

Etude de la compilation des langages logiques de programmation par contraintes sur les domaines finis: le système clp(FD)

Daniel Diaz

▶ **To cite this version:**

Daniel Diaz. Etude de la compilation des langages logiques de programmation par contraintes sur les domaines finis: le système clp(FD). Informatique et langage [cs.CL]. Université d'Orléans, 1995. Français. NNT : . tel-00981853

HAL Id: tel-00981853
https://theses.hal.science/tel-00981853

Submitted on 23 Apr 2014

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

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

THESE

présentée à

L'UNIVERSITE D'ORLEANS

pour obtenir le grade de

DOCTEUR EN INFORMATIQUE

par

Daniel Diaz

ETUDE DE LA COMPILATION DES LANGAGES LOGIQUES DE PROGRAMMATION PAR CONTRAINTES SUR LES DOMAINES FINIS : LE SYSTEME `clp(FD)`

Soutenue le 13 Janvier 1995 devant la Commission d'examen

G. Filé	.....	Président
J.P. Delahaye	.....	Rapporteur
P. van Hentenryck	.....	Rapporteur
D.H.D. Warren	.....	Rapporteur
F. Benhamou	.....	Examinateur
P. Codognet	.....	Examinateur
P. Deransart	.....	Examinateur
G. Ferrand	.....	Examinateur

Table des matières

1 Introduction **1**

1.1 Les domaines finis 2

1.2 Approche RISC versus approche CISC 4

1.3 Opération *Ask* et contraintes conditionnelles 6

1.4 Contraintes booléennes 7

1.5 Le système `clp(FD)` 8

2 La programmation logique par contraintes **10**

2.1 Les systèmes de contraintes 10

2.2 Le cadre PLC($\mathcal{C}$) - syntaxe et sémantique 12

2.2.1 Syntaxe des programmes logiques avec contraintes 12

2.2.2 Sémantique des programmes logiques avec contraintes 13

2.3 Le cas $\mathcal{C}$ =`FD` 14

2.3.1 Les domaines finis 14

2.3.2 La contrainte X `in` r - syntaxe et intuition 15

2.3.3 Sémantique de X `in` r et opération *Tell* 19

2.3.4 La relation de satisfaction 22

3 Implantation de `wamcc` 24

3.1 La machine abstraite de Warren 24

3.1.1 La pile locale (ou de contrôle) 25

3.1.2 La pile de restauration (ou trail) 28

3.1.3 La pile globale (ou heap) 29

3.1.4 La représentation des termes 30

3.1.5 Registres 32

3.1.6 Economie et récupération mémoire 33

3.1.7 Le jeu d'instructions 38

3.2 Exécution du code WAM : traduction vers C 48

3.2.1 Cahier des charges 48

3.2.2 Méthodes classiques pour exécuter la WAM 49

3.2.3 La solution adoptée : traduire Prolog vers C 50

3.2.4 Le problème du contrôle de Prolog en C 51

3.2.5 La méthode Janus 52

3.2.6 La méthode KL1 53

3.2.7 La méthode Erlang 54

3.2.8 La méthode `wamcc` 56

3.3 Caractéristiques de `wamcc` 61

3.3.1 Processus de compilation 61

3.3.2 Gestion des piles 62

3.3.3 Fichiers de configuration 63

3.3.4 Gestion de la modularité 65

3.4 Evaluation de `wamcc` 66
3.4.1 Les fonctionnalités du système `wamcc` 67
3.4.2 Le jeu de benchmarks 68
3.4.3 `wamcc` versus des Prolog universitaires 69
3.4.4 `wamcc` versus des Prolog professionnels 70

4 Implantation de `clp(FD)` 74

4.1 Extension de la WAM 74
4.1.1 Intégration des variables domaine 74
4.1.2 Nouvelles structures de données 77
4.1.3 Jeu d'instructions 83
4.1.4 Opération *Tell* 89
4.2 Intégration de `clp(FD)` dans `wamcc` 90
4.3 Evaluation de `clp(FD)` 93
4.3.1 Le jeu de benchmarks 93
4.3.2 Evaluation de l'implantation de base 94
4.3.3 Optimisations 95
4.3.4 Evaluation de l'implantation finale 103

5 Contraintes booléennes 107

5.1 Un panorama des résolveurs booléens 108
5.1.1 Méthodes basées sur la résolution 108
5.1.2 Méthodes basées sur les diagrammes de décisions binaires (BDD) . 109
5.1.3 Méthodes énumératives 110
5.1.4 Programmation en nombres entiers 0-1 111

5.1.5 Méthodes basées sur la propagation. 112
5.1.6 PLC versus résolveurs dédiés 112
5.2 Contraintes booléennes 113
5.3 Codage des booléens en `clp(FD)` : `clp(B/FD)` 115
5.4 Evaluation des performances de `clp(B/FD)` 116
5.4.1 Le jeu de benchmarks 116
5.4.2 `clp(B/FD)` et CHIP 117
5.4.3 `clp(B/FD)` et les autres résolveurs 118
5.5 `clp(B)` un résolveur dédié pour les booléens 121
5.5.1 La contrainte primitive l_0 `<=` $l_1,\ldots,l_n$ 122
5.5.2 Définition des contraintes booléennes 124
5.5.3 Extension de la WAM 124
5.5.4 La procédure de consistance 129
5.5.5 Evaluation de `clp(B)` 132
5.6 Conclusion 132

6 Détection de la satisfaction de contraintes 134

6.1 Introduction 134
6.2 Approximation 1 : test à la clôture 137
6.3 Approximation 2 : test sur les domaines 139
6.4 Approximation 3 : test sur les intervalles 142
6.4.1 Equivalence des conditions suffisantes 145

7 Contraintes complexes 149

7.1 Contraintes arithmétiques linéaires 149

7.1.1 Normalisation 149
7.1.2 Compilation en code *inline* 150
7.1.3 Compilation en appel de sous-contraintes de librairie 151
7.2 Opération *Ask* 152
7.2.1 Le problème des séries magiques 154
7.2.2 Contrainte `atmost` 155
7.2.3 Contrainte de cardinalité 155
7.2.4 Contrainte `element` 156
7.2.5 Contraintes arithmétiques non-linéaires 156
7.3 Généralisation de la contrainte X `in` r 157
7.3.1 Contraintes résolues par *full lookahead* 157
7.3.2 Fonctions utilisateurs 158
7.4 Disjonction constructive 162
7.4.1 Un exemple simple 163
7.4.2 "L'union fait la force" 165
7.4.3 Autres exemples 166
7.5 Contraintes définies par des relations 168

8 Régulation du trafic aérien avec `clp(FD)` 170

8.1 Introduction 171
8.2 Problem Context 173
8.2.1 Air Traffic Flow Management Overview 173
8.2.2 The Slot Allocation Problem 174
8.3 `clp(FD)` in a Nutshell 175

8.3.1 The Constraint X `in` r . . . 176
8.3.2 High-Level Constraints and Propagation Mechanism . . . 177
8.3.3 Optimizations . . . 178
8.3.4 Performances . . . 179
8.3.5 `atmos_interval` Constraint . . . 179
8.4 Slot Allocation Satisfying Capacity Constraints . . . 180
8.4.1 A Small Example . . . 180
8.4.2 `clp(FD)` Model . . . 180
8.4.3 `clp(FD)` Implementation of our Small Example . . . 181
8.4.4 Optimization Trials - Heuristics . . . 182
8.4.5 Results . . . 183
8.4.6 Extension of the Model to Integrate Flow Rate Constraints . . . 185
8.4.7 A Simulation Aid Tool for Regulators - Cost estimation of Regulation Plans . . . 187
8.5 Conclusion and Further Works . . . 187

9 Conclusion **188**

A Programme `crypta` **191**

B Programme `eq10` **193**

C Programme `eq20` **195**

D Programme `alpha` **198**

E Programme `queens` **200**

F Programme `five` **202**

G Programme `cars` **204**

H Programme `bridge` **209**

I Manuel d'utilisation de `wamcc` **216**

I.1 Using `wamcc` - Modularity 217

I.2 From Prolog modules to Unix Executables 219

I.2.1 Compiling Prolog Modules 219

I.2.2 Generating Object Files 221

I.2.3 Linking Object Files 221

I.2.4 Stack Overflow Messages 222

I.2.5 Makefile Generator - `bmf_wamcc` Utility 222

I.3 Built-in Predicates 223

I.3.1 Input / Output 223

I.3.2 Arithmetic 228

I.3.3 Term Management 230

I.3.4 Test Predicates 234

I.3.5 Control 235

I.3.6 List Processing 237

I.3.7 Operators 238

I.3.8 Modification of the Program 239

I.3.9 All Solutions 240

I.3.10 Global Variables 242

I.3.11 Miscellaneous 247

I.4 Debugger 249

J Manuel d'utilisation de `clp(FD)` 254

J.1 Introduction 255

J.2 Finite Domain variables 256

J.3 Finite Domain built-in predicates / constraints 257

J.3.1 The constraint X `in` r 257

J.3.2 Linear arithmetic constraints 258

J.3.3 Other arithmetic constraints 259

J.3.4 Domain Handling 260

J.3.5 Enumeration predicates 261

J.3.6 Symbolic constraints 261

J.3.7 Symbolic constraints 262

J.4 Boolean built-in predicates / constraints 262

J.4.1 Basic boolean constraints 262

J.4.2 Symbolic boolean constraints 263

Table des tableaux

1 syntaxe de la contrainte X `in` r 16

2 sémantique dénotationnelle de l'opération $Tell$ 20

3 exemple de code WAM 48

4 fichier de description de la WAM 64

5 performances de `wamcc` (temps en sec.) 68

6 `wamcc` versus autres Prolog universitaires (temps en sec.) 69

7 `wamcc` versus Prolog professionnels (temps en sec.) 71

8 fragment du code généré pour `'x=y+c'` 88

9 fragment de code C généré pour `'x=y+c'` 92

10 fragment de code assembleur Sparc généré pour `'x=y+c'` 92

11 version de base de `clp(FD)` versus CHIP (temps en sec.) 94

12 décomposition des $Tell$s dans la version de base 95

13 gain de la file optimisée 98

14 décomposition des $Tell$s avec une file optimisée 98

15 gain de l'optimisation 2 100

16 décomposition des $Tell$s dans la version finale 104

17 gain de la version finale 104

18 détail du gain final 104

19 `clp(FD)` versus CHIP (temps en sec.) 106

20 `clp(FD)` versus compilateur CHIP (temps en sec.) 106

21 théorie de propagation booléenne B 114

22 définition du résolveur booléen de `clp(B/FD)` 116

23 `clp(B/FD)` versus CHIP (temps en sec.) 118

24 `clp(B/FD)` versus un BDD (temps en sec.) 119

25 `clp(B/FD)` versus une méthode énumérative (temps en sec.) 120

26 `clp(B/FD)` versus la consistance locale booléenne (temps en sec.) 121

27 `clp(B/FD)` versus une méthode de R.O. (temps en sec.) 122

28 syntaxe de la contrainte l_0 `<=` $l_1,\ldots,l_n$ 123

29 définition du résolveur booléen de `clp(B)` 124

30 code généré pour `and(X,Y,Z)` 130

31 `clp(B)` versus les autres résolveurs 133

32 définition de $inf(t)$ et $sup(t)$ 139

33 définition de $A(r)$ et $M(r)$ 140

34 nouvelle définition de $A(r)$ et $M(r)$ 143

35 problème des séries magiques 155

36 syntaxe étendue de la contrainte X `in` r 159

37 code de `X in -{val(Y)} & -{val(Y)-I} & -{val(Y)+I}` 161

38 `queens` optimisé avec fonctions utilisateurs 162

39 fragment of the constraint system syntax 176

40 implementation of our small problem with `clp(FD)` constraints 182

41 some runtime examples 185

Table des figures

1 exemple d'arbre de recherche standard 26

2 structures de la trail 29

3 représentation des termes dans la WAM 31

4 architecture de la WAM 33

5 processus de compilation 61

6 disposition des piles en mémoire 62

7 représentation interne d'un environnement 77

8 représentation interne d'une contrainte 78

9 représentations internes d'un domaine 80

10 représentation interne d'une variable DF 82

11 structures de données pour les contraintes 82

12 nouvelle représentation interne d'une variable DF 97

13 représentation interne définitive d'une variable DF 100

14 impact des optimisations 105

15 BDD représentant la formule $(x \wedge y) \vee z$ 110

16 représentation interne d'une variable booléenne 126

17 représentation interne d'une contrainte dans les listes de dépendances . . . 127

18 structures de données nécessaires pour la contrainte `Z <= [-X,Y]` 128

19 proportion de chaque issue de la procédure de consistance 131

20 Display of SPORT system 172

21 Graphical representation of a small problem 180

22 UM traffic before CLP process 184

23 UM traffic after CLP process 184

Chapitre 1

Introduction

La Programmation Logique avec Contraintes (PLC) est un domaine de recherches très actif depuis quelques années et des langages comme CHIP, PLC($\mathcal{R}$) ou PrologIII ont prouvé que cette approche ouvrait la Programmation Logique (PL) classique à un vaste champ d'applications. L'idée de base de la PLC est de remplacer le mécanisme d'unification de la PL par la résolution de contraintes sur des domaines particuliers, en considérant le résolveur de contraintes comme une boîte noire qui se charge de tester la satisfiabilité des contraintes et, possiblement, de les réduire à une forme normale. Bien que cette dichotomie soit très importante du point de vue théorique, et qu'elle ait permis par exemple d'importer de nombreux théorèmes de la sémantique de la PL vers la PLC, elle n'est pas très satisfaisante du point de vue pratique. En effet la résolution de contraintes et l'interface entre le moteur logique et le résolveur restent ainsi entourés d'un mystère nébuleux, ce qui n'aide pas à une vision claire de l'architecture d'un système de PLC. Il est d'ailleurs à noter qu'il y a un curieux manque de littérature sur l'aspect pratique de la PLC...

On peut considérer qu'une des avancées majeures de la Programmation Logique dans les années 80 a été la définition de la machine abstraite de Warren (WAM), qui est devenue un standard de facto (une alternative à l'ISO) pour la compilation de Prolog. La WAM a aidé de nombreux chercheurs à acquérir une meilleure compréhension de l'exécution de Prolog et à développer des systèmes de PL efficaces. De plus, la WAM s'est avérée suffisamment flexible pour servir de base à diverses extensions de la PL telles que les contraintes, la concurrence, le parallélisme, l'ordre supérieur, etc. Pour revenir à la PLC, on ne peut que déplorer le fait que l'approche boîte noire ne donne que peut d'informations

sur l'architecture d'un système de PLC et ne se prête pas à la définition d'une machine abstraite pour la compilation et le traitement des contraintes. Un problème supplémentaire ici est le fait qu'il doit en réalité y avoir autant de machines abstraites qu'il y a de résolveurs, c'est-à-dire de domaines de contraintes.

Dans cette thèse nous nous sommes donc concentrés sur la définition et l'implantation d'un système de PLC sur les domaines finis. L'idée maîtresse qui a dirigé ce travail était la volonté de définir une architecture simple, claire et minimale. Le problème majeur était donc de déterminer quels étaient les éléments minimaux qu'il fallait utiliser pour pouvoir ensuite reconstruire un système complet, avec comme contraintes supplémentaires l'extensibilité et l'efficacité. Il s'agissait ensuite de concevoir une architecture concrète et une machine abstraite pour le traitement des contraintes sur les domaines finis.

1.1 Les domaines finis

Parmi les différents domaines de calcul étudiés en PLC, celui des domaines finis semble être le plus prometteur, car il est très utile dans de nombreuses applications industrielles comme par exemple les problèmes combinatoires, l'ordonnancement, les optimisations de stocks, la simulation de circuits, le diagnostic, l'aide à la décision ou même les problèmes booléens.

Les domaines finis ont été introduits en PLC par Pascal van Hentenryck dans le langage CHIP vers la fin des années 80. Un domaine fini est tout simplement un ensemble de valeurs, numériques ou symboliques, de cardinalité finie, comme par exemple {1,2,5,12}, 1..45, ou {*rouge,vert,bleu*}. Les contraintes proposées sont aussi bien des contraintes *arithmétiques* telles que des équations, inéquations ou diséquations entre des termes linéaires[1] que des contraintes *symboliques* qui ne relèvent pas des opérations mathématiques habituelles, comme par exemple la relation `atmost(N,[`X_1`,...,`X_m`],V)` qui signifie qu'au plus N variables X_i sont égales à l'entier V. Les algorithmes de résolution de contraintes utilisés dérivent des techniques de consistance des "Constraint Satisfaction Problems" (CSP) introduits en Intelligence Artificielle pour la reconnaissance des formes il y a une vingtaine d'années, cf. les travaux de Montanari et de Waltz. L'idée de base est de construire et de

[1] bien que cette restriction à la linéarité puisse facilement être levée.

mettre à jour un réseau de contraintes entre des variables (en nombre fini) pouvant prendre une valeur dans un domaine fini. On a donc un graphe dont les noeuds sont les variables et les arcs les contraintes. La satisfiabilité de l'ensemble de contraintes est assurée en propageant de proche en proche (propagation dite locale) les valeurs possibles des variables à travers les contraintes qui les lient entre elles. On utilise en général, pour des raisons d'efficacité, une technique appelée *consistance d'arcs* qui propage à travers le réseau uniquement les contraintes unaires (domaines des variables), plutôt que la *consistance de chemin* ou la technique plus générale de *k-consistance* qui propage des relations concernant k variables. L'efficacité de la consistance d'arc, ainsi que de certaines extensions/simplifications dont il sera question dans cette thèse, a été montrée pour le traitement des équations, inéquations et diséquations linéaires par exemple, ainsi que pour de nombreuses applications industrielles traitées avec le langage CHIP, qui intègre de telles techniques.

Il est à noter cependant que la consistance d'arc ne permet pas d'obtenir par elle seule une méthode correcte pour s'assurer de la satisfiabilité d'un ensemble de contraintes. La consistance d'arc peut en effet ignorer certaines valeurs inconsistantes des domaines et répondre qu'un ensemble de contraintes est consistant alors qu'il ne l'est pas. Considérons par exemple 3 variables X, Y et Z dont les domaines sont $\{0,1\}$ et l'ensemble de contraintes $\{X \neq Y, Y \neq Z, Z \neq X\}$. En propageant uniquement les domaines des variables ($\{0,1\}$) à travers le réseau de contraintes, on ne peut déduire que cet ensemble est inconsistant, puisque pour chaque contrainte il y a une instanciation possible des variables satisfaisant cette contrainte. Chaque contrainte est traitée localement (d'où, encore, le nom de propagation locale), et seuls les domaines des variables permettent de transmettre quelque information entre les contraintes. Notons que dans l'exemple précédent il faudrait propager des relations binaires représentant les couples de valeurs possibles pour deux variables liées par une contrainte pour trouver l'insatisfiabilité globale de l'ensemble de contraintes. Cependant un tel schéma peut s'avérer très coûteux et il n'est pas sûr que cela soit payant en général. Il faut donc, en plus de la consistance d'arc qui peut réduire les domaines des variables mais n'est pas suffisante, une phase d'*énumération* qui instanciera incrémentalement les variables non encore déterminées aux valeurs possibles de leurs domaines. Différentes heuristiques peuvent être intégrées lors de cette phase pour tenter de réduire la combinatoire. Notons cependant que l'instanciation d'une variable doit être suivie d'une phase de propagation de cette valeur dans le réseau de contraintes pour possiblement réduire les domaines des autres variables, toujours pour diminuer la combinatoire. Une solution

est trouvée lorsque l'on aura instancié toutes les variables, et si ce n'est pas le cas alors l'insatisfiabilité aura été prouvée.

On voit bien que la phase d'énumération, nécessaire pour s'assurer de la correction du calcul et construire une solution, est complexe et coûteuse. C'est pourquoi les langages de PLC sur les domaines finis l'exécutent uniquement à la fin de l'exécution du programme, juste avant de proposer une solution réponse à l'utilisateur. Pendant toute la durée du calcul, seule la phase de consistance d'arc aura été effectuée pour tester la satisfiabilité de l'ensemble courant de contraintes. Ceci implique que le système a peut-être exploré inutilement des branches d'échec qui n'auront été découvertes comme telles qu'ultimement, mais cela ne remet pas en cause la correction ni la complétude des calculs.

1.2 Approche RISC versus approche CISC

Nous avons suivi jusqu'ici l'approche traditionnelle de la PLC en présentant pour les domaines finis un résolveur de type boîte noire basé sur les techniques de consistance d'arc (propagation locale des domaines des variables). En fait, ceci est très grossier, et il faut en pratique traiter chaque contrainte particulièrement et spécialiser la propagation locale pour chacune d'elle. Le résolveur de CHIP par exemple suit cette approche, et il consiste, du point de vue de l'implantation, en une collection de procédures chargées du traitement particulier de chaque contrainte, écrites en langage C par souci d'efficacité. Ceci ne se fait cependant qu'aux dépens de l'extensibilité et de la clarté, et le programmeur ne sait jamais exactement quel traitement est effectué par le résolveur et en est réduit à espérer que l'implanteur ait bien fait son travail et qu'il n'aura pas besoin de définir de nouvelles contraintes pour une application donnée.

Nous allons proposer une autre approche, appelée métaphoriquement l'approche RISC par analogie avec la technologie des microprocesseurs. La boîte de verre (i.e. transparente) remplace la boîte noire. Cette méthode permet d'obtenir d'excellentes performances et allie donc simplicité et efficacité. Comme pour les microprocesseurs, il vaut mieux avoir un jeu d'instructions simple et limité mais optimisé plutôt qu'un traitement lourd et spécifique de chaque cas particulier.

L'idée de base est de définir des contraintes primitives, simples et en nombre limité, et de

rebâtir grâce à elles les contraintes complexes usuelles, comme par exemple les contraintes arithmétiques (ex. $X = 3 * Y + Z$, $X < Y$ ou $Z \neq 2 * Y - 3$, etc) ou diverses contraintes symboliques (ex. `atmost(N,[`X_1`,...,`X_m`],V)`). On a donc un processus en deux étapes : les contraintes complexes sont traduites lors de la compilation en des ensembles de contraintes primitives, et, lors de l'exécution, le résolveur a uniquement à gérer ces contraintes primitives. Le résolveur est donc ainsi beaucoup plus simple, uniforme et homogène. En outre le niveau des contraintes primitives donne un degré de liberté de plus et peut être considéré comme une sorte de langage de base pour exprimer les mécanismes de propagation et la méthode de résolution choisie pour traiter une contrainte complexe. De plus le résolveur est maintenant ouvert (à l'utilisateur) car de nouvelles contraintes peuvent être ajoutées facilement : il suffit de spécifier leur traduction en contraintes primitives. En outre une machine abstraite pour la résolution des contraintes sur les domaines finis peut alors être plus facilement développée à partir de l'ensemble (réduit) des contraintes primitives.

Quelles sont donc ces contraintes primitives qui permettent de rebâtir à peu de frais toutes les contraintes usuelles sur les domaines finis ? Une idée nouvelle a été proposée il y a quelques années par Pascal van Hentenryck. Il suffit en fait d'une unique contrainte primitive d'appartenance : la contrainte X `in` r où X est une variable domaine fini et r dénote un ensemble d'entiers. La sémantique intuitive d'une telle contrainte est de forcer X à appartenir au domaine dénoté par r (i.e. $X \in r$). r peut être défini comme un intervalle $t_1 .. t_2$ ou à partir d'autres domaines par des opérations d'union, intersection,... La puissance de cette primitive provient du fait que r peut aussi bien dénoter un ensemble constant (ex. `1..10`) qu'un ensemble *dépendant* de la valeur d'autres variables grâce à l'utilisation de *domaines/termes indexicaux* (`min(`Y`)`, `max(`Y`)`, `dom(`Y`)`). Une telle contrainte fournit une valeur dépendant du domaine courant des autres variables et évolue donc au fil des réductions de ces variables. Nous verrons précisément dans cette thèse comment compiler des contraintes de haut niveau en contraintes primitives, comme bien sûr les équations ou inéquations linéaires (et non-linéaires aussi d'ailleurs), mais aussi les contraintes symboliques. Nous verrons également qu'il est possible de définir un certains nombre d'optimisations globales pour le traitement de la contrainte X `in` r pour éviter les propagations inutiles. Du fait de l'architecture RISC, toutes les contraintes de haut niveau bénéficieront de ces optimisations et nous montrerons qu'un tel résolveur est très efficace, meilleur en tout cas que les résolveurs boîtes noires actuels.

En résumé, un résolveur de contraintes suivant l'approche RISC se décompose en deux parties : un pré-processeur traduisant les contraintes complexes (visibles à l'utilisateur) en contraintes primitives, et un résolveur proprement dit traitant uniquement les contraintes primitives.

1.3 Opération *Ask* et contraintes conditionnelles

Abordons maintenant le problème de l'extensibilité des résolveurs de contraintes. La difficulté principale lorsqu'on définit des contraintes complexes, comme par exemple les contraintes symboliques proposées par le langage CHIP (`atmost`, etc), est de spécifier un contrôle sur la propagation de contraintes et en particulier de préciser que certaines contraintes ne doivent être considérées que de manière *conditionnelle*, i.e. sous réserve qu'une certaine condition soit vérifiée. Ceci se rapproche beaucoup des mécanismes de retard que l'on trouve en Prolog, comme par exemple les déclarations `wait` ou `delay` de certains systèmes ou le séminal `freeze` de PrologII. Ces mécanismes de retard sont liés, en programmation logique, à des conditions d'instanciation de certaines variables Prolog ; le mécanisme correspondant nécessaire en PLC considérera de manière plus générale comme conditions des contraintes sur les variables du domaine d'intérêt. Supposons par exemple que l'on veuille définir la contrainte `atmost(N,[`X_1`,...,`X_m`],V)`. Il faut pouvoir exprimer que dès que N variables auront la valeur V, toutes les autres auront des valeurs différentes de V. On veut donc avoir comme conditions des contraintes (non exprimables par les simples tests sur l'instanciation des `delay` ou `freeze`) et en cas de satisfaction ajouter de nouvelles contraintes ($Xj \neq V$). CHIP propose pour cela une construction `if_then_else` à la signification immédiate, mais celle-ci reste limitée puisque seules des équations ou inéquations linéaires comportant deux variables au plus peuvent apparaître en condition. Ceci est dû à l'approche boîte noire qui a pour conséquence la nécessité de définir pour chaque type particulier de contrainte condition un test spécifique.

Heureusement, l'approche RISC apporte là encore une solution intéressante grâce à son modèle simple et surtout homogène. En effet nous verrons comment générer à partir des contraintes primitives des conditions logiques décrivant si une contrainte est impliquée par l'ensemble courant de contraintes ou non. Ces conditions logiques se dérivent aisément de la syntaxe des contraintes X `in` r. Une contrainte conditionnelle peut donc être compilée

en un test logique qu'il sera facile de tester efficacement lors de l'exécution du programme.

Notons que l'idée d'avoir une contrainte condition pour l'exécution d'une autre contrainte ou d'un prédicat logique dérive en fait des travaux sur les *langages concurrents avec contraintes (CC)* où cette construction, appelée *Ask*, sert de mécanisme de synchronisation de base entre processus concurrents. Ce travail ouvre donc naturellement sur l'extension d'un langage de PLC sur les domaines finis vers un langage CC sur les domaines finis.

1.4 Contraintes booléennes

Un autre exemple intéressant pour montrer la flexibilité de l'approche RISC est l'étude des contraintes booléennes : celles-ci (*et*, *ou* et *non*, pour rester simple) sont à valeur dans un domaine fini ({0,1}!) mais sont cependant différentes des contraintes usuelles. Il est donc intéressant de voir s'il est possible d'encoder efficacement ces contraintes en contraintes primitives X `in` r et de comparer le résolveur booléen ainsi obtenu avec les autres résolveurs existants utilisant des méthodes et algorithmes complètement différents. La résolution des contraintes booléennes est un problème déjà ancien mais qui nourrit des recherches toujours très actives. De nombreuses méthodes ont été développées, soit générales, soit pour des types particuliers de formules. Il y a quelques années l'utilisation de techniques de propagation locale a été proposée par le langage CHIP, qui en fait dispose de deux résolveurs booléens : l'un basé sur l'unification booléenne, et l'autre utilisant la propagation locale et réutilisant certaines procédures du résolveur sur les domaines finis. Il s'avère en fait que le résolveur utilisant la propagation est bien plus efficace que l'autre, à tel point que dans CHIP il est le résolveur par défaut pour les booléens.

Il est en fait très facile de définir les opération booléennes de base (*et*, *ou* et *non*) en termes de contraintes primitives. Le résolveur booléen est réduit à moins de 10 lignes de code ! C'est-à-dire à 3 définitions de contraintes en termes de X `in` r. Notons en outre que cet encodage se fait à un niveau plus bas qu'une simple transformation des contraintes booléennes en expressions arithmétiques par exemple et qu'on peut ainsi espérer une plus grande efficacité. En outre, ce résolveur est ouvert à l'utilisateur qui peut ajouter de nouvelles contraintes, pour intégrer par exemple directement des contraintes d'implication, d'équivalence, etc.

Ceci est cependant assez évident, du fait que les booléens sont un cas particulier de domaines finis et peut sembler un simple exercice d'école. Le plus surprenant cependant est que le résolveur ainsi réalisé soit très efficace : il est plus rapide que le résolveur de CHIP d'environ un ordre de magnitude, et se compare favorablement à la plupart des résolveurs booléens ad hoc basés sur d'autres algorithmes, comme les méthodes énumératives, les BDD ou les techniques de recherche opérationnelle. Notons finalement que ces résolveurs utilisent en général de nombreuses heuristiques pour améliorer les performances, alors que notre résolveur n'en a pour l'instant aucune, et que l'on peut donc encore espérer une amélioration des performances.

1.5 Le système `clp(FD)`

L'approche RISC des domaines finis peut être implantée très efficacement dans un moteur logique classique basé sur la WAM, la "Warren Abstract Machine", qui est l'architecture standard des compilateurs Prolog et de bon nombre de systèmes logiques étendus. Un tel système est implanté dans le langage `clp(FD)` décrit dans cette thèse.

Il a cependant d'abord fallu, avant de réaliser `clp(FD)`, implanter un système Prolog classique pour servir de support au système de programmation par contraintes. Celui-ci est appelé `wamcc` car il traduit Prolog vers C via la WAM. Il ne s'agit pas simplement de développer un système de programmation en logique de plus, mais d'étudier comment définir de la manière la plus simple et minimale possible un système répondant aux demandes suivantes :

- *extensibilité* : le système doit être une plate-forme expérimentale, modifiable et extensible. Ceci implique que son architecture soit simple, sans les optimisations complexes courantes dans les systèmes Prolog commerciaux qui pourraient aller à l'encontre de l'intégration de diverses extensions, voire la rendre complètement impossible.
- *portabilité* : pour espérer une large diffusion et éviter l'obsolescence liée à l'attachement à une machine particulière.
- *efficacité* : pour répondre au besoin des utilisateurs, que ce soit pour la partie Prolog ou pour le traitement des contraintes.

- *modularité* : pour pouvoir décomposer une application en différents modules, et ainsi pouvoir traiter des applications de taille importante.

Le système ainsi développé, `wamcc`, s'avère à la fois simple d'architecture (donc extensible facilement, à l'opposé d'un système comme Quintus ou Sicstus Prolog par exemple) et efficace : ses performances sont équivalentes à celles de Quintus Prolog 2.5, qui est basé sur un émulateur de code WAM optimisé écrit en assembleur. Notons que Quintus Prolog est un produit commercial ayant nécessité plusieurs années de développement.

L'idée de base de l'architecture de `clp(FD)` pour le traitement des contraintes est d'avoir une structure des données simple et homogène pour les domaines de variables, les contraintes et les environnements. Il s'agit donc d'étendre l'architecture de `wamcc`, et en particulier la WAM, de manière minimale. Ceci est rendu possible par l'approche RISC, grâce au traitement d'un seul type de contrainte : la contrainte indexicale X `in` r. Ceci permettra un traitement uniforme des contraintes et un certain nombre d'optimisations.

L'une des particularité du langage `clp(FD)` est de ne pas seulement proposer un jeu d'instructions pour compiler des contraintes primitives vers un langage de bas niveau (une machine abstraite pour les contraintes sur les domaines finis) qui serait émulé, mais de compiler ensuite ce langage intermédiaire vers le langage C, considéré ainsi comme un assembleur générique. On suit en cela naturellement l'approche de `wamcc` qui compile Prolog vers C. Ceci permet d'optimiser bon nombre d'opérations (numériques par exemple), et d'utiliser les optimisations de très bas niveau des compilateurs C.

`clp(FD)` est ainsi un langage très efficace : la partie Prolog (car la PLC contient Prolog !) est aussi efficace que Quintus Prolog, et, pour ce qui est du traitement des contraintes sur les domaines finis, ce langage est nettement plus rapide que CHIP, de 2 à 10 fois plus rapide, voire plus, selon les exemples testés. Ainsi l'approche RISC prouve qu'elle permet de gagner à la fois en simplicité et en efficacité.

`clp(FD)` semble donc réaliser notre objectif initial : concevoir un système de programmation logique avec contraintes sur les domaines finis extensible, simple, efficace, et modulaire.

Chapitre 2

La programmation logique par contraintes

2.1 Les systèmes de contraintes

La manière la plus simple de définir une contrainte est de la voir comme une formule logique atomique (*relation*) qui est interprétée dans une structure particulière et non pas dans une interprétation de Herbrand comme le sont les prédicats logiques classiques. C'est d'ailleurs la définition la plus usitée et traditionnelle en PLC, originellement proposée par Jaffar et Lassez [44]. Cependant une formalisation plus générale a récemment été proposée par V. Saraswat [63], qui considère les systèmes de contraintes comme une généralisation des systèmes d'information de Scott [64] qui ont prouvé leur utilité dans divers domaines de sémantique de la programmation depuis plus d'une décennie. Cette approche met l'accent sur la définition d'une relation de *satisfaction* (*entailment* en anglais) de contraintes qui définit le coeur de la sémantique du système de contraintes. Cette relation de satisfaction indiquera comment les contraintes se déduisent les unes des autres et devra vérifier certaines propriétés que nous allons tout de suite énoncer.

Définition 2.1 *Un* **système de contraintes** *est un couple* $(D, \vdash)$ *tel que :*

- *D est un ensemble de formules atomiques clos par conjonction et quantification existentielle, contenant les constantes* **vrai** *et* **faux** *usuelles.*

- $\vdash$ *est une* **relation de satisfaction** *entre un ensemble fini de formules (noté S) et une formule qui satisfait les règles d'inférence suivantes (classiques en calcul des séquents) :*

$$S, c \vdash c \ \ (\textbf{Struct}) \qquad \frac{S_1 \vdash c_1 \quad S_2, c_1 \vdash c_2}{S_1, S_2 \vdash c_2} \ \ (\textbf{Cut})$$

$$\frac{S, c_1, c_2 \vdash f}{S, c_1 \wedge c_2 \vdash f} \ (\wedge \vdash) \qquad \frac{S \vdash c_1 \quad S \vdash c_2}{S \vdash c_1 \wedge c_2} \ (\vdash \wedge)$$

$$\frac{S, c_1 \vdash c_2}{S, \exists X.\, c_1 \vdash c_2} \ (\exists \vdash) \qquad \frac{S \vdash c[t/X]}{S \vdash \exists X.\, c} \ (\vdash \exists)$$

Dans $(\exists \vdash)$, X ne doit pas être une variable libre dans S, c_2.

- $\vdash$ *est* générique.
  C'est-à-dire que pour toute variable X de S et pour tout terme t :
  $S[t/X] \vdash c[t/X]$ *lorsque* $S \vdash c$

Les systèmes de contraintes définis comme ceci ont un certain nombre de propriétés intéressantes, en particulier l'ensemble de tous ces systèmes forme une catégorie cartésienne fermée, c'est-à-dire close par produit cartésien et par exponentiation.

En général, lorsque l'on voudra définir un système de contraintes, on ne définira pas une relation de satisfaction ex nihilo, mais on utilisera plutôt une relation déjà existante (il faudra alors montrer qu'elle satisfait les bonnes propriétés).

Définition 2.2 *Un* **pré-système de contraintes** *est un couple $(D, \vdash)$ tel que :*

- *D est un ensemble de formules atomiques.*
- $\vdash$ *est une* **relation de satisfaction** *satisfaisant* (**Struct**), (**Cut**) *et étant générique.*

Le théorème suivant permet de définir un système de contraintes à partir d'un pré-système de contraintes.

Théorème 2.1 *[63] Soit $(D', \vdash')$ un pré-système de contraintes. Soit D la clôture de D' pour la quantification existentielle et pour la conjonction. Soit $\vdash$ la clôture de $\vdash'$ par les règles d'inférences. Alors $(D, \vdash)$ est un système de contraintes.*

Par exemple, un système de contraintes peut être construit de manière directe à partir d'une théorie logique du premier ordre (c'est-à-dire d'un ensemble de formules du premier ordre). Prenons une théorie T et définissons D comme la clôture des formules dans le vocabulaire de T par conjonction et quantification existentielle. On définit alors la relation $\vdash_T$ comme suit : $S \vdash_T d$ si et seulement si S implique logiquement d, en utilisant comme extension les axiomes de T. On peut alors montrer facilement que $(D, \vdash_T)$ vérifie les propriétés des systèmes de contraintes données ci-dessus.

On voit ainsi que cette définition généralise naturellement la vision traditionnelle des contraintes comme des formules interprétées dans une structure particulière. Ici la structure est toujours implicite, on ne donne que la relation de satisfaction qui est un élément fondamental à définir pour comprendre comment l'information s'accumule et se propage grâce aux contraintes.

2.2 Le cadre PLC($\mathcal{C}$) - syntaxe et sémantique

Nous présentons la syntaxe des programmes logiques avec contraintes et leur sémantique opérationnelle.

2.2.1 Syntaxe des programmes logiques avec contraintes

Définissons maintenant, étant donné un système de contraintes $\mathcal{C} = (D, \vdash)$, un programme logique avec contraintes sur $\mathcal{C}$.

Considérons un ensemble $\mathcal{P}$ de prédicats et un ensemble énumérable $\mathcal{V}$ de variables qui seront utilisés pour définir les prédicats logiques dans le langage PLC($\mathcal{C}$). Les symboles utilisés dans $\mathcal{P}$ doivent être disjoints de ceux de D, et on considérera en outre que l'ensemble des variables apparaissant dans D est contenu dans $\mathcal{V}$. On suppose finalement que D contient un prédicat "=" qui sera traité comme l'identité et sera utilisé en notation infixe.

Définition 2.3 *Un* **programme logique avec contraintes de PLC($\mathcal{C}$)** *est un ensemble de clauses définies de la forme :*

$$p(\overline{X}) :- q_1(\overline{Y_1}), ..., q_n(\overline{Y_n})$$

où $\overline{X}, \overline{Y_1}, ..., \overline{Y_n}$ *sont des vecteurs de variables de* $\mathcal{V}$ *deux à deux distinctes,* $p(\overline{X}) \in \mathcal{P}$ *et* $q_1(\overline{Y_1}), ..., q_n(\overline{Y_n}) \in \mathcal{P} \cup D$.

$p(\overline{X})$ *est appelé la tête de la clause et* $q_1(\overline{Y_1}), ..., q_n(\overline{Y_n})$ *son corps.*

On remarquera que le corps de la clause contient toutes les contraintes et que les arguments des prédicats sont donc toujours des variables. Ainsi lors d'un appel de procédure il y aura uniquement identification des variables (prédicat "=") entre l'appelant et l'appelé et il est alors inutile d'utiliser un mécanisme d'unification.

2.2.2 Sémantique des programmes logiques avec contraintes

L'exécution d'un programme logique avec contraintes part d'une clause particulière appelée *but*. Il s'agit d'une clause dont la tête est vide et qui se réduit donc à un corps[1]. D'un point de vue procédural, un but va donc contenir un mélange de prédicats, qui seront les procédures à exécuter, et de contraintes, qui représenteront un état initial pour les variables. D'un point de vue logique, le but représente une formule que l'on veut démontrer à l'aide des axiomes du programme.

Par souci de simplicité, nous nous intéresserons ici uniquement à la sémantique opérationnelle de la PLC et formaliserons de manière classique l'exécution d'un programme par une séquence de buts qui partira du but initial pour aboutir à un état final ou boucler.

Une *configuration* est un couple $< \Gamma \ ; \ \theta >$, où Γ est un ensemble de prédicats logiques et des contraintes représentant le calcul à exécuter, et où θ est un ensemble de contraintes représentant l'état courant des variables du calcul. Ceci reprend (et étend) le formalisme classique de la programmation logique où un état de calcul est représenté par un ensemble de buts courants et une substitution.

Définition 2.4 *Soit* P *un programme et* G *un but de PLC(*$\mathcal{C}$*). Un* **calcul** *de* P *à partir de* G *est une séquence (possiblement infinie) de configurations*
$< \Gamma_0 \ ; \ \theta_0 > \rightarrow < \Gamma_1 \ ; \ \theta_1 > \rightarrow \ ... \ \rightarrow < \Gamma_n \ ; \ \theta_n > \rightarrow \ ...,$

où $< \Gamma_0 \ ; \ \theta_0 >$ *est* $< G \ ; \ \emptyset >$, *et où* $< G_{i+1} \ ; \ \theta_{i+1} >$ *se dérive de* $< G_i \ ; \ \theta_i >$ *comme suit.*

Supposons G_i *de la forme* $p_0(\overline{X_0}), p_1(\overline{X_1}), ..., p_k(\overline{X_k})$, *on a :*

[1] est-ce un but enviable ?

- *soit* $p_0(\overline{X_0}) \in \mathcal{P}$ *(c'est-à-dire que* p_0 *est un prédicat logique).*
  Dans ce cas, s'il existe une clause avec un prédicat de tête p_0 *dans* P*, soit* $p_0(\overline{Y_0}) :- q_1(\overline{Y_1}), ..., q_n(\overline{Y_n})$ *un renommage (avec des variables qui n'apparaissent pas dans* $< \Gamma_i \ ; \ \theta_i >$*) de cette clause.*
  Alors Γ_{i+1} *est égal à* $q_1(\overline{Y_1}), ..., q_n(\overline{Y_n}), p_1(\overline{X_1}), ..., p_k(\overline{X_k})$
  et θ_{i+1} *est égal à* $\theta_i \cup \bigwedge_{i=1}^{m} (X_0(i) = Y_0(i))$*, où* m *est l'arité de* p_0*.*

- *soit* $p_0(\overline{X_0}) \in D$ *(c'est-à-dire que* p_0 *est une contrainte).*
  Alors si $p_0(\overline{X_0})$ *est consistant avec* θ_i*, c'est-à-dire formellement*

$$\theta_i \cup p_0(\overline{X_0}) \not\vdash \textbf{faux}$$

  on rajoute cette contrainte dans θ_i *pour obtenir une nouvelle configuration avec* Γ_{i+1} *égal à* $p_1(\overline{X_1}), ..., p_k(\overline{X_k})$ *et* θ_{i+1} *égal à* $\theta_i \cup p_0(\overline{X_0})$*.*

Un tel calcul peut, s'il termine, finir dans une configuration de la forme $< \emptyset \ ; \ \theta >$, auquel cas on a un succès et θ est l'ensemble de contraintes réponse, ou dans une configuration $< \Gamma \ ; \ \theta >$ avec Γ non vide, qui représente alors un échec du calcul (soit parce que l'exécution a rencontré un prédicat non défini, soit parce que l'ensemble de contraintes est devenu inconsistant).

2.3 Le cas $\mathcal{C}$ =FD

Nous allons nous intéresser ici à la définition d'un système de contraintes sur les domaines finis (noté FD).

2.3.1 Les domaines finis

Définition 2.5 *Un* **domaine** *est un ensemble fini non vide de constantes.*

En ce qui nous concerne, nous ne nous intéresserons qu'à des ensembles d'entiers naturels ($\mathcal{N}$) et plus particulièrement à des ensembles appartenant aux parties de $\{0, 1, ..., infinity\}$ où $infinity$ est un entier naturel particulier (représentant la plus grande valeur qu'une

variable DF puisse prendre[2]). *Dom* est l'ensemble de tous les domaines.

Nous utiliserons la notation $k_1..k_2$ comme abréviation de l'ensemble des entiers $\{k \ t.q. \ k_1 \leq k \leq k2\}$.

Définition 2.6 *Soit d un domaine, on définit $min(d)$ et $max(d)$ comme les bornes de d :*

- $min(d) = k \ t.q. \ k \in d \quad et \quad \forall k' \in d \quad k \leq k'$.
- $max(d) = k \ t.q. \ k \in d \quad et \quad \forall k' \in d \quad k \geq k'$.

Nous utiliserons les opérations d'inclusion, d'intersection, d'union et de complémentation (notée $d_1 \setminus d_2$ pour désigner le complémentaire de d_2 dans d_1).

En ce qui concerne les entiers, on utilise les opérations classiques sur les $+$, $-$, $*$ ainsi que les divisions arrondies par défaut et excès (notées respectivement $\lfloor / \rfloor$ et $\lceil / \rceil$).

Nous définissons alors des opérations "point à point" sur les domaines.

Définition 2.7 *Soit d un domaine et i un entier, le domaine associé à $d \cdot i$ avec $\cdot \in \{+, -, *\}$ est défini par $d \cdot i = \{k \ t.q. \ k = k' \cdot i \quad et \quad k' \in d\}$. Enfin, $d/i = \{k \ t.q. \ k = \lfloor k'/i \rfloor \quad et \quad k' \in d\}$*

Notons que le comportement des opérations en cas de débordement (par rapport au domaine $0..infinity$) n'est pas spécifié et correspond à une utilisation erronée de ces opérations.

Définition 2.8 *Soit d un domaine, une* **variable domaine** *sur d est une variable ne pouvant être instanciée qu'à une des valeurs de d.*

Nous noterons $\mathcal{V}_d$ l'ensemble des variables DF.

2.3.2 La contrainte X `in` r - syntaxe et intuition

Le système de contraintes `FD` est basé sur une unique contrainte d'appartenance liant une variable à un domaine. Nous avons 3 types de données syntaxiques : les contraintes (c), les domaines (r) et les termes arithmétiques (t et ct pour les termes constants). L'ensemble

[2]du point de vue de l'implantation, cette valeur dépend de la machine et du paramétrage du système.

des contraintes syntaxiques est nommé $Contr$, celui des domaines syntaxiques $DomSyn$ et celui des termes syntaxiques $TermSyn$.

Définition 2.9 *Une contrainte est une formule de la forme* X `in` r *où* $X \in \mathcal{V}_d$ *et* $r \in DomSyn$ *(cf. syntaxe en table 1).*

$c ::=$	X `in` r	
$r ::=$	$t_1..t_2$	(intervalle)
	$\{t\}$	(singleton)
	R	(paramètre domaine)
	`dom(`Y`)`	(domaine indexical)
	r_1 `:` r_2	(union)
	r_1 `&` r_2	(intersection)
	`-`r	(complémentation)
	r `+` ct	(addition point à point)
	r `-` ct	(soustraction point à point)
	r `*` ct	(multiplication point à point)
	r `/` ct	(division point à point)
$t ::=$	`min(`Y`)`	(terme indexical min)
	`max(`Y`)`	(terme indexical max)
	ct | t_1`+`t_2 | t_1`-`t_2 | t_1`*`t_2 | t_1`/<`t_2 | t_1`/>`t_2	
$ct ::=$	C	(paramètre terme)
	n | `infinity` | ct_1`+`ct_2 | ct_1`-`ct_2 | ct_1`*`ct_2 | ct_1`/<`ct_2 | ct_1`/>`ct_2	

Tableau 1 : syntaxe de la contrainte X `in` r

Nous utiliserons X=n comme abréviation de X `in` $n..n$.

Intuitivement, la contrainte X `in` r contraint X à appartenir au domaine dénoté par r. Celui-ci peut, non seulement dénoter un domaine constant (ex. 1..10) mais aussi un *domaine indexical*, i.e. dépendant de la valeur d'autres variables. Pour cela, les indexicaux suivants sont utilisés :

- `dom(`Y`)` qui représente le domaine courant de Y.
- `min(`Y`)` qui représente la valeur minimale du domaine courant de Y.
- `max(`Y`)` qui représente la valeur maximale du domaine courant de Y.

Chaque domaine/terme indexical fournit une valeur évoluant tout au long du calcul. Ainsi, une contrainte X `in` r sera réévaluée à chaque modification du domaine d'une des variables dont elle dépend. Notons également que la définition d'une contrainte peut utiliser des *paramètres* domaine (resp. terme). Ce sont simplement des variables Prolog devant être liées à une liste d'entiers (resp. à un entier).

Notons que puisque les domaines sont finis, notre système de contraintes sera clos par négation puisque si $c \equiv X$ `in` r (i.e. $X \in r$) alors $\neg c \equiv X$ `in -`r (i.e. $X \in 0..infinity \setminus r$).

Définition 2.10 *Un store*[3] *est un ensemble fini de contraintes.*

Un store *est dit en* **forme normale** *ssi pour toute variable $X \in \mathcal{V}_d$ il ne contient au plus qu'une seule contrainte X* `in` r.

A partir d'un *store* S nous obtenons un *store* S' en forme normale en regroupant toutes les contraintes X `in` r_1, X `in` r_2, ... , X `in` r_n portant sur une même variable X et en les remplaçant par une seule contrainte du type X `in` r_1 `&` r_2 `&` ... `&` r_n. Notons que ces ensembles sont équivalents car ils ont, de manière triviale, les mêmes tuples de valeurs solutions pour les variables.

Nous ne considérerons désormais que des *stores* en forme normale et l'écriture $S \cup \{c\}$ sous-entend que le *store* résultant est en forme normale.

Nous noterons *Store*, l'ensemble de tous les *stores*.

Pendant un calcul, une contrainte X `in` r peut *échouer*, *être satisfaite* ou *être à satisfaire*. Considérons par exemple le *store* {`X in 3..20, Y in 5..7:10..100`} :

- `X in 10..50` est satisfaite avec un nouveau *store* :
  {`X in 3..20 & 10..50, Y in 5..7:10..100`} soit :
  {`X in 10..20, Y in 5..7:10..100`}.
- `X in 30..50` échoue.
- `X in min(Y)..40` est à satisfaire avec un nouveau *store* :
  {`X in 3..20 & 5..40 & min(Y)..40, Y in 5..7:10..100`} soit :
  {`X in 5..20 & min(Y)..40, Y in 5..7:10..100`}.

[3] nous utiliserons le mot anglais *store* car il n'a pas de traduction française heureuse.

Remarquons que la contrainte indexicale `X in min(Y)..40` fournit une contrainte évaluée dans le *store* courant (ex. `X in 5..40`) et *reste* telle quelle pour propager les réductions futures de `Y`.

- `X in dom(Y)+1` est à satisfaire avec un nouveau *store* :
  {`X in 3..20 & 6..8:11..101 & dom(Y)+1, Y in 5..7:10..100`} soit :
  {`X in 6..8:11..20 & dom(Y)+1, Y in 5..7:10..100`}.

Voyons alors comment définir une contrainte de haut niveau (appelée *contrainte utilisateur*) à partir de contraintes X `in` r. La contrainte X `in` r doit être vue comme un moyen de spécifier le schéma de propagation. En fait, X `in` r permet de spécifier *quoi propager*. Par exemple, les contraintes $X = Y + C$ et $X + Y = Z$ peuvent se définir comme suit :

Exemple 2.1

```
'x=y+c'(X,Y,C):- X in min(Y)+C..max(Y)+C,
                 Y in min(X)-C..max(X)-C.
```
◇

Exemple 2.2

```
'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).
```
◇

Dans cette version seules les modifications des bornes des variables sont propagées (on parle de *partial lookahead*). Donc, si un "trou" apparaît au "milieu" d'un domaine, ce "trou" n'est pas propagé. En ce qui concerne $X = Y + C$ il serait possible de propager toute modification du domaine (i.e. on parle alors de *full lookahead*) grâce à la définition suivante :

Exemple 2.3

```
'x=y+c'(X,Y,C):- X in dom(Y)+C,
                 Y in dom(X)-C.
```
◇

Notons toutefois que nous ne pouvons pas définir $X + Y = Z$ en *full lookahead*. Nous présenterons en section 7.3.1, une extension de la syntaxe de X `in` r permettant la définition de tels schémas de propagation par *full lookahead*.

Les contraintes $X \geq Y$ et $X \geq A * Y$ peuvent se définir comme suit :

Exemple 2.4

```
'x≥y'(X,Y):-    X in min(Y)..infinity,
                Y in 0..max(X).
```
◇

Exemple 2.5

```
'x≥ay'(X,A,Y):- X in A*min(Y)..infinity,
                Y in 0..max(X)/<A.
```
◇

La contrainte $X \neq Y$ se définit par :

Exemple 2.6

```
'x≠y'(X,Y):-    X in -dom(Y),
                Y in -dom(X).
```
◇

Nous verrons plus bas qu'une contrainte comme X `in -dom(`Y`)` est *anti-monotone*. Intuitivement, le domaine dénoté par `-dom(`Y`)` augmente au fur et à mesure du calcul (i.e. puisque le domaine de Y diminue, son complémentaire augmente). Ceci pose un problème du point de vue de l'implantation puisque une valeur inconsistante pour X (i.e. n'appartenant pas au complémentaire du domaine de Y) peut devenir consistante dans la suite du calcul du fait d'une réduction de Y. Une telle contrainte ne pourra donc être traitée que lorsque Y sera instancié. On parle alors de *forward checking*. Il serait possible d'utiliser un `freeze` ou autre `delay` pour retarder l'évaluation d'une telle contrainte. Nous avons choisi de définir un pseudo-terme indexical `val(`X`)` qui retarde l'activation de toute contrainte dans lequel il apparaît jusqu'à ce que X soit clos. Nous étudierons en section 6 un moyen plus élégant pour réaliser un tel retardement grâce à l'opération Ask.

2.3.3 Sémantique de X `in` r et opération $Tell$

L'opération $Tell$ permet d'ajouter une contrainte au *store* courant. Sa sémantique dénotationnelle est présentée en table 2.

La fonction $\mathcal{T}\ [\![X \texttt{ in } r]\!]$ traduit la sémantique de l'opération $Tell$ de X `in` r dans un *store* S. Celle-ci consiste à modifier X et à réévaluer les contraintes dépendant de X pour assurer la consistance (i.e. propagation). La première phase est assurée par la fonction sémantique intermédiaire $\mathcal{T}'\ [\![X \texttt{ in } r]\!]$ et la seconde est assurée par l'utilisation d'un

$DomSyn$	: domaines syntaxiques	$\mathcal{T}$	: $Contr \rightarrow Store \rightarrow Store$
$TermSyn$	: termes syntaxiques	$\mathcal{T}'$	: $Contr \rightarrow Store \rightarrow Store$
Dom	: domaines	$\mathcal{E}_r$	: $DomSyn \rightarrow Store \rightarrow Dom$
$\mathcal{N}$	: entiers naturels	$\mathcal{E}_t$	: $TermSyn \rightarrow Store \rightarrow \mathcal{N}$
$Contr$	: contraintes X `in` r		
$Store$	: *stores*		

$$
\begin{array}{ll}
\mathcal{T}\ [\![c]\!]\ s & = \texttt{fix}\ (\lambda\ s\ .\ \bigcup_{c' \in s \cup \{c\}} \mathcal{T}'\ [\![c']\!]\ s) \\
\mathcal{T}'\ [\![x\ \texttt{in}\ r]\!]\ s & = \textbf{let}\ \texttt{d} = \lceil \mathcal{E}_r\ [\![r]\!]\ s \rceil\ \textbf{in}\ s \cup \{\ x\ \texttt{in}\ d\ \} \cup \{\ x\ \texttt{in}\ r\ \} \\
 & \\
\mathcal{E}_r\ [\![t_1 \texttt{..} t_2]\!]\ s & = \mathcal{E}_t\ [\![t_1]\!]\ s\ ..\ \mathcal{E}_t\ [\![t_2]\!]\ s \\
\mathcal{E}_r\ [\![\{t\}]\!]\ s & = \{\mathcal{E}_t\ [\![t]\!]\ s\} \\
\mathcal{E}_r\ [\![R]\!]\ s & = \texttt{lookup_range}(R) \\
\mathcal{E}_r\ [\![\texttt{dom(}Y\texttt{)}]\!]\ s & = \texttt{cur_domain}(X\texttt{,}s) \\
\mathcal{E}_r\ [\![r_1\ \texttt{:}\ r_2]\!]\ s & = \mathcal{E}_r\ [\![r_1]\!]\ s\ \cup\ \mathcal{E}_r\ [\![r_2]\!]\ s \\
\mathcal{E}_r\ [\![r_1\ \texttt{\&}\ r_2]\!]\ s & = \mathcal{E}_r\ [\![r_1]\!]\ s\ \cap\ \mathcal{E}_r\ [\![r_2]\!]\ s \\
\mathcal{E}_r\ [\![\texttt{-}r]\!]\ s & = 0..infinity\ \setminus\ \mathcal{E}_r\ [\![r]\!]\ s \\
\mathcal{E}_r\ [\![r\ \texttt{+}\ ct]\!]\ s & = \mathcal{E}_r\ [\![r]\!]\ s\ +\ \mathcal{E}_t\ [\![ct]\!]\ s \\
\mathcal{E}_r\ [\![r\ \texttt{-}\ ct]\!]\ s & = \mathcal{E}_r\ [\![r]\!]\ s\ -\ \mathcal{E}_t\ [\![ct]\!]\ s \\
\mathcal{E}_r\ [\![r\ \texttt{*}\ ct]\!]\ s & = \mathcal{E}_r\ [\![r]\!]\ s\ *\ \mathcal{E}_t\ [\![ct]\!]\ s \\
\mathcal{E}_r\ [\![r\ \texttt{/}\ ct]\!]\ s & = \mathcal{E}_r\ [\![r]\!]\ s\ /\ \mathcal{E}_t\ [\![ct]\!]\ s \\
 & \\
\mathcal{E}_t\ [\![n]\!]\ s & = n \\
\mathcal{E}_t\ [\![\texttt{infinity}]\!]\ s & = infinity \\
\mathcal{E}_t\ [\![C]\!]\ s & = \texttt{lookup_term}(C) \\
\mathcal{E}_t\ [\![\texttt{min(}Y\texttt{)}]\!]\ s & = min(\texttt{cur_domain}(X\texttt{,}s)) \\
\mathcal{E}_t\ [\![\texttt{max(}Y\texttt{)}]\!]\ s & = max(\texttt{cur_domain}(X\texttt{,}s)) \\
\mathcal{E}_t\ [\![t_1\ \texttt{+}\ t_2]\!]\ s & = \mathcal{E}_t\ [\![t_1]\!]\ s\ +\ \mathcal{E}_t\ [\![t_2]\!]\ s \\
\mathcal{E}_t\ [\![t_1\ \texttt{-}\ t_2]\!]\ s & = \mathcal{E}_t\ [\![t_1]\!]\ s\ -\ \mathcal{E}_t\ [\![t_2]\!]\ s \\
\mathcal{E}_t\ [\![t_1\ \texttt{*}\ t_2]\!]\ s & = \mathcal{E}_t\ [\![t_1]\!]\ s\ *\ \mathcal{E}_t\ [\![t_2]\!]\ s \\
\mathcal{E}_t\ [\![t_1\ \texttt{/<}\ t_2]\!]\ s & = \lfloor \mathcal{E}_t\ [\![t_1]\!]\ s\ /\ \mathcal{E}_t\ [\![t_2]\!]\ s \rfloor \\
\mathcal{E}_t\ [\![t_1\ \texttt{/>}\ t_2]\!]\ s & = \lceil \mathcal{E}_t\ [\![t_1]\!]\ s\ /\ \mathcal{E}_t\ [\![t_2]\!]\ s \rceil \\
 & \\
\texttt{cur_domain}(X\texttt{,}s) & = \mathcal{E}_r\ \texttt{lookup_store}(X\texttt{,}s)\ \emptyset \\
\texttt{lookup_store}(X\texttt{,}s) & = \textbf{if}\ \exists\ X\ \texttt{in}\ r \in s\ \textbf{then}\ r\ \textbf{else}\ \texttt{0..infinity} \\
 & \\
\texttt{lookup_range}(R) & \text{retourne le domaine lié à } R \\
\texttt{lookup_term}(C) & \text{retourne l'entier lié à } C
\end{array}
$$

Tableau 2 : sémantique dénotationnelle de l'opération $Tell$

point fixe sur le *store* résultat rendu par $\mathcal{T}\ [\![X \texttt{ in } r]\!]$ qui réévalue toutes les contraintes de $S \cup \{X \texttt{ in } r\}$ (via $\mathcal{T}'$) jusqu'à l'obtention d'un état stable.

La fonction $\mathcal{T}'\ [\![X \texttt{ in } r]\!]$ "ajoute" au *store* deux versions de la contrainte $X \texttt{ in } r$ permettant une prise en compte des contraintes indexicales dont la valeur évolue au fil des calculs.

(a) une version de $X \texttt{ in } r$ où r est *évalué* dans S grâce à la fonction sémantique $\mathcal{E}_r\ [\![r]\!]$ (l'écriture $\lceil \mathcal{E}_r\ [\![r]\!]\ s \rceil$ représente le domaine syntaxique associé à l'évaluation de r). Cette version permet de disposer explicitement du domaine de X à chaque étape.

(b) une version de $X \texttt{ in } r$ où r est inchangé. Ceci permettra la prise en compte (future) des indexicaux dans les *stores* plus contraints.

L'évaluation d'un indexical (ex. `dom(`X`)` nécessite la récupération du domaine courant d'une variable. Du fait de version (a) (cf. ci-dessus), le domaine de toute variable X dans S s'obtient en récupérant la contrainte $X \texttt{ in } r$ qui lui est associée dans S (cf. fonction `lookup_store`) et en évaluant r *sans tenir compte des indexicaux* (pour éviter les boucles infinies), ce qui revient à évaluer r dans le *store* vide (cf. fonction `cur_domain`).

En vue de simplifier encore un peu les notations nous utiliserons les abréviations suivantes relatives à un *store* S :

- $X_S =$`cur_domain(`X`,`S`)` (i.e. la valeur du domaine de X dans S).
- $min(X)_S = min(X_S)$.
- $max(X)_S = max(X_S)$.
- $r_S = \mathcal{E}_r\ [\![r]\!]\ S$ (i.e. domaine dénoté par r dans S).
- $t_S = \mathcal{E}_t\ [\![t]\!]\ S$ (i.e. entier dénoté par t dans S).

Définition 2.11 *Soit S et S' deux ensembles de contraintes, on dit que S' est* **plus contraint** *que S (noté $S' \sqsubseteq S$) ssi $\forall X \in \mathcal{V}_d \quad X_{S'} \subseteq X_S$.*

Nous pouvons alors définir une équivalence entre *stores* comme suit :

Définition 2.12 *Deux* stores *S_1 et S_2 sont* **équivalents** *(noté $S_1 \Leftrightarrow S_2$) ssi $S_1 \sqsubseteq S_2$ et $S_2 \sqsubseteq S_1$.*

Le fait d'utiliser un point fixe pour $Tell$ impose que cette opération soit monotone. De manière plus précise, si une valeur est inconsistante pour une variable DF à un moment donné, elle ne doit pas (re)devenir consistante par la suite :

Définition 2.13 *Un domaine r est* **monotone** *(resp.* **anti-monotone***) ssi* $\forall S, S' \quad S' \sqsubseteq S \Rightarrow r_{S'} \subseteq r_S$ *(resp.* $r_S \subseteq r_{S'}$*).*

Une contrainte $c \equiv X$ `in` r est (anti-)monotone ssi r est (anti-)monotone.

Nous pouvons alors imposer que l'ajout d'une contrainte c ne puisse se faire que si c est monotone (i.e. tout *store* S ne contient donc que des contraintes monotones). De ce fait, le domaine de toute variable $X \in \mathcal{V}_d$ est monotone.

Définition 2.14 *Un* store *S est* **consistant** *ssi il ne contient aucune variable de domaine vide, i.e.* $\forall X \in \mathcal{V}_d \quad X_S \neq \emptyset$.

Définition 2.15 *Une variable X est* **instanciée** *à l'entier n dans un* store *S ssi* $X_S = \{n\}$.

On dit aussi que X est *clos* dans S (noté $ground(X)$). On étend de manière évidente cette définition à un domaine et à une contrainte.

2.3.4 La relation de satisfaction

Définissons maintenant la relation de satisfaction au coeur de notre système de contraintes.

Définition 2.16 *Un* store *S* **satisfait** *une contrainte $c \equiv X$ `in` r ssi c est vraie dans tout store S' plus contraint que S, i.e.*

$$S \vdash c \text{ ssi } \forall S' \quad S' \sqsubseteq S \Rightarrow X_{S'} \subseteq r_{S'}$$

Un store *S* **contredit** *une contrainte $c \equiv X$ `in` r ssi S satisfait $\neg c$, i.e. $S \vdash X$ `in` $-r$.*

Grâce à cette relation nous pouvons définir une équivalence entre contraintes permettant d'assurer que deux contraintes fournissent les mêmes tuples de variables comme solutions.

Définition 2.17 *Deux contraintes c_1 et c_2 sont* **équivalentes** *ssi* $\forall S \quad S \vdash c_1 \Leftrightarrow S \vdash c_2$.

Proposition 2.1 *Si $S \vdash c$ alors $S \cup \{c\} \Leftrightarrow S$*

Preuve :
montrons la proposition contraposée.
Supposons que $S \cup \{c\} \not\Leftrightarrow S$, du fait que $S \cup \{c\} \sqsubseteq S$ on en déduit que $S \cup \{c\} \sqsupset S$.
Autrement dit $\exists Y \;\; Y_{S\cup\{c\}} \supset Y_S$ (i.e. Y est réduit par l'ajout de c).
Soit $c \equiv X$ `in` r, il est évident que X a aussi été réduit (sinon aucune autre variable n'aurait été réduite). On en conclut que $r_S \subset X_S$ ce qui implique que $S \not\vdash c$. □

Soit $Contr$ notre ensemble de formules atomiques (la constante **vraie** correspondant à `X in 0..infinity` et **faux** à `X in 1..0`) et $\vdash$ la relation de satisfaction définie ci-dessus. Montrons que $(Contr, \vdash)$ est un pré-système de contraintes.

Proposition 2.2 $\vdash$ *vérifie* (**Struct**), (**Cut**) *et est générique.*
Preuve :
(**Struct**) : soit $S = S_0 \cup \{X$ `in` $r\}$ il nous faut montrer que $S \vdash X$ `in` r.
Il suffit de vérifier que $\forall S' \;\; S' \sqsubseteq S \Rightarrow X_{S'} \subseteq r_{S'}$.
Ceci est trivialement vérifié car $S = \mathcal{T} \;[\![X$ `in` $r]\!]\; S_0$ donc tout $S' \sqsubseteq S$ contient X `in` r du fait de la version (b) (cf. sémantique de $Tell$) assurant que $X_{S'} \subseteq r_{S'}$.

(**Cut**) : il nous faut prouver que si $S_1 \vdash c_1$ et $S_2 \cup \{c_1\} \vdash c_2$ alors $S_1 \cup S_2 \vdash c_2$.
On a $S_1 \cup S_2 \cup \{c_1\} \sqsubseteq S_2 \cup \{c_1\}$.
Sachant que $S_2 \cup \{c_1\} \vdash c_2$ et du fait que $S_1 \vdash c_1 \rightarrow S_1 \Leftrightarrow S_1 \cup \{c_1\}$ (proposition 2.1)
on en conclut que : $S_1 \cup S_2 \vdash c_2$.

$\vdash$ **est générique** : il nous faut montrer que si $S \vdash c$ alors $S[n/X] \vdash c[n/X]$.
Notons que $S[n/X] \Leftrightarrow S \cup \{X{=}n\}$
et que $c[n/X]$ n'est rien d'autre que la contrainte c pré-évaluée dans $\{X{=}n\}$.
Montrons donc que si $S \vdash c$ alors $\forall S' \;\; S' \sqsubseteq S \cup \{X{=}n\} \Rightarrow X_{S'\cup\{X=n\}} \subseteq r_{S'\cup\{X=n\}}$.
Du fait que $S' \sqsubseteq S \cup \{X{=}n\}$
ceci revient a montrer que si $S \vdash c$ alors $\forall S' \;\; S' \sqsubseteq S \cup \{X{=}n\} \Rightarrow X_{S'} \subseteq r_{S'}$.
Or ce n'est là qu'un cas particulier de : $\forall S' \;\; S' \sqsubseteq S \Rightarrow X_{S'} \subseteq r_{S'}$ (i.e. $S \vdash c$)
du fait que $S \cup \{X{=}n\} \sqsubseteq S$. □

Soit D la clôture par quantification existentielle et pour la conjonction de $Contr$. Par souci de simplification on notera aussi $\vdash$ la clôture par les règles d'inférences de notre relation de satisfaction. Le théorème 2.1 nous indique que `FD`$=(D, \vdash)$ est un système de contraintes.

Chapitre 3

Implantation de wamcc

Dans cette partie, nous détaillerons l'implantation de `wamcc`, le langage Prolog de base sur lequel est construit `clp(FD)`. La solution proposée étant basée sur l'incontournable machine abstraite de Warren (WAM), nous commencerons par son étude. Après quoi nous nous poserons le problème de l'exécution du code abstrait obtenu pour aboutir à la solution retenue : traduire Prolog vers C. Une analyse des performances terminera ce chapitre.

3.1 La machine abstraite de Warren

Jusqu'en 1983 la compilation de Prolog semblait réservée à quelques illuminés et pour le commun des implanteurs il y avait contradiction dans les termes "Prolog" et "compilation". La révélation de la WAM [75] à réellement rendu abordable la compilation de Prolog par tous. Et bien que certaines alternatives plus optimisées aient été proposées (ex. la BAM [73]), la WAM reste un standard grâce à sa simplicité et son efficacité. Le succès de la WAM tient également au fait que sa conception permet des extensions aisées (ex. backtracking intelligent, test d'occurrence, parallélisme, contraintes, etc...).

Nous nous proposons d'aboutir à la définition de la WAM en partant de la notion d'arbre de recherche qui traduit la sémantique opérationnelle d'un programme Prolog. Cette démarche a pour intérêt d'introduire les particularités de la WAM tout au long du cheminement, alors qu'une présentation par énumération du jeu d'instructions nécessite autant de digressions

*le contenu de ce chapitre a été publiée dans [23].

pour tenter de les justifier. Notons que nous partons tout de suite de l'exécution de Prolog "complet", à l'inverse de [3] qui introduit plusieurs niveaux de programmes logiques (uniquement des faits clos, des faits quelconques, puis une clause par prédicat, et, enfin, les prédicats indéterministes).

Pour une autre présentation de la WAM le lecteur pourra se référer à [3].

3.1.1 La pile locale (ou de contrôle)

Le lecteur étant supposé familier avec les concepts de base de Prolog, nous nous contenterons de rappeler brièvement la notion d'arbre de recherche.

Définition 3.1 *Un* **état de recherche** *est un triplet* $< r, \sigma, b >$ *où :*

- r *est un numéro de clause,*
- σ *est une substitution (sous la forme* $\{X_1 \leftarrow t_1, ..., X_n \leftarrow t_n\}$*),*
- b *est une suite d'atomes* $B_1, ..., B_p$*.*

Définition 3.2 *Un* **arbre de recherche standard** *à partir du but* $Q_1...Q_m$ *est un arbre fini ou infini, où chaque noeud est occurrence d'un état de recherche.*

La racine est occurrence de $< 0, \{\}, Q1...Qm >$*.*

Un noeud occurrence de $< r, \sigma, b >$ *avec* b *vide est appelé* **feuille succès***.*

Pour tout autre noeud ν *occurrence de* $< r, \sigma, b >$ *et pour tout* ν' *fils de* ν *occurrence de* $< r', \sigma', b' >$ *alors :*

- *l'application* $\nu' \rightarrow r'$ *est une bijection de l'ensemble des noeuds fils de* ν *dans l'ensemble des numéros de clause* r' *tel que* B_1 *est unifiable avec la tête de la clause de numéro* r' *(*Cr'*).*
- ν *est une* **feuille échec** *s'il n'y a aucun* r'*.*
- $\forall \nu' \exists A'_0 \leftarrow A'_1...A'_n$ *une variante de* Cr' *(renommage de clause) sans variables communes avec les buts des états des noeuds ancêtres de* ν'*.*

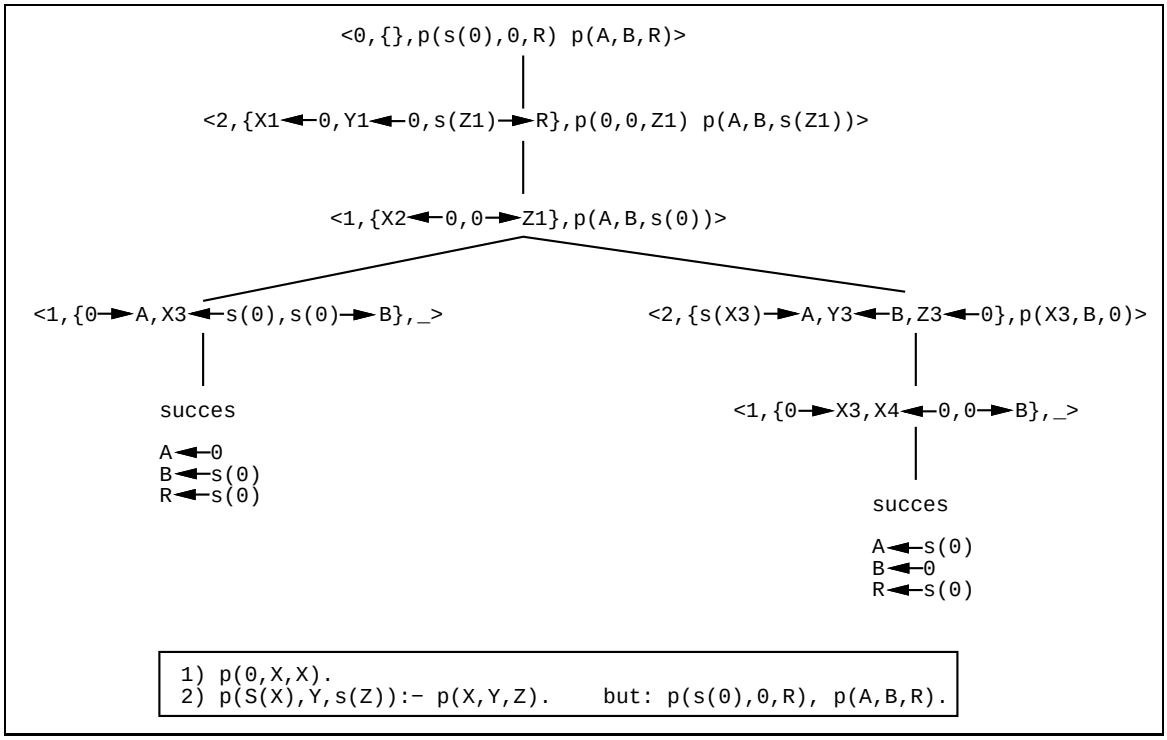


Figure 1 : exemple d'arbre de recherche standard

- *σ' est l'unificateur minimal de B_1 et A'_0 et b' est le résultat du remplacement dans $\sigma'(b)$ de B_1 par $\sigma'(A'_1...A'_n)$.*
- *l'ordre des fils de ν est l'ordre dans r'.*

La figure 1 montre un exemple d'arbre de recherche standard. Le prédicat `p(X,Y,Z)` a pour lecture déclarative $X + Y = Z$ (un entier n étant codé par le terme $s^n(0)$). Le but lancé est `p(s(0),0,R), p(A,B,R)` qui peut se lire comme : "soit R le résultat de 1+0, quelles sont les valeurs A et B telles que $A + B = R$".

Remarquons que la structure récursive de l'arbre se manifeste par le sens que l'on peut donner au sous arbre associé à chaque noeud indépendamment de ses ancêtres. En effet, à partir d'un noeud occurrence de $< r, \sigma, b >$, l'on obtient un arbre de recherche standard pour le but b.

De plus, la *stratégie standard* (i.e. de Prolog) consiste en un parcours en profondeur d'abord et de gauche à droite de l'arbre de recherche. Ce parcours, qui induit un ordre sur les solutions, devient aisément automatisable grâce à une pile (que nous appellerons *pile de*

contrôle ou *pile locale*). Cette pile contiendra les triplets correspondants aux états de recherche. Lorsqu'un échec survient (aucune tête de clause ne s'unifie avec le littéral courant) la procédure de backtraking devra dépiler un certain nombre d'éléments jusqu'à l'obtention d'un triplet pour lequel il existe une alternative. En vue de diminuer cette recherche, donc d'accéder en temps constant à l'élément susceptible de fournir une nouvelle solution, il est possible de déterminer, lors de l'avancée, si le noeud courant donnera lieu à un retour (backtracking) et, dans ce cas, d'empiler un élément particulier appelé *point de choix*. Ainsi, la pile locale gère le contrôle de Prolog que l'on peut séparer en deux phases :

- avancée : appels imbriqués de procédure. Classiquement on utilise un *environnement* (ou bloc d'activation) où sont stockées les variables (locales) de la clause et les informations de contrôle utiles à la sortie du bloc courant (i.e. de la clause). Ces environnements sont similaires à ceux nécessaires lors de l'implantation d'un langage qui gère des variables locales (ex. C). Ainsi une clause peut être comparée à une fonction, et ses variables (nouvelles instances à chaque utilisation de la clause) à des variables locales.

- retour-arrière (backtracking) : consistant à reprendre le calcul à la dernière alternative non encore exploitée. Ainsi un *point de choix* stocke les informations nécessaires à la reprise du calcul. On y trouve donc la sauvegarde de la plupart des registres de base ainsi que l'adresse de la nouvelle clause à essayer.

Dans la WAM originale ces deux types de blocs de contrôle sont entrelacés et donc chaque bloc contient un pointeur vers le bloc précédent de même type pour permettre le dépilement. Deux registres de base `E` et `B` pointent respectivement sur le dernier environnement et sur le dernier point de choix. Ainsi la pile locale contient deux piles entrelacées avec deux sommets de pile et la possibilité, à partir de chaque bloc, d'accéder au bloc précédent de même type. Lorsqu'un nouveau bloc doit être alloué, le calcul de son adresse revient au calcul du maximum entre les deux sommets de pile. La solution avec deux piles distinctes (dites de contrôle et de choix) est adoptée dans certaines implantations commerciales. Toutefois l'utilisation d'une seule pile simplifie la "synchronisation" des points de choix et des environnements grâce au calcul du maximum décrit précédemment. Ainsi, lorsqu'un environnement est libéré alors qu'un point de choix a été créé au-dessus de lui (par un des ses fils), l'espace de cet environnement n'est pas réutilisé puisque il sera nécessaire lors du backtracking.

3.1.2 La pile de restauration (ou trail)

Intéressons-nous à présent au problème de la représentation des substitutions. Le fait que chaque état $< r, \sigma, b >$ donne lieu à un environnement où sont stockées les variables locales de la clause revient à représenter σ par un vecteur où un élément est associé de manière bi-univoque à une variable de la clause. Chaque élément de $\sigma[i]$ indique alors la valeur associée à la ième variable de la clause. La figure 1 nous permet de constater que certaines substitutions affectent des variables n'apparaissant pas dans la clause courante. Ceci est mis en évidence par des flèches de gauche à droite. Ces liaisons affectent alors des $\sigma[i]$ d'un environnement antérieur à l'environnement courant engendrant ainsi un nouveau problème : lors du backtracking, si cet environnement est également antérieur à celui du dernier point de choix, il faudra *défaire* ces liaisons pour pouvoir relancer, *avec les mêmes données*, le calcul sur une autre alternative. Sur la figure 1, nous pouvons remarquer, au sujet de la liaison de `A` avec `0` (branche de gauche du point de choix), que le fait d'écrire cette liaison à cet endroit (et non pas à l'état racine où apparaît la variable) indique bien que, lors du backtracking, nous voulons "oublier" cette liaison (la meilleure preuve étant que, dans la branche droite, `A` est lié à `s(X3)`).

La solution pour remettre à l'état initial (non lié) de telles variables est d'utiliser une autre pile, dite *pile de restauration* (ou *trail*). Lorsqu'une variable antérieure au dernier point de choix doit être liée, nous empilons sa référence. Ainsi, à condition de mémoriser le sommet de cette pile dans les points de choix, il suffit lors du backtracking de remettre à l'état "libre" toutes les variables référencées dans la trail entre le sommet actuel et celui enregistré dans le dernier point de choix.

La figure 1 nous montre également que lorsque nous sommes en présence de deux variables, rien n'indique comment orienter la substitution (est-ce $X \leftarrow Y$ ou $X \rightarrow Y$?). Nous pouvons toutefois remarquer que, dans ce cas, une des variables appartient forcément à la clause courante (renommée), c'est donc celle-ci qui enregistrera la liaison pour éviter un éventuel empilement en trail. Ainsi :

La liaison entre deux variables s'effectue toujours de la plus récente vers la plus ancienne, i.e. de celle d'adresse la plus grande vers celle d'adresse la plus petite.

Nous verrons par la suite d'autres avantages à une telle règle.

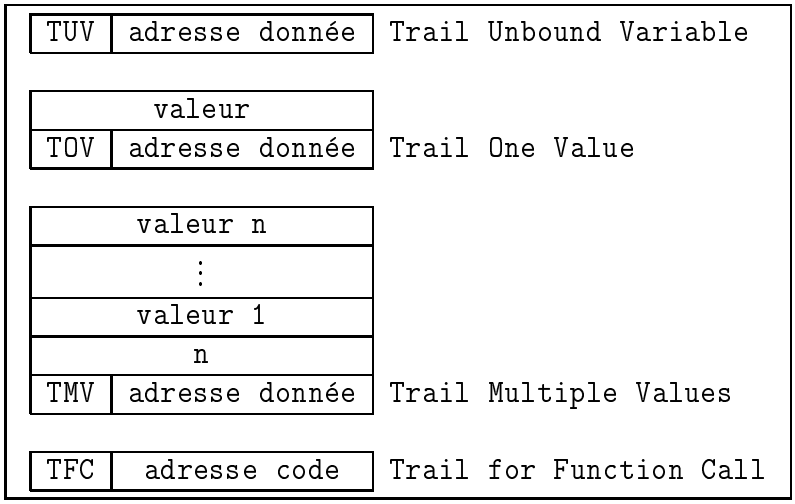


Figure 2 : structures de la trail

Dans la WAM originale, la trail est une pile à une seule entrée permettant de stocker les références (i.e. adresses) des variables à ré-initialiser. Dans un souci d'extensibilité, la trail que nous utilisons est à entrées multiples pour permettre d'associer de l'information aux adresses à ré-initialiser. En plus de la ré-initialisation classique d'une variable Prolog à l'état libre, la trail de `wamcc` permet de ré-initialiser un ou plusieurs mots avec des valeurs sauvegardées. Enfin, un entrée "fonction" permet, au moment des ré-initialisations, d'invoquer une fonction C. Ces différentes structures sont distinguées au moyen de *mots étiquetés*[1], c'est-à-dire qu'un mot mémoire est une paire de la forme <*étiquette,valeur*>. L'étiquette indiquant le type de donnée codé par le champ valeur. La figure 2 détaille les différentes structures de ré-initialisation.

3.1.3 La pile globale (ou heap)

D'après ce que nous venons de voir, si une liaison implique une mise en pile de restauration, c'est qu'elle concerne une variable et un terme autre qu'une variable. Toutefois, nous n'avons pas abordé la manière dont sont stockés les termes (notamment ceux de hauteur supérieure à 1). En effet les vecteurs σ enregistrés dans les environnements ne peuvent avoir qu'une taille fixe. Où stocker les termes ? Une solution consiste à utiliser une nouvelle pile (dite *pile globale* ou *heap*) pour stocker les termes structurés. Elle a une structure de pile

[1] traduction de *tagged word*.

dans la mesure où les nouveaux termes sont empilés sur son sommet (pointé par le registre `H`) et qu'elle est dépilée lors du backtracking. Elle à une structure de tas du fait que des liaisons existent aussi bien du bas de la pile vers le haut que du haut vers le bas.

3.1.4 La représentation des termes

Les termes sont codés par de mots étiquetés comme dans la majorité des langages dynamiquement typés (cf. [38] pour une étude complète de la codification des types dans de tels langages). La figure 3 schématise la représentation interne de chaque terme, à savoir :

variable : la partie valeur est une référence vers le terme auquel est liée la variable. Une variable libre est simplement représentée comme une auto-référence (i.e. si α est son adresse, son contenu est `<REF,`α`>`). Notons que l'affectation d'un tel mot à un registre crée un lien du registre vers la variable. On appelle *déréférenciation* l'opération consistant à suivre un chaînage de variables liées jusqu'à ce qu'une variable libre ou un terme différent d'une variable soit rencontré.

constante : la partie valeur pointe dans une table de hash-code stockant toutes les constantes, ramenant ainsi la comparaison de deux constantes à la comparaison de deux entiers.

entier : la partie valeur code l'entier.

liste vide : elle est simplement représentée par la constante particulière `'[]'`.

liste non vide : la partie valeur pointe une cellule du tas contenant le Car, la suivante contenant le Cdr.

structure : la partie valeur pointe une cellule du tas contenant le foncteur (une adresse dans la table de hash-code des constantes) et l'arité (nombre n de sous-termes). Consécutivement à ce mot viennent les n mots étiquetés associés aux sous-termes.

Une des principales caractéristiques de la WAM est due au choix de représentation des termes composés (i.e. listes, structures) par *recopie de structure*. En effet, un terme est traité différemment suivant qu'il est *décomposé* (accès à une instance déjà existante) ou *construit* (création d'une nouvelle instance à partir d'un modèle). Pour une liaison en décomposition

la variable sera simplement liée à l'instance déjà existante alors qu'en construction la variable est liée à une nouvelle copie du modèle. La WAM définit ainsi deux modes lors de l'unification de termes structurés :

- READ : correspondant à une décomposition. Dans ce cas le terme existe dans le tas et un registre de base, nommé `S`, contient son adresse. L'unification des sous-termes peut avoir lieu par rapport à `S`.
- WRITE : correspondant à une construction. Dans ce cas le terme est construit sur le sommet du tas (pointé par `H`).

Pour éviter l'utilisation d'un registre spécifique pour coder le mode le registre `S` est mis à `NULL` en mode WRITE.

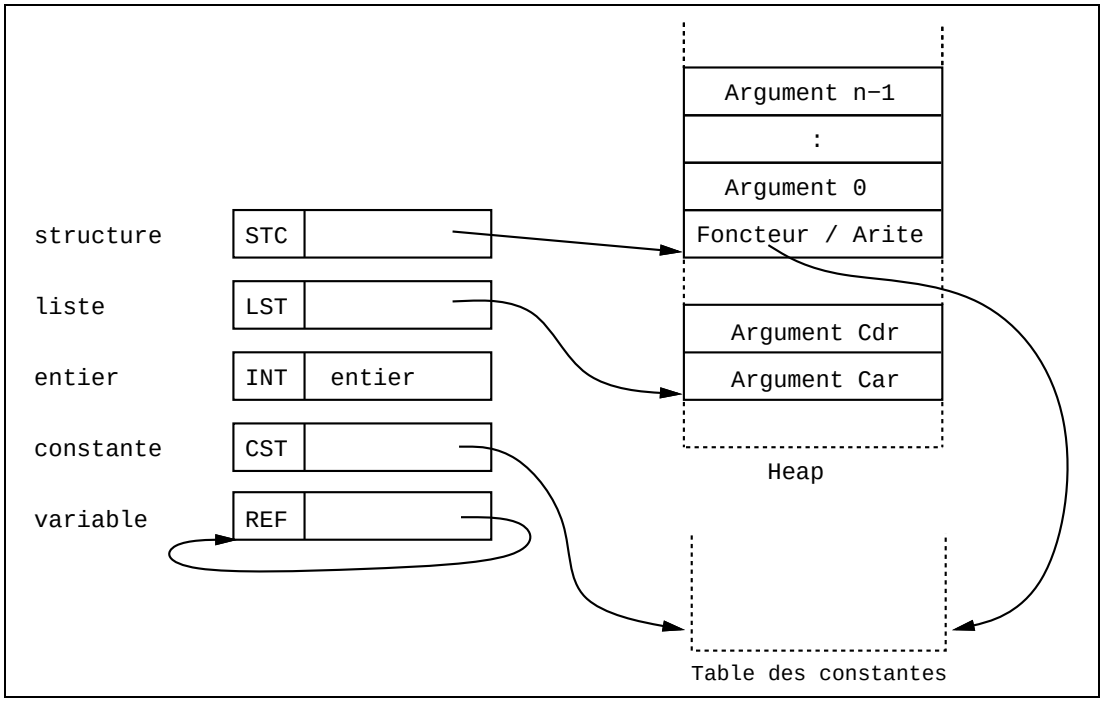


Figure 3 : représentation des termes dans la WAM

3.1.5 Registres

La WAM utilise les registres de base suivants :

`PC`	(Program Counter)	pointeur de programme.
`CP`	(Continuation Program)	pointeur de continuation.
`E`	(Environment)	pointeur sur environnement courant.
`B`	(Backtrack)	pointeur sur point de choix courant.
`BC`	(Backtrack Cut)	pointeur sur point de choix pour coupure.
`H`	(Heap)	pointeur sur sommet du heap.
`S`	(Structure pointer)	pointeur sur structure à décomposer.
`TR`	(Trail)	pointeur sur sommet de la trail.
`A[i]`	(Arguments)	banc de registres arguments (aussi nommé `X[i]`).

Le registre `BC` est une extension à la WAM permettant de prendre en compte la coupure (`!/0`). Nous ne détaillerons pas la gestion de la coupure du fait que `wamcc` traite celle-ci de manière classique (cf. si besoin [3, 15]).

Les registres arguments (notés `A[i]`) servent d'interface pour les données entre l'appelant et l'appelé. Ces registres sont chargés par l'appelant et sont unifiés avec la tête de clause par l'appelé, ce qui a pour effet de charger son environnement (les variables de la clause reçoivent en effet leurs valeurs grâce à l'unification). Si celle-ci réussit, la clause est utilisable ; pour chacun des prédicats du corps les registres sont chargés avec les arguments appropriés et le contrôle est transféré au prédicat concerné. S'il est possible de détecter les variables telles qu'entre leur première et leur dernière occurrence aucun appel à un prédicat ne sera fait, alors celles-ci peuvent être gérées directement dans les registres plutôt que dans l'environnement. De telles variables sont qualifiées de temporaires (notées `X[i]`) par opposition aux variables gérées au travers de l'environnement qui sont dites permanentes (notées `Y[j]`).

Définition 3.3 *Une variable* temporaire *est une variable n'apparaissant que dans un seul but, la tête et le premier but ne comptant que pour un.*

Une variable est permanente *si elle n'est pas temporaire.*

Bien évidemment, il n'y a aucune différence entre les registres `A[i]` et `X[i]` ; physiquement ce sont les mêmes, nous les distinguerons uniquement pour bien préciser les concepts utilisés. L'avantage des temporaires réside dans le fait que certaines instructions de chargement et de récupération d'arguments pourront donner lieu à des instructions de copie

(par exemple charger `A[1]` avec le contenu du registre `X[1]`).

La figure 4 montre l'utilisation de la mémoire dans la WAM.

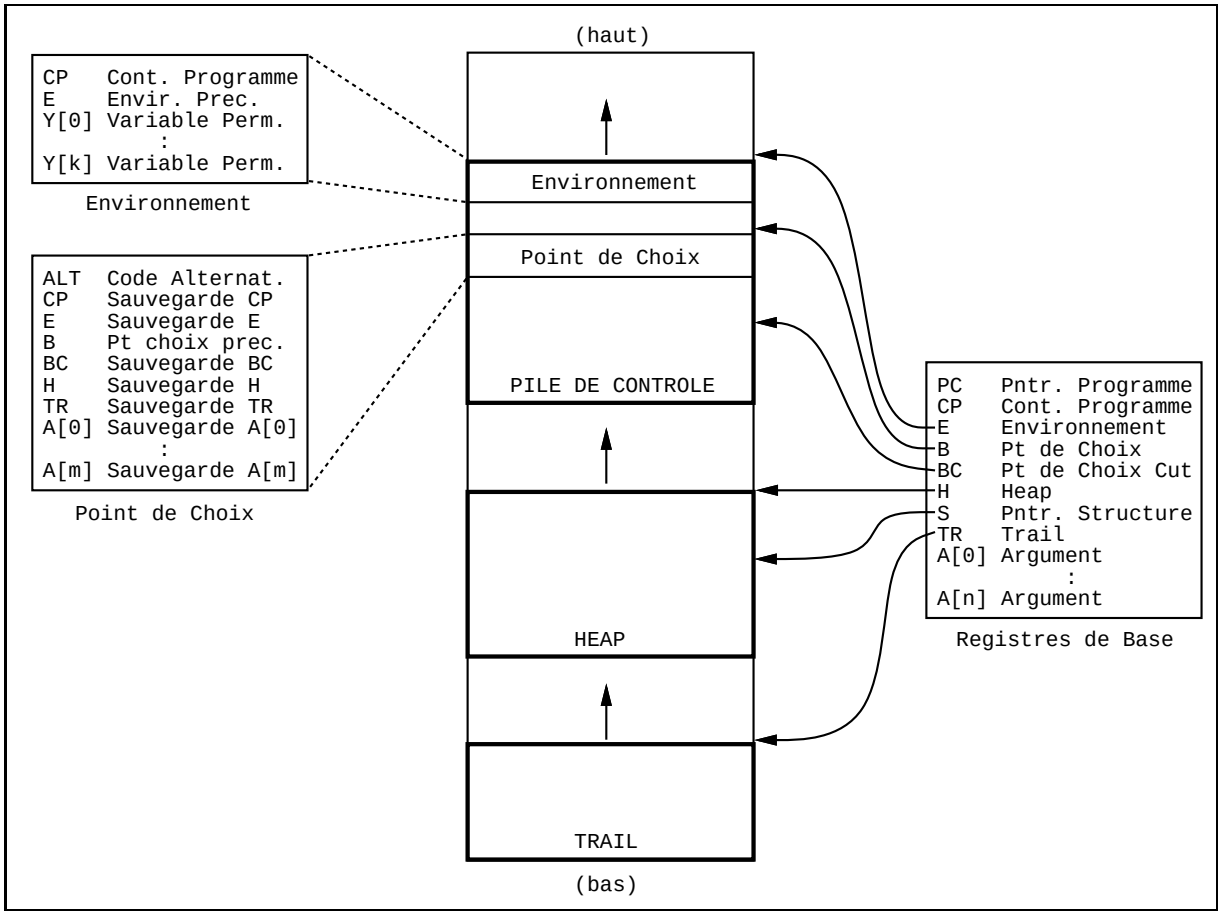


Figure 4 : architecture de la WAM

3.1.6 Economie et récupération mémoire

De par son caractère non-déterministe, Prolog est sujet à des crises de gourmandise mémoire aiguës. Il est donc important que l'architecture permette un maximum d'économie et de récupération d'espace dans les piles. Notons tout de suite que toute pile est en partie récupérée lors d'un backtracking puisque son sommet est ré-initialisé à cette occasion.

En pile locale

D'après ce qui a été dit sur les variables temporaires, les faits et les clauses n'ayant qu'un seul but ne nécessitent pas d'environnement. Pour une clause ayant plus d'un but dans son corps, l'environnement n'est indispensable que jusqu'à la fin de la clause. Ainsi, est-il possible de récupérer l'espace alloué à un environnement à la fin de la clause (après l'appel du dernier but et avant de donner le contrôle au code pointé par la continuation). Toutefois, il est possible de mieux faire. En effet, en ce qui concerne le dernier but d'une clause, les registres arguments sont chargés et il est alors possible de récupérer l'environnement *avant* d'appeler le dernier but au lieu de le faire après. Notons, que cela ne présente un intérêt que si aucun point de choix n'a été généré après cet environnement (au-dessus). Ceci permet donc, en cas de déterminisme, l'*optimisation de l'appel terminal*, comme cela se fait pour les langages fonctionnels, puisque l'espace mémoire utilisé reste constant. Toutefois, nous devons prendre garde à ce qu'aucune référence à l'environnement ne persiste après la récupération de celui-ci. Pour prévenir de telles références fantômes[2] les impératifs suivant doivent être respectés :

Dans la pile locale aucune référence ne doit exister du bas vers le haut.

Aucune référence ne doit exister de la pile globale vers la pile locale.

Attendu que les termes sont dans le heap, le seul cas pouvant engendrer des liaisons vers le haut en pile locale est dû à des liaisons entre variables permanentes. La règle qui consiste à lier la variable la plus récente vers la plus ancienne (déjà vue pour limiter les variables mises en trail) convient tout à fait. Etudions le problème concernant la liaison (d'une variable) du heap avec un élément de la pile locale. Si cet élément est lui-même une variable alors la même règle que précédemment peut résoudre le problème à la condition de placer physiquement le heap "en-dessous" (i.e. à des adresses inférieures) de la pile locale. Si l'élément à lier n'est pas une variable, il y a simplement recopie de cet élément (du fait de la recopie de structure). Enfin, le cas où un sous-terme est lié à une variable permanente est résolu en liant la variable permanente à une nouvelle variable créée sur le tas (on parle alors de *globalisation* de la variable permanente).

Toutefois, il subsiste un problème lors de la récupération de l'environnement. En effet, considérons une variable permanente d'adresse α telle que nous ne puissions nous assurer,

[2]traduction de dangling reference.

de manière statique (i.e. à la compilation), qu'elle ne sera pas libre lors de l'appel du dernier but. Si c'est le cas, le chargement dans un registre `A[i]` se traduira par : `A[i]:=<REF,`α`>`. Or, le mot d'adresse α sera libéré et, lors de l'appel, `A[i]` contiendra une référence fantôme. Notons, que c'est également le cas pour une variable dont la déréférence appartient à l'environnement courant. Le chargement dans un registre argument de telles variables devra prévenir ce problème en globalisant si besoin ces variables. De telles variables sont qualifiées de dangereuses :

Définition 3.4 *Une variable est* dangereuse *si elle est permanente, et que sa première occurrence n'est ni en tête ni dans une structure.*

Un variable permanente dont la première occurrence est en tête ne peut être dangereuse car l'unification de la tête la liera avec une adresse inférieure à l'environnement courant (ou avec une constante). De même une variable dont la première occurrence est dans une structure, de par la représentation des termes et du fait qu'il ne peut y avoir de liaison du heap vers la pile locale, ne pourra référencer une adresse de l'environnement courant.

Warren a généralisé l'optimisation de l'appel terminal par la technique du "tassage continu" (trimming) qui consiste en une récupération de l'environnement au fur et à mesure des appels des buts du corps de la clause. En effet, les variables permanentes n'ont pas toutes la même durée de vie. Une fois appelé le but où elles figurent pour la dernière fois, le mot qui leur est associé peut être récupéré. Toutefois cette technique présente peu d'avantages en comparaison des inconvénients qu'elle induit (d'autant plus qu'elle est assujettie à la même condition de déterminisme que précédemment). En effet, cette technique non seulement complique la compilation mais elle impose que les environnements soient créés "vers le haut", c'est-à-dire que `E` pointe la base de l'environnement dans lequel l'espace pour les variables permanentes est alloué, des plus utilisées au moins utilisées. Un registre supplémentaire `NB_Y` indiquant le nombre de variables permanentes encore en activité doit toujours être initialisé avant l'appel d'un but. Dans un souci d'homogénéité, les points de choix sont également représentés vers le haut et doivent stocker l'arité du prédicat ayant donné lieu à une alternative (i.e. nombre de registres `A[i]` sauvegardés). Ainsi, lorsqu'un nouvel environnement ou un point de choix doit être créé, l'adresse physique α de l'allocation s'obtient par :

Si `B>E`
  Alors α`:=B+ARITY(B)`
  Sinon α`:=E+NB_Y`
Finsi

Nous avons préféré ne pas implanter cette généralisation et représenter les environnements et les points de choix "vers le bas", `E` et `B` pointant le premier mot libre après le bloc comme tout pointeur de pile qui se respecte ! Ainsi, le calcul de l'adresse d'allocation s'obtient simplement par :

α`:=max(B,E)`

En pile de restauration

Nous ne pouvons rien récupérer en dehors du backtracking puisque cette pile contient justement les informations nécessaires pour le prochain retour-arrière. L'occupation mémoire de cette pile est proportionnelle au nombre de points de choix. Aussi, serait-il bon de pouvoir minimiser ce nombre.

En pile globale

Là encore il n'y a pas de règle générale pour récupérer de l'espace pendant l'avancée. Toutefois, le fait de récupérer les environnements entraîne des trous dans cette pile. Prenons le cas d'une variable permanente `Y` liée à un terme `t` du heap. Si aucune autre variable n'est liée à `t`, lors de la libération de l'environnement de `Y`, les mots occupés dans le heap pour `t` sont désormais inutiles. Un "garbage-collector" peut donc être mis en oeuvre sur la pile globale. Le fait que la pile globale ne puisse être récupérée qu'au backtracking a conduit Warren à apporter une restriction à la notion de variables temporaires. En effet, la gestion de certaines variables au travers des registres a pour inconvénient majeur de ne pas pas offrir de notion d'adresse du mot stockant la variable. Supposons alors que la première occurrence d'une telle variable apparaisse dans l'appel d'un des buts du corps. Cette variable devra donc être chargée dans un registre et pour cela elle devra être allouée à l'adresse α de manière à pouvoir effectuer l'affectation : `A[i]:=<REF,`α`>`. Du fait que nous avons considéré cette variable comme temporaire, seule la pile globale est susceptible

de l'accueillir (l'environnement courant n'ayant pas prévu de place pour elle). Warren a décidé de classer de telles variables comme permanentes et a donc apporté des restrictions à la définition des temporaires. Les résultats empiriques ayant montré un gain mémoire marginal (sûrement inférieur aux progrès réalisés en termes de configurations mémoires disponibles depuis 1983) nous n'avons pas adopté ces restrictions, privilégiant à nouveau la simplicité de compilation.

Indexation

Comme nous l'avons vu précédemment pour l'optimisation de l'appel terminal et pour l'occupation de la trail, il est souhaitable de diminuer le nombre de points de choix crées. Le principe de l'indexation consiste à partitionner l'ensemble des clauses d'un même prédicat en fonction des valeurs possibles de certains arguments (clés) et de générer des instructions de gestion de points de choix pour chaque partition indépendamment. A l'exécution, la partition à utiliser est déterminée en fonction de la valeurs des arguments. En ne s'occupant pas des autres partitions nous diminuons les chances de créer des points de choix. Dans la WAM, la clé d'indexation est constituée par le foncteur principal du premier argument de la tête qui peut être :

- une variable,
- un entier,
- une constante (y compris la constante de liste vide),
- une liste (non vide),
- un terme structuré.

En ce qui concerne les constantes, les entiers et les termes structurés, une sous-partition est réalisée suivant la valeur impliquée.

3.1.7 Le jeu d'instructions

Du fait de l'utilisation de registres arguments pour assurer l'échange des données, un prédicat peut être compilé de manière indépendante de tout contexte. Ainsi, l'unité de compilation est le prédicat. De plus, la compilation d'une des clauses du prédicat ne nécessite aucune connaissance des autres clauses. Seules les instructions de gestion des points de choix (i.e. indexation) nécessitent la "vision globale" de toutes les clauses. La principale caractéristique du jeu d'instruction de la WAM est basée sur l'étude statique du prédicat pour déterminer a priori (statiquement à la compilation) des situations qui sans cela ne seraient détectées qu'a posteriori (dynamiquement à l'exécution). Pour chaque cas détectable une séquence d'instructions adaptée est générée. Par exemple, la gestion des points de choix est prise en charge par des instructions spécialisées (création du point de choix par la première clause, mise à jour par les autres excepté la dernière qui se charge de sa suppression). De même l'unification est décomposée en fonction des arguments de la clause pour éviter l'appel à la fonction générale d'unification entre deux termes quelconques.

Les instructions de la WAM peuvent être classées en quatre groupes :

- instructions de récupération de registres (unification),
- instructions de chargement de registres,
- instructions de contrôle,
- instructions d'indexation (gestion des points de choix),

Ce découpage correspond à la présentation que nous allons faire du jeu d'instructions. Celle-ci sera ascendante : partant de la compilation de la tête de clause et des différents buts du corps nous verrons comment les instructions de contrôle permettent d'agencer les codes obtenus. Après quoi, nous expliquerons comment relier les instructions produites pour chaque clause, grâce aux instructions d'indexation, de manière à prendre en charge la gestion des points de choix.

Instructions de récupération de registres

Ces instructions sont produites par la compilation de la tête. Rappelons que l'unification d'une tête de clause avec les arguments a deux buts distincts quoique réalisés conjointement : vérifier que la clause est utilisable et initialiser ses variables. En particulier, une variable *singleton*[3] ne demande aucun traitement puisque elle est unifiable avec tout terme et qu'il ne sert à rien de la renseigner du fait qu'elle n'a qu'une seule occurrence.

Comme indiqué précédemment, Warren a décomposé l'unification pour des raisons évidentes de performances. En effet, lors de la compilation il est possible de distinguer les cas suivant pour chaque argument de la tête :

- première occurrence d'une variable (donc pas encore liée),
- autre occurrence d'une variable, (donc liée),
- constante,
- entier,
- liste vide (traitée comme une constante particulière),
- liste non vide (elle contient un Car et un Cdr),
- structure.

Dans le cas de termes composés, l'unification des sous-termes est également décomposée grâce à des instructions spécifiques. Dans la présentation qui suit nous notons le ième registre `A` plutôt que `A[i]` pour des raisons de clarté. Quand à l'écriture `V`, elle dénote une variable temporaire (i.e. `X[j]` ou permanente (i.e. `Y[j]`). Le code WAM généré pour unifier le ième registre avec le ième argument de la tête dépend de ce dernier comme suit :

- première occurrence d'une variable `V` :
  `get_variable(V,A)`
- autre occurrence d'une variable `V` :
  `get_value(V,A)`

[3] variable n'ayant qu'une seule occurrence dans la clause.

- constante `C` :
  `get_constant(C,A)`
- entier `N` :
  `get_integer(N,A)`
- liste vide :
  `get_nil(A)`
- liste non vide :
  `get_list(A)`
  `unify_...` (unification du Car)
  `unify_...` (unification du Cdr)
- structure `F/N` :
  `get_structure(F/N,A)`
  `unify_...` (unification du premier sous-terme)
  ⋮
  `unify_...` (unification du dernier sous-terme)

L'instruction `get_variable(V,A)` effectue une simple copie de `V` dans `A` (ce qui peut donner lieu à une optimisation si `V` est une variable temporaire). L'instruction `get_value(V,A)` applique la fonction d'unification sur `V` et `A`.

L'instruction `get_constant(C,A)` vérifie que `A` est lié à la constante `C` ou à une variable libre, auquel cas elle lie celle-ci à `C`. Les instructions `get_integer(N,A)` procède de manière similaire.
L'instruction `get_nil(A)` est une abréviation de `get_constant([],A)`.

La récupération d'un terme structuré utilise les instructions spécialisées `unify_...`. L'unification relative à un terme composé a deux comportements distincts suivant la nature de l'argument à unifier avec le terme :

- si c'est un terme de même foncteur et de même arité, alors une vraie unification doit avoir lieu sur les sous-termes (mode READ).
- si c'est une variable libre, il y a création du terme sur le heap et liaison de la variable à celui-ci (mode WRITE).

L'instruction `get_list(A)` déréférence `A` ; si le mot obtenu est une variable libre elle est liée à une liste créée sur le heap (les instructions `unify_...` créeront le Car et le Cdr). Si le mot est une liste, les instructions `unify_...` unifieront le Car et le Cdr.
L'instruction `get_structure(F/N,A)` s'exécute de manière similaire.

Le code associé à la compilation d'un sous-terme dépend de sa nature comme suit :

- première occurrence d'une variable `V` :
  si `V` n'est pas une variable singleton : `unify_variable(V)`
  sinon, soit `K` le nombre de variables singleton successives : `unify_void(K)`

- autre occurrence d'une variable `V` :
  s'il est possible de déterminer statiquement si sa première occurrence a conduit à une liaison avec le heap (i.e. première occurrence de `V` en structure, ou, dans le cas d'une temporaire, première occurrence en corps, du fait de la globalisation que cela a entraîné) : `unify_value(V)`
  sinon (nous devons éviter de lier le heap vers la pile locale) : `unify_local_value(V)`

- constante `C` :
  `unify_constant(C)`

- entier `N` :
  `unify_integer(N)`

- liste vide :
  `unify_nil`.

L'instruction `unify_variable(V)` lie `V` à `*S`[4] en mode READ et à une variable libre empilée sur le heap en mode WRITE. L'instruction `unify_value(V)` unifie `V` à `*S` en mode READ et empile `V` sur le heap en mode WRITE. Dans ce cas, il faut être certain que cela n'entraînera pas de liaison du heap vers la pile locale. C'est la raison d'être de l'instruction `unify_local_value(V)` qui, en mode READ, opère comme `unify_value(V)` et, en mode WRITE, commence par déréférencer `V` ; si le mot obtenu est une variable libre permanente alors il est nécessaire de la globaliser sinon il y a empilement de ce mot (et non pas de `V`) sur le heap (cf. remarque au sujet de `put_unsafe_value(V,A)` ci-dessous).

[4]Nous désignerons par `*S` le mot contenu à l'adresse pointée par `S`.

L'instruction `unify_void(K)` permet d'optimiser les variables singleton dans les structures. En mode READ cette instruction ajoute `K` à `S`, en mode WRITE, elle empile `K` variables libres sur le heap.

L'instruction `unify_constant(C)`, en mode READ, est similaire à `get_constant(C,*S)`, et, en mode WRITE, empile la constante `C` sur le heap. L'instruction `unify_integer(N)` procède de manière similaire.
L'instruction `unify_nil` est une abréviation de `unify_constant([])`.

Nous pouvons remarquer qu'il n'y a pas d'instructions d'unification propres aux sous-termes composés. En récupération de registres, ils sont unifiés avec de nouvelles variables temporaires `X'` par `unify_variable(X')`, celles-ci étant ensuite décomposées par les instructions `get_...(...,X')`. Ceci revient à aplatir les termes.
Par exemple : `A[i]` $\rightarrow$ `f(g(Z,a))` revient à `A[i]` $\rightarrow$ `f(X')` et `X'` $\rightarrow$ `g(Z,a)`.

En ce qui concerne le chargement de registres, les nouvelles variables temporaires sont chargées par les instructions `put_...(...,X')` avant d'être unifiées (dans le terme composé) par `unify_value(X')`. Ici aussi cela revient à une mise à plat des termes.
Par exemple : `A[i]` $\leftarrow$ `f(g(Z,a))` revient à `X'` $\leftarrow$ `g(Z,a)` et à `A[i]` $\leftarrow$ `f(X')`.

Instructions de chargement de registres

Comme pour les instructions de récupération, nous distinguerons le type de l'argument à charger. Il sera nécessaire de prendre en compte les variables dangereuses pour prévenir le problème de références fantômes. Ici encore l'instruction dépend du ième argument d'un but à charger comme suit :

- première occurrence d'une variable `V` :
  `put_variable(V,A)`
- autre occurrence d'une variable `V` :
  si `V` n'est pas dangereuse ou si le but courant n'est pas le dernier : `put_value(V,A)`
  sinon : `put_unsafe_value(V,A)`
- constante `C` :
  `put_constant(C,A)`

- entier `N` :
  `put_integer(N,A)`
- liste vide :
  `put_nil(A)`
- liste non vide :
  `put_list(A)`
  `unify_...` (chargement par recopie du Car)
  `unify_...` (chargement par recopie du Cdr)
- structure `F/N` :
  `put_structure(F/N,A)`
  `unify_...` (chargement par recopie du premier sous-terme)
  ⋮
  `unify_...` (chargement par recopie du dernier sous-terme)

L'instruction `put_variable(V,A)` initialise `V` et `A` avec une variable libre si `V` est permanente sinon elle lie `V` et `A` à une variable libre empilée sur le heap.

L'instruction `put_value(V,A)` effectue une simple copie de `V` dans `A` (ce qui peut donner lieu à une optimisation si `V` est une variable temporaire). Dans le cas d'une variable permanente du dernier but, nous devons être certain que la copie ne liera pas `A` à l'environnement courant, pour pouvoir récupérer l'environnement de façon sûre. L'instruction `put_unsafe_value(V,A)` prend en charge les variables dangereuses (susceptibles de créer des références fantômes). Cette instruction déréférence `V` et teste si le mot obtenu pointe vers l'environnement courant (i.e. le mot est `<REF,`α`>` avec $\alpha >$ `E`). Dans l'affirmative, il y a globalisation de la variable, sinon `put_unsafe_value(V,A)` copie ce mot (et non pas `V`) dans `A`. Donc `put_unsafe_value(V,A)` ne se comporte jamais comme `put_value(V,A)` puisqu'elle copie le mot déréférencé dans `A` alors que `put_value(V,A)` y copie `V`. Une variable dangereuse ayant n occurrences dans le dernier but nécessitera n instructions `put_unsafe_value(V,A)`. La première effectuera l'éventuelle globalisation et les autres copieront le mot déréférencé.

L'instruction `put_constant(C,A)` initialise `A` avec la constante `C` et `put_integer(N,A)` procède de manière similaire.
L'instruction `put_nil(A)` est une abréviation de `put_constant([],A)`.

L'instruction `put_list(A)` initialise `A` avec `<LST,H>` et le mode à WRITE de manière à ce que les instructions `unify_...` qui suivent recopient le Car et le Cdr sur le heap.

L'instruction `put_structure(F/N,A)` effectue un traitement semblable.

Instructions de contrôle

Le rôle de ces instructions est de gérer les appels et retours de procédure ainsi que les environnements. De par la définition de variables permanentes, les faits et les clauses dont le corps est réduit à un seul but ne nécessitent pas d'environnement. De plus, l'appel du dernier but est distingué des autres dans la mesure où il doit se charger du retour. En fait, une instruction de retour existe (pour les faits), et nous pourrions considérer l'appel du dernier but comme un appel quelconque suivi de l'instruction de retour, mais ce serait moins performant. Tout ceci conduit aux instructions de contrôle suivantes :

- pour un fait `p(...).` :
  < récupération des registres >
  `proceed`

- pour une clause `p(...):- q(...).` :
  < récupération des registres >
  < chargement des registres pour le but `q` >
  `execute(q)`

- pour une clause `p(...):-` q_1`(...),` q_2`(...), ...,` q_k`(...).` :
  `allocate(N)`
  < récupération des registres >
  < chargement des registres pour le but q_1 >
  `call(`q_1`)`
  < chargement des registres pour le but q_2 >
  `call(`q_2`)`
  ⋮
  < chargement des registres pour le but q_k >
  `deallocate`
  `execute(`q_k`)`

L'instruction `allocate(N)` créée un environnement pour `N` variables permanentes. L'instruction `deallocate` permet de récupérer cet environnement.

L'instruction `call(P/N)` initialise `CP` à l'adresse qui suit le `call` et donne le contrôle au prédicat `P/N`. L'instruction `execute(P/N)` procède pareillement sans toutefois modifier le contenu de `CP` préalablement restauré par l'instruction `deallocate`.

Le retour de procédure est assuré par l'instruction `proceed` qui se contente donc d'affecter `CP` à `PC`.

Instructions d'indexation

Ces instructions permettent de regrouper le code de chaque clause d'un prédicat et sont donc les instructions de plus haut niveau. Elles ont la responsabilité de gérer les points de choix. Elles peuvent créer jusqu'à deux points de choix à l'entrée d'un prédicat. C'est pour cette raison que l'on parle *d'indexation à deux niveaux*. Des indexations à un seul niveau sont évidemment possibles mais occupent plus de place. Ces niveaux proviennent du fait qu'un premier argument de tête de clause qui est une variable n'est pas discriminant pour l'indexation.

- Niveau 1 :
  Les clauses $C_1, ..., C_n$ sont éclatées en groupes $G_0, ..., G_m$ tels que chaque groupe G_i ne contienne qu'une clause dont le premier argument est une variable (a) ou que des clauses dont le premier argument n'est pas une variable (b).

  Le code suivant est alors généré :

  si $m = 0$:

  $<$ code pour G_0 $>$

  sinon

  `try_me_else(`L_1`)`

  $<$ code pour G_0 $>$

  L_1: `retry_me_else(`L_2`)`

  $<$ code pour G_1 $>$

  $\vdots$

L_m: `trust_me_else_fail`
< code pour G_m >

L'instruction `try_me_else(`L_{else}`)` a la charge de la création d'un point de choix dans lequel elle désigne comme alternative le code d'adresse L_{else}. L'instruction `retry_me_else(`L_{else}`)` a pour rôle de restaurer les registres de base et de mettre à jour le point de choix en précisant que la nouvelle alternative est L_{else}. Enfin, l'instruction `trust_me_else_fail` restaure les registres de base et supprime le point de choix. Pour ces trois instructions le contrôle se poursuit par l'instruction suivante du code.

- Niveau 2 :
  Pour un groupe G_i du type (a), le < code pour G_i > ne contient que le code de son unique clause. Pour un groupe de type (b) il contient les instructions de niveau 2 suivantes :

  `switch_on_term(`L_{var}`,`L_{cte}`,`L_{int}`,`L_{lst}`,`L_{stc}`)`

  s'il n'y a pas de constantes alors L_{cte}=`fail`, sinon le code suivant est généré :

  L_{cte}: `switch_on_constant(N,[(cte`$_1$`,`L_{cte_1}`),...,(cte`$_N$`,`L_{cte_N}`)])`

  pour chaque constante cte_j $(j = 1..N)$ le code suivant est généré :
  si une seule clause a cte$_j$ comme 1er argument alors L_{cte_j}=L_{j_1} (j_i étant le numéro de la $i$ème clause ayant `cte`$_j$ comme premier argument) sinon :

  L_{cte_j}: `try(`L_{j_1}`)`
  `retry(`L_{j_2}`)`
  ⋮
  `trust(`L_{j_k}`)`

  idem pour les entiers (L_{int}) et les structures (L_{stc}).

  s'il n'y a pas de listes alors L_{lst}=`fail`, sinon le code suivant est généré :
  (où j_i est le numéro de la $i$ème clause ayant une liste comme premier argument) :

L_{lst}: `try(`L_{j_1}`)`
`retry(`L_{j_2}`)`
⋮
`trust(`L_{j_k}`)`

Si G_i contient une seule clause `L`$_{var}$`=L`$_1$ sinon :

L_{var}: `try_me_else(`L_{var_2}`)`
L_1: < code clause 1 >

L_{var_2}: `retry_me_else(`L_{var_3}`)`
L_2: < code clause 2 >
⋮

L_{var_p}: `trust_me_else_fail`
L_p: < code clause p >

L'instruction `switch_on_term(`L_{var}`,`L_{cte}`,`L_{int}`,`L_{lst}`,`L_{stc}`)` donne le contrôle à l'adresse appropriée suivant le type du mot auquel `A[0]` est lié.

L'instruction `switch_on_constant(N,[(cte`$_1$`,`L_{cte_1}`),...,(cte`$_N$`,`L_{cte_N}`)])` gère le contrôle suivant le type de constante grâce à une table qui, à une constante `cte`, associe une adresse `L`. `L` correspond à l'adresse de la clause dont le premier argument est `cte` ou à celle d'un code de niveau 2 (`try`, `retry`, `trust`) dans le cas où plusieurs clauses ont `cte` comme premier argument de la tête. Idem pour les entiers, les listes et les structures.

Les instructions `try(L)`, `retry(L)` et `trust(L)` ont respectivement le même rôle que `try_me_else(`L_{else}`)`, `retry_me_else(`L_{else}`)` et `trust_me_else_fail` ; la seule différence réside dans l'alternative stockée dans le point de choix : pour `try(L)`, `retry(L)` et `trust(L)` c'est l'instruction suivante et non pas L_{else}. Le contrôle, pour sa part, se poursuit par l'instruction d'adresse `L`.

La table 3 montre le code compilé du prédicat `p/3` utilisé en figure 1.

```
p/3:  switch_on_term(L3,fail,L1,fail,L2)

L1:   switch_on_integer(1,[(0,L4)])

L2:   switch_on_structure(1,[(s/1,L6)])

L3:   try_me_else(5)

L4:   get_integer(0,X[0])          % p(0,X,X).
      get_value(X[1],X[2])
      proceed

L5:   trust_me_else_fail

L6:   get_structure(s/1,X[0])      % p(s(X),Y,s(Z)):- p(X,Y,Z).
      unify_variable(X[0])
      get_structure(s/1,X[2])
      unify_variable(X[2])
      execute(p/3)
```

Tableau 3 : exemple de code WAM

3.2 Exécution du code WAM : traduction vers C

Nous étudierons dans cette section diverses solutions pour exécuter le code WAM et celle choisie pour `wamcc` : traduire Prolog vers C. Ce choix sera justifié par rapport aux objectifs que doit remplir `wamcc` et que nous définissons tout de suite.

3.2.1 Cahier des charges

En 1991, nous avons décidé de développer un Prolog "maison" dans le but d'expérimenter diverses extensions. En effet, l'étude de Sicstus Prolog nous a vite convaincu du travail colossal que représentait la modification d'une implantation "professionnelle". Nous avons donc défini un cahier des charges pour `wamcc` comprenant les points suivants (par ordre de priorité) :

- *evolutivité* : puisque ce Prolog devait servir de plate-forme expérimentale. Ceci impliquait donc un moteur Prolog simple ne pouvant bénéficier d'optimisations poussées

qui font grossir le code (ex. l'émulateur de Sicstus ne nécessite pas moins de 35000 lignes de C).

- *portabilité* : de manière à permettre une large diffusion et éviter l'obsolescence inhérente à l'attachement à une machine donnée.

- *efficacité* : pour ne pas "disqualifier" les futures extensions, en particulier les contraintes, sous le prétexte que le Prolog de base n'est pas assez efficace. En effet, bien souvent un langage est d'abord jugé sur les (contre-)performances qu'il offre sur les "benchmarks" classiques et ensuite seulement sur les facilités d'expressions et les nouveautés qu'il offre. Ainsi, nous nous étions fixés comme objectif d'être aussi rapide que la version 0.6 de Sicstus, celle-ci servant alors de mesure-étalon pour l'évaluation des performances. Cette version était basée sur un émulateur très optimisé écrit en C.

- *support de la modularité* : permettant ainsi de découper une application Prolog en plusieurs modules. Ceci permet, dans un premier temps, de ne pas s'occuper des *prédicats prédéfinis* (built-ins) qui seront écrits en Prolog par la suite dans des modules séparés.

3.2.2 Méthodes classiques pour exécuter la WAM

Le code WAM généré par le compilateur nécessite un traitement pour être réellement exécutable. Il existe trois méthodes classiques pour exécuter du code WAM :

- utiliser une machine Prolog,

- utiliser un émulateur de code WAM,

- produire du code natif pour la machine hôte.

En ce qui concerne une machine Prolog, le code fourni par la compilation est directement exécutable puisque ce code correspond au langage "assembleur" d'une telle machine. Toutefois, comme dans toute machine, il n'est pas possible de modifier ce langage de plus bas niveau donc nous ne pourrions étendre la WAM pour prendre en compte les contraintes.

L'émulation est généralement la première solution qui vient à l'esprit. Dans cette approche le code produit (*byte-code*) est simplement vu comme des *données* qui seront traitées par l'émulateur. Les avantages d'un émulateur résident dans sa simplicité d'écriture (3000 lignes de C pour une version non optimisée), dans sa portabilité s'il est écrit en C et dans sa facilité à créer et modifier dynamiquement du code WAM. Ces avantages ont convaincu la plupart des concepteurs de Prolog. L'inconvénient majeur de cette approche est évidemment le surcoût de l'émulation puisque, pour chaque instruction, nous n'échappons pas au cycle : *recherche*, *décodage* et *exécution*. Un autre inconvénient provient du fait qu'il est difficile de fournir un exécutable autonome puisque l'émulateur doit être présent au moment de l'exécution, posant ainsi des problèmes de distribution liés aux droits commerciaux limitant la diffusion d'un produit professionnel.

La production de code natif a des caractéristiques inverses de celles de l'émulation. L'écriture d'un générateur de code natif est une tâche difficile qui requiert une connaissance approfondie de la machine cible. C'est à cette condition que les performances sont au rendez-vous (en particulier sur les processeurs RISC). Bien entendu, de tels compilateurs ne sont pas portables et l'adaptation à une autre architecture n'est pas si aisée ; là où une machine peut être efficace telle autre peut avoir un comportement très médiocre. D'un autre côté l'écriture d'un compilateur produisant du code natif comprend bon nombre de parties classiques pénibles à développer bien que les techniques adéquates soient connues (ex. allocateur de registres).

3.2.3 La solution adoptée : traduire Prolog vers C

Malheureusement, les objectifs de notre cahier des charges ne coïncidaient ni avec les caractéristiques de l'émulation ni avec celles de la production de code natif. Aussi, avons-nous pensé à la traduction de Prolog vers C qui, sur le papier, devait permettre de bénéficier des avantages des deux approches. Ainsi, la portabilité serait assurée et, en fin de compte, du code natif serait produit (par le compilateur C) ce qui devait compenser le manque d'optimisations par l'absence des phases de recherche et de décodage. En ce qui concerne la modularité, celle de C permettrait de compiler des programmes Prolog séparemment et de les réunir au moment de l'édition de liens (les problèmes de visibilité des prédicats étant résolus par les déclarations C appropriées). Enfin, le fait de compiler vers C permettrait

de bénéficier d'interfaces simples avec ce langage aussi bien qu'avec tout autre langage compilé et fournirait de "vrais" fichiers exécutables.

Toutefois, nous ne disposions au moment de ce choix (1991) d'aucune référence dans le domaine de la traduction de Prolog vers C. Aujourd'hui encore, trois ans après, aucune implantation de Prolog basée sur une traduction vers C n'est disponible en dehors de `wamcc`. En revanche, dans ce laps de temps, cette approche à été adoptée pour implanter trois langages logiques déterministes avec gardes[5] : Janus [39], KL1 [19] et Erlang [42]. Ils utilisent tous des schémas de traduction différents qu'il est intéressant de comparer à celui utilisé dans `wamcc` en ce qui concerne Prolog "déterministe".

3.2.4 Le problème du contrôle de Prolog en C

Dans cette section nous nous concentrerons uniquement sur la manière dont Janus, KL1, Erlang et `wamcc` gèrent le contrôle. En effet, celui-ci est le point crucial de la traduction vers C du fait que le code WAM est un code "à plat" où tous les transferts de contrôle sont effectués à l'aide de simples branchements. Ce contrôle a du mal à s'accommoder de celui de C prévu au contraire pour supporter les structures de haut niveau que constituent blocs et fonctions et, paradoxalement, peu fourni pour supporter du contrôle de bas niveau. Ainsi, le problème majeur consiste à trouver un moyen efficace de traduire les branchements WAM. Notre présentation se basera sur l'exemple suivant :

```
p:  allocate    /* p:- q, r. */
    call(q)
    deallocate
    execute(r)

q:  proceed     /* q. */
```

Ce petit exemple met en oeuvre toutes les instructions de contrôle de Prolog dans le cas déterministe. Notamment la manière de traduire les instructions `call` et `execute` fera apparaître comment sont gérés les branchements directs (i.e. branchement à un label précisé) alors que la traduction de l'instruction `proceed` devra résoudre le problème des branchements indirects (i.e. branchement à l'adresse pointée par une variable, le registre `CP` en l'occurrence).

[5] classiquement appelés *commited-choice languages.*

3.2.5 La méthode Janus

Dans méthode, un branchement WAM est traduit par un branchement `goto` C. Le fait qu'un `goto` ne puisse adresser que du code à l'intérieur d'une même fonction, conduit à un programme C composé d'une seule fonction dotée d'une instruction `switch` pour simuler les `goto` indirects. Ainsi, notre exemple donnera lieu à :

```
fct_switch()
{
 label_switch:
   switch(PC) {
      case p:                 /* p:- q,r . */
      label_p:
         push(CP);            allocate
         CP=p1;               call(q)
         goto label_q;          :

      case p1:
         pop(CP);             deallocate
         goto label_r;        execute(r)

      case q:                 /* q. */
      label_q:
         PC=CP;               proceed
         goto label_switch;     :
      ⋮
   }
}
```

Cette méthode est assez efficace, mais a pour inconvénient majeur qu'un programme donne lieu à une seule fonction. Dès lors, hormis pour les programmes "jouets" de quelques clauses, cette méthode engendre une fonction énorme que le compilateur C ne peut alors traiter en un temps raisonnable (cf. [62]). Nous attribuons cela au temps (exponentiel) que nécessite l'allocateur de registres. Par ailleurs, la gestion de la modularité est difficile à mettre en oeuvre du fait qu'elle nécessite la consultation d'une table dynamique pour appeler un prédicat d'un autre module en vue de donner le contrôle à la fonction "switch" de ce module. De même, la gestion correcte du backtracking lors de ces changements de contexte n'est pas une tâche facile. Ainsi, dans le cas où la modularité est supportée, un appel extra-modules sera beaucoup plus coûteux qu'un appel intra-modules.

3.2.6 La méthode KL1

Puisque la solution avec une seule fonction n'est pas réalisable, il faut découper le fichier C (i.e. le code WAM) en plusieurs fonctions. L'idée de traduire un prédicat par une fonction semble alors naturelle. La situation est la suivante : nous ne désirons effectuer que des branchements et le découpage en fonctions est obligatoire. Dès lors, il faut exécuter des appels de fonctions pour traduire les branchements. Toutefois cela ne suffit pas puisque, avant la fin d'une fonction appelée, un autre branchement (i.e. un autre appel de fonction) aura lieu ayant pour effet de ne jamais retourner des fonctions invoquées (du point de vue de C il n'y a que des appels de fonction imbriqués). Le problème est alors que les données de contrôle empilées par C ne sont jamais récupérées. Il est donc nécessaire que chaque fonction termine "proprement" par une instruction de retour. A charge à une fonction *superviseur* d'effectuer le branchement prévu (i.e. continuation). Ceci conduit au code suivant pour notre exemple :

```
fct_supervisor()
{
   while(PC)
      (*PC)();
}

void fct_p()       /* p:- q,r. */
{
   push(CP);       allocate
   CP=fct_p1;      call(q)
   PC=fct_q;         :
}

void fct_p1()
{
   pop(CP);        deallocate
   PC=fct_r;       execute(r)
}

void fct_q()       /* q. */
{
   PC=CP;          proceed
}
```

Notons que le code ci-dessus peut être optimisé en supprimant le registre `PC` puisque l'information qu'il véhicule peut être retournée par les fonctions. Ainsi, une fonction effectue un calcul "en ligne" et lorsqu'un branchement est requis elle se termine en retournant l'adresse du code à qui le contrôle doit être donné.

L'analyse de cette méthode montre qu'un branchement WAM donne lieu à un retour de fonction (au superviseur) suivi d'un appel de fonction. Ce qui est bien évidemment beaucoup plus coûteux qu'une instruction de branchement assembleur générée par un compilateur "code natif". L'appel extra-modules est désormais possible et n'est pas plus pénalisant qu'un appel intra-modules. La première version de `wamcc` utilisait cette approche et les performances obtenues étaient environ deux fois moindres que celles de Sicstus 0.6. En ce qui concerne KL1, un compromis à été trouvé pour diminuer les appels et retours de fonctions. En effet, tous les prédicats d'un même module sont placés dans une seule fonction. Ainsi, en présence d'un seul module KL1 à un comportement similaire à Janus. Dans ce cas, la fonction superviseur, ne sert qu'à changer de contexte lors des appels extra-modules. Ceux-ci devenant alors plus coûteux que les appels intra-modules. Notons toutefois que cette méthode (avec ou sans l'optimisation "KL1") est la plus viable en restant 100% ANSI C.

3.2.7 La méthode Erlang

Ici aussi un prédicat donne lieu à une fonction C. Toutefois, pour éviter les appels de fonction, Erlang tire profit des possibilités offertes par le compilateur GNU C (gcc) en matière d'étiquettes. En effet, gcc considère celles-ci comme des objets de première classe et permet donc de stocker l'adresse d'une étiquette dans une variable pointeur et d'effectuer, par la suite, un branchement indirect à la valeur pointée par une telle variable. L'idée consiste donc à traduire un branchement (direct ou indirect) WAM par un branchement C indirect à l'*intérieur* (i.e. un label) d'une fonction. Une table globale des adresses de branchement est alors nécessaire et doit être initialisée par un premier appel de chaque fonction. Ainsi, notre exemple donne lieu à :

```
void fct_p()                  /* p:- q,r. */
{
   jmp_tbl[p]=&&label_p;      (initialisation)
   jmp_tbl[p1]=&&label_p1;
   return;

 label_p:
   push(CP);                  allocate
   CP=&&label_p1;             call(q)
   goto *jmp_tbl[q];            :

 label_p1:
   pop(CP);                   deallocate
   goto *jmp_tbl[r];          execute(r)
}

void fct_q()                  /* q. */
{
   jmp_tbl[q]=&&label_q;      (initialisation)
   return;

 label_q:
   goto *CP;                  proceed
}
```

Ainsi, tous les branchements se font de manière indirecte grâce à des `goto` et à une table globale des adresses. Dans le but de réduire le coût de cette indirection, Erlang procède comme KL1 et Janus, en compilant dans une seule fonction les prédicats d'un même module. Ainsi, seuls les appels extra-modules nécessitent la consultation de la table et seront donc plus coûteux que les appels intra-modules.

Notons toutefois que cette approche n'est valide que sous certaines hypothèses (non formulées dans [42]) concernant le code produit par gcc :

- il ne doit pas nécessiter de variables locales, i.e. tout le code doit pouvoir se contenter des registres. En effet, le fait de se brancher au milieu de la fonction donc de sauter le *prologue* de celle-ci, fait qu'il n'y a pas réservation d'espace dans la pile de C pour les variables locales[6]. Ce qui implique soit de n'utiliser que des variables globales soit de toujours compiler avec l'optimiseur de registres (option `-O2` de gcc) puisque sans cela

[6]rappelons que le prologue d'une fonction décrémente le pointeur de pile `sp` (la pile C pointe vers le bas) du nombre de variables locales nécessaires à la fonction

aucune variable locale n'est mise en registre. Et même dans ces deux cas rien n'assure que gcc n'aura pas besoin de variables locales pour ses calculs intermédiaires.

- aucune instruction associée au prédicat ne doit être "déplacée" avant l'étiquette de branchement (ex. `label_p`). Or il semble difficile de garantir cela. Considérons l'accès à un élément du tableau `jmp_tbl[]`. Celui-ci se traduit par une instruction de chargement de l'adresse de base du tableau dans un registre R puis par une instruction d'accès à l'élément approprié. Si plusieurs accès au même tableau ont lieu (ex. comme c'est le cas dans `label_p` et `label_p1`), le compilateur peut décider de calculer une seule fois le chargement de l'adresse de base dans le registre R et de placer cette instruction en début de fonction où il suppose qu'elle sera toujours exécutée. Le problème surviendra lorsque, par la suite, un branchement au milieu du code entraînera l'utilisation du registre R non encore initialisé. Notons que de tels déplacements de code sont courants et permettent d'optimiser le "pipe-line" des processeurs RISC (qui est à l'origine des bonnes performances de ces processeurs).

3.2.8 La méthode `wamcc`

Les trois solutions proposées jusqu'ici ont ceci en commun qu'elles se comportent similairement en présence d'un seul module, donnant alors lieu à une seule énorme fonction que le compilateur C a du mal à compiler. De plus, l'appel extra-modules, quand il est possible, est plus coûteux que l'appel intra-modules. Ainsi, la manière de découper une application en modules influence-t-elle non seulement les temps de compilations mais également les temps d'exécutions et ce, bien évidemment, de manière inversement proportionnelle.

Dès la seconde version de notre langage, notre objectif a été de réussir à ce qu'un branchement direct WAM donne lieu, en fin de compte, à un branchement direct de l'assembleur de la machine. Du fait que le découpage en fonctions est obligatoire, ces branchements doivent atteindre du code à l'intérieur d'un fonction. Le fait de vouloir produire des branchements directs implique une résolution des adresses de branchement statique (à la compilation) plutôt que dynamique (à l'exécution). Or, le couple formé par le compilateur et l'éditeur de liens sait se charger de cela en ce qui concerne les adresses de fonctions (i.e. de code). Le compilateur génère des instructions avec "trous" lorsque qu'elles référencent l'adresse d'une fonction non définie dans le source courant. L'éditeur de liens, en présence de tous

les objets, se charge alors de combler les "trous" en résolvant les adresses. La solution consiste à insérer dans l'assembleur produit par le compilateur un label en début de chaque fonction grâce à une directive `asm("...")`. Pour manipuler l'adresse d'un de ces labels, disons x, il suffit de faire croire au compilateur que x est une fonction externe. Ce qui se fait simplement en déclarant un prototype pour la fonction x et en utilisant le symbole (constante) x puisqu'en C le nom d'une fonction représente son adresse. Le compilateur génère donc une instruction avec un "trou" qui sera résolue par l'éditeur de liens à la vue des noms de labels insérés et ce qu'ils soient dans le module courant ou non. Ainsi, le coût de l'appel extra-modules est égal à celui de l'appel intra-modules. Tout ceci conduit à la structure suivante :

```
void label_p();                 /* prototypes */
void label_p1();
void label_q();
void label_r();

#define Direct_Goto(lab)        lab()
#define Indirect_Goto(p_lab)    (*p_lab)()

void fct_p()                    /* p:- q,r. */
{
 asm("label_p:");
   push(CP);                    allocate
   CP=label_p1;                 call(q)
   Direct_Goto(label_q);          :
}

void fct_p1()
{
 asm("label_p1:");
   pop(CP);                     deallocate
   Direct_Goto(label_r);        execute(r)
}

void fct_q()                    /* q. */
{
 asm("label_q: ");
   Indirect_Goto(CP);           proceed
}
```

Deux macros permettent d'effectuer les branchements directs et indirects. Le traitement effectué par celles-ci dépend de l'architecture. Par exemple sur une machine RISC :

`Direct_Goto(lab)` invoque simplement la fonction qui aurait pour nom `lab`.

`Indirect_Goto(p_lab)` invoque simplement la fonction qui aurait pour adresse la valeur de `p_lab`.

En effet, sur une machine RISC (ex. Sparc, MIPS R3000,...), l'instruction d'appel de fonction transfère le contrôle à l'adresse indiquée (comme un branchement) et initialise le registre de continuation du processeur (pour le retour). Du fait de l'architecture, cette instruction s'exécute aussi rapidement qu'un simple branchement. Comme elle n'empile rien, nous pouvons l'utiliser pour effectuer un branchement. Le fait de procéder ainsi évite de devoir insérer l'instruction de branchement dans l'assembleur. De plus, les instructions de branchement RISC ne peuvent accéder qu'à du code relativement proche de l'instruction courante (le déplacement par rapport à celle-ci étant codé sur quelques bits). Or nous pouvons accéder à du code potentiellement très éloigné du fait de la gestion des modules. L'instruction d'appel de fonction n'est pas soumise à cette limitation. Enfin, signalons que le fait que ce soit le C qui génère l'instruction d'appel de fonction lui permet d'optimiser le *delay slot* pour tirer profit du "pipe-line". Rappelons que sur certains processeurs RISC, l'instruction se trouvant après un branchement ou un appel de fonction (*delay slot*) est toujours exécutée car elle a déjà été chargée dans le pipe-line. Les compilateurs essayent d'utiliser cette particularité en déplaçant une instruction pertinente après le branchement. Quand cela n'est pas possible c'est l'instruction vide `nop` qui est émise.

Remarques récapitulatives :

- désormais les branchements directs s'effectuent aussi rapidement que possible puisqu'ils donnent lieu à de vraies instructions de branchement de l'assembleur de la machine (où, dans le cas d'un RISC, à une instruction d'appel de fonction de même coût).
- l'appel extra-modules n'est pas plus coûteux que l'appel intra-modules.
- comparée aux approches précédentes où tous les prédicats d'un même module donnaient lieu à une seule fonction, ici une clause donne lieu à autant de fonctions qu'il

y a de buts dans le corps de la clause (la tête et le premier but comptant pour un). Donc le code produit est beaucoup plus rapide à compiler (cf. section 3.4).

- les particularités de gcc ne sont pas utilisées en ce qui concerne le contrôle et un compilateur classique permettant l'insertion de code assembleur suffit.

- chaque fonction n'a qu'un seul point d'entrée "direct" qui se trouve au tout début donc seul le prologue est "sauté". Pour permettre l'utilisation de variables locales, un espace de travail suffisamment grand est réservé dans la pile C une fois pour toutes avant le lancement du calcul en définissant un tableau dans une fonction intermédiaire. Ainsi, le pointeur de pile C `sp` pointe la fin du tableau et ne sera plus modifié puisque les prologues ne seront pas exécutés. Donc, les variables locales (référencées par rapport à `sp`) seront allouées dans l'espace de ce tableau.

- la seule hypothèse concerne donc le fait que le prologue ne s'occupe que de décrementer `sp`. Or ceci est généralement le cas excepté sur quelques machines où le compilateur C référence les variables locales, non pas par rapport à `sp` mais par rapport à un autre registre `fp` (frame pointer) qui est affecté en début de fonction à la valeur de `sp`. Cette façon de procéder ne sert qu'à aider le debugger C et il est généralement possible de la désactiver grâce une option du compilateur (ex. `-fomit_frame_pointer` en gcc). Dans le cas contraire il serait possible de générer une instruction assembleur pour initialiser ce registre.

- il est tout à fait possible d'exécuter de vrais appels de fonctions à l'intérieur de ces "pseudo-fonctions". En particulier les macros associées aux instructions WAM s'expansent, pour la plupart, en appel de fonctions de la librairie de `wamcc`. Ceci permet de privilégier la taille du code, donc la vitesse de compilation, au (léger) détriment de la vitesse d'exécution.

Etudions maintenant le code nécessaire à l'amorçage du calcul qui vient d'être décrit. Soit à exécuter un premier prédicat (généralement un top-level) d'adresse `p_lab` :

```
#include <setjmp.h>

jmp_buf jumper;

void Label_Success();
void Label_Fail();

Bool Call_Prolog(WamCont p_lab)
{
 Create_Choice_Point();
 ALTB(B)=Label_Fail;
 CP=Label_Success;

 ret_val=setjmp(jumper);
 if (ret_val==0)
     Call_Next(p_lab);

 Delete_Choice_Point();
 return ret_val==2;
}
```

```
void Call_Next(WamCont p_lab)
{
 int t[1024];

 Indirect_Goto(p_lab);
}

void Call_Prolog_Success(void)
{
 asm("Label_Success:");

 longjmp(jumper,2);
}

void Call_Prolog_Fail(void)
{
 asm("Label_Fail:");

 longjmp(jumper,3);
}
```

La fonction `Call_Prolog` a pour rôle de lancer l'exécution du prédicat d'adresse `p_lab`. Cette fonction commence par créer un point de choix pour y stocker l'adresse où se brancher en cas d'échec (`Label_Fail`). Le pointeur `CP` (indiquant quel code doit être exécuté après la réussite du prédicat appelé) est initialisé à l'adresse `Label_Success`. Enfin, un `setjmp` est effectué pour faire une copie de tous les registres de la machine, permettant par la suite leurs restaurations et la reprise du calcul à l'instruction suivant le `setjmp`. Après cette sauvegarde, la fonction `Call_Next` est appelée. Le rôle de celle-ci est de réserver suffisamment d'espace de travail dans la pile C pour les possibles variables locales (cf. déclaration du tableau `t`). Le contrôle est alors donné au prédicat à appeler. Ce dernier s'exécute comme indiqué précédemment. En cas de réussite (resp. d'échec) le contrôle est transmis au code d'adresse `Label_Success` (resp. Label_Fail) qui se contente de retourner dans la fonction `Call_Prolog` grâce à un `longjmp` dont le second paramètre indique la réussite avec la valeur 2 (resp. l'échec avec la valeur 3).

3.3 Caractéristiques de wamcc

3.3.1 Processus de compilation

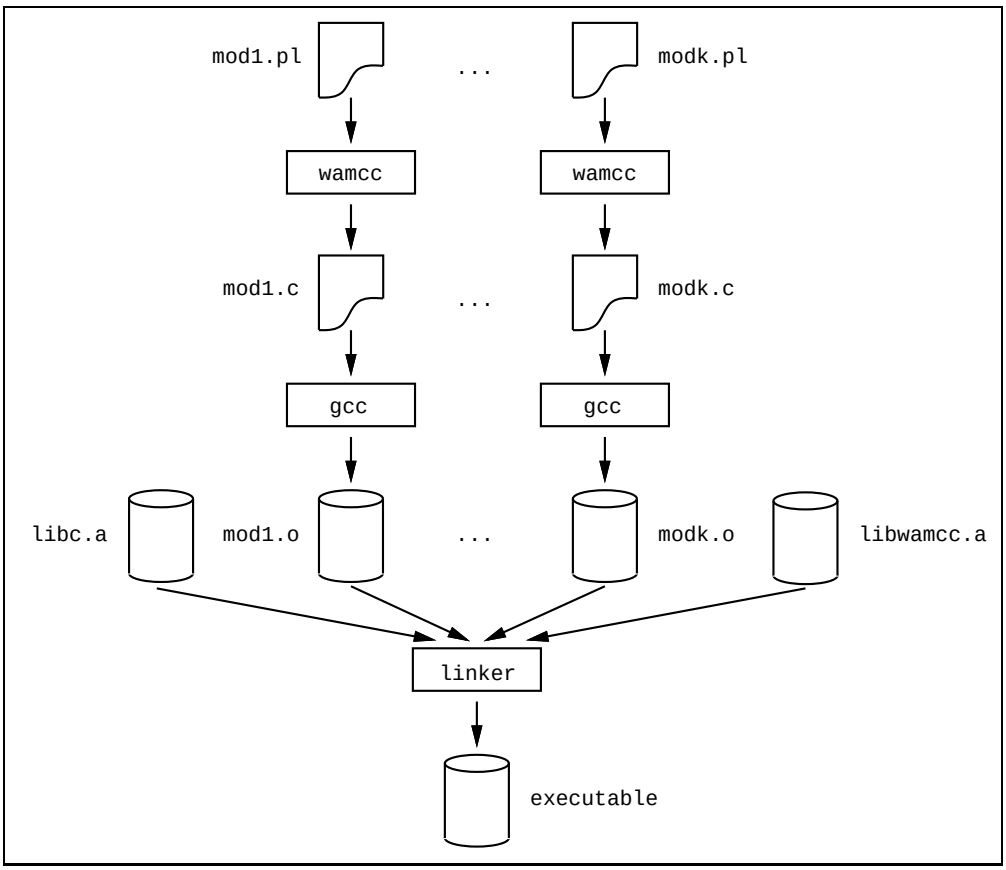


Figure 5 : processus de compilation

Pour compiler une application, les éléments suivants sont nécessaires :

- le compilateur `wamcc` qui, à partir d'un programme (module) Prolog génère le fichier source C qui lui est associé. Ce compilateur est lui-même écrit en Prolog.
- un compilateur C acceptant l'inclusion d'instructions assembleurs et sa librairie de fonctions standard. Le compilateur GNU C (gcc) convient tout à fait d'autant plus qu'il est disponible pour pratiquement toute architecture.
- un éditeur de liens (fourni avec tout système d'exploitation).
- la librairie associée à `wamcc`.

La figure 5 schématise les différentes étapes nécessaires à l'obtention d'un exécutable à partir des modules Prolog $\texttt{mod}_1$,..., $\texttt{mod}_k$. Ce schéma peut s'enrichir aisément pour intégrer, au moment de l'édition de liens, du code écrit en C ou en d'autres langages. Pour faciliter le développement d'une application, un programme est fourni avec `wamcc` permettant de générer un fichier de dépendance pour cette application. L'utilitaire `make` d'Unix peut alors prendre en charge la reconstruction de l'application en ne re-compilant que les parties nécessaires.

3.3.2 Gestion des piles

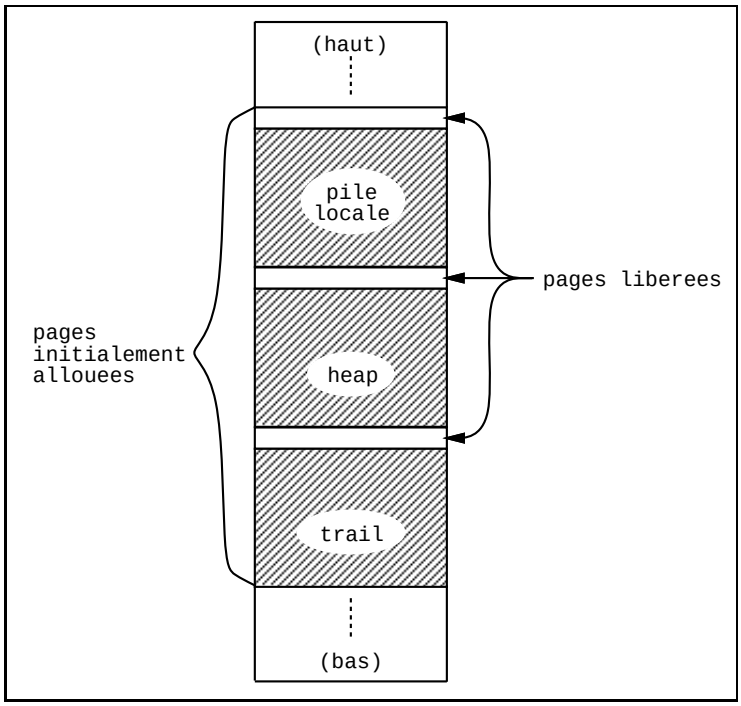


Figure 6 : disposition des piles en mémoire

Il est indispensable de contrôler l'accroissement des piles de manière à prévenir l'utilisateur lorsque l'une d'entre elles déborde. Ce contrôle est généralement effectué par le logiciel soit à chaque fois qu'une allocation est nécessaire (potentiellement plusieurs fois par clauses pour le heap), soit à l'entrée de chaque clause (mais il faut alors tester toutes les piles), soit grâce à des instructions spécialisées insérées dans le code WAM. Dans tous les cas ce contrôle est coûteux d'autant plus que le matériel des machines récentes contient tout ce qu'il faut pour le prendre en charge. En effet, rappelons que ces machines utilisent la

notion de mémoire virtuelle. C'est-à-dire que le programmeur n'a pas à s'occuper de la taille réelle ni des adresses physiques ; pour lui la mémoire est linéaire, de très grande taille (ex. 4 Go sur une machine 32 bits). Lorsqu'une donnée doit être lue/écrite, le gestionnaire de mémoire détecte si la *page* où elle réside est physiquement présente en mémoire ou non (on parle alors de *défaut de page*). Dans ce cas, le gestionnaire de mémoire la charge après avoir, si besoin, recopié une autre page sur disque. Le gestionnaire déclenche un *signal d'exception* lors d'un défaut de page alors que celle-ci n'a pas été alloué. L'idée consiste donc à déclencher un tel signal lors du débordement d'une pile. Pour cela, il suffit que toute pile soit suivie d'une page non allouée (cf. figure 6). Lors du débordement de la pile, la tentative de lecture ou d'écriture dans cette page déclenchera un signal qui sera capturé par une fonction C ayant à charge de diagnostiquer quelle pile est fautive et d'émettre un message d'erreur. La méthode la plus simple consiste à utiliser la fonction `mmap` qui permet de "mapper" un fichier en mémoire à partir d'une certaine adresse. Les lectures et écritures de ce fichier se faisant simplement en lisant et en écrivant dans la mémoire. Il existe généralement un "device" spécial (`/dev/zero`) qui rend toujours zéro lors qu'une nouvelle donnée est lue et dans lequel les écritures ne sont pas répercutées. Ceci convient tout à fait pour nos piles. Grâce à la fonction `munmap` chaque page suivant une pile est rendue au gestionnaire de mémoire. En cas d'absence des fonctions de la famille de `mmap` il est possible d'utiliser celles permettant la gestion du partage de mémoire entre processus (`shmget`,...) puisqu'elles aussi permettent d'acquérir et libérer de la mémoire. Enfin, pour les machines ne possédant aucune de ces facilités, les piles sont allouées grâce aux fonctions d'allocation de la librairie standard de C (`malloc`) et le contrôle de débordement se fait par programme à l'appel de chaque prédicat.

3.3.3 Fichiers de configuration

Pour permettre l'extensibilité et la portabilité du système, deux fichiers de configurations sont utilisés, l'un pour décrire la WAM et l'autre pour décrire les particularités des différentes machines supportées. Etudions tout d'abord le fichier de définition de la WAM (cf. table 4). On y trouve la description des :

registres : pour chaque registre sa priorité, son type et son nom sont spécifiés. Les registres de la WAM seront alloués si possible dans des registres de la machine cible en

respectant les priorités indiquées. Ceci est possible avec gcc qui permet d'accéder aux registres machines. Un registre WAM ne pouvant être assigné à un registre machine est alloué au tout début du tas (c'est aussi le cas pour les registres arguments).

types de données : l'on définit ici tous les types de mots <étiquette,valeur> pour les données en heap. Plus précisément l'on nomme les étiquettes et l'on précise la nature de la partie valeur qui peut être un entier (signé ou non), une adresse dans l'une des piles de la WAM ou une adresse dans l'espace alloué dynamiquement par C (i.e. par `malloc`).

piles : pour chaque pile l'on précise son nom, sa taille par défaut et son sommet. Ces piles sont allouées dans l'ordre où elles sont décrites.

```
@begin regs
@reg 3 WamCont  CP                    /* continuation pointer      */
@reg 2 WamWordP E                     /* last environment pointer  */
@reg 2 WamWordP B                     /* last choice point pointer */
@reg 4 WamWordP BC                    /* backtrack cut pointer     */
@reg 1 WamWordP H                     /* top of the heap           */
@reg 2 WamWordP TR                    /* top of the trail          */
@reg 9 WamWordP S                     /* Unification pointer       */
@end regs

@begin tags
@tag INT int                          /* Integer                   */
@tag REF stack                        /* Reference                 */
@tag CST malloc                       /* Constant                  */
@tag LST stack                        /* List                      */
@tag STC stack                        /* Structure                 */
@end tags

@begin stacks
@stack trail   512 TR                 /* Trail  stack              */
@stack global 4096 H                  /* Global stack              */
@stack local  1024 ((B>=E) ? B : E)   /* Local  stack              */
@end stacks
```

Tableau 4 : fichier de description de la WAM

Ainsi, l'ajout de nouveaux registres (ex. pour les contraintes), de nouveaux types de données ou de nouvelles piles ne nécessite qu'une mise à jour de ce fichier. La description des

différents type de données permet l'adaptation automatique des tailles des parties étiquettes et valeurs ainsi que de la définition des macros d'"étiquetage" et de "desétiquetage" des mots mémoires.

L'autre fichier de configurations indique, pour chaque machine, les informations suivantes :

options C : options de compilation C particulières pour cette machine.

labels asm : macros permettant de générer les labels assembleurs.

gotos : définition des macros telles que `Direct_Goto` et `Indirect_Goto` rencontrées plus haut.

registres : registres utilisables pour la WAM (l'on précise aussi s'ils doivent être sauvegardés lors des appels de fonction de la librairie C car détruits par ceux-ci).

gestion des piles : méthode d'allocation des piles (`malloc`, `mmap` ou `shmget`) et le type de contrôle de débordement à effectuer (par logiciel ou par matériel).

Lors de l'installation sur une machine donnée, un programme C se charge de *configurer* le système à partir des deux fichiers. Il génère alors de nouveaux fichiers d'en-tête C permettant d'installer tout le système. Cette manière de procéder permet d'écrire la plupart du code de manière indépendante de la machine. Par exemple, qu'un registre WAM puisse être alloué à un registre machine ou non, le même nom (précisé dans le fichier de description de la WAM) peut être utilisé par la suite. C'est à la charge du programme de configuration de générer les bonnes définitions (de registres ou de macros) pour permettre cette transparence.

3.3.4 Gestion de la modularité

Un fichier source Prolog constitue un *module* et il est donc possible de décomposer une grosse application en plusieurs modules. Les points importants du système de modules sont :

- chaque module a son propre espace de noms de prédicats ce qui est un point important pour le développement de grosses applications.

- le système de modules de `wamcc` est basé sur les procédures, ce qui signifie que seuls les prédicats sont locaux à un module alors que les termes sont communs à tous les modules.
- le système de modules est "plat" et non pas "hiérarchique", tout module est visible des autres modules.
- aucun surcoût n'est dû à l'appel d'un prédicat se trouvant dans un module différent de celui de l'appelant.
- chaque prédicat (défini par l'utilisateur ou prédéfini) appartient à un seul module.
- par défaut un prédicat n'est visible que dans le module où il est défini (i.e. il est *privé*) sauf si ce prédicat a été déclaré comme étant *public* auquel cas il est visible dans tous les autres modules.
- un prédicat public peut être localement redéfini dans un module.
- parmi tous les modules composant une application, l'un d'entre eux doit être déclaré comme étant le module principal.

Ces caractéristiques sont celles de la plupart des Prolog qui gèrent les modules et semblent correspondre avec ce que définira la future norme ISO. Il est à remarquer que ces caractéristiques s'accommodent très bien de la compilation vers C puisque C à des règles de visibilité similaire en ce qui concerne les fonctions (publiques ou privées) et les données dynamiques (visibles par tous).

3.4 Evaluation de `wamcc`

Dans tout ce qui suit, les tailles de programmes sources sont indiquées en nombre de lignes (lignes vides et commentaires compris), les tailles d'objets et d'exécutables en Ko (tables des symboles non incluses) et les temps d'exécution en secondes (temps système non inclus) mesurés sur un Sparc 2 (28.5 Mips).

3.4.1 Les fonctionnalités du système wamcc

Du point de vue de l'utilisateur, wamcc offre les facilités suivantes :

- un compilateur Prolog vers C.
- une gestion de la modularité permettant le développement d'applications importantes.
- un ensemble de prédicats prédéfinis. En plus des prédicats classiques (entrées-sorties, arithmétique, manipulation de termes, test de types, contrôle, gestion de listes, opérateurs,...) wamcc offre des extensions non déclaratives telles que variables globales, affectations destructives "backtrackables" ou non, tableaux,...
- la possibilité d'inclure du code C.
- un "top-level" permettant de charger du code dynamiquement. Ce mode interprété est particulièrement pratique en phase de mise au point.
- un debugger Prolog (similaire à ceux de Sicstus ou Quintus).
- un debugger WAM permettant de vérifier/modifier les structures de base de la WAM. Ceci est très utile lorsque la WAM doit être modifiée/étendue.
- un utilitaire de génération de fichiers pour make.

Le compilateur wamcc traduisant un module Prolog en un source C est entièrement écrit en Prolog (3000 lignes). Il est d'ailleurs "auto-amorcé"[7] pour fournir un exécutable de 400 Ko en moins de 5 minutes.

En ce qui concerne la librairie de wamcc (cf. section 3.3.1) elle comprend :

- les fonctions dépendantes de la machine (ex. allocation de piles) (600 lignes de C).
- les fonctions d'exécutions des instructions WAM (2500 lignes de C).
- les fonctions de gestion de table de hash code (500 lignes de C).
- les fonctions de gestion des atomes, des prédicats et du code dynamique (i.e. assert, retract, consult,...) (2000 lignes de C).

[7] traduction de *"bootstrapped"*.

- les fonctions du debugger Prolog et WAM (2700 lignes de C).
- les prédicats prédéfinis qui sont en majorité écrits en Prolog (2300 lignes), le C étant utilisé pour le code de bas niveau.

La taille de cette librairie est de 160 Ko (dont 130 Ko pour les prédicats prédéfinis). Ainsi, la taille d'un exécutable correspondra à la somme des tailles des différents modules plus 160 Ko de librairie plus environ 30 Ko de librairie C.

3.4.2 Le jeu de benchmarks

Programme	Lignes	Temps de compilation	Taille de l'objet	Taille de l'exécutable	Temps d'exécution
boyer	395	64.0	48	240	3.450
browse	111	21.0	12	208	4.020
cal	202	18.0	12	208	0.300
chat_parser	1184	290.0	132	328	0.980
crypt	96	16.0	10	200	0.016
ham	90	14.0	10	200	4.330
meta_qsort	146	18.0	10	200	0.045
nand	574	202.0	76	264	0.120
poly_10	112	16.0	11	200	0.300
queens (16)	95	7.0	3	192	2.440
queens_n (8)	79	9.0	5	200	0.700
queens_n (10)	79	9.0	5	200	13.680
reducer	388	50.0	37	232	0.270
sdda	327	32.0	23	216	0.015
sendmore	66	12.0	8	200	0.230
tak	35	5.0	2	192	0.550
zebra	57	9.0	6	200	0.260

Tableau 5 : performances de wamcc (temps en sec.)

La table 5 présente les performances de wamcc sur un jeu de benchmarks classiques. Pour chaque programme nous trouvons :

- le nombre de lignes de Prolog du source.
- le temps de compilation total du programme (Prolog→C, gcc, édition des liens).

- la taille de l'objet obtenu et de l'exécutable final.
- le temps d'exécution.

3.4.3 wamcc versus des Prolog universitaires

Programme	wamcc 2.2	BinProlog 3.0	XSB-Prolog 1.4.0	SWI-Prolog 1.8.11
boyer	3.450	6.700	11.450	21.200
browse	4.020	7.930	11.850	18.180
cal	0.300	0.920	1.420	5.120
chat_parser	0.980	1.200	1.790	2.050
crypt	0.016	0.017	0.040	0.100
ham	4.330	5.280	8.840	12.650
meta_qsort	0.045	0.100	0.140	0.130
nand	0.120	0.320	$overflow$	0.420
poly_10	0.300	0.420	0.720	1.200
queens (16)	2.440	4.670	6.480	31.220
queens_n (8)	0.700	0.920	1.560	3.450
queens_n (10)	13.680	16.030	28.541	56.180
reducer	0.270	0.550	$overflow$	0.930
sdda	0.015	0.030	0.050	0.030
sendmore	0.230	1.100	0.670	2.580
tak	0.550	1.400	1.430	651.000
zebra	0.260	0.400	0.530	0.580
facteur d'accélération	1.000	2.005	2.716	5.632

Tableau 6 : wamcc versus autres Prolog universitaires (temps en sec.)

Nous nous proposons dans un premier temps de comparer wamcc à d'autres concurrents de sa catégorie : les Prolog universitaires. Ces implantations ont généralement été réalisées dans des universités à des fins de recherche par une seule personne et sont distribuées gratuitement sur le réseau internet via ftp. Le choix des langages Prolog utilisés pour la comparaison a été guidé par la popularité de ceux-ci. Nous trouvons donc :

BinProlog 3.0 : ce langage est basé sur la "binarisation" des clauses qui consiste principalement en un calcul où les continuations sont rendues explicites. La WAM est adapté à cette particularité et le code obtenu est émulé. L'émulateur est écrit en C.

XSB-Prolog 1.4.0 : ce langage est le successeur du populaire SB-Prolog. Il utilise aussi un émulateur écrit en C mais intègre, en plus, des techniques d'évaluation partielle pour spécialiser les appels en partie instanciés. La compilation peut donc être assez longue. Celle-ci est également capable de détecter certains cas de déterminisme ainsi que de compiler efficacement des structures de contrôle telles que le `if-then-else` Prolog.

SWI-Prolog 1.8.11 : ce Prolog se caractérise par une très grande vitesse de compilation et par la grande variété de prédéfinis dont il dispose. Il est à ce jour l'un des Prolog "amateurs" les plus utilisés.

La table 6 montre les temps d'exécution des différents systèmes ainsi que les facteurs d'accélération[8] de `wamcc` par rapport aux autres systèmes. Dans cette table, *overflow* signifie que trop de mémoire est requise et ce malgré nos tentatives d'accroître les tailles de piles. En moyenne `wamcc` est 2 fois plus efficace que BinProlog, 2.7 fois plus rapide que XSB-Prolog et 5.6 fois plus rapide que SWI-Prolog (sans tenir compte de `tak` sur lequel SWI-Prolog est extrêmement inefficace).

3.4.4 `wamcc` versus des Prolog professionnels

Ici nous comparerons `wamcc` à des Prolog ayant été développés par plusieurs personnes sur plusieurs années. Toutes ces implantations reposent sur des techniques d'optimisations unanimement reconnues au sein de la communauté Prolog comme le prouve le grand nombre d'articles (ICLP/ILPS) les présentant. Notre comparaison fera intervenir :

Sicstus : ce Prolog est très populaire pour avoir été une des premières implantations efficaces et de coût très faible. Il est même devenu une référence systématique en matière d'efficacité. Les versions initiales ne comprenaient qu'un émulateur écrit en C. Aujourd'hui, Sicstus a été remanié et comprend en plus de l'émulateur un mode "natif" sur certaines machines. Nous comparerons `wamcc` à la version émulée ainsi qu'à la version "code natif".

Quintus : ce langage a, pendant longtemps, été le plus rapide du marché, ce qui justifie sa notoriété (et aussi le fait qu'un certain Richard O'Keefe soit l'un de ses

[8] traduction de *speedup*.

Programme	`wamcc` 2.2	Sicstus 2.1 (émulé)	Sicstus 2.1 (natif)	Quintus 2.5.1	Aquarius
`boyer`	3.450	4.940	2.350	2.850	2.750
`browse`	4.020	6.630	2.020	3.340	1.380
`cal`	0.300	0.890	0.540	0.500	0.290
`chat_parser`	0.980	1.130	0.500	0.650	0.350
`crypt`	0.016	0.027	0.013	0.017	0.010
`ham`	4.330	5.050	2.090	3.000	0.950
`meta_qsort`	0.045	0.048	0.021	0.050	0.015
`nand`	0.120	0.200	0.084	0.130	0.040
`poly_10`	0.300	0.320	0.150	0.250	0.070
`queens (16)`	2.440	4.930	1.280	2.820	0.610
`queens_n (8)`	0.700	0.980	0.370	0.580	0.130
`queens_n (10)`	13.680	18.200	7.250	10.780	2.250
`reducer`	0.270	0.270	0.120	0.270	0.100
`sdda`	0.015	0.023	0.016	0.017	0.010
`sendmore`	0.230	0.630	0.170	0.280	0.080
`tak`	0.550	1.020	0.390	1.620	0.060
`zebra`	0.260	0.300	0.230	0.230	0.160
facteur d'accélération	1.000	1.582	$\downarrow$ 1.624	$\downarrow$ 1.026	$\downarrow$ 3.398

Tableau 7 : `wamcc` versus Prolog professionnels (temps en sec.)

implanteurs...). Quintus est également basé sur un émulateur mais celui-ci est écrit en assembleur. Notons que nous ne disposons que de la version 2.5.1 qui n'est pas la plus récente.

Aquarius : c'est aujourd'hui le Prolog le plus rapide. C'est un système complexe qui génère du code natif en passant par une machine abstraite originale plus performante que la WAM (la BAM). Le compilateur effectue énormément de travail (analyse de flots de données, interprétation abstraite, détection de déterminisme,...). Mais en retour les temps de compilation sont très pénalisants. La production de code natif à partir du code BAM est aussi très sophistiquée (elle comporte par exemple un re-ordonnanceur d'instructions pour le Sparc,...). C'est peut être ce côté "monstrueux" impliquant une compilation très lente qui a limité la diffusion de ce produit.

La table 7 présente les temps d'exécution de ces différents systèmes ainsi que le facteur d'accélération (ou de ralentissement si précédé du symbole $\downarrow$) de `wamcc` par rapport à

eux. Nous pouvons constater que notre objectif initial est largement atteint puisque `wamcc` est 1.5 fois plus rapide que Sicstus émulé. Par ailleurs, il est 1.6 fois plus lent que Sicstus "natif" se situant ainsi entre les deux modes de compilation de Sicstus. En ce qui concerne Quintus, `wamcc` soutient bien la comparaison puisqu'il est en moyenne aussi rapide. Enfin, `wamcc` est en moyenne 3.5 fois plus lent qu'Aquarius. Cet écart est dû en grande partie à la très bonne performance d'Aquarius sur `tak` (9 fois plus rapide que `wamcc`) qui détecte, lors de la compilation, que le calcul est déterministe, qui optimise au mieux l'appel terminal et qui est très optimisé pour les calculs entiers. Notons, que sur ce programme Aquarius est 2 fois plus rapide que la version équivalente écrite en C ! Sans ce redoutable benchmark `wamcc` est en moyenne 3 fois plus lent. En effet, sur des programmes utilisant davantage les particularités de Prolog (unification et backtracking) l'écart est moindre (ex. sur `zebra` Aquarius est seulement 1.6 fois plus rapide que `wamcc` et sur `boyer` il est environ 1.3 fois plus rapide).

Nous pouvons donc constater que `wamcc`, un système écrit en quelques mois, se compare très honorablement à des systèmes sophistiqués ayant demandé plusieurs homme-années de développement. Dans tous les cas, les facteurs ne sont pas à la mesure de la taille des systèmes. En effet, si nous comparons la complexité des noyaux de Sicstus (35000 lignes de C) et de `wamcc` (6000 lignes de C), le facteur 1.6 en faveur de Sicstus n'est pas si important. D'autant plus que le compilateur de Sicstus nécessite pas moins de 9000 lignes de Prolog contre seulement 3000 pour `wamcc`. En ce qui concerne Quintus, l'utilisation intensive le l'assembleur (noyau + certains prédéfinis) donne lieu à un code beaucoup plus difficile à maintenir que celui de `wamcc` sans pour autant que ses performances soient meilleures que celles de notre système. Bien sûr, les performances d'Aquarius sont remarquables mais elles le sont principalement sur des programmes purement arithmétiques et déterministes. Ce qui ne reflète pas à notre avis les applications type de Prolog. Par contre, les temps de compilation d'Aquarius nous semblent trop pénalisants. Celui-ci nécessite 38 minutes pour compiler `reducer`, un programme de moins de 400 lignes. En comparaison, `wamcc` n'a besoin que de 50 secondes pour fournir un exécutable qui certes est 2.7 fois plus lent mais s'exécute en dessous de la seconde. En ce qui concerne la taille des exécutables Peter Van Roy lui-même reconnaît qu'Aquarius génère un code 5 fois plus important que celui de Quintus à cause de l'analyse globale. Nos mesures nous ont montré que ce code est en moyenne 3 à 4 fois plus gros que celui produit par `wamcc`.

Enfin, en dehors de toute considération de temps d'exécution ou d'espace disque, le point important de `wamcc` réside dans sa minimalité due à son architecture "pyramidale", basée sur le fait que des compilateurs performants savent se charger de certaines tâches telles qu'allocation de registres et autres optimisations. De plus, cette approche permet de s'adapter aux évolutions de ces outils. Par exemple, avec gcc il est désormais possible d'utiliser les registres de la machine. De même, lors d'un portage futur de `wamcc` sur MSDOS, nous utiliserons turbo C puisque celui-ci est plus performant que gcc aussi bien en temps de compilation qu'en qualité de code produit (d'après une étude comparative du magazine *byte*). L'avantage que procure le fait de "réutiliser" plutôt que de "réécrire" est colossal. D'un côté c'est une économie de temps qui peut alors être consacrée à des recherches plus originales. D'un autre côté c'est l'assurance de ne pas engendrer des implantations monstrueuses dont la maintenance exige des compétences trop pointues.

Chapitre 4

Implantation de clp(FD)

Dans cette section nous nous intéresserons à l'implantation de `clp(FD)`, ce qui consistera d'abord à étudier une extension de la WAM pour prendre en compte les variables et les contraintes DF, puis à étudier l'intégration de cette extension dans `wamcc`. A cette occasion nous mettrons en évidence que le fait de compiler vers C nous permet d'obtenir du code efficace pour les contraintes. Enfin, nous évaluerons les performances de `clp(FD)`.

4.1 Extension de la WAM

4.1.1 Intégration des variables domaine

Pour permettre de raisonner avec des contraintes un nouveau type de données est ajouté : les variables DF qui sont donc capables d'enregistrer des ensembles d'entiers. Ces variables seront différenciées des autres variables grâce à une nouvelle étiquette (`FDV`). De manière à permettre une plus grande transparence, `clp(FD)` rend ces variables compatibles avec les variables Prolog et les entiers en permettant à l'utilisateur de fournir une variable Prolog (considérée comme une variable DF de domaine $0..\infty$) ou un entier n (considéré comme une variable DF de domaine $n..n$) là où une variable DF est attendue. De ce fait, l'utilisateur n'est pas contraint de déclarer les variables DF utilisées (au contraire de ce qu'impose CHIP). L'ajout de ce nouveau type de variable affecte légèrement la WAM comme suit :

*le contenu de ce chapitre a été publiée dans [24, 25].

Manipulation de données

Les variables DF (tout comme les variables Prolog) ne peuvent être dupliquées à l'inverse de ce que fait l'algorithme de recopie de structure pour les termes composés. Ainsi, le chargement d'une variable libre dans un registre crée une référence vers cette variable alors que le chargement d'une constante consiste réellement en une recopie de cette constante dans le registre. Remarquons que du fait de la représentation des variables par des auto-références, la même instruction de copie peut être utilisée aussi bien pour une variable Prolog que pour un autre terme. En ce qui concerne le chargement d'une variable DF nous avons alors deux possibilités :

- utiliser à nouveau la même instruction de copie. Dans ce cas il faut considérer les mots du type <`FDV`,α> comme ceux du type <`REF`,α> lors des déréférenciations pour atteindre le mot final (éventuellement une auto-référence). Ce qui complique donc l'algorithme de déréférenciation du fait qu'il a désormais deux types de mots permettant les références (i.e. liaisons).

- ne pas modifier l'algorithme de déréférenciation mais prendre soin de ne pas recopier les variables DF. Ainsi, lorsqu'un mot source M_s doit être chargé dans un mot (destination) M_d si M_s est une variable DF alors M_d est lié à M_s sinon M_s est physiquement copié dans M_d.

C'est cette deuxième solution qui a été adoptée dans `clp(FD)` car la déréférenciation est une opération exécutée très souvent et la première solution pénaliserait les portions de Prolog n'utilisant pas les variables DF. Ainsi, un mot de la forme <`FDV`,*valeur*> n'est jamais dissocié des autres informations associées à une variable DF (domaine, liste de dépendances,...)[1]. La partie *valeur* d'un mot <`FDV`,*valeur*> est alors inutile (ou peut être utilisée pour encoder une des informations de la variable). Dans notre cas nous utilisons une auto-référence pour permettre, à la vue du *seul* mot <`FDV`,α> de connaître l'adresse α de la variable DF concernée. Ceci permet de réutiliser les fonctions de manipulation des données Prolog (ex. affichage) qui acceptent en entrée un mot étiqueté. Il suffit alors d'étendre ces fonctions en prenant en compte le cas *étiquette=FDV*. En aucun cas il n'est nécessaire d'ajouter un argument aux fonctions pour passer l'adresse des variables DF.

[1] au contraire de ce qui se passe pour les structures où il y peut y avoir plusieurs mots <`STC`,α> pointant la même structure qui se trouve alors "séparée" de ces mots et réside à l'adresse α.

Unification

Du fait de la compatibilité entre variables Prolog, variables DF et entiers, une variable DF X peut être unifiée avec :

- une variable Prolog Y : Y est simplement liée à X.
- un entier n : ceci revient à ajouter la contrainte X `in` $n..n$.
- une autre variables DF Y : ceci revient à ajouter les contraintes X `in dom(`Y`)` et Y `in dom(`X`)`.

Sauvegarde et restauration des domaines

La prise en charge des DF entraîne la nécessité de pouvoir sauvegarder des valeurs dans la trail (ex. domaines modifiés). Ce qui est déjà supporté par notre architecture WAM. En revanche, le fait que les domaines soient réduits en plusieurs étapes entraîne que le critère de sauvegarde standard de la WAM (i.e. adresse du mot à modifier plus ancienne que celle dernier point de choix) conduira à la mise en trail de chaque modification de domaine. Bien évidemment, un seul domaine (l'original) nécessite d'être sauvegardé par point de choix. Pour cela nous utilisons la méthode d'estampillage de [1] qui consiste à ajouter un nouveau registre `STAMP` pour numéroter les points de choix ainsi qu'un champ pour chaque variable DF enregistrant le numéro de point de choix de sa dernière sauvegarde en trail. Le registre `STAMP` est donc incrémenté lorsqu'un point de choix est crée et décrémenté à la suppression de celui-ci. Dans ce cas, X doit être sauvegardé en trail si $Stamp(X) \neq STAMP$.

Indexation

La manière la plus simple de gérer une variable DF du point de vue de l'indexation est de la considérer comme une variable Prolog donc d'essayer toutes les clauses. Bien évidemment il serait possible d'effectuer une indexation plus complexe pour ne prendre en compte que les clauses ayant des entiers ou des variables comme premier argument. Il serait même possible d'étendre les constantes de `clp(FD)` pour prendre en compte les intervalles d'entiers permettant par là même une écriture déclarative et efficace des fonctions définies par morceaux.

4.1.2 Nouvelles structures de données

La prise en charge des contraintes DF nécessite la définition de nouvelles structures de données. Celles-ci seront toutes stockées dans le heap.

Représentation des environnements

Un environnement représente le contexte dans lequel doit être (ré)exécutée une contrainte. Plus précisement il contient l'adresse des variables DF et des paramètres domaines et les valeurs des paramètres entiers (cf. figure 7). Un nouveau registre `AF` pointe alors l'environnement courant.

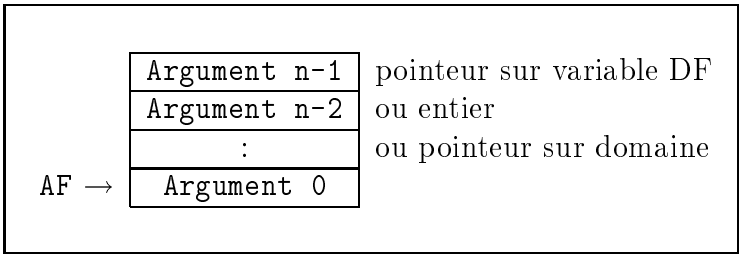


Figure 7 : représentation interne d'un environnement

Dans ce qui suit les variables DF sont référencées $fv(i)$ (Frame Variable) et les paramètres $fp(j)$ (Frame Parameter) où i et j sont des entiers servant d'indice dans l'environnement (i.e. par rapport à `AF`). Ainsi, la clause (de l'exemple 2.1) :

```
'x=y+c'(X,Y,C):- X in min(Y)+C..max(Y)+C,
                 Y in min(X)-C..max(X)-C.
```

sera transformée à la compilation en le pseudo-code `clp(FD)` :

```
'x=y+c'(X,Y,C):- créer un environnement à 3 éléments,
                 charger X à l'indice 0 (X=fv(0)),
                 charger Y à l'indice 1 (Y=fv(1)),
                 charger C à l'indice 2 (C=fp(2)),
                 fv(0) in min(fv(1))+fp(2)..max(fv(1))+fp(2),
                 fv(1) in min(fv(0))-fp(2)..max(fv(0))-fp(2).
```

Représentation des contraintes

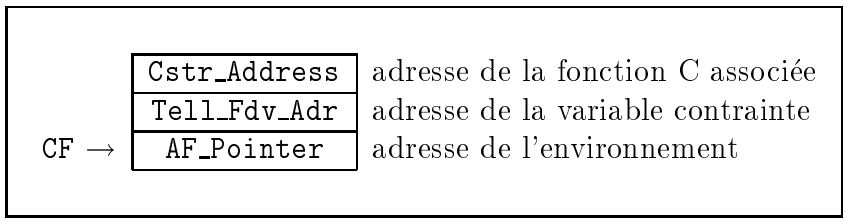


Figure 8 : représentation interne d'une contrainte

Une structure interne est associée à chaque contrainte X `in` r et stocke les différentes informations suivantes (cf. figure 8) :

- l'adresse de l'environnement dans lequel doit être activée cette contrainte.
- l'adresse de la variable X (qui est contrainte).
- l'adresse du code d'exécution (qui se charge d'évaluer le domaine r pour ensuite appeler la fonction $Tell$ qui mettra à jour X).

Un nouveau registre `CF` pointe alors la contrainte courante.

Représentation des domaines

Il y a deux représentations possibles pour un domaine :

Min-Max : seuls le *min* et le *max* sont maintenus. Cette représentation est donc utilisée pour des intervalles et peut stocker des valeurs incluses dans $0..infinity$.

Sparse : en plus du *min* et du *max* un vecteur de bits est utilisé pour stocker, en extension, l'ensemble des valeurs du domaine. Cette représentation permet de stocker des valeurs incluses dans $0..vector_max$. Par défaut $vector_max$ vaut 127 et peut être redéfini grâce à une variable d'environnement ou via le prédicat prédéfini (`fd_vector_max/1`). Les vecteurs ne sont pas dynamiques et ce prédicat ne peut donc être utilisé qu'une seule fois.

La représentation initiale d'un domaine D est toujours une représentation *Min-Max* et devient une représentation *Sparse* aussitôt qu'un "trou" apparaît dans D (ex. à cause d'une

opération de complémentation, d'une union,...). Lorsqu'un domaine est devenu *Sparse* il ne reviendra pas à une représentation *Min-Max*. Le domaine vide est représenté avec un $min > max$. Ceci permet, en mode *Min-Max*, d'effectuer l'intersection (opération la plus utilisée) entre D_1 et D_2 sans tester si l'un ou l'autre des domaines est vide. En effet, le résultat obtenu par : $max(min(D_1), min(D_2))..min(max(D_1), max(D_2))$ retourne un $min > max$ dans le cas ou soit D_1 soit D_2 est vide.

Lorsque un domaine D, représenté en *Min-Max*, passe à une représentation *Sparse* certaines valeurs peuvent être perdues du fait que $vector_max$ est bien plus petit que $infinity$. Pour gérer ces pertes influant sur la complétude des réponses, `clp(FD)` considère que ce domaine a été *extra-contraint* puisque tout se passe comme si D avait été soumis à une contrainte D `in 0..`$vector_max$ par le résolveur. Un indicateur est alors adjoint à tout domaine pour indiquer s'il a perdu des valeurs (i.e. s'il est extra-contraint). Cet indicateur est mis à jour par toutes les opérations sur les domaines. Par exemple, l'union de deux domaines est extra-contraint si au moins l'un des deux l'est (i.e. *ou* des deux indicateurs) ; de même, l'intersection de deux domaines est extra-contraint si les deux le sont (i.e. *et* des indicateurs), etc... Ainsi, cette information elle aussi est propagée. Lorsqu'une contrainte sur la variable X échoue, si le domaine de X est extra-contraint le solveur affiche un message pour prévenir que, du fait de la taille trop petite du vecteur de bits, certaines solutions peuvent être perdues (incomplétude des réponses). L'utilisateur peut alors décider d'allouer plus d'espace pour les vecteurs. Voici un exemple où une variable domaine X a un domaine extra-contraint, qui redevient "normal" par la suite (on considère que $vector_max = 127$) :

Contrainte sur X	Domaine de X	Extra contraint ?	Valeurs perdues
`X in 0..512`	0..512	`non`	$\emptyset$
`X in 0..3:10..512`	0..3 : 10..127	`oui`	128..512
`X in 0..100`	0..3 : 10..100	`non`	$\emptyset$

Dans cet exemple, quand la contrainte `X in 0..3:10..512` est ajoutée, certaines valeurs sont perdues (indicateur positionné à vrai). Toutefois, l'ajout ultérieur de la contrainte `X in 0..100` les élimine de toute façon donc l'indicateur retourne à faux.

L'exemple suivant met en évidence une incomplétude du solveur :

Contrainte sur X	Domaine de X	Extra contraint ?	Valeurs perdues
`X in 0..512`	0..512	`off`	$\emptyset$
`X in 0..3:10..512`	0..3 : 10..127	`on`	128..512
`X in 256..300`	$\emptyset$	`on`	*Attention...*

Dans cet exemple, la contrainte `X in 256..300` échoue du fait de la perte de valeurs 128..512 donc un message d'avertissement est affiché.

Cette représentation à été adoptée car elle est simple, correcte et que les algorithmes associés aux opérations sur les domaines peuvent être implantés efficacement. Il aurait été possible d'utiliser une représentation par union d'intervalles qui n'aurait pas eu les mêmes limitations, de même qu'il aurait été possible d'enregistrer une "base" β permettant de coder tous domaines "à trous" appartenant à l'intervalle $\beta..\beta + vector_max$. Toutefois, ceci aurait nui à l'efficacité sans être vraiment utile puisque dans la plupart des problèmes les domaines impliqués sont petits et proches de 0. Si tel n'est pas le cas il est toujours possible d'énoncer son problème à une translation près.

La figure 9 montre les représentations internes d'un domaine.

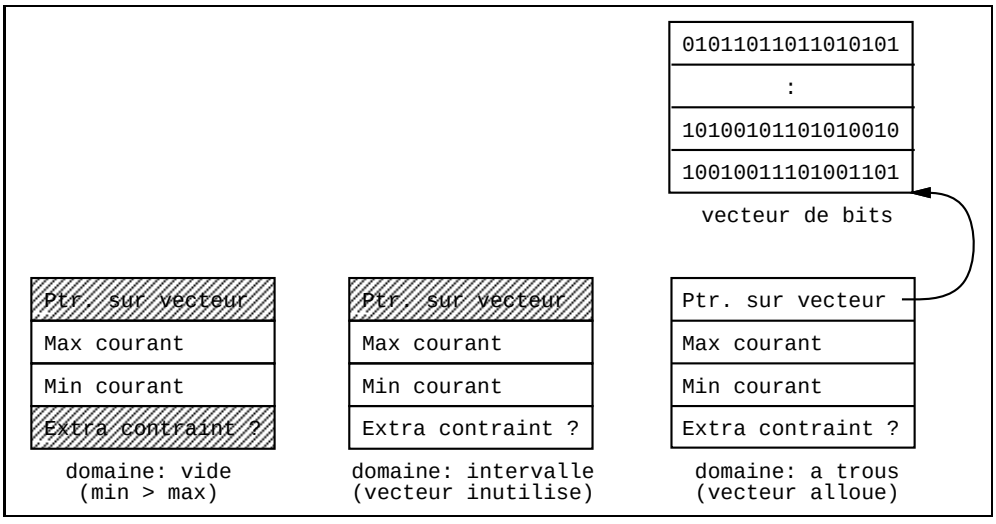


Figure 9 : représentations internes d'un domaine

Représentation des variables DF

Les informations associées à une variable DF X sont divisées en 2 parties (cf. figure 10) :

domaine : on y trouve le nombre d'éléments et la représentation du domaine (non vide). Cette partie utilise une estampille pour éviter les mises en trail multiples.

listes de dépendances : liste des contraintes dépendant de X. En vue de diminuer le nombre de contraintes activées inutilement plusieurs listes sont distinguées. En effet, il est inutile de réveiller une contrainte ne dépendant que de `min(`X`)` lorsque seul le max de X a été modifié. Les différentes listes sont :

- `Chain_Min` : liste des contraintes dépendant de `min(`X`)` et pas de `max(`X`)`.
- `Chain_Max` : liste des contraintes dépendant de `max(`X`)` et pas de `min(`X`)`.
- `Chain_Min_Max` : liste des contraintes dépendant de `min(`X`)` et de `max(`X`)`.
- `Chain_Dom` : liste des contraintes dépendant de `dom(`X`)`.
- `Chain_Val` : liste des contraintes dépendant de `val(`X`)` (i.e. à activer quand X sera clos).

Notons qu'il est facile d'assurer qu'une contrainte n'appartient pas à plusieurs listes pour une variable donnée. Cette partie utilise sa propre estampille car elle est mise à jour indépendamment du domaine.

Les listes `Chain_Min`, `Chain_Max` et `Chain_Min_Max` référencent des contraintes utilisant une propagation du type *partial lookahead*. Les listes `Chain_Min` et `Chain_Max` référencent les inéquations et la liste `Chain_Min_Max` les équations. La liste `Chain_Dom` contient les contraintes utilisant un schéma de *full lookahead*. La liste `Chain_Val` référence les contraintes utilisant le *forward checking* (ex. diséquations).

La figure 11 schématise les différentes structures de données nécessaires aux contraintes.

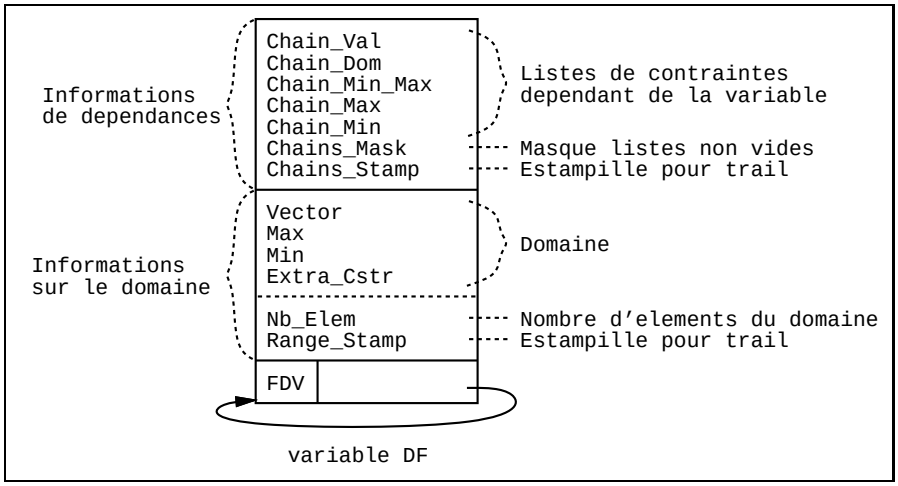


Figure 10 : représentation interne d'une variable DF

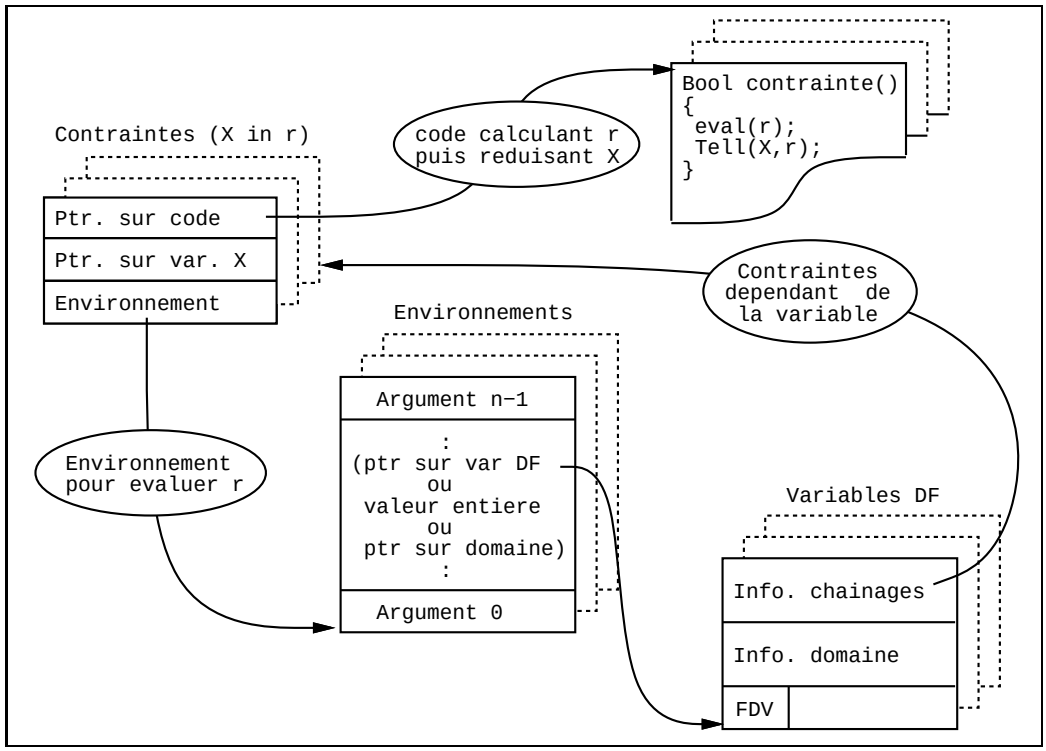


Figure 11 : structures de données pour les contraintes

Registres

En vue de gérer les différentes structures de données présentées ci-dessus les registres suivants sont nécessaires :

`CC`	(Constraint Cont.)	pointeur de continuation après contrainte.
`STAMP`	(Stamp)	compteur d'estampilles des points de choix.
`BP`	(Base Pointer)	pointeur de début de la file de propagation.
`TP`	(Top Pointer)	pointeur de fin de la file de propagation.
`AF`	(Argument Frame)	pointeur d'environnement courant.
`CF`	(Constraint Frame)	pointeur de contrainte courante.
`T[i]`	(Terms)	banc de registres entiers.
`R[i]`	(Ranges)	banc de registres domaines.

Le registre `CC` sert de registre de continuation lors de l'appel des contraintes. L'ajout de ce registre permet de compiler une contrainte X `in` r comme un prédicat *inline* (ne donnant pas lieu à un appel classique) ce qui évite de devoir sauvegarder `CP`.

Le rôle du registre `STAMP` est de numéroter les points de choix pour éviter les mises en trail multiples (cf. section 4.1.1).

Les registres `BP` et `TP` pointent les extrémités de la file de propagation.

Les registres `AF` et `CF` désignent respectivement l'environnement courant et la contrainte courante (cf. section 4.1.2).

Enfin, un banc de registres `T[i]` pouvant stocker des termes (des entiers) et un autre pour les domaines (`R[i]`) sont utilisés pour évalue le domaine dénoté par r d'une contrainte X `in` r.

4.1.3 Jeu d'instructions

Le jeu d'instructions de la WAM est étendu pour prendre en compte les contraintes grâce à trois groupes d'instructions émis pour toute clause contenant au moins une contrainte :

- instructions d'interfaçage avec Prolog.
- instructions d'installation des contraintes.
- instructions d'exécution des contraintes.

Instructions d'interfaçage avec Prolog

Ces instructions ont à charge de créer l'environnement dans lequel devront être évaluées les contraintes. En effet, toutes les contraintes d'une même clause se partagent un même environnement qui est alors créé et initialisé juste avant l'appel de la première d'entre-elles. Ce qui correspond principalement à réserver suffisamment d'espace dans le heap, à initialiser `AF` et, pour chaque argument, à stocker son adresse (si c'est une variable DF ou un paramètre domaine) ou sa valeur (dans le cas d'un paramètre entier) (cf. représentation interne des environnements en section 4.1.2). Dans ce qui suit, l'écriture `V`, elle dénote une variable temporaire (i.e. `X[j]` ou permanente (i.e. `Y[j]`) comme en section 3.1.7.

`fd_set_AF(nb_arg,V)`
: réserve l'espace dans le heap pour un environnement de `nb_arg` arguments. `AF` et la variable `V` pointent sur cet environnement.

`fd_variable_in_A_frame(V)`
: lie `V` à une variable DF créée sur le heap (de domaine $0..infinity$) et range son adresse dans le mot pointé par `AF`. `AF` est alors incrémenté.

`fd_value_in_A_frame(V)`
: suivant que la valeur w de la déréférence de `V` 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`. `AF` est alors incrémenté.
  - une variable DF : son adresse est rangée dans le mot pointé par `AF`. `AF` est alors incrémenté.

`fd_range_parameter_in_A_frame(V)`
: `V` doit être lié à une liste d'entiers et le domaine correspondant est créé sur le heap. L'adresse de ce domaine est copiée dans le mot pointé par `AF`. `AF` est alors incrémenté.

`fd_term_parameter_in_A_frame(V)`
: `V` doit être un entier qui est alors rangé dans le mot pointé par `AF`. `AF` est alors incrémenté.

Pour chaque contrainte, les instructions suivantes sont alors produites :

`fd_install_constraint(install_proc,V)`

ré-initialise `AF` avec le contenu de `V` 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

Pour chaque contrainte, une procédure d'installation est générée dont le rôle est de créer et de charger une structure de donnée interne pour cette contrainte (cf. représentation interne des contraintes en section 4.1.2). Cette procédure initialise également les listes de dépendances appropriées de toutes les variables utilisées par cette contrainte. Par exemple, pour la contrainte $c \equiv X$ `in min(`Y`)..infinity`, le code d'installation ajoutera un pointeur vers c dans la liste de contraintes dépendant de `min(`Y`)`.

`fd_create_C_frame(constraint_proc,tell_fv)`

crée sur le heap une structure pour la contrainte dont le code d'exécution se trouve à l'adresse `constraint_proc` et dont la variable contrainte est la `tell_fv`*ième*. `CF` pointe cette structure.

`fd_install_`$\left\{\begin{array}{c}\texttt{ind_min}\\ \texttt{ind_max}\\ \texttt{ind_min_max}\\ \texttt{ind_dom}\\ \texttt{dly_val}\end{array}\right\}$`(fv)`

Ces instructions sont utilisées quand la contrainte (pointée par `CF`) utilise le *min* (ou le max, ou les 2,...) de la `fv`*ième* variable. Un nouvel élément est ajouté dans la liste correspondante à la `fv`*ième* variable.

`fd_proceed`

rend le contrôle à l'adresse pointée par `CC`.

Instructions d'exécutions des contraintes

Pour chaque contrainte X `in` r, une procédure est chargée de son (re)calcul. Celle-ci peut être décomposée en 4 phases :

- chargement des paramètres, des termes et domaines indexicaux dans les registres de travail.
- évaluation du domaine dénoté par r.
- appel de la fonction $Tell$ pour mettre à jour X en fonction de r.
- retour à l'appelant par `fd_proceed` (cf. ci-dessus).

chargement des paramètres, des termes et domaines indexicaux. La première partie du code d'exécution d'une contrainte consiste à extraire de l'environnement (pointé par `AF`) les arguments utilisés pour les charger dans des registres de travail. Le fait que tout les chargements aient lieu au début de la fonction permet d'optimiser l'utilisation des registres et d'éviter les chargements inutiles. Par exemple, si une contrainte utilise `dom(`X`)` et `min(`X`)` seul le domaine de X doit être chargé puisqu'il contient déjà le min de X.

`fd_range_parameter(R[r],fp)`
: charge le domaine pointé par le `fp`*ième* paramètre dans le registre `R[r]`.

`fd_term_parameter(T[t],fp)`
: charge la valeur du `fp`*ième* paramètre dans `T[t]`.

`fd_ind_`$\left\{ \begin{array}{c} \texttt{min} \\ \texttt{max} \end{array} \right\}$`(T[t],fv)`
: charge le $\left\{ \begin{array}{c} min \\ max \end{array} \right\}$ de la `fv`*ième* variable dans `T[t]`.

`fd_ind_min_max(T[tmin],T[tmax],fv)`
: charge le min et le max de la `fv`*ième* variable dans `T[tmin]` et `T[tmax]`.

`fd_ind_dom(R[r],fv)`
: charge le domaine de la `fv`*ième* variable dans `R[r]`.

`fd_dly_val(T[t],fv,lab_else)`

si la `fv`*ième* variable est un entier, sa valeur est copiée dans `T[t]` sinon le contrôle est donné au code d'étiquette `lab_else`.

Evaluation du domaine dénoté par r. La compilation de r est classique et similaire à celle des expressions arithmétiques (i.e. dirigée par la syntaxe). L'arbre syntaxique est parcouru des feuilles vers la racine, chaque feuille et chaque noeud donnant lieu à une instruction spécifique. Pour les feuilles correspondant à des paramètres ou des termes indexicaux, des instructions de copie sont produites pour initialiser les registres appropriés à partir de ceux chargés par la partie précédente. Le code final peut bénéficier d'un optimiseur de registre[2].

`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]`$\leftarrow 0..infinity \setminus$ `R[r]`.

`fd_compl_of_singleton(R[r],T[t])` exécute `R[r]`$\leftarrow 0..infinity \setminus \{$`T[t]`$\}$.

`fd_range_`$\left\{\begin{array}{l}\texttt{add}\\ \texttt{sub}\\ \texttt{mul}\\ \texttt{div}\end{array}\right\}$`_term(R[r],T[t])` exécute `R[r]`$\leftarrow$`R[r]`$\left\{\begin{array}{l}+_{pointwise}\\ -_{pointwise}\\ *_{pointwise}\\ /_{pointwise}\end{array}\right\}$`T[t]`.

`fd_range_copy(R[r],R[r1])` exécute `R[r]`$\leftarrow$`R[r1]`.

`fd_integer(T[t],n)` exécute `T[t]`$\leftarrow$`n`.

`fd_term_`$\left\{\begin{array}{c}\texttt{add}\\ \texttt{sub}\\ \texttt{mul}\\ \texttt{floor_div}\\ \texttt{ceil_div}\end{array}\right\}$`_term(T[t],T[t1])` exécute `T[t]`$\leftarrow$`T[t]`$\left\{\begin{array}{c}+\\ -\\ *\\ \lfloor/\rfloor\\ \lceil/\rceil\end{array}\right\}$`T[t1]`.

`fd_term_copy(T[t],T[t1]` exécute `T[t]`$\leftarrow$`T[t1]`.

[2]dans `clp(FD)` nous avons réutilisé l'allocateur/optimiseur de registres de la WAM.

Appel de la fonction *Tell*. Rappelons que la contrainte est pointée par `CF` qui permet également d'atteindre X. Donc seul le résultat de l'évaluation de r doit être précisé par les instructions appelant *Tell*. Dans le but d'optimiser les domaines réduits à un intervalle nous distinguerons les cas X *in* $t_1..t_2$ du cas général X `in` r. La description complète de la fonction *Tell* sera donnée en section 4.1.4.

`fd_tell_range(R[r])`

ajoute la contrainte X `in R[r]` quand r n'est pas un intervalle.

`fd_tell_interval(T[tmin],T[tmax])`

ajoute la contrainte X `in T[tmin]..T[tmax]` (i.e. r est un intervalle).

La table 8 présente un fragment du code généré pour notre exemple typique `'x=y+c'` (cf. exemple 2.1).

```
'x=y+c':  fd_set_AF(3,X[3])                    3 éléments 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                   exécute cstr_1
          fd_install_constraint(inst_2,X[3])   installe cstr_2
          fd_call_constraint                   exécute cstr_2
          proceed                              retour Prolog

inst_1:   fd_create_C_frame(cstr_1,0)
          fd_install_ind_min_max(fv(1))        utilise min(Y) et 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'exécution

inst_2:   (...)
```

Tableau 8 : fragment du code généré pour `'x=y+c'`

4.1.4 Opération *Tell*

Comme nous l'avons vu précédemment, la fonction associée à chaque contrainte X `in` r commence par évaluer le domaine dénoté par r avant d'appeler la fonction $Tell$. Celle-ci doit mettre à jour X en fonction de r et, en cas de modification du domaine de X, doit réveiller toutes les contraintes dépendant de X. Les grandes lignes de cette opération peuvent être résumées comme suit :

Si X est un entier, celui-ci doit appartenir à r :

- $X \in r$: succès (**VérifEntier**)
- $X \notin r$: échec (**EchecEntier**)

sinon (X est une variable dont le domaine courant est r_X) soit $r' = r \cap r_X$:

- $r' = \emptyset$: échec (**EchecDomaine**)
- $r' = r_X$ (i.e. $r_X \subset r$) : succès (**VérifDomaine**)
- sinon : le domaine de X est remplacé par r' (**RéducDomaine**). C'est à cette occasion que X peut devenir clos. La réduction du domaine de X est alors répercutée à toutes les contraintes dépendant de X par la phase de propagation[3]. C'est ici que nous bénéficions du fait d'avoir plusieurs listes de dépendances distinctes pour éviter des réveils inutiles.

Notons que lorsque le domaine courant de X est déjà compris dans celui dénoté par r (issues **VérifDomaine** et **VérifEntier**), cette opération est inutile. En section 4.3.3, nous analyserons les sources de tels appels pour voir s'il est possible de les détecter et d'en réduire le nombre.

La phase de propagation gère l'ensemble des contraintes à réactiver. La réexécution d'une de ces contraintes peut à son tour enrichir cet ensemble. Du fait que l'ordre de ces réactivations n'influe pas sur la correction, nous avons toute latitude sur la manière de gérer cet ensemble. Nous pouvons le représenter explicitement (tas, pile, file,...) ou adopter un schéma d'exécution basé sur des continuations implicites. Dans ce cas, après réduction du domaine de X, toute contrainte dépendant de X est immédiatement exécutée par un

[3]la valeur du registre `CC` doit être empilée sur la pile locale pour être restaurée après la propagation.

appel imbriqué. Ceci est très similaire à l'exécution des buts en PL où l'on peut choisir entre la stratégie, en profondeur d'abord, de Prolog et des gestions plus complexes des buts de la résolvante comme cela se fait dans les langages logiques concurrents. En ce qui concerne l'implantation de `clp(FD)`, nous avons choisi une représentation explicite par file de l'ensemble des contraintes à réactiver car le léger surcoût qu'elle engendre est largement contrebalancé par la flexibilité qu'elle offre. De plus, l'expérience nous a montré que l'obtention des solutions était généralement plus rapide grâce à une propagation en largeur d'abord (i.e. file) qu'avec une propagation en profondeur d'abord (i.e. pile).

4.2 Intégration de `clp(FD)` dans `wamcc`

`clp(FD)` est implanté au dessus de `wamcc` et bénéficie donc de toutes les facilités de ce Prolog. Le compilateur a été modifié pour générer le code WAM étendu décrit précédemment (environ 1000 lignes de Prolog en plus). La librairie associée à `clp(FD)` étend celle de `wamcc` par :

- fonctions d'exécutions des instructions étendant la WAM, i.e. le solveur proprement dit (1700 lignes de C).
- fonctions relatives aux opérations sur les domaines telles qu'intersection, union,... (1700 lignes de C).
- prédicats prédéfinis propres aux variable DF et aux contraintes (1000 lignes de Prolog et 800 lignes de C).

Ces extensions repésentent seulement 100 Ko de code supplémentaires par rapport à la librairie de `wamcc` (260 Ko contre 160 Ko). Tous ces chiffres permettent de se rendre compte de la minimalité de l'extension proposée pour les contraintes.

Du point de vue de l'exécution, toute contrainte DF donne lieu à deux fonctions. La première contient le code d'installation et la seconde celui d'exécution. Puisque chaque (ré)exécution de contrainte donne lieu à un appel de fonction C, le registre `CC` n'est plus nécessaire (le registre de continuation du processeur est implicitement utilisé). La fonction d'exécution associée à une contrainte (ré)évalue la contrainte pointée par `CF` et retourne un booléen suivant la réussite ou l'échec de ce calcul. Du fait qu'une telle fonction est invoquée

plusieurs milliers de fois il est important de veiller à la qualité du code qui la compose. Pour cette raison, nous évitons la définition explicite de `CF` comme registre global de la WAM étendue pour plutôt le passer en tant que paramètre de la fonction d'exécution. Il est de ce fait considéré comme une variable locale de cette fonction. De même le registre `AF` est simplement défini comme une variable locale dans la fonction et est initialisé dès l'entrée à partir de `CF`. Pour augmenter la vitesse d'évaluation des contraintes, les registres `R[]` et `T[]` sont définis, eux aussi, comme des variables locales dans chaque fonction. Ceci est d'autant plus intéressant que sur les machines RISC l'accès aux variable locales est moins coûteux que l'accès à des données globales. De plus, le compilateur C essaye d'allouer les variables locales dans des registres de la machine pour la durée de la fonction. En particulier les registres de termes de la WAM étendue (`T[]`) peuvent être alloués dans des registres machines ramenant l'évaluation des contraintes sur des intervalles à des opérations entre registres. Pour illustrer cela considérons à nouveau la contrainte utilisateur `'x=y+c'` (cf. exemple 2.1). La table 8 précédemment rencontrée montrait le code WAM étendu associé à cette contrainte. La compilation vers C de ce code, en ce qui concerne la première contrainte `X in min(Y)+C..max(Y)+C`, est présentée en table 9. Cette traduction donne lieu à une fonction ne manipulant que des entiers au travers de variables locales que le compilateur C alloue facilement dans des registres. Ce qui peut être vérifié sur la table 10 qui présente le code assembleur Sparc produit par le compilateur gcc.

```
static Bool cstr_1(WamWord *CF)      X in min(Y)+C..max(Y)+C

{
 WamWord *AF=AF_Pointer(CF);
 WamWord *fdv_adr;
 WamWord  tr0,tr1,tr2;
 Bool     ok;

 fdv_adr=(WamWord *) (AF[1]);        fd_ind_min_max(T[0],T[1],fv(1))
 tr0=Range(fdv_adr)->min;               :
 tr1=Range(fdv_adr)->max;               :

 tr2=(int) (AF[2]);                  fd_term_parameter(T[2],fp(2))

 tr0+=tr2;                           fd_term_add_term(T[0],T[2])
 tr1+=tr2;                           fd_term_add_term(T[1],T[2])

 fdv_adr=Tell_Fdv_Adr(CF);           fd_tell_interval(T[0],T[1])
 ok=Tell_Interval(fdv_adr,tr0,tr1);     :
 return ok;                          fd_proceed
}
```

Tableau 9 : fragment de code C généré pour 'x=y+c'

```
_cstr_1:                    en entrée: %i0=CF
     save %sp,-104,%sp      prologue
     ld [%i0],%o3           %o3=AF=AF_Pointer(CF)
     ld [%i0+4],%o0         %o0=adresse de X
     ld [%o3+4],%o2         %o2=adresse de Y
     ld [%o3+8],%o3         %o3=C
     ld [%o2+32],%o1        %o1=min(Y)
     ld [%o2+36],%o2        %o2=max(Y)
     add %o1,%o3,%o1        %o1=min(Y)+C
     call _Tell_Interval,0  %o0=Tell_Interval(%o0,%o1,%o2)
     add %o2,%o3,%o2        %o2=max(Y)+C (delay slot)
     ret                    retour de %o0
     rstore %g0,%o0,%o0     épilogue (delay slot)
```

Tableau 10 : fragment de code assembleur Sparc généré pour 'x=y+c'

4.3 Evaluation de clp(FD)

4.3.1 Le jeu de benchmarks

Un ensemble de benckmarks classiques a été utilisé pour tester les performances de la version de clp(FD) :

- crypta : une addition "cryptée" portant sur 10 variables de domaine initial 0..9, 2 de domaine 0..1, 3 équations linéaires et 45 diséquations [67].
- eq10 : un système de 10 équations linéaires sur 7 variables de domaine initial 0..10.
- eq20 : un système de 20 équations linéaires sur 7 variables de domaine initial 0..10.
- alpha : un problème de chiffrage (codage) nécessitant 26 variables de domaines initiaux 1..26, 20 équations et 325 diséquations.
- queens : le très célèbre problème des N-reines [67] avec N variables de domaine initial 1..N et 3*N*(N-1)/2 diséquations.
- five : le puzzle des cinq maisons [67] nécessitant 25 variables de domaine intial 1..5, 11 équations linéaires, 50 diséquations et 3 disjonctions de 2 équations linéaires.
- cars : le problème du *car sequencing* [33] avec 10 variables de domaine intial 1..6, 50 de domaine 0..1, 49 inéquations et 56 contraintes symboliques (element, atmost [67]).

Les programmes crypta, eq10, eq20 et alpha permettent de tester les aptitudes de clp(FD) à résoudre les équations linéaires pures et avec diséquations. Les autres programmes testent l'efficacité de clp(FD) sur divers autres aspects tels que *forward checking* (queens), disjonctions (five) et contraintes symboliques du type de element ou atmost (cars).

Dans tous les programmes seule la première solution est calculée et l'énumération se fait de manière standard excepté lorsque ff est précisé indiquant alors une énumération basée sur l'heuristique *first-fail* qui énumère d'abord sur la variable de plus petit domaine [67]).

4.3.2 Evaluation de l'implantation de base

Pour permettre d'évaluer la version de base de `clp(FD)` nous l'avons comparée à la version commerciale 3.2 du langage CHIP de COSYTEC. Exactement les mêmes programmes ont été utilisés avec les deux langages sur un Sparc 2 (28.5 Mips). Les temps indiqués sont en secondes et ne tiennent pas compte du temps système. La table 11 présente les temps d'exécution des deux systèmes ainsi que le facteur d'accélération (ou de ralentissement si précédé du symbole $\downarrow$) de `clp(FD)` par rapport à CHIP. En moyenne, `clp(FD)` est 1.5 fois plus rapide que CHIP en ce qui concerne les équations linéaires (avec parfois des pointes à 6) et 3 fois plus rapide pour les autres programmes. Ces performances sont très honnêtes au regard de la simplicité du système. Toutefois, l'analyse de la décomposition des opérations *Tell* nous révèle qu'en moyenne 72 % de ces opérations sont inutiles parce que réussissant sans aucune réduction de domaine (cf. table 12). Le meilleur cas étant `five` avec "seulement" 57 % et le pire étant `queens 70 ff` avec 91 %. Nous allons donc étudier comment réduire le nombre de *Tell*s inutiles.

Programme	CHIP 3.2	`clp(FD)` 2.21	facteur accélération
`crypta`	0.120	0.090	1.33
`eq10`	0.170	0.110	1.54
`eq20`	0.300	0.170	1.76
`alpha`	61.800	9.290	6.65
`alpha ff`	0.280	0.160	1.75
`queens 16`	2.830	1.620	1.74
`queens 64 ff`	0.990	0.220	4.50
`queens 70 ff`	42.150	47.960	$\downarrow$ 1.13
`queens 81 ff`	1.620	0.430	3.76
`five`	0.030	0.010	3.00
`cars`	0.120	0.040	3.00

Tableau 11 : version de base de `clp(FD)` versus CHIP (temps en sec.)

Implantation de base			Décomposition des *Tell*s				
Programme	Temps exéc.	*Tell* (nombre)	Réduc domaine	Vérif domaine	Vérif entier	Echec entier	Echec domaine
`crypta`	0.090	8919	2073	4087	2707	11	41
`eq10`	0.110	15746	3018	8679	4000	6	43
`eq20`	0.170	24546	5154	12497	6846	12	37
`alpha`	9.290	904936	254349	348261	293866	3810	4630
`alpha ff`	0.160	15124	2668	7793	4646	14	3
`queens 16`	1.620	64619	21132	6954	34700	834	999
`queens 64 ff`	0.220	4556	1813	276	2446	2	1
`queens 70 ff`	47.960	2009404	171859	81159	1747810	5387	3189
`queens 81 ff`	0.430	10633	3004	609	7011	6	3
`five`	0.010	566	227	52	273	14	0
`cars`	0.040	2483	402	1271	810	0	0

Tableau 12 : décomposition des *Tell*s dans la version de base

4.3.3 Optimisations

Nous allons déterminer les sources des appels inutiles à *Tell* et, dans certains cas, définir des optimisations pour les éviter. L'impact de celles-ci sera évalué en pourcentage de *Tell*s (totaux et inutiles) évités et en pourcentage de temps d'exécution économisé. La première mesure est intéressante parce que indépendante de la machine donc générale. La seconde mesure permet toutefois de se faire une idée de l'impact d'une optimisation sur le temps de calcul.

Equivalence de contraintes

Le fait d'écrire plusieurs contraintes X `in` r pour une même contrainte de haut niveau a pour conséquence que ces contraintes sont souvent équivalentes et donnent lieu à des appels inutiles.

Considérons la contrainte $X = Y + 5$, (`'x=y+c'(X,Y,5)`, cf. exemple 2.1) dans le *store* courant :

{`X in 5..15, Y in 0..10`}

donnant :

{`X in 5..15`, `Y in 0..10`,
`X in min(Y)+5..max(Y)+5` (C_X),
`Y in min(X)-5..max(X)-5` (C_Y)}

Que se passe-t-il lors de l'ajout de la contrainte `X in 12..100` ? X est initialisé à 12..15 donc son *min* est propagé à Y via C_Y (`Y in 7..10`). Or, du fait que le *min* de Y a été modifié, C_X (`X in 12..15`) sera réexécuté inutilement (i.e. le $Tell$ ne modifie pas le domaine de X). Evidemment, il est inutile d'évaluer à nouveau X à partir de Y puisque Y vient juste d'être évalué à partir de X et que C_X et C_Y sont équivalents.

Optimisation 1 : *lors de l'ajout de la contrainte c, il est inutile de réexécuter c' si c' est équivalent à c.*

Dans la première version de `clp(FD)` (cf. [24]), cette optimisation était implantée. Il n'y avait pas à proprement parler de détection des contraintes équivalentes mais plutôt un impératif pour l'utilisateur : toutes les contraintes définies dans une même clause devaient être équivalentes. Du fait que toutes ces contraintes se partageaient le même environnement (pointé par `AF`) le test d'équivalence revenait à tester l'adresse des environnements nécessaires aux contraintes impliquées. L'économie réalisée était de l'ordre de 18 % de $Tell$ représentant 12 % du temps d'exécution dans le meilleur des cas (équations linéaires) mais était nulle dans le pire des cas (ex. `queens`).

Cette optimisation a été abandonnée dans la version actuelle de `clp(FD)` car d'une part son gain en temps d'exécution est marginal et d'autre part elle est soumise à des conditions assez fortes (ex. inutilisable avec des divisions à cause des arrondis) et/ou difficiles à contrôler (ex. équivalence des contraintes écrites par l'utilisateur). Enfin, la file de propagation devait contenir des triplets de la forme <*variable X,* `AF`*, listes à réactiver*> pour permettre de réactiver les listes (indiquées) de contraintes dépendant de X en testant l'adresse des environnements pour détecter l'équivalence. De ce fait, plusieurs triplets pour la même variable (avec des pointeurs d'environnements différents) pouvaient apparaître dans la file.

La suppression de cette optimisation nous permet désormais d'utiliser une file dont les éléments sont des couples de la forme <*variable, listes à réactiver*>. Il est aisé d'assurer qu'il y a au plus un seul couple pour toute variable X (en regroupant les listes à réactiver en présence de plusieurs couples pour une même variable). De ce fait, *la taille de notre file de propagation est bornée par le nombre de variables.* Il est alors possible de représenter cette

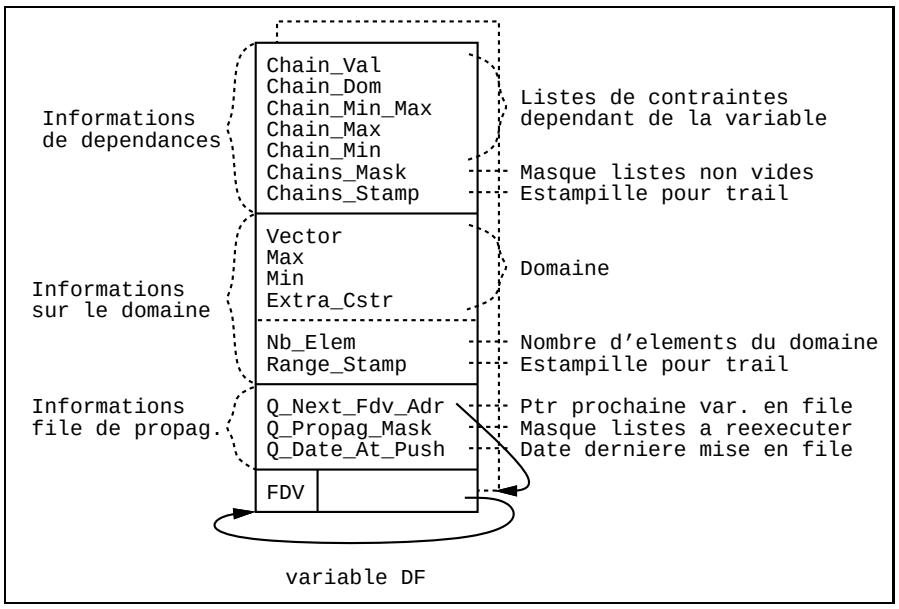


Figure 12 : nouvelle représentation interne d'une variable DF

file directement au travers des variables DF. La file ne nécessite plus d'allocation explicite de mémoire grâce à un chaînage des variables. Lorsqu'une variable DF est modifiée elle est juste chaînée aux autres variables dont certaines contraintes doivent être réactivées. Deux cas peuvent alors se produire :

- la variable est déjà chaînée, il suffit de mettre à jour la liste des contraintes à réveiller.
- la variable n'est pas encore chaînée, il suffit de la chaîner.

Notons que ce procédé évite des mises en file multiples d'une même variable donc de mêmes contraintes (cf. optimisation 3). Le moyen le plus efficace de tester si une variable est chaînée consiste à dater toute mise en file. Un registre `DATE` est alors ajouté et est incrémenté à chaque appel de plus haut niveau d'une contrainte (i.e. par `fd_call_constraint` et non pas à chaque appel issu de la propagation). Ainsi, une variable X est chaînée si $Date_At_Push(X) = DATE$. La représentation interne d'une variable est donc étendue pour prendre en compte les informations de chaînage (cf. figure 12). Les registres `BP` et `TP` sont toujours utilisés mais pointent désormais la première et la dernière variable DF de la file.

Impact : la table 13 présente les gains de cette nouvelle organisation (cf. table 14 pour plus d'informations). La diminution moyenne de 5 % sur le nombre de $Tell$s inutiles est

due aux ré-exécutions évitées et sera expliquée lors de la présentation de l'optimisation 3. Le bénéfice moyen de 16 % sur le temps d'exécution est en grande partie (environ 10 %) dû aux simplifications de certaines opérations inhérentes à la nouvelle file.

	nb. de *Tell*s		Temps
	total	inutile	exéc.
Pire (`queens 70 ff`)	0 %	0 %	12 %
Moyen	4 %	5 %	16 %
Meilleur (`alpha ff`)	13 %	16 %	25 %

Tableau 13 : gain de la file optimisée

File optimisée			Décomposition des *Tell*s				
Programme	Temps exéc.	*Tell* (nombre)	Réduc domaine	Vérif domaine	Vérif entier	Echec entier	Echec domaine
`crypta`	0.080	8302	2074	3754	2422	11	41
`eq10`	0.090	14341	2995	7792	3505	8	41
`eq20`	0.140	22059	5026	11164	5820	13	36
`alpha`	8.030	871838	254938	324838	283622	3817	4623
`alpha ff`	0.120	13176	2762	6392	4005	13	4
`queens 16`	1.390	64619	21132	6954	34700	834	999
`queens 64 ff`	0.170	4556	1813	276	2446	2	1
`queens 70 ff`	41.970	2009404	171859	81159	1747810	5387	3189
`queens 81 ff`	0.340	10633	3004	609	7011	6	3
`five`	0.010	558	225	52	267	14	0
`cars`	0.040	2439	402	1265	772	0	0

Tableau 14 : décomposition des *Tell*s avec une file optimisée

Satisfaction de contraintes

Une autre source d'appels inutiles à *Tell* est due aux contraintes satisfaites qu'il est alors inutile de remettre en cause. Considérons par exemple la contrainte $X \neq Y$ (`'x≠y'(X,Y)`) dans le *store* :

{`X in 1..10, Y in 1..10`}

donnant :

{`X in 1..10, Y in 1..10,`

`X in -{val(Y)}` (C_X),
`Y in -{val(X)}` (C_Y)}

Quand X est initialisé à 5, C_Y est réveillé et 5 est supprimé du domaine de Y fournissant le *store* suivant :

{`X=5`, `Y in 1..4:6..10`, C_X, C_Y}

Supposons alors que la contrainte `Y=8` soit ajoutée. Après modification du domaine de Y, la phase de propagation ré-exécutera C_X vérifiant inutilement que $5 \neq 8$. En effet, la contrainte C_X est désormais satisfaite puisque 5 n'appartient plus au domaine de Y.

Du fait qu'un résolveur sur les DF basé sur la propagation locale n'est pas complet, il ne serait pas réaliste de vouloir détecter "au plus juste" la satisfaction d'une contrainte. Ceci entraînerait souvent l'énumération des variables à chaque $Tell$. Ainsi, la meilleure manière de faire consiste à utiliser une approximation de la condition de satisfaction. Au chapitre 6 nous détaillerons ce principe. Pour l'instant, considérons que la seule approximation (i.e. condition suffisante) pour détecter la satisfaction d'une contrainte X `in` r est un test de clôture sur X, i.e. si X est clos dans S alors S satisfait X `in` r. Ainsi, dans l'exemple précédent, quand la contrainte `Y=8` est ajoutée, `X in -{val(Y)}` est détectée comme étant satisfaite car X est clos. Evidemment, ceci n'est vrai que si X est devenu clos *avant* (et non pas *pendant*) la phase de propagation courante (i.e. toutes les propagations dues à la réduction de X doivent avoir été effectuées).

Optimisation 2 : *il est inutile de ré-exécuter une contrainte* X `in` r *si* X *a été instanciée avant la phase de propagation courante.*

Le registre `DATE` introduit pour la gestion de notre file optimisée est réutilisé pour dater l'instanciation finale d'une variable DF. Une nouvelle cellule est prévue pour cela dans la représentation des variables FD (cf. figure 13). Par la suite une contrainte X `in` r n'est pas ré-exécutée si $ground(X) \wedge INT_Date(X) < DATE$. Pour simplifier ce test $INT_Date(X)$ vaut la plus grande valeur possible tant que X n'est pas définitivement instancié.

Impact : la table 15 présente les gains obtenus avec cette optimisation. Elle est extrêmement performante pour les programmes utilisant la résolution par *forward checking* (ex. 94 % des $Tell$s inutiles évités pour `queens 70 ff` correspondant à un gain de 74 %) et moins pour les équations linéaires (gain moyen de 23 % des $Tell$s inutiles correspondant à

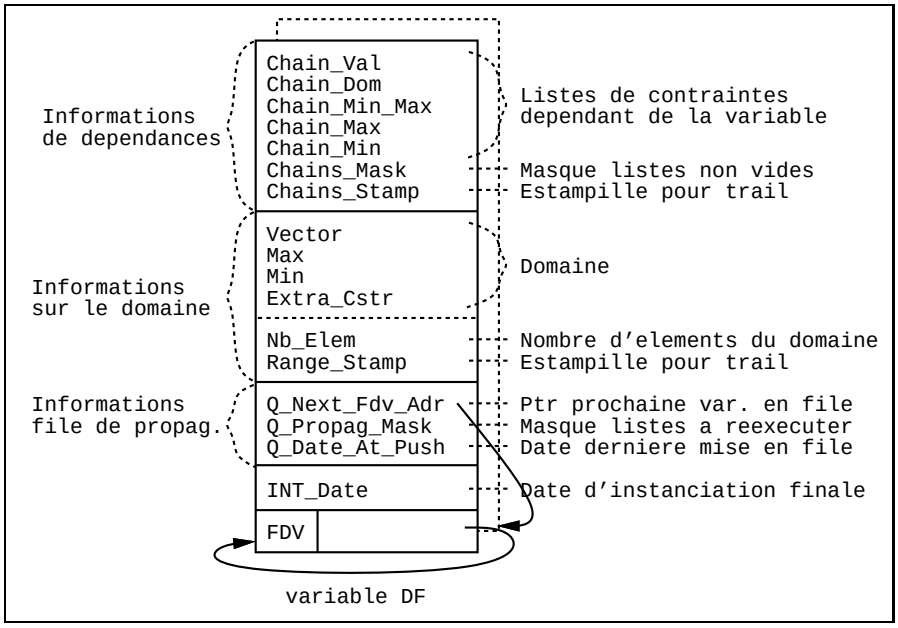


Figure 13 : représentation interne définitive d'une variable DF

6 % du temps d'exécution).

	nb. de *Tell*s		Temps
	total	inutile	exéc.
Pire (`cars`)	5 %	6 %	25 %
Moyen	34 %	47 %	24 %
Meilleur (`queens 70 ff`)	85 %	94 %	74 %

Tableau 15 : gain de l'optimisation 2

Occurrences multiples de contraintes en propagation

La dernière source de réactivations inutiles de contraintes vient du fait qu'il y a dans la file de propagation plusieurs occurrences d'une même contrainte. Nous pouvons distinguer deux sources pour ce phénomène :

(a) raffinements successifs d'une même variable entraînant des occurrences multiples des contraintes dépendant de cette variable.

(b) raffinements de variables distinctes dont dépend une même contrainte entraînant des occurrences multiples de cette contrainte.

Du fait que l'ordre dans lequel ces contraintes seront réveillées n'a pas d'importance du point de vue de la correction, cela peut conduire à des réactivations inutiles : à tout instant il suffit qu'au plus une occurrence de chaque contrainte soit en file.

Optimisation 3 : *si une contrainte est déjà présente en file de propagation il est inutile de l'y ajouter à nouveau.*

Dans la version initiale de `clp(FD)` [24] l'optimisation 3 était implantée en datant les mises en file des contraintes ainsi que les exécutions de contraintes (grâce à un mot supplémentaire dans la représentation des contraintes). L'exécution d'une contrainte c en file était alors inutile si $Date_Exec(c) > Date_Mise_En_File(c)$. Dans le meilleur cas, le gain était alors de 26 % de $Tell$s correspondant à 17 % du temps d'exécution et nul dans le pire des cas.

Dans la version courante de `clp(FD)`, ceci a été abandonné pour les raisons suivantes :

- les ré-exécutions du type (a) sont automatiquement évitées grâce à notre gestion de la file de propagation au travers des variables DF puisque, lors de l'ajout en file d'une variable, si celle-ci s'y trouve déjà seul le masque des listes à ré-exécuter est mis à jour (*ou* logique).

- il était nécessaire de dater toutes les exécutions de contraintes (y compris celles dues à la propagation) ce qui n'est pas nécessaire pour les dates requises par la file et par l'optimisation 2. De ce fait, dès qu'un calcul devenaint long le registre `DATE` débordait nécessitant un lourd traitement de remise à zéro de toutes les dates.

- l'élimination systématique des "doublons" permet d'éviter les réactivations du type (b) (et que celles-ci) mais n'offre qu'un gain marginal du fait que l'ordre d'évaluation des contraintes est modifié.

Pour illustrer ce dernier cas, considérons un ensemble de contraintes exprimées sous la forme $C_X(Y_1, ..., Y_n)$ pour une contrainte sur X dépendant des variables $Y_1 ... Y_n$. Soit alors les contraintes : $C_X(T)$, $C_Y(X)$, $C_Z(X, Y, T)$, $C_A(Z)$ et un réseau R de contraintes inter-dépendantes portant sur un ensemble de variables incluant A mais dont l'intersection avec $\{X, Y, Z, Y\}$ est vide. De ce fait la mise à jour de Z entraîne une mise à jour de A donc un re-calcul de R arbitrairement important mais qui n'influence aucune des variables X, Y, Z, T. Cet exemple met en évidence un impact négatif de l'optimisation 3, qui en ne rajoutant pas la contrainte $C_Z(X, Y, T)$ à la file de propagation sous prétexte qu'elle s'y trouve déjà,

engendre de deux calculs de R au lieu d'un seul. Le point de départ est dû à une modification de T .

Sans optimisation 3 :

Etape	Exécution de	utile ?	file de propagation
1	T *in ...*	oui	$-\!\!- C_X(T)\ C_Z(X,Y,T)$
2	$C_X(T)$	oui	$C_Z(X,Y,T) -\!\!- C_Y(X)\ C_Z(X,Y,T)$
3	$C_Z(X,Y,T)$	oui	$C_Y(X)\ C_Z(X,Y,T) -\!\!- C_A(Z)$
4	$C_Y(X)$	oui	$C_Z(X,Y,T)\ C_A(Z) -\!\!- C_Z(X,Y,T)$
5	$C_Z(X,Y,T)$	oui	$C_A(Z)\ C_Z(X,Y,T) -\!\!- C_A(Z)$
6	$C_A(Z)$	oui	$C_Z(X,Y,T)\ C_A(Z) -\!\!- R$
7	$C_Z(X,Y,T)$	non	$C_A(Z)\ R -\!\!-$
8	$C_A(Z)$	non	$R -\!\!-$
9	R	oui	$-\!\!- propag_R$
10	$propag_R$	oui	$-\!\!-$

Nous pouvons distinguer 3 occurrences de $C_Z(X,Y,T)$ en file : (i) après modification de T (étape 1), (ii) après modification de X (étape 2) et (iii) après modification de Y (étape 4). Sans optimisation, c'est l'occurrence (ii) qui calcule la valeur finale de Z (donc de A en étape 6) avant de lancer le calcul de R qui sera alors définitif. L'occurrence (iii) correspond alors à une ré-exécution inutile (étape 7) de même que la seconde évaluation de A à l'étape 8. Avec l'optimisation 3, l'occurrence (ii) est supprimée et R est calculé une première fois à partir d'une valeur de Z intermédiaire (due à l'occurrence (i)) via une valeur intermédiaire de A. Z ne prend sa valeur définitive que lors de l'occurrence (iii) entraînant ainsi un re-calcul de R.

Avec optimisation 3 :

Etape	Exécution de	utile ?	file de propagation
1	T *in ...*	oui	$-\!\!- C_X(T)\ C_Z(X,Y,T)$
2	$C_X(T)$	oui	$C_Z(X,Y,T) -\!\!- C_Y(X)$ ($C_Z(X,Y,T)$ non rajouté)
3	$C_Z(X,Y,T)$	oui	$C_Y(X) -\!\!- C_A(Z)$
4	$C_Y(X)$	oui	$C_A(Z) -\!\!- C_Z(X,Y,T)$
5	$C_A(Z)$	oui	$C_Z(X,Y,T) -\!\!- R$
6	$C_Z(X,Y,T)$	oui	$R -\!\!- C_A(Z)$
7	R	oui	C_A(Z) $-\!\!- propag_R$
8	$C_A(Z)$	oui	propag_$R -\!\!- R$
9	$propag_R$	oui	$-\!\!- R$
10	R	oui	$-\!\!- propag_R$
11	$propag_R$	oui	$-\!\!-$

Notre exemple artificiel est un cas pathologique pour l'optimisation 3. En moyenne celle-ci réduit tout de même le nombre d'appels inutiles à *Tell* tout en augmentant légèrement le nombre de *Tell*s utiles (cf. l'exemple). Cette optimisation est intéressante lorsqu'elle prend en charge les ré-exécutions inutiles de type (a) et (b) comme c'était les cas dans notre première version mais devient peu convaincante lorsqu'elle ne s'occupe que des réévaluations du type (b) comme c'est le cas avec une file "optimisée".

4.3.4 Evaluation de l'implantation finale

Nous avons identifié trois sources d'appels inutiles à *Tell*. Les optimisations visant à éviter ces appels profitent à toutes les contraintes et sont de ce fait *globales* à l'opposé des optimisation *locales* (i.e. ad-hoc) des résolveurs boîte noire. Remarquons que l'exécution d'une contrainte est très rapide (ex. sur un Sparc 2 à 40 Mhz il s'en exécute jusqu'à 80000 par seconde). De plus, comme nous le verrons ci-dessous, les *Tell*s inutiles sont beaucoup moins coûteux que les appels inévitables. De ce fait, l'ajout d'une optimisation ne doit pas entraîner de surcoût trop fort pour avoir un impact sur le temps d'exécution. Par rapport à la première version de `clp(FD)` qui optimisait les trois cas d'appels inutiles nous n'avons retenu que l'optimisation 2 (satisfaction de contrainte) qui est celle dont l'impact est le plus important (ex. `queens 70 ff` environ 4 fois plus rapide grâce à cette optimisation). L'abandon de l'optimisation 1 (contraintes équivalentes) nous a permis d'optimiser notre file de propagation. L'adoption de celle-ci permet un gain en temps de 10 % grâce à la simplification de certaines opérations et prend en charge implicitement un certain cas d'optimisation 3 (occurrences multiples de contraintes). L'optimisation des autres cas n'apporte qu'un gain marginal du fait du changement de l'ordre d'évaluation. Nous avons donc une fois de plus privilégié la simplicité de l'architecture lorsque seul un gain marginal était au rendez-vous. Analysons de plus près les résultats de notre nouvelle approche.

La table 16 analyse les issues des opérations *Tell*s pour la version finale (i.e. avec file optimisée et optimisation 2). L'impact combiné de la file optimisée et de l'optimisation 2 est résumé en table 17 (cf. table 18 pour plus de détails). La figure 14 permet d'évaluer graphiquement l'apport des optimisations.

En moyenne, il est possible de réduire de moitié (52 %) le nombre de *Tell*s ramenant ainsi la proportion de *Tell*s inutiles par rapport aux utiles à 54 % (cette proportion était de

File optimisée + optim 2			Décomposition des *Tell*s				
Programme	Temps exéc.	*Tell* (nombre)	Réduc domaine	Vérif domaine	Vérif entier	Echec entier	Echec domaine
`crypta`	0.070	7057	2074	3754	1177	11	41
`eq10`	0.080	12479	2995	7792	1643	7	42
`eq20`	0.130	18963	5026	11164	2724	11	38
`alpha`	7.770	641204	254938	324838	52988	2023	6417
`alpha ff`	0.110	10429	2762	6392	1258	7	10
`queens 16`	0.890	33481	21132	6954	3562	834	999
`queens 64 ff`	0.130	2143	1813	276	33	2	1
`queens 70 ff`	11.070	292381	171859	81159	30787	5387	3189
`queens 81 ff`	0.210	3787	3004	609	165	6	3
`five`	0.010	397	225	52	106	14	0
`cars`	0.030	2324	402	1265	657	0	0

Tableau 16 : décomposition des *Tell*s dans la version finale

	nb. de *Tell*s		Temps
	total	inutile	exéc.
Pire (`cars`)	6 %	8 %	25 %
Moyen	37 %	52 %	36 %
Meilleur (`queens 70 ff`)	85 %	94 %	77 %

Tableau 17 : gain de la version finale

	Nombre total de *Tell*s			Nombre de *Tell*s inutiles			Temps d'exécution		
Programme	Initial	Final	Gain	Initial	Final	Gain	Initial	Final	Gain
`crypta`	8919	7057	21 %	6794	4931	27 %	0.090	0.070	22 %
`eq10`	15746	12479	21 %	12679	9435	26 %	0.110	0.080	27 %
`eq20`	24546	18963	23 %	19343	13888	28 %	0.170	0.125	26 %
`alpha`	904936	641204	29 %	642127	377826	41 %	9.290	7.770	16 %
`alpha ff`	15124	10429	31 %	12439	7650	38 %	0.160	0.110	31 %
`queens 16`	64619	33481	48 %	41654	10516	75 %	1.620	0.890	45 %
`queens 64 ff`	4556	2143	53 %	2722	309	89 %	0.220	0.130	41 %
`queens 70 ff`	2009404	292381	85 %	1828969	111946	94 %	47.960	11.070	77 %
`queens 81 ff`	10633	3787	64 %	7620	774	90 %	0.430	0.210	51 %
`five`	566	397	30 %	325	158	51 %	0.010	0.010	0 %
`cars`	2483	2324	6 %	2081	1922	8 %	0.040	0.030	25 %
Gain moyen			37 %			52 %			36 %

Tableau 18 : détail du gain final

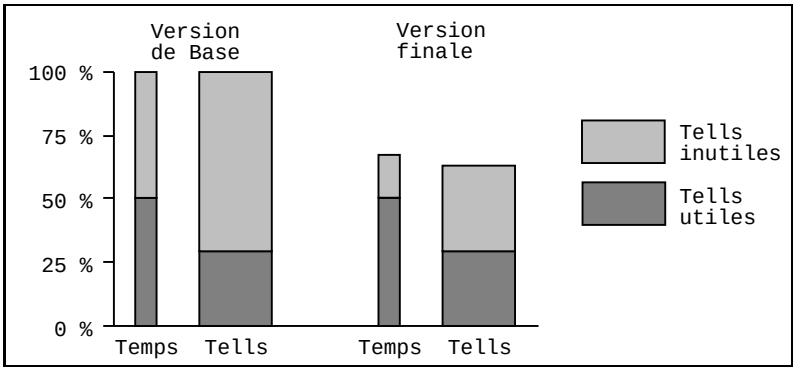


Figure 14 : impact des optimisations

72 % dans la version initiale). Parmi ceux-ci nous pensons qu'un nombre important peut être encore éliminé par une optimisation 2 plus précise (cf. chapitre 6).

L'économie de temps d'exécution est d'environ 36 %. Une partie de celle-ci est due aux simplifications inhérentes à l'architecture de file optimisée (environ 10 %). Ceci peut être confirmé en se référant à la table 15 qui montre qu'une diminution de 47 % du nombre d'appels inutiles offre un gain de 24 %. Donc le gain propre à la diminution de 52 % est de 26 % (ce qui vérifie les 36% - 10 %). Ceci nous donne deux informations importantes :

- la simplification de certaines opérations a permis d'économiser environ 10 % du temps d'exécution. Ceci s'explique par le grand nombre de fois où les contraintes sont exécutées. De petites modifications peuvent avoir de grandes répercussions.
- puisque un gain de 52 % de $Tell$s inutiles se reflète par un gain en temps de 26 % nous en déduisons que dans la version initiale les 72 % d'appels inutiles correspondaient à 50 % du temps d'exécution.
- le gain total possible en temps d'exécution est donc de 50 %. Ce gain est à comparer à nos 26 %. Là encore une meilleur optimisation 2 devrait nous permettre de nous rapprocher de ces 50 %.
- puisque ces 72 % d'appels inutiles consomment autant de temps que les 28 % d'appels utiles, nous en déduisons qu'un appel utile est en moyenne 2.5 fois plus coûteux qu'un appel inutile.

Le fait qu'un appel utile soit environ 2.5 fois plus coûteux qu'un appel inutile est dû au peu de travail que nécessite ce dernier puisqu'il n'évalue que le domaine dénoté par r avant

de détecter que X n'est pas réduit. En particulier, dans le cas d'équations (résolues par *lookahead partiel*) les évaluations de r ne font intervenir que des opérations arithmétiques sur des entiers, ce qui est généralement très rapide (cf. section 4.2).

La table 19 compare la version finale de `clp(FD)` à la version commerciale 3.2 de CHIP sur un Sparc 2. Nous incluons également le benchmark `bridge` [67]. Sur les équations linéaires `clp(FD)` est environ 2 fois plus rapide que CHIP (avec des pointes à 8) et sur les autres problèmes le facteur moyen est de l'ordre de 4 en faveur de `clp(FD)`.

Programme	CHIP 3.2	`clp(FD)` 2.21	facteur accélération
`crypta`	0.120	0.070	1.71
`eq10`	0.170	0.080	2.12
`eq20`	0.300	0.130	2.30
`alpha`	61.800	7.770	7.95
`alpha ff`	0.280	0.110	2.54
`queens 16`	2.830	0.890	3.17
`queens 64 ff`	0.990	0.130	7.61
`queens 70 ff`	42.150	11.070	3.80
`queens 81 ff`	1.620	0.210	7.71
`five`	0.030	0.010	3.00
`cars`	0.120	0.030	4.00
`bridge`	2.750	0.640	4.29

Tableau 19 : `clp(FD)` versus CHIP (temps en sec.)

Nous pouvons également comparer `clp(FD)` au compilateur CHIP. Celui-ci n'est pas disponible mais [2] présente les temps pour le problème `queens` et de `bridge` sur un Sparc 1+ (les temps ont donc été normalisés par un facteur 1.6). Sur ces exemples, `clp(FD)` est environ 3 fois plus rapide que le compilateur CHIP (cf. table 20).

Programme	CHIP compil.	`clp(FD)` 2.21	facteur accélération
`queens 16 ff`	0.040	0.010	4.00
`queens 64 ff`	0.490	0.130	3.76
`queens 256 ff`	14.560	6.930	2.10
`bridge`	2.068	0.640	3.23

Tableau 20 : `clp(FD)` versus compilateur CHIP (temps en sec.)

Chapitre 5

Contraintes booléennes

Un exemple intéressant pour montrer la flexibilité de l'approche RISC est l'étude des contraintes booléennes : celles-ci (*et*, *ou* et *non*, pour rester simple) sont à valeur dans un domaine fini ($0..1$) mais sont cependant différentes des contraintes usuelles sur les DF. Il est donc intéressant de voir s'il est possible d'encoder efficacement ces contraintes en contraintes primitives X `in` r et de comparer le résolveur booléen ainsi obtenu avec les autres résolveurs existants utilisant des méthodes et algorithmes complètement différents. La résolution des contraintes booléennes est un problème déjà ancien mais qui nourrit des recherches toujours très actives. De nombreuses méthodes ont été développées, soit générales soit pour des types particuliers de formules. Il y a quelques années l'utilisation de techniques de propagation locale a été proposée par le langage CHIP, qui en fait dispose de deux résolveurs booléens : l'un basé sur l'unification booléenne, et l'autre utilisant la propagation locale et réutilisant certaines procédures du résolveur sur les domaines finis. Il s'avère en fait que pour beaucoup d'applications le résolveur utilisant la propagation est bien plus efficace que l'autre, à tel point que dans CHIP il est le résolveur par défaut pour les booléens.

Il est en fait très facile de définir les opération booléennes de base (*et*, *ou* et *non*) en termes de contraintes primitives. Le résolveur booléen est réduit à moins de 10 lignes de code `clp(FD)` ! C'est-à-dire à 3 définitions de contraintes en termes de X `in` r. Notons en outre que cet encodage se fait à un niveau plus bas qu'une simple transformation

*le contenu de ce chapitre a été publié dans [26, 27, 28].

des contraintes booléennes en expressions arithmétiques par exemple et qu'on peut ainsi espérer une plus grande efficacité. En outre, ce résolveur est ouvert à l'utilisateur qui peut ajouter de nouvelles contraintes, pour intégrer par exemple directement des contraintes d'implication, d'équivalence, etc.

Ceci est cependant assez évident, du fait que les booléens sont un cas particulier de domaines finis, et peut sembler un simple exercice d'école. Le plus surprenant cependant est que le résolveur ainsi réalisé soit très efficace : il est plus rapide que le résolveur de CHIP d'environ un ordre de magnitude, et se compare favorablement à la plupart des résolveurs booléens ad hoc basés sur d'autres algorithmes, comme les méthodes énumératives, les BDD ou les techniques de recherche opérationnelle. Notons finalement que ces résolveurs utilisent en général de nombreuses heuristiques pour améliorer les performances, alors que notre résolveur n'en a pour l'instant aucune, et que l'on peut donc encore espérer une amélioration des performances.

5.1 Un panorama des résolveurs booléens

Bien que le problème de la satisfiabilité des formules booléennes ne soit pas nouveau, la conception de méthodes efficaces est encore un champ de recherche actif. Par ailleurs, il faut noter qu'il est très souvent important de pouvoir connaître les modèles (i.e. les affectations des variables), quand ils existent, de ces formules. A cette fin, plusieurs méthodes, basées sur des structures de données et sur des algorithmes très différents, ont été proposées et nous allons maintenant nous employer à classifier ces résolveurs booléens à partir de systèmes effectivement implantés.

5.1.1 Méthodes basées sur la résolution

La méthode de résolution, originellement conçue pour la logique des prédicats, peut évidemment être spécialisée au cas qui nous intéresse, i.e. propositionnel. Cette méthode utilise une représentation clausale des formules booléennes où chaque littéral représente une variable booléenne ; il s'agit donc d'opérer en forme normale conjonctive. Le principe de la méthode consiste en une suite d'étapes dont chacune d'entre elles prend deux clauses contenant des occurrences opposées et produit une nouvelle clause logiquement équivalente

à la conjonction des deux précédentes ; ce processus se poursuit jusqu'à production de la clause vide (ou inconsistance) ou de quelque conséquence logique recherchée. Par exemple, une certaine forme de cette méthode (dite SL-résolution), apparaît dans la version actuelle de Prolog-III [31] [4]. Cependant, les faibles performances de ce résolveur limitent son emploi à de petits problèmes. Plusieurs raffinements ont été proposés en vue de limiter l'espace de recherche potentiellement énorme de cette méthode et on renverra à [56] pour plus ample information. Retenons simplement qu'il ne semble pas y avoir d'amélioration possible importante dès qu'on sort de petites classes de problèmes.

5.1.2 Méthodes basées sur les diagrammes de décisions binaires (BDD)

Les diagrammes de décisions binaires (BDD) ont connu récemment un grand succès en tant que moyen efficace de coder les fonctions booléennes [13] et il était naturel d'essayer de les utiliser dans un résolveur booléen. L'idée de base des BDD consiste à avoir une représentation compacte de la forme normale de Shanon d'une formule booléenne. Une formule a une telle forme normale si elle est réduite à une constante (0 ou 1) ou à une expression de la forme $ite(x, F, G)$, pour *"if x then F else G"*, où F et G sont en forme normale. L' expression *if-then-else* de cette forme $ite(x, F, G)$ représente la formule $(x \wedge F) \vee (\neg x \wedge G)$. Une manière efficace de coder et de manipuler cette forme normale utilise un BDD réduit et ordonné qui sera représenté par un DAG dont les noeuds intérieurs sont étiquetés par les variables et les feuilles par les constantes. Un noeud intérieur x ayant deux fils F et G représente une expression *if-then-else* $ite(x, F, G)$. En établissant un ordre total sur les variables booléennes, il est possible de construire pour toute formule booléenne un BDD respectant cet ordre (au sens suivant : $x < y$ ssi il existe un chemin de x à y) et partageant (i.e. factorisant) les sous-expressions communes. La forme normale ainsi obtenue est unique. A titre d'exemple la formule $F = (x \wedge y) \vee z$ et l'ordre $x < y < z$ donnent le BDD de la figure 15.

Notons que la taille et la forme des BDD dépendent fortement de l'ordre des variables choisi car un "bon" ordre permettra de partager un plus grand nombre de sous-expressions. De la sorte, le nombre des noeuds d'un BDD en fonction du nombre de variables de la formule peut varier de linéaire à exponentiel.

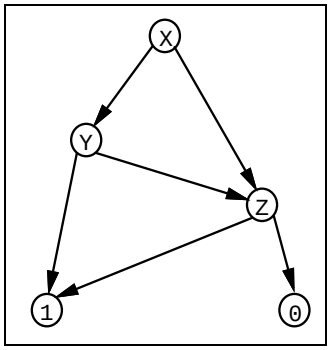


Figure 15 : BDD représentant la formule $(x \wedge y) \vee z$

Les BDD ont été utilisés dans de nombreux résolveurs, ainsi l'unification booléenne [50] du résolveur de CHIP utilise une telle représentation des formules booléennes [14] [65]. Citons encore le résolveur d'Adia [57], sa version améliorée (la deuxième méthode de [59]) et la combinaison de `wamcc`, cf. section 3 et de Adia [35]. Ce dernier système consiste en l'intégration d'un résolveur booléen basé sur les BDD à l'intérieur d'un compilateur Prolog basé sur la WAM. Ses performances sont quatre fois meilleures que celles de l'unification booléenne de CHIP [35]. De tels résolveurs sont efficaces pour des applications telles que la vérification de circuits booléens mais, dès que les problèmes ne sont plus aussi symétriques, il y a dégradation des performances car les BDD construits durant le calcul sont extrêmement grands. Il est de plus coûteux de maintenir (i.e. recalculer) une forme normale telle que les BDD à chaque fois qu'une nouvelle contrainte est ajoutée. Cette absence d'incrémentalité se retrouve au niveau du choix de l'ordre sur les variables dont on a précédemment dit l'importance : il n'est pas vraiment possible d'avoir dans les langages de contraintes des heuristiques complexes sur cet ordre. Dernier inconvénient, l'impossibilité de ne calculer qu'une seule solution (au lieu de toutes).

5.1.3 Méthodes énumératives

En gros, ces méthodes (qui incluent le célèbre algorithme Davis-Putman) consistent à essayer les différentes affectations possibles en instanciant incrémentalement les variables (à 0 ou à 1) et en testant la satisfiabilité de manière plus ou moins sophistiquée. L'idée principale consiste donc à construire, implicitement ou explicitement, un arbre de décision

en instanciant les variables et en le parcourant par backtracking. La consistance des contraintes booléennes est testée dès que leurs variables sont closes. [56] et [51] décrivent des améliorations possibles de tels tests et [58] montre comment calculer des unificateurs principaux de tous les modèles. Afin d'être plus efficaces, de nouvelles méthodes utilisent une forme clausale matricielle pour représenter les contraintes booléennes, citons les vecteurs de bits de [51] et les matrices creuses de la première méthode de [59]. In fine, remarquons là encore la possibilité d'agrémenter ces méthodes d'heuristiques variées...

5.1.4 Programmation en nombres entiers 0-1

On a récemment proposé une tout autre méthode qui consiste à transformer un problème de satisfaction de contraintes, et en particulier un problème sur les booléens, en un problème de résolution d'ensemble d'inéquations linéaires en nombres entiers tel que la satisfiabilité du problème initial se ramène à un problème d'optimisation dudit ensemble d'inéquations [43]. On pourra utiliser pour ce dernier problème des méthodes traditionnelles de recherche opérationnelle telles que les méthodes branch-and-cut de la programmation linéaire en 0-1. Partant de la forme clausale, on traduira chaque clause de manière immédiate. Par exemple, la clause $x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4$ se traduira en l'inéquation linéaire $x_1 + (1 - x_2) + x_3 + (1 - x_4) \geq 1$, i.e. $x_1 - x_2 + x_3 - x_4 \geq -1$. On arrivera à une solution ou à l'inconsistance en dérivant de nouvelles inéquations, ce qui sera fait en éliminant variable après variable par combinaisons linéaires d'inéquations, d'une manière rappelant effectivement la résolution. Une méthode apparentée apparaît dans [5, 6], où diverses heuristiques sont utilisées dans le choix de la prochaine variable à éliminer et sont encodées dans une fonction d'optimisation qui guidera la recherche vers une solution optimale. La méthode consiste donc à générer une suite de vecteurs $X_1, ..., X_k$ telle que chacun des X_k ait ses éléments dans l'ensemble {0,1} et soit une solution optimale (satisfaisant les contraintes initiales).

Une telle méthode peut s'avérer très efficace, surtout pour de gros problèmes, et apparaît plus adaptée à la recherche d'une seule solution plutôt que de toutes.

5.1.5 Méthodes basées sur la propagation.

Elles reposent sur des techniques de propagation locale développées pour les domaines finis qui son nées des problèmes de satisfaction de contraintes et ont été introduits en PLC par CHIP. L'idée de base consiste ici à gérer un réseau de contraintes entre un ensemble de variables pouvant prendre leurs valeurs dans un ensemble fini de constantes, en assurant une consistance locale et en propageant celle-ci à travers les contraintes reliant les variables. En PLC on n'implante habituellement que l'arc-consistance. Ainsi, la phase de propagation est suivie par une phase dite d'énumération où les variables non encore déterminées sont incrémentalement instanciées à quelque valeur de leur domaine réduit à la phase précédente. Diverses heuristiques peuvent être incorporées à ce stade en vue du choix de la prochaine variable à instancier. Une instanciation peut ainsi mener à la réactivation de contraintes non encore totalement satisfaites, donc à la réduction de domaines d'autres variables, etc. On continue jusqu'à trouver une solution.

Le langage CHIP par exemple incorpore un résolveur booléen basé sur ces techniques. Il s'avère en fait que le résolveur utilisant la propagation est bien plus efficace que celui basé sur l'unification booléenne, à tel point que dans CHIP il est le résolveur par défaut pour les booléens.

5.1.6 PLC versus résolveurs dédiés

Il faut aussi distinguer dans la classification précédente entre les résolveurs booléens dédiés, destinés à prendre en entrée un ensemble de formules booléennes, et les résolveurs qui sont intégrés dans des langages de PLC, car ceux-ci offrent une flexibilité bien plus grande en proposant un langage logique complet pour énoncer le problème et générer les formules booléennes à résoudre. Seuls PrologIII, CHIP et `clp(B/FD)` tombent dans cette dernière catégorie.

Les avantages de l'intégration dans un langage de PLC sont les suivants.

- d'abord, le langage logique peut être utilisé comme métalangage pour poser les contraintes booléennes au lieu de donner une formulation booléennes explicite, par exemple une forme clausale, qui est en général complexe et assez illisible.

- ensuite, les heuristiques de calcul peuvent être intégrées dans le programme lui-même, ceci étant donc à l'opposé d'un résolveur booléen fermé avec un nombre limité d'heuristiques pré-définies.
- enfin, ceci permet diverses extensions comme par exemple les pseudo-booléens [8] ou les logiques multi-valuées [72].

Les contraintes pseudo-booléennes en particulier sont très importantes car elles permettent en général une formulation plus simple et un élagage de l'espace de recherche plus important. Elles ouvrent aussi sur un domaine d'application important en recherche opérationnelle. Notons que les contraintes pseudo-booléennes sont immédiatement disponibles dans un système comme `clp(B/FD)`.

5.2 Contraintes booléennes

Définition 5.1 *Soit $\mathcal{V}$ un ensemble énumérable de variables. Une* **contrainte booléenne** *sur $\mathcal{V}$ est une des formules suivantes :*

$$and(X,Y,Z)\ ,\ or(X,Y,Z)\ ,\ not(X,Y)\ ,\ X=Y\ ,\quad \text{pour } X,Y,Z \in \mathcal{V} \cup \{0,1\}$$

La signification intuitive de ces formules est : $X \wedge Y \equiv Z$, $X \vee Y \equiv Z$, $X \equiv \neg Y$, et $X \equiv Y$. On notera $\mathcal{B}_{\mathcal{V}}$ l'ensemble des contraintes booléennes sur $\mathcal{V}$, et on utilisera par la suite $\mathcal{B}$, lorsque l'ensemble de variables n'a pas d'importance particulière.

Définissons maintenant les règles de propagation (de valeurs) pour les contraintes booléennes.

Définition 5.2 *Soit B la théorie du premier ordre sur les formules présentée en table 21.*

Notons qu'il est facile d'enrichir, si besoin est, ce système de contraintes par d'autres contraintes booléennes telles que *xor* (ou exclusif), *nand* (non et), *nor* (non ou), $\Leftrightarrow$ (équivalence), ou $\Rightarrow$ (implication), en donnant les règles correspondantes. Mais ces contraintes peuvent aussi être définies grâce aux contraintes de base.

On peut alors définir une relation de satisfaction $\vdash_B$ entre les contraintes booléennes, donc par suite un système de contraintes selon le formalisme présenté au chapitre 2.

0=0	1=1
and(X,Y,Z), X=0 → Z=0	and(X,Y,Z), Y=0 → Z=0
and(X,Y,Z), X=1 → Z=Y	and(X,Y,Z), Y=1 → Z=X
and(X,Y,Z), Z=1 → X=1	and(X,Y,Z), Z=1 → Y=1
or(X,Y,Z), X=1 → Z=1	or(X,Y,Z), Y=1 → Z=1
or(X,Y,Z), X=0 → Z=Y	or(X,Y,Z), Y=0 → Z=X
or(X,Y,Z), Z=0 → X=0	or(X,Y,Z), Z=0 → Y=0
not(X,Y), X=0 → Y=1	not(X,Y), X=1 → Y=0
not(X,Y), Y=0 → X=1	not(X,Y), Y=1 → X=0

Tableau 21 : théorie de propagation booléenne B

Définition 5.3 *Considérons un store S et une contrainte booléenne b.*

$S \vdash_B b$ si et seulement si S implique b avec les axiomes auxiliaires de B.

Soit $\mathcal{B}_{\wedge,\exists}$ la clôture de $\mathcal{B}$ par conjonction et quantification existentielle, le système de contraintes booléennes *est $(\mathcal{B}_{\wedge,\exists}, \vdash_B)$.*

Ceci est la construction usuelle, cf. [63], pour formaliser un système de contraintes à partir d'une théorie du premier ordre.

Il est à noter que les règles de B (donc $\vdash_B$) encodent précisément les mécanismes de propagation qui seront utilisés pour simplifier et résoudre les contraintes booléennes. On a ainsi donné une sémantique opérationnelle au résolveur de contraintes. Il est cependant important de s'assurer que notre système de contraintes (défini opérationnellement) est bien équivalent aux expressions booléennes traditionnelles. Pour cela, nous devons prouver que la relation de satisfaction dérive les mêmes résultats que la sémantique déclarative des booléens donnée par les tables de vérité des opérations *and*, *or* et *not*.

Proposition 5.1 *Les contraintes $and(X,Y,Z)$, $or(X,Y,Z)$, et $not(X,Y)$ sont satisfaites pour des valeurs de X, Y et Z si et seulement si ce triplet de valeurs est donné dans la table de vérité de l'opération correspondante.*

Preuve :
Il faut montrer que, pour $and(X,Y,Z)$ et $or(X,Y,Z)$, dès que X et Y sont liés à une valeur, alors la valeur de Z est correcte, c'est-à-dire unique (si plusieurs règles peuvent se

déclencher, alors elles donnent toutes le même résultat) et égal à la valeur trouvée dans la table de vérité. Il faut aussi montrer que toutes les lignes de cette table sont effectivement calculées. Ceci est vérifié par une analyse de cas triviale. Pour $not(X, Y)$, on vérifie aisément que pour toute valeur de X, Y a la valeur opposée. □

5.3 Codage des booléens en `clp(FD)` : `clp(B/FD)`

Nous allons maintenant voir comment définir la traduction de chacune des contraintes booléennes en contraintes primitives de `clp(FD)`, c'est-à-dire en contraintes X `in` r. Nous prouverons également la correction et la complétude de ce résolveur, en montrant qu'il encode bien la "sémantique opérationnelle" définie par la théorie B.

La syntaxe des contraintes primitives X `in` r permet d'utiliser des opérations arithmétiques sur les bornes des domaines (`r`). Considérons donc certaines relations mathématiques satisfaites par les contraintes booléennes :

$and(X, Y, Z)$	satisfait	$Z = X \times Y$
		$Z \leq X \leq Z \times Y + 1 - Y$
		$Z \leq Y \leq Z \times X + 1 - X$
$or(X, Y, Z)$	satisfait	$Z = X + Y - X \times Y$
		$Z \times (1 - Y) \leq X \leq Z$
		$Z \times (1 - X) \leq Y \leq Z$
$not(X, Y)$	satisfait	$X = 1 - Y$
		$Y = 1 - X$

La définition du résolveur est alors triviale et présentée dans la table 22. Il encode simplement les relations ci-dessus.

Proposition 5.2 *Les contraintes and, or, et not sont correctes et complètes.*

Preuve :

La preuve de correction consiste simplement à montrer que chaque triplet de valeur (dans $\{0, 1\}$) satisfaisant la relation ci-dessus est un élément de la table de vérité correspondante. La complétude par rapport à la sémantique déclarative (tables de vérité) est montrée en sens inverse. On peut aussi relier directement le résolveur de `clp(B/FD)` à la théorie de propagation B et montrer que chaque fois qu'une règle de B s'applique, le triplet de

```
and(X,Y,Z):-  Z in min(X)*min(Y)..max(X)*max(Y),
              X in min(Z)..max(Z)*max(Y)+1-min(Y),
              Y in min(Z)..max(Z)*max(X)+1-min(X).

or(X,Y,Z):-   Z in min(X)+min(Y)-min(X)*min(Y)..
                   max(X)+max(Y)-max(X)*max(Y),
              X in min(Z)*(1-max(Y))..max(Z),
              Y in min(Z)*(1-max(X))..max(Z).

not(X,Y):-    X in {1-val(Y)},
              Y in {1-val(X)}.
```

Tableau 22 : définition du résolveur booléen de `clp(B/FD)`

variables dans l'ensemble de contraintes résultant satisfait les relations mathématiques encodées par le résolveur de contraintes. Là encore une analyse de cas simple suffit à prouver ce résultat. Par exemple si $and(X, Y, Z), Y = 1 \rightarrow Z = X$ s'applique, alors $Z \leq X \leq Z \times Y + 1 - Y$ est vérifié dans l'ensemble de contraintes résultant. □

5.4 Evaluation des performances de `clp(B/FD)`

5.4.1 Le jeu de benchmarks

Pour évaluer les performances de `clp(B/FD)`, nous avons utilisé des programmes booléens traditionnels :

- `schur`: le lemme de Schur.
  Ce problème consiste à trouver un coloriage en trois couleurs des entiers $\{1 \ldots n\}$ tel qu'il n'y ait pas de triplet monochrome (x, y, z) tel que $x + y = z$. Le programme utilise $3 \times n$ variables pour indiquer, pour chaque entier, sa couleur. Ce problème a une solution ssi $n \leq 13$.

- `pigeon`: le problème des pigeons.
  Il consiste à mettre n pigeons dans m cages (avec au plus un pigeon par cage). La formulation booléenne utilise $n \times m$ variables pour indiquer, pour chaque pigeon, le numéro de sa cage. Il y a évidemment une solution ssi $n \leq m$.

- `queens`: le problème des reines.
  Il faut placer n reines sur un échiquier $n \times n$ de telle manière qu'aucune reine n'en attaque une autre. La formulation booléenne utilise $n \times n$ variables pour indiquer, pour chaque case de l'échiquier, s'il y a une reine dessus ou non.

- `ramsey`: le problème de Ramsey.
  Trouver un coloriage à trois couleurs du graphe complet à n sommets tel qu'il n'y ait pas de triangle monochrome. Le programme utilise trois variables pour chaque arête pour indiquer sa couleur. Ce problème a une solution ssi $n \leq 16$.

Pour ces programmes, on calcule toutes les solutions sauf si cela est explicitement mentionné. Les résultats présentés pour `clp(B/FD)` n'incluent aucune heuristique et ont été mesurés sur une station de travail Sparc 2 (28.5 Mips). La section suivante compare `clp(B/FD)`avec la version commerciale de CHIP (CHIP 3.2). Nous avons choisi CHIP comme principale comparaison parce que, d'une part, c'est un produit commercial et, d'autre part, c'est également un langage de PLC (et pas seulement un résolveur booléen) et qu'il accepte donc les même programmes que `clp(B/FD)`. De plus le résolveur booléen de CHIP est également basé sur des techniques de propagation héritées des domaines finis[1]. Nous avons aussi comparé `clp(B/FD)` avec des résolveurs booléens dédiés, les résultats en sont présentés dans les sections suivantes.

5.4.2 `clp(B/FD)` et CHIP

Les temps pour CHIP ont aussi été mesurés sur une station Sparc 2. Exactement les mêmes programmes ont été exécutés sur les deux systèmes.

`clp(B/FD)` est environ huit fois plus rapide que CHIP, en étant parfois meilleur de deux ou trois ordres de magnitude (cf. la table 23). Ce facteur huit peut être comparé au facteur quatre entre `clp(FD)` et CHIP pour des programmes de PLC sur les domaines finis. La raison principale de cette différence est, à notre avis, que dans CHIP le résolveur booléen est écrit au dessus du résolveur sur les domaines finis alors que dans `clp(B/FD)` le résolveur booléen est spécifique, grâce à l'utilisation de la contrainte primitive X `in` r, et qu'il

[1] l'autre résolveur de CHIP, basé sur l'unification booléenne, devient vite impraticable pour des problèmes complexes : aucun des programmes de test présentés ici n'a pu être exécuté en l'utilisant, à cause de problèmes d'occupation mémoire trop importante.

Programme	CHIP 3.2	clp(B/FD) 2.21	$\frac{\text{CHIP}}{\text{clp(B/FD)}}$
schur 13	0.830	0.100	8.30
schur 14	0.880	0.100	8.80
schur 30	9.370	0.250	37.48
schur 100	200.160	1.174	170.49
pigeon 6/5	0.300	0.050	6.00
pigeon 6/6	1.800	0.360	5.00
pigeon 7/6	1.700	0.310	5.48
pigeon 7/7	13.450	2.660	5.05
pigeon 8/7	12.740	2.220	5.73
pigeon 8/8	117.800	24.240	4.85
queens 8	4.410	0.540	8.16
queens 9	16.660	2.140	7.78
queens 10	66.820	8.270	8.07
queens 14 1st	6.280	0.870	7.21
queens 16 1st	26.380	3.280	8.04
queens 18 1st	90.230	10.470	8.61
queens 20 1st	392.960	43.110	9.11
ramsey 12 1st	1.370	0.190	7.21
ramsey 13 1st	7.680	1.500	5.12
ramsey 14 1st	33.180	2.420	13.71
ramsey 15 1st	9381.430	701.106	13.38
ramsey 16 1st	31877.520	1822.220	17.49

Tableau 23 : clp(B/FD) versus CHIP (temps en sec.)

bénéficie de l'implantation optimisée de celle-ci directement.

5.4.3 clp(B/FD) et les autres résolveurs

Nous comparons ici clp(B/FD) avec des résolveurs booléens dédiés utilisant des algorithmes très différents.

Ces résolveurs ne sont pas des langages de programmation, ils prennent en entrée un ensemble de contraintes et les résolvent. Il y a donc autant de formulations que d'instances du problème. A l'opposé, clp(B/FD) génère les contraintes lors de l'exécution du programme (ce surcoût est limité à environ 20 %, donc peu important), et un seul programme suffit pour toutes les instances d'un même problème. Un autre point à souligner est que

n'avons pas pu exécuter exactement les mêmes programmes et que nous avons donc utilisé les temps donnés dans les articles de référence (qui incorporent en général de nombreuses heuristiques).

`clp(B/FD)` et les BDD

Adia est un résolveur booléen efficace basé sur les BDD. Les mesures de temps d'exécution proviennent de [59], sur une station de travail Sparc IPX (28.5 Mips). Quatre heuristiques différentes sont utilisées, et nous avons choisi de comparer `clp(B/FD)` avec le meilleur et le pire des temps donnés. Notons que les méthodes basées sur les BDD calculent toutes les solutions et sont donc inutilisables pour les gros problèmes tels que `queens` pour $n \geq 9$ et `schur` for $n = 30$.

Ici encore `clp(B/FD)` a de très bonnes performances, voir la table 24 dans lequel le signe $\downarrow$ devant un nombre signifie un facteur de ralentissement de `clp(B/FD)` par rapport à Adia. On voit que `clp(B/FD)` est en général plus rapide d'au moins un ordre de grandeur sauf pour le problème des pigeons. Il est à noter que les performances des méthodes basées sur les BDD par rapport aux méthodes énumératives ou par propagation varient énormément d'un problème à l'autre, cf. [66] pour une étude sur un autre jeu de benchmarks.

Programme	Pire BDD	Meilleur BDD	`clp(B/FD)` 2.21	Pire BDD / `clp(B/FD)`	Meil. BDD / `clp(B/FD)`
`schur 13`	3.260	1.110	0.100	32.60	11.10
`schur 14`	5.050	1.430	0.100	50.50	14.30
`pigeon 7/6`	1.210	0.110	0.310	3.90	$\downarrow$ 2.81
`pigeon 7/7`	3.030	0.250	2.660	1.13	$\downarrow$ 10.64
`pigeon 8/7`	4.550	0.310	2.220	2.04	$\downarrow$ 7.16
`pigeon 8/8`	15.500	0.580	24.240	$\downarrow$ 1.56	$\downarrow$ 41.79
`queens 6`	2.410	1.010	0.060	40.16	16.83
`queens 7`	12.030	4.550	0.170	70.76	26.76
`queens 8`	59.210	53.750	0.490	120.83	109.69

Tableau 24 : `clp(B/FD)` versus un BDD (temps en sec.)

clp(B/FD) et une méthode énumérative

[58] propose des résultats pour une méthode énumérative à la Davis-Putnam utilisée pour l'unification booléenne. Les temps sont donnés sur une station Sun 3/80 (1.5 Mips) et ont été normalisés par un facteur 1/19. clp(B/FD) est en moyenne 6.5 fois plus rapide, voir la table 25.

Programme	Enum.	clp(B/FD) 2.21	$\frac{\text{Enum}}{\text{clp(B/FD)}}$
schur 13	0.810	0.100	8.10
schur 14	0.880	0.100	8.80
pigeon 5/5	0.210	0.060	3.50
pigeon 6/5	0.120	0.050	2.40
pigeon 6/6	2.290	0.360	6.36
pigeon 7/6	0.840	0.310	2.70
queens 7	0.370	0.170	2.17
queens 8	1.440	0.540	2.66
queens 9	6.900	2.140	3.22

Tableau 25 : clp(B/FD) versus une méthode énumérative (temps en sec.)

clp(B/FD) et la consistance locale booléenne

Nous comparons ici à [51] dans lequel sont présentés les résultats d'un résolveur basé sur une méthode de consistance locale booléenne. Les temps sont donnés sur un Macintosh SE/30, équivalent à un Sun 3/50 (1.5 Mips). Nous les avons donc normalisés par un facteur 1/19. Ce résolveur comporte deux heuristiques d'énumération, la plus importante étant la possibilité d'ordonner dynamiquement les variables par rapport au nombre de contraintes dans lesquelles elles se retrouvent. clp(B/FD) utilise par contre tout simplement un ordre statique.

Un point intéressant à remarquer est que les différences de performances entre les deux méthodes sont dans des facteurs constants à l'intérieur de chaque classe de problème. clp(B/FD) est un peu plus lent sur schur d'un facteur 1.4, trois fois plus rapide sur pigeon, et quatre fois plus rapide sur queens, voir la table 26. Nous conjecturons que les deux résolveurs réalisent à peu près le même travail d'élagage de l'espace de recherche, bien qu'ils soient basés sur des structures de données pour les contraintes très différentes.

Programme	BCons.	clp(B/FD) 2.21	BCons / clp(B/FD)
schur 13	0.070	0.100	↓ 1.42
schur 14	0.080	0.100	↓ 1.25
pigeon 7/6	0.870	0.310	2.80
pigeon 7/7	7.230	2.660	2.71
pigeon 8/7	6.820	2.220	3.07
pigeon 8/8	67.550	24.240	2.78
queens 8	1.810	0.540	3.35
queens 9	7.752	2.140	3.62
queens 10	32.720	8.270	3.95
queens 14 1st	3.140	0.870	3.60
queens 16 1st	17.960	3.280	5.47

Tableau 26 : clp(B/FD) versus la consistance locale booléenne (temps en sec.)

clp(B/FD) et une méthode de recherche opérationnelle

Nous comparons ici avec la méthode FAST93 [6], qui est basée sur des techniques de programmation 0-1 de la recherche opérationnelle. Les temps sont donnés pour une station Sparc Station 1+ (18 MIPS), donc normalisés avec un facteur 1/1.6 dans la table 27. Il faut noter que dans ces tests, seule la première solution est calculée, car FAST93 ne peut calculer toutes les solutions. Pour le problème des pigeons, FAST93 a de bonnes performances quand le problème est insatisfiable (même pour des gros problèmes), c'est-à-dire quand il y plus de pigeons que de cages ($N > M$). En effet, cette méthode peut déduire rapidement que l'inéquation ne peut être satisfaite. La formulation purement booléenne que nous avons utilisée pour clp(B/FD) n'a pas d'aussi bons résultats pour les gros problèmes, mais il est très facile en clp(B/FD) d'ajouter une contrainte non-booléenne $N < M$ (qui détectera immédiatement l'inconsistance), car notre système est intégré dans un résolveur sur les domaines finis. Notons que ceci serait impossible pour un résolveur purement booléen, et ceci explique pourquoi nous n'avons pas pu utiliser cette formulation dans nos comparaisons.

5.5 clp(B) un résolveur dédié pour les booléens

Dans la section précédente nous avons vu que les techniques de propagation locales offraient une alternative satisfaisante pour résoudre efficacement des problèmes booléens et,

Programme	FAST93	clp(B/FD) 2.21	FAST93 / clp(B/FD)
pigeon 7/7 1st	0.250	0.020	12.50
pigeon 8/7 1st	1.940	2.220	↓ 1.14
pigeon 8/8 1st	0.630	0.030	21
pigeon 9/8 1st	4.230	20.190	↓ 4.77
pigeon 9/9 1st	0.690	0.040	17.25
ramsey 10 1st	11.500	0.110	104.54
ramsey 12 1st	81.440	0.190	428.42

Tableau 27 : clp(B/FD) versus une méthode de R.O. (temps en sec.)

en particulier, clp(FD) grâce à sa primitive X `in` r. Toutefois, seul un ensemble réduit des possibilités offertes par la primitive X `in` r furent nécessaires lors de la définition de clp(B/FD). Nous allons donc concevoir un résolveur spécialisé pour les booléens que nous appellerons clp(B) basé sur une simplification de clp(FD) pour le cas booléen. Ce travail est intéressant à divers titres :

- il va nous permettre d'évaluer le surcoût de clp(FD) lorsque seuls les booléens sont utilisés (i.e. lorsque seule une part réduite des possibilités de X `in` r est utilisée).
- il nous permettra de justifier encore plus précisément pourquoi les méthodes de propagation locales sont bien adaptées au problèmes booléens.
- il va nous faire découvrir une machine abstraite étonnamment simple permettant d'étendre aisément n'importe quel Prolog (i.e. WAM) pour prendre en charge des contraintes booléennes.

5.5.1 La contrainte primitive l_0 `<=` $l_1, \ldots, l_n$

D'un point de vue pratique, clp(B) est très similaire à clp(FD) puisqu'il est le résultat d'une spécialisation. Une telle spécialisation est possible grâce aux propriétés qui apparaissent lorsque l'on se restreint à des domaines 0..1. Citons par exemple :

- il n'est plus nécessaire de coder le domaine explicitement dans la représentation d'une variable booléenne (puisque l'on sait que c'est toujours 0..1).

- une telle variable ne peut être réduite qu'une seule fois (comme les variables Prolog). Ceci signifie que les estampilles ne sont plus nécessaires.
- si l'on se reporte à notre théorie de propagation booléenne (cf. table 21) l'on se rend compte que les règles ne se déclenchent que lorsqu'une des variables booléennes devient instanciée à une certaine valeur et ont pout effet d'instancier une autre variable. D'où l'idée de spécialiser les contraintes dépendant d'une variable en deux listes : la liste des contraintes à réveiller lorsque la variable vaudra 0 et celle des contraintes à réveiller lorsque la variable vaudra 1.

Ceci nous conduit à définir une primitive plus adaptée que X `in` r pour décrire les propagations booléennes. Celle-ci est de la forme l_0 `<=` $l_1, \ldots, l_n$ où chaque l_i est soit un littéral positif (X) ou un littéral négatif (`-`X) (cf. table 28). Notons la différence avec la formulation de `clp(B/FD)` où la primitive X `in` r était utilisée de manière plus "calculatoire" pour déterminer la valeur à affecter à la variable X.

c ::=	l_0 `<=` `[`$l_1, \ldots, l_n$`]`	
l ::=	X	(littéral positif)
	`-`X	(littéral négatif)

Tableau 28 : syntaxe de la contrainte l_0 `<=` $l_1, \ldots, l_n$

Définition 5.4 *A chaque littéral l_i l'on associe X_i, la variable correspondante, tel que si $l_i \equiv$ `-`X ou $l_i \equiv X$ alors $X_i = X$.*

De même l'on définit $Bvalue_i$ comme la valeur de vérité du littéral l_i, i.e. si $l_i \equiv$ `-`X (resp. $l_i \equiv X$) alors $Bvalue_i = 0$ (resp. $Bvalue_i = 1$).

La sémantique déclarative de la contrainte l_0 `<=` $l_1, \ldots, l_n$ est évidemment "l_0 doit être vrai dans tout *store* qui satisfait $l_1 \wedge \ldots \wedge l_n$" (un littéral l_i étant satisfait dans un *store* S ssi $X_{i_S} = Bvalue_i$).

Sans aucune perte de généralité nous pouvons considérer que le corps n'est composé que de 1 ou 2 littéraux (i.e. $n = 1$ ou $n = 2$). En effet, le cas $n = 0$ revient à unifier X_0 à la valeur $Bvalue_0$ et le cas $n > 2$ peut être réécrit en remplaçant l_0 `<=` `[`l_1`,`l_2`,`l_3`,...,`l_n`]` par l_0 `<=` `[`l_1`,`I_2`]`, I_2 `<=` `[`l_2`,`I_3`]`,..., I_{n-1} `<=` `[`l_{n-1}`,`l_n`]`, où chaque I_k est une nouvelle variable

booléenne. Dans `clp(B)` un pré-processeur se charge de ces réécritures de code. Cette décomposition nous permet d'implanter très efficacement l'opération *Tell* pour notre unique primitive l_0 `<=` $l_1, \ldots, l_n$ puisque seuls les cas $n = 1$ et $n = 2$ doivent être pris en compte.

5.5.2 Définition des contraintes booléennes

Tout comme nous l'avons fait précédemment pour construire `clp(B/FD)` nous allons définir la traduction de chacune des contraintes booléennes en contraintes primitives de `clp(B)`, c'est-à-dire en contraintes l_0 `<=` $l_1, \ldots, l_n$. Du fait de l'adéquation de cette primitive pour spécifier des règles de propagation nous obtenons une transcription directe (cf. table 29) de la théorie présentée en table 21[2]

```
and(X,Y,Z):-   Z <= [X,Y],  -Z <= [-X],  -Z <= [-Y],
               X <= [Z],     -X <= [Y,-Z],
               Y <= [Z],     -Y <= [X,-Z].

or(X,Y,Z):-   -Z <= [-X,-Y], Z <= [X],     Z <= [Y],
              -X <= [-Z],     X <= [-Y,Z],
              -Y <= [-Z],     Y <= [-X,Z].

not(X,Y):-     X <= [-Y],   -X <= [Y],
               Y <= [-X],   -Y <= [X].
```

Tableau 29 : définition du résolveur booléen de `clp(B)`

5.5.3 Extension de la WAM

Les principales modifications de la WAM pour prendre en compte les variables booléennes (et la contrainte primitive l_0 `<=` $l_1, \ldots, l_n$) sont très similaires (quoique plus simples) à ce que nous avons fait pour les domaines finis (cf. section 4.1 pour plus de détails).

Un nouveau type de données est ajouté : les variables booléennes qui résideront dans le heap. Ces variables seront différenciées des autres variables grâce à une nouvelle étiquette (`BLV`). L'ajout de ce nouveau type de variable affecte légèrement la WAM comme suit :

[2] les preuves de corrections et de complétudes sont d'autant plus triviales et laissées au soin du lecteur.

Manipulation de données

Tout comme les variables DF, les variable booléennes ne peuvent être dupliquées. Pour assurer cela nous utilisons le même principe que pour les variables domaines.

Unification

Une variable booléenne X peut être unifiée avec :

- une variable Prolog Y : Y est simplement liée à X.
- un entier n: si $n = 0$ ou $n = 1$ la paire (X, n) est mise en file de propagation et la *procédure de consistance* est invoquée (cf. section 5.5.4).
- une autre variable booléenne Y: ceci revient à ajouter les contraintes suivantes : X`<=[`Y`]`, `-`X`<=[-`Y`]`, Y`<=[`X`]` et `-`Y`<=[-`X`]`.

Sauvegarde et restauration des domaines

Tout comme pour les variables domaines, il nous faut pouvoir sauvegarder des valeurs dans la trail (ex. le passage d'une variable booléenne à un entier lors de l'instanciation de celle-ci). Ceci est déjà supporté par notre architecture WAM. En revanche, il n'est plus nécessaire d'utiliser des méthodes d'estampillages comme cela était le cas avec les variables domaines puisque une variable booléenne ne sera réduite qu'une seule fois (pour prendre sa valeur définitive).

Indexation

Ici encore l'on procède comme pour les variables domaines en considérant une variable booléenne comme une variable Prolog classique et donc en essayant toutes les clauses.

Nouvelles structures de données pour les booléens

`clp(B)` utilise une file de propagation pour la phase de consistance. Notre présentation se basera sur une file explicite dont le début et la fin sont pointés par les registres `BP` et

`TP` (comme dans la version initiale de `clp(FD)`). Toutefois il est possible d'utiliser une file implicite similairement à ce qui a été présenté dans la version optimisée de `clp(FD)`. Les éléments de cette file sont de couples $(X, Bvalue)$ indiquant que la valeur $Bvalue$ doit être affecté à la variable X.

L'autre structure de données nécessaire permet d'enregistrer les informations d'une variable booléenne X (cf. figure 16) et se compose de :

- le mot étiqueté (auto-référence comme pour les variables domaines, cf. section 4.1.1).
- listes de dépendances : pour raisons évidentes d'efficacité nous distinguons les contraintes à activer lorsque X sera instancié à 0 (i.e. contraintes dépendant de $-X$) de celles à activer lorsque X sera instancié à 1 (i.e. contraintes dépendant de X).

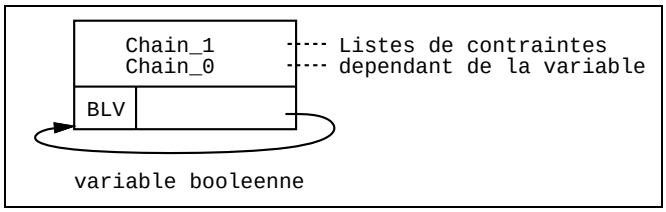


Figure 16 : représentation interne d'une variable booléenne

Intéressons nous désormais à la représentation des contraintes elles-mêmes. Du fait qu'il a au plus 2 littéraux dans le corps d'une contrainte $c \equiv l_0$ `<=` $l_1, \ldots, l_n$ il est possible d'associer c aux variables dont elle dépend. En effet :

- dan le cas $n = 1$: c ne dépend que d'une seule variable (i.e. X_1) et doit être résolue aussitôt que $X_1 = Bvalue_1$. Il est donc commode de coder c entièrement dans un enregistrement de la liste appropriée des contraintes dépendant de X_1.
- dans le cas $n = 2$: c dépend à la fois de X_1 et de X_2. Nous pouvons choisir de code c dans la liste appropriée de X_1 **ainsi** que dans la liste appropriée de X_2. Aussitôt que X_1 est instancié à $Bvalue_1$ il nous faut vérifier que X_2 (l'autre variable impliquée) est instancié à $Bvalue_2$ pour pouvoir résoudre c. Pour permettre d'effectuer cette vérification il nous faut garder un référence de X_1 vers X_2 (et réciproquement de X_2 vers X_1)

Ainsi, contrairement à ce qui s'est passé avec les domaines finis, nous n'aurons pas une structure particulière pour enregistrer les contraintes car elles seront entièrement codées

dans les listes de dépendances. Chaque élément des listes de dépendances encode une contrainte l_0 `<=` $l_1, \ldots, l_n$ et contient les informations suivantes (c.f. figure 17) :

- l'adresse de variable contrainte (i.e. X_0).
- la valeur qui faudra lui affecter (i.e. $Bvalue_0$).
- dans le cas $n = 2$: l'adresse de l'autre variable impliquée,.
- dans le cas $n = 2$: la valeur que devra vérifier l'autre variable impliquée.

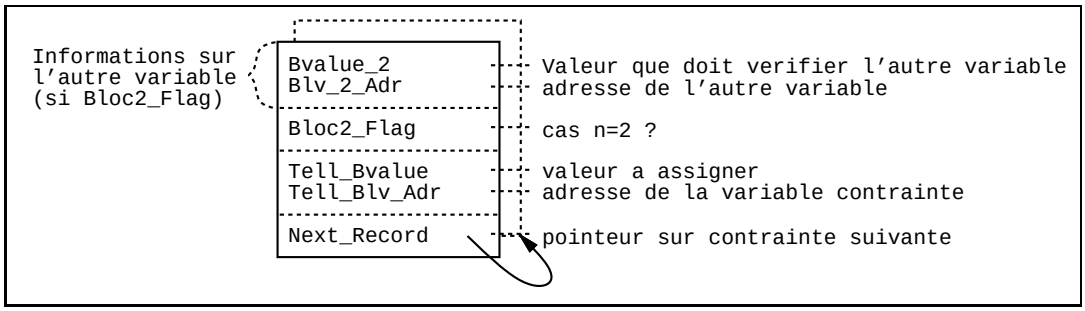


Figure 17 : représentation interne d'une contrainte dans les listes de dépendances

Notons que dans le cas $n = 2$ un enregistrement est nécessaire dans la liste de dépendance appropriée de X_1 (référençant l'autre variable X_2) ainsi que dans la liste appropriée de X_2 (référençant X_1). Cette duplication d'information est très limitée puisqu'elle se réduit à un surcoût de 2 mots. La solution alternative nécessiterait d'utiliser un mot de plus pour stocker un compteur du nombre de variables. La figure 18 montre un "instantané" de la mémoire où apparaissent les structures de données impliquées dans la contrainte `Z <= [-X,Y]` (qui pourrait être utilisée pour définir `xor(X,Y,Z)`).

Jeu d'instructions

La compilation d'une contrainte l_0 `<=` $l_1, \ldots, l_n$ se décompose en deux parties :

- chargement des variables X_0,...,X_n dans des arguments de la WAM,
- installation et exécution de la contrainte, i.e. création des enregistrements appropriés dans les listes de dépendances des variables X_1,...,X_n et détection si la corps de la contrainte est satisfait par le *store* courant. Dans l'affirmative la paire $(X_0, Bvalue_0)$ est mise en file de propagation et la *procédure de consistance* est invoquée.

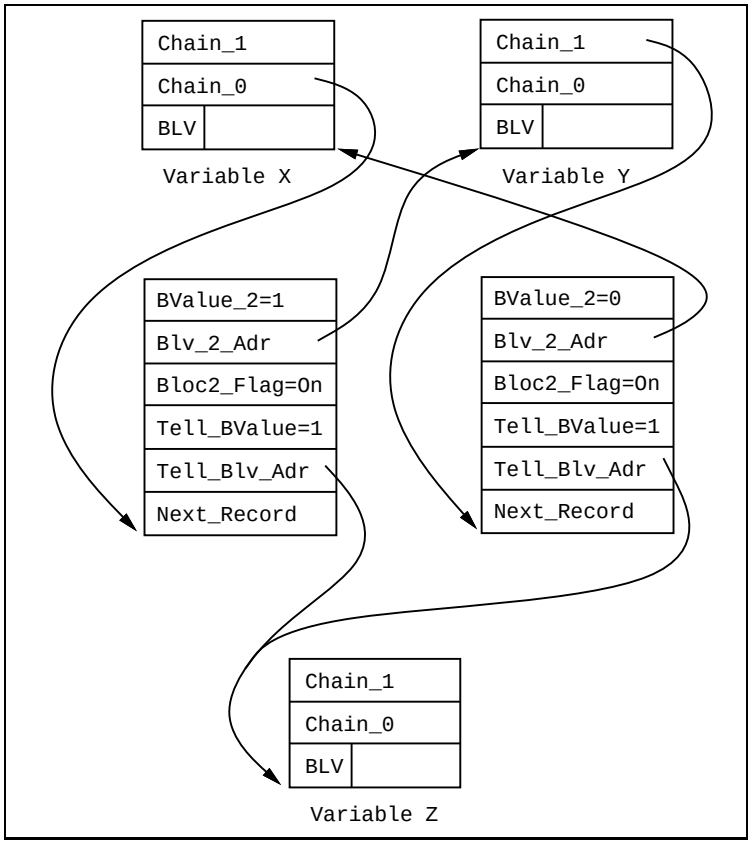


Figure 18 : structures de données nécessaires pour la contrainte `Z <= [-X,Y]`

Dans la présentation qui suit nous notons `Xi` le ième registre de la WAM plutôt que `X[i]` pour des raisons de clarté (nous simplifierons encore en écrivant `X` si l'indice du registre n'a pas d'importance). Quand à l'écriture `V`, elle dénote une variable temporaire (i.e. `X[j]` ou permanente (i.e. `Y[j]`) comme en section 3.1.7.

Les instructions de chargement sont :

`b_load_variable(V,X)`

lie `V` à une variable booléenne créée sur le tas et range son adresse dans le registre `X`.

`b_load_value(V,X)`

suivant que la valeur w de la déréférence de `V` est :

- une variable libre : similaire à `b_load_variable(`w`,X)`.
- un entier n : échoue si $n \neq 0$ et $n \neq 1$ sinon empile n sur le heap et range son adresse dans le registre `X`.

- une variable booléenne : son adresse est rangée dans le registre `X`.

Les instructions d'installation et d'exécution sont :

`b_install_and_tell_cstr1(X0,bvalue0,X1,bvalue1)` (pour le cas $n = 1$)

deux possibilités suivant `X1` :

- `X1` est un entier : si `X1` = `bvalue1` la paire (`X0`,`bvalue0`) est mise en file et la procédure de consistance est invoquée (sinon la contrainte réussit puisque le corps n'est pas satisfait).
- `X1` est une variable booléenne : un enregistrement est ajouté à la liste appropriée de `X1` référençant `X0` et `bvalue0`.

`b_install_and_tell_cstr2(X0,bvalue0,X1,bvalue1,X2,bvalue2)` (pour le cas $n = 2$)

trois possibilités suivant `X1` et `X2` :

- `X1` est un entier : similaire à `b_install_and_tell_cstr1(X0,bvalue0,X2,bvalue2)`.
- `X2` est un entier : similaire à `b_install_and_tell_cstr1(X0,bvalue0,X1,bvalue1)`.
- `X1` et `X2` sont deux variables booléennes : un enregistrement est ajouté à la liste appropriée de `X1` référençant `X0` et `bvalue0` ainsi que l'autre variable (i.e. `X2` et `bvalue2`). De même un enregistrement est ajoutée à la liste appropriée de `X2` référençant `X0` et `bvalue0` ainsi que l'autre variable (i.e. `X1` et `bvalue1`).

Remarquons que seules 4 nouvelles instructions sont nécessaires pour intégrer ce résolveur booléen dans la WAM. L'extension proposée est donc réellement minimale et notre expérience nous a montré que quelques jours suffisaient pour incorporer ce solveur booléen à un compilateur Prolog dont les sources sont bien connus (`wamcc` en l'occurrence). La table 30 présente le code généré pour la contrainte utilisateur `and(X,Y,Z)`.

5.5.4 La procédure de consistance

Cette procédure doit assurer la consistance (locale) du *store*. Elle répète les étapes suivantes jusqu'à ce que la file de propagation soit vide (ou qu'un échec soit détecté). Soit $(X, Bvalue)$ la paire couramment pointée par `BP` :

```
and/3:  b_load_value(X[0],X[0])                          X(0)=adresse de X
        b_load_value(X[1],X[1])                          X(1)=adresse de Y
        b_load_value(X[2],X[2])                          X(2)=adresse de Z
        b_install_and_tell_cstr2(X[2],1,X[0],1,X[1],1)    Z <= [X,Y]
        b_install_and_tell_cstr1(X[2],0,X[0],0)          -Z <= [-X]
        b_install_and_tell_cstr1(X[2],0,X[1],0)          -Z <= [-Y]
        b_install_and_tell_cstr1(X[0],1,X[2],1)           X <= [Z]
        b_install_and_tell_cstr2(X[0],0,X[1],1,X[2],0)   -X <= [Y,-Z]
        b_install_and_tell_cstr1(X[1],1,X[2],1)           Y <= [Z]
        b_install_and_tell_cstr2(X[1],0,X[0],1,X[2],0)   -Y <= [X,-Z]
        proceed                                          retour Prolog
```

Tableau 30 : code généré pour `and(X,Y,Z)`

SI X est un entier, celui-ci doit être égal à $Bvalue$:

- $X = Bvalue$: succès (**VérifEntier**)
- $X \neq Bvalue$: échec (**EchecEntier**)

sinon X (une variable booléenne) est instancié à l'entier $Bvalue$ (**RéducDomaine**) et toutes les contraintes dépendant de X (i.e. chaque enregistrement de `Chain_`$Bvalue$) est reconsidéré comme suit :

- cas $n = 1$: la paire $(X_0, Bvalue_0)$ est mise en file.
- cas $n = 2$: supposons que $X = X_1$ (le cas $X = X_2$ étant similaire), la variable X_2 doit être testée pour décider si la contrainte peut être résolue :
  - X_2 est un entier : si $X_2 = Bvalue_2$ alors la paire $(X_0, Bvalue_0)$ est mise en file sinon la contrainte est déjà résolue (**DéjàRésolue**).
  - X_2 est une variable booléenne : la contrainte ne peut encore être résolue (**Suspend**).

Ainsi, chaque contrainte (i.e. paire) $(X, Bvalue)$ présente en file de propagation sera activée avec pour issue un des trois possibilités suivantes :

- **RéducDomaine** : la variable booléenne X est instanciée à $Bvalue$.
- **VérifEntier** : X est déjà égal à $Bvalue$.

- **EchecEntier** : X est un entier différent de $Bvalue$ (i.e. $X = 1 - Bvalue$).

Quand une contrainte $(X, Bvalue)$ a pour issue **RéducDomaine** la phase de propagation reconsidère toutes les contraintes dépendant de la liste appropriée de X. Chaque contrainte de la liste est soit mise en file pour être activée (débouchant sur une des issues décrites ci-dessus) soit ignorée (dans le cas $n = 2$) dû à :

- **Suspend** : l'autre variable impliquée n'est pas encore instanciée. Ce n'est que lorsque celle-ci le sera (et à la bonne valeur) que la contrainte sera résolue.
- **DéjàRésolue** : l'autre variable impliquée est instanciée mais pas à la valeur désirée. Le corps de la contrainte n'est donc pas satisfait (c'est sa négation qui l'est) et donc la contrainte est trivialement satisfaite.

La figure 19 nous permet d'évaluer les proportions de chaque issues pour quelques instances des programmes de test.

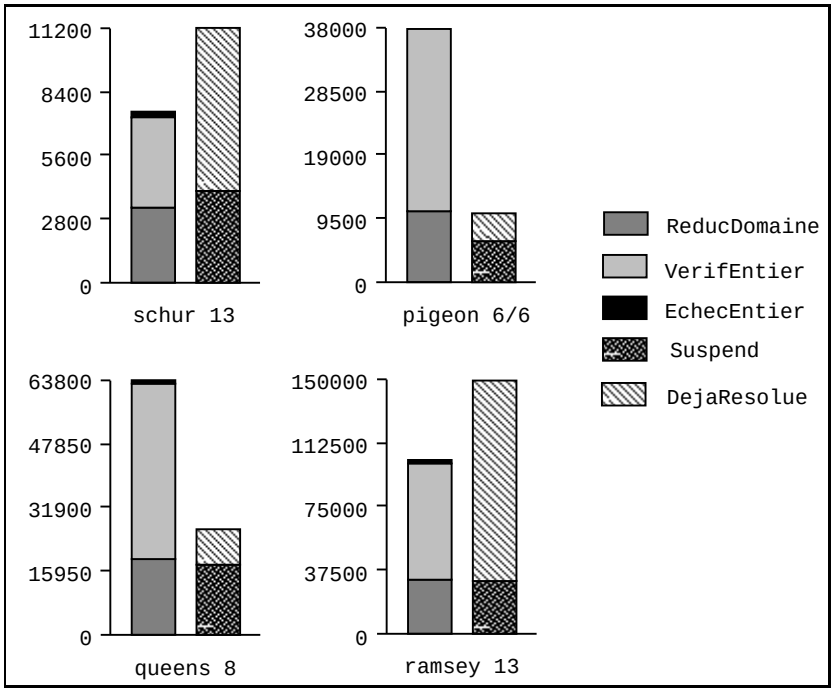


Figure 19 : proportion de chaque issue de la procédure de consistance

Tout comme pour `clp(FD)`, l'issue **VérifEntier** correspond à un $Tell$ inutile puisqu'il réussit sans modifier la variable contrainte. Ici encore des optimisations sont envisageables

pour éviter de telles réactivations inutiles. Toutefois, les résultats empiriques nous ont montrés que ce genre d'optimisations permettaient bien de réduire le nombre de réactivations inutiles mais que cela ne se traduisait pas par un gain sur le temps d'exécution. En effet, un *Tell* inutile se réduit à une comparaison entre deux entiers (i.e. $X = Bvalue$) et la détection de l'inutilité de cette opération nécessite aussi un test d'entiers.

Notons que l'issue **DéjàRésolue** correspond aussi à un travail inutile puisque la contrainte est déjà satisfaite (cf. optimisation 2 de `clp(FD)` en section 4.3.3).

5.5.5 Evaluation de `clp(B)`

La table 31 montre les performances de `clp(B)` (temps en sec.) et les facteurs d'accélérations (ou de ralentissements si précédé su symbole ↓) de `clp(B)` par rapport aux autres principaux résolveurs testés précédemment. Une étude de ce tableau nous montre que `clp(B)` est environ deux fois plus rapide que `clp(B/FD)`. Ce facteur ne varie que très légèrement (de 1.5 à 2.5) suivant les problèmes et traduit le fait que les deux systèmes effectuent les mêmes élagages de l'arbre de recherche. L'écart avec les autres résolveurs devient donc encore plus grand, cf. facteur de plus de 300 avec CHIP sur `schur 100` ou de plus de 200 avec les BDD sur `queens 8`. Dans un moindre degré `clp(B)` est près de 10 fois plus rapide que la méthode énumérative et la méthode de consistance booléenne.

5.6 Conclusion

Nous avons étudié dans ce chapitre l'impact de la propagation locale pour résoudre des problèmes booléens. Grâce à l'approche RISC de `clp(FD)` nous avons pu définir facilement un résolveur de contraintes booléennes : `clp(B/FD)`. Celui-ci est très efficace, en moyenne huit fois plus rapide que le meilleur résolveur booléen de CHIP, et plusieurs fois plus rapide que les résolveurs booléens dédiés, que ceux-ci soient basés sur les BDD, l'énumération à la Davis-Putnam, la consistance locale booléenne ou des techniques de recherche opérationnelle. Dans un second temps nous avons conçu `clp(B)`, un résolveur entièrement dédié aux booléens en étudiant les simplifications que l'on pouvait apporter à `clp(FD)` lorsque seuls les contraintes booléennes étaient utilisées. Le gain en performance est de l'ordre d'un facteur 2.

Programme	clp(B) Temps (s)	clp(B/FD) / clp(B)	CHIP / clp(B)	Meil. BDD / clp(B)	Enum / clp(B)	BCons / clp(B)
schur 13	0.040	2.50	20.57	27.75	20.25	1.75
schur 14	0.040	2.50	22.00	35.75	22.00	2.00
schur 30	0.100	2.50	93.70	*overflow*	?	?
schur 100	0.620	1.89	322.83	*overflow*	?	?
pigeon 6/5	0.020	2.50	15.00	3.00	2.00	6.50
pigeon 6/6	0.180	2.00	10.00	↓ 1.80	12.72	4.88
pigeon 7/6	0.110	2.81	15.45	1.00	7.63	7.90
pigeon 7/7	1.390	1.91	9.67	↓ 5.56	?	5.20
pigeon 8/7	0.790	2.81	16.12	↓ 2.54	?	8.63
pigeon 8/8	12.290	1.97	9.58	↓ 21.18	?	5.49
queens 7	0.090	1.88	?	50.55	4.11	?
queens 8	0.230	2.34	19.17	233.73	6.26	7.86
queens 9	0.860	2.48	19.37	*overflow*	8.02	9.01
queens 10	3.000	2.75	22.27	*overflow*	?	10.90
queens 14 1st	0.500	1.74	12.56	*overflow*	?	6.28
queens 16 1st	1.510	2.17	17.47	*overflow*	?	11.89
queens 18 1st	4.450	2.35	20.27	*overflow*	?	?
queens 20 1st	17.130	2.51	22.93	*overflow*	?	?
ramsey 12 1st	0.130	1.46	10.53	*overflow*	?	?
ramsey 13 1st	0.690	2.17	11.13	*overflow*	?	?
ramsey 14 1st	1.060	2.28	31.30	*overflow*	?	?
ramsey 15 1st	292.220	2.39	32.10	*overflow*	?	?
ramsey 16 1st	721.640	2.52	44.17	*overflow*	?	?

Tableau 31 : clp(B) versus les autres résolveurs

Un point important à développer, pour améliorer encore les performances de clp(B/FD) ou de clp(B), est d'intégrer des heuristiques plus complexes pour la phase d'énumération, adaptées au cas particulier des booléens. En effet clp(FD) n'a comme heuristique que le *first-fail* classique, qui va choisir d'énumérer d'abord les valeurs de la variable dont le domaine est le plus petit. Ceci ne sert à rien dans le cas des booléens (toutes les variables non encore instanciées ont un domaine de cardinalité 2). Il faudrait donc pouvoir déterminer par exemple le ou les variables qui apparaissent dans le plus de contraintes pour généraliser cette notion de *first-fail*. Ceci nécessite cependant de pouvoir déterminer si une contrainte est toujours vérifiée ou démentie dans l'ensemble courant de contraintes, techniques que nous allons étudier dans le chapitre 6.

Chapitre 6

Détection de la satisfaction de contraintes

6.1 Introduction

Nous allons nous intéresser ici au problème que pose la détection de la *satisfaction* d'une contrainte. Comme nous l'avons déjà vu, une telle détection permet d'optimiser l'opération $Tell$. Mais elle permettra aussi de définir l'opération Ask (cf. section 7.2) qui, intuitivement, attend qu'une certaine contrainte soit satisfaite pour lancer l'exécution d'un calcul (consistant souvent en des ajouts de nouvelles contraintes). Ce problème de détection de satisfaction appartenant à la redoutable famille NP, nous ne pouvons espérer (tout comme pour *arc-consistency*) avoir à la fois une détection *complète* et *efficace*. C'est pour cette raison que nous utiliserons une *approximation* de la condition exacte de satisfaction. Nous présenterons trois schémas de détection se différenciant par le degré de précision de l'approximation utilisée. Nous utiliserons les exemples suivants pour les comparaisons de précision.

Exemple 6.1

```
'x≠y'(X,Y):-   X in -dom(Y),    (c_X)
               Y in -dom(X).    (c_Y)
```
◇

*le contenu de ce chapitre a été publié dans [17].

Exemple 6.2

```
'x≥2y'(X,Y):- X in 2*min(Y)..infinity    (c_X)
              Y in 0..max(X)/<2.         (c_Y)
```
◇

Remarquons que les contraintes de l'exemple 6.1 sont anti-monotones alors que celles de l'exemple 6.2 sont monotones (cf. section 2.3).

Rappelons la définition de la relation de satisfaction $\vdash$ sous-jacente à notre système de contraintes :

Définition 6.1 *Un* store S **satisfait** *une contrainte* $c \equiv X$ `in` r *ssi* c *est vraie dans tout store* S' *plus contraint que* S*, i.e.*

$$S \vdash c \text{ ssi } \forall S' \quad S' \sqsubseteq S \Rightarrow X_{S'} \subseteq r_{S'}$$

L'inconvénient de cette définition est qu'elle met en oeuvre des tests *pour tout store* plus contraint que le courant. Elle ne peut donc être utilisée de manière efficace pour tester la satisfaction. De plus, du fait qu'elle vérifie (**Struct**) (i.e. $S \cup \{c\} \vdash c$) elle ne nous est d'aucune utilité pour améliorer l'optimisation 2. En effet, pour cela il nous faut détecter qu'une contrainte c *appartenant* au *store* courant ne nécessite plus d'être réévaluée car elle est satisfaite. Or, d'après (**Struct**) du fait que c appartient au *store*, c est satisfait. Donc $\vdash$ n'est pas apte à capturer les étapes intermédiaires que représentent les réductions de domaines. Définissons alors une relation de satisfaction plus forte :

Définition 6.2 *Un* store S **satisfait fortement** *une contrainte* $c \equiv X$ `in` r *(noté* $\vdash_f$*) ssi*

$$S \vdash_f c \text{ ssi } \forall S' \quad S' \sqsubseteq S \Rightarrow X_S \subseteq r_{S'}$$

Désormais nous ne considérons plus $X_{S'}$ mais X_S. Il reste toutefois le fait que cette définition implique l'évaluation de r dans tout *store* plus contraint que S. Toutefois, si nous pouvons nous assurer que $\forall S' \quad S' \sqsubseteq S \Rightarrow r_S \subseteq r_{S'}$ (i.e. r est anti-monotone) alors il suffit de vérifier $X_S \subseteq r_S$, ce qui n'implique que de tester le *store* courant. De plus, cette relation est particulièrement bien adaptée pour améliorer l'optimisation 2 puisqu'elle détecte la satisfaction d'une contrainte X `in` r à la seule vue du *store* courant, ce qui ne peut se faire qu'en concluant que X ne sera plus réduit.

La proposition suivante nous montre que $\vdash_f$ est une relation plus forte que $\vdash$.

Proposition 6.1 *Si* $S \vdash_f c$ *alors* $S \vdash c$.

Preuve :
triviale du fait que $\forall S' \;\; S' \sqsubseteq S \Rightarrow X_{S'} \subseteq X_S$. □

Nous allons donc nous intéresser à la détection de $S \vdash_f c$ plutôt qu'à la détection de $S \vdash c$. Soulignons que ceci constitue la première approximation pour la détection de la satisfaction.

Définition 6.3 *Soit une contrainte* $c \equiv X$ `in` r, $E(c)$ *est la formule syntaxique définie comme :* $E(c) \equiv$ `dom(`X`)` $\subseteq r$.

Etant donné un *store* S, cette formule s'évalue à vrai ou faux de manière évidente par la valeur de vérité de $E(c)_S \equiv X_S \subseteq r_S$. La proposition suivante est celle sur laquelle reposera la détection de satisfaction.

Proposition 6.2 *Si* c *est anti-monotone alors :* $E(c)_S \Rightarrow S \vdash_f c$.

Preuve :
puisque c est anti-monotone nous avons $\forall S' \;\; S' \sqsubseteq S \Rightarrow r_S \subseteq r_{S'}$
supposons que $E(c)_S$ soit vraie (i.e. $X_S \subseteq r_S$)
nous pouvons donc en conclure que $\forall S' \;\; S' \sqsubseteq S \Rightarrow X_S \subseteq r_S \subseteq r_{S'}$
c'est-à-dire que $\forall S' \;\; S' \sqsubseteq S \Rightarrow X_S \subseteq r_{S'}$, ce qui revient à dire que $S \vdash_f c$. □

Remarques :

- la propriété 6.2 nous indique que $E(c)$ est une *condition suffisante* pour détecter la satisfaction de toute contrainte c anti-monotone.
- cette propriété reste vraie si, au lieu de tester tout le domaine de X, on se contente de tester l'intervalle qui le contient, (i.e. $E(X$ `in` $r) \equiv$ `min(X)..max(X)` $\subseteq r$) puisque $\forall S \;\; X_S \subseteq$ (`min(`X`)..max(`X`)`$)_S$. Ceci constitue une approximation supplémentaire. Dans ce qui suit nous présenterons des approximations de r et non pas de X. Ainsi, lorsque nous en viendrons à avoir des conditions du type `min(X)..max(X)` $\subseteq r$ ce sera uniquement qu'elles sont équivalentes à `dom(X)` $\subseteq r$ (i.e. r est un intervalle). Il est toutefois possible d'ajouter cette approximation à tous les schémas que nous présenterons.
- l'aspect statique de cette condition, générée à partir de la syntaxe d'une contrainte,

permet d'envisager une compilation de ces conditions.

- $E(c)$ contient toutes les variables présentes dans c. Ainsi, cette condition doit être (re)testée à chaque fois qu'une de ces variables est modifiée. Du point de vue de l'implantation, les chaînages de dépendances des variables DF pourront être étendus pour également référencer ces conditions qui seront alors (ré)évaluées au moment propice.

- il est facile de généraliser cette condition pour l'adapter aux contraintes utilisateur exprimées comme conjonctions de contraintes X `in` r. En effet, soit une contrainte utilisateur c définie comme :

  $$c \ :- \ c_1, \ldots, c_n.$$

  alors :

  $$E(c) = \bigwedge_{i=1}^{n} E(c_i) \qquad\qquad E(\neg c) = \bigvee_{i=1}^{n} E(\neg c_i)$$

- remarquons que dans bien des cas $E(c)$ peut être simplifié si les contraintes c_i composant c sont équivalentes (c'est le cas de nos deux exemples). Aussi est-il possible de choisir comme condition pour c n'importe laquelle des conditions pour c_i ou même leur disjonction. Notons toutefois qu'une telle détection d'équivalence n'est pas triviale. Nous montrerons en section 6.4.1 comment détecter certaines équivalences des formules $E(c_i)$ permettant la même simplification de $E(c)$.

Qu'en est-il toutefois de la satisfaction d'une contrainte c qui n'est pas anti-monotone (ce qui est généralement le cas) ? L'idée consiste à dériver de cette contrainte c une approximation sous la forme d'une contrainte c' anti-monotone telle que sa satisfaction implique celle de c. La détection de sa satisfaction se fait alors grâce à la proposition 6.2. Les 3 approximations suivantes se distinguent par la manière de définir c'.

6.2 Approximation 1 : test à la clôture

Dans cette approximation, c' correspond tout simplement à c mais le test de détection n'est effectué qu'à partir du moment où c est anti-monotone. Pour cela il suffit d'attendre que c soit totalement instanciée, i.e. qu'elle dénote un ensemble constant. Dans ce qui suit

nous utiliserons également cette approximation pour les contraintes anti-monotones dans un souci d'homogénéité de la précision.

Définition 6.4 *Soit une contrainte* $c \equiv X$ `in` r *;* $E_1(c)$ *est la formule syntaxique définie comme :* $E_1(c) \equiv$ `ground(`r`)` $\wedge$ `dom(`X`)` $\subseteq r$.

Remarques :

- cette stratégie peut être vue comme un *forward checking* pour la détection.
- en ce qui concerne $E_1(c \equiv c_1 \wedge ... \wedge c_n)$, si nous pouvons détecter que tous les c_i sont équivalents, il vaut mieux prendre la disjonction des conditions $E(c_i)$ puisque chacune est soumise à ses propres restrictions de clôture. La conjonction entrainerait bien souvent une détection uniquement lorsque toutes les variables sont instanciées (y compris X dans X `in` r). Toutefois, comme nous l'avons déjà dit, il n'est pas trivial de décider de l'équivalence entre contraintes dans le cas général.
- à partir de l'exemple 6.1 nous dérivons :

  $E_1(c_X) =$ `ground(Y)` $\wedge$ `dom(X)` $\subseteq$ `-dom(Y)` $\equiv$ `ground(Y)` $\wedge$ `Y` $\notin$ `dom(X)`
  $E_1(c_Y) =$ `ground(X)` $\wedge$ `dom(Y)` $\subseteq$ `-dom(X)` $\equiv$ `ground(X)` $\wedge$ `X` $\notin$ `dom(Y)`

  puisque c_X et c_Y sont équivalents :

  $$\begin{aligned} E_1(X \neq Y) &= E_1(c_X) \vee E_1(c_Y) \\ &\equiv \texttt{(ground(Y)} \wedge \texttt{Y} \notin \texttt{dom(X))} \vee \\ &\quad \texttt{(ground(X)} \wedge \texttt{X} \notin \texttt{dom(Y))} \end{aligned}$$

  La satisfaction de $X \neq Y$ est déclenchée aussitôt que l'une des deux variables est instanciée et elle est détectée aussitôt que le domaine de l'autre variable ne contient plus la valeur de la variable instanciée.
- à partir de l'exemple 6.2 nous dérivons :

  $$\begin{aligned} E_1(c_X) &= \texttt{ground(Y)} \wedge \texttt{dom(X)} \subseteq \texttt{2*Y..infinity} \\ &\equiv \texttt{ground(Y)} \wedge \texttt{min(X)} \geq \texttt{2*Y} \\ E_1(c_Y) &= \texttt{ground(X)} \wedge \texttt{dom(Y)} \subseteq \texttt{0..X/<2} \\ &\equiv \texttt{ground(X)} \wedge \texttt{max(Y)} \leq \texttt{X/<2} \end{aligned}$$

  puisque c_X et c_Y sont équivalents :

$$E_1(X \geq 2 * Y) = E_1(c_X) \vee E_1(c_Y)$$
$$\equiv \texttt{(ground(Y)} \wedge \texttt{min(X)} \geq \texttt{2*Y)} \vee$$
$$\texttt{(ground(X)} \wedge \texttt{max(Y)} \leq \texttt{X/<2)}$$

Ici aussi la satisfaction de $X \geq 2 * Y$ est déclenchée aussitôt que l'une des deux variables est instanciée.

- l'avantage de cette approximation est le très faible coût de la détection puisqu'elle n'est déclenchée que tardivement. Evidemment cet avantage devient un inconvénient en ce qui concerne la précision de la détection qui est assez faible.
- enfin, ce schéma peut comporter diverses variantes pour la détection de X `in` r comme d'attendre que X soit également clos, d'attendre que seul un ensemble restreint de variables de r soit clos (assurant toutefois que r est anti-monotone), etc...

6.3 Approximation 2 : test sur les domaines

L'idée consiste, dans le cas d'une contrainte c qui n'est pas anti-monotone, à produire une contrainte c' plus forte que c qui elle est anti-monotone.

Définition 6.5 *Soit inf (resp. sup) la fonction de $TermSyn$ dans lui-même définie en table 32.*

Terme t	$inf(t)$	$sup(t)$
ct	ct	ct
`min(`Y`)`	`min(`Y`)`	`max(`Y`)`
`max(`Y`)`	`min(`Y`)`	`max(`Y`)`
t_1 . t_2 (. $\in$ {+,*})	$inf(t_1)$. $inf(t_2)$	$sup(t_1)$. $sup(t_2)$
t_1 . t_2 (. $\in$ {-,/<,/>})	$inf(t_1)$. $sup(t_2)$	$sup(t_1)$. $inf(t_2)$

Tableau 32 : définition de $inf(t)$ et $sup(t)$

Intuitivement, $inf(t)$ (resp. $sup(t)$) représente la plus petite (resp. grande) valeur que t puisse prendre.

Proposition 6.3 *(correction de inf et sup)*
$\forall S, t \quad Inf(t)_S \leq t_S \leq Sup(t)_S$.

Preuve :
triviale par induction sur les termes et de par les propriétés de (dé)croissance des opérations arithmétiques $+, -, *, \lfloor / \rfloor, \lceil / \rceil$. □

Définition 6.6 *Soit A (resp. M) la fonction de DomSyn dans lui-même présentée en table 33. Par abus de notation on définit $A(c)$ (resp. $M(c)$) pour toute contrainte $c \equiv X$ `in` r comme la contrainte X `in` $A(r)$ (resp. X `in` $M(r)$).*

Domaine r	$A(r)$	$M(r)$
$t_1 .. t_2$	$sup(t_1) .. inf(t_2)$	$inf(t_1) .. sup(t_2)$
`dom(`Y`)`	A`(min(`Y`)..max(`Y`))`	`dom(`Y`)`
r_1 . r_2 $(. \in \{:, \&\})$	$A(r_1)$. $A(r_2)$	$M(r_1)$. $M(r_2)$
`-`r	`-`$M(r)$	`-`$A(r)$
r . ct $(. \in \{+, -, *, /\})$	$A(r)$. ct	$M(r)$. ct

Tableau 33 : définition de $A(r)$ et $M(r)$

Proposition 6.4 *$A(r)$ est un domaine anti-monotone et $M(r)$ est un domaine monotone.*

Preuve :
triviale par induction structurelle sur les domaines et du fait de la proposition 6.3. □

Proposition 6.5 $\forall S, r \quad A(r)_S \subseteq r_S \subseteq M(r)_S$

Preuve :
triviale par induction structurelle sur les domaines. □

Corollaire 6.1 **(correction)** $\forall S, c \quad S \vdash_f A(c) \Rightarrow S \vdash_f c$

Preuve :
il suffit de montrer que $S \vdash_f X$ `in` $A(r) \Rightarrow S \vdash_f X$ `in` r.
C'est-à-dire $\forall S' \quad S' \sqsubseteq S \quad dom(X)_S \subseteq A(r)_{S'} \Rightarrow dom(X)_S \subseteq r_{S'}$. Ce qui se déduit trivialement de la proposition précédente. □

Définition 6.7 *Soit une contrainte $c \equiv X$ `in` r ; $E_2(c)$ est la formule syntaxique définie comme : $E_2(c) \equiv$* `dom(`X`)` $\subseteq$ $A(r)$.

Remarques :

- cette stratégie peut être vue comme un *full lookahead* pour la détection.

- à partir de l'exemple 6.1 nous dérivons :

  $E_2(c_X) = \texttt{dom(X)} \subseteq \texttt{-dom(Y)} \equiv \texttt{dom(X)} \cap \texttt{dom(Y)} = \emptyset$
  $E_2(c_Y) = \texttt{dom(Y)} \subseteq \texttt{-dom(X)} \equiv \texttt{dom(Y)} \cap \texttt{dom(X)} = \emptyset$

  puisque c_X et c_Y sont équivalents :

  $$E_2(X \neq Y) = E_2(c_X) \wedge E_2(c_Y) \equiv E_2(c_X)$$
  $$\equiv \texttt{dom(X)} \cap \texttt{dom(Y)} = \emptyset.$$

  La satisfaction de $X \neq Y$ est donc détectée aussitôt que les domaines de X et Y sont *disjoints*. E_2 est donc bien plus précis que E_1.

- à partir de l'exemple 6.2 nous dérivons :

  $E_2(c_X) = \texttt{dom(X)} \subseteq \texttt{2*max(Y)..infinity} \equiv \texttt{min(X)} \geq \texttt{2*max(Y)}$
  $E_2(c_Y) = \texttt{dom(Y)} \subseteq \texttt{0..min(X)/<2} \equiv \texttt{max(Y)} \leq \texttt{min(X)/<2}$

  puisque c_X et c_Y sont équivalents :

  $$E_2(X \geq 2 * Y) = E_2(c_X) \wedge E_2(c_Y) \equiv E_2(c_X)$$
  $$\equiv \texttt{min(X)} \geq \texttt{2*max(Y)}$$

  La satisfaction de $X \geq 2 * Y$ sera détectée aussitôt que tout le domaine de X sera plus grand ("à droite") que le domaine $2 * Y$. Ici aussi E_2 est donc bien plus précis que E_1.

- l'avantage de cette approximation est la précision de la détection. Son inconvénient majeur est le coût des opérations sur les domaines. Par exemple, la détection de $X \neq Y$ demande une intersection *à chaque* modification du domaine de X ou du domaine de Y. Ceci peut être bien souvent trop coûteux. Notons toutefois que dans certains cas cette condition ne peut être vraie que si X et/ou r sont clos. C'est notamment le cas si $r = \texttt{dom}(Y)$ ou si $r = t_1..t_2$ et que t_1 et t_2 sont les mêmes termes aux sous-termes indexicaux $\texttt{min}(Y)$ et $\texttt{max}(Y)$ près (i.e. $inf(t_1) = inf(t_2)$). Ces cas sont faciles à détecter et peuvent donner lieu à une compilation spécifique (se ramenant à l'approximation 1).

6.4 Approximation 3 : test sur les intervalles

Le but de cette approximation est de fournir un compromis entre les deux premières approximations. Elle est bien plus précise que l'approximation 1 (mais un peu moins efficace) tout en étant moins précise que l'approximation 2 (mais plus efficace). L'idée principale est d'éviter les opérations sur les domaines en généralisant, quand cela est possible, la simplification faite précédemment lorsque nous avons écrit :

$E_2(c_X) = \texttt{dom(X)} \subseteq \texttt{2*max(Y)..infinity} \equiv \texttt{min(X)} \geq \texttt{2*max(Y)}$

En effet, les conditions générées avec l'approximation 2 sont de la forme $\texttt{dom}(X) \subseteq A(r)$ faisant donc toujours intervenir des opérations sur les domaines. Toutefois, si $A(r)$ est un intervalle $t_1..t_2$ nous pouvons réduire cette condition à la condition équivalente :

$\texttt{min}(X) \geq t_1 \wedge \texttt{max}(X) \leq t_2$

Ce faisant nous n'avons *rien perdu en précision* tout en gagnant sur le coût des opérations.

Pour appliquer ce principe nous devrons nous limiter à un sous-ensemble de contraintes sur lequel $A(r)$ peut être exprimé sous la forme d'un intervalle (ou d'unions/intersections d'intervalles).

L'obtention d'un intervalle pour le test va se faire en deux temps. Premièrement redéfinissons M pour le cas $r = \texttt{dom}(Y)$ tel que $M(\texttt{dom}(Y)) = M(\texttt{min}(Y)..\texttt{max}(Y))$ au lieu de l'ancienne valeur $\texttt{dom}(Y)$ (cf. table 34). Notons que cela ne remet pas en cause la validité de la proposition 6.5 du fait que $\forall S \;\; X_S \subseteq (\texttt{min}(X)..\texttt{max}(X))_S$. Ceci ne remet donc pas en cause le corollaire 6.1 assurant la correction. Cette modification nous assure désormais que tous les domaines "terminaux" (i.e. les feuilles de l'arbre syntaxique) de $A(r)$ sont des intervalles. Notons également que nous avons restreint les domaines possibles pour r (i.e. X `in` r*ct et X `in` r`/`ct ne peuvent être traités dans ce schéma).

Définition 6.8 *Soit une contrainte* $c \equiv X$ `in` r *;* $\mathcal{E}(c)$ *est la formule syntaxique définie comme :* $\mathcal{E}(c) \equiv \texttt{min(X)..max(X)} \subseteq A(r)$.

Le système de réécriture suivant va alors simplifier l'expression $\mathcal{E}(c)$ pour faire disparaître le test d'inclusion de domaine et le remplacer par des tests de bornes d'intervalles. Les termes manipulés par le système de réécriture sont donc de la forme $r_1 \subseteq r_2$ où r_1 et r_2 sont des domaines. Etant donnée la liste de règles suivante, nous itérons du haut vers le

Domaine r	$A(r)$	$M(r)$
$t_1..t_2$	$sup(t_1)..inf(t_2)$	$inf(t_1)..sup(t_2)$
`dom(`Y`)`	A`(min(`Y`)..max(`Y`))`	M`(min(`Y`)..max(`Y`))`
r_1 . r_2 $(. \in \{$`:`,`&`$\})$	$A(r_1)$. $A(r_2)$	$M(r_1)$. $M(r_2)$
`-`r	`-`$M(r)$	`-`$A(r)$
r . ct $(. \in \{$`+`,`-`$\})$	$A(r)$. ct	$M(r)$. ct

Tableau 34 : nouvelle définition de $A(r)$ et $M(r)$

bas sur celle-ci, appliquant une règle si elle n'a pas encore été utilisée. Une règle s'applique si sa partie gauche filtre une sous-formule de la condition à normaliser et la remplace par la partie droite de la règle. Lorsqu'une règle s'applique, le processus est itéré à partir du début de la liste.

- **COM** : les opérations de complémentation sont explicitées :

  `-(`t_1`..`t_2`)` $\rightarrow$ `0..`t_1`-1:`t_2`+1..infinity`

- **IU** : les intersections donnent lieu à des conjonctions et les unions à des disjonctions :

  $r \subseteq r_1$ `&` $r_2 \rightarrow r \subseteq r_1 \wedge r \subseteq r_2$
  $r \subseteq r_1$ `:` $r_2 \rightarrow r \subseteq r_1 \vee r \subseteq r_2$

- **DIS** : les opérations sur l'ensemble du domaine sont distribuées sur les bornes :

  `(`t_1`..`t_2`)` . $ct \rightarrow t_1.ct..t_2.ct$ $\quad (. \in \{$`+`,`-`$\})$

- **INC** : les inclusions deviennent des tests de bornes :

  $t_1..t_2 \subseteq t_3..t_4 \rightarrow t_1 \geq t_3 \wedge t_2 \leq t_4$

Définition 6.9 *Soit une contrainte c et $\mathcal{E}(c)$ la condition définie ci-dessus. L'on définit alors $E_3(c)$ comme la simplification de $\mathcal{E}(c)$ par les règles de réécritures.*

Proposition 6.6 *(terminaison de la réécriture) La simplification de toute condition $\mathcal{E}$ termine.*

Preuve :
La *terminaison* est assurée du fait que **COM** remplace une opération de complémentation par une union, **IU** remplace une intersection (resp. une union) par une conjonction (resp. une disjonction), **DIS** remplace une opération sur l'ensemble du domaine par une opération sur les bornes et **INC** remplace une inclusion par une conjonction. □

Proposition 6.7 *(correction de la réécriture) E(c) est équivalent à sa forme simplifiée* $E_3(c)$.

Preuve :
ce qui est assuré du fait que toute règle remplace une partie gauche par une partie droite équivalente. □

Exemple 6.3

$Soit\ c \equiv$ `X in -dom(Y)`, $A(c) =$ `X in -(min(Y)..max(Y))`

$\mathcal{E}(c)$ = `min(X)..max(X)` $\subseteq$ `-(min(Y)..max(Y))`

$\rightarrow_{COM}$ `min(X)..max(X)` $\subseteq$ `0..min(Y)-1 : max(Y)+1..infinity`

$\rightarrow_{IU}$ `min(X)..max(X)` $\subseteq$ `0..min(Y)-1` $\vee$
`min(X)..max(X)` $\subseteq$ `max(Y)+1..infinity`

$\rightarrow_{INC}$ `(min(X)` $\geq$ `0` $\wedge$ `max(X)` $\leq$ `min(Y)-1)` $\vee$
`(min(X)..max(X)` $\subseteq$ `max(Y)+1..infinity)`

$\rightarrow_{INC}$ `(min(X)` $\geq$ `0` $\wedge$ `max(X)` $\leq$ `min(Y)-1)` $\vee$
`(min(X)` $\geq$ `max(Y)+1` $\wedge$ `max(X)` $\leq$ `infinity)` ◇

Remarques :

- cette stratégie peut être vue comme un *partial lookahead* pour la détection.
- à partir de l'exemple 6.1 nous dérivons :

  $\mathcal{E}(c_X) =$ `min(X)..max(X)` $\subseteq$ `-(min(Y)..max(Y))`
  $\mathcal{E}(c_Y) =$ `min(Y)..max(Y)` $\subseteq$ `-(min(X)..max(X))`

  après simplification par réécriture (cf. exemple 6.3) :

  $E_3(c_X) \equiv$ `(min(X)` $\geq$ `0` $\wedge$ `max(X)` $\leq$ `min(Y)-1)` $\vee$
  `(min(X)` $\geq$ `max(Y)+1` $\wedge$ `max(X)` $\leq$ `infinity`
  $E_3(c_Y) \equiv$ `(min(Y)` $\geq$ `0` $\wedge$ `max(Y)` $\leq$ `min(X)-1)` $\vee$
  `(min(Y)` $\geq$ `max(X)+1` $\wedge$ `max(Y)` $\leq$ `infinity)`

  qui se simplifie en :

  $E_3(c_X) \equiv$ `max(X)` $\leq$ `min(Y)-1` $\vee$ `min(X)` $\geq$ `max(Y)+1`
  $E_3(c_Y) \equiv$ `max(Y)` $\leq$ `min(X)-1` $\vee$ `min(Y)` $\geq$ `max(X)+1`

  puisque c_X et c_Y sont équivalents :

$$E_3(X \neq Y) = E_3(c_X) \wedge E_3(c_Y) \equiv E_3(c_X)$$
$$\equiv \texttt{max(X)} \leq \texttt{min(Y)+1} \vee \texttt{min(X)} \geq \texttt{max(Y)+1}$$

Ainsi, la satisfaction de $X \neq Y$ est détectée aussitôt que les domaines de X et Y *ne se chevauchent plus*. E_3 est donc moins précis que E_2 mais bien plus que E_1.

- à partir de l'exemple 6.2 nous dérivons :

  $\mathcal{E}(c_X) = \texttt{dom(X)} \subseteq \texttt{2*max(Y)..infinity}$
  $\mathcal{E}(c_Y) = \texttt{dom(Y)} \subseteq \texttt{0..min(X)/<2}$

  et après réécriture :

  $E_3(c_X) = \texttt{min(X)} \geq \texttt{2*max(Y)} \wedge \texttt{max(X)} \leq \texttt{infinity} \equiv \texttt{min(X)} \geq \texttt{2*max(Y)}$
  $E_3(c_Y) = \texttt{min(Y)} \geq \texttt{0} \wedge \texttt{max(Y)} \leq \texttt{min(X)/<2} \equiv \texttt{max(Y)} \leq \texttt{min(X)/<2}$

  puisque c_X et c_Y sont équivalents :

  $$E_3(X \geq 2 * Y) = E_3(c_X) \wedge E_3(c_Y) \equiv E_3(c_X)$$
  $$\equiv \texttt{min(X)} \geq \texttt{2*max(Y)}$$

  Nous obtenons donc la même précision que celle fournie par l'approximation 2.

- $E_3(c)$ ne contient que des opérations sur les entiers et devient de ce fait beaucoup plus efficace à tester que $E_2(c)$ dans le cas général. Ici encore les cas détectables où $E_3(c)$ ne peut être vraie que sous certaines conditions de clôtures peuvent être optimisés (cf. remarques pour l'approximation 2).

- $E_3(c)$ moins précis que $E_2(c)$ seulement si c contient des opérations de complémentation du fait de la redéfinition de $M(\texttt{-}r)$.

6.4.1 Equivalence des conditions suffisantes

Dans ce qui suit, nous définissons des règles de réécriture permettant de normaliser les conditions de satisfaction pour pouvoir détecter leur équivalence. Soit c une contrainte linéaire et $E_3(c)$ sa condition de satisfaction. La normalisation de $E_3(c)$ se fait en réécrivant cette formule en une *forme normale disjonctive* où chaque terme dans $E_3(c)$ est remplacé par sa *forme normale additive*. Etant donnée la liste de règles suivantes, nous itérons du haut vers le bas sur celle-ci, appliquant une règle si elle n'a pas encore été utilisée. Une

règle s'applique si sa partie gauche filtre une sous-formule de la condition à normaliser et la remplace par la partie droite de la règle. Lorsqu'une règle s'applique, le processus est itéré à partir du début de la liste.

- **DNF** : la *forme normale disjonctive* d'une condition est calculée par la règle suivante :

  $E \wedge (E_1 \vee E_2) \rightarrow (E \wedge E_1) \vee (E \wedge E_2)$
  $(E_1 \vee E_2) \wedge E \rightarrow (E_1 \wedge E) \vee (E_2 \wedge E)$

- **ANF** : les règles suivantes calculent la *forme normale additive* d'un terme :

  $t*(t_1 \cdot t_2) \rightarrow t*t_1 \cdot t*t_2 \quad (\cdot \in \{+,-\})$
  $(t_1 \cdot t_2)*t \rightarrow t_1*t \cdot t_2*t \quad (\cdot \in \{+,-\})$

- **STA** : les soustractions deviennent des additions :

  $t_1 - t \cdot t_2 \rightarrow t_1 \cdot t_2 + t \quad (\cdot \in \{\leq, \geq\})$
  $t_1 \cdot t_2 - t \rightarrow t_1 + t \cdot t_2 \quad (\cdot \in \{\leq, \geq\})$

- **DTM** : les divisions deviennent des multiplications :

  $t_1 / < t \geq t_2 \rightarrow t_1 \geq t_2 * t$
  $t_1 / > t \leq t_2 \rightarrow t_1 \leq t_2 * t$

  $t_1 \leq t_2 / < t \rightarrow t_1 * t \leq t_2$
  $t_1 \geq t_2 / > t \rightarrow t_1 * t \geq t_2$

Définition 6.10 *Soit une contrainte c et $E_3(c)$ sa condition de satisfaction. On définit alors $\Pi(c)$ comme la normalisation de $E_3(c)$ par les règles de réécriture.*

Soient $\Pi(c_1)$ et $\Pi(c_2)$ deux conditions normalisées associées aux contraintes c_1 et c_2. On dit que $\Pi(c_1)$ et $\Pi(c_2)$ sont égales (modulo la commutativité et l'associativité de $\wedge$ et de $\vee$) si chaque paire d'inégalités associées dans $\Pi(c_1)$ et $\Pi(c_2)$ sont égales. Deux inégalités $t_1 \leq t_2$ et $t_3 \leq t_4$ sont égales ssi t_1 et t_3 sont égaux et t_2 et t_4 sont égaux, où l'égalité entre termes est définie comme l'identité modulo la commutativité et l'associativité de $+$ et $*$.

Proposition 6.8 *(terminaison) La normalisation de toute condition $E_3(c)$ termine.*

Preuve :
la *terminaison* est assurée du fait que : **DNF** remplace une conjonction par deux conjonctions de taille inférieure, **ANF** remplace un produit par deux produits de taille inférieure,

STA remplace une soustraction par une addition et **DTM** remplace une division par une multiplication. □

Proposition 6.9 *(correction) Si* $\Pi(c_1)$ *et* $\Pi(c_2)$ *sont décidés équivalents alors* $E_3(c_1)$ *est vrai ssi* $E_3(c_2)$ *est vrai.*

Preuve :
il suffit pour cela de prouver que chaque règle conserve l'équivalence c'est-à-dire que pour toute règle $l \rightarrow r$ on a $l \Leftrightarrow r$. Ce qui est trivialement vrai pour **DNF**, **ANF** et **STA**. Seules les règles de **DTM** nécessitent une attention particulière à cause du problème posé par les divisions entières. Nous allons étudier le cas général $x \; div \; y \cdot z \Leftrightarrow x \cdot z * y$ où x, y et z sont des entiers et $y > 0$, $div \in \{\lfloor / \rfloor, \lceil / \rceil\}$ et $\cdot \in \{<, \leq, \geq, >\}$. La table suivante montre la validité des 8 formules possibles.

Formule	Preuve
$\lfloor x/y \rfloor < z \Leftrightarrow x < z * y$	voir preuve 1
$\lceil x/y \rceil > z \Leftrightarrow x > z * y$	voir preuve 1'
$\lfloor x/y \rfloor \geq z \Leftrightarrow x \geq z * y$	voir preuve 2
$\lceil x/y \rceil \leq z \Leftrightarrow x \leq z * y$	voir preuve 2'
$\lfloor x/y \rfloor > z \not\Leftrightarrow x > z * y$	$x = 7, y = 2, z = 3$
$\lceil x/y \rceil < z \not\Leftrightarrow x < z * y$	$x = 7, y = 2, z = 4$
$\lfloor x/y \rfloor \leq z \not\Leftrightarrow x \leq z * y$	$x = 7, y = 2, z = 3$
$\lceil x/y \rceil \geq z \not\Leftrightarrow x \geq z * y$	$x = 7, y = 2, z = 4$

Les preuves suivantes se basent sur le fait que $\lfloor x/y \rfloor \leq x/y < \lfloor x/y \rfloor + 1$.

preuve 1: $\lfloor x/y \rfloor < z \Leftrightarrow x < z * y$

$\Rightarrow$: $\lfloor x/y \rfloor < z \Rightarrow \lfloor x/y \rfloor + 1 \leq z$ or $x/y < \lfloor x/y \rfloor + 1$ donc $x/y < z$ i.e. $x < z * y$.

$\Leftarrow$: $x < z * y \Rightarrow x/y < z$ or $\lfloor x/y \rfloor \leq x/y$ donc $\lfloor x/y \rfloor < z$.

preuve 2: $\lfloor x/y \rfloor \geq z \Leftrightarrow x \geq z * y$

$\Rightarrow$: $\lfloor x/y \rfloor \geq z$ or $x/y \geq \lfloor x/y \rfloor$ donc $x/y \geq z$ i.e. $x \geq z * y$.

$\Leftarrow$: $x \geq z * y \Rightarrow x/y \geq z$ or $\lfloor x/y \rfloor + 1 > x/y$ donc $\lfloor x/y \rfloor + 1 > z \Rightarrow \lfloor x/y \rfloor \geq z$.

La preuve 1' (resp. 2') s'obtient à partir de la preuve 1 (resp. 2) en interchangeant : $\lfloor x/y \rfloor$ et $\lceil x/y \rceil$, $<$ et $>$, $\leq$ et $\geq$, $-$ et $+$. □

Comme nous l'avons vu il n'est pas toujours possible de remplacer une division entière par une multiplication. Toutefois, cela est toujours le cas pour les contraintes "bien écrites" (intuitivement, dans un intervalle monotone, c'est la division arrondie par excès qui doit être utilisée dans une borne inférieure et celle par défaut dans la borne supérieure). Cette manière d'écrire une contrainte permet en effet le meilleur élagage. Que se passe-t-il si cette méthodologie n'est pas respectée ? Reconsidérons la contrainte $X \geq 2 * Y$ pour la définir comme :

Exemple 6.4

```
'x≥2y'(X,Y):- X in 2*min(Y)..infinity    (c_X)
              Y in 0..max(X)/>2.         (c_Y)
```
◇

donc :
$\Pi(c_X) \equiv E_3(c_X) \equiv$ `min(X) ≥ 2*max(Y)` et
$\Pi(c_Y) \equiv E_3(c_Y) \equiv$ `max(Y) ≤ min(X)/<2`

les 2 formules sont normalisées donc pas égales. Ceci n'est pas un cas d'incomplétude de la méthode car ces deux formules ne sont pas équivalentes comme on peut le vérifier sur le *store* suivant : $S = \{$`X in 9..15, Y in 0..4`$\}$ puisque $E_3(c_X)_S$ est vraie alors que $E_3(c_Y)_S$ est fausse. Ces deux conditions n'ont pas la même précision, l'approximation de $E_3(c_Y)$ est plus grande que celle de $E_3(c_X)$.

Enfin, notons que cet algorithme est *incomplet* puisque, par exemple, il ne peut détecter l'équivalence entre les deux formules suivantes :

$$E_3(\texttt{X in 1..5 : 6..10}) = \texttt{(min(X) } \geq \texttt{ 1 } \wedge \texttt{ max(X) } \leq \texttt{ 5) } \vee \texttt{ (min(X) } \geq \texttt{ 6 } \wedge \texttt{ max(X) } \leq \texttt{ 10)}$$
$$E_3(\texttt{X in 1..10}) = \texttt{ min(X) } \geq \texttt{ 1 } \wedge \texttt{ max(X) } \leq \texttt{ 10}$$

Chapitre 7

Contraintes complexes

Dans ce chapitre nous étudierons comment peuvent être définies certaines contraintes de haut niveau. Cela nous amènera parfois à définir certaines extensions au système présenté jusqu'alors.

7.1 Contraintes arithmétiques linéaires

Définition 7.1 *Une contrainte arithmétique linéaire est une expression $E \cdot F$ où E et F sont deux expressions arithmétiques linéaires et $\cdot \in \{=, \neq, <, \leq, \geq, >\}$.*

7.1.1 Normalisation

La compilation d'une contrainte arithmétique consiste tout d'abord à *normaliser* la contrainte.

Définition 7.2 *La forme normale d'une contrainte arithmétique $E \cdot F$ est une expression de la forme $S \cdot T$ où $S = a_1 * x_1 + \ldots + a_k * x_k + c$ et $T = a_{k+1} * x_{k+1} + \ldots + a_n * x_n + d$. Chaque x_i est une variable distincte, chaque a_i est un entier > 0, c et d sont deux entiers positifs et soit c soit d vaut 0.*

*une partie du contenu de ce chapitre a été publiée dans [25].

Par exemple, la normalisation de $2*F+2*H-20 = F+3*H-G-10$ donne $F+G = H+10$. Cette normalisation permet de regrouper les variables et d'obtenir des approximations pour celles-ci (i.e. les intervalles $min..max$) plus précises que celles obtenues en traitant séparément plusieurs occurrences d'une même variable. En effet, *arc-consistency* donnant lieu à des approximations (en particulier avec un raisonnement sur les bornes), si toutes les occurrences d'une même variable X sont traitées séparément elles donnent lieu à autant d'approximations. Au niveau de la variable X, l'approximation résultante englobera toutes les approximations associées aux diverses occurrences. Par contre, si toutes les occurrences de X sont factorisées, la seule approximation résultante est beaucoup moins grossière.

Après cette étape, chaque terme normalisé (S et T) est alors trié par ordre décroissant sur les coefficients de manière à ajouter les contraintes effectuant le plus grand élagage d'abord. Dans `clp(FD)` la normalisation et le tri sont faits à la compilation (i.e. statiquement) plutôt qu'à l'exécution (i.e. dynamiquement). Ceci dans un évident souci de rapidité. Nous ne pouvons toutefois pas extraire autant d'information puisque nous n'avons aucune connaissance des liaisons dynamiques. Sans aller chercher très loin, une simple analyse de modes nous permettrait d'émettre du code plus spécialisé en évitant de considérer comme une variable DF ce qui se révélera être un simple entier.

A partir d'une forme normale, il existe deux manières de compiler les contraintes arithmétiques :

- compilation en code *inline*,
- compilation en appel de sous-contraintes de librairie.

7.1.2 Compilation en code *inline*

Dans ce schéma de compilation, une contrainte X `in` r est générée pour chaque variable x_i. Chacune définit donc une variable en fonction de $n-1$ autres. Par exemple, $F+G = H+10$ sera traduit comme :

$F = H + 10 - G$ `F in min(H)+10-max(G)..max(H)+10-min(G)` (c_F)

$G = H + 10 - F$ `G in min(H)+10-max(F)..max(H)+10-min(F)` (c_G)

$H = F + G - 10$ `H in min(F)+min(G)-10..max(F)+max(G)-10` (c_H)

L'inconvénient majeur de cette méthode est que la taille du code produit est quadratique en fonction de la taille de l'entrée [16]. Un autre inconvénient provient du fait que beaucoup de calculs redondants sont faits par toutes les contraintes. Par exemple, dans $A+B+D = F+G+H+T$ si D est modifié alors $F+G+H+T$ est évalué 2 fois (pour mettre à jours A et B) et $A+B+D$ est évalué 4 fois (pour mettre à jour F, G, H et T). Enfin le dernier inconvénient provient du fait que toute modification d'une variable entraîne une réévaluation de *toutes* les autres variables. Or, bien souvent la modification du domaine d'une variable n'a aucune répercussion sur les autres variables (du fait de l'incomplétude de *arc-consistency*). Ne pouvant détecter cela, ce schéma de compilation procède alors à $n-1$ réévaluations inutiles. Considérons par exemple un schéma de propagation par *lookahead partiel* (ne propageant que les bornes) et la décomposition ci-dessus dans le *store* :

$\{$`F in 0..15`, `G in 0..15`$\}$

donnant :

$\{$`F in 0..15`, `G in 0..15`, `H in 0..20`, c_F, c_G, $c_H\}$

Supposons maintenant que la contrainte `F in 5..15` soit ajoutée au *store*. c_G est alors réévaluée et fournit $-5..30$ qui contient déjà le domaine courant de G (qui n'est donc pas réduit). c_H est à son tour réévaluée et fournit $-10..20$ qui contient déjà le domaine de H qui n'est donc pas non plus modifié. Et ainsi de suite pour toutes les autres variables.

Tous ces défauts nous ont fait adopter l'approche suivante dans `clp(FD)`.

7.1.3 Compilation en appel de sous-contraintes de librairie

L'idée consiste à *décomposer* la contrainte linéaire en plusieurs sous-contraintes linéaires en introduisant des variables intermédiaires. Chaque sous-contrainte linéaire donne alors lieu à un appel spécifique à une contrainte définie en librairie. Par exemple, $F+G = H+10$ peut être traduit en :

$F+G=I$ `'x+y=z'(F,G,I)`

$I=H+10$ `'x=y+c'(I,H,10)`

Le code produit par cette méthode est donc très petit puisqu'il n'est constitué que d'appels de prédicats. Mais le plus grand avantage provient de l'introduction de variables intermédiaires qui factorisent des calculs et évitent donc des calculs redondants. De plus, le fait que toute variable ne dépend plus de toutes les autres variables évite également beaucoup de réveils inutiles de contraintes. Considérons à nouveau l'exemple précédent et la décomposition ci-dessus dans le *store* :

$\{$`F in 0..15`, `G in 0..15`$\}$

donnant :

$\{$`F in 0..15`, `G in 0..15`, `I in 10..30`, `H in 0..20`, $F + G = I$, $I = H + 10\}$

Supposons maintenant que la contrainte `F in 5..15` soit ajoutée au *store*. G est alors réévalué à partir de `min(I)-max(F)..max(I)-min(F)` $= 0..25$ qui contient déjà le domaine courant de G (qui n'est donc pas réduit). I est à son tour réévalué à partir de `min(F)+min(G)..max(F)+max(G)` $= 5..30$ qui contient déjà le domaine de I qui n'est donc pas non plus modifié. Le calcul s'arrête alors ici et évite de réévaluer inutilement H (et potentiellement beaucoup d'autres variables).

Ainsi, cette méthode pallie tous les désavantages du schéma de compilation en code *inline*.

Il y a évidemment différentes manières de décomposer une contrainte arithmétique influençant de manière très significative sur les performances. Intuitivement un découpage trop fin engendre un très grand nombre de variables intermédiaires (donc un surcoût important) et un découpage trop large réduit les possibilités d'optimiser les réveils inutiles que nous venons de montrer (puisqu'à l'extrême une décomposition en 1 seule contrainte équivaut à la compilation en code *inline*). En outre, moins la décomposition est fine plus la librairie nécessaire est importante (i.e. plus elle contient de sous-contraintes). Les mesures empiriques nous ont montré qu'une bonne stratégie consiste en une décomposition par groupes de 3 variables et ne nécessite pas une librairie trop importante.

7.2 Opération *Ask*

L'opération *Ask* permet de lier l'exécution d'un calcul à la réussite d'une contrainte. Cette opération à vue le jour dans le cadre des langages de programmation logique concurrents

avec contraintes (CC) [60, 61] où elle sert de mécanisme de synchronisation entre agents. Comme nous l'avons déjà dit, la contrainte X `in` r permet de spécifier *quelle information doit être propagée*. L'adjonction du *Ask* permet de spécifier *quand l'information doit être propagée*. Le *Ask* est un outil précieux pour définir des contraintes complexes puisqu'il permet de sortir du cadre restreint du *contrôle dirigé par les programmes* pour obtenir un *contrôle dirigé par les données*. Or ceci est indispensable pour définir des systèmes de contraintes du type de `FD` où les données ont autant d'importance (cf. propagation). Il est évident qu'aucun des mécanismes de retardement "ajoutés" à Prolog (ex. `freeze`, `dif`) n'est assez puissant pour capturer la richesse d'un système de contraintes et un mécanisme plus général est nécessaire. Ainsi, la définition d'un système de contraintes peut être vue comme une application des CC (la première ?).

Définition 7.3 *Soit c une contrainte et A un but, l'opération Ask entre c et A (notée $c \rightarrow A$) se comporte comme A dans un* store *S si $S \vdash c$ et réussit si $S \vdash \neg c$.*

Ainsi, l'opération $Ask\ c \rightarrow A$ doit être lue, opérationnellement, comme *si c alors A* et a le comportement suivant :

- $S \vdash c$: A est exécuté dans S.
- $S \vdash \neg c$: le *Ask* réussit simplement.
- $S \nvdash c$ et $S \nvdash \neg c$: le *Ask suspend* jusqu'à ce que le *store* contienne plus d'information pour décider de la satisfaction ou de la contradiction de c.

Remarques :

- cette opération est aussi appelée *implication bloquante* [69] du fait qu'elle est proche de l'implication intuitionniste et qu'elle est bloquante tant que le *store* ne contient pas assez d'information pour conclure.
- il est possible de compiler les *Ask*s tout comme les *Tell*s du fait que les conditions suffisantes de détection de la satisfaction sont générées de manière statique (cf. section 6). De plus, il est possible de spécifier divers degrés de précision pour ces conditions.
- *Ask* permet à l'utilisateur de spécifier des schémas de contrôle dirigés par les données donc de contrôler le processus de résolution de contraintes ce qui permet d'obtenir

des contraintes plus performantes.

- *Ask* permet de définir de manière déclarative des contraintes "câblées" dans les solveurs du type "boîtes noires".

L'opération *Ask* n'est pas (encore) implantée dans `clp(FD)` mais nous montrerons en section 7.3.2 comment simuler un cas particulier de *Ask* ($c \rightarrow A$) où A est lui-même une contrainte. On appelle ce type de *Ask* une *contrainte conditionnelle* [17]. L'implication logique $c_1 \Rightarrow c_2$ se définit comme $c_1 \rightarrow c_2$ et $\neg c_2 \rightarrow \neg c_1$ et l'équivalence entre c_1 et c_2 se définit de manière classique comme $c_1 \Rightarrow c_2$ et $c_2 \Rightarrow c_1$ qui se traduit, en fin de compte, par 4 *Ask*s.

Etudions tout de suite des utilisations de contraintes conditionnelles.

7.2.1 Le problème des séries magiques

Le problème des séries magiques [67] consiste à trouver une suite d'entiers $\{x_0, \ldots, x_{n-1}\}$ telle que chaque x_i représente le nombre d'occurrences de l'entier i dans la suite. Ainsi, pour $n = 4$ la série $\{1, 2, 1, 0\}$ est magique. La formulation originale [67] utilisait un `freeze` sur chaque X_i pour déclencher les ajouts de contraintes. Notre formulation, basée sur celle de [54], se contentera d'encoder la relation suivante :

$$x_i = \sum_{j=0}^{n-1} (x_j = i)$$

où $(x = y)$ vaut 1 si $x = y$ et 0 si $x \neq y$.

Ce qui se fait aisément en définissant une contrainte intermédiaire $(X = A) \Leftrightarrow B$ où X est une variable DF, A est un entier et B une variable booléenne (i.e. de domaine initial 0..1) valant 1 ssi $X = A$.

Exemple 7.1

```
'x=a ⇔ b'(X,A,B):- X = A  → B = 1,
                   X ≠ A  → B = 0,
                   B = 1  → X = A,
                   B = 0  → X ≠ A.
```
◇

Pour montrer la puissance de cette formulation, comparons-la à celle de CHIP utilisant un `freeze` (sur un Sparc 2, 28.5 Mips). `clp(FD)` commence par être 4 fois plus rapide que

CHIP pour $n = 10$ et finit par être plus de 400 fois plus rapide pour $n = 50$ (cf. table 35). La formulation avec `freeze` ne permet pas un aussi bon élagage de l'espace de recherche que celle avec *Ask*.

Programme	CHIP 3.2	clp(FD) 2.21	facteur accélération
magic 10 ff	0.180	0.040	4.50
magic 20 ff	1.510	0.130	11.61
magic 30 ff	11.200	0.270	41.48
magic 40 ff	66.750	0.470	142.02
magic 50 ff	334.870	0.720	465.09

Tableau 35 : problème des séries magiques

7.2.2 Contrainte `atmost`

La contrainte `atmost(N,[`X_1`,...,`X_m`],V)` est vraie ssi *au plus* N variables X_i sont égales à l'entier V. Cette contrainte peut être définie grâce à la relation :

$$\sum_{j=0}^{n} (x_j = V) \ \leq \ N$$

où $(x = y)$ vaut 1 si $x = y$ et 0 si $x \neq y$. Cette contrainte se définit simplement par le biais de la contrainte $(X = A) \Leftrightarrow B$ précédemment étudiée (cf. exemple 7.1).

7.2.3 Contrainte de cardinalité

La contrainte `cardinality(L,[`C_1`,...,`C_m`],U)` [68, 69] est vraie ssi parmi les m contraintes C_i il y en a au moins L et au plus U de vraies. Cette contrainte "câblée" dans les solveurs "boîtes noires" peut être définie en associant une variable booléenne B_i à la réussite de chaque contrainte C_i et en posant :

$$U \ \leq \ \sum_{j=0}^{m} B_i \ \leq \ N$$

ce qui peut s'encoder par :

Exemple 7.2

```
cardinality(L,Cs,U):- N in L..U,
                      card(Cs,N).
card([],0).
card([C|Cs],N):- B in 0..1,
                 C   → B=1,
                 ¬C  → B=0,
                 B=1 → C,
                 B=0 → ¬C,
                 N = M + B,
                 card(Cs,M).
```
◇

7.2.4 Contrainte `element`

La contrainte `element(I,[`E_1`,...,`E_n`],X)` est vraie ssi $X = E_I$ où I et X sont des variables DF et E_i des entiers.
Ce qui revient à encoder les relations $V = e \Leftrightarrow I \; in \; i_1 \; : \; ... \; : \; i_p$ pour toutes les occurrences $i_1, ..., i_p$ de la valeur e.

7.2.5 Contraintes arithmétiques non-linéaires

Traditionnellement les contraintes non-linéaires ne sont pas directement supportées par les résolveurs et sont retardées jusqu'à ce qu'elles deviennent linéaires. Par exemple, la résolution de $X * Y = Z$ n'aura lieu qu'à partir du moment où X ou Y est clos. Toutefois, un tel déclenchement tardif diminue les possibilités d'élagage. Dans le cas de $X * Y = Z$, le problème provient du fait que X doit être mis à jour à chaque modification de Y ou de Z par l'évaluation de Z/Y et nous devons donc prévenir le cas $Y = 0$ (similairement pour Z/X). L'on peut alors utiliser un Ask pour définir cette contrainte de manière déclarative et efficace comme suit :

Exemple 7.3

```
'xy=z'(X,Y,Z):- Y ≠ 0 → X in min(Z)/>max(Y)..max(Z)/<min(Y),
                X ≠ 0 → Y in min(Z)/>max(X)..max(Z)/<min(X),
                Z in min(X)*min(Y)..max(X)*max(Y).
```
◇

L'élagage effectué est beaucoup plus important que celui obtenu en retardant l'évaluation jusqu'à obtention de la linéarité du fait que $X \neq 0$ est une condition d'attente beaucoup

plus faible que $ground(X)$. Ainsi, la contrainte `'xy=z'(X,Y,110)` dans le *store* :

{`X in 1..40, Y in 6..30`}

réduira le domaine de X à 5..11 et celui de Y à 10..22. Du fait que la contrainte n'est pas encore linéaire, la première version n'effectuerait aucune réduction et la recherche de toutes les solutions entraînerait alors, au moment de l'énumération, l'essai de 40 valeurs pour X. En comparaison, la version utilisant le Ask ne nécessiterait que l'essai de 7 valeurs.

7.3 Généralisation de la contrainte X `in` r

7.3.1 Contraintes résolues par *full lookahead*

Grâce à la contrainte X `in` r nous pouvons spécifier des schémas de propagation du type *full-lookahead* entre deux variables et (éventuellement) une constante comme montré en exemple 2.3

Il n'est toutefois pas possible d'utiliser ce type de propagation pour des contraintes de plus grande arité. Il est néanmoins possible d'étendre la syntaxe des domaines de X `in` r pour prendre en compte les opérations arithmétiques assurant une consistance entre domaines.

Définition 7.4 *Soient d_1 et d_2 deux domaines alors*

$$d_1 \cdot d_2 = \bigcup_{k \in d_2} d_1 \cdot k \quad (\cdot \in \{+, -, *, /, \})$$

Ainsi, si le domaine de $d_1 = \{2, 5\}$ et $d_2 = \{1, 7, 11\}$ le domaine dénoté par $d_1 + d_2 = \{3, 6, 9, 12, 13, 16\}$.

Ces opérations nous permettent par exemple de définir la contrainte `div(X,Y,Q,R)` vérifiant $X = Q * Y + R$ (où Q est le quotient de X/Y et R le reste) comme suit :

Exemple 7.4

```
div(X,Y,Q,R):-Y in 1..infinity,
              R #< Y,
              X in dom(Y)*dom(Q)+dom(R),
              Q ≠ 0 → Y in (dom(X)-dom(R))/dom(Q),
```

```
Q in (dom(X)-dom(R))/dom(Y),
R in dom(X)-dom(Y)*dom(Q).
```
◇

Une telle contrainte dans le *store* :

```
{X in 2..10, Y=2, R=1}
```

réduira le domaine de X aux valeurs $\{3, 5, 7, 9\}$ et celui de Q à 1..4 comme désiré.

Notons que cette extension s'implante aisément (par extension triviale du jeu d'instructions) et peut être prise en compte par les mécanismes de détection de satisfaction de contraintes précédemment décrits.

7.3.2 Fonctions utilisateurs

La contrainte X `in` r nous permet de définir un domaine grâce à des fonctions sur les domaines (ex. intersection, union, etc...) et à des fonctions sur les termes (addition, soustraction, etc...). Notons que la définition de ces fonctions primitives ne repose que sur la (grande) expérience en matière de domaines finis de P. Van Hentenryck [69]. Il semble toutefois naturel de généraliser la syntaxe de X `in` r pour permettre à l'utilisateur de définir ses propres fonctions sur les domaines et sur les termes (cf. table 36). De telles fonctions sont appelées *fonctions utilisateurs*. Dans `clp(FD)` les fonctions utilisateurs sont écrites en C pour des raisons d'efficacité et du fait que le moteur Prolog sous-jacent (i.e. `wamcc`) supporte déjà l'ajout de code C externe.

Simulation du *Ask*

Les fonctions utilisateurs nous permettent d'encoder les contraintes conditionnelles (i.e. *Ask* du type $c_1 \rightarrow c_2$). En effet, supposons que nous décidions d'adopter une approximation de type 2 pour détecter la satisfaction de c_1 (cf. section 6). Les conditions $E_2(X$ `in` $r)$ étant du type `dom(`X`)` $\subseteq A(r)$. Il nous suffit de définir *une seule* fonction utilisateur `if_incl(`r_0`,`r_1`,`r_2`)` qui retourne r_2 si $r_0 \subseteq r_1$ et $0..infinity$ sinon.

Dans ce cas $c_1 \rightarrow c_2$ (avec $c_1 \equiv X_1$ `in` r_1 et $c_2 \equiv X_2$ `in` r_2) se définit simplement comme la contrainte $d \equiv X_2$ `in if_incl(dom(`X_1`),`$A(r_1)$`,`r_2`)`. Opérationnellement chaque fois que X_1 ou r_1 est modifié la contrainte d est réveillée, le test de satisfaction de c_1 est réévalué.

$c ::=$	X `in` r	
$r ::=$	$t_1 .. t_2$	(intervalle)
	$\{t\}$	(singleton)
	R	(paramètre domaine)
	`dom(`Y`)`	(domaine indexical)
	r_1 `:` r_2	(union)
	r_1 `&` r_2	(intersection)
	`-`r	(complémentation)
	r `+` ct	(addition point à point)
	r `-` ct	(soustraction point à point)
	r `*` ct	(multiplication point à point)
	r `/` ct	(division point à point)
	f_r`(`a_1`,...,`a_k`)`	(fonction utilisateur)
$a ::=$	$r \mid t$	(argument de fonction)
$t ::=$	`min(`Y`)`	(terme indexical min)
	`max(`Y`)`	(terme indexical max)
	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`)`	(fonction utilisateur)
$ct ::=$	C	(paramètre terme)
	n | `infinity` | ct_1`+`ct_2 | ct_1`-`ct_2 | ct_1`*`ct_2 | ct_1`/<`ct_2 | ct_1`/>`ct_2	

Tableau 36 : syntaxe étendue de la contrainte X `in` r

Tant qu'il n'est pas encore vrai, le domaine de X_2 n'est pas modifié (car la fonction retourne $0..infinity$). Dès que ce test est vrai alors X_2 est mis à jour par la contrainte c_2 (car la fonction retourne r_2).

Notons que cette manière de faire n'est pas la plus performante puisque :

- tant que le test est faux il est inutile d'évaluer r_2 (ce qui est fait lors de l'appel de la fonction `if_incl`).
- aussitôt que le test est vrai il est inutile de continuer à l'évaluer car nous savons qu'il le sera toujours dans la suite du calcul (cf. section 6).

Malgré cela les résultats obtenus sont très bons (cf. séries magiques en section 7.2.1), ce qui est de très bonne augure pour une future implantation de Ask qui évitera ces défauts.

Nouvelles fonctionnalités

Comme nous l'avons vu précédemment, les fonctions utilisateurs permettent d'encoder les contraintes conditionnelles. Ainsi, la contrainte `'xy=z'(X,Y,Z)` de l'exemple 7.3 peut être encodée par :

Exemple 7.5

```
'xy=z'(X,Y,Z):- X in div_e(min(Z),max(Y))..div_d(max(Z),min(Y)),
                Y in div_e(min(Z),max(X))..div_d(max(Z),min(X)),
                Z in min(X)*min(Y)..max(X)*max(Y).
```
◇

La fonction `div_e(`x`,`y`)` (resp. `div_d(`x`,`y`)`) retourne $\lceil x/y \rceil$ (resp. $\lfloor x/y \rfloor$) si $y \neq 0$ et 0 (resp. $infinity$) sinon.

Considérons maintenant le cas particulier où $X = Y$ (i.e. $X^2 = Z$). L'utilisation de `'xy=z'(X,X,Z)` dans le *store* :

`{X in 1..100, Z in 5..24}`

réduira le domaine de X à 1..24 mais ne modifie pas le domaine de Z. Il est toutefois possible d'améliorer l'étendue de ces réductions du fait que $X = \sqrt{Z}$. Ceci est similaire à ce qui se passe si la normalisation n'est pas effectuée pour les équations linéaires (cf. section 7.1.1) et que les diverses occurrences d'une variable sont traitées séparément. Définissons alors la contrainte $X^2 = Z$ comme suit :

Exemple 7.6

```
'xx=z'(X,Z):- X in sqrt_e(min(Z))..sqrt_d(max(Z)),
              Z in min(X)*min(X)..max(X)*max(X).
```
◇

La fonction `sqrt_e(`x`)` (resp. `sqrt_d(`x`)`) retourne la racine carrée de x arrondie à l'entier supérieur (resp. inférieur). Cette contrainte dans le *store* :

`{X in 1..100, Z in 5..24}`

réduira le domaine de X à 3..4 et celui de Z à 9..16.

Notons, pour finir qu'il est évidemment possible d'utiliser un schéma de propagation par *full-lookahead* comme indiqué précédemment. En effet, dans le cas de termes non-linéaires, l'approximation par les intervalles peut être trop grossière. Pour revenir sur notre exemple

$X^2 = Y$, avec un *store* initial :

{X in 1..100, Z in 5..24}

on obtiendrait une réduction de X à 3..4 (inchangé) et de Z à $\{9, 16\}$ qui est beaucoup plus précise que 9..16.

Optimisations

Le fait de pouvoir définir des fonctions pour le calcul de contraintes nous permet d'optimiser l'évaluation de certaines contraintes. Considérons la contrainte `diff(X,Y,I)` utilisée dans `queens` pour spécifier que $X \neq Y$, $X \neq Y + I$, $X \neq Y - I$ définie comme suit :

Exemple 7.7

```
diff(X,Y,I):- X in -{val(Y)} & -{val(Y)-I} & -{val(Y)+I},     (c_X)
              Y in -{val(X)} & -{val(X)-I} & -{val(X)+I}.     (c_Y)
```
◇

La table 37 montre le code produit par la compilation de la contrainte c_X (celui de la contrainte c_Y étant similaire).

cstr_1:	fd_dly_val(1,1,end)	T(1)=Y (si Y est clos)
	fd_compl_of_singleton(0,1)	R(0)=-{Y}
	fd_term_copy(3,1)	T(3)=Y
	fd_term_parameter(2,2)	T(2)=I
	fd_term_sub_term(3,2)	T(3)=Y-I
	fd_compl_of_singleton(3,3)	R(3)=-{Y-I}
	fd_inter(0,3)	R(0)=-{Y} & -{Y-I}
	fd_term_add_term(1,2)	T(1)=Y+I
	fd_compl_of_singleton(1,1)	R(1)=-{Y+I}
	fd_inter(0,1)	R(0)=-{Y} & -{Y-I} & -{Y+I}
	fd_tell_range(0)	X in -{Y} & -{Y-I} & -{Y+I}
end:	fd_proceed	retour d'exécution

Tableau 37 : code de `X in -{val(Y)} & -{val(Y)-I} & -{val(Y)+I}`

Ce code effectue 3 complémentations et 2 intersections. Or le domaine que nous voulons affecter à X n'est rien d'autre que $0..infinity \setminus \{Y, Y - I, Y + I\}$. Il est alors possible de redéfinir la contrainte `diff(X,Y,I)` comme :

Exemple 7.8

```
diff(X,Y,I):- X in f_diff(val(Y),I),     (c_X)
              Y in f_diff(val(X),I).     (c_Y)
```
◇

La fonction C `f_diff(y,i)` rend le domaine $0..infinity$ duquel elle a retiré successivement les valeurs y, $y-i$, $y+i$. Les 2 intersections sont donc évitées. A titre de comparaison, la table 38 présente les temps obtenus avec cette définition optimisée. Celle-ci est environ 1.5 fois plus rapide que celle non optimisée et 7 fois plus rapide que CHIP 3.2.

Programme	CHIP 3.2	clp(FD) 2.21	clp(FD) 2.21+fct
queens 16	2.830	0.890	0.570
queens 64 ff	0.990	0.130	0.100
queens 70 ff	42.150	11.070	7.830
queens 81 ff	1.620	0.210	0.170

Tableau 38 : `queens` optimisé avec fonctions utilisateurs

7.4 Disjonction constructive

La prise en compte de disjonctions de contraintes est un des point clés de la PLC ou des CSP du fait que les contraintes disjonctives apparaissent dans beaucoup de problèmes réels tels qu'ordonnancement disjonctif, *job-shop*, problèmes de "sac à dos" ou placements d'objets dans un plan. La manière la plus simple (et traditionnelle) pour gérer une disjonction de contraintes en PLC consiste à utiliser le non-déterminisme supporté par le moteur logique (i.e. Prolog) sous-jacent [67]. Malheureusement, le fait de traduire une disjonction de contraintes par un point de choix conduit vite à de piètres performances du fait du schéma naïf de backtracking utilisé par Prolog. Le backtracking intelligent peut résoudre quelques cas d'inefficacité [20] mais devient inadapté si le réseau de contraintes est fortement connecté du fait que tout point de choix est alors considéré comme pertinent (ce qui nous ramène au schéma naïf).

L'approche la plus prometteuse consiste à éviter la création de points de choix et, lorsque nécessaire, de les créer de manière dynamique plutôt que statique. Ainsi, pour les disjonctions aussi, l'approche "dirigé par les données" est préférable à l'approche "dirigé par les programmes". Une telle approche est parfaitement illustrée dans le *principe d'Andorra* [76]

qui se situe au coeur de langages tels que Andorra-I [32] ou AKL [41] et favorise les calculs déterministes et retardant les buts non-déterministes aussi longtemps que possible (i.e. tant qu'un calcul déterministe peut être effectué). Ce concept trouve ses racines dans les tout premiers développements de Prolog comme par exemple dans les procédures de recherche du type *sidetracking* [55] favorisant l'exploration des buts possédant le moins d'alternatives. Remarquons qu'il ne s'agit là que d'une variante du célèbre principe *first-fail*.

Ces idées ont été approfondies plus encore dans le cadre PLC où, en plus de vouloir éviter de faire des choix trop tôt (donc potentiellement erronés), l'on a cherché à obtenir un *comportement actif* des contraintes disjonctives. C'est ainsi qu'à vu le jour l'opérateur de *disjonction constructive* du langage `cc(FD)` [69]. Le concept de base de cette notion étant de factoriser les contraintes satisfaites par toutes les branches alternatives et de les ajouter au *store* aussitôt que possible sans créer de point de choix. Ceci peut être formalisé simplement, pour des systèmes de contraintes définis par des treillis [63], en considérant un opérateur *glb* (*greatest lower bound*) entre contraintes (en plus du classique *lub* associé à la conjonction) défini comme : $glb(c_1, c_2) = \{c \,/\, c_1 \vdash c \wedge c_2 \vdash c\}$. Ainsi, une contrainte disjonctive est utilisée de manière active pour élaguer l'arbre de recherche et sans créer de points de choix. La puissance d'une telle approche dans des applications réelles a été démontrée dans [48]. Toutefois, ce mécanisme peut être assez coûteux dans le cas des DF du fait que les contraintes disjonctives doivent être reconsidérées à chaque étape de propagation pour réévaluer l'information commune issue des disjonctions (i.e. contraintes satisfaites à ajouter au *store*). La solution consiste ici aussi à n'utiliser qu'une approximation et à ne détecter qu'un sous-ensemble de l'information commune.

Dans cette section, nous montrons que notre système de contraintes permet d'encoder un cas particulier de disjonction constructive pour lequel le même élagage est effectué mais de manière plus simple et plus efficace. Nous montrerons également que la plupart des utilisations de la disjonction constructive font partie de ce cas particulier.

7.4.1 Un exemple simple

Nous allons considérer le fameux puzzle des cinq maisons de Lewis Carroll longtemps utilisé comme benchmark dans les communautés Prolog et PLC. L'énoncé du problème

met en jeu cinq personnes vivant dans cinq maisons avec différentes professions, nationalités, animaux favoris et boissons favorites. Le problème consiste à trouver les affectations personne-maison-profession-nationalité-animal-boisson vérifiant les quatorze faits qui décrivent le problème.

La formulation de ce problème en PLC (cf. `five` [67]) utilise cinq variables pour chaque personne pour encoder sa maison, sa profession, sa nationalité, son animal et sa boisson favoris. Les faits donnent lieu à des contraintes d'égalité ou d'inégalité sur ces variables. Trois de ces faits expriment une disjonction entre contraintes. Par exemple le fait "la maison du norvégien est à côté de la maison bleue" signifie que la maison du norvégien peut être à gauche *ou* à droite de la bleue. Cela mène à une contrainte de la forme :

`N5 = C4 + 1` $\vee$ `N5 = C4 - 1`

Ce qui conduit à la définition du prédicat `plus_or_minus` :

Exemple 7.9

```
plus_or_minus(X,Y,C):- X = Y-C.
plus_or_minus(X,Y,C):- X = Y+C.
```
◇

Un tel prédicat créera un point de choix pour chaque invocation. Toutefois, nous pouvons définir un prédicat ayant la même sémantique déclarative mais déterministe et plus efficace grâce à l'opération d'*union* entre domaines fournie par le système de contraintes :

Exemple 7.10

```
plus_or_minus(X,Y,C):- X in dom(Y)-C : dom(Y)+C,
                       Y in dom(X)+C : dom(X)-C.
```
◇

Pour vérifier le comportement (actif) de cette définition supposons l'ajout de la contrainte `plus_or_minus(X,Y,1)` dans le *store* :

{`X in 1..3, Y in 1..5`}

Le prédicat défini en `clp(FD)` supprimera la valeur impossible 5 du domaine de Y (et ne créera jamais de point de choix) alors que la première définition ne peut le faire. Grâce à cette définition, `five` est deux fois plus rapide que la version de base (utilisée dans la comparaison avec CHIP).

7.4.2 "L'union fait la force"

L'idée consiste ici à définir une formule F à partir de $E = c_1 \vee c_2 \vee ... \vee c_n$ telle qu'elle ne contienne plus de disjonction. Deux cas sont alors intéressants :

(a) $E \Leftrightarrow F$: il suffit d'ajouter F au *store* et aucun point de choix n'est nécessaire.

(b) $E \Rightarrow F$: l'ajout de F au *store* ne suffit pas à assurer la correction qui sera alors garantie par un point de choix.

Etudions un cas concret de type (a) que l'on rencontre souvent. Considérons la disjonction $E = c_1 \vee c_2 \vee ... \vee c_n$, où les contraintes c_i sont de la forme X_1 `in` $r_1^i \wedge \ \ldots \ \wedge X_k$ `in` r_k^i telles que toutes les contraintes X_j `in` r_j^i, pour un i donné, soient équivalentes et que toutes les contraintes c_i portent sur le même ensemble de variables $\{X_1, ..., X_k\}$. Intuitivement, cela correspond à une disjonction de contraintes utilisateurs ayant toutes les variables en commun et où chaque contrainte utilisateur s'exprime sous forme d'une conjonction de contraintes X `in` r toutes équivalentes (ex. dans le cas précédent pour `plus_or_minus`). Définissons alors F par rapport à E comme suit :

$$E \equiv \quad c_1 \vee \ldots \vee c_n \quad \equiv \bigvee \begin{array}{l} X_1 \texttt{ in } r_1^1 \wedge \ \ldots \ \wedge X_k \texttt{ in } r_k^1 \\ \ldots \\ X_1 \texttt{ in } r_1^n \wedge \ \ldots \ \wedge X_k \texttt{ in } r_k^n \end{array}$$

$$F \equiv \bigwedge \begin{array}{l} X_1 \texttt{ in } r_1^1 \vee \ \ldots \ \vee X_1 \texttt{ in } r_1^n \\ \ldots \\ X_k \texttt{ in } r_k^1 \vee \ \ldots \ \vee X_k \texttt{ in } r_k^n \end{array} \equiv \bigvee \begin{array}{l} X_1 \texttt{ in } r_1^1 \ : \ \ldots \ : \ r_1^n \\ \ldots \\ X_k \texttt{ in } r_k^1 \ : \ \ldots \ : \ r_k^n \end{array}$$

E et F sont deux formulations équivalentes du fait que dans E toutes les contraintes d'une même conjonction sont équivalentes. Nous obtenons une formulation déterministe puisque l'apect disjonctif est ramené au niveau du système de contraintes grâce à l'opération d'union. Ainsi, pour une variable X, l'on calcule le domaine associé à chaque branche de la disjonction et c'est l'union de ces domaines auquel X est contraint. Ceci a donc pour effet de retirer de X les valeurs incompatibles quelle que soit l'alternative. Remarquons que le mécanisme de propagation assure qu'une telle union est réévaluée aussitôt qu'un des composants est modifié procurant ainsi un comportement identique à celui de la disjonction constructive. Toutefois, ce traitement est beaucoup plus simple donc plus efficace puisqu'il

ne nécessite pas de faire le *Tell* de toutes les alternatives independamment pour ensuite en extraire l'information commune.

Bien que la plupart des utilisations courantes de la disjonction constructive fassent partie de ce cas, il est intéressant d'étudier ce qu'il est possible de faire si nous ne pouvons extraire qu'une approximation, i.e. une formule F telle que $E \Rightarrow F$. Dans ce cas, le seul ajout de F au *store* ne suffit pas à assurer la correction et le recours aux points de choix est indispensable. Toutefois, la contrainte F permet déjà un certain élagage et la création des points de choix peut être retardée (par exemple jusqu'au moment de l'énumération). Considérons par exemple la disjonction $E =$ `(X=4` $\wedge$ `Y=3)` $\vee$ `(X=8` $\wedge$ `Y=6)` nous pouvons en déduire $F =$ `(X=4` $\vee$ `X=8)` $\wedge$ `(Y=3` $\vee$ `Y=6)` tel que $E \Rightarrow F$. L'ajout de F au *store* réduira le domaine de X à $\{4, 8\}$ et celui de Y à $\{3, 6\}$. L'élagage ainsi obtenu est tout de même important et permet de retarder la création d'un point du choix assurant la correction.

7.4.3 Autres exemples

Etudions à présent quelques contraintes classiques pour lesquelles la disjonction constructive a déjà été proposée et montrons qu'elles peuvent toutes bénéficier de la transformation équivalente déterministe précédement vue.

Maximum de deux valeurs

Dans [69] nous trouvons la définition de la contrainte `max(X,Y,Z)` assurant que Z est la valeur maximum de X et Z. Cette contrainte peut être exprimée comme :

Exemple 7.11

```
'max(x,y)=z'(X,Y,Z):- Z in min(X)..infinity,
                      Z in min(Y)..infinity,
                      Z in dom(X) : dom(Y).
```
◇

Les deux premières contraintes assurent que Z n'est jamais plus petit que X ou que Y et la derniere contrainte assure que Z vaut l'une ou l'autre de ces valeur. Une telle contrainte

dans le *store*

{X in 5..10, Y in 7..11, Z in 1..12}

reduira le domaine de Z à 7..11.

Ordonnancement disjonctif

Dans les problèmes d'ordonnancement avec ressources partagées, il est habituel d'imposer que deux tâches utilisant une même ressource ne puissent pas se dérouler ensemble, i.e. l'une doit s'exécuter strictement avant l'autre. Soit une tâche 1 dont la date de début est $T1$ et de durée $D1$ et une tâche 2 de date $T2$ et de durée $D2$. La contrainte de non chevauchement se traduira par :

$$T1 + D1 \leq T2 \vee T2 + D2 \leq T1.$$

Ce qui se traduit dans le système de contraintes `FD` par :

```
T1 in 0..max(T2)-D1 ∧ T2 in min(T1)+D1..infinity ∨
T2 in 0..max(T1)-D2 ∧ T1 in min(T2)+D2..infinity.
```

Ceci peut alors être défini en `clp(FD)` comme :

Exemple 7.12

```
no_overlap(T1,D1,T2,D2):-
    T1 in 0..max(T2)-D1 : min(T2)+D2..infinity,
    T2 in 0..max(T1)-D2 : min(T1)+D1..infinity.
```
◇

Considérons l'ajout de la contrainte `no_overlap(T1,4,T2,8)` dans le *store* :

{T1 in 1..10, T2 in 1..10}

entraînant alors la réduction du domaine de $T1$ à $1..6 \cup 9..10$ et de celui de $T2$ à $1..2 \cup 5..10$.

Valeur absolue

Dans beaucoup de problèmes il est nécessaire de pouvoir raisonner sur des distances donc en termes de valeur absolue (ex. placements, allocation de fréquences,...). A cet effet, CHIP

propose une contrainte "cablée" `distance` et [70] définit une contrainte $|X-Y| \geq C$. Celle-ci peut être définie en `clp(FD)` à partir de la définition de $X \geq Y$ (cf. exemple 2.4) comme suit :

Exemple 7.13

```
'|x-y|>=c'(X,Y,C):- X in min(Y)+C..infinity : 0..max(Y)-C,
                    Y in min(X)+C..infinity : 0..max(X)-C.
```
◇

Considérons le *store* :

{`X in 1..10`, `Y in 1..10`}

l'ajout de la contrainte `'|x-y|>=c'(X,Y,8)` réduit le domaine de X et de Y à $\{1, 2, 9, 10\}$ similairement à ce qui est présenté dans [70].

7.5 Contraintes définies par des relations

Nous allons nous intéresser ici à la définition de contraintes vérifiant des relations définies en extensions par un ensemble de tuples. Considérons par exemple la relation définissant la multiplication qualitative entre x et y notée $x \otimes y$. L'on ne s'intéresse qu'au signe de x et de y qui peut être positif (+), négatif (-) ou indéfini (?). Choisissons de coder + par 0, - par 1 et ? par 2. La table de multiplication qualitative se présente donc comme suit :

x	y	$x \otimes y$
+	+	+
+	-	-
+	?	?
-	+	-
-	-	+
-	?	?
?	+	?
?	-	?
?	?	?

table A

x	y	$x \otimes y$
0	0	0
0	1	1
0	2	2
1	0	1
1	1	0
1	2	2
2	0	2
2	1	2
2	2	2

table B

Définissons alors la contrainte `mul_qualit(X,Y,Z)` telle que $X \otimes Y = Z$. C'est à dire que $< X, Y, Z >$ doit vérifier un des tuples de la table B. Pour cela, introduisons une nouvelle variable T indiquant le numéro du tuple solution. Au départ T à donc pour domaine 1..9.

Dès lors il nous suffit de relier toute colonne i de la table B à T par une contrainte du type `element(T,`i`,`V_i`)` où V_i est la variable associée à la colonne i (i.e. X, Y ou Z). Ceci nous conduit à la définition suivante :

Exemple 7.14

```
mul_qualit(X,Y,Z):- element(T,[0,0,0,1,1,1,2,2,2],X),
                    element(T,[0,1,2,0,1,2,0,1,2],Y),
                    element(T,[0,1,2,1,0,2,2,2,2],Z).
```
◇

Considérons le *store* :

{`X in 0..2`, `Y in 0..2`, `Z in 0..1`}

l'ajout de la contrainte `mul_qualit(X,Y,Z)` réduira le domaine de X et de Y à 0..1 car si le signe de Z est défini, ceux de X et Y le sont aussi.

Il serait évidemment possible d'écrire de manière plus optimisée cette contrainte (en tenant compte des propriétés de la multiplication qualitative) notamment grâce à des Asks. Toutefois, cette manière de procéder permet d'encoder déclarativement n'importe quelle relation (quelque soit son arité) sans se soucier de ses propriétés. Dans `clp(FD)`, une contrainte prédéfinie `relation(Tuples,Vars)` est fournie pour permettre à l'utilisateur de contraindre un tuples de variables ($Vars$) à prendre comme solution un des tuples fournie sous forme de liste dans $Tuples$.

Chapitre 8

Régulation du trafic aérien avec clp(FD)

Nous présentons ici les premiers résultats des travaux menés en collaboration avec le Centre d'Etudes de la Navigation Aérienne (CENA) situé à Orly. Ce centre à la charge de réguler le trafic aérien traversant tout le territoire français. Les résultats préliminaires nous ont encouragé à écrire un article [18]. Nous intégrons celui-ci tel quel (en anglais) car il correspond à un travail principalement mené par les personnels du CENA. Notre rôle a consité à aider ces personnes dans la formalisation du problème et à définir les contraintes spécifiques nécessaires à l'application.

*le contenu de ce chapitre a été publiée dans [18].

Using clp(FD) to Support Air Traffic Flow Management

Denise Chemla[1,3], Daniel Diaz[2], Philippe Kerlirzin[1], Serge Manchon[1]

[1] CENA, Orly Sud 205, 94542 Orly Aérogare Cedex, France
[2] INRIA, Domaine de Voluceau, 78153 Le Chesnay, France
[3] SYSECA, 315, bureaux de la Colline, 92213 Saint-Cloud, France

Abstract. In this paper, a Constraint Logic Programming (CLP) approach is used to solve an Air Traffic Flow Management (ATFM) problem, the aircraft departure slot allocation. Moreover, our purpose is to show that CLP, combining the declarativity of logic programming with the efficiency of constraint solving, is well suited to model many combinatorial optimization problems involved in the ATFM domain. clp(FD), a Constraint Logic Programming language with Finite Domain constraints has been chosen to implement our practical application.

8.1 Introduction

The density of traffic over Europe has been steadily increasing for several years. This growth is difficult to manage and causes delays for passengers and work overloads for controllers. ATFM aims at adapting a variable demand (the airplanes which want to fly) to the variable available capacity of the system of control so as to use this capacity at best. It has significant safety and economic consequences as well.

Our research is pursued in the French Air Navigation Research Center (CENA), that is involved in the development of the future Air Traffic Control system. This work will be integrated into the SPORT decision support system for traffic flow management. This system helps flow managers in analyzing traffic data and in preparing flow management measures. It is operational in the six French Air Control Centers and at the Eurocontrol Central Flow Management Unit located in Brussels. Figure 20 is a view of the SPORT system showing the French sectors and the most congested routes.

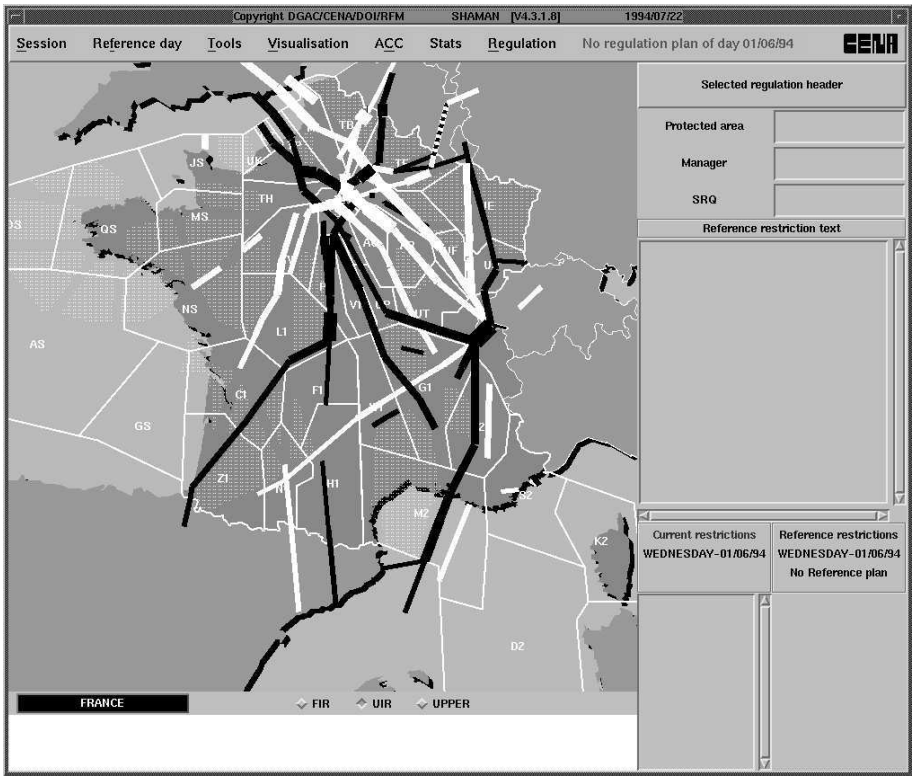


Figure 20 : Display of SPORT system

In this paper, a CLP approach is used to solve the ATFM problem of departure slot allocation. `clp(FD)`, a CLP language with Finite Domain constraints has been chosen to implement this practical application. The departure slot allocation is done manually until now, so we couldn't compare our approach with linear methods that could have been yet used. Such a comparison (linear versus CLP methods to solve ATFM problems will be done in our next research).

The structure of this paper is as follows: Sect. 2 gives a brief description of ATFM; the third one presents the `clp(FD)` language features and a new constraint developed for our needs; Sect. 4 shows how `clp(FD)` can be used to solve the departure slot allocation problem under capacity and/or flow rate constraints.

8.2 Problem Context

8.2.1 Air Traffic Flow Management Overview

ATFM aims at adapting a variable demand to the variable capacity of the system of control. Its first objective is to assure, by smoothing the flow of aircraft, that unacceptable levels of traffic congestion do not develop. Its second goal is to perform this task without imposing unnecessary flow restrictions.

France is overflown by all European air-carriers and even more. Its airspace is a patchwork of about 90 sectors. Each of them is under the responsibility of a pair of controllers. A flight crosses several sectors along its route. The radar controller works on a radar position and gives instructions to pilots via a radio link. He (she) maintains separation between planes and keep them away from specific dangers such as military areas, storms. The planning controller takes it on to find convenient entry and exit flight levels and the right coordination with neighbouring sectors. When traffic allows it, sectors can be grouped (there are about 120 possible groups of sectors).

The European ATFM activity is structured in three levels:

1. **strategic level**: at this level, long term measures are defined such as the traffic orientation scheme that dictates the routes operators have to use to go from specific departure areas to specific arrival areas. National measures are also defined at this level: modulation of controllers working hours, agreements between military and civil air traffic services, or use of main platforms scheduling;

2. **pre-tactical level**: an important feature of the sector is its capacity, i.e. the maximum number of flights that can enter the sector per hour. This capacity is variable along the day and along the year. Generally, it is greater than the demand when one team of controllers manages one sector. However, some sectors are regularly overloaded due to a limited number of controllers, to structural reasons, or to peak traffic: in that case, the demand can be greater than the capacity during certain periods of the day. The pre-tactical ATFM consists in preparing, two days before the tactical day,

a regulation plan which is a set of flow rate restrictions intended to avoid overloads within critical sectors.

3. **tactical level**: is sub-divided into two processes:

   - **slot allocation**: airline operators affected by the regulation plan have to ask for departure slots two hours before scheduled take-off, so that each aircraft enters critical sectors at the right time. In the French flow management unit, slots are allocated according to a *first-demander-first-served* principle.
   - **real time supervision**: during the pre-tactical phase, relying on traffic periodicity, flow managers forecast the traffic to come using recorded data. Because of last time changes (weather conditions, technical failures, ...), it is necessary to monitor the effects of the regulation plan and to adapt some restrictions in real time to cope with excess demand and under-used capacities.

8.2.2 The Slot Allocation Problem

First of all, we will focus on solving the slot allocation problem under capacity constraints; a small example is presented in Sect. 4. We will then extend the model in order to integrate another type of constraints, called "flow rate constraints", to organize the delays undergone by the flights in the first application.

Capacity Constraint Definition.

A capacity constraint is a relation between an airspace volume A (a sector or a group of sectors), a temporal period T and an hourly rate $N/\delta t$ (N is the maximum number of aircraft that can enter the sector each δt minutes). The constraint is satisfied if during T, at most N flights per contiguous slices of δt minutes width enter A[1]. N is called **capacity** of A. The problem consists in avoiding overloads all along the tactical day by delaying certain flights. In our model, we have made the choice that capacity constraints affect all flights without any discrimination: no flight is privileged with regard to CLP slot allocation.

[1] We have developed a new constraint, the `atmost_interval` constraint that enables the implementation of a certain number of aircraft per contiguous slices of δt-minute width; the cumulative constraint (of CHIP) allows reasoning on sliding windows of δt-minute width and is so too stringent for our needs.

Description Of The Slot Allocation Problem Model.

The slot allocation problem under capacity constraint can be defined by its input and output data. The input data are:

- the demand: constituted of a set of filled flight plans:
  $\{F_i : (O_i, D_i, SR_i : (S_{i,1}, EET_{i,1}, \ldots, S_{i,n}, EET_{i,n}))\}$,
  where F_i is a flight identifier, O_i and D_i are its origin and destination, $S_{i,1}, \ldots, S_{i,n}$ are the sectors crossed by the flight, $EET_{i,1}, \ldots, EET_{i,n}$ are the expected (by the flight carrier) entry times in those sectors ($EET_{i,1}$ is the expected *departure* time of the flight). There are 6000 flight plans a day on average.

- the resources: defined by a set of airspace volume capacity constraints:
  $\{CC_j : (\{S_{j,1}, \ldots, S_{j,m}\}, Capa_j, H1_j, H2_j)\}$
  where CC_j is a constraint identifier, $S_{j,1}, \ldots, S_{j,m}$ are the constrained airspace volumes by the capacity constraint, $Capa_j$ is the capacity (half-hourly maximum number of flights entering in the constrained airspace volume), $H1_j$ and $H2_j$ are the bounds of the application period of the constraint. An example of such a capacity constraint is
  $$CC_1 : ('UT','TU',\ 26,\ 600,\ 660)$$
  that expresses that at most 26 aircraft can enter the group of sectors $\{'UT','TU'\}$ from 10am to 11am (in minutes from 0am).

The output data is a set of satisfactory departure times $\{SET_{k,1}\}$ of the flights F_k such that all capacity constraints are satisfied and the average delay undergone by a flight is minimized.

8.3 clp(FD) in a Nutshell

As introduced in Logic Programming by the CHIP language, clp(FD) [24] is a constraint logic language based on finite domains, where constraint solving is done by propagation and consistency techniques originating from Constraint Satisfaction Problems [49, 53, 71].

```
c ::=  X in r   (constraint)

r ::=  t..t     (interval range)
       {t}      (singleton range)
       ...

t ::=  C        (parameter)
       n        (integer)
       min(X)   (indexical min)
       max(X)   (indexical max)
       val(X)   (delayed value)
       t + t    (addition)
       t - t    (subtraction)
       t * t    (multiplication)
       ...
```

Tableau 39 : fragment of the constraint system syntax

The novelty of `clp(FD)` is the use of a unique primitive constraint which allows users to define their own high-level constraints. The black-box approach gives way to glass-box approach.

8.3.1 The Constraint X `in` r

The main idea is to use a single primitive constraint X `in` r, where X is a *finite domain (FD) variable* and r denotes a *range*, which can be not only a *constant range*, e.g. 1..10 but also an *indexical range* using:

- $min(Y)$ which represents the minimal value of Y (in the current store),
- $max(Y)$ which represents the maximal value of Y,
- $val(Y)$ which represents the value of Y as soon as Y is ground.

A fragment of the syntax of this (simple) constraint system is given in table 39.

The intuitive meaning of such a constraint is: "X must belong to r in any store".

The initial domain of an FD variable is $0..\infty$ and is gradually reduced by X `in` r constraints which replace the current domain of X (D_X) by $D'_X = D_X \cap r$ at each modification of r. An inconsistency is detected when D'_X is empty. Obviously, such a detection is correct if the range denoted by r can only decrease. So, there are some monotone restrictions about the constraints [69]. To deal with the special case of anti-monotone constraints we use the general *forward checking* propagation mechanism [40] which consists in awaking a constraint only when its arguments are *ground* (i.e. with singleton domains). In `clp(FD)` this is achieved using a new indexical term $val(X)$ which delays the activation of a constraint in which it occurs until X is ground.

As shown in the previous table, it is possible to define a constraint w.r.t. the *min* or the *max* of some other variables, i.e. reasoning about the bounds of the intervals (*partial lookahead* [67]). `clp(FD)` also allows operations about the whole domain in order to also propagate the "holes" (*full lookahead* [67]). Obviously, these possibilities are useless when we deal with boolean variables since the domains are restricted to 0..1.

8.3.2 High-Level Constraints and Propagation Mechanism

From X `in` r constraints, it is possible to define high-level constraints (called *user constraints*) as Prolog predicates. Each constraint specifies how the *constrained variable* must be updated when the domains of other variables change. In the following examples X, Y are FD variables and C is a *parameter* (runtime constant value).

```
'x+y=c'(X,Y,C):-  X in C-max(Y)..C-min(Y),   (C1)
                  Y in C-max(X)..C-min(X).   (C2)

'x-y=c'(X,Y,C):-  X in min(Y)+C..max(Y)+C,   (C3)
                  Y in min(X)-C..max(X)-C.   (C4)
```

The constraint `x+y=c` is a classical FD constraint reasoning about intervals. The domain of X is defined w.r.t. the bounds of the domain of Y.

In order to show how the propagation mechanism works, let us trace the resolution of the system $\{X + Y = 4, X - Y = 2\}$ (translated via `'x+y=c'(X,Y,4)` and `'x-y=c'(X,Y,2)`):

after executing `'x+y=c'(X,Y,4)`, the domain of X and Y are reduced to 0..4 (C_1 is in the current store: X *in* $-\infty..4$, $C_2 : Y$ *in* $-\infty..4$). And, after executing `'x-y=c'(X,Y,2)`, the domain of X is reduced to 2..4 ($C_3 : X$ *in* 2..6), which then reduces the domain of Y to 0..2 ($C_4 : Y$ *in* 0..2).

Note that the unique solution $\{X = 3, Y = 1\}$ has not yet been found. So, in order to efficiently achieve consistency, the traditional method (arc-consistency) only checks that, for any constraint C involving X and Y, for each value in the domain of X there exists a value in the domain of Y satisfying C and vice-versa. So, once arc-consistency has been achieved and the domains have been reduced, an enumeration (called labeling) has to be done on the domains of the variables to yield the exact solutions. Namely, X is assigned to one value in D_X, its consequences are propagated to other variables, and so on. If an inconsistency arises, other values for X are tried by backtracking. Note that the order used to enumerate the variables and to generate the values for a variable can improve the efficiency in a very significant manner (see heuristics in [67]).

In our example, when the value 2 is tried for X, C_2 and C_4 are woken (because they depend on X). C_2 sets Y to 2 and C_4 detects the inconsistency when it tries to set Y to 0. The backtracking reconsiders X and tries value 3 and, as previously, C_2 and C_4 are reexecuted to set (and check) Y to 1. The solution $\{X = 3, Y = 1\}$ is then obtained.

8.3.3 Optimizations

The uniform treatment of a single primitive for all complex user constraints leads to a better understanding of the overall constraint solving process and allows for (a few) global optimizations, as opposed to the many local and particular optimizations hidden inside the black-box. When a constraint X `in` r has been reexecuted, if $D'_X = D_X$ it was useless to reexecute it (i.e. it has neither failed nor reduced the domain of X). Hence, we have designed three simple but powerful optimizations for the X `in` r constraint [24, 25] which encompass many previous particular optimizations for FD constraints:

- some constraints are *equivalent* so only the execution of one of them is needed. In the previous example, when C_2 is called in the store $\{X$ *in* $0..4, Y$ *in* $0..\infty\}$ Y is set to 0..4. Since the domain of Y has been updated, all constraints depending on Y are

reexecuted and C_1 (X *in* 0..4) is woken unnecessarily (C_1 and C_2 are equivalent).

- it is useless to reexecute a constraint as soon as it is entailed. In `clp(FD)`, only one approximation is used to detect the entailment of a constraint X `in` r which is "*X is ground*". So, it is useless to reexecute a constraint X `in` r as soon as X is ground.
- when a constraint is woken more than once from several distinct variables, only one reexecution is necessary. This optimization is obvious since the order of constraints, during the execution, is irrelevant for correctness.

These optimizations make it possible to avoid on average 50% of the total number of constraint executions on a traditional set of FD benchmarks (see [24, 25] for full details) and up to 57% on the set of boolean benchmarks presented below.

8.3.4 Performances

Full implementation results about the performances of `clp(FD)` can be found in [24, 25], and show that this "glass-box" approach is sound and can be competitive in terms of efficiency with the more traditional "black-box" approach of languages such as CHIP. On a traditional set of benchmark programs, mostly taken from [67], the `clp(FD)` engine is on average about four times faster than the CHIP system, with peak speedup reaching eight.

8.3.5 `atmos_interval` Constraint

To model capacity constraints, we needed to define a new constraint, the `atmost_interval` constraint.

The symbolic constraint `atmost_interval`$(N, [X_1, ..., X_m], L, U)$ is a user-defined constraint that holds if and only if at most N variables X_i are included within the interval $[L, U]$. This constraint can be defined via the relation:

$$Cardinal\{X_i / L \leq X_i \leq U\} \leq N$$

A boolean B_i is associated with each variable X_i and set to 1 if $L \leq X_i \leq U$ and to 0 otherwise. The sum of all B_i must be less than or equal to N. It is worth noticing that

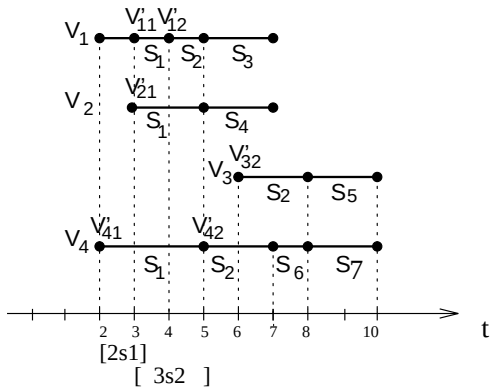


Figure 21 : Graphical representation of a small problem

such a constraint should be "wired" in CHIP by the designers of the system whereas it is defined in `clp(FD)` as a user constraint.

8.4 Slot Allocation Satisfying Capacity Constraints

8.4.1 A Small Example

In the graphical representation (Fig. 21), 4 flights are represented as connected segments. Each segment corresponds to the crossing of a sector by a flight and is characterized by its length proportional to the crossing duration. The capacity constraints are represented on time axis: only 2 aircraft are allowed to enter S1 between times 2 and 3, and only 3 aircraft are allowed to enter S2 between times 3 and 5. Variables V_i represent the expected departure times of flights; variables V'_j are S1 and S2 the expected entry times (S1 and S2 are the only constrained sectors so only those variables are necessary).

8.4.2 `clp(FD)` Model

The slot allocation problem under capacity constraints can be modelled using 3 types of constraints:

1. *Domain constraints on departure time variables*: we saw that a CLP variable corresponding to the departure time is associated with each flight; in order to satisfy

capacity constraints, the departure of a flight can be delayed, up to 3 hours (= 180 min, during our experiments).

Each departure time variable will have to satisfy the following constraint:

$$V_i \ in \ EET_{i,1}..EET_{i,1} + max_delay$$

where V_i is the departure time variable of flight F_i and $EET_{i,1}$ is the constant corresponding to the requested time of the flight (see 2.2 and 4.2).

2. *relations between sector entry time variables and departure time variables*: for each capacity constraint, a variable is created for a flight if the entering time in the first sector S_i of its route SR_j that belongs to A (where A is the constraint group of sectors - possibly a singleton) is within T (the constraint period) - see 4.1 and 4.2. Therefore, we set a new constraint on each of these variables V', as follows:

$$V'\# = V + \Delta$$

where V is the departure time and Δ is the time the flight needs to reach the sector S_i (translation constant).

3. `atmost_interval` *constraints*: finally, each capacity constraint is modelled using an `atmost_interval` constraint, defined in 3.5. Its arguments are the capacity, the list of the variables identified in step 2, and the bounds of the constraint period interval (see 3.5 and 4.1).

This model is interesting because of its simplicity and transparency: since a flight can cross many sectors, it can be affected by several `atmost_interval` constraints. Regulators speak about "combining" restrictions but it is difficult for them to evaluate the effects of such restrictions. Such an overlapping problem is modelled in a transparent way. Another interest of our model is its extensibility: for instance, it would be obvious to affect a distinct delay to flights if we considered that some special flights could not be delayed.

8.4.3 `clp(FD)` Implementation of our Small Example

The `clp(FD)` implementation of the small problem presented in 4.3 is provided in table 40. The solution found by `clp(FD)` to this problem is $S1 = \{V1 = 2, V2 = 3, V3 = 6, V4 = 4\}$.

```
Solution([V1, V2, V3, V4]) :-
       V1 in 2..12,
       V2 in 3..13,
       V3 in 6..16,
       V4 in 2..12,
       V'11 #= V1+1,
       V'21 #= V2,
       V'41 #= V4,
       V'12 #= V1+2,
       V'22 #= V2+2,
       V'32 #= V3,
       V'42 #= V4+3,
       atmost_interval(2, [V'11, V'21, V'41], 2, 3),
       atmost_interval(3, [V'12, V'22, V'32, V'42], 3, 5),
       labeling([V1, V2, V3, V4]).
```

Tableau 40 : implementation of our small problem with clp(FD) constraints

Flights 1, 2 and 3 can take-off at their requested time, while flight 4 undergoes a 2 unit-of-time delay.

8.4.4 Optimization Trials - Heuristics

To solve real cases, we needed to implement some heuristics that we describe in the three points here below:

1. labeling strategy: clp(FD) labeling works on a list of variables L and backtracks first on the last element of L, then on the last but one and so on. This has a shortcoming: a solution of average delay d_1 can be labeled before a solution of average delay d_2 with $d_2 \leq d_1$. In our small example, the solution $S2 = \{V1 = 2, V2 = 4, V3 = 6, V4 = 2\}$ is not found whereas it is better in term of average delay than $S1$. For that reason, we have implemented a new labeling strategy that enumerates solutions in the order of increasing average delays. The solution $S2$ is encountered by such a labeling before $S1$. But we could not use this labeling in practical examples because it is too slow to find a solution. To solve practical problems, we have used an heuristic that leads clp(FD) labeling to find a good solution first. It consists first in ordering take-off variables in L: the lower bound of the domain of an element i of L is always less than

or equal to the lower bound of the domain of its successor in L. The second part of the heuristic consists in setting constraints according to an increasing order among the beginning of their application period. Thanks to this heuristic, clp(FD) finds a solution that minimizes the average delay;

2. time granularity: the variable domains have bounds from 0 to 1440 (number of minutes of a day); if we allow flights to be delayed up to 3 hours, domains can contain 180 values. Those size considerations can be redhibitory in practical examples (see the size of such examples in next section). So, to reduce memory size model, we have chosen to divide all variables and domain bounds by a "time granularity" that can be 5 or 10 minutes (or else);

3. discrete approach: because of the number of variables and constraints involved, it is difficult to treat a day taken as a whole. So, we have cut it in slices of 4 hours: when a flight is delayed by the constraints of a slice, its maximum delay is reduced accordingly.

8.4.5 Results

Figures 22 and 23 show traffic histograms of UM sector before and after the clp(FD) process: Fig. 22 depicts an overload between 10a.m. and 11a.m. while Fig. 23 has absorbed it.

Table 41 provides some runtime characteristics: the total number of variables is equal to the sum of the number of "indomain" constraints and of the number of "equality" constraints.

clp(FD) was processed on a pattern containing about 100 days of a year. This proves a certain stability with regard to the density of the trafic. When no solution is found, we decrease the period length and/or increase the maximum delay that can be undergone by a flight.

To conclude this subsection, we can underline the fact that dealing only with capacity constraints to make slot allocation has some weaknesses: delays are distributed among all flights without any discrimination. So, if a regulation plan were created in such a way, it

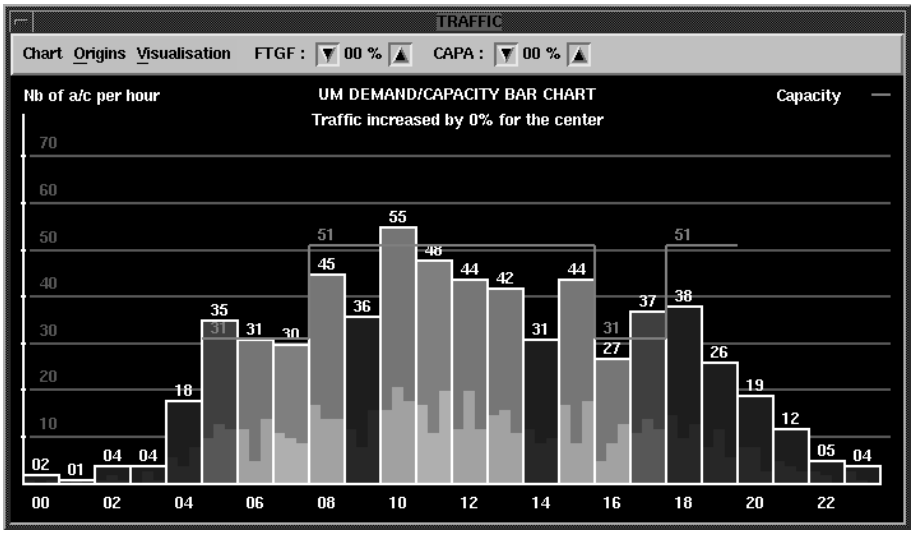


Figure 22 : UM traffic before CLP process

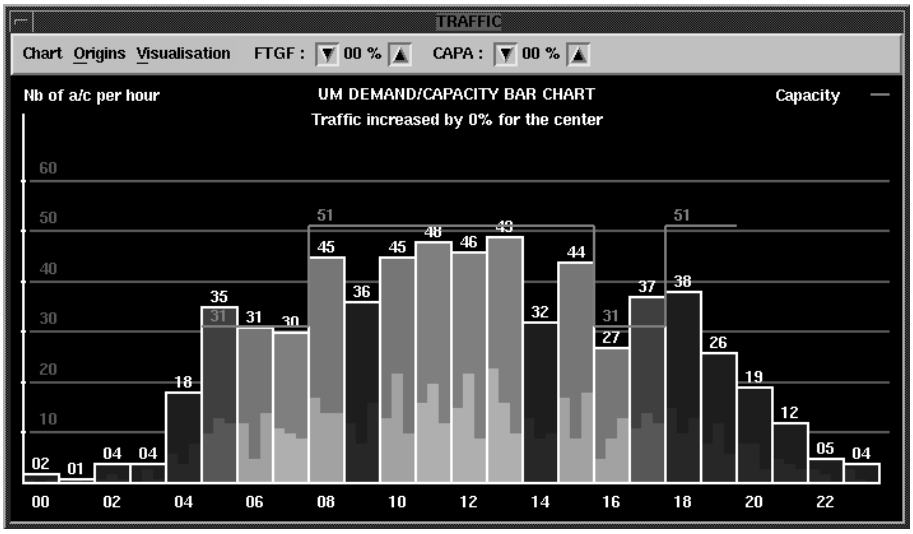


Figure 23 : UM traffic after CLP process

period length (h)	runtime	average delay (among delayed flights)	average delay (among all flights)	max delay	delayed flights (p.c.)	number of indom. constr.	numer of atmost constr.	number of equality constr.	number of flights
3	$0'39''$	$19.16'$	$5.38'$	$50'$	28.10	957	193	13031	4714
4	$2'03''$	$18.83'$	$4.45'$	$55'$	23.64	1184	256	21700	4714
5	$4'08''$	$17.50'$	$1.90'$	$50'$	10.88	1397	320	29006	4714
5	$0'44''$	$25.78'$	$10.20'$	$75'$	39.58	815	218	12069	4714
6	$1'42''$	$5.73'$	$2.51'$	$16'$	43.80	1071	282	19269	4714
3	*error*					814	199	9967	4714
10	*error*					1761	393	40187	4714

Tableau 41 : some runtime examples

would have been impossible to adapt the restrictions to last minute changes during the real time supervision phase. Hereafter, we describe an extension of the model seen so far which permits to organize delays in a fairly manner. It corresponds to the way French flow managers work.

8.4.6 Extension of the Model to Integrate Flow Rate Constraints

Flow Constraint Definition.

A flow constraint (called in Europe a "regulation") is a relation between a traffic flow F, a temporal period T and a rate $N/\delta t$ (N is the maximum number of aircraft that can feed the flow each δt minutes). F can be defined by a set of origins and/or a set of destinations and/or a set of beacons and/or a set of sectors and/or a flight level layer. Those properties of a flow are the characteristics that a flight must fulfill to be submitted to the flow constraint.

Generic example of flow constraint:

from *SetOfOrigins* **entering** *GroupOfSectors*
h1 - *h2* : $N/\delta t$

Instantiated example of flow constraint:

from *UK* **to** *Balearic*
10am **-** *11am* **:** 1/8

The constraint is satisfied if during T, there are at most N flights belonging to F per slice of δt-minute width.

`clp(FD)` Model.

We can detail the 3 types of constraints shown in 4.4 that are necessary to model the extension to flow rate constraints:

1. *domain constraints on departure time variables*: this step is identical to the first step defined in 4.4;
2. *relations between sector entry time variables and departure time variables*: capacity constraints are modelled in the same way than in 4.4; for each flow constraint, a variable V' that corresponds to the adequate instant is created for a flight if the flight belongs to the constrained flow and "reaches" the constraint (at instant V') within T. "Reaches" means that we will be interested in different instants of the flight according to the constraint type. This instant can be a sector entry time, a beacon over-flying time, a departure or arrival time. Except this, step 2 is identical to the second step above-mentioned in 4.4;
3. `atmost_interval` *constraints*: this step is identical to the third step above-mentioned in 4.4;

As we can see, the integration of flow rate constraints is very natural. This illustrates the declarativity and extensibility of our model.

8.4.7 A Simulation Aid Tool for Regulators - Cost estimation of Regulation Plans

Whatever the point of view may be, either local or global, regulators cannot have a precise idea of the effects of restrictions on traffic flows they impose because of the very large volume of data, the great interdependency between sectors, and the complexity of the air route network.

The interest of a simulation aid tool is to let the prominent rôle and the final choice to the regulators when they have at their disposal the cost estimation of a regulation plan provided by `clp(FD)`. Cost estimation can integrate criteria such as the average or maximum delay, the number of delayed flights, the number and duration of planned restrictions, the average number of restrictions affecting a flight, the difference between demand and capacity (it allows to save a security margin for imponderables). Such a tool can help them to avoid imposing unnecessary restrictions on flows. Our work has been integrated into the simulation aid tool SPORT (from which we have provided hardcopies in this paper). This integration has been easy to do because `clp(FD)` allows to obtain a C runtime program.

8.5 Conclusion and Further Works

This paper has shown how CLP is well adapted to solve ATFM problems such as departure slot allocation satisfying different types of constraints. Describing a possible extension of this practical application (flow rate regulation), we have highlighted expressiveness and flexibility of the CLP approach. It seems that numerous ATFM applications can benefit from CLP advantages. Among them, we will now investigate other applications like rerouting, automatic search of flow rate regulations, evaluation of a capacity change cost, evaluation of flow constraints cost. The efficiency of `clp(FD)` language gives us a good hope to realize interesting further works.

Chapitre 9

Conclusion

Nous voici donc arrivés au terme de cette thèse dont le fil conducteur fut la conception et la réalisation du système `clp(FD)` : un langage logique de programmation avec contraintes sur les domaines finis. En ce qui concerne la partie purement "Prolog" nous avons choisi une approche originale traduisant Prolog vers C via la WAM. Ceci nous a permis de disposer d'un Prolog simple, portable, modulaire et efficace. En effet, grâce à l'introduction de quelques lignes d'assembleur pour adapter le contrôle de C à celui de Prolog, nous disposons d'un système compact (une grande partie de la compilation étant laissée au compilateur C) et dont les performances sont comparables au meilleurs compilateurs commerciaux (ex. Quintus). Ce compilateur a ensuite été étendu pour gérer des contraintes sur les domaines finis. Là encore nous avons adopté une approche originale basée sur les travaux de P. van Hentenryck. L'idée était de rompre avec l'approche traditionnelle (résolveur vu comme une "boîte noire") pour éviter ses inconvénients (extensions impossibles, ignorance de l'implantation, complexité,...). Nous avons donc choisi une approche "boîte de verre" basée sur la définition de contraintes primitives, simples et en nombre limité permettant de rebâtir les contraintes complexes (arithmétiques et symboliques) usuelles. En ce qui concerne les domaines finis, il est remarquable qu'une seule contrainte primitive soit suffisante. Celle-ci peut être considérée comme un langage de base pour exprimer les mécanismes de propagation et la méthode de résolution choisie pour traiter une contrainte complexe. Ceci nous a conduit à un processus en deux étapes : traduction des contraintes complexes en primitives (à la compilation) et implantation d'un résolveur pour prendre en charge la primitive (à l'exécution). Le résolveur ainsi obtenu est simple, uniforme et

homogène. Le système de contraintes est, de plus, ouvert à l'utilisateur qui peut définir de nouvelles contraintes en spécifiant leur traduction en termes de primitives. Le traitement d'une seule primitive nous a permis de définir des optimisations pour éviter les réveils de contraintes inutiles. Ces optimisations profitent évidemment à toutes les contraintes de haut niveau puisque bâties sur cette primitive. Ceci est à comparer aux optimisations ad-hoc des solveurs "boîte noires". L'étude des performances de `clp(FD)` montre que ces choix étaient justifiés et que l'approche "boîte de verre" conduit à un résolveur plus efficace que ceux bâtis sur une schéma de "boîte noire". En effet, le système `clp(FD)` se révèle être en moyenne 4 fois plus rapide que CHIP dont l'efficacité n'est plus à démontrer. Un des apports essentiels de cette thèse est la définition d'une machine abstraite pour la compilation des contraintes sur les domaines finis. Ceci est important pour deux raisons :

- c'est la première fois qu'un schéma de compilation pour les contraintes sur les domaines finis est présenté. De plus ce schéma est à la fois simple et efficace et sa granularité est suffisamment fine pour permettre une compréhension aisée et une traduction instantanée dans tout langage (y compris l'assembleur)[1].

- cette machine abstraite permet réellement de comprendre "comment" implanter un compilateur pour les contraintes sur les domaines finis. L'on peut comparer cela (toutes proportions gardées !) à ce qui s'est passé avec la WAM. Avant cette machine la compilation de Prolog semblait impossible (on a dit de Prolog comme de Lisp que c'était un langage incompilable) ou réservée à un cercle restreint d'implanteurs. Depuis la WAM, l'écriture d'un compilateur Prolog ne fait plus figure d'épouvantail et bon nombre d'équipes de recherche ont développé leur propre système. Il en est de même avec la définition de notre machine abstraite pour la compilation des contraintes sur les domaines finis qui sert aujourd'hui de base pour l'intégration de contraintes dans des systèmes tels que : `AKL(FD)` à SICS (S. Janson, B. Carlson), Oz(FD) au DFKI (G. Smolka, J. Wuertz), langage successeur de LIFE à Simon Fraser University (H. Aït-Kaci, S. Le Huitouze), Scheme(FD) à l'INRIA (J-M. Geffroy),...

Nous avons montré les capacités d'extension du solveur en définissant un résolveur booléen écrit en 10 lignes de `clp(FD)`. Celui-ci s'est révélé être 10 fois plus rapide que le résolveur

[1] ce qui n'est pas le cas du peu qui est présenté dans l'article "Overview of the CHIP compiler" qui, avouons-le, ne donne aucune information pratique sur "comment" compiler les contraintes. D'ailleurs le compilateur CHIP, 3 ans après, n'a toujours pas vu le jour.

équivalent de CHIP mais aussi bien plus efficace que beaucoup de solveurs spécialement conçus pour les booléens.

Le système `clp(FD)` est désormais un système complet et opérationnel. Il est disponible par `ftp` et a déjà été récupéré à plus de 350 exemplaires. Des industriels ont été intéressés par `clp(FD)`, citons par exemple:

- le Centre d'Etudes de la Navigation Aérienne (à Orly) qui utilise `clp(FD)` pour la régulation du trafic aérien (cf. chapitre 8).
- Dassault Aviation (à St Cloud) pour faire du raisonnement qualitatif. Les essais préliminaires ont montré que `clp(FD)` était 200 fois plus rapide que le logiciel développé par Dassault.
- N.A.S.A. (en Californie) pour choisir, parmi plusieurs modélisations d'un problème donné, celle qui minimise une certaine fonction de coût (dépendant du nombre de variables, de contraintes,...).

Enfin, nous avons été agréablement surpris de constater que `clp(FD)` était utilisé comme support de cours sur les contraintes (à SICS par M. Carlsson, à l'université de Gênes par A. Messora et dans une université de Grèce).

Notre expérience nous a montré combien l'extensibilité du résolveur était cruciale pour les applications réelles. A tel point qu'il nous faut maintenant sortir du moule trop restrictif des "calculs dirigés par les instructions" pour nous orienter vers des "calculs dirigés par les données". Le cadre CC (langages concurrents avec contraintes) semble le plus prometteur grâce à la simplicité avec laquelle il permet les synchronisations. En effet, par rapport au cadre PLC, il ne nécessite qu'une seule nouvelle opération (nommée *Ask*) permettant de suspendre l'exécution d'un calcul jusqu'au succès d'une certaine contrainte. La détection de la satisfaction de cette contrainte est évidemment le point clé pour implanter le *Ask*. Là encore nous bénéficions de l'approche "boîte de verre" puisqu'il nous suffit de savoir détecter la satisfaction de l'unique primitive. La méthode présentée permet de dériver d'après la syntaxe de la contrainte (i.e. statiquement), une condition qui, aussitôt vraie, nous assure que la contrainte considérée est satisfaite (i.e. condition suffisante). Nous avons donc en main les outils nécessaires au passage à la concurrence. Au travail...

Annexe A

Programme crypta

```
/*-------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)         INRIA Rocquencourt - ChLoE Project */
/*                                                                         */
/* Name           : crypta.pl                                              */
/* Title          : crypt-arithmetic                                       */
/* Original Source: P. Van Hentenryck's book                               */
/* Adapted by     : Daniel Diaz - INRIA France                             */
/* Date           : September 1992                                         */
/*                                                                         */
/* Solve the operation:                                                    */
/*                                                                         */
/*    B A I J J A J I I A H F C F E B B J E A                              */
/*  + D H F G A B C D I D B I F F A G F E J E                              */
/*  ----------------------------------------                               */
/*  = G J E G A C D D H F A F J B F I H E E F                              */
/*                                                                         */
/* Solution:                                                               */
/*  [A,B,C,D,E,F,G,H,I,J]                                                  */
/*  [1,2,3,4,5,6,7,8,9,0]                                                  */
/*-------------------------------------------------------------------------*/

crypta(LD):-
        fd_vector_max(9),
        LD=[A,B,C,D,E,F,G,H,I,J],

        alldifferent(LD),
        domain(LD,0,9),
```

```
domain([Sr1,Sr2],0,1),

B in 1..9,
D in 1..9,
G in 1..9,

   A+10*E+100*J+1000*B+10000*B+100000*E+1000000*F+
   E+10*J+100*E+1000*F+10000*G+100000*A+1000000*F
#= F+10*E+100*E+1000*H+10000*I+100000*F+1000000*B+10000000*Sr1,


   C+10*F+100*H+1000*A+10000*I+100000*I+1000000*J+
   F+10*I+100*B+1000*D+10000*I+100000*D+1000000*C+Sr1
#= J+10*F+100*A+1000*F+10000*H+100000*D+1000000*D+10000000*Sr2,

   A+10*J+100*J+1000*I+10000*A+100000*B+
   B+10*A+100*G+1000*F+10000*H+100000*D+Sr2
#= C+10*A+100*G+1000*E+10000*J+100000*G,

labeling(LD).
```

Annexe B

Programme eq10

```
/*------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)         INRIA Rocquencourt - ChLoE Project */
/*                                                                        */
/* Name           : eq10.pl                                               */
/* Title          : linear equations                                      */
/* Original Source: Thomson LCR                                           */
/* Adapted by     : Daniel Diaz - INRIA France                            */
/* Date           : September 1992                                        */
/*                                                                        */
/* A system involving 7 variables and 10 equations                        */
/*                                                                        */
/* Solution:                                                              */
/*  [X1,X2,X3,X4,X5,X6,X7]                                                */
/*  [ 6, 0, 8, 4, 9, 3, 9]                                                */
/*------------------------------------------------------------------------*/

eq10(Lab):-
        LD = [X1,X2,X3,X4,X5,X6,X7],
        domain(LD,0,10),

           0+98527*X1+34588*X2+5872*X3+59422*X5+65159*X7
        #= 1547604+30704*X4+29649*X6,

           0+98957*X2+83634*X3+69966*X4+62038*X5+37164*X6+85413*X7
        #= 1823553+93989*X1,

           900032+10949*X1+77761*X2+67052*X5
```

```
#= 0+80197*X3+61944*X4+92964*X6+44550*X7,

   0+73947*X1+84391*X3+81310*X5
#= 1164380+96253*X2+44247*X4+70582*X6+33054*X7,

   0+13057*X3+42253*X4+77527*X5+96552*X7
#= 1185471+60152*X1+21103*X2+97932*X6,

   1394152+66920*X1+55679*X4
#= 0+64234*X2+65337*X3+45581*X5+67707*X6+98038*X7,

   0+68550*X1+27886*X2+31716*X3+73597*X4+38835*X7
#= 279091+88963*X5+76391*X6,

   0+76132*X2+71860*X3+22770*X4+68211*X5+78587*X6
#= 480923+48224*X1+82817*X7,

   519878+94198*X2+87234*X3+37498*X4
#= 0+71583*X1+25728*X5+25495*X6+70023*X7,

   361921+78693*X1+38592*X5+38478*X6
#= 0+94129*X2+43188*X3+82528*X4+69025*X7,

labeling(LD).
```

Annexe C

Programme eq20

```
/*------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)         INRIA Rocquencourt - ChLoE Project */
/*                                                                        */
/* Name           : eq20.pl                                               */
/* Title          : linear equations                                      */
/* Original Source: Thomson LCR                                           */
/* Adapted by     : Daniel Diaz - INRIA France                            */
/* Date           : September 1992                                        */
/*                                                                        */
/* A system involving 7 variables and 20 equations                        */
/*                                                                        */
/* Solution:                                                              */
/*  [X1,X2,X3,X4,X5,X6,X7]                                                */
/*  [ 1, 4, 6, 6, 6, 3, 1]                                                */
/*------------------------------------------------------------------------*/

eq20(LD):-
        LD = [X1,X2,X3,X4,X5,X6,X7],
        domain(LD,0,10),

           876370+16105*X1+6704*X3+68610*X6
        #= 0+62397*X2+43340*X4+95100*X5+58301*X7,

           533909+96722*X5
        #= 0+51637*X1+67761*X2+95951*X3+3834*X4+59190*X6+15280*X7,

           915683+34121*X2+33488*X7
```

```
#= 0+1671*X1+10763*X3+80609*X4+42532*X5+93520*X6,

   129768+11119*X2+38875*X4+14413*X5+29234*X6
#= 0+71202*X1+73017*X3+72370*X7,

   752447+58412*X2
#= 0+8874*X1+73947*X3+17147*X4+62335*X5+16005*X6+8632*X7,

   90614+18810*X3+48219*X4+79785*X7
#= 0+85268*X1+54180*X2+6013*X5+78169*X6,

   1198280+45086*X1+4578*X3
#= 0+51830*X2+96120*X4+21231*X5+97919*X6+65651*X7,

   18465+64919*X1+59624*X4+75542*X5+47935*X7
#= 0+80460*X2+90840*X3+25145*X6,

   0+43525*X2+92298*X3+58630*X4+92590*X5
#= 1503588+43277*X1+9372*X6+60227*X7,

   0+47385*X2+97715*X3+69028*X5+76212*X6
#= 1244857+16835*X1+12640*X4+81102*X7,

   0+31227*X2+93951*X3+73889*X4+81526*X5+68026*X7
#= 1410723+60301*X1+72702*X6,

   0+94016*X1+35961*X3+66597*X4
#= 25334+82071*X2+30705*X5+44404*X6+38304*X7,

   0+84750*X2+21239*X4+81675*X5
#= 277271+67456*X1+51553*X3+99395*X6+4254*X7,

   0+29958*X2+57308*X3+48789*X4+4657*X6+34539*X7
#= 249912+85698*X1+78219*X5,

   0+85176*X1+57898*X4+15883*X5+50547*X6+83287*X7
#= 373854+95332*X2+1268*X3,

   0+87758*X2+19346*X4+70072*X5+44529*X7
#= 740061+10343*X1+11782*X3+36991*X6,

   0+49149*X1+52871*X2+56728*X4
#= 146074+7132*X3+33576*X5+49530*X6+62089*X7,
```

```
   0+29475*X2+34421*X3+62646*X5+29278*X6
#= 251591+60113*X1+76870*X4+15212*X7,

   22167+29101*X2+5513*X3+21219*X4
#= 0+87059*X1+22128*X5+7276*X6+57308*X7,

   821228+76706*X1+48614*X6+41906*X7
#= 0+98205*X2+23445*X3+67921*X4+24111*X5,

labeling(LD).
```

Annexe D

Programme alpha

```
/*-------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)         INRIA Rocquencourt - ChLoE Project */
/*                                                                         */
/* Name           : alpha.pl                                               */
/* Title          : alphacipher                                            */
/* Original Source: Daniel Diaz - INRIA France                             */
/* Adapted by     :                                                        */
/* Date           : January 1993                                           */
/*                                                                         */
/* This problem comes from the news group rec.puzzle.                      */
/* The numbers 1 - 26 have been randomly assigned to the letters of the */
/* alphabet. The numbers beside each word are the total of the values   */
/* assigned to the letters in the word. e.g for LYRE L,Y,R,E might equal*/
/* 5,9,20 and 13 respectively or any other combination that add up to 47*/
/* Find the value of each letter under the equations:                      */
/*                                                                         */
/*    BALLET  45     GLEE  66     POLKA      59     SONG     61         */
/*    CELLO   43     JAZZ  58     QUARTET    50     SOPRANO  82         */
/*    CONCERT 74     LYRE  47     SAXOPHONE 134     THEME    72         */
/*    FLUTE   30     OBOE  53     SCALE      51     VIOLIN  100         */
/*    FUGUE   50     OPERA 65     SOLO       37     WALTZ    34         */
/*                                                                         */
/* Solution:                                                               */
/*[A, B,C, D, E,F, G, H, I, J, K,L,M, N, O, P,Q, R, S,T,U, V,W, X, Y, Z]*/
/*[5,13,9,16,20,4,24,21,25,17,23,2,8,12,10,19,7,11,15,3,1,26,6,22,14,18]*/
/*-------------------------------------------------------------------------*/
```

```
alpha(LD):-
        fd_vector_max(26),
        LD=[A,B,C,_D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z],

        alldifferent(LD),
        domain(LD,1,26),

        B+A+L+L+E+T       #= 45,
        C+E+L+L+O         #= 43,
        C+O+N+C+E+R+T     #= 74,
        F+L+U+T+E         #= 30,
        F+U+G+U+E         #= 50,
        G+L+E+E           #= 66,
        J+A+Z+Z           #= 58,
        L+Y+R+E           #= 47,
        O+B+O+E           #= 53,
        O+P+E+R+A         #= 65,
        P+O+L+K+A         #= 59,
        Q+U+A+R+T+E+T     #= 50,
        S+A+X+O+P+H+O+N+E #= 134,
        S+C+A+L+E         #= 51,
        S+O+L+O           #= 37,
        S+O+N+G           #= 61,
        S+O+P+R+A+N+O     #= 82,
        T+H+E+M+E         #= 72,
        V+I+O+L+I+N       #= 100,
        W+A+L+T+Z         #= 34,

        labeling(LD).
%       labelingff(LD).
```

Annexe E

Programme queens

```
/*-------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)         INRIA Rocquencourt - ChLoE Project */
/*                                                                         */
/* Name           : queens.pl                                              */
/* Title          : N-queens problem                                       */
/* Original Source: P. Van Hentenryck's book                               */
/* Adapted by     : Daniel Diaz - INRIA France                             */
/* Date           : January 1993                                           */
/*                                                                         */
/* Put N queens on an NxN chessboard so that there is no couple of         */
/* queens threatening each other.                                          */
/*                                                                         */
/* Solution:                                                               */
/* N=4  [2,4,1,3]                                                          */
/* N=8  [1,5,8,6,3,7,2,4]                                                  */
/* N=16 [1,3,5,2,13,9,14,12,15,6,16,7,4,11,8,10]                           */
/*-------------------------------------------------------------------------*/

queens(N,L):-
        fd_vector_max(N),
        length(L,N),
        domain(L,1,N),
        safe(L),
        labeling(L).
%       labelingff(L).
```

```
safe([]).

safe([X|L]):-
        noattack(L,X,1),
        safe(L).



noattack([],_,_).

noattack([Y|L],X,I):-
        diff(X,Y,I),
        I1 is I+1,
        noattack(L,X,I1).



diff(X,Y,I):-
        X in -{val(Y)} & -{val(Y)-I} & -{val(Y)+I},
        Y in -{val(X)} & -{val(X)-I} & -{val(X)+I}.
```

Annexe F

Programme five

```
/*------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)         INRIA Rocquencourt - ChLoE Project */
/*                                                                        */
/* Name           : five.pl                                               */
/* Title          : five house puzzle                                     */
/* Original Source: P. Van Hentenryck's book                              */
/* Adapted by     : Daniel Diaz - INRIA France                            */
/* Date           : September 1992                                        */
/*                                                                        */
/* A logic puzzle                                                         */
/*                                                                        */
/* Solution:                                                              */
/*  [N1,N2,N3,N4,N5,     [3,4,5,2,1,                                      */
/*   C1,C2,C3,C4,C5,      5,3,1,2,4,                                      */
/*   P1,P2,P3,P4,P5,      5,1,4,2,3,                                      */
/*   A1,A2,A3,A4,A5,      4,5,1,3,2,                                      */
/*   D1,D2,D3,D4,D5]      4,1,2,5,3]                                      */
/*------------------------------------------------------------------------*/

five_house(L):-
        fd_vector_max(5),
        L=[N1,N2,N3,N4,N5,
        C1,C2,C3,C4,C5,
        P1,P2,P3,P4,P5,
        A1,A2,A3,A4,A5,
        D1,D2,D3,D4,D5],
```

```
        domain(L,1,5),

        N5 #= 1,
        D5 #= 3,

        alldifferent([C1,C2,C3,C4,C5]),
        alldifferent([P1,P2,P3,P4,P5]),
        alldifferent([N1,N2,N3,N4,N5]),
        alldifferent([A1,A2,A3,A4,A5]),
        alldifferent([D1,D2,D3,D4,D5]),

        N1 #= C2,
        N2 #= A1,
        N3 #= P1,
        N4 #= D3,
        P3 #= D1,
        C1 #= D4,
        P5 #= A4,
        P2 #= C3,
        C1 #= C5+1,

        plus_or_minus(A3,P4,1),
        plus_or_minus(A5,P2,1),
        plus_or_minus(N5,C4,1),

        labeling(L).


        % partial lookahead

plus_or_minus(X,Y,C):-
        X #= Y+C.

plus_or_minus(X,Y,C):-
        X+C #= Y.


        % constructive disjunction and partial lookahead
/*
plus_or_minus(X,Y,C):-
        X in min(Y)+C..max(Y)+C:min(Y)-C..max(Y)-C,
        Y in min(X)+C..max(X)+C:min(X)-C..max(X)-C.
*/
```

Annexe G

Programme cars

```
/*------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)         INRIA Rocquencourt - ChLoE Project */
/*                                                                        */
/* Name           : cars.pl                                               */
/* Title          : car sequencing problem                                */
/* Original Source: Dincbas, Simonis and Van Hentenryck                   */
/* Adapted by     : Daniel Diaz - INRIA France                            */
/* Date           : September 1992                                        */
/*                                                                        */
/* Car sequencing problem with 10 cars                                    */
/* Solution:                                                              */
/*   [1,2,6,3,5,4,4,5,3,6]                                                */
/*   [1,3,6,2,5,4,3,5,4,6]                                                */
/*   [1,3,6,2,6,4,5,3,4,5]                                                */
/*   [5,4,3,5,4,6,2,6,3,1]                                                */
/*   [6,3,5,4,4,5,3,6,2,1]                                                */
/*   [6,4,5,3,4,5,2,6,3,1]                                                */
/*                                                                        */
/*------------------------------------------------------------------------*/

cars(X):-
        fd_vector_max(6),
        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,


% redundant constraints

        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,

labeling(X).
```

Annexe H

Programme bridge

```
/*-------------------------------------------------------------------------*/
/* Benchmark (Finite Domain)         INRIA Rocquencourt - ChLoE Project */
/*                                                                         */
/* Name           : bridge.pl                                              */
/* Title          : bridge scheduling problem                              */
/* Original Source: P. Van Hentenryck's book and                           */
/*                  COSYTEC (vers. of "Overview of a CHIP Compiler")       */
/* Adapted by     : Daniel Diaz - INRIA France                             */
/* Date           : October 1994                                           */
/*                                                                         */
/* Find a scheduling that minimizes the time to build a 5-segment bridge*/
/*                                                                         */
/* Solution:                                                               */
/*                                                                         */
/* Optimal (End=104)                                                       */
/*                                                                         */
/*[[start,0,0],[a1,4,3],[a2,2,13],[a3,2,7],[a4,2,15],[a5,2,1],[a6,5,38],*/
/*[p1,20,9],[p2,13,29],[ue,10,0],[s1,8,10],[s2,4,18],[s3,4,29],[s4,4,42]*/
/*[s5,4,6],[s6,10,46],[b1,1,18],[b2,1,22],[b3,1,33],[b4,1,46],[b5,1,10],*/
/*[b6,1,56],[ab1,1,19],[ab2,1,23],[ab3,1,34],[ab4,1,47],[ab5,1,11],     */
/*[ab6,1,57],[m1,16,20],[m2,8,36],[m3,8,44],[m4,8,52],[m5,8,12],        */
/*[m6,20,60],[l1,2,30],[t1,12,44],[t2,12,56],[t3,12,68],[t4,12,92],     */
/*[t5,12,80],[ua,10,78],[v1,15,56],[v2,10,92],[k1,0,42],[k2,0,80],      */
/*[stop,0,104]]                                                          */
/*-------------------------------------------------------------------------*/

bridge(K,Ende):-
```

```
        setup(K,Ende,Disj),
        minof(choice(Disj,K),Ende).


setup(K,Ende,Disj):-
        jobs(L),
        make_vars(L,K),
        member([stop,_,Ende],K),
        precedence(M),
        make_precedence(M,K),
        max_nf(M1),
        make_max_nf(M1,K),
        max_ef(M2),
        make_max_ef(M2,K),
        min_af(M3),
        make_min_af(M3,K),
        min_sf(M4),
        make_min_sf(M4,K),
        min_nf(M5),
        make_min_nf(M5,K),
        resources(R),
        make_disj(R,K,[],Disj1),
        reverse(Disj1,Disj).

choice(Disj,K):-
        disjunct(Disj),
        label(K).


make_vars([],[]).
make_vars([H|T],[[H,D,A]|R]):-
        duration(H,D),
        A in 0..200,
        make_vars(T,R).

make_precedence([],_).
make_precedence([[A,B]|R],L):-
        member([A,Ad,Aa],L),
        member([B,Bd,Ba],L),
        greatereqc(Ba,Aa,Ad),                % Ba #>= Aa+Ad,
        make_precedence(R,L).

make_max_nf([],_).
```

```
make_max_nf([[A,B,C]|R],L):-
        member([A,Ad,Aa],L),
        member([B,Bd,Ba],L),
        C1 is C + Ad,
        smallereqc(Ba,Aa,C1),                % Ba #<= Aa+C1,
        make_max_nf(R,L).

make_max_ef([],_).
make_max_ef([[A,B,C]|R],L):-
        member([A,Ad,Aa],L),
        member([B,Bd,Ba],L),
        C1 is Ad + C - Bd,
        smallereqc(Ba,Aa,C1),                % Ba #<= Aa+C1,
        make_max_ef(R,L).

make_min_af([],_).
make_min_af([[A,B,C]|R],L):-
        member([A,Ad,Aa],L),
        member([B,Bd,Ba],L),
        greatereqc(Ba,Aa,C),                 % Ba #>= Aa+C,
        make_min_af(R,L).

make_min_sf([],_).
make_min_sf([[A,B,C]|R],L):-
        member([A,Ad,Aa],L),
        member([B,Bd,Ba],L),
        C1 is C - Bd,
        smallereqc(Ba,Aa,C1),                % Ba #<= Aa+C1,
        make_min_sf(R,L).

make_min_nf([],_).
make_min_nf([[A,B,C]|R],L):-
        member([A,Ad,Aa],L),
        member([B,Bd,Ba],L),
        C1 is C + Ad,
        greatereqc(Ba,Ad,C1),                % Ba #>= Ad+C1,
        make_min_nf(R,L).

make_disj([],R,D,D).
make_disj([[H,R]|T],K,Din,Dout):-
        el_list(R,K,R1),
        make_disj1(R1,Din,D1),
        make_disj(T,K,D1,Dout).
```

```
make_disj1([],D,D).
make_disj1([H|T],Din,Dout):-
        make_disj2(H,T,Din,D1),
        make_disj1(T,D1,Dout).

make_disj2(H,[],D,D).
make_disj2([A,B],[[C,D]|S],Din,Dout):-
        make_disj2([A,B],S,[[A,B,C,D]|Din],Dout).

el_list([],_,[]).
el_list([H|T],L,[[A,D]|S]):-
        member([H,D,A],L),
        el_list(T,L,S).

disjunct([]).
disjunct([[A,B,C,D]|R]):-
        disj(A,B,C,D),
        disjunct(R).

disj(Aa,Ad,Ba,Bd):-
        greatereqc(Ba,Aa,Ad).                % Ba #>= Aa+Ad.

disj(Aa,Ad,Ba,Bd):-
        greatereqc(Aa,Ba,Bd).                % Aa #>= Ba+Bd.




label([]).
label([[A,Ad,Aa]|R]):-
        indomain(Aa),
        label(R).




          /* constraint definitions */

smallereqc(X,Y,C):-                         % X #<= Y+C
        X in 0       ..max(Y)+C,
        Y in min(X)-C..infinity.
```

```
greatereqc(X,Y,C):-                          % X #>= Y+C.
        X in min(Y)+C..infinity,
        Y in 0       ..max(X)-C.



/*

                DATA

*/

jobs([start,a1,a2,a3,a4,a5,a6,p1,p2,ue,s1,s2,s3,s4,s5,s6,
        b1,b2,b3,b4,b5,b6,ab1,ab2,ab3,ab4,ab5,ab6,m1,m2,m3,m4,m5,m6,
        l1,t1,t2,t3,t4,t5,ua,v1,v2,k1,k2,stop]).

duration(start,0).
duration(a1,4).
duration(a2,2).
duration(a3,2).
duration(a4,2).
duration(a5,2).
duration(a6,5).
duration(p1,20).
duration(p2,13).
duration(ue,10).
duration(s1,8).
duration(s2,4).
duration(s3,4).
duration(s4,4).
duration(s5,4).
duration(s6,10).
duration(b1,1).
duration(b2,1).
duration(b3,1).
duration(b4,1).
duration(b5,1).
duration(b6,1).
duration(ab1,1).
```

```
duration(ab2,1).
duration(ab3,1).
duration(ab4,1).
duration(ab5,1).
duration(ab6,1).
duration(m1,16).
duration(m2,8).
duration(m3,8).
duration(m4,8).
duration(m5,8).
duration(m6,20).
duration(l1,2).
duration(t1,12).
duration(t2,12).
duration(t3,12).
duration(t4,12).
duration(t5,12).
duration(ua,10).
duration(v1,15).
duration(v2,10).
duration(k1,0).
duration(k2,0).
duration(stop,0).

precedence([[start,a1],[start,a2],[start,a3],[start,a4],[start,a5],
        [start,a6],[start,ue],[a1,s1],[a2,s2],[a5,s5],
        [a6,s6],[a3,p1],[a4,p2],[p1,s3],[p2,s4],
        [p1,k1],[p2,k1],[s1,b1],[s2,b2],
        [s3,b3],[s4,b4],[s5,b5],[s6,b6],[b1,ab1],
        [b2,ab2],[b3,ab3],[b4,ab4],[b5,ab5],[b6,ab6],
        [ab1,m1],[ab2,m2],[ab3,m3],[ab4,m4],[ab5,m5],
        [ab6,m6],[m1,t1],[m2,t1],[m2,t2],[m3,t2],
        [m3,t3],[m4,t3],[m4,t4],[m5,t4],[m5,t5],
        [m6,t5],[m1,k2],[m2,k2],[m3,k2],[m4,k2],
        [m5,k2],[m6,k2],[l1,t1],[l1,t2],[l1,t3],
        [l1,t4],[l1,t5],[t1,v1],[t5,v2],[t2,stop],
        [t3,stop],[t4,stop],[v1,stop],[v2,stop],[ua,stop],
        [k2,stop]]).

max_nf([[start,l1,30],[a1,s1,3],[a2,s2,3],[a5,s5,3],
        [a6,s6,3],[p1,s3,3],[p2,s4,3]]).

min_sf([[ua,m1,2],[ua,m2,2],[ua,m3,2],[ua,m4,2],
```

```
                [ua,m5,2],[ua,m6,2]]).

max_ef([[s1,b1,4],[s2,b2,4],[s3,b3,4],[s4,b4,4],[s5,b5,4],[s6,b6,4]]).

min_nf([[start,l1,30]]).

min_af([[ue,s1,6],[ue,s2,6],[ue,s3,6],[ue,s4,6],[ue,s5,6],[ue,s6,6]]).


resources([[crane,[l1,t1,t2,t3,t4,t5]],
        [bricklaying,[m1,m2,m3,m4,m5,m6]],
        [schal,[s1,s2,s3,s4,s5,s6]],
        [excavator,[a1,a2,a3,a4,a5,a6]],
        [ram,[p1,p2]],
        [pump,[b1,b2,b3,b4,b5,b6]],
        [caterpillar,[v1,v2]]]).
```

Annexe I

Manuel d'utilisation de wamcc

wamcc 2.21 User's Manual

Daniel Diaz

INRIA-Rocquencourt
Domaine de Voluceau
78153 Le Chesnay
FRANCE

Daniel.Diaz@inria.fr

July 1994

This manual is based on DECsystem-10 Prolog User's Manual by D.L. Bowen, L. Byrd, F.C.N. Pereira, L.M. Pereira, D.H.D. Warren and on SICStus Prolog User's Manual by Mats Carlsson

I.1 Using wamcc - Modularity

wamcc is a Prolog compiler which translates Prolog to C via the WAM (Warren Abstract Machine). A Prolog file will give rise to a C source file which will be compiled by gcc. Several Prolog files can be compiled separately and linked by the loader to provide a Unix executable. A prolog file is a *module*. It is possible (and recommended) to split a big application into several little modules. Each module has its own independent predicate name space. This is an important feature for the development of larger programs. The module system of wamcc Prolog is procedure based. This means that only the predicates are local to a module, whereas terms are global. The module system is flat, not hierarchical, so all modules are visible to one another. No overhead is incurred on compiled calls to predicates in other modules. Each predicate in the Prolog system, whether built-in or user defined, belongs to a module. By default, a predicate is generally only visible in the module where it is defined (i.e. *private* predicate) except if this predicate has been declared as *public* (see directive `:- public` below). Public predicates are visible in every modules. A public predicate can be *locally* redefined in any module. There is a particular module: the *main module* corresponding to the module containing the main program[1] (see directives `:- main` below). At run-time only public predicates and private predicates defined in the main module will be visible under the top level. wamcc also supports *dynamic predicates* which are interpreted (see section I.3.8).

The following directives are specially handled at compile-time:

`:- public` $\texttt{Pred}_1\texttt{/Arity}_1$`, ...,` $\texttt{Pred}_n\texttt{/Arity}_n$`.`

This directive specifies that each $\texttt{Pred}_i\texttt{/Arity}_i$ is a public predicate.

`:- dynamic` $\texttt{Pred}_1\texttt{/Arity}_1$`, ...,` $\texttt{Pred}_n\texttt{/Arity}_n$`.`

This directive specifies that each $\texttt{Pred}_i\texttt{/Arity}_i$ is a dynamic predicate (see section I.3.8 for dynamic code facilities).

`:- main.`

`:- main(+Modules).`

[1] The main module will contain the C funtion `main()`.

`:- main(+Modules,+Stacks).`

This directive specifies the current module is the *main module*. `Modules` is the list of needed modules (omitted if there are no other modules). `Stacks` is a list of *stack parameters* of the form `stack(+Name,+EnvVarName,+Size)`. `Name` is the stack name (see section I.4). `EnvVarName` is the name of the environment variable to consult at run-time to read the size of the stack. If `EnvVarName` is the empty atom (i.e. `''`) no environment variable will be consulted at run-time for this stack. `Size` is the default stack size in Kbytes (if the variable `EnvVarName` is not defined at run-time). If there is not any definition for a stack *stack_name* the default environment variable is *stack_name*`SZ` in upper case (see also section I.2.4).

Note that other directives are not executed at compile-time but they are compiled into special code which will be executed at run-time[2]. More precisely, at run-time, all directives of each module defined in `Modules` are executed then the directives of the main module are executed. When the main module is compiled, a directive is added to invoke the top level (see `top_level/2` in section I.3.5). If several directives are defined in a same module, they will be executed in the same order as they appear in the Prolog file.

Example:

```
x.pl                        y.pl                        z.pl

:- main([y,z]).             :- public odd/1.            :- public p/1.

:- public even/1.           odd(s(X)):- even(X).        p(X):- u(X).

even(0).                    :- public q/1.              u(b).

even(s(X)):- odd(X).        q(X):- p(X).                :- write(r).

p(a).                       :- write(q1).
                            :- write(q2).
s(X):- q(X).

:- p(X), write(X).
:- s(X), write(X).
```

[2]The only exception concerns operator declarations `op/3` which are both executed at compile-time and run-time.

At run-time:

exec directive	goal	output
1 of y	`write(q1)`	q1
2 of y	`write(q2)`	q2
1 of z	`write(r)`	r
1 of x	`p(X), write(X)`	a
2 of x	`s(X), write(X)`	b
3 of x	`top_level(true,true)`	...

- The order in which directives are executed is: directives of `y.pl`, directives of `z.pl` and then directives of `x.pl`.
- In the module `x.pl` the public predicate `p/1` has been redefined so the directive `p(X), write(X)` will print `a`. However, the directive `s(X), write(X)` will print `b` since in `y.pl` only the public declaration of `p/1` in `z.pl` is visible.
- Under the top-level, the query `| ?- p(X)` will succed with `X=a`. The query `q(X)` will succeed with `X=b`, the query `u(X)` will fail since only private predicates defined in `x.pl` can be called from the top-level. The query `even(X)` will generate all even natural numbers through backtraking.

I.2 From Prolog modules to Unix Executables

I.2.1 Compiling Prolog Modules

The `wamcc` compiler is entirely written in `wamcc` Prolog (and compiled with `wamcc`). It allows the user to compile one or several Prolog Modules. The general syntax of `wamcc` is as follows:

```
wamcc [option | filename]...
```

Options:

`-c`	produce a .c file
`-wam`	produce a .wam file
`-fast_math`	do not test types in math expressions
`-no_test_stack`	do not test types in math expressions
`-no_inline`	do not include code to test stack overflow
`-dbg`	compile for prolog debugging
`-dbg2`	compile for prolog and wam debugging
`-v`	verbose mode
`-h`	display help

Remarks:

- if `wamcc` is invoked without any arguments then a (classical) Prolog top level is called.
- by default the compilation proceeds in C mode. When a Prolog Module `prog.pl` is compiled in C mode, `wamcc` generates the following files:

  `prog.h` a header file (variable declarations,...)

  `prog.c` the C source file

  `prog.usr` a user file. In this file the user can add his own C code to interface with Prolog via `pragma_c/1` inline predicate. This file is generated if it does not exist otherwise it is left unchanged.

  With the `-wam` options, `wamcc` generates a `prog.wam` file in a Prolog syntax.

- the `-fast_math` option makes it possible to generate faster code since no type checking is done for variables appearing in mathemetic expressions (they are assumed to contain integers).
- the `-dbg` and `-dbg2` options produce additional information for debugging (see also section I.4).
- the `-no_inline` option is useful when debugging since it allows the user to trace *inline predicates*. An inline predicate, is a predicate which does not give rise to a classical Prolog call and thus it is not visible by the debugger. The `-no_inline` option allows the user to trace calls to inline predicates as well.

- the `no_test_stack` is only useful for architecture which cannot test stack overflows by hardware (e.g. Sony Mips, NeXT stations). On these machines, tests are done by software and are quite expensive. With this option, overflows are not checked and thus the program runs faster. However, overflows are not detected.
- if the `-v` option is not specified `wamcc` runs silently.
- `wamcc` returns 0 if the compilation has succeeded and 1 otherwise.

I.2.2 Generating Object Files

From the C files generated by `wamcc` it is possible to obtain an object file using the GNU C compiler. A shell-script `w_gcc`[3] can also is provided t make this compilation. `w_gcc` only invokes `gcc` with some necessary appropriate options (like `-Ipath`...)[4]. `w_gcc` can be used as simple as the C compiler. The following sequence generates a file `prog.o` from the Prolog file `prog.pl`:

```
%wamcc prog -fast_math

%w_gcc -c -O2 prog.c
```

I.2.3 Linking Object Files

From one or several object files $\texttt{prog}_1\texttt{.o}$, ..., $\texttt{prog}_n\texttt{.o}$ it is possible to generate an executable simply by invoking `w_gcc` with these files as follows:

```
%w_gcc -o prog prog1.o ... progn.o -lwamcc
```

The command will create the executable `prog` by linking all object files `prog[1-n].o`. A library containing the run-time and the built-in predicates must be provided via the option `-l`*library*. There are two libraries available `libwamcc.a` and `libwamcc_pp.a`. The former is the standard library while the later allows the user to obtaind *profile information* when quiting the program.

[3] the name of this shell script is `w_`*C_compiler_name*.

[4] so if `wamcc` is moved from one directory to another one, `w_gcc` must be edited to update the pathnames.

hex2pl and pl2hex Utilities

At compile-time, each predicate gives rise to a C identifier which is the hexadecimal representation of the predicate. At linking-time, if a symbol is multiply defined or not defined the linker will emit a message involving the hexadecimal name of the predicate. Two utilities are provided to allow the user to translate in both directions Prolog atoms and hexadecimal representations:

`%pl2hex` *prolog_name* (Prolog to hexadecimal)

Example: `pl2hex append/3` displays `X617070656E64_3`.

`%hex2pl` *hexa_name* (hexadecimal to Prolog)

Example: `hex2pl X617070656E64_3` will display `append/3`.

I.2.4 Stack Overflow Messages

When one stack overflows, `wamcc` exits with the following message: "Fatal Error: *stack_name* stack overflow (size:*current_size* Kb, env. variable: *env_var_name*)" where *stack_name* is the name of the stack which has overflowed, *current_size* is its actual size in Kbytes and *env_var_name* is the name of the environment variable which is consulted to define the size of this stack. You can then (re)define this variable (e.g. with the Unix `setenv` command) and reexecute the program (see also directive `:- main` in section I.1).

I.2.5 Makefile Generator - bmf_wamcc Utility

In order to simplify the overall compilation process, it is possible to define makefiles. A simple program `bmf_wamcc` (Build Make File) provides a way to define automatically makefiles. The obtained makefile can be customized if necessary. The general syntax of `bmf_wamcc` is as follows:

```
bmf_wamcc [option | filename]...
```

Options:

`-o file`	choose `file` as main module (default: first module)
`-P pflags`	use `pflags` for the `wamcc` compiler
`-C cflags`	use `pflags` for the `w_gcc` compiler (default: -O2)
`-L cflags`	use `pflags` for the linker (default: -s)
`-v`	verbose mode
`-h`	display help

A `filename` is a module name (possibly suffixed `.pl`), another object file (`.o`) or an archive file (`.a`).
This command creates a makefile *main_module*`.mk`.

For instance, to create the executable `x` corresponding to the example given in section I.1:

```
%bmf_wamcc x y z -v
```

This creates the makefile `x.mk`. The following command then (re)creates the executable `x`:

```
%make -f x.mk
```

Each makefile generated by `bmf_wamcc` can (re)create a profile executable whose name is *executable_name*`_pp`[5]. For instance the following command will create the executable `x_pp`:

```
%make -f x.mk x_pp
```

I.3 Built-in Predicates

I.3.1 Input / Output

DEC-10 Prolog File Input/Output

The set of file manipulation predicates is inherited from DEC-10 Prolog. The file `user` represents the terminal.

`see(+File)`

The file `File` becomes the current input.

`seeing(?File)`

`File` is unified with the name of the current input.

[5] `..._pp` stands for *Prolog profile*.

`seen`

Closes the current input and resets it to `user`.

`tell(+File)`

The file `File` becomes the current output.

`telling(?File)`

`File` is unified with the name of the current output.

`told`

Closes the current output and resets it to `user`.

Characted Input/Output

`nl`

A new line is started on the current output.

`get0(?N)`

`N` is the characted code of the next character read from the current input. On end of file `N` is -1.

`get(?N)`

`N` is the characted code of the next character that is not a layout characted read from the current input.

`skip(+N)`

Skips just past the next character code `N` from the current input. `N` may be an arithmetic expression.

`put(+N)`

The character code `N` is output onto the current output. `N` may be an arithmetic expression.

`tab(+N)`

`N` spaces are output onto the current output. `N` may be an arithmetic expression.

Input and Output of Terms

`read_line(?X)`

Reads characters from the current input until a `NEWLINE` character is read. The `NEWLINE` character is discarded and the result is an atom. Fails if the end of the file is encountered.

`read_word(?X)`

Skips leading separator characters and reads characters from the current input until a separator character is reached (but not read). Fails if the end of the file is encountered.

`read_integer(?X)`

Skips leading separator characters and reads the next integer from the current input. Fails if the end of the file is encountered.

`read(?Term)`

The next term delimited by a full-stop is read from the current input. When the end of the file is reached, `Term` is unified with the term `end_of_file`.

`read_term(?Term,+Options)`

Same as `read/1` with a list of options. `Options` is a list of :

`variables(?Vars)`

`Vars` is bound to a list of variables of `Term`.

`variable_names(?Names)`

`Names` is bound to a list `Name=Var` pairs where each `Name` is an atom indicating the name of a non-anonymous variable in the term and `Var` is the corresponding variable.

`singletons(?Names)`

`Name` is bound to a list `Name=Var` pairs, one for each non-anonymous variable only appearing once in the term.

`syntax_errors(+Val)`

controls what action to take on syntax errors. Possible values are : `dec10` (the syntax error is reported and read is repeated), `error` (an exception is raised), `fail` (the error is reported and the read fails), `quiet` (the read quietly fails).

`write(?Term)`

The term `Term` is written onto the current output according to the current operator declaration.

`write_canonical(?Term)`

Similar to `write(Term)` but the term is written according to the standard syntax. The output can be parsed by `read/1`.

`writeq(?Term)`

Similar to `write(Term)` but the names of atoms ans functors are quoted where necessary.

`write_term(+Term,+Options)`

Same as `write/1` with a list of options (`Bool` is either `false` or `true`):

`quoted(Bool)`

If selected, functors are quoted where necessary to make the result acceptable as input to `read/1`.

`ignore_ops(+Bool)`

If selected, `Term` is written in standard parenthesized notation instead of using operators.

`numbervars(+Bool)`

If selected, occurrences of `'$VAR'(I)` where `I` is an integer $>= 0$ are written as (`A` + (`I` mod 26))(`I`/26). For `I=0,...` you get the variable names A,B,..., Z, A1, B1, etc.

`max_depth(N)`

Depth limit on printing. `N` is an integer. -1 (the default) means no limit.

`format(+Format,+Arguments)`

Print `Arguments` onto the current output according to format `Format`. `Format` is a list of formatting characters. `format/2` and `format/3` is a Prolog interface to the C `stdio` function `printf()`. It is due to Quintus/Sicstus Prolog.

`Arguments` is a list of items to be printed. If there is only one item it may be supplied as an atom. If there are no items then an empty list should be supplied.

The default action on a format character is to print it. The character `~`and `%` introduce a control sequence. To print a `~`or a `%` repeat it.

The general format of a control sequence is `~NC`. The character `C` determines the type of the control sequence. `N` is an optional numeric argument. An alternative form of `N` is `*`. `*` implies that the next argument in `Arguments` should be used as a numeric argument in the control sequence.

The following control sequences are available.

`~a` The argument is an atom. The atom is printed without quoting.

`~Nc` The argument is a number that will be interpreted as a character code. `N` defaults to one and is interpreted as the number of times to print the character.

`~Nd` The argument is an integer. `N` is interpreted as the number of digits after the decimal point. If `N` is 0 or missing, no decimal point will be printed.

`~ND` The argument is an integer. Identical to `~Nd` except that `,` will separate groups of three digits to the left of the decimal point.

`~Nr` The argument is an integer. `N` is interpreted as a radix. `N` should be >= 2 and <= 36. If `N` is missing the radix defaults to 8. The letters `a-z` will denote digits larger than 9.

`~NR` The argument is an integer. Identical to `~Nr` except that the letters `A-Z` will denote digits larger than 9.

`~Ns` The argument is a list of character codes. Exactly `N` characters will be printed. `N` defaults to the length of the string.

`~i` The argument, which may be of any type, is ignored.

`~k` The argument, which may be of any type, will be passed to `write_canonical/1`.

`~q` The argument may be of any type. The argument will be passed to `writeq/1`.

`~w` The argument may be of any type. The argument will be passed to `write/1`.

`~~` Takes no argument. Prints `~`.

`~Nn` Takes no argument. Prints `N` newlines. `N` defaults to 1.

`~?` The argument is an atom and is considered as the current format (indirection). Example:

```
format("month: ~?, year: ~?",['~a',january,'~d',1994]).
```

will print: `month: january, year: 1994`.

`%F F` is a C `printf` format for integers and atoms (i.e. C integer and C strings).

Example: `format("%02d %3.3s %4d",[1,january,1994]).`

will print: `01 jan 1994`.

`formata(+Format,+Arguments)`

Like `format/2` but `Format` is an atom and `Arguments` must be a list (faster than `format/2`).

`pp_clause(+Clause)`

Pretty-prints the clause `Clause` onto the current output (used by `listing/1`).

`pp_clause(+Head,?Body)`

Like `pp_clause((Head :- Body))`.

I.3.2 Arithmetic

Arithmetic is performed by built-in predicates which take as argument arithmetic expressions and evaluate them. An arithmetic expression is a term built from the numbers, variables and functors that represent arithmetic functions. When an arithmetic expression is evaluated, each variable must be bound to an arithmetic expression. However, if you use the option `-fast_math` then the compiler assumes that each variable will be bound to an integer and does not check its type.

The range of integers are `[-2^28,+2^28-1]`. Floats are not supported in this version.

`+(X)`	`X`
`-X`	negative of `X`
`X+Y`	sum of `X` and `Y`
`X-Y`	difference of `X` and `Y`
`X*Y`	product of `X` and `Y`
`X//Y`	integer quotient of `X` and `Y`
`X mod Y`	integer remainder after dividing `X` by `Y`
`X/\Y`	bitwise and of `X` and `Y`
`X\/Y`	bitwise or of `X` and `Y`
`X^Y`	bitwise exclusive or of `X` and `Y`
`\(X)`	bitwise not of `X`
`X<<Y`	`X` shifted left by `Y` places
`X>>Y`	`X` shifted right by `Y` places
`[X]`	a list of one number `X` evaluates to `X`

Arithmetic expressions are just Prolog terms. If you want one evaluated you must pass it as an argument to one of the following built-in (where `X` and `Y` stand for arithmetic expressions and `Z` for some term).

`Z is X` (*inline predicate*)
: `X` is evaluated and the value is unified with `Z`.

`X=:=Y` (*inline predicate*)
: `X` is equal to `Y`.

`X=\=Y` (*inline predicate*)
: `X` is not equal to `Y`.

`X<Y` (*inline predicate*)
: `X` is less than `Y`.

`X=<Y` (*inline predicate*)
: `X` is less than or equal to `Y`.

`X>Y` (*inline predicate*)
: `X` is greater than `Y`.

`X>=Y` (*inline predicate*)

`X` is greater than or equal to `Y`.

I.3.3 Term Management

Term Comparison

The predicates make references to a *standard total ordering* which is as follows:

- Variables in standard order (roughly oldest first).
- Integers in numeric order.
- Atoms in alphabetic order.
- Compound term, ordered first by the arity, then by the name of the principal functor, then by the arguments (in left-to-right order). Recall that lists are equivalent to compound terms with principal `./2`.

For the following predicates `X` and `Y` are terms.

`compare(?Op,?X,?Y)` (*inline predicate*)

Compares `X` and `Y` and unifies `Op` with `=`, `<` or `>`.

`X==Y` (*inline predicate*)

`X` is literally equal to `Y`.

`X\==Y` (*inline predicate*)

`X` is not literally equal to `Y`.

`X@<Y` (*inline predicate*)

`X` is literally less than `Y`.

`X@=<Y` (*inline predicate*)

`X` is literally less than or equal to `Y`.

`X@>Y` (*inline predicate*)

`X` is literally greater than `Y`.

`X@>=Y` (*inline predicate*)

`X` is literally greater than or equal to `Y`.

Some further predicates involving comparison of terms are:

`sort(+L1,?L2)`

The elements of the list `L1` are sorted into the total orderer and any identical elements are merged yielding the list `L2` (complexity: $O(N\ log\ N)$ where N is the length of `L1`).

`keysort(+L1,?L2)`

The list `L1` must consist of items of the form `Key-Value`. These items are sorted into order according to the value of `Key` yielding the list `L2`. No merging takes place and this predicate is stable (if `K-A` occurs before `K-B` then `K-A` will take place before `K-B` in the output). (complexity: $O(N\ log\ N)$ where N is the length of `L1`).

Constant Processing

There are 3 ways of representing character-string data:

- Atoms (e.g. `'Hello World'`). Atoms are stored in the symbol table (a hash-table).
- Lists of one-character atoms (e.g. `[H,e,l,l,o,' ',w,o,r,l,d]`).
- Strings (e.g. "Hello World") where a string is a list of numeric codes (e.g. `[72,101,108,108,111,32,87,111,114,108,100]`).

`atom_length(+Atom,?Length)`

Length of `Atom` is `Length`.

`atom_concat(?Atom1,?Atom2,+Atom3)`

`atom_concat(+Atom1,+Atom2,-Atom3)`

Concatenes `Atom1` and `Atom2` to give `Atom3`.

`sub_atom(+Atom,?Start,?Length,?Atom1)`

The subatom of `Atom` beginning at the `Start`*th* character and `Length` characters long is `Atom1`.

`char_code(+Char,?Code)`

`char_code(?Char,+Code)`

Unifies the character `Char` with its the character code `Code`.

`chars_codes(+Chars,?Codes)`

`chars_codes(?Chars,+Codes)`

Interconverts the list of chars `Chars` with the list of corresponding character codes `Codes`.

`atom_codes(+Atom,?Codes)`

`atom_codes(?Atom,+Codes)`

Interconverts `Atom` with the corresponding list of character codes `Codes`.

`atom_chars(+Atom,?Chars)`

`atom_chars(?Atom,+Chars)`

Interconverts `Atom` with the corresponding list of characters `Chars`.

`number_atom(+Number,?Atom)`

`number_atom(?Number,+Atom)`

Interconverts `Atom` with the corresponding number `Number`.

`number_codes(+Number,?Codes)`

`number_codes(?Number,+Codes)`

Interconverts `Number` with the corresponding list of character codes `Codes`.

`number_chars(+Number,?Chars)`

`number_chars(?Number,+Chars)`

Interconverts `Number` with the corresponding list of characters `Chars`.

`name(+X,?Codes)`

`name(?X,+Codes)`

If `X` is an atom, equivalent to `atom_codes(X,Codes)`. If `X` is a number, equivalent to `number_codes(X,Codes)`. If `X` is uninstantiated, if `Codes` can be interpreted as a number `X` is unified with that number otherwise with the atom whose name corresponds to `Codes`.

Term Processing

`?Term1=?Term2` (*inline predicate*)

Unifies `Term1` and `Term2`.

`functor(+Term,?Name,?Arity)`

`functor(?Term,+Name,+Arity)` (*inline predicate*)

The principal functor of term `Term` has name `Name` and arity `Arity`, where `Name` is either an atom or, provided `Arity` is 0, an integer. Initially, either `Term` must be instantiated, or `Name` and `Arity` must be instantiated to, respectively, either an atom and an integer in [0..255] or an atomic term and 0. In the case where `Term` is initially uninstantiated, the result of the call is to instantiate `Term` to the most general term having the principal functor indicated.

`arg(+ArgNo,+Term,?Arg)` (*inline predicate*)

Initially, `ArgNo` must be instantiated to a positive integer and `Term` to a compound term. The result of the call is to unify `Arg` with the argument `ArgNo` of term `Term`. The arguments are numbered from 1 upwards.

`+Term =.. ?List`

`?Term =.. +List` (*inline predicate*)

`List` is a list whose head is the atom corresponding to the principal functor of `Term`, and whose tail is a list of the arguments of `Term`. If `Term` is uninstantiated, then `List` must be instantiated either to a list of determinate length whose head is an atom, or to a list of length 1 whose head is a number. Note that this predicate is not strictly necessary, since its functionality can be provided by `arg/3` and `functor/3`, and using the latter two is usually more efficient.

`copy_term(?Term,?CopyOfTerm)`

`CopyOfTerm` is a renaming of `Term`, such that brand new variables have been substituted for all variables in `Term`.

`setarg(+ArgNo,+CompoundTerm,?NewArg)`

Replaces destructively argument `ArgNo` in `CompoundTerm` with `NewArg` and undoes it on backtracking. This should only used if there is no further use of the "old" value of the replaced argument.

`numbervars(?Term,+N,?M)`

Unifies each of the variables in term `Term` with a special term `'$VAR'(i)` where `i` ranges from `N` to `M`-1 (see `write/1` or `writeq/1`). This predicate is used by `listing/1`.

I.3.4 Test Predicates

The following test the type of the term `X`:

`var(?X)` (*inline predicate*)

Tests whether `X` is currently uninstantiated (`var` is short for variable). An uninstantiated variable is one which has not been bound to anything, except possibly another uninstantiated variable.

`nonvar(?X)` (*inline predicate*)

Tests whether `X` is currently instantiated. This is the opposite of `var/1`.

`atom(?X)` (*inline predicate*)

Checks that `X` is currently instantiated to an atom (i.e. a non-variable term of arity 0, other than a number).

`integer(?X)` (*inline predicate*)

Checks that `X` is currently instantiated to an integer.

`number(?X)` (*inline predicate*)

Checks that `X` is currently instantiated to a number (i.e. an integer).

`atomic(?X)` (*inline predicate*)

Checks that `X` is currently instantiated to an atom or number.

`compound(?X)` (*inline predicate*)

Checks that `X` is currently instantiated to a term of arity > 0 i.e. a list or a structure.

`callable(?X)` (*inline predicate*)

Checks that `X` is currently instantiated to a callable term (i.e. an atom or a compound term).

I.3.5 Control

`+P , +Q`

Prove `P` and if it succeeds, then prove `Q`.

`+P ; +Q`

Prove `P` or if it fails (or if the continuation fails), prove `Q` instead of `P`.

`!`

The effect of the cut symbol is as follows. When first encountered as a goal, cut succeeds immediately. If backtracking should later return to the cut, the effect is to fail the parent goal, i.e. that goal which matched the head of the clause containing the cut, and caused the clause to be activated.

`\+ +P`

If the goal `P` has a solution, fail, otherwise succeed. This is not real negation ("`P` is false"), but a kind of pseudo-negation meaning "`P` is not provable".

`+P -> +Q ; +R`

Analogous to "if `P` then `Q` else `R`" This form of if-then-else only explores the first solution to the goal `P`.

`+P -> +Q`

Equivalent to `(P -> Q ; fail)`.

`otherwise`

`true`

These always succeed.

`false`

`fail`

These always fail.

`repeat`

Generates an infinite sequence of backtracking choices.

`for(I,A,B)`

Generates, on backtracking, values for `I` ranging from `A` to `B`.

`call(+Term)`

`Term` must be instantiated to a term (which would be acceptable as the body of a clause). The goal `call(Term)` is then executed exactly as if that term appeared textually in its place, except that any cut (`!`) occurring in `Term` only cuts alternatives in the execution of `Term`.

`halt(+Status)`

`halt`

Terminates the Prolog process with the status `Status`. `halt/0` is a shorthand for `halt(0)`.

`halt_or_else(+Program,+Status)`

`halt_or_else(+Program)`

Same as `halt(Status)` if there is a top level else same as `call(Program)`.

`abort`

Stops the current goal and returns under the top level if exists or exits with the status 1.

`catch(+Goal,?Catcher,+Recovery)`

`throw(?Ball)`

`catch/3` calls `Goal`. If this succeeds or fails, so does the call to `catch/3`. If however, during the execution of `Goal`, there is a call to `throw(Ball)` (i.e. an exception is raised), then `Ball` is copied and the stack is unwound back to the call to `catch/3`, whereupon the copy of `Ball` is unified with `Catcher`. If this unification succeeds,

then `catch/3` calls the goal `Recovery` (a handler) in order to determine the success or failure of `catch/3`. Otherwise, the stack keeps unwinding, looking for an earlier invocation of `catch/3`.

I.3.6 List Processing

The following predicates are inspired from the `lists` library of SICStus Prolog. The following predicates are available:

`append(?Prefix,?Suffix,?Combined)`
: is true when `Combined` is the combined list of the elements in `Prefix` followed by the elements in `Suffix`. It can be used to form `Combined` or it can be used to find `Prefix` and/or `Suffix` from a given `Combined`.

`member(?Element,?List)`
: is true when `Element` is a member of `List`. It may be used to test for membership in a list, but it can also be used to enumerate all the elements in `List`.

`memberchk(+Element,+List)`
: is true when `Element` is a member of `List`, but `memberchk/2` only succeeds once and can therefore not be used to enumerate the elements in `List`.

`reverse(?List,?Reversed)`
: is true when `Reversed` has the same elements as `List` but in a reversed order.

`delete(+List,+Element,?Residue)`
: is true when `Residue` is the result of removing all identical occurrences of `Element` in `List`.

`select(?Element,?List,?List2)`
: is true when the result of removing an occurrence of `Element` in `List` is `List2`.

`permutation(?List,?Perm)`
: is true when `Perm` is a permutation of `List`.

`prefix(?Prefix,?List)`
: is true when `Prefix` is a prefix of `List`.

`suffix(?Suffix,?List)`

is true when `Suffix` is a suffix of `List`.

`sublist(?Sub,?List)`

is true when `Sub` contains some of the elements of `List`.

`last(?List,?Last)`

is true when `Last` is the last element in `List`.

`length(?List,?Length)`

If `List` is instantiated to a list of determinate length, then `Length` will be unified with this length. If `List` is of indeterminate length and `Length` is instantiated to an integer, then `List` will be unified with a list of length `Length`. The list elements are unique variables. If `Length` is unbound then `Length` will be unified with all possible lengths of `List`.

`nth(?N,?List,?Element)`

`nth/3` is true when `Element` is the `N`th element of `List`. The first element is number 1.

`max_list(+ListOfNumbers,?Max)`

is true when `Max` is the largest of the elements in `ListOfNumbers`.

`min_list(+ListOfNumbers,?Min)`

is true when `Min` is the smallest of the numbers in the list `ListOfNumbers`.

`sum_list(+ListOfNumbers,?Sum)`

is true when `Sum` is the result of adding the `ListOfNumbers` together.

I.3.7 Operators

`op(+Precedence,+Type,+Name)`

Declares the atom `Name` to be an operator of the stated `Type` and `Precedence`. `Name` may also be a list of atoms in which case all of them are declared to be operators. If `Precedence` is 0 then the operator properties of `Name` (if any) are cancelled.

I.3.8 Modification of the Program

`wamcc` allows for dynamic code, i.e. code which can be considered as data. This allows the user to add and retract clauses dynamically and to consult programs which is very useful when debugging since this avoid to recompile the code. A *dynamic predicate* is either a predicate which has been declared with the directive `:- dynamic` or a predicate whose first clause has been dynamically created (*asserted*). In this version dynamic code is always public (can be seen by any module). Note that a *static predicate* (i.e. compiled predicate) cannot be redefined by a dynamic predicate.

For the predicates defined below, the argument `Head` must be instantiated to an atom or a compound term. The argument `Clause` must be instantiated either to a term `Head :- Body` or, if the body part is empty, to `Head`. An empty body part is represented as `true`.

`asserta(+Clause)`

The current instance of `Clause` is interpreted as a clause and is added to the current interpreted program as the first clause. The predicate concerned must be currently be dynamic or undefined. Any uninstantiated variables in the `Clause` will be replaced by new private variables.

`assertz(+Clause)`

Like `asserta/2`, except that the new clause becomes the last clause for the predicate concerned.

`clause(+Head,?Body)`

The clause `Head :- Body` exists in the current interpreted program. The predicate concerned must currently be dynamic. `clause/2` may be used in a non-determinate fashion, i.e. it will successively find clauses matching the argument through backtracking.

`retract(+Clause)`

The first clause in the current interpreted program that matches `Clause` is erased. The predicate concerned must currently be dynamic. `retract/1` may be used in a non-determinate fashion, i.e. it will successively retract clauses matching the argument through backtracking.

`reinit_predicate(+Name/+Arity)`

Erase all clauses of the predicate specified by `Name/Arity`. The predicate definition is retained.

`abolish(+Name/+Arity)`

Erase all clauses of the predicate specified by `Name/Arity`. The predicate definition is also erased.

`listing(+Name/+Arity)`

Lists the interpreted predicate(s) specified by `Name/Arity` Any variables in the listed clauses are internally bound to ground terms before printing.

`consult(+Files)`

Consults source files. `Files` is either the name of a file or a list of filenames. Note that it is possible to call `consult(user)` and then enter a clauses directly on the terminal (ending with `^D`).

When a directive is read it is immediately executed. Any predicate defined in the files erases any clauses for that predicate already present in the interpreter. Recall that a static predicate cannot be redefined by an interpreted predicate.

`[File|Files]`

shorthand for `consult([File|Files])`.

I.3.9 All Solutions

When there are many solutions to a problem, and when all those solutions are required to be collected together, this can be achieved by repeatedly backtracking and gradually building up a list of the solutions. The following built-in predicates are provided to automate this process.

`setof(?Template,+Goal,?Set)`

Read this as "`Set` is the set of all instances of `Template` such that `Goal` is satisfied, where that set is non-empty". The term `Goal` specifies a goal or goals as in

`call(Goal)`. `Set` is a set of terms represented as a list of those terms, without duplicates, in the standard order for terms (see section I.3.3). If there are no instances of `Template` such that `Goal` is satisfied then the predicate fails.

The variables appearing in the term `Template` should not appear anywhere else in the clause except within the term `Goal`. Obviously, the set to be enumerated should be finite, and should be enumerable by Prolog in finite time. It is possible for the provable instances to contain variables, but in this case the list `Set` will only provide an imperfect representation of what is in reality an infinite set.

If there are uninstantiated variables in `Goal` which do not also appear in `Template`, then a call to this built-in predicate may backtrack, generating alternative values for `Set` corresponding to different instantiations of the free variables of `Goal`. (It is to cater for such usage that the set `Set` is constrained to be non-empty.) Two instantiations are different iff no renaming of variables can make them literally identical.

Variables occurring in `Goal` will not be treated as free if they are explicitly bound within `Goal` by an existential quantifier. An existential quantification is written `Y^Q` meaning "there exists a `Y` such that `Q` is true", where `Y` is some Prolog variable.

`bagof(?Template,+Goal,?Bag)`

This is exactly the same as `setof/3` except that the list (or alternative lists) returned will not be ordered, and may contain duplicates. The effect of this relaxation is to save a call to `sort/2`, which is invoked by `setof/3` to return an ordered list.

`findall(?Template,+Goal,?Bag)`

`Bag` is a list of instances of `Template` in all proofs of `Goal` found by Prolog. The order of the list corresponds to the order in which the proofs are found. The list may be empty and all variables are taken as being existentially quantified. This means that each invocation of `findall/3` succeeds *exactly once*, and that no variables in `Goal` get bound. Avoiding the management of universally quantified variables can save considerable time and space.

`findall(?Template,+Goal,?Bag,?Remainder)`

Same as `findall/3`, except that `Bag` is the list of solution instances appended to `Remainder`, which is typically unbound.

I.3.10 Global Variables

wamcc provides a simple and powerful way to assign and read global variables. There are 3 kinds of objects which can be associated to a global variable:

- a copy of a term,
- a link to a term,
- an array of objects.

The initial value of a global variable is the integer 0. A global variable is referenced by a name (i.e. name = an atom) possibly indiced if it is an array (i.e. name = a compound term). In the following, GVarName represents such a reference to a global variable and its syntax is as follows:

```
GVarName ::=  atom
              atom(Index,...,Index)

Index    ::=  integer
              GVarName
```

When a GVarName is used as an index, the value of this variable must be an integer. Indexes range from 0 to Size-1 if the array has Size elements.

The space necessary for copies and arrays are allocated dynamically and recovered as soon as possible. For instance, when an atom is associated to a global variable whose current value is an array, the space for this array is recovered (unless if the assignment must be undone when backtracking occurs, see below).

`g_assign(+GVarName,+Term)` (*inline predicate*)

Associates a copy of the term Term to GVarName. This assignment is not undone when backtracking occurs. See below about array (re)definitions.

`g_assignb(+GVarName,+Term)` (*inline predicate*)

Same as g_assign/2 but the assignment is undone when backtraking occurs. See below about array (re)definitions.

`g_link(+GVarName,+Term)` (*inline predicate*)

Makes a link between `GVarName` to the term `Term`. This allows the user to give a name to any Prolog term (in particular non-ground terms). Note that such an assignment is always undone when backtracking occurs (since the term could no longer exists). Note also that if `Term` is an atom or an integer, `g_link` and `g_assignb` have the same behavior. Finally: `g_link` is not expansive at all neither for the memory nor for the execution time point of views. See below about array (re)definitions.

`g_read(+GVarName,?Term)` (*inline predicate*)

Unifies `Term` with the associated value of `GVarName`. See below about array readings.

`g_array_size(+GVarName,?Size)` (*inline predicate*)

Unifies `Size` with the dimension (an integer > 0) of the array stored by `GVarName`. Fails if `GVarName` is not an array.

The predicates `g_assign/2`, `g_assignb/2` and `g_link/2` define an array when `Term` is a compound term with principal functor `g_array/1-2`. Then an array is assigned to `GVarName` (backtrackable assignment or not depending on the predicate used). There are 3 forms for the term `g_array/1-2`:

`g_array(+Size)`

if `Size` is an integer > 0 then defines an array of `Size` elements which are all initialized with the integer 0 w.r.t to the predicate used (`g_assign/2`, `g_assignb/2` or `g_link/2`).

`g_array(+Size,+Term)`

As above but the elements are initialized with the term `Term` instead of 0. Note that `Term` can contain other array definitions allowing thus for multi-dimensional arrays.

`g_array(+ListOfElements)`

As above if `ListOfElements` is a list of length `Size` except that the elements of the array are initialized w.r.t the elements of the list `ListOfElements` (which can contain other array definitions).

The compound term with principal functor `g_array_extend/1-2` can can be used similarly to `g_array/1-2` except that is does not initialize existing elements of the (possibly) previous

array.

When an array is read, a term of the form `g_array([`$\texttt{Elem}_0$`,...,`$\texttt{Elem}_{Size-1}$`])` is returned.

Example: a simple counter:

```
| ?- [user].
inc(Var,Value):- g_read(Var,Value), X is Value+1, g_assign(Var,X).
^D
yes

| ?- inc(a,X).

X = 0
yes
| ?- inc(a,X).

X = 1
yes
```

Example: `g_assign` does not undo its assignment at backtracking whereas `g_assignb` undoes it.

```
| ?-  g_assign(x,1),
      (   g_read(x,Old), g_assign(x,2)
      ;   g_read(x,Old), g_assign(x,3)
      ).

Old = 1 ? ;

Old = 2       % the value 2 has not been undone
yes

| ?-  g_assign(x,1),
      (   g_read(x,Old), g_assignb(x,2)
      ;   g_read(x,Old), g_assignb(x,3)
      ).

Old = 1 ? ;

Old = 1       % the value 2 has been undone
yes
```

Example: `g_assign` and `g_assignb` create a copy of the term whereas `g_link` does not. `g_link` can often avoid to pass big data structures (e.g. dictionnaries,...) as arguments of many predicates.

```
| ?- g_assign(foo,f(X)), X=12, g_read(foo,Y).

X = 12
Y = f(_60)
yes

| ?- g_link(foo,f(X)), X=12, g_read(foo,Y).

X = 12
Y = f(12)
yes
```

Example: array definitions.

```
| ?- g_assign(w,g_array(3)), g_read(w,X).

X = g_array([0,0,0])

| ?- g_assign(w(0),16), g_assign(w(1),32), g_assign(w(2),64),
     g_read(w,X).

X = g_array([16,32,64])
yes

| ?- g_assign(k,g_array([16,32,64])), g_read(k,X).      % simpler

X = g_array([16,32,64])
yes

| ?- g_assign(k,g_array(3,null)), g_read(k,X).

X = g_array([null,null,null])
yes
```

Example: array extension.

```
| ?- g_assign(a,g_array([10,20,30])), g_read(a,X).

X = g_array([10,20,30])
```

```
yes
| ?- g_assign(a,g_array_extend(5,null)), g_read(a,X).

X = g_array([10,20,30,null,null])
yes
| ?- g_assign(a,g_array([10,20,30])), g_read(a,X).

X = g_array([10,20,30])
yes
| ?- g_assign(a,g_array_extend([1,2,3,4,5,6])),  g_read(a,X).

X = g_array([10,20,30,4,5,6])
yes
```

Example: 2-D array definition.

```
| ?- g_assign(w,g_array(2,g_array(3))), g_read(w,X).

X = g_array([g_array([0,0,0]),g_array([0,0,0])])
yes
| ?- (   for(I,0,1), for(J,0,2), K is I*3+J, g_assign(w(I,J),K),
         fail
     ;   g_read(w,X)
     ).

X = g_array([g_array([0,1,2]),g_array([3,4,5])])
yes

| ?- g_read(w(1),X).

X = g_array([3,4,5])
yes
```

Example: hybrid array.

```
| ?- g_assign(w,g_array([1,2,g_array([a,b,c]),g_array(2,z),5])),
     g_read(w,X).

X = g_array([1,2,g_array([a,b,c]),g_array([z,z]),5])
yes
| ?- g_read(w(1),X), g_read(w(2,1),Y), g_read(w(3,1),Z).

X = 2
Y = b
```

```
Z = z

yes
| ?- g_read(w(1,2),X).

Error: Illegal array index <2> for <w>
```

I.3.11 Miscellaneous

`pragma_c(+Code)` (*inline predicate*)

Inserts at compile-time the C code `Code` in the resulting C file. The use of this predicates requires a good knowledge of the underlying Prolog engine[6].

`statistics`

Displays on the terminal statistics relating to memory usage and run time.

`statistics(?Key,?Value)`

This allows a program to gather various execution statistics. For each of the possible keys `Key`, `Value` is unified with a list of values, as follows:

stack_name	`[Size_used,Size_free]` in bytes stack_name : `local`, `global`, `trail`
runtime	`[Since_start,Since_last]` in ms
systime	`[Since_start,Since_last]` in ms

`cputime(?T)`

similar to `statistics(runtime,[T|_])`.

`version`

Displays the introductory banner.

`top_level(+BannerBool,+CatchBool)`

invokes a (sub) top level. `BannerBool` is `true` or `false` and indicates if the introductory banner must be displayed at the invocation. `CatchBool` indicates if the exceptions raised by `throw/1` which are not captured must be captured by top level (which simply write the `Ball` sent by `throw/1` between curly brackets). If `CatchBool` is false

[6] do not hesitate to contact the author for more low-level information.

then the (sub) top level ends, the exception is still raised and should be captured by an other handler. The global variable `'$top_level'` counts the number of nested top levels actually alive (can be used to test if a top level is active).

`gensym(?X)`

`gensym(+Prefix,+X)`

Generates an atom of the form `Prefix`k where `k` is the current value of the global variable `Prefix` (so it must be an integer). This value is then incremented for the next call to `gensym`. if `Prefix` is omitted then `'$sym'` is used.

`compiler_name(?X)`

Unifies `X` with the name of the compiler.

`wam_version(?X)`

Unifies `X` with the current version.

`wam_year(?X)`

Unifies `X` with the year of the current version.

`argc(?X)`

Unifies `X` with the number of Unix arguments (of the command-line).

`argv(+N,X)`

Unifies `X` with the `N`th Unix argument (starting at 0). (see also `Unix(argv(L))`).

`Unix(+Term)`

Allows certain interactions with the operating system. Under Unix the possible forms of `Term` are as follows:

`access(+Path,+Mode)`

Tests if `Mode` is the accessability of `Path` as in the C-function `access(2)`.

`argv(?Args)`

`Args` is unified with a list of atoms of the program arguments.

`cd`

Changes the current working directory to the home directory.

`cd(+Path)`

Changes the current working directory to `Path`.

`exit(+Status)`

Terminates the Prolog process with the status `Status`. (Equivalent to `halt/1`).

`getenv(+Name,?Value)`

Unifies `Value` with the value of the environment variable `Name`.

`shell`

Starts a new interactive Unix shell named in the Unix environment variable `SHELL`. The control is returned to Prolog upon termination of the shell.

`shell(+Command)`

Passes `Command` to a new Unix shell named in the Unix environment variable `SHELL` for execution.

`shell(+Command,?Status)`

Passes `Command` to a new Unix shell named in the Unix environment variable `SHELL` for execution. Unifies `Status` with the returned status of `Command`.

`system(+Command)`

Passes `Command` to a new Unix `sh` process for execution.

`system(+Command,?Status)`

Passes `Command` to a new Unix `sh` process for execution. Unifies `Status` with the returned status of `Command`.

I.4 Debugger

The debugger is based on the *procedure box* model as described in Chapter eight of *Programming in Prolog* by W.F. Clocksin and C.S. Mellish (Springer-Verlag, 1981) which is recommended as an introduction. The proposed debug options are similar to those provided by SICStus or Quintus. The debugger can be used for interpreted code (i.e. dynamic predicates) or for compiled code (i.e. static predicates) compiled with the `-dbg` option. The wam debugger is only available for static code compiled with the `-dbg2` option. The basic built-in predicates to control the debugger are as follows

`debug`

Switches the debugger on (showing spy-points).

`nodebug`

Switches the debugger off.

`debugging`

Prints onto the terminal information about the current debugging state.

`trace`

Switches the debugger on (showing everything).

`leash(+Mode)`

Leashing Mode is set to `Mode`. `Mode` is a list whose elements can be `call`, `exit`, `fail` or `redo`.

`notrace`

Equivalent to `nodebug/0`.

`spy +Name/+Arity`

Sets a spy-point on the predicate whose principal functor is `Name` and arity is `Arity`.

`nospy +Name/+Arity`

Removes the spy-point from the predicate whose principal functor is `Name` and arity is `Arity`.

`nospyall`

Removes all spy-points that have been set.

During the debugging, the trace messages look like:

```
S I J Port: Goal ?
```

where `S` is a *spy-point indicator* and is `+` if there is a spy-point on the predicate `Goal` (or else `S` is `' '`). `N` is an *invocation number*. This unique number can be used to cross correlate the trace messages for the various ports, since it is unique for every invocation. `M` is an `indice number` which represents the number of direct *ancestors* this goal has. `Port` specifies the

particular port (`call`, `exit`, `fail`, `redo`). `Goal` is the current goal. The `?` indicates that you should type in one of the following debug commands[7]:

`creep` **or** `RET`

Single-steps to the very next port.

`skip <n>`

Skips over the execution of predicates called by the current goal. If you specify an invocation number (less or greater than the current one) then the execution continues until the goal whose indice number is `n` is reached.

`leap`

Resumes running your program, only stopping when a spy-point is reached.

`abort`

Aborts the current execution.

`goals` **or** `goalsb`

Prints the list of ancestors to the current goal. `goalsb` also prints the remainding choice-points.

`leash <l>...`

Sets the leashing mode to $\texttt{l}_1$ ... $\texttt{l}_k$ where each $\texttt{l}_i$ is `call`, `exit`, `fail` or `redo` (similar to `leash/1`).

`nobebug` **or** `notrace`

Switches the debugger off.

`=`

Prints onto the terminal information about the current debugging state (similar to `debugging/0`).

`+ <pred/arity>`

Sets a spy-point on the current goal or on the goal whose principal functor is `pred` and arity `arity` (similar to `spy/1`).

[7] only the first character(s) of the commands are required, and `< x >` denotes an optional element.

`- <pred/arity>`

Removes the spy-point from the current goal or from the goal whose principal functor is `pred` and arity `arity` (similar to `nospy/1`).

`< <n>`

Sets the *printdepth* limit to `n` or resets it to `10` if `n` is not specified.

`exact`

In this mode all failures are traced.

`noexact`

In this mode failures occuring when unifying the head are not traced (like in SICStus/Quitus).

`help`

Displays a summarize of the options displayed above.

There are also some low-level (i.e. WAM level) commands (only available for static code compiled with the `-dbg2` option):

`write adr <n>`

Uses `write/1` to print `n` Prolog terms starting at `adr`.

`data adr <n>`

Displays (dump) `n` words starting at `adr`.

`modify adr <n>`

Displays (dump) and makes it possible to modify `n` words starting at `adr`.

`where sadr`

Displays the real address corresponding to `sadr`.

`deref adr`

Displays the dereferenced word located at `adr`.

`envir <sadr>`

Diplays the current environment or the one located at `sadr`.

`backtrack <sadr>`

Diplays the current choice point or the one located at `sadr`.

An address (`adr`) has the following syntax: `bank_name < [n] >`. A stack address has the following syntax: `stack_name < [n] >`. `bank_name` can be one of the following name and `n` is an optional offset specifier (integer):

`bank_name ::=`	`reg`	WAM general registers
	`x`	WAM temporaries
	`y`	current permanent variables
	`stack_name`	a stack
`stack_name::=`	`local`	local stack
	`global`	global stack
	`trail`	trail stack

Annexe J

Manuel d'utilisation de clp(FD)

clp(FD) 2.21 User's Manual

Daniel Diaz

INRIA-Rocquencourt
Domaine de Voluceau
78153 Le Chesnay
FRANCE

Daniel.Diaz@inria.fr

July 1994

This manual only concerns the finite domain constraint facilities. Refer to wamcc *User's manual* for information about the underlying Prolog engine.

J.1 Introduction

clp(FD) is based on the wamcc Prolog compiler and extends it with Finite Domain (FD) constraints. It is recommended to read the wamcc *User's Manual* [21] which explains how to use the underlying Prolog language (built-in predicates, compilation process,...). In this manual some executable names have changed for clp(FD). The folowing table shows the correspondance between old (i.e. wamcc) names and new ones (i.e. clp(FD)):

Executable	wamcc names	clp(FD) names
Prolog compiler	wamcc	clp_fd
Gcc compiler	w_gcc	fd_gcc
Build Makefile	bmf_wamcc	bmf_clp_fd
Library	libwamcc.a	libclp_fd.a
Profile library	libwamcc_pp.a	libclp_fd_pp.a

Some papers [24, 25, 26, 27, 28, 17] presents clp(FD) and its extensions. Basically, clp(FD) deals with only one basic constraint X `in` r (see section J.3.1). X is a finite domain variable and r denotes a *range*, which can be not only a *constant range*, e.g. 1..10 but also an *indexical range* using:

- $min(Y)$ which represents the minimal value of Y (in the current store),
- $max(Y)$ which represents the maximal value of Y,
- $val(Y)$ which represents the definitive value of Y,
- $dom(Y)$ which represents the whole domain of Y.

From the basic X `in` r constraints, it is possible to define high-level constraints, called *user constraints*, as Prolog predicates. Each constraint specifies how the *constrained variable* must be updated when the domains of other variables change. In the clp(FD) system, basic user constraints are already defined as built-in predicates (see section J.3). CHIP-like constraints such as equations, inequations and disequations can be used directly by the programmer. A preprocessor will translate them at compile time. So, clp(FD) offers the usual constraints over finite domains as proposed by CHIP together with the possibility to define new constraints in a declarative way.

J.2 Finite Domain variables

A new type of data is introduced: *FD variables* which can take a value in its domain (reduced step by step by X `in` r constraints). An FD variable is fully compatible with Prolog integers and Prolog variables. Namely, each time a FD variable is expected in a constraint (i.e. in X `in` r and other user constraints) it is possible to pass a Prolog integer (treated as a singleton range) or a Prolog variable (bound to an initial range $0..infinity$). Since domains are *finite*, $infinity$ stands for the greatest integer (see also `fd_infinity/1`). Since Prolog variables and FD variables are fully compatible, no domain declarations are needed. However, in some cases this can cause a failure due to overflows (e.g. $infinity \times infinity$). In particular intermediate variables do not need any domain declaration as it is required in CHIP.

There are 2 representations for an FD variable:

- **interval representation**: only the *min* and the *max* of the variable are maintained. In this representation it is possible to store values included in $0..infinity$.
- **sparse representation**: an additional bit-vector is used to store the set of possible values for a variable. In this representation it is possible to store values included in $0..vector_max$. By default $vector_max$ is set to 127 and can be redefined via an environment variable `VECTORMAX` or via the built-in predicate `fd_vector_max/1` (see section J.3).

The initial representation for an FD variable X is always an interval representation and is switched to a sparse representation when a "hole" appears in the domain (e.g. due to union, complementation,...)[1]. When this switching occurs some values can be lost since $vector_max$ is less than $infinity$. We say that "X is extra constrained" since X is constrained by the solver to the domain $0..vector_max$. A flag *extra_cstr* is associated to each FD variable to indicate if some values have been lost and is updated by all operations. An "extra constrained" FD variable is written followed by the `@` symbol. When a failure occurs on a variable extra constrained a message `Warning: Vector too small - maybe lost solutions` is displayed.

[1] As soon as a variable uses a sparse representation it will not switch back to an interval representation even if there are no longer holes in its domain.

Example ($vector_max = 127$):

Constraint on X	Domain of X	Lost values	Extra Cstr Flag
`X in 0..512`	`0..512`	$\emptyset$	`off`
`X in 0..3:10..512`	`0..3:10..127`	`128..512`	`on`
`X in 0..100`	`0..3:10..100`	$\emptyset$	`off`

In this example, when the constraint X *in* 0..3 : 10..512 is told some solutions are lost. However, when constraint X *in* 0..100 is told, no longer values are lost.

Other example:

Constraint on X	Domain of X	Lost values	Extra Cstr Flag
`X in 0..512`	`0..512`	$\emptyset$	`off`
`X in 0..3:10..512`	`0..3:10..127`	`128..512`	`on`
`X in 256..300`	$\emptyset$	*Warning ...*	`on`

In this example, the constraint X *in* 256..300 fails due to the lost of 128..512 so a message is displayed onto the terminal. The solution would consist in defining:

```
%setenv VECTORMAX 512
```

Finally, note that bit-vectors are not *dynamic*, i.e. all vectors must have the same size (0..*vector_max*). So the use of `fd_vector_max/1` is limited to the initial definition of vector sizes and must occur before any constraint.

J.3 Finite Domain built-in predicates / constraints

J.3.1 The constraint X `in` r

`?X in +R`

enforces `X` to belong to one element of the range denoted by `R`. The syntax of X `in` r is given by the following table:

c ::=	X `in` r	(constraint)
r ::=	t_1`..`t_2	(interval range)
	$\{t\}$	(singleton range)
	R	(range parameter)
	`dom(`Y`)`	(indexical domain)
	r_1 `:` r_2	(union)
	r_1 `&` r_2	(intersection)
	`-`r	(complementation)
	r `+` ct	(range by term addition)
	r `-` ct	(range by term subtraction)
	r `*` ct	(range by term multiplication)
	r `/` ct	(range by term division)
	r `mod` ct	(range by term modulo)
	r_1 `+` r_2	(range by range addition)
	r_1 `-` r_2	(range by range subtraction)
	r_1 `*` r_2	(range by range multiplication)
	r_1 `/` r_2	(range by range division)
	r_1 `mod` r_2	(range by range modulo)
	f_r`(`a_1`,...,`a_k`)`	(user range function)
a ::=	r | t	(user function argument)
t ::=	`min(`Y`)`	(indexical min)
	`max(`Y`)`	(indexical max)
	`val(`Y`)`	(delayed value)
	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`)`	(user term function)
ct ::=	C	(term parameter)
	n | `infinity` | ct_1`+`ct_2 | ct_1`-`ct_2 | ct_1`*`ct_2 | ct_1`/<`ct_2 | ct_1`/>`ct_2	

J.3.2 Linear arithmetic constraints

A linear term is of the for $A_1 * X_1 + -... + -A_n * X_n$ where each A_i must be an integer and can be omitted if it is 1. Each X_1 is a variable, an FD variable or an integer (i.e. a constant). $+-$ denotes either the plus symbol + or the minus symbol $-$. In the following `S` and `T` are linear terms.

`S#=T`

`S` is equal to `T`.

`S#\=T`

`S` is not equal to `T`.

`S#<T`

`S` is less than `T`.

`S#<=T`

`S` is less than or equal to `T`.

`S#>T`

`S` is greater than `T`.

`S#>=T`

`S` is greater than or equal to `T`.

J.3.3 Other arithmetic constraints

`'min(x,y)=z'(?X,?Y,?Z)`

`Z` is the minimum value between `X` and `Y`.

`'max(x,y)=z'(?X,?Y,?Z)`

`Z` is the maximum value between `X` and `Y`.

`'|x-y|=z'(?X,?Y,?Z)`

`Z` is the absolute value of `X-Y`.

`'xx=y'(?X,?Y)`

`Y` is the square of `X`.

`'xy=z'(?X,?Y,?Z)`

`Z` is equal to `X*Y` (i.e. non-linear equation).

J.3.4 Domain Handling

`domain(+Vars,+Lower,+Upper)`

constraints each variable of the list `Vars` to belong to the domain `Lower..Upper`.

`fd_vector_max(?N)`

if `N` is a variable, unifies `N` to the maximum value which can be stored in a bit-vector representation. If `N` is an integer, define the maximum value (such a definition can only be done once *before* stating any constraint (see section J.2).

`fd_infinity(?N)`

Unifies `N` with the $infinity$ (i.e. greatest) value.

`fd_var(?X)`

Succeeds if `X` is bound to an FD variable.

`fd_min(?X,?N)`

Unifies `N` with the current minimum value of the FD variable `X`. Note that this is not a constraint.

`fd_max(?X,?N)`

Unifies `N` with the current maximum value of the FD variable `X`.

`fd_dom(?X,?L)`

Unifies `L` with the current domain of the FD variable `X`. `L` is a list of integers.

`fd_size(?X,?N)`

Unifies `N` with the current size of the domain of the FD variable `X`.

`fd_extra_cstr(?X,?F)`

Unifies `F` with the current *extra constraint flag* (i.e. 0/1) of the FD variable `X` (see section J.2).

`fd_has_vector(?X)`

Succeeds if `X` is an FD variable (not an integer) which uses a bit-vector representation.

`fd_use_vector(?X)`

Enforces the FD variable `X` to use a bit-vector representation.

J.3.5 Enumeration predicates

`indomain(?X)`

assignes a consistent value to the FD variable `X` from the minimum of `X` (trough backtracking, all possible values can be enumerated).

`labeling(+L)`

assignes a value for each FD variable of the list `L` using `indomain/1`.

`labelingff(+L)`

assignes a value for each FD variable of the list `L` using the *first-fail* heuristics.

`deleteff(?X,+L,?Rest)`

Unifies `X` with the FD variable with the smallest domain among the FD variables in the list `L`. Also unifies `Rest` with the list of remainding FD variables except `X`.

J.3.6 Symbolic constraints

`alldifferent(+L)`

enforces the constraint $X \neq Y$, for each pair of variable `X,Y` in the list of FD variables `L`.

`element(?I,+L,?V)`

the `I`th element of the list of integers `L` must be equal to the value `V`. `I` and `V` are FD variables.

`atmost(+N,+L,+V)`

at most `N` variables of the list of FD variables `L` are equal to the value `V`. `N` and `V` are integers.

`relation(+Tuples,+Vars)`

enforces the list of FD variables `Vars` to verify the relation coded by `Tuples`. `Tuples` is a list of tuples of the relation, each tuple is a list of integers. Example:
`and(X,Y,Z):- relation([[0,0,0],[0,1,0],[1,0,0],[1,1,1]],[X,Y,Z]).`

`relationc(+CTuples,+Vars)`

similar to `relation/2` but the tuples are given column by column (faster than

relation/2). Example:

```
and(X,Y,Z):- relationc([[0,0,1,1],[0,1,0,1],[0,0,0,1]],[X,Y,Z]).
```

J.3.7 Symbolic constraints

`minof(+Goal,?Var)`

uses a *depth-first branch and bound* to find the minimum optimal value of `Var` using `Goal` as generator (e.g. `labeling/1`).

`maxof(+Goal,?Var)`

uses a *depth-first branch and bound* to find the maximum optimal value of `Var` using `Goal` as generator (e.g. `labeling/1`).

J.4 Boolean built-in predicates / constraints

`clp(FD)` offers a set of boolean constraints based on Finite Domains. A boolean variable is nothing more than a FD variable with an initial domain 0..1. So arc-consistency is also used for booleans and thus the enumeration phase is required as for traditional FD constraints. No declarations are needed for boolean variables. When a varialble is involved in a boolean constraint it is automatically set to the initial domain 0..1.

J.4.1 Basic boolean constraints

`not(?X,?Y)`

true if $X = \neg Y$.

`and(?X,?Y,?Z)`

true if $Z = X \wedge Y$.

`and0(?X,?Y)`

true if $0 = X \wedge Y$ (i.e. $\neg X \vee \neg Y$).

`and0(?X,?Y,?Z)`

true if $0 = X \wedge Y \wedge Z$ (i.e. $\neg X \vee \neg Y \vee \neg Z$).

`or(?X,?Y,?Z)`

true if $Z = X \vee Y$.

`or1(?X,?Y)`

true if $1 = X \vee Y$ (i.e. $X \vee Y$).

`or1(?X,?Y,?Z)`

true if $1 = X \vee Y \vee Z$ (i.e. $X \vee Y \vee Z$).

`xor(?X,?Y,?Z)`

true if $Z = X \ xor \ Y$.

`equiv(?X,?Y,?Z)`

true if $Z = X \Leftrightarrow Y$.

`equiv1(?X,?Y)`

true if $1 = X \Leftrightarrow Y$ (i.e. $X \Leftrightarrow Y$).

J.4.2 Symbolic boolean constraints

`at_least_one(+L)`

true if at least one boolean variable of the list `L` is equal to 1.

`at_most_one(+L)`

true if at most one boolean variable of the list `L` is equal to 1.

`only_one(+L)`

true if only one boolean variable of the list `L` is equal to 1.

Bibliographie

[1] A. Aggoun and N. Beldiceanu. Time Stamps Techniques for the Trailed Data in CLP Systems. In *Actes du Séminaire 1990 - Programmation en Logique*, Tregastel, France, CNET 1990.

[2] A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In *8th International Conference on Logic Programming*, Paris, France, MIT Press, 1991. Also in *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.

[3] H. Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. Logic Programming Series, MIT Press, 1991.

[4] F. Benhamou. Boolean Algorithms in PrologIII. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.

[5] H. Bennaceur and G. Plateau. FASTLI: An Exact Algorithm for the Constraint Satisfaction Problem: Application to Logical Inference. Research Report, LIPN, Université Paris-Nord, Paris, France, 1991.

[6] H. Bennaceur and G. Plateau. Logical Inference Problem in Variables 0/1. in *IFORS 93 Conference*, Lisboa, Portugal, 1993.

[7] BNR-Prolog User's Manual. Bell Northern Research. Ottawa, Canada, 1988.

[8] A. Bockmayr. Logic Programming with Pseudo-Boolean Constraints. Research Report MPI-I-91-227, Max Planck Institut, Saarbrucken, Germany, 1991.

[9] M. Bonnard, S. Manchon, and P. Planchon. Bilan des études de la division AOC sur la régulation du trafic aérien, 1992.

[10] S. Manchon, D. Chemla, C. Gobier, and P. Kerlirzin. Dossier de spécifications du Système Prétactique pour Optimiser la Régulation du Trafic aérien: SPORT V4.3, 1992.

[11] M. Bruynooghe. An Interpreter for Predicate Logic Programs: Basic Principles. Research Report cw10, Katholieke Universiteit Leuven, Belgium, 1976.

[12] M. Bruynooghe. The Memory Management of Prolog Implemantations. in Workshop'80, pp12-20, 1980.

[13] R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on computers*, no. 35 (8), 1986, pp 677–691.

[14] W. Büttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, no. 4 (1987), pp 191-205.

[15] M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine.* PhD dissertation, SICS, Sweden, 1990.

[16] B. Carlson, M. Carlsson. Constraint Solving and Entailment Algorithms for `cc(FD)`. Research Report, SICS, Sweden, 1993.

[17] B. Carlsson, M. Carlsson, D. Diaz. Entailment of Finite Domain Constraints. In *11th International Conference on Logic Programming*, Santa Margherita, Italy, MIT Press, 1994.

[18] D. Chemla, D. Diaz, P. Kerlirzin and S. Manchon. Using `clp(FD)` to Support Air Traffic Flow Management. In *3rd International Conference on the Practical Application of Prolog*, Paris, France, 1995.

[19] T. Chikayama, T. Fujise and D. Sekita. A portable and Efficient Implementation of KL1. in ICOT/NSF *Workshop on Parallel Logic Programming and its Programming Environments*, CIS-TR-94-04, Department of Computer Information Science, Oregon, 1994.

[20] P. Codognet, F. Fages and T. Sola. A metalevel compiler for CLP(FD) and its combination with intelligent backtracking. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.

[21] D. Diaz. `wamcc` *Prolog Compiler User's Manual.* INRIA, Le Chesnay, France, 1994.

[22] D. Diaz. `clp(FD)` *User's Manual.* INRIA, Le Chesnay, France, 1994.

[23] P. Codognet and D. Diaz. `wamcc`: Compiling Prolog to C. In *12th International Conference on Logic Programming*, Tokyo, Japan, MIT Press, 1995.

[24] P. Codognet and D. Diaz. A Minimal Extension of the WAM for `clp(FD)`. In *10th International Conference on Logic Programming*, Budapest, Hungary, MIT Press, 1993.

[25] P. Codognet and D. Diaz. Compiling Constraint in `clp(FD)`. To appear in *Journal of Logic Programming.*

[26] P. Codognet and D. Diaz. Boolean Constraint Solving Using `clp(FD)`. In *International Logic Programming Symposium*, Vancouver, British Columbia, Canada, MIT Press, 1993.

[27] P. Codognet and D. Diaz. `clp(B)`: Combining Simplicity and Efficiency in Boolean Constraint Solving. In *Programming Language Implementation and Logic Programming* Madrid, Spain, Springer-Verlag, 1994.

[28] P. Codognet and D. Diaz. A Simple and Efficient Boolean Solver for Constraint Logic Programming. To appear in *Journal of Automated Reasoning.*

[29] P. Codognet and D. Diaz. Finite Domain Constraints in Constraint Logic Programming. In *14th European Conference on Operational Research*, Jerusalem, Israel, 1995.

[30] D. Colin de Verdière. Utilisation des techniques de recherche opérationnelle pour les études Air Traffic Management, 1992.

[31] A. Colmerauer. An introduction to Prolog-III. *Communications of the ACM*, 33 (7), July 1990.

[32] Vítor Santos Costa, D. H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the basic Andorra model. In *8th International Conference on Logic Programming*, Paris, France, MIT Press, 1991.

[33] M. Dincbas, H. Simonis and P. Van Hentenryck. Solving the Car-Sequencing Problem in Constraint Logic Programming. In *ECAI-88*, Munich, W. Germany, August 1988.

[34] M. Dincbas, H. Simonis and P. Van Hentenryck. Solving large combinatorial problems in Logic Programming. *Journal of Logic Programming*, 8 (1,2), 1990.

[35] G. Dore and P. Codognet. A Prototype Compiler for Prolog with Boolean Constraints. In *GULP'93, Italian Conference on Logic Programming*, Gizzeria Lido, Italy, 1993.

[36] G. Gallo, G. Urbani, Algorithms for Testing the Satisfiability of Propositional Formulae. *Journal of Logic Programming*, no. 7 (1989), pp 45-61.

[37] J.M. Garot. Airspace Management in Europe: issues and solutions. In *IFORS 1993: 13th International Conference of Operational Research*, Lisbon, Portugal, 1993.

[38] D. Gudeman. Representing Type Information in Dynamically Typed Languages. Technical Report, University of Arizona, Arizona, 1993.

[39] D. Gudeman, K. De Bosschere and S. Debray. `jc`: An Efficient and Portable Sequential Implementation of Janus. In *Joint International Conference and Symposium on Logic Programming*, Washington, MIT Press, 1992.

[40] R. M. Haralick and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14 (1980)*, pp 263-313

[41] S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In *7th International Conference on Logic Programming*, Jerusalem, Israel, MIT Press, 1990.

[42] B. Haussman. Turbo Erlang: Approaching the Speed of C. In *Implementations of Logic Programming Systems*, Evan Tick (ed.), Kluwer 1994.

[43] J. N. Hooker and C. Fedjki. Branch-and-Cut Solution of Inference Problems in Propositional Logic. Research Report, Carnegie-Mellon University Pittsurh, Pennsylvania, 1987.

[44] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Principles Of Programming Languages*, Munich, Germany, January 1987.

[45] J. Jaffar and S. Michaylov. A Methodology for Managing Hard Constraints in CLP Systems. In *proceedings of Sigplan PLDI*, Toronto, Canada, ACM Press 1991.

[46] J. Jaffar, S. Michaylov, P. J. Stuckey and R. Yap. An Abstract Machine for CLP($\mathcal{R}$). In *proceedings of Sigplan PLDI*, San Francisco, ACM Press 1992.

[47] J. Jourdan. Modelisation of terminal zone aircraft sequencing in constraint logic programming, 1992.

[48] J. Jourdan and T. Sola. The Versatility of Handling Disjunctions as Constraints In *Programming Language Implementation and Logic Programming*, Talin, Estonia, 1993.

[49] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence 8 (1977)*, pp 99-118.

[50] U. Martin and T. Nipkow. Boolean Unification – The story so far. *Journal of Symbolic Computation*, no. 7 (1989), pp 191-205.

[51] J-L. Massat. Using Local Consistency Techniques to Solve Boolean Constraints. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.

[52] C.S. Mellish. An alternative to Structure-Sharing in Logic Programming edited by K.L. Clark and S.A. Tarnlund, Academic Press, pp99-106, 1982.

[53] B. A. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence 5 (1989)*, pp 188-224.

[54] W.J. Older and F. Benhamou. Programming in clp(BNR). In *Position Papers of 1st PPCP*, Newport, Rhode Island, 1993.

[55] L. M. Pereira and A. Porto. Intelligent backtracking and sidetracking in horn clause programs. Technical Report CIUNL 2/79, Universitade Nova de Lisboa, 1979.

[56] A. Rauzy. *L'Evaluation Sémantique en Calcul Propositionnel.* PhD thesis, Université Aix-Marseille II, Marseille, France, January 1989.

[57] A. Rauzy. Adia. Technical Report, LaBRI, Université Bordeaux I, 1991.

[58] A. Rauzy. Using Enumerative Methods for Boolean Unification. In *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (Eds.). MIT Press, 1993.

[59] A. Rauzy. Some Practical Results on the SAT Problem. Draft, 1993.

[60] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Research Report CMU-CS-89-108, Carnegie-Mellon University, 1989. Also (revised) MIT Press, 1993.

[61] V. A. Saraswat. *Concurrent Constraint Programming*, MIT Press, 1993.

[62] T. Solla. PhD thesis (forthcoming), THOMSON-CSF, France, 1994.

[63] V. Saraswat. The Category of Constraint Systems is Cartesian-Closed. In *Logic In Computer Science*, IEEE Press 1992.

[64] D. S. Scott. Domains for Denotational Semantics. In *ICALP'82, International Colloquium on Automata Languages and Programming*, 1982.

[65] H. Simonis, M. Dincbas. Propositional Calculus Problems in CHIP. ECRC, Technical Report TR-LP-48, 1990.

[66] T. E. Uribe and M. E. Stickel. Ordered Binary Decision Diagrams and the Davis-Putnam Procedure. In *CCL'94, Constraints in Computational Logics*, Munich, Germany, Springer-Verlag, 1994.

[67] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series, MIT Press, 1989.

[68] P. Van Hentenryck and Y. Deville. The Cardinality Operator: A new Logical Connective for Constraint Logic Programming. In *8th International Conference on Logic Programming*, Paris, France, MIT Press, 1991.

[69] P. Van Hentenryck, V. Saraswat and Y. Deville. Constraint processing in cc(FD). Draft, 1991.

[70] P. Van Hentenryck, V. Saraswat and Y. Deville. Design, Implementation and Evaluation of the Constraint language cc(FD). Draft, 1993.

[71] P. Van Hentenryck, Y. Deville and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence 57 (1992)*, pp 291-321.

[72] P. Van Hentenryck, H. Simonis and M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artificial Intelligence* no 58, pp 113-159, 1992.

[73] P. Van Roy. Can Logic Programming run as fast as Imperative Programming ? Report No UCB/CSD90/600, Berkeley, California, 1990.

[74] P. Van Roy and A. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. IEEE Computer, pp 54-67, 1992.

[75] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Oct. 1983.

[76] D. H. D. Warren. The Andorra Principle. Internal Report, Gigalips Group, 1987.

Des contraintes naît la beauté

Léonard de Vinci.

L'erreur est la règle ;
la vérité est l'accident de l'erreur.

Georges Duhamel.

Il y a assez de lumière pour ceux qui ne désirent que de voir,
et assez d'obscurité pour ceux qui ont une disposition contraire.

Blaise Pascal.

Résumé

Ce travail porte sur la compilation des langages de programmation logique par contraintes sur les domaines finis (DF). Plutôt que d'adopter l'approche usuelle considérant le résolveur comme une "boîte noire" nous avons choisi l'approche "boîte de verre" de P. van Hentenryck. Dans celle-ci, le résolveur gèrer une seule contrainte primitive. Toutes les contraintes complexes (équations, contraintes symboliques,...) sont traduites en des appels de contraintes primitives. Le résolveur est ainsi simple et homogène. De plus, l'utilisateur peut définir ses propres contraintes en termes de cette primitive. Cette primitive nous permet de définir une machine abstraite pour la compilation des contraintes DF. En outre, le traitement d'une seule primitive permet de définir des optimisations globales dont bénéficient toutes les contraintes de haut niveau. Toutes ces idées sont détaillées et aboutissent à la définition du langage `clp(FD)`. L'étude des performances de `clp(FD)` montre que cette approche est très efficace, meilleure en tout cas que les résolveurs boîtes noires. Nous étudions également les aptitudes de `clp(FD)` à résoudre des contraintes booléennes car elles sont un cas particulier des DF. Là encore `clp(FD)` se compare très bien avec des résolveurs spécialisés. Nous nous intéressons enfin à la détection de la satisfaction des contraintes pour permettre à l'utilisateur de spécifier des calculs dirigés par les données (plutôt que par les instructions). Ce travail débouche donc tout naturellement sur l'implantation des langages concurrents

Mots-clés :

Programmation Logique, Prolog, implantation, contraintes, domaines finis, booléens, langages concurrents, Intelligence Artificielle.