, 119
DARCY, 155
DAVIS, xi, 34
DecLIC
    modèle de calcul, 120
DecLIC, xii–xiv, 22, 34, 35, 49, 87, 89–91, 93, 97–101, 105, 119–121, 123, 126, 179, 193
depth-first
    stratégie de découpage, 174
DEWEY, 139, 140, 149, 180
DIAZ, xii, 97, 106, 107, 115, 119, 140
DiSCiPl
    projet, xiii
distillation
    algorithmes de ~, 3
division
    par zéro
        traitement, 6
domaine, 16
    d'approximation, 18
    de définition, 18
    de variation, 19
    représentation, 115
double
    format
        extended, 5
`double`, 5
    format, 9
    interprétation, 9
dual
    d'un intervalle, 73
dual(), 73
débogage
    de correction, 129
    de performance, 129
décomposition
    algébrique, 203
    algébrique cylindrique, 64
    cylindrique, 203
    de $\mathbb{R}^n$, 203
    ensemble de ~, 41
    naturelle, 42, 53
    trigonométrique cylindrique, 66
délinéabilité, 64
    d'un ensemble de polynômes, 203
dénormalisé, voir nombre
dépendance
    problème de ~, 17

ECL^{i}PS^e, xiii, 129, 131, 135
ECSP, 38
EDALAT, 11
EIA4
    algorithme, 81
    complexité, 85
élimination
    de quantificateurs, voir quantificateur
élimination de GAUSS, 94
ensemble
    algébrique, 68
    contrôlable, 68
    semi-algébrique, 203
    tolérable, 67
    unifié, 67
énumération, 129
environmment
    de clause, 108
environnement, 115
epsilon
    de la machine, 7
ϵ
    de la machine, 7
epsilon-inflation, 160, 164
ε-solution set, 72
erreur, 129
ERSHOV, 171
étape
    d'extension (CAD), 64
    de base (CAD), 64
    de projection (CAD), 64
every
    interprétation ~, 59
exposant
    biaisé, voir biais
    d'un nombre flottant, 5
extension, voir intervalles
    aux bornes, 26
    canonique, 41
    certaine, 24
    choix, 22
    comparaison, 22
    d'une contrainte réelle, 37
    d'une relation, 24
    ensembliste, 24

faible, voir intervalles
lexicographique, 139
naturelle aux intervalles, 19
ordre de convergence, 20
possible, 24
extrapolation, 94

facteur
d'amélioration, 102
faible
approximation intérieure, 68
FALTINGS, 80
FAROUKI, 70
FASE, 171
Facial Analysis and Synthesis of Expressions, 153
projet, xiii
fd-variable, 98
`fdlibm`
librairie, 11
FERRAND, vii, 1
fi_lib, 155, 165, 167
fi_lib++, 155, 165, 167
FIKES, 33
`float`, 5
flottant
nombre ~, voir nombres flottants
Flying Saucer$_{4,1}$, 171
flying
machine, 5
focus
d'une boîte, 137
fonction
d'approximation, 18
d'inclusion, 19
monotone, 19
d'intervalle, 18
lipschitzienne, 21
paramétrée, 156
fonctionnelle
contrainte, 93
format
`double`, 9
`single`, 9
tiny, 6
des nombres flottants, voir nombres flottants
forme
centrée, 20, 21
de BERNSTEIN, 20, **70**
de TAYLOR, 20, 21
imbriquée, 20, 22
moyenne, 20, 22
naturelle, voir extension
prénexe, 64
forte
consistance, 94
Fortran, 179
Fortran 2000, 155

ForwardEvaluation
algorithme, 51
fractionnaire, voir partie
FREUDER, 34, 180

GARDEÑES, 75
GARDEÑES, 61
GARLOFF, 63, 69–71, 172–174, 181, 191, 193
GARLOFF/GRAF, 175
GAUSS
élimination, 94
GAUSS, xi, 94
GAUSS-SEIDEL, 94
gdb, 129
GEORGET, 180
globalisation, voir variable dangereuse
GlobSat(), 69
GNU Prolog, 94
GOLDBERG, 10, 12, 13
GPL
liste de propagation globale, 120
Grace, xiii, 129–132
gradual underflow, 8
GRAF, 63, 69–71, 172–174, 181, 191, 193
GRANVILLIERS, vii, 34, 56, 94
GRÖEBNER
bases, 94
GRÖBNER, 94

HANSEN, 17, 21
HC3, 93
HC3
algorithme, 43
HC3revise
algorithme, 43
correction, 42
HC4
algorithme, 53
HC4revise
algorithme, 50
Helios, 94
HICKEY, 17, 43, 94
HONG, 60, 70, 173
HOPE, 89
HORNER
forme de, 20
HORNER, 20, 22, 24
HORNER, 24
HPM
High-speed Prolog Machine, 106
Hull$_{\square}(\rho)$, 26
Hull$_{\circ}(\rho)$, 26
Hull(), 18
hull-consistance, 40
HYVÖNEN, 93

i-variable, 98

IBM 360/370, 5
ICA3
    algorithme, 82
ICA4
    algorithme, 83
ICAb3
    algorithme, 84
ICAb5
    algorithme, 85
ICL, 98
ICLP, xii
ILOG Solver, 89
ILOG Solver, 89, 153
Ilog Solver, 94
imbriquée
    forme ~, 20
improper interval, 73
impérative
    prog. ~ avec contraintes, 94
inclusion
    fonction d'~, 19
IncNar
    algorithme, 121
[incr Tcl], 146
indexical, 113
industriel
    contexte, 94
inexistence de racine
    critère, 72
infini, 6
Intel, 160
interface, 157
interfaçage
    instructions d'~, 116
Interlog II, 93
interprétation
    d'un pavé solution, 59
    du format `double`, 9
intervalle
    arithmétique, 17
        propriétés, 25
    canonique, 26
    centre, 19
    dirigé, 72
    dual, 73
    et division par 0, 17
    flottant, 25
        notation, 25
    impropre, 73, 74
    modal, 74
    non ~-convexité, 93
    notation, 16
    propre, 72, 74
        associé, 73
    représentation d'un ens. réel par un ~, 16
    réel, 16
    taille, 19
    vide, 16
    étendu, 72
intervalles
    extension *faible* aux ~, 19
    extension aux ~, 19
    union d'~, 16
intérieur
    d'un intervalle, 26
intérieure
    opération, 75
invariant
    polynôme ~, 64

JAFFAR, 37, 41
JAIL, xiii, xiv, 151, 153, 155, 156, 158, 160–167, 179
JANSSES, 106
JARDILLIER, 61, 63, 69, 77, 79, 80, 171, 174
Java, 13, 89, 90, 155, 157
JIRSTRAND, 63

k-B-consistance, 93
K-C-S
    rapport, 5
KAHAN, William Morton, 3
KAHAN, 3, 4, 10, 13, 41, 155
κ
    morphisme, 19
KAUCHER
    arithmétique de ~, 73
KAUCHER, 17, 61, 63, 72–75, 79, 179, 181
KAUCHER/MARKOV, 72
$kB(w)$-consistance, 94
KEARFOTT, 15
KEIROUTZ, 89
KEIROUZ, 89
KNUTH, 10, 12, 16
KUPRIYANOVA, 67, 74
KUTSIA, 63, 70, 72

LA PORTE, Michel, 3
LA PORTE, 3
LANGUÉNOU, vii, 61, 63, 69, 77, 79, 80, 171, 174
LEBBAH, 50, 94
LEFEVRE, 11
LERCH, 165, 167
LHOMME, xi, 34, 93
lipschitzienne
    fonction, 21
LIÉNARD, 172
logarithmique
    notation ~, voir notation
logique trivaluée, 80
`long double`, 5

MØLLER, 3
machine abstraite de WARREN, 97

MACKWORTH, 33, 34
MAHER, 37, 41
mantisse
    d'un nombre flottant, 5
MARKOV
    arithmétique de ~, 75
MARKOV, 17, 61, 63, 72, 73, 75, 79, 179
Mathematica, 179
MCALLESTER, 33
mean value form, voir forme
MIELGO, 61
missile
    Patriot, 3
    Scud, 3
modalité
    d'un intervalle, 74
modélisation
    langage, 94
monotone
    fonction d'inclusion ~, 19
Monte-Carlo, voir arithmétique
MOORE, xi, 15, 17, 19, 20, 25
MORÉ, 56
moyenne
    forme, voir forme
MULLER, 10
Multi-Garnet, 129, 132
multiplicité d'une variable, 18
MYERS, 157

N-contrainte, 119
NaN
    Not a Number, 6
    apparition d'un ~, 7, 11
    non ordonné, 13
    utilisation dans JAIL, 162
    utilisation du champ f, 8
Nar
    propriétés, 39
Nar()
    algorithme, 39
naturelle, voir extension
    décomposition, voir décomposition
nested
    form, voir imbriquée
NEUMAIER, 15
NEWTON, xi, 17, 22, 45, 94, 101
Newton, xii, 94
NEWTON-RAPHSON, xi
`nextafter()`, 9
nombre
    dénormalisé, 8
nombre flottant
    normalisé, 7
nombres flottants
    formats, 5
none
    interprétation ~, 59
normalisation
    effet pervers, 8
normalisé
    nombre ~ et norme IEEE 754, 7
    nombre flottant, 7
notation
    de DEWEY, 139
    logarithmique de RUTISHAUSER, 6
NPL
    liste de propagation NEWTON, 120
Numerica, xii, 94
Numerical analysis
    as study of rounding errors, 3

occurrences
    théorème des ~ simples, 20
occurrences simples
    théorème des ~, 25
OCI
    opérateur de contraction intérieur, 78
OLDER, xi, 18, 38, 93
only
    interprétation ~, 59
OpAC
    an Open Architecture for Constraints, 169
OpAC
    architecture, 170
OpAC, xiii, xiv, 151, 153, 169, 179
Opium, 129
optimisation, 94
    sur-~ par le compilateur, 13
opérateur
    arithmétique intérieur, 75
    d'approximation intérieure, 78
    de TAYLOR, 94
    de contraction, 38
        optimal, 38
    de contraction de projection, 40, **41**
    de contraction intérieur, 78
    de contraction intérieure
        optimal, 79
opération
    arithmétique intérieure, 75
ordre
    de convergence, 19, 20
Oz, xiii, 129, 131, 135
Oz Explorer, 129, 131, 132
Oz explorer, xiii, 129

p-variable, 98
parenthésage, voir *bracket*
partie fractionnaire, 6
partition
    d'une relation réelle, 203
passe
    ascendante d'évaluation, 51

de propagation descendante, 51, 52
Patriot
missile, 3
PAU, 60, 63, 66, 67
pavé
interprétation, 59
taille, 19
paysan russe
méthode du ~, 165
PDL
Push-Down List, 107
pentagone
problème du ~, 98
perturbationnel
modèle, 132
PETROV, vii, 171
PICHAT, Michèle, 3
PICHAT, 3, 41
pile
au-dessus d'un connexe de $\mathbb{R}^n$, 203
de restauration, 107
globale, 107
locale, 107
pipe-line, 161
point
remarquable, 64
point de choix, 107, 109
structure, 109
structure de représentation, 147
polynôme
de BERNSTEIN, 70
invariant, 64
trigonométrique, 66
POPOVA, 61, 74, 179
possibly relation, 160
POTTS, 11
PRIEST, 12, 13, 155, 165
primitive, voir contrainte
contrainte ~ au sens de CLEARY, 41
Profil, 155, 165, 167, 179
programmation
impérative avec contraintes, 94
Projection$_{3,4}$, 171
projection
d'une contrainte, 44
d'une relation, 18
projet
Arénaire, 11
DiSCiPl, xiii
FASE, xiii
Prolog IV, 89
Prolog, xi, 90, 93, 94, 97, 98, 106–108, 112, 115, 116, 129, 140, 142, 148, 153, 180
Prolog IV, xii, 89, 94, 98
promotion
en C, 13
propagation
et S-boxes, 148
liste, 120
proper interval, 72
précision
d'un format de nombres flottants (ϵ), 7
prénexe
forme, 64
PUGET, vii, 1

QEPCAD, 173
quad
format, 3
quantificateur
élimination, 64
quantité
indéfinie, voir NaN
quasi-primitive
contrainte, 98
quasi-zéro, 45

racine carrée
d'un nombre négatif, 6
et arrondi, 161
raffinement
modèle de ~, 132
RAJAN, 70
RAPHSON, xi, 17, 22
rapport
K-C-S, 5
RATSCHEK, 22
RATZ, 17
redondante
contrainte, 94
REF-ARF, 89
registre
chargement, 110
de DecLIC, 121
du débogueur avec S-boxes, 147
relation
associée à une contrainte, 37
remarquable
point, 64
restriction
du cadre de travail, 86
RevIncNar
algorithme, 140
RODA, 63
ROKNE, 22
RUSSO, 157
RUTISHAUSER, 6
RVInterval, 155, 165, 167
règle
de chaînage, 108
région
$\mathcal{P}$-invariante, 64
répartition des nombres flottants, 7

rétroaction, 60

S-box, **138**
    opérateur de contraction d'une ~, 138
SAM-HAROUD, 80
SANNELLA, 89
SCHICHO, 60, 63, 66, 67, 70, 72
School Problem$_{3,1}$, 171
School Problem$_{3,2}$, 171
SCHULTE, 191
Scud
    missile, 3
secteur
    (f_1, f_2)-~, 203
section
    f-~, 203
SEIDEL, xi
semi-depth-first
    stratégie de découpage, 174
semi-groupe, 17
SHARY, 59, 61, 67–69, 74, 181
signature, 37, 157
signe
    d'un intervalle, 73
    d'une borne, 73
Simple Circle, 171
simples
    occurrences, voir occurrences
simplexe
    algorithme, 94
single
    format, 5
        extended, 5
`single`
    format, 9
SkyBlue, 129, 132
SNYDER, 77
some
    interprétation ~, 59
sommation
    algorithmes de ~, 3
sous-distributivité, 25
splitting, 170, 175
STAHL, 20, 22, 70
Standard Template Library, 155
STOLFI, 15
store, 135
    et décomposition, 44
structuration
    du *store*, 136
SUN Microsystems, 160
sur-optimisation, 13
sémantique
    approchée d'un ECSP, 38
    déclarative d'un ECSP, 38

table
    Table maker's dilemma, 10
tas, voir pile globale
TAYLOR
    forme, voir forme
TAYLOR, 20–22
Taylor, 94
Tcl/Tk, 129, 146, 150
Tcl/Tk, 146
templated class, 156
temporaire
    variable, 108
terme
    représentation, 110
thrashing, 33
tiny
    format, 6
TISSERAND, 11
TMD
    Table maker's dilemma, 10
tolerable solution set, 67
tolérable
    ensemble, 67
trail, 107, 122, 148
trait, 157
TREFETHEN, Lloyd Nicholas, 3
TREFETHEN, 3
trigonométrique
    décomposition ~ cylindrique, 66
    polynôme, 66
trivaluée
    logique, 80
trust rounding, 162
typage
    des variables, 93
tête de clause, 108

ULLRICH, 74
ulp
    unit in the last place, 7, 160
unifié
    ensemble, 67
Union$_\square(\rho)$, 26
Union$_\circ(\rho)$, 26
Union(), 18
union-consistance, 40
united solution set, 67

VAN EMDEN, 41
VAN HENTENRYCK, vii, 1
VAN EMDEN, 40
variable
    dangereuse, 111
    fd-~, 98
    i-~, 98
    p-~, 98
    représentation, 115
    temporaire, 108

VELLINO, 38, 93
VerGO, 155
VIGNES, Jean, 3
VIGNES, 3

WALLACE, 180
WALSTER, 17, 24, 158, 179
WALTZ
    algorithme de ~, 33
WALTZ, xi, 33
WAM, 97
    architecture de la ~, 105
    extension pour DecLIC, 126
    gestion mémoire, 107
wamcc, 106, 107, 115
WARD, 59
WARREN, 97, 105, 106, 112
Wilkinson
    problème, 104
WILKINSON, 3, 17
WOLFF VON GUDENBERG, 165, 167
WOpAC, 151, 153, 169, 170, 172–174

YOSHIGAHARA, 89, 90, 193
YOSHIGAHARA
    problème de ~, 89
YOUNG, 15

Z
    machine ~, 6
ZIV, 11
zone de code, 106
ZUSE, 6
ZYUZIN, 74

Annexes

ANNEXE A

Annexe à la partie II

Théorème A.1 ([107]). *Soient ρ une relation réelle n-aire et $\boldsymbol{B} = I_1 \times \cdots \times I_n$ un pavé d'intervalles. Pour tout $k \in \{1, \ldots, n\}$, on a :*

$$\pi_k(\rho \cap \boldsymbol{B}) = I_k \cap \rho^{(k)}(\boldsymbol{B})$$

Démonstration. Considérons le cas $k = 1$:

$$\begin{aligned}
\pi_1(\rho \cap \boldsymbol{B}) &= \pi_1(\{(\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \mathbb{R}^n \mid (\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \rho \wedge \mathsf{r}_1 \in I_1 \wedge \cdots \wedge \mathsf{r}_n \in I_n\}) \\
&= \{\mathsf{r}_1 \in \mathbb{R} \mid \exists \mathsf{r}_2, \ldots, \exists \mathsf{r}_n, (\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \rho \wedge \mathsf{r}_1 \in I_1 \wedge \cdots \wedge \mathsf{r}_n \in I_n\} \\
&= I_1 \cap \{\mathsf{r}_1 \in \mathbb{R} \mid \exists \mathsf{r}_2, \ldots, \exists \mathsf{r}_n, (\mathsf{r}_1, \ldots, \mathsf{r}_n) \in \rho \wedge \mathsf{r}_2 \in I_2 \wedge \cdots \wedge \mathsf{r}_n \in I_n\} \\
&= I_1 \cap \rho^{(1)}(\boldsymbol{B})
\end{aligned}$$

La preuve est identique pour $k = 2, \ldots, n$. ■

ANNEXE **B**

Annexe à la partie III

Définition B.1 (Ensemble semi-algébrique). Un ensemble $\mathcal{S} \subseteq \mathbb{R}^n$ est dit *semi-algébrique* s'il est constructible par un ensemble finis d'unions, d'intersections et/ou de complémentations d'ensembles $\mathcal{S}_i$ de la forme :

$$\mathcal{S}_i = \{\mathbf{r} \in \mathbb{R}^n \mid f(\mathbf{r}) \geqslant 0\}$$

avec f un polynôme à coefficients réels ($f \in \mathbb{R}[x_1, \ldots, x_n]$).

Définition B.2 (Décomposition de $\mathbb{R}^n$). Étant donné un sous-ensemble $\rho \subseteq \mathbb{R}^n$, on appellera *décomposition (ou partition) de* ρ tout ensemble fini $\{\gamma_1, \ldots, \gamma_p\}$ de p connexes de $\mathbb{R}^n$ disjoints deux à deux, dont l'union est égale à ρ :

$$\rho = \bigcup_{i=1}^{p} \gamma_i, \text{ avec } \forall i, j \in \{1, \ldots, p\}, i \neq j \colon \gamma_i \cap \gamma_j = \varnothing$$

Définition B.3 (Pile au-dessus d'un connexe de $\mathbb{R}^n$). Étant donnés un connexe γ de $\mathbb{R}^n$ et deux fonction continues sur γ, $f_1, f_2 \colon \mathbb{R}^n \to \mathbb{R}^n$, on appelle f_1-*section* l'ensemble $\{(\mathbf{r}, f_1(\mathbf{r})) \mid \mathbf{r} \in \gamma\}$ et (f_1, f_2)-*secteur* le connexe de $\mathbb{R}^n$ compris entre la f_1-section et la f_2-section ($\{(\mathbf{r}, \mathbf{s}) \mid \mathbf{r} \in \gamma, f_1(\mathbf{r}) < \mathbf{s} < f_2(\mathbf{r})\}$). Une *pile au-dessus de* γ est une décomposition de $\gamma \times \mathbb{R}$ constituée de f_i-sections et de (f_i, f_{i+1})-secteurs.

Définition B.4 (Décomposition algébrique). Une décomposition $\Gamma = \{\gamma_1, \ldots, \gamma_p\}$ de $\mathbb{R}^n$ est dite *algébrique* si, pour tout $i \in \{1, \ldots, p\}$, γ_i est un ensemble semi-algébrique.

Définition B.5 (Décomposition cylindrique). Une décomposition $\Gamma = \{\gamma_1, \ldots, \gamma_p\}$ de $\mathbb{R}^n$ est dite *cylindrique* si :

Cas $n = 1$. Γ est une partition de $\mathbb{R}$ en un ensemble fini de points et des intervalles (finis ou infinis) bornés par ces points ;

Cas $n > 1$. Il existe une décomposition cylindrique $\Gamma' = \{\gamma'_1, \ldots, \gamma'_m\}$ de $\mathbb{R}^{n-1}$ et une pile σ_i au-dessus de chaque γ'_i, avec un indice $j \in \{1, \ldots, p\}$ tel que $\sigma_i = \gamma_j$.

Définition B.6 (Délinéabilité d'un ensemble de polynômes). Soit $\mathcal{P} = \{p_1, \ldots, p_r\} \subset \mathbb{R}[x_1, \ldots, x_{n-1}][x_n]$ un ensemble de polynômes et soit ρ un connexe de $\mathbb{R}^{n-1}$. Étant donné $\mathbf{r} = (\mathsf{r}_1, \ldots, \mathsf{r}_{n-1}) \in \mathbb{R}^{n-1}$, notons $p_{i,\mathbf{r}}(x_n)$ le polynôme à une variable $p_i(\mathsf{r}_1, \ldots, \mathsf{r}_{n-1}, x_n)$. On dira que $\mathcal{P}$ est *délinéable sur* ρ (ou « ρ est un connexe $\mathcal{P}$-délinéable ») si les trois conditions suivantes sont satisfaites :

1. pour tout i, $1 \leqslant i \leqslant r$, le nombre total de racines complexes de $p_{i,\mathbf{r}}$ (en comptant la multiplicité de chacune) est constant quand $\mathbf{r}$ parcourt ρ ;
2. pour tout i, $1 \leqslant i \leqslant r$, le nombre de racines complexes distinctes de $p_{i,\mathbf{r}}$ est constant quand $\mathbf{r}$ parcourt ρ ;
3. pour tout i, j, $1 \leqslant i < j \leqslant r$, le nombre total de racines complexes communes à $p_{i,\mathbf{r}}$ et à $p_{j,\mathbf{r}}$ est constant quand $\mathbf{r}$ parcourt ρ ;

On peut montrer (*cf.* [121] pour une référence à la preuve) que si l'ensemble $\mathcal{P}$ est délinéable sur ρ, le nombre total de racines réelles distinctes de $\mathcal{P}$ est invariant sur ρ. De plus, ρ est alors un ensemble semi-algébrique.

Langages et environnements en programmation par contraintes d'intervalles

Frédéric GOUALARD

Résumé

La programmation par contraintes d'intervalles est une approche prometteuse pour la résolution de systèmes de contraintes réelles non-linéaires : l'emploi de l'arithmétique d'intervalles garantit la complétude des résultats et l'efficacité des méthodes de filtrage par le calcul de consistances partielles est identique ou (très) supérieure à celle d'algorithmes spécialisés.
Dans cette thèse, nous nous intéressons à l'extension des capacités de la programmation par contraintes d'intervalles suivant trois axes :

1. *Accélération du processus de calcul :* les librairies d'intervalles utilisées par les algorithmes de résolution doivent être à la fois correctes et efficaces. Nous décrivons une librairie C++ paramétrée de calcul sur les intervalles basée sur la notion de *trait* et montrons que la flexibilité obtenue par la paramétrisation du type des bornes n'induit pas de surcoût à l'exécution et autorise une plus grande fiabilité pour la portabilité de la librairie. Nous présentons aussi une extension de la définition de box-consistance autorisant la mise au point d'un algorithme pour son calcul dont l'efficacité est toujours au moins égale à celle des algorithmes utilisés habituellement pour calculer la box-consistance ou la hull-consistance ;
2. *Extension du domaine d'application :* l'emploi des intervalles garantit la complétude des résultats mais pas leur correction alors que cette propriété est cruciale pour certaines applications. Nous décrivons des algorithmes de calcul d'approximations intérieures de relations réelles assurant cette correction ; nous introduisons ensuite des algorithmes autorisant la résolution de systèmes avec des variables quantifiées universellement ;
3. *Définition d'un environnement de programmation adapté :* nous présentons une méthode d'abstraction du store de contraintes permettant sa visualisation à des fins de débogage/optimisation/explication de programmes contenant des contraintes, puis décrivons un outil basé sur cette technique.

Mots-clés : programmation par contraintes, variable quantifiée, positionnement de caméra, débogage, arithmétique d'intervalles, consistance locale

Abstract

Interval constraint programming is a promising approach for solving non-linear real constraint systems: interval arithmetics guarantees the completeness of the results while the efficiency of the filtering methods by partial consistency computation is identical or (very) superior to the one of tailored algorithms.
In this thesis, we focus on the extension of interval constraint programming capabilities through three axis:

1. *Acceleration of the computation process:* interval libraries used by the solving algorithms need be both correct and efficient. We describe a parameterized C++ library based on the traits notion, and we show that the flexibility achieved through parameterization of the type of the bounds does not induce any additional cost on execution, while providing enhanced reliability as for the library portability. We also present an extension of the box consistency definition that allows us devising an algorithm to compute it whose efficiency is always at least equal to the one of the algorithms usually used to compute box consistency or hull consistency ;
2. *Extension of the applicability domain:* the use of intervals ensures completeness but not the correctness of the results, while this last property is crucial for some applications. We describe algorithms to compute inner approximations of real relations ensuring correctness ; we then introduce algorithms that permit solving systems with universally quantified variables ;
3. *Definition of a suitable programming environment:* we present a method for providing an abstraction of the constraint store that allows visualizing it for debugging/optimizing/explaining purposes for constraint programs ; we then describe a tool based on this technique.

Keywords: constraint programming, quantified variable, camera control, debugging, interval arithmetic, local consistency

Classification ACM

Catégories et descripteurs de sujets : D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic*; G.1.0 [**Numerical Analysis**]: General—*Interval arithmetic*; G.1.5 [**Numerical Analysis**]: Roots of Nonlinear Equations—*systems of equations*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems—*Animations*

Termes généraux : Algorithms, Theory, Reliability