CLP^{++}, Programmation Logique avec Contraintes en C++

Renaud Keriven[1]

Rapport de Recherche CERMICS 92.8
Novembre 1992

[1] Renaud.Keriven@cermics.enpc.fr

Abstract

This paper is the result of some work done about Constraint Logic Programming. After a survey of two languages — ALICE and CHIP — CLP^{++} , a C++ library on finite domains, is described. An object-oriented approach of constraints is proposed. Some new concepts used for our implementation are introduced: *weak operations on finite domains* which lead to a smarter and more general handling of domain reduction, an *extension of the active behaviour of a constraint* and a notion of *strategic behaviour* dealing with some non algebraic constraints such as the *cardinality operator*. Finally, we claim for a fast and powerful language like C++ packaged with a CLP library, rather than a dedicated language like CHIP.

Résumé

Le présent rapport expose les résultats d'un travail sur la Programmation Logique avec Contraintes (CLP). Après un aperçu de deux langages existants — ALICE et CHIP — nous présentons CLP^{++} , une bibliothèque C++ sur les domaines finis. Nous proposons une organisation hiérarchique orientée objet des contraintes. Nous introduisons des concepts originaux : des *opérations faibles sur les domaines finis* permettant un traitement élégant et plus général des réductions de domaines, une *extension du rôle actif des contraintes* et une notion de *rôle stratégique des contraintes* tenant compte des possibilités de contraintes non algébriques telles que l'*opérateur de cardinalité*. Enfin, nous pensons que l'approche de type bibliothèque pour un langage impératif puissant et rapide comme C++ est vraisemblablement préférable à celle d'un langage dédié à la programmation par contraintes.

Introduction

L'objectif initial du travail de stage dont nous présentons ici les résultats était d'appréhender les difficultés soulevées par l'implémentation d'un langage de Programmation Logique avec Contraintes (CLP) sur les domaines finis.

ALICE est avant tout un langage pour poser des problèmes mathématiques. Toutefois la méthode spécifique de résolution décrite par Laurière est riche d'idées sur le traitement des contraintes. Nous y consacrons donc un court chapitre.

L'étude du langage CHIP développé à l'ECRC nous semble un point départ quasi obligé pour l'approche d'un CLP. C'est pourquoi, dans un deuxième chapitre, nous développons les concepts de *generate and test*, *standard-backtracking*, *forward-checking* et *looking-ahead* et détaillons leur incarnation en CHIP. L'exemple classique des huit reines nous servira à comprendre l'utilité et la puissance de ces concepts. Nous terminons le chapitre par une confrontation entre ALICE et CHIP.

Le troisième et dernier chapitre est consacré à CLP^{++}, une maquette de solveur de contraintes sur les domaines finis réalisée en C++. Le développement de CLP^{++} a été l'occasion d'introduire des concepts originaux, notamment ceux d'*opérations faibles entre domaines finis* et de *rôle stratégique d'une contrainte*. Nous comparons CLP^{++} à CHIP, puis, après un rapide exposé de certains détails d'implémentation, nous étudions quelques applications ... avant de conclure.

Chapitre 1

ALICE

Dans ce chapitre, nous présentons ALICE en tant que langage pour poser des problèmes mathématiques. Nous expliquons ensuite ce que la méthode de résolution implémentée par Laurière a de spécifique, et surtout les idées qu'elle apporte en matière traitement des contraintes.

ALICE [Lau78] est un solveur capable de résoudre des problèmes dont l'énoncé est posé en termes mathématiques usuels. Basé sur la théorie des ensembles, la logique du premier ordre et des notions de théorie des graphes, il est plutôt spécialisé dans les problèmes de recherche opérationnelle. De nombreuses techniques développées dans ALICE sont basées sur une manipulation astucieuse des contraintes et sur l'idée de réduction *a priori* de l'arbre de recherche.
Les objets connus d'ALICE sont:

- Les ensembles (finis).
- Les opérateurs sur les ensembles.
- Les produits d'ensembles.
- Les vecteurs et les matrices.
- Les graphes et les chemins dans les graphes.
- Les fonctions.

Les contraintes sur ces objets sont écrites en termes logiques ou algébriques, à l'aide de symboles tels que:

- $+, -, *, /, =, \geq, <$
- $\sum, modulo$
- $\exists, \forall$
- $\in$
- $\Rightarrow, \Leftrightarrow$

- $\cap, \cup, \subset, \supset$
- $\vee, \wedge, \neg$
- bijection, injection, surjection, chemin, circuit, ...
- min, max
- cardinal,...

Voyons comment quelques problèmes usuels s'expriment en ALICE.

1.1 Enoncé des problèmes

1.1.1 Les cryptogrammes arithmétiques

Ce sont les problèmes du type:

Trouver le n-uplet (S, E, N, D, M, O, R, Y) *tel que*

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

chaque lettre étant mise pour un chiffre, deux lettres différentes valant un chiffre différent.

Dans le cas d'une addition, l'énoncé mathématique se fait généralement en introduisant des variables pour les retenues[1]. Ce qui peut se poser de la façon suivante:

Soient:

- $n_c \in \mathbb{N}$, le nombre de lignes.
- $n_l \in \mathbb{N}$, le nombre de colonnes.
- $L = [1, n_l]_{\mathbb{N}}$, l'ensemble des indices de toutes les lignes.
- $L' = [1, n_l - 1]_{\mathbb{N}}$, l'ensemble des indices des lignes sauf la dernière.
- $C = [1, n_c]_{\mathbb{N}}$, l'ensemble des indices des colonnes.
- Let, un ensemble de symboles (les lettres, plus un symbole spécial ζ dont la valeur sera nulle)
- $Chi = [0, 9]_{\mathbb{N}}$, l'ensemble des chiffres.
- $Chi' = [0, n_l - 2]_{\mathbb{N}}$, l'ensemble des valeurs des retenues.
- $Car = [1, n_c]_{\mathbb{N}}$, l'ensemble des indices des retenues.
- A, la matrice à indices dans $L \times C$ et à valeurs dans Let décrivant l'addition posée[2] .

[1] Les retenues ne sont pas nécessaires en CHIP. Elles se sont en simple MU-Prolog. Pour ce qui est d'ALICE, un énoncé sans les retenues est automatiquement traduit en énoncé avec retenues...

[2] Pour en faire une matrice, on remplace les vides par un ζ puis on impose $f(\zeta) = 0$.

Ex: $A = \begin{pmatrix} Z & S & E & N & D \\ Z & M & O & R & E \\ M & O & N & E & Y \end{pmatrix}$ avec $Z = 0$

```
GIVEN CST NL  NC
      SET LIN =   INT 1   NL
          COL =   INT 1   NC
          LIG =   LIN /   NL
          LET =   OBJ
          CHI =   INT 0   9
          CHG =   INT 0   NL  -  2
          CAR =   INT 1   NC
      MAT A   ->  PDT LIN COL LET

FIND  INJ F   ->  LET CHI
      FUN R   ->  CAR CHG
WITH  ALL J   IN  COL (   SUM I   IN  LIG (   F   A
      (   I   J   )   )   +   R   J   =   F   A   (
      NL  J   )   +   10  *   R   (   J   -   1   )
WITH  F   A   (   NL  1   )   DIF 0
WITH  R   0   =   0
WITH  F   Z   =   0
WITH  R   NC  =   0
END
3,5
S,E,N,D,M,O,R,Y,Z
Z,S,E,N,D
Z,M,O,R,E
M,O,N,E,Y
```

Figure 1.1: Send + More = Money en ALICE

Trouver:

1. Une injection[3] f, $Let \xrightarrow{f} Chi$
2. Une fonction r, $Car \xrightarrow{r} Chi'$

Telles que:

- $\forall\, j \in C\ \sum_{i \in L'} f(A(i,j)) + r(j) = f(a(n_l, j)) + 10r(j-1)$
- $f(A(n_l, 1)) \neq 0$
- $r(0) = 0$
- $f(\zeta) = 0$
- $r(n_c) = 0$

A la syntaxe près, certes un peu rébarbative, le problème se programme en mêmes termes avec ALICE (Figure 1.1). Ce sera le cas pour tous les problèmes suivants, dont seul l'énoncé mathématique sera donné.

[3] C'est plus exactement la restriction de f à $Let \setminus \{\zeta\}$ qui est une injection. Pour des soucis de clarté, ignorons cette petite erreur, déjà présente dans [Lau78]

1.1.2 Les huit reines

Il s'agit de placer n reines sur un échiquier $n \times n$, sans qu'aucune d'elles ne soit en prise. En général, on choisit $n = 8$. Deux formulations mathématiques sont possibles.

Formulation algébrique

Soient:

- $n \in \mathbb{N}$.
- $L = [1, n]_{\mathbb{N}}$, l'ensemble des lignes.
- $C = [1, n]_{\mathbb{N}}$, l'ensemble des colonnes.

Trouver:

1. Toutes les bijections f, $L \xrightarrow{f} C$

Telles que:

- $\forall (i, j) \in L^2,\ i \neq j \Rightarrow \begin{cases} f(j) \neq f(i) + j - i \\ f(j) \neq f(i) + i - j \end{cases}$

Par les graphes

Soient:

- $n \in \mathbb{N}$.
- $C = [1, n^2]_{\mathbb{N}}$, l'ensemble des cases.
- $G \subseteq C^2$, le graphe[4] des cases en disjonction.

Trouver:

1. Tous les sous-ensembles $F \subset C$.

de degré minimum n et respectant les disjonctions G, c'est-à-dire tels que:

- $|F| \geq n$
- $\forall (i, j) \in F^2,\ i \neq j \Rightarrow (i, j) \notin G$

Ce deuxième énoncé est plus efficace en ALICE, ce qui n'a rien d'étonnant puisque son algorithme spécialisé dans la recherche d'un sous graphe respectant les disjonctions est plus rapide que celui traitant les contraintes numériques.

C'est là une des grandes forces d'ALICE par rapport aux systèmes de résolution de contraintes en programmation logique, qui, jusqu'à présent, ne travaillent que sur des formulations algébriques. D'une manière générale, dès qu'un problème peut se poser un termes de propriétés sur des graphes ou des fonctions (bijectivité, injectivité, etc.), ALICE pose le problème de manière plus élégante, et le résout plus efficacement que CHIP.

[4] Qui dans ALICE doit être donné explicitement: pour chaque $c \in C$, on précise la liste des $d \in C$ en disjonction avec c, ce qui pour les huit reines est quelque peu épuisant.

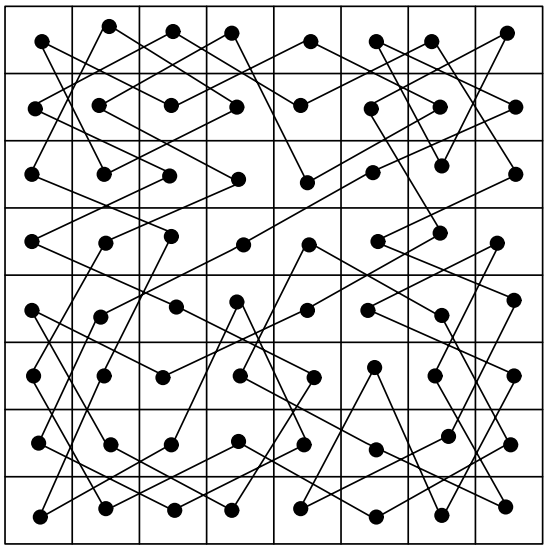

Figure 1.2: Une solution du Cavalier d'Euler

1.1.3 Le cavalier d'Euler

Il s'agit de passer par toutes les cases d'un échiquier une et une seule fois avec un cavalier (figure 1.2). Ici, la différence entre ALICE et CHIP à son comble:

- ALICE utilise un algorithme spécialisé sur ce problème. Autant dire que l'énoncé du problème n'a rien de général.
- En CHIP, ce problème est impossible à traiter. A moins de programmer complètement un algorithme de résolution – ce qui reviendrait à n'utiliser CHIP que comme un simple Prolog.

Bref, l'énoncé en ALICE se réduit à:

Soient:

- $n \in \mathbb{N}$.
- $C = [1, n^2]_{\mathbb{N}}$, l'ensemble des cases.
- $G \subseteq C^2$, le graphe symétrique des successeurs possibles.

Trouver:

1. Toutes les bijections f, $C \xrightarrow{f} C$

Dont le graphe, inclus dans G, décrit un chemin circulaire Γ dans C, c'est-à-dire telles que:

- $\forall c \in C, (c, f(c)) \in G$
- $\exists\, \Gamma = (c_1, \ldots, c_{n^2}) \in C^{n^2}, \begin{cases} \forall i, 1 \leq i < n^2 \Rightarrow f(c_i) = f(c_{i+1}) \\ f(c_{n^2}) = c_1 \end{cases}$

1.1.4 Le voyageur de commerce

Un voyageur de commerce doit passer une et une seule fois dans chacune des villes qu'il doit visiter. Le problème est de minimiser la distance parcourue, les distances entre les villes étant données. Là encore, ALICE fait appel à un algorithme particulier...

Soient:

- $n \in \mathbb{N}$
- $X = [1, n]_{\mathbb{N}}$, l'ensemble des noeuds.
- val une valuation, $X^2 \xrightarrow{val} \mathbb{N}$.

Trouver:

1. Une bijection f, $X \xrightarrow{f} X$.

Dont le graphe décrit un chemin circulaire sur X, minimal sur la valuation val, c'est-à-dire telle que:

$$\sum_{x \in X} val(x, f(x)) = \min_{g, X \xrightarrow{g} X} \sum_{x \in X} val(x, g(x))$$

1.2 Méthodes employées

ALICE est écrit en Fortran sous un système aujourd'hui disparu. Il semble qu'il soit désormais impossible de se le procurer, ni-même d'en voir une démonstration. C'est pourquoi son étude est un peu floue et ne repose que sur un article ([Lau78]). Bien des comportements d'ALICE tels qu'ils y sont décrits peuvent sembler étranges et non justifiés. Il n'en reste pas moins que l'article introduit des idées importantes en matière de résolution de systèmes de contraintes.

1.2.1 La méthode générale

[Lau78] fait un exposé théorique des méthodes utilisées dans ALICE. Ces méthodes sont ensuite appliquées sur des exemples traités en détail. En fait, l'exposé théorique n'est pas assez précis pour être compris, et le lecteur essaie rapidement de se raccrocher aux exemples. Or, à maintes reprises, les exemples utilisent des raisonnements originaux qui n'étaient pas prévus par la méthode générale. Parfois même, certaines manipulations de contraintes semblent le fait d'une intelligence qui n'a rien d'automatique. On a plusieurs fois l'impression que les problèmes sont résolus "à la main" et non par un automate.

Avant de citer quelques-uns de ces passages "choquants" de l'article, il est tout de même bon d'exposer le principe de base d'ALICE. Ce principe , clef de voûte d'ALICE, constitue le point commun réunifiant les différents algorithmes plus ou moins spécialisés qui se cachent derrière lui.

Derrière une abondance de règles détaillées entre lesquelles il est difficile de faire un lien et dont le lecteur ne peut extraire une méthode globale, voici ce qu'il faut retenir:

- ALICE possède une double représentation interne du problème: sous forme de graphe et sous forme algébrique. Les autres solveurs adoptent l'une seulement de ces deux représentations.
- Le graphe ne comporte pas seulement une représentation des ensembles de départ et d'arrivée des fonctions à trouver, mais des informations supplémentaires du type: cliques de disjonction, degrés minimum et maximum des images, valuations des arcs, valeurs extrêmes des valuations des arcs partant d'un ensemble ou y arrivant, images ou antécédents possibles des éléments, etc.

- Les deux représentations dialoguent , s'enrichissant mutuellement.
- Les problèmes à résoudre sont variés. Certains doivent trouver une solution satisfaisant un ensemble de contraintes, d'autres sont plutôt des problèmes d'optimisation. ALICE résout la question en proposant un algorithme en trois phases, certaines phases étant éventuellement inutiles suivant le problème à traiter:

  1. Recherche d'un solution admissible pour les contraintes.
  2. Recherche d'une "bonne" solution tenant compte de la fonction de coût.
  3. Recherche de l'optimum.

1.2.2 Exemples

Le but de cette section n'est pas de présenter la résolution d'un problème particulier par ALICE, mais de citer quelques-uns des passages peu convaincants des exemples donnés dans [Lau78].

Cas 1

En ce qui concerne les équations ou inéquations linéaires sur les entiers naturels, ALICE réduit les domaines des variables de la même manière que CHIP (voir section 2.4.3). Comment alors expliquer ce passage?

$$\left.\begin{array}{r} x_1 \in \mathbb{N} \\ x_3 \in \mathbb{N} \\ 0 \leq -2 + 2x_1 + 5x_3 \end{array}\right\} \Rightarrow (x_1, x_3) = (0,1) \text{ ou } (1,0)$$

Avec la méthode exposée, ALICE aurait du simplement en déduire que $x_1 \leq 1$ et $x_3 \leq 1$. Eliminer $(0,0)$ et $(1,1)$ pour (x_1, x_3) n'est a priori possible qu'en parcourant l'arbre de recherche. Or, Laurière présente bien l'implication ci-dessus comme une réduction de contraintes...

Cas 2

Pour ce qui est des substitutions entre équations, ALICE choisit toujours la bonne au bon moment, sans qu'aucune règle ne soit mentionnée. Ainsi, on a:

$$\left.\begin{array}{r} N \neq E \\ N = E + R_2 \end{array}\right\} \Rightarrow R_2 \neq 0$$

Ou mieux:

$$\left.\begin{array}{l} M_1 - M_2 + M_3 - M_4 = 0 \\ M_1 + M_2 + M_3 + M_4 = 3 \end{array}\right\} \Rightarrow 2(M_1 + M_3) = 3 \Rightarrow \textit{Impossible}$$

Remarquons au passage la déduction sur la parité, qui, si elle peut être automatisable, n'est pas citée comme méthode employée...

Cas 3

Enfin, certaines choses font vraiment trop "cas particulier":

> Lorsqu'ALICE repère(!) un cryptogramme arithmétique dont l'énoncé n'utilise pas les retenues, elle le repose avec les retenues.

ou encore:

> Dans le cas du carré magique[5], voyant que la fonction assignant un entier à chaque case du carré est une bijection sur $[1, n]_{\mathbb{N}}$, ALICE rajoute au système l'équation $\sum f(i) = n(n-1)/2$, puis par addition des contraintes sur les lignes du carré, en déduit la valeur de la somme, et enfin par substitution de contraintes convenablement choisies, trouve la valeur de la case centrale du carré[6].

On n'aurait pas mieux fait à la main!

1.3 Pourquoi s'intéresser à ALICE?

Après avoir légèrement malmené ALICE, on peut se demander pourquoi s'y être intéressé!

Certes l'article [Lau78] est flou sur bien des points. Certes ALICE ne peut plus être montrée. Il n'en reste pas moins qu'ALICE a existé et que, même si les méthodes employées de manière interne s'inséraient mal dans un cadre théorique global, l'article regorge de bonnes idées à glaner ça et là. C'est sans doute la raison pour laquelle cet article reste une référence.

Comment mettre à profit toutes ces idées? Là est le problème. Concevoir un système les utilisant toutes reviendrait à comprendre ALICE...

Enfin, [Lau78] présente ALICE comme "un langage pour poser les problèmes **et** un programme pour les résoudre". Ce qui est certain, c'est que le premier but est atteint.

[5] Soit à placer dans un carré les n premiers entiers de sorte que les sommes sur les lignes, sur les colonnes et les diagonales soient toutes égales.

[6] Pour un carré de coté impair

Chapitre 2

CHIP

Nous étudions maintenant le langage CHIP développé à l'ECRC. Nous présentons les concepts de generate and test, standard-backtracking, forward-checking *et* looking-ahead *ainsi que leur incarnation en CHIP. L'exemple des huit reines nous sert ensuite à comprendre l'utilité de ces concepts et la puissance de CHIP. Nous terminons le chapitre par une confrontation entre ALICE et CHIP.*

2.1 Programmation Logique avec Contraintes

Pour résoudre un problème combinatoire discret, on peut:

- Traduire le problème en programmation entière, puis utiliser les algorithmes usuels. Cette méthode introduit en général un nombre important de variables secondaires et de contraintes supplémentaires, ce qui rend le problème insoluble en pratique. Cette démarche n'utilise pas d'heuristique propre au problème. Beaucoup d'informations sont perdues dans la traduction.

- Utiliser un solveur comme ALICE, basé sur des techniques de consistance. C'est en général très efficace, mais il faut que le problème puisse être posé sous une forme connue du solveur. Cette solution manque de flexibilité, le solveur étant une boite noire dont il est impossible de modifier le comportement.

- Repartir de zéro, et écrire un programme spécifique en langage impératif tel que C. Ce qui est évidemment très efficace mais aussi très long. On exploite toutes les particularités du problème mais il est difficile de retoucher au programme après coup...

[JL87] et [JM87] ont introduit la notion de Programmation Logique avec Contraintes (CLP). Présentant la Programmation Logique usuelle comme un cas particulier de CLP sur l'univers de Herbrand, cette théorie rend naturel le traitement des contraintes algébriques en Programmation Logique.

Le langage CHIP s'appuie sur les idées apportées par cette théorie. Il travaille sur les "domaines finis", c'est-à-dire sur les sous-ensembles finis de $\mathbb{N}$.

Par réaction aux trois méthodes ci-dessus, CHIP se veut:

- Déclaratif (puisqu'il s'agit d'un langage de Programmation Logique).
- Apportant des méthodes et des outils généraux mais sans imposer la façon de les utiliser.
- Fournissant des méthodes spécialisées, implémentations efficaces des méthodes générales dans des cas souvent rencontrés.
- Numérique et symbolique.
- Libérant le programmeur des tâches de propagation des contraintes, et de recherche arborescente.

En fait, il semble que le but ne soit pas totalement atteint. Lorsque les implémentations spécialisées ne suffisent plus, quand le programmeur a recours aux méthodes générales, CHIP n'est plus très efficace. Mais pouvait-il en être autrement?

2.2 Théorie générale

Le but de cette section est d'introduire les termes de *generate and test*, *standard-backtracking*, *forward-checking* et *looking-ahead*. Leur signification est ici donnée dans un sens général. La section suivante exposera comment ils doivent être entendus en programmation sous CHIP.

2.2.1 Le problème en contraintes binaires

Soient $x_1, \ldots, x_n$ les variables intervenant dans le problème de satisfaction de contraintes, et $D_1, \ldots, D_n$ les domaines dans lesquels elles prennent respectivement leurs valeurs. On suppose que le problème est décomposé en contraintes binaires: $C_{ij}(x_i, x_j)$ pour $(i < j)$.

2.2.2 La procédure de backtrack

On peut concevoir la résolution du problème de satisfaction de contraintes comme la tâche de trouver tous les $(v_1, \ldots, v_n)$ satisfaisant un certaine propriété $P_n(v_1, \ldots, v_n)$, vérifiée si les contraintes sont satisfaites pour $x_1 = v_1, \ldots, x_n = v_n$.

La procédure générale de backtrack consiste alors à trouver une suite de propriétés intermédiaires $P_k(\langle v_1, \ldots, v_k\rangle, \langle d_{k+1}, \ldots, d_n\rangle)$ où $\langle v_1, \ldots, v_k\rangle$ représente les valeurs assignées aux $x_1, \ldots, x_k$ et $\langle d_{k+1}, \ldots, d_n\rangle$ les domaines au sein desquels peuvent encore prendre leur valeur les $x_{k+1}, \ldots, x_n$. Les P_k doivent satisfaire les propriétés suivantes:

1. $\forall\ k+1 \leq l \leq n,\ d_l \neq \emptyset$ et $d_l \subseteq D_l$
2. Si $v_{k+1} \in d_{k+1}$ et $e_k \subseteq d_l$ $(k+2 \leq l \leq n)$ alors:
   $P_{k+1}(\langle v_1, \ldots, v_{k+1}\rangle, \langle e_{k+2}, \ldots, e_n\rangle) \Rightarrow P_k(\langle v_1, \ldots, v_k\rangle, \langle d_{k+1}, \ldots, d_n\rangle)$

Autrement dit, s'il n'existe pas de domaines $d_{k+1} \subseteq D_{k+1}, \ldots, d_n \subseteq D_n$ non vides satisfaisant $P_k(\langle v_1, \ldots, v_k\rangle, \langle d_{k+1}, \ldots, d_n\rangle)$, alors on ne peut étendre cette solution partielle en un $v_{k+1} \in d_{k+1}$ et des $e_{k+2} \subseteq d_{k+2}, \ldots, e_n \subseteq d_n$ non vides vérifiant $P_{k+1}(\langle v_1, \ldots, v_{k+1}\rangle, \langle e_{k+2}, \ldots, e_n\rangle)$.

D'où une procédure d'obtention de $(v_1, \ldots, v_n)$ par induction:

1. [Initialisation] Assigner k à 0. Trouver $d_1, \ldots, d_n$ tels que $P_0(\langle\rangle, \langle d_1, \ldots, d_n\rangle)$ soit vérifiée, puis aller en 2. Si aucun $d_1, \ldots, d_n$ ne peut être trouvé, terminer (Pas de solution).

2. [Calcul des successeurs] (A ce stade, $P_k(\langle v_1, \ldots, v_k\rangle, \langle d_{k+1}, \ldots, d_n\rangle)$ est vérifiée, et $0 \leq k < n$) Calculer l'ensemble S_k de tous les $\langle v_{k+1}, e_{k+2}, \ldots, e_n\rangle$ tels que:
   (a) $e_i \subseteq d_i$ $(k+2 \leq i \leq n)$
   (b) $P_{k+1}(\langle v_1, \ldots, v_{k+1}\rangle, \langle e_{k+2}, \ldots, e_n\rangle)$ est vraie
   (c) Pour tout $f_{k+2} \subseteq d_{k+2}, \ldots, f_n \subseteq d_n$,
   Si $P_{k+1}(\langle v_1, \ldots, v_{k+1}\rangle, \langle f_{k+2}, \ldots, f_n\rangle)$ est vraie
   alors $f_i \subseteq e_i$ $(k+2 \leq i \leq n)$
3. [Tous les successeurs ont-ils été essayés?] Si S_k est vide aller en 6.
4. [Avancer en profondeur] Choisir un élément $\langle w_{k+1}, e_{k+2}, \ldots, e_n\rangle$ de S_k, le détruire de S_k, assigner $\langle v_{k+1}, d_{k+2}, \ldots, d_n\rangle$ à cet élément, et augmenter k de 1
5. [Solution?] (A ce stade $P_k(\langle v_1, \ldots, v_k\rangle, \langle d_{k+1}, \ldots, d_n\rangle)$ est vraie et $0 < k \leq n$). Si $k < n$ aller en 2. Sinon afficher la solution $(v_1, \ldots, v_n)$ et continuer en 6.
6. [Backtrack] (Toutes les extensions de $(v_1, \ldots, v_k)$ ont été explorées). Décroître k de 1. Si $k \geq 0$, retourner en 3, sinon terminer.

2.2.3 Les différentes méthodes

Le choix des propriétés P_k détermine la méthode employée.

Generate and test

En *generate and test*, $P_k(\langle v_1, \ldots, v_k\rangle, \langle d_{k+1}, \ldots, d_n\rangle)$ est vraie pour $k < n$ ssi:

1. $v_i \in D_i (1 \leq i \leq k)$

On ne teste les contraintes que lorsque toutes les variables ont été assignées.

Standard-backtracking

En *standard-backtracking*, $P_k(\langle v_1, \ldots, v_k\rangle, \langle d_{k+1}, \ldots, d_n\rangle)$ est vraie pour $k < n$ ssi:

1. $v_i \in D_i (1 \leq i \leq k)$
2. $\forall\, i, j\ (1 \leq i < j \leq k), C_{ij}(v_i, v_j)$ est vraie

Ici, on vérifie que chaque variable nouvellement assignée est consistante avec celles déjà assignées.

Forward-checking

En *forward-checking*, $P_k(\langle v_1, \ldots, v_k\rangle, \langle d_{k+1}, \ldots, d_n\rangle)$ est vraie pour $k < n$ ssi:

1. $v_i \in D_i (1 \leq i \leq k)$
2. $\forall\, i, j\ (1 \leq i < j \leq k), C_{ij}(v_i, v_j)$ est vraie
3. $\forall\, l\ (k < l \leq n),\ d_l = \{v_l \in D_l \mid C_{1l}(v_1, v_l), \ldots, C_{kl}(v_k, v_l)$ sont vraies$\}$

En plus du *standard-backtracking*, on vérifie que chaque variable non encore assignée possède au moins une valeur consistante avec les variables déjà assignées.

Looking-ahead

En *looking-ahead*, $P_k(\langle v_1, \ldots, v_k\rangle, \langle d_{k+1}, \ldots, d_n\rangle)$ est vraie pour $k < n$ ssi:

1. $v_i \in D_i (1 \leq i \leq k)$
2. $\forall\ i, j\ (1 \leq i < j \leq k), C_{ij}(v_i, v_j)$ est vraie
3. $\forall\ l\ (k < l \leq n),\ d_l = \{v_l \in D_l |$
   (a) $C_{1l}(v_1, v_l), \ldots, C_{kl}(v_k, v_l)$ sont vraies
   (b) $\exists\ w_{k+1}, \ldots, w_{l-1}, w_{l+1}, \ldots, w_n \in D_{k+1}, \ldots, D_{l-1}, D_{l+1}, \ldots, D_n$ qui satisfont les contraintes $C_{k+1l}(w_{k+1}, v_l), \ldots, C_{l-1l}(w_{l-1}, v_l), C_{ll+1}(v_l, w_{l+1}), \ldots, C_{ln}(v_l, w_n)$ }

En plus du *forward-checking*, on vérifie que chaque variable non encore assignée possède au moins une valeur consistante avec les autres variables non assignées.

2.2.4 Comparaisons

Dans le *generate and test*, tout l'arbre de recherche est exploré, puisqu'on ne vérifie la consistance de la solution que lorsque toutes les variables sont assignées.

Dans le *standard-backtracking*, l'arbre de recherche est restreint par une utilisation *a posteriori* des contraintes. Si la recherche d'un v_{k+1} pour étendre la solution partielle $\langle v_1, \ldots, v_k\rangle$ en un $\langle v_1, \ldots, v_{k+1}\rangle$ échoue, alors l'arbre de recherche est coupé à cet endroit. Si le gain est considérable par rapport au *generate and test*, c'est malgré tout *après* avoir assigné les variables qu'on découvre l'échec, d'où un comportement non optimum en ce sens que:

- On redécouvre plusieurs fois les mêmes faits: le fait que certaines valeurs des variables ne satisfont pas une certaine contrainte est redécouvert à chaque assignation de ces variables à ces valeurs, ce qui peut arriver plusieurs fois au cours de la recherche.
- Les échecs sont détectés trop tardivement, en particulier après des assignations de variables inutiles (non impliquées dans la contrainte non satisfaite).
- Le point de backtrack est mal choisi: à la découverte d'un échec, on revient sur le choix de la dernière assignation, qui n'est pas nécessairement responsable de l'échec.

Le *forward-checking* et plus encore le *looking-ahead* réduisent évidemment davantage l'arbre de recherche. Ils sont fondés sur l'idée souvent fructueuse[1] qu'il vaut mieux limiter l'arbre de recherche, quitte à perdre du temps en chacun de ses nœuds.

2.2.5 Application en Prolog

En Prolog, le problème des huit reines se programme de manière naturelle en *generate and test* (voir figure 2.1). Le prédicat `eight_queens` est bien construit sur une génération (`permutation`) puis un test (`safe`).

Dans le cas des huit reines, il n'est pas très compliqué de modifier le programme pour faire du *standard-backtracking* (figure 2.2). Toutefois, ce dernier exemple requiert un effort supplémentaire

[1] Mais à toujours contrôler! Le *forward-checking* est d'ailleurs souvent plus rapide que le *looking-ahead*, ce dernier étant pour certaines contraintes trop coûteux à mettre en œuvre.

```
eight_queens([X1,X2,X3,X4,X5,X6,X7,X8]) :-
        permutation([X1,X2,X3,X4,X5,X6,X6,X7,X8],[1,2,3,4,5,6,7,8]),
        safe([X1,X2,X3,X4,X5,X6,X6,X7,X8]).

permutation([],[]).
permutation([X|Xs],Ls) :-
        delete(X,Ls,Rs),
        permutation(Xs,Rs).

delete(X,[X|Xs],Xs).
delete(X,[Y|Ys],[Y|Rs]) :-
        delete(X,Ys,Rs).

safe([]).
safe([X|Xs]) :-
        noattack(X,Xs),
        safe(Xs).

noattack(X,Xs) :-
        noattack(X,Xs,1).

noattack(X,[],Nb).
noattack(X,[Y|Ys],Nb) :-
        X != Y - Nb,
        X != Y + Nb,
        Nb1 is Nb + 1,
        noattack(X,Ys,Nb1).
```

Figure 2.1: Les huit reines en *generate and test*

du programmeur. En général, les problèmes rencontrés se prêtent moins facilement que le cas des huit reines à l'implémentation en *standard-backtracking*. Il faut souvent imbriquer les phases de génération et de test, ce qui rend le programme peu clair.

Certains Prologs, tels que MU-Prolog ou Prolog-II, possèdent un mécanisme de *délai*. Lorsqu'un prédicat est rencontré en cours d'exécution, il peut être *endormi* si ses arguments ne sont pas suffisamment instanciés. C'est le cas de la non-égalité qui est endormie tant que ses deux arguments ne sont pas instanciés. Un mécanisme est alors mis en place pour *réveiller* les prédicats endormis lorsque les assignations ultérieures le permettent, ceci sans perte d'efficacité (pointeurs entre les variables et les prédicats endormis qu'elles peuvent réveiller).

Dans un Prolog muni d'un mécanisme de *délai*, il est redevient naturel de programmer en *standard-backtracking*: c'est le *coroutining*. Ainsi, dans l'exemple des huit reines, il suffit d'échanger l'ordre d'appel de la génération et du test pour obtenir un comportement de *standard-backtracking* (figure 2.3). L'appel de `safe` va générer toutes les contraintes sur les `Xi`, contraintes qui seront endormies. Puis lorsque `permutation` va assigner les `Xi`, ces contraintes seront progressivement réveillées, et détecteront dès que possible une incompatibilité entre les variables déjà instanciées, coupant aussitôt l'arbre de recherche.

Pour ce qui est du *forward-checking*, il est possible de le programmer en Prolog standard dans le cas des huit reines, mais cela n'a plus rien de très déclaratif (voir dans [Van89])! C'est même

```
eight_queens(X) :-
        queens(X,[],[1,2,3,4,5,6,7,8]).

queens([],Placed,[]).
queens([X|Xs],Placed,Values) :-
        delete(X,Values,New_values),
        noattack(X,Placed),
        queens(Xs,[X|Placed],New_values).

delete ...

noattack ...
```

Figure 2.2: Les huit reines en *standard-backtracking*

```
eight_queens([X1,X2,X3,X4,X5,X6,X7,X8]) :-
        safe([X1,X2,X3,X4,X5,X6,X6,X7,X8]).
        permutation([X1,X2,X3,X4,X5,X6,X6,X7,X8],[1,2,3,4,5,6,7,8]),

safe ....

permutation ...
```

Figure 2.3: Les huit reines avec *coroutining*

assez compliqué...

2.3 Sémantique de CHIP

L'objectif de cette section est de présenter de manière précise le comportement du *forward-checking* et du *looking-ahead* dans CHIP. Seules les règles d'inférence relatives au *forward-checking* et au *looking-ahead* sont données, telles qu'elles apparaissent dans [Van89]. Une connaissance des règles d'inférence usuelles de Prolog et du cadre théorique s'y rapportant est donc nécessaire pour profiter de ce qui suit...

2.3.1 Forward-checking

La définition suivante précise quels sont les prédicats concernés par la *Règle d'inférence du forward-checking* (FCIR).

Définition 2.1 (Forward-checkable) *Soit $p(t_1, \ldots, t_n)$ un atome. $p(t_1, \ldots, t_n)$ est dit* forward-checkable *ssi:*

1. *p est une contrainte.*

2. *Un et un seul des t_i est une variable de domaine, les autres étant déjà instanciés*[2]*. Ce terme est appelé la* forward-variable.

La règle d'inférence est alors:

Définition 2.2 (FCIR) *Soit P un programme, $G_i = \leftarrow A_1, \ldots, A_m, \ldots, A_k$ un but. Le but G_{i+1} est* dérivé par la FCIR *du but G_i et du programme P en utilisant la substitution θ_{i+1} ssi:*

1. *A_m est* forward-checkable *et x^d est la* forward-variable*, de domaine d.*
2. *$e = \{a \in d \mid P \models A_m\{x^d/a\}\} \neq \emptyset$.*
3. *θ_{i+1} est la substitution:*
   - *$\{x^d/c\}$ si $e = \{c\}$;*
   - *$\{x^d/z^e\}$ où z^e est une nouvelle variable de domaine e sinon.*
4. *G_{i+1} est le but $\leftarrow (A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta_{i+1}$.*

La FCIR traite les contraintes de manière active:

- Elle réduit *a priori* l'arbre de recherche puisque certaines valeurs impossibles de la *forward-variable* ne seront plus reconsidérées.
- Elle instancie la *forward-variable* lorsque son domaine se restreint à une valeur.

2.3.2 Looking-ahead

Les prédicats concernés par la *Règle d'inférence du looking-ahead* (LAIR) sont définis par:

Définition 2.3 (Lookahead-checkable) *Soit $p(t_1, \ldots, t_n)$ un atome. $p(t_1, \ldots, t_n)$ est dit* look-ahead-checkable *ssi:*

1. *p est une contrainte.*
2. *Un au moins des t_i est une variable de domaine, les autres étant déjà instanciés ou des variables de domaine. Ceux des t_i qui sont des variables de domaine sont appelés les* look-ahead-variables.

La règle d'inférence du *looking-ahead* est:

Définition 2.4 (LAIR) *Soit P un programme, $G_i = \leftarrow A_1, \ldots, A_m, \ldots, A_k$ un but. Le but G_{i+1} est* dérivé par la LAIR *du but G_i et du programme P en utilisant la substitution θ_{i+1} ssi:*

1. *A_m est* lookahead-checkable *et $x_1, \ldots, x_n$ sont les* lookahead-variables*, de domaines $d_1, \ldots, d_n$.*
2. *Pour chaque x_j, $e_j = \{v_j \in d_j \mid \exists\ v_1 \in d_1, \ldots, v_{j-1} \in d_{j-1}, v_{j+1} \in d_{j+1}, \ldots, v_n \in d_n$ tels que $P \models A_m\theta$ avec $\theta = \{x_1/v_1, \ldots, x_n/v_n\}\} \neq \emptyset$.*

[2] "Ground".

3. *z_j est la constante c si $e_j = \{c\}$, ou une nouvelle variable de domaine sur e_j sinon.*
4. *$\theta_{i+1} = \{x_1/z_1, \ldots, x_n/z_n\}$.*
5. *G_{i+1} est, soit le but $\leftarrow (A_1, \ldots, A_{m-1}, A_{m+1}, \ldots, A_k)\theta_{i+1}$ si au plus l'un des z_i est une variable de domaine, soit $\leftarrow (A_1, \ldots, A_k)\theta_{i+1}$ sinon.*

La LAIR peut être vue comme une généralisation de la FCIR. Lorsqu'il n'y a qu'une *lookahead-variable*, la LAIR se comporte comme la FCIR, sinon, elle réduit le domaine des *lookahead-variables*, et instancie celles qui peuvent l'être. De plus, ses conditions d'application (*lookahead-checkable*) sont moins restrictives que pour la FCIR (*forward-checkable*).

Toutefois, son implémentation est souvent trop coûteuse, voire impossible. [Van89] introduit donc une notion restreinte: le *looking-ahead partiel*.

2.3.3 Looking-ahead partiel

Les prédicats concernés par la *Règle d'inférence du looking-ahead partiel* (PLAIR) sont les mêmes que pour la LAIR (*lookahead-checkable*). La règle d'inférence devient:

Définition 2.5 (PLAIR) *Soit P un programme, $G_i = \leftarrow A_1, \ldots, A_m, \ldots, A_k$ un but. Le but G_{i+1} est* dérivé par la PLAIR *du but G_i et du programme P en utilisant la substitution θ_{i+1} ssi:*

1. *A_m est* lookahead-checkable *et $x_1, \ldots, x_n$ sont les* lookahead-variables, *de domaines $d_1, \ldots, d_n$.*
2. *Pour chaque x_j, $d_j \supseteq e_j \supseteq \{v_j \in d_j \mid \exists\, v_1 \in d_1, \ldots, v_{j-1} \in d_{j-1}, v_{j+1} \in d_{j+1}, \ldots, v_n \in d_n$ tels que $P \models A_m\theta$ avec $\theta = \{x_1/v_1, \ldots, x_n/v_n\}\} \neq \emptyset$.*
3. *z_j est la constante c si $e_j = \{c\}$, ou une nouvelle variable de domaine sur e_j sinon.*
4. *$\theta_{i+1} = \{x_1/z_1, \ldots, x_n/z_n\}$.*
5. *G_{i+1} est le but $\leftarrow (A_1, \ldots, A_k)\theta_{i+1}$.*

La PLAIR ne précise pas le choix des e_j. Notion intermédiaire entre la FCIR et la LAIR, elle est introduite pour rendre compte de l'implémentation des contraintes spécialisées *"built-in"* en CHIP.

2.4 En pratique

La présente section décrit trois prédicats *"built-in"*:

- La non-égalité, (`X != Y`), spécialisation de la FCIR.
- Le maximum, (`max(X,Y,Z)`), spécialisation de la LAIR.
- Les équations ou inéquations linéaires

$$a_1X_1 + \cdots + a_nX_n + a \top b_1Y_1 + \cdots + b_mY_m + b$$

avec $\top \equiv$ ('=', '<' ou '$\leq$'), $a_i \in \mathbb{N}, a \in \mathbb{N}, b_j \in \mathbb{N}, b \in \mathbb{N}$

spécialisations de la PLAIR.

```
lookahead max(d,d,d).
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- X < Y.
```

Figure 2.4: Le prédicat max redéfini

2.4.1 La non-égalité

En CHIP, la non-égalité est implémentée en *forward-checking*. Le prédicat `X != Y` est donc réveillé lorsqu'il est *forward-checkable*, c'est-à-dire quand l'un des deux termes `X` ou `Y` est déjà assigné, l'autre étant une variable de domaine. La sémantique opérationnelle est la suivante:

- Si `X` est une variable de domaine D, et `Y` un entier naturel, on pose $D' = D \backslash \{Y\}$. Si $D' = \{e\}$, alors `X` est liée à `e`, sinon `X` est liée à une nouvelle variable `Z` de domaine D'. Dans les deux cas la contrainte n'échoue pas.

- Si `Y` est une variable de domaine et `X` un entier, on traite le cas de la même façon.

- Si `X` et `Y` sont tous deux entiers naturels, la contrainte échoue s'ils sont égaux, sinon elle est satisfaite.

2.4.2 Le prédicat maximum

Le prédicat `max(X,Y,Z)`, vrai si $Z = max(X, Y)$, est un bon candidat pour la LAIR. On peut de manière non ambiguë réduire les domaines des paramètres (choix des e_j dans la définition 2.4). `Max(X,Y,Z)` pourrait être programmé en CHIP en utilisant la déclaration `lookahead` qui rend un prédicat sujet à la LAIR en définissant sous quelles conditions il devient *lookahead-checkable* (On précise par un `d` quels paramètres doivent au moins être des variables de domaine, les autres devant être entièrement assignés). Ce qui donnerait le programme de la figure 2.4, si on veut déclencher la LAIR sur `max(X,Y,Z)` dès que `X,Y` et `Z` sont des variables de domaine.

Toutefois, la LAIR est très coûteuse en CHIP sur les prédicats non *"built-in"* [3]. Dans le cas de `max`, une implémentation spécifique respectant la LAIR est réalisée. Supposons par exemple que `X`, `Y` et `Z` soient trois variables de domaine, on peut alors retirer du domaine de `Z` les valeurs inférieures à $max(min(X), min(Y))$ et celles supérieures à $min(max(X), max(Y))$. Des considérations analogues à peine un peu plus compliquées sont appliquées à `X` et `Y`.

Permettons nous à ce propos une REMARQUE IMPORTANTE: on raisonne trop souvent sur les domaines en s'imaginant des intervalles de $\mathbb{N}$, alors qu'il s'agit de parties finies de $\mathbb{N}$. Ainsi, on peut supposer que c'est par soucis de limiter son exposé et non par oubli, que l'auteur de [Van89] ne précise pas qu'il faut dans l'exemple ci-dessus retirer du domaine de `Z` les valeurs qui ne sont pas dans l'union des domaines de `X` et de `Y`... Sinon ce n'est plus de la LAIR mais de la PLAIR. Je n'ai pas eu l'honneur d'utiliser CHIP pour vérifier...

[3] Du coup elle n'est pas très souvent utilisable! En fait son implémentation se veut générale et respecte parfaitement la définition 2.4. Pour ce faire, on est amené à réellement essayer toutes les valeurs possibles de chacune des *lookahead-variables*, puis à voir celles qui sont consistantes avec un jeu quelconque des autres variables, ce qui peut entraîner loin...

2.4.3 Les équations ou inéquations linéaires

Les cas des équations ou inéquations linéaires

$$a_1X_1 + \cdots + a_nX_n + a \top b_1Y_1 + \cdots + b_mY_m + b$$

$$\text{avec } \top \equiv (\text{`}=\text{'}, \text{`}<\text{'} \text{ ou } \text{`}\leq\text{'}), a_i \in \mathbb{N}, a \in \mathbb{N}, b_j \in \mathbb{N}, b \in \mathbb{N}$$

est un cas très important et très fréquemment utilisé. Le comportement de CHIP se base sur des idées déjà appliquées par ALICE.

Prenons le cas de l'égalité:

$$a_1X_1 + \cdots + a_nX_n + a = b_1Y_1 + \cdots + b_mY_m + b$$

On peut (dès que la contrainte est *lookahead-checkable*) calculer les intervalles de variation respectifs $[min_a, max_a]_{\mathbb{N}}$ et $[min_b, max_b]_{\mathbb{N}}$ de $a_1X_1 + \cdots + a_nX_n + a$ et de $b_1Y_1 + \cdots + b_mY_m + b$. Puis réduire les domaines des variables avec des inégalités telles que:

$$max(min_a, min_b) - (\sum_{k=1, k\neq i}^{n} (a_k max(X_k)) + a) \leq X_i$$

$$X_i \leq min(max_a, max_b) - (\sum_{k=1, k\neq i}^{n} (a_k min(X_k)) + a)$$

Cette réduction des domaines n'entre pas dans le cadre de la LAIR[4] , d'où l'introduction de la PLAIR comme justification théorique.

2.5 Application

A l'aide d'un seul exemple, celui des huit reines, on peut voir que le *forward-checking* se programme simplement en CHIP... si toutefois on a bien compris le mécanisme de réveil.CHIP se veut un langage souple mettant à la disposition de l'utilisateur les outils nécessaires à la résolution de son problème, sans pour autant lui imposer de stratégie. Il est donc du ressort du programmeur de choisir la bonne méthode.

2.5.1 Une première version

Considérons le programme "naïf" donné figure 2.5. La déclaration `domain eight_queens(1..8)` définit le domaine des variables de la liste passée en argument comme étant $[1, 8]_{\mathbb{N}}$. Le prédicat `indomain(X)` fait prendre à la variable `X` toutes les valeurs encore dans son domaine au moment de l'appel.

Ce programme est correct et résout le problème des huit reines en *forward-checking*. Pourtant son comportement n'est pas optimum. Suivons le début de son exécution pas à pas.

Le prédicat `indomain(X1)` choisit tout d'abord d'assigner `1` à la variable `X1`. Puis l'appel du prédicat récursif `noattack(X1,[X2,X3,X4,X5,X6,X7,X8])` retire des domaines des autres variables

[4]Une manipulation plus fine des domaines peut réduire davantage l'arbre de recherche. C'est ce qui a été testé dans la maquette CLP^{++} présentée au chapitre 3.

```
domain eight_queens(1..8).
eight_queens([X1,X2,X3,X4,X5,X6,X7,X8]) :-
        queens([X1,X2,X3,X4,X5,X6,X7,X8]).

queens([]).
queens([X|Y]) :-
        indomain(X),
        noattack(X,Y),
        queens(Y).

noattack(X,Xs) :-
        noattack(X,Xs,1).

noattack(X,[],Nb).
noattack(X,[Y|Ys],Nb) :-
        X != Y,
        X != Y - Nb,
        X != Y + Nb,
        Nb1 is Nb + 1,
        noattack(X,Ys,Nb1).
```

Figure 2.5: Les huit reines en *forward-checking* naïf

les valeurs rendues impossibles du fait du placement de `X1` en `1`. On est dans la situation de la figure 2.6, où les cases en gris sont des cases impossibles. Aucune des contraintes de non-égalité générées par `noattack` n'a été endormie puisque toutes étaient *forward-checkable* (`X` entier naturel et `Y`, `Y-Nb`, `Y+Nb` variables de domaine). Au contraire, elles ont joué un rôle actif de réduction de domaine.

L'appel de `indomain(X2)` la valeur `3` va tout de suite être considérée pour `X2`[5]. Puis le prédicat `noattack(X2,[X3,X4,X5,X6,X7,X8])` va retirer les valeurs impossibles des variables restantes, et ainsi de suite...

Après le choix de `X3` pour laquelle `5` est la première valeur permise, la situation est celle de la figure 2.7. On voit alors qu'il n'y a plus que `4` comme valeur permise pour `X6`, ce qui revient à avoir déjà placé la sixième reine.

Toutefois, aucune des contraintes entre `X6` et les variables restantes `X4,X5,X7` et `X8` n'est réveillée par l'assignation de `X6`, pour la simple raison que ces contraintes ne sont pas encore générées. Ainsi, il faudra attendre au tour suivant l'assignation de `X4` à la valeur `2` pour générer le contrainte `X4 != X6 - 2` et constater un échec. Le programme suivant fait mieux...

2.5.2 Une version améliorée

La bonne méthode consiste à:

[5] Notons qu'en *standard-backtracking* à la MU-Prolog, les valeurs `1` et `2` auraient été testées, puisque le réveil de la non-égalité ne se fait qu'une fois les deux arguments assignés: les contraintes jouent un rôle **passif**: elles vérifient la cohérence dès que possible mais n'interviennent pas sur la génération des variables en retirant par exemple certaines valeurs de leur domaine. D'où le qualificatif d'**actif** donné aux contraintes de CHIP.

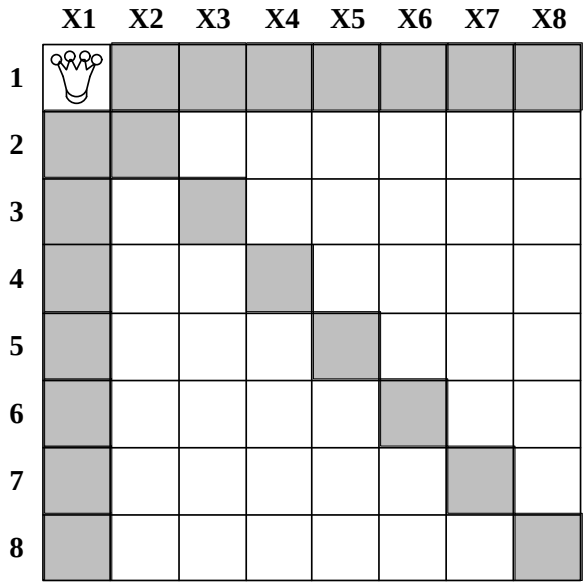

Figure 2.6: Forward-checking "naïf" après un choix

1. Générer **toutes** les contraintes.
2. **Puis** explorer l'arbre de recherche.

En fait, le programme de la figure 2.5 pensait naïvement générer convenablement les contraintes au fur et à mesure, dès qu'elles devenaient utiles, tel qu'on le faisait en MU-Prolog...

Le programme de la figure 2.8, qui applique la méthode ci-dessus est de loin plus efficace. Le prédicat `safe` génère et endort toutes les contraintes puis `label`[6] explore l'arbre.

La différence d'efficacité entre les deux versions du programme se constate dès le troisième choix (voir figure 2.9) puisque plus aucun choix n'est ensuite à faire! En effet, l'assignation `X6=4` réveille les contraintes où `X6` intervient, ce qui notamment restreint le domaine de `X8` à la seule valeur `X8=7`, d'où de nouveaux réveils, puis `X4=8`, `X7=2` et enfin constatation de l'échec puisque `X5` se retrouve avec un domaine vide. Ainsi, la branche `(X1,X2,X3)=(1,3,5)` est constatée infructueuse sans nouveau choix de variable.

2.5.3 First-fail principle

Le prédicat `label` ci-dessus génère les variables dans l'ordre de la liste qui lui est passée en argument. Il existe un prédicat *"built-in"* en CHIP rendant possible l'application du *first-fail principle*:

> Pour vérifier la cohérence d'une solution, mieux vaut **d'abord considérer ce qui a le plus grand risque d'échouer**.

Ce qui dans notre cas peut se traduire en:

> Générer en priorité les variables dont le domaine est le plus restreint.

[6] Qui est une procédure générale, utilisable dans tous problèmes où la méthode ci-dessus est appliquée

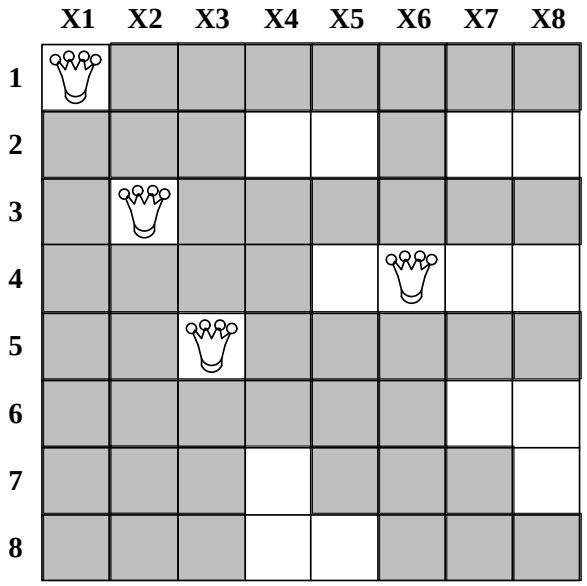

Figure 2.7: Forward-checking "naïf" après trois choix

Nombre de reines	Programme naïf	Test and Generate	First-Fail Principle
4	0.10s	0.09s	0.09s
6	0.46s	0.25s	0.29s
8	1.53s	0.74s	0.77s
10	1.70s	0.91s	0.57s
12	4.22s	2.08s	1.74s
14	34.03s	12.26s	1.73s
16	185.78s	70.07s	1.09s

Tableau 2.1: Temps d'exécution pour les n reines

Le prédicat `deleteff(X,Vars,Remain)` assigne à `X` celle des variables de la liste `Vars` dont le domaine est le plus restreint, et place dans `Remain` la liste des variables restantes. D'où la simple modification de `label` pour appliquer le *first-fail principle* (figure 2.10).

Dans le cas des n reines, l'application de ce principe donne de très bons résultats (voir tableau 2.1).

```
domain eight_queens(1..8)
eight_queens([X1,X2,X3,X4,X5,X6,X7,X8]) :-
        safe([X1,X2,X3,X4,X5,X6,X7,X8]),
        label([X1,X2,X3,X4,X5,X6,X7,X8]).

safe([]).
safe([F|T]) :-
        noattack(F,T),
        safe(T).

label([]).
label([X|Y]) :-
        indomain(X),
        label(Y).

noattack ....
```

Figure 2.8: Les huit reines en *forward-checking*

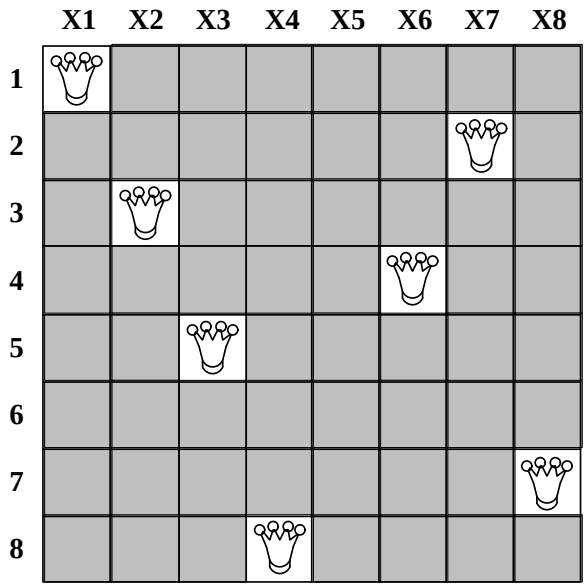

Figure 2.9: Forward-checking "amélioré" après trois choix

```
label([]).
label([F|T]) :-
        deleteff(X,[F|T],Ls),
        indomain(X),
        label(Ls).
```

Figure 2.10: "First-fail principle" pour les huit reines

2.6 ALICE et CHIP

Il est délicat de vouloir comparer CHIP et ALICE, notamment à cause du flou qui règne autour d'ALICE. Ce qui est sûr, c'est que les deux programmes adoptent des philosophies totalement opposées:

- ALICE se veut un solveur efficace et complet. Tout est mis en œuvre pour atteindre cette efficacité. La cohérence théorique et la souplesse d'utilisation passent en second. ALICE n'est pas un véritable langage, en ce sens qu'on ne peut l'utiliser pour développer une application complète. J.L. Laurière en est conscient en présentant ALICE comme:
  - Un langage **pour poser les problèmes**.
  - Un programme pour les résoudre.

  et non comme un langage à part entière.
- CHIP est avant tout un langage de Programmation Logique, possédant toute la puissance d'un Prolog avec *délai*. C'est dans le cadre naturel de [JL87], qu'il travaille sur les domaines finis et résout un certain nombre de problèmes avec contraintes sur ces domaines finis. Le côté solveur n'est qu'un plus, certes important puisque c'est ce plus qui fait de CHIP un langage commercial compétitif.

Pour ce qui est du nombre de problèmes traités, il est à peu près évident qu'ALICE est plus puissant. Même en supposant que tous les problèmes annoncés dans [Lau78] n'ont pas été réellement traités, il est certain que tous sont exprimables en ALICE, ce qui n'est pas le cas en CHIP("cavalier d'Euler" par exemple).

Toutefois, CHIP n'est pas figé dans son développement. Un certain nombre de contraintes *"built-in"* peuvent être et sont rajoutées par ses concepteurs au fur et à mesure des besoins. Ainsi, dès le départ, CHIP possédait un algorithme de *branch-and-bound* lui permettant de résoudre des problèmes d'optimisation du type *minmax*. Evidemment, ces contraintes ne s'insèrent pas dans le schéma théorique général[7].

En fait, le programmeur qui utilise CHIP se heurte à deux catégories de problèmes:

1. Les contraintes ou stratégies de résolution qui n'existent pas en *"built-in"*, mais qu'il peut reprogrammer avec les méthodes générales de CHIP - ce qui n'est pas très efficace - ou de zéro en utilisant CHIP comme un Prolog.
2. Les contraintes ou stratégies qui n'existent pas en *"built-in"*, et qu'il est impossible de reprogrammer sans toucher aux sources de CHIP.

On peut espérer que la majorité des problèmes se situent dans la première catégorie. Ce qui signifie que dans la pratique il faudra faire un effort de programmation supplémentaire, mais on pourra toujours obtenir une maquette. Parfois le manque d'efficacité de la maquette sera supportable, parfois il faudra retoucher aux sources de CHIP pour une implémentation directe en *"built-in"*; dans ce cas, la maquette aura au moins servi à valider le modèle.

Prenons l'exemple de `deleteff` qui est utilisé pour appliquer le first-fail principle. Le problème est de trouver dans une liste de variables celle dont le domaine est le plus restreint. En son absence, implémenter un tel prédicat en CHIP aurait été possible, mais vraisemblablement très inefficace. On aurait pu malgré tout vérifier l'utilité du first-fail principle sur le problème des n reines.

[7] Déjà alourdi par l'introduction d'un *looking-ahead partiel*...

A l'opposé, il est quelquefois nécessaire[8] de disposer d'un prédicat signifiant que parmi une liste de contraintes, exactement n d'entre elles sont vraies, et les autres fausses. Une implémentation à base de variables booléennes[9] est en théorie possible mais elle introduit beaucoup trop de variables secondaires pour pouvoir aboutir. Certains prédicats ont donc du être rajoutés à CHIP:

- Le prédicat `atmost` impose qu'au plus n contraintes sont vraies. Bien que décrit dans aucune publication, ce prédicat existe en CHIP. Il a du être implémenté quand le besoin s'en est fait sentir.
- Plus récemment, l'opérateur de cardinalité introduit par Van Hentenryck dans [VY91], généralise un peu le problème ("entre n et p contraintes sont vraies", génération à un meta-niveau de la liste des contraintes,...). Il fera vraisemblablement partie d'une future version de CHIP.

En conclusion, on peut dire que, du cadre théorique initial au langage final, un certain nombre de concessions est inévitable, et à trois titres:

1. Certains opérateurs ne peuvent raisonnablement suivre qu'une version affaiblie de la théorie (ex: PLAIR).
2. D'autres sont bien plus efficaces implémentés directement "en dur". D'où beaucoup de cas particuliers, qui ont tendance à devenir les opérateurs les plus utilisés en pratique.
3. Enfin, pour résoudre les problèmes se posant dans une application réelle, il faut rajouter des opérateurs spécialisés sans rapport avec le reste (ex: `min-max`).

Peut-être CHIP est-il encore un peu jeune. La démarche pour concevoir un tel langage serait donc:

1. Concevoir un cadre théorique solide, le plus large possible.
2. Implémenter la langage sur ces bases théoriques.
3. Confronter la langage à la réalité, et implémenter ce qui fait défaut.
4. Quand le tout est stabilisé, concevoir un nouveau cadre théorique tenant compte de tous les rajouts, puis éventuellement revoir l'implémentation si elle peut bénéficier de ces nouvelles idées.

Il est probable que:

- Beaucoup de langages s'arrêtent à l'étape 1, éventuellement à la 2.
- Certains sont confrontés à la réalité, mais rien de plus n'est implémenté.
- Un langage qui a réussi est un langage ayant passé l'étape 3. Mais une fois le tout stabilisé, aucun cadre théorique unificateur n'est recherché. D'ailleurs est-ce bien utile?
- Enfin, l'étape 4 est souvent réductrice et non unificatrice, d'où retour à l'étape 2.
- De toute façon, si l'étape 4 est passée avec succès, l'utilisateur devient plus exigeant. Retour en 1! Ce qui est somme tout rassurant...

[8] Voir le cas de la multiplication à la section 3.7.4.

[9] Au même titre que les domaines finis, CHIP connaît les booléens et les traite de manière spécifique.

Chapitre 3

CLP^{++}

*Nous présentons ici CLP^{++}, une maquette de solveur de contraintes sur les domaines finis réalisée en C++. Destiné à appréhender les problèmes posés par la programmation d'un tel système, CLP^{++} a été l'occasion d'introduire des concepts originaux, notamment ceux d'*opérations entre domaines finis *et de* rôle stratégique d'une contrainte. *Nous comparons CLP^{++} à CHIP. Enfin, après un rapide exposé de certains détails d'implémentation, nous étudions quelques applications avant de conclure.*

CLP^{++} **n'est pas** un éventuel CLP($\mathcal{X}$) amélioré. Il s'agit simplement d'une maquette en C++[1] implémentant un solveur de contraintes sur les domaines finis. Le but de cette maquette est double:

1. Pédagogique: rien ne remplace l'expérience d'une implémentation. C'est évidemment le seul moyen de découvrir les problèmes qui se posent inévitablement quand on passe de la théorie à la pratique. Mais c'est aussi un des meilleurs moyens de prendre du recul par rapport à une théorie appréhendée au travers d'articles.

   > Sur le papier, on ne peut que comprendre passivement une théorie informatique. Ce n'est qu'en implémentant qu'on redécouvre les choix qui ont été ceux des auteurs, choix qui, alors seulement, peuvent être adoptés ou rejetés.

2. Expérimental: seule une maquette personnelle permet de tester facilement différents algorithmes originaux, et de comparer leur efficacité à ceux employés par les auteurs.

Enfin, il n'est jamais mauvais de programmer un petit peu...

3.1 Les caractéristiques de CLP^{++}

Ont été implémentés dans CLP^{++}:

[1] D'où le nom...

- **Les domaines finis sur** $\mathbb{N}$. Il apparaît que certaines parties de l'algorithme de résolution s'implé-mentent plus facilement avec des variables intermédiaires sur les entiers relatifs. Toutefois, les contraintes d'inégalité ne sont exploitables que sur les entiers naturels. CLP++ travaillent de manière interne sur $\mathbb{Z}$ tout en gardant une sémantique sur $\mathbb{N}$.
- Des opérations originales sur ces domaines:
  - Intersection, union, réductions, etc.
  - Mais aussi **addition, soustraction et multiplication avec seuil**[2].
  - **Enumération de domaines**[3].
- **Des contraintes**:
  - Considérées comme des **objets jouant un rôle passif ou actif**:
    - Passif: capables de dire si elles sont vraies, fausses, ou seulement possibles.
    - Actif: capables de **réduire les domaines suivant une sémantique ajustable**[4] **entre le *looking-ahead* et le *looking-ahead partiel*, ou d'injecter des nouvelles contraintes au système**.
  - Sur les domaines finis:
    - **L'opérateur cardinal** introduit dans [VY91].
    - Egalité, inégalités et non-égalité sur les **polynômes de degré partiel un en chacune des variables** (ex: `X*Y+2*Z` mais pas `X*X+1`).
- Une recherche avec backtrack, mécanisme de *délai* et first-fail principle.

3.2 Algorithme de résolution

Voici, décrit succinctement, le principe de l'algorithme de résolution.

1. Les contraintes possèdent deux comportements correspondant aux deux rôles, actif et passif:
   (a) Un comportement "Preuve" correspondant au rôle passif. Il détecte si la contrainte est vraie ou fausse et retourne:
   - "Vraie" si la contrainte est vérifiée.
   - "Fausse" si la contrainte est fausse.
   - "Possible" sinon.

   (b) Un comportement "Action" correspondant au rôle actif. Il cherche à réduire le domaine des variables, et éventuellement détecte en cours le caractère vrai ou faux de la contrainte. Il retourne:
   - "Vraie" si la contrainte devient vérifiée.
   - "Fausse" si la contrainte devient fausse.
   - "Active" si des domaines ont été réduits.
   - "Inactive" sinon.

[2] Voir la section 3.3.2

[3] Voir section 3.6.2

[4] Rôle du seuil de la section 3.3.2

Lorsque des domaines sont réduits, c'est à la charge de la contrainte de réveiller les contraintes nécessaires.

2. Les contraintes possèdent trois états:
   (a) Un état "Vraie" pour une contrainte déjà vérifiée.
   (b) Un état "Active" pour une contrainte encore susceptible de jouer un rôle.
   (c) Un état "Endormie" pour une contrainte à réveiller.

   NB: Il n'y a pas d'état "Fausse" puisqu'une contrainte fausse termine immédiatement la recherche.

3. Une structure de pile de sauvegarde (trail) mémorise les modifications des variables, des états des contraintes, etc.
   - A l'ouverture, elle s'initialise.
   - Jusqu'à l'ouverture d'un trail de niveau inférieur, elle mémorise les modifications.
   - A la fermeture, elle rétablit ce qui a été mémorisé.

4. L'algorithme de résolution est composé de trois modules:
   (a) Un module de propagation des contraintes.
   (b) Un module de choix.
   (c) Un module de résolution.

5. Le module de propagation utilise les contraintes pour réduire le domaine des variables et éventuellement détecter un échec ou une réussite. Il retourne:
   - "Echec" si une contrainte fausse est détectée.
   - "Réussite" si toutes les contraintes sont vérifiées.
   - "Terminé" sinon (plus de propagation possible).

   Il est construit sur le schéma suivant:

```
◇ Tant qu'il existe une contrainte "Active"
    ◇ Pour toutes les contraintes "Actives" faire
        ◇ "Prouver" la contrainte
        ◇ Si la preuve retourne "Fausse" alors
            ◇ Retourner "Echec".
        ◇ Si la preuve retourne "Vraie" alors
            ◇ Marquer la contrainte comme "Vraie"
        ◇ Si la preuve retourne "Possible" alors
            ◇ Faire "Agir" la contrainte
            ◇ Si l'action retourne "Fausse" alors
                ◇ Retourner "Echec"
            ◇ Si l'action retourne "Vraie" alors
                ◇ Marquer la contrainte comme "Vraie"
            ◇ Si l'action retourne "Inactive" alors
                ◇ Marquer la contrainte comme "Endormie"
    ◇ Si toutes les contraintes sont "Vraies" alors
        ◇ Retourner "Réussite"
◇ Retourner "Terminé"
```

6. Le module de choix intervient quand la propagation est épuisée. Il émet des suppositions, qui pour l'instant[5] portent sur la valeur d'une variable, qu'il choisit en accord avec le first-fail principle.

   Il ne retourne pas de valeur. Son schéma est le suivant:

   - ◇ Choisir la variable `X` à énumérer
   - ◇ Si plus de variable alors
     - ◇ Retourner
   - ◇ Pour toutes les valeurs du domaine de `X`
     - ◇ Initialiser une pile de sauvegarde (ouvrir un trail)
     - ◇ Assigner cette valeur à `X`
     - ◇ Propager les contraintes
     - ◇ Si la propagation retourne "Réussite" alors
       - ◇ Imprimer la solution
     - ◇ Si la propagation retourne "Terminé" alors
       - ◇ Appeler le module de choix (récursion)
     - ◇ Rétablir les valeurs (fermer le trail)
   - ◇ Retourner

7. Le module de résolution initialise les contraintes et lance le module de choix. Il se compose des étapes suivantes:

   - ◇ Marquer toutes les contraintes comme actives
   - ◇ Ouvrir un trail
   - ◇ Propager les contraintes
   - ◇ Si la propagation retourne "Réussite" alors
     - ◇ Afficher la solution.
   - ◇ Si la propagation retourne "Terminé" alors
     - ◇ Initialiser le choix des variables (labeling)
     - ◇ Lancer le module de choix
   - ◇ Fermer le trail
   - ◇ Retourner

8. Le module de propagation est aménagé pour aussi résoudre certains problèmes d'ordonnancement (voir section 3.7.5).

3.3 Les domaines finis

3.3.1 Sur les entiers relatifs

Les domaines finis sont ici les parties finies de l'ensemble des entiers relatifs ($\mathbb{Z}$). Si les variables de domaines seront supposées positives pour la résolution des contraintes d'inégalité, il est en effet intéressant de disposer de domaines sur $\mathbb{Z}$ pour mener les calculs pendant les réductions de domaines. Ce petit détail n'a pas d'effet sur l'efficacité de l'implémentation (section 3.6.2).

[5] Mais d'autres suppositions seraient utiles. Voir section 3.5.3.

3.3.2 Opérations sur les domaines finis

Outre les opérations usuelles:

- Ajout ou destruction d'un élément.
- Test d'appartenance au domaine.
- Union, intersection et différence de domaines.

il sera utile de disposer d'opérations arithmétiques:

- Addition.
- Soustraction.
- Multiplication.

Une première définition de ces opérations aurait pu être la suivante:

Définition 3.1 (Opérations fortes sur les domaines) *Soit X et Y deux domaines sur $\mathbb{Z}$, soit $\top$ un opérateur arithmétique sur $\mathbb{Z}$ ($\top \in \{+, -, \times\}$), on pose:*

$$X \top Y = \{n \in \mathbb{Z} \mid \exists\, (x, y) \in\ X \times Y, n = x \top y\}$$

En fait, il est coûteux d'énumérer les domaines X et Y pour calculer $X \top Y$ si X et Y sont importants. Du fait de l'implémentation des domaines par des tableaux de bits, l'énumération est une opération peu efficace (mais déjà améliorée par des "énumérateurs" - cf section 3.6.2). On affaiblit alors la définition précédente.

L'idée est de travailler sur les intervalles complets si le coût de l'opération dépasse un certain seuil. Il convient donc de définir d'abord des opérations sur les intervalles.

Définition 3.2 (Opérations sur les intervalles) *Soit I et J deux intervalles de $\mathbb{Z}$, soit $\top$ un opérateur arithmétique sur $\mathbb{Z}$ ($\top \in \{+, -, \times\}$), on pose:*

$$I \top J = \text{ le plus petit intervalle contenant } \{n \in \mathbb{Z} \mid \exists\, (i, j) \in\ I \times J, n = i \top j\}$$

Ce qui évidemment s'implémente sans énumérer les intervalles; si $I = [i_1, i_2]_{\mathbb{Z}}$, $J = [j_1, j_2]_{\mathbb{Z}}$:

- $I + J = [i_1 + j_1, i_2 + j_2]_{\mathbb{Z}}$
- $I - J = [i_1 - j_2, i_2 - j_1]_{\mathbb{Z}}$
- $I \times J = [min(i_k j_l), max(i_k j_l)]_{\mathbb{Z}}$

On peut ensuite définir ces mêmes opérations sur les domaines.

Définition 3.3 (Intervalle d'un domaine fini) *Soit X un domaine fini, on désigne par $min(X)$ le plus petit élément de X, et par $max(X)$ son plus grand élément. On définit alors* l'intervalle de X *par:*

$$interv(X) = [min(X), max(X)]_{\mathbb{Z}}$$

Définition 3.4 (Opérations faibles sur les domaines finis) *Soit $\mathcal{D}(\mathbb{Z})$ l'ensemble des domaines finis sur $\mathbb{Z}$, soit $\top$ un opérateur arithmétique sur $\mathbb{Z}$ ($\top \in \{+,-,\times\}$), soit γ une fonction réelle associant à chaque domaine son coût d'énumération ($\mathcal{D}(\mathbb{Z}) \xrightarrow{\gamma} \mathbb{R}$), et $\Gamma \in \mathbb{R}$ un seuil, alors on définit* l'opérateur $\top$ sur $\mathcal{D}(\mathbb{Z})$ *par:*

$$\forall\, X, Y \in \mathcal{D}(\mathbb{Z})^2, X \top Y = \begin{cases} \{n \in \mathbb{Z} \mid \exists\, (x,y) \in\ X \times Y, n = x \top y\} \text{ si } \gamma(X)\gamma(Y) < \Gamma. \\ interv(X) \top interv(Y) \text{ sinon.} \end{cases}$$

Dans la suite, on fera, sauf mention contraire, toujours référence aux opérations faibles sur les domaines.

3.4 Contraintes algébriques

CLP++ peut traiter les contraintes d'égalité, inégalité ou non-égalité entre deux polynômes de $\mathcal{Q}_1(\mathbb{Z})$ c'est-à-dire de degré partiel maximum un en chacune des variables et à coefficients entiers relatifs (ex: `X*Y+2*Z` mais pas `X*X+1`). Ces contraintes sont donc du type $P \top Q$ avec $\top \in \{=, <, \leq, \neq\}$ et $P, Q \in\ \mathcal{Q}_1(\mathbb{Z})$.

Muni des opérations faibles sur les domaines, il est facile d'implémenter ces contraintes.

3.4.1 Domaines d'un polynôme

Définition 3.5 (Domaine d'un polynôme) *Soit $P \in \mathcal{Q}_1(\mathbb{Z})$, on appelle* domaine de P *le domaine fini $domain(P) \in \mathcal{D}(\mathbb{Z})$ défini par:*

- $domain(z) = \{z\}\ \forall\, z \in\ \mathbb{Z}$.
- $domain(X) =$ *son domaine de variation* $\in \mathcal{D}(\mathbb{N}) \subset \mathcal{D}(\mathbb{Z})$ *si X est une variable*[6].
- $domain(P \top Q) = domain(P) \top domain(Q)\ \forall\, P, Q \in \mathcal{Q}_1(\mathbb{Z}), \top\ \in \{+,-,\times\}$

Définition 3.6 (Domaines d'un polynôme par rapport à une variable) *Soit $P \in \mathcal{Q}_1(\mathbb{Z})$, soit X une variable (non nécessairement dans P), alors P s'écrit de manière unique $P = X.Q + R$ où $Q, R \in \mathcal{Q}_1(\mathbb{Z})$ et Q, R ne contiennent pas X. On appelle* domaines de P par rapport à X *les deux domaines $Qdomain(P,X)$ et $Rdomain(P,X)$ définis par:*

- $Qdomain(P,X) = domain(Q)$.
- $Rdomain(P,X) = domain(R)$.

3.4.2 Intervalles d'un polynôme

On définit de la même façon, en prenant pour une variable son intervalle de variation au lieu de son domaine:

- $interv(P)$.
- $Qinterv(P,X)$
- $Rinterv(P,X)$

[6] *Toutes les variables du polynôme sont des variables du problème, toutes déjà instanciées à une variable de domaine.*

3.4.3 Rôle passif

Soit la contrainte $(\mathcal{C})$ $P_1 \top P_2$ avec $P_1, P_2 \in \mathcal{P}_1(\mathbb{Z})$. On peut maintenant vérifier facilement si $(\mathcal{C})$ est vraie, fausse ou de valeur indéterminée:

1. Si $\top \equiv =$
   - Si $interv(P_1) - interv(P_2) = [0,0]_{\mathbb{Z}}$[7] alors $(\mathcal{C})$ est vraie[8].
   - Si $domain(P_1) \cap domain(P_2) = \emptyset$ alors $(\mathcal{C})$ est fausse.
   - Sinon $(\mathcal{C})$ est de valeur indéterminée.
2. Si $\top \equiv \neq$
   - Si $domain(P_1) \cap domain(P_2) = \emptyset$ alors $(\mathcal{C})$ est vraie.
   - Si $interv(P_1) - interv(P_2) = [0,0]_{\mathbb{Z}}$ alors $(\mathcal{C})$ est fausse.
   - Sinon $(\mathcal{C})$ est de valeur indéterminée.
3. Si $\top \equiv <$
   - Si $interv(P_2) - interv(P_1) \subset [1, +\infty[_{\mathbb{Z}}$ alors $(\mathcal{C})$ est vraie.
   - Si $interv(P_2) - interv(P_1) \subset]-\infty, 0]_{\mathbb{Z}}$ alors $(\mathcal{C})$ est fausse.
   - Sinon $(\mathcal{C})$ est de valeur indéterminée.
4. Si $\top \equiv \leq$
   - Si $interv(P_2) - interv(P_1) \subset [0, +\infty[_{\mathbb{Z}}$ alors $(\mathcal{C})$ est vraie.
   - Si $interv(P_2) - interv(P_1) \subset]-\infty, -1]_{\mathbb{Z}}$ alors $(\mathcal{C})$ est fausse.
   - Sinon $(\mathcal{C})$ est de valeur indéterminée.

L'utilisation des intervalles au lieu des domaines d'affaiblit pas la sémantique, mais accélère les calculs. Lorsque les domaines peuvent être utilisés, le comportement passif est plus puissant que celui qui existe en CHIP (sauf si $\Gamma = 0$, cf sections 3.4.5 et 3.4.6).

3.4.4 Rôle actif

Le contraintes algébriques ont un rôle actif de réduction de domaine. Soit toujours la contrainte $(\mathcal{C})$ $P_1 \top P_2$ avec $P_1, P_2 \in \mathcal{Q}_1(\mathbb{Z})$. Soit X une variable de P_1 ou de P_2. On pose:

- $Qd = Qdomain(P_1, X) - Qdomain(P_2, X)$.
- $Rd = Rdomain(P_2, X) - Rdomain(P_1, X)$.
- $Qi = Qinterv(P_1, X) - Qinterv(P_2, X)$.
- $Ri = Rinterv(P_2, X) - Rinterv(P_1, X)$.

Suivant $\top$, la réduction de domaine se fera par:

---

[7] Ce qui signifie que les intervalles sont égaux **et réduits à un élément**.

[8] Il ne s'agit pas de l'affaiblissement d'une éventuelle notion utilisant les domaines: avec $domain(P_1) - domain(P_2) = \{0\}$ la sémantique est la même, et les calculs plus coûteux.

- Si $\top \equiv =$

$$\left\{ \begin{array}{l} \text{Si } \gamma(Qd)\gamma(Rd) < \Gamma \text{ alors } X = X \cap \{\frac{r}{q} \mid q \in Qd, r \in Rd, q \text{ divise } r\}. \\ \text{Sinon} \left\{ \begin{array}{l} min(Qi) > 0 \Rightarrow \ X = X \setminus [\ \lfloor \frac{max(Ri)}{min(Qi)} \rfloor + 1\ , +\infty[_{\mathbb{Z}} \\ min(Qi) < 0 \Rightarrow \ X = X \setminus]-\infty,\ \lceil \frac{max(Ri)}{min(Qi)} \rceil - 1\ ]_{\mathbb{Z}} \\ max(Qi) < 0 \Rightarrow \ X = X \setminus [\ \lfloor \frac{min(Ri)}{max(Qi)} \rfloor + 1\ , +\infty[_{\mathbb{Z}} \\ max(Qi) > 0 \Rightarrow \ X = X \setminus]-\infty,\ \lceil \frac{min(Ri)}{max(Qi)} \rceil - 1\ ]_{\mathbb{Z}} \end{array} \right. \end{array} \right. \quad (3.1)$$

- Si $\top \equiv \neq$

$$\left. \begin{array}{l} Qi = [q,q]_{\mathbb{Z}} \\ Ri = [r,r]_{\mathbb{Z}} \\ q \neq 0 \\ q \text{ divise } r \end{array} \right\} \Rightarrow X = X \setminus \{\frac{r}{q}\}$$

- Si $\top \equiv <$

$$\left\{ \begin{array}{l} min(Qi) > 0 \Rightarrow \ X = X \setminus [\ \lceil \frac{max(Ri)}{min(Qi)} \rceil\ , +\infty[_{\mathbb{Z}} \\ min(Qi) < 0 \Rightarrow \ X = X \setminus]-\infty,\ \lfloor \frac{max(Ri)}{min(Qi)} \rfloor\ ]_{\mathbb{Z}} \end{array} \right.$$

- Si $\top \equiv \leq$

$$\left\{ \begin{array}{l} min(Qi) > 0 \Rightarrow \ X = X \setminus [\ \lfloor \frac{max(Ri)}{min(Qi)} \rfloor + 1\ , +\infty[_{\mathbb{Z}} \\ min(Qi) < 0 \Rightarrow \ X = X \setminus]-\infty,\ \lceil \frac{max(Ri)}{min(Qi)} \rceil - 1\ ]_{\mathbb{Z}} \end{array} \right.$$

3.4.5 Rôle du seuil

On aurait pu concevoir des algorithmes de réduction de domaine ou de contrôle de vérité sur les contraintes algébriques utilisant les opérations fortes sur les domaines (définition 3.1). Il aurait fallu rajouter un mécanisme basculant sur des considérations d'intervalle quand les domaines deviennent trop importants. Le but du seuil Γ est en fait de réaliser cette bascule de manière automatique en cours d'algorithme.

3.4.6 Par rapport à CHIP

Si Γ est nul alors les contraintes se comportent comme en CHIP. Sinon, le comportement est effectivement plus puissant qu'en CHIP dans certains cas d'égalité et de non-égalité[9]:

- Les contraintes peuvent jouer un rôle actif et passif dès qu'elles sont *lookahead-checkable*, même l'égalité et la non-égalité. Par exemple:
  - $2X = 3 \Rightarrow Faux$
  - $\left. \begin{array}{l} X + Y \neq 2Z + 1 \\ X = \{1,3,5\} \\ Y = \{3,9\} \end{array} \right\} \Rightarrow Vrai$

[9]En fait dans tous les cas où les domaines sont utilisés à la place des intervalles.

- $\left.\begin{array}{l} 2X = Y \\ X = \{1,2,3\} \\ Y = \{1,2,3\} \end{array}\right\} \Rightarrow \left\{\begin{array}{l} X = \{1\} \\ Y = \{2\} \end{array}\right.$

- Les produits de variables sont possibles: les polynômes sont dans $\mathcal{Q}_1(\mathbb{Z})$ et non dans $\mathcal{P}_1(\mathbb{Z})$ (polynômes de degré *total* maximum un). Quand CHIP rencontre un produit de variables, il attend que l'une des deux soit instanciée pour pouvoir continuer[10]. Par exemple:

  - $\left.\begin{array}{l} XY = 13 \\ X = \{2,4,8\} \end{array}\right\} \Rightarrow Faux$

  - $\left.\begin{array}{l} XY = Z - 1 \\ X = \{2,3\} \\ Y = \{1,2\} \\ Z = \{2,3,5,6,7,8\} \end{array}\right\} \Rightarrow Z = \{3,5,7\}$

Toutefois, les temps de réponse ne sont pas toujours améliorés en raison du coût supplémentaire apporté par les calculs sur les domaines là où CHIP n'utilise que les intervalles. Quand la réduction de l'arbre de recherche est importante, CLP^{++} fait mieux que CHIP. Sinon il faut baisser Γ pour être aussi rapide que CHIP.

3.5 Opérateur cardinal

Il s'agit de la contrainte $\#(n, \langle C_1, \ldots, C_p \rangle)$ où les C_i sont des contraintes, et $n \in \mathbb{N}$, qui est vraie ssi:

n exactement des contraintes C_i sont vraies, les $p - n$ autres étant fausses.

3.5.1 Rôle passif

Soit t le nombre de contraintes C_i vraies, et f les nombre de contraintes fausses, alors on a les implications:

- $t > n$ ou $f > p - n \Rightarrow \#(n, \langle C_1, \ldots, C_p \rangle)$ est fausse.
- $t = n$ et $f = p - n \Rightarrow \#(n, \langle C_1, \ldots, C_p \rangle)$ est vraie.

3.5.2 Rôle actif

L'opérateur cardinal n'a pas un rôle actif de réduction de domaine, mais il rajoute des contraintes au système. Contraindre davantage un système est bien jouer un rôle actif dans sa résolution. Ainsi:

- Si $t = n$ alors $\#(n, \langle C_1, \ldots, C_p \rangle)$ est retirée du système et remplacée par la négation des contraintes C_i non encore décidées.
- Si $f = p - n$ alors $\#(n, \langle C_1, \ldots, C_p \rangle)$ est retirée du système et remplacée par les contraintes C_i non encore décidées.

[10] Voir cas de la multiplication section 3.7.4.

3.5.3 Notion de rôle stratégique

L'opérateur cardinal pourrait jouer un rôle supplémentaire qu'on pourrait qualifier de *rôle stratégique*. Il pourrait guider l'algorithme de résolution qui se contente pour l'instant de choisir une nouvelle variable à générer (*"labeling"*) quand la phase de propagation n'a pas suffi. Une autre stratégie possible serait d'injecter de nouvelles contraintes fournies par une contrainte de type opérateur cardinal. Par exemple:

- Si $t = n - 1$ alors l'une des contraintes encore indécidées est vraie et les autres fausses.
- Si $f = p - n - 1$ alors l'une des contraintes encore indécidées est fausse et les autres vraies.

On pourrait créer, soit un point de choix sur la valeur d'une variable, soit un point de choix sur la valeur d'un jeu de contraintes. Il suffirait comme d'habitude de commencer par le choix le plus restreint.

3.6 Implémentation

Le but de cette section n'est pas d'étudier en détail l'implémentation de CLP++ ni même de donner les informations nécessaires à une compréhension globale de sa programmation. Il s'agit plutôt de citer en vrac:

- Certains problèmes qui se sont posés et auxquels on ne pense pas forcément.
- Certaines solutions adoptées, ou certaines organisations hiérarchiques des classes C++.

Quelques passages de listing sont donnés, principalement extraits des fichiers d'entête du C++ décrivant la structure des classes, leur organisation et leurs méthodes.

Les passages omis sont marqués `/.../`, à ne pas confondre avec les `...` d'une déclaration de procédure avec nombre d'arguments variable.

3.6.1 Les intervalles

Pas de problème particulier ni de choix crucial à faire pour programmer les intervalles de $\mathbb{Z}$. Les opérations précisées par la définition 3.2 sont implémentées ainsi que l'union (Figure 3.1).

3.6.2 Les domaines finis

Les domaines finis sont implémentés à l'aide d'un tableau de bits (voir figures 3.2 et 3.3):

- Le premier bit du tableau correspond à un certain entier relatif (`origin`), les bits suivants aux entiers successifs. Un bit levé signifie l'appartenance de l'entier qui lui correspond.
- Ce tableau de bit est mémorisé sur des mots de quatre octets (entiers longs).
- Le nombre d'éléments dans le domaine est mémorisé pour:
  - Repérer un domaine vide.
  - Calculer facilement le coût d'énumération du domaine (voir ci-dessous).

09/11/92 interv.h Page 1/1

```
/.../

/*
 *  Intervalle
 */

class interval {

private:
    integer a,b;        // [a,b]

public:
    interval() /.../                        // Constructeur vide
    interval(integer x) /.../               // Constructeur [x,x]
    interval(integer x, integer y) /.../    // Constructeur [x,y]

    integer MinValue() const /.../      // Renvoie borne inf
    integer MaxValue() const /.../      // Renvoie borne sup

    int HasValue(integer i)             // Teste l'appartenance

    friend interval operator+(const interval &,const interval &);// Somme
    friend interval operator-(const interval &,const interval &);// Soustrac.
    friend interval operator|(const interval &,const interval &);// Union
    friend interval operator*(const interval &,const interval &);// Produit

    friend int operator==(const interval &x, const interval& y)  // Egalite
        /.../

    friend ostream& operator<<(ostream &s,const  interval& i)   // Impression
        /.../
};
```

Figure 3.1: interv.h (1/1)

Intersection, union, appartenance

L'appartenance est une opération à coût minime puisqu'il suffit de regarder si un bit est levé. Rappelons toutefois des règles de calcul trop souvent ignorées, et pourtant essentielles lorsqu'on travaille sur les bits:

- Petit rappel pour comprendre:
  - Décalage à gauche: `<<`
  - Décalage à droite: `>>`
  - Et arithmétique: `&`
  - Ou arithmétique: `|`
  - Ou exclusif arithmétique: `^`
  - Complément à deux: `~`
  - Modulo: `%`

09/11/92 domain.h Page 1/2

```
/.../
typedef unsigned long word; // Tableau de bits: par mots de 4 octets

                            // Un mot = 32 bits
#define wordShift    5      // Pour une division rapide: n/32 = n>>5
#define wordMask  0x1f      // Pour un modulo rapide:    n%32 = n&0x1f

/.../

/*
 *  Domaine
 */

class domain {

    friend class trail;         // Le trail accede aux domaines pour
                                // les memoriser rapidement
    friend class enumerator;    // Les enumerateurs de domaine doivent
                                // aussi etre rapides

private:
    integer cardinal;       // Nombre d'elements du domaine
    integer minValue;       // Le plus petit element
    integer maxValue;       // Le plus grand
    word *bitString;        // Tableau de bits
    int size;               // Nombre d'octets du tableau de bits
    integer origin;         // Entier relatif correspondant au 1er
                            // bit du 1er mot

public:
    domain() /.../                      // Constructeur domaine vide
    domain(integer i) /.../             // Constructeur domaine [i,i]
    domain(integer l,integer h) /.../   // Constructeur domaine [l,h]
    domain(const domain &d) /.../       // Constructeur-Assignation

    ~domain() /.../                     // Destucteur

    void operator=(const domain &d) /.../   // Assignation

    void DeleteValue(integer);      // Retire un element
    void DeleteOver(integer);       // Retire au dessus d'un entier
    void DeleteUnder(integer);      // Retire en dessous d'un entier

    int HasValue (integer) const;   // Teste l'appartance
    int IsVoid() const /.../        // Vide?
    integer Cardinal() const /.../  // Nombre d'elements

    float EnumCost() const /.../    // Cout de l'enumeration

    integer MinValue() const /.../  // Le plus petit
    integer MaxValue() const /.../  // Le plus grand
    interval Interval () const;     // L'intervalle le contenant

    void SetTo() /.../                      // Assigne au vide
    void SetTo(integer i) /.../             // Assigne a [i,i]
    void SetTo(integer l,integer h) /.../   // Assigne a [l,h]

    friend int operator==(const domain&,const domain &);    // Egalite
```

Figure 3.2: domain.h (1/2)

09/11/92 domain.h Page 2/2

```
        friend domain operator&(const domain &,const domain &); // Intersection
        friend domain operator|(const domain &,const domain &); // Union
        friend domain operator^(const domain &,const domain &); // Difference
        friend domain operator+(const domain &,const domain &); // Somme
        friend domain operator-(const domain &,const domain &); // Soustraction
        friend domain operator*(const domain &,const domain &); // Multiplication

        friend ostream& operator<<(ostream&,const domain&);     // Impression

    private:
        /.../
    };

    /.../

    /*
     *  Enumerateur de domaine
     */

    class enumerator{

    private:
        int i,i1;           // Indices dans le tableau de bits
        int bit;            // Le bit actuel
        integer origin;     // Entier du 1er bit du 1er mot
        word *bitString;    // Tableau de bit

    public:
        enumerator() {}                         // Constructeur
        integer Start(domain const &);          // Initialisation
        integer NextValue();                    // Renvoie l'element suivant
    };
    /.../
```

Figure 3.3: domain.h (2/2)

- Savoir si le bit n[11] de l'entier i est levé[12]:

```
if ( i & (1 << n) ) {
    /.../
}
```

- Lever le bit n de l'entier i:

```
i |= (1 << n) ;
```

- Baisser le bit n de l'entier i:

```
i &= ~(1 << n) ;
```

- Basculer le bit n de l'entier i:

[11] Bit 0 = bit de droit

[12] Souvent, puisque n varie entre 0 et 31, on mémorise les `(1 << n)` (puissances de 2) dans un tableau.

```
i ^= (1 << n) ;
```

- Diviser l'entier i par 32 (ne surtout pas faire `i/32;`) :

```
i >> 5;      // Shifter de n <=> Diviser par 2^n
```

- Prendre le modulo 32 de l'entier i (ne surtout pas faire `i%32;`) :

```
i & 0x1F;   // Prendre les n derniers bits <=> Modulo 2^n
```

L'union et l'intersection de domaines sont rapides, puisqu'elles se ramènent à un ET et un OU arith-métiques. Il faut toutefois mettre les bons mots les uns en face des autres (origines et longueurs des tableaux différentes). **Mais surtout il ne faut pas avoir à décaler les bits au sein des mots**. Pour cela il faut de prendre comme origine un multiple de 32.

Enumération de domaines finis

Pour les opérations faibles décrites par la définition 3.4, comme dans bien d'autres occasions, il est nécessaire d'énumérer un domaine, c'est-à-dire de générer tous ses éléments. **La structure en tableau de bits n'est pas du tout faite pour cela!** Un objet d'énumération a été créé pour essayer de remédier au problème. La méthode consiste à se promener de bit et bit, de mot en mot tout en détectant rapidement par des masques:

- Qu'il n'y a plus de bits levés dans le mot courant.
- Que le prochain mot est vide.

D'où un gain d'un facteur de 32 dans les zones vides!
L'utilisation de ces objets énumérateurs est alors très simple:

```
domain d;
/.../
integer i;
enumerator en;
/.../
for (i=en.start(d); i!=noMoreValue ; i=en.NextValue()) {
    /.../
}
```

3.6.3 Les listes

La structure de listes a du être implémentée en C++. Deux points à signaler (voir figures 3.4 et 3.5):

1. Une classe de base, liste de pointeurs. Des classes dérivées pour lesquelles les objets pointés sont d'un type spécifique. D'où des comportements spécialisés suivant les objets (entrée/sortie et autres).
2. Une utilisation des procédures à nombre d'arguments variable permettant des commandes du type:

09/11/92 list.h Page 1/2

```
/.../

#include <stdarg.h>     // Nombre d'argument variable

/.../

                        // Pour repasser les arguments variables a une
                        // sous-procedure (compiler-dependant!)
#ifdef ATT
#define arguments (va_list)(&va_alist)
#else
#ifdef BORLAND
#define arguments ...
#else
/.../
#endif
#endif

typedef void *pointer;


/*
 *  Liste de base
 */

class list {

protected:
    int size;               // Taille de la liste
    pointer *table;         // Vers les elements

public:
    list() /.../                        // Constructeur vide
    list(const list &x) /.../           // Constructeur-Assignation
```

Figure 3.4: list.h (1/2)

```
domainList l(d1,d2,d3);
/.../
l.append(domainList(d4,d5));
```

Notons à ce propos un manque dans la spécification ANSI de C (ou C++): une procédure peut recevoir des arguments en nombre variable (utilisation de `va_list`) mais ne peut les repasser facilement à une autre procédure (sauf manipulations tordues dépendant du compilateur, ce qui a été fait...)

3.6.4 Les expressions

"Expression" désigne ici toute entité syntaxique du type entier, variable, polynôme ou contrainte. Les mots-clés reconnus sont ceux qui correspondent aux contraintes décrites plus haut (plus la contrainte spécifique aux pentominos - voir section 3.7.6).

- Un mini-analyseur lexical est implémenté pour faciliter l'entrée des données. Mais rappelons que CLP^{++} n'est qu'une maquette. La syntaxe adoptée est donc de type préfixe (`*(X,2)`).

09/11/92 list.h Page 2/2

```
        list(pointer, ...);                // Constructeur-Liste

        virtual ~list();                   // Destructeur

        void operator=(const list &x) /.../ // Assignation

        int Size() const /.../                 // Renvoie la taille

        pointer& operator[](int i) const /.../  // Element (left-value)
        pointer Ith(int i) const /.../          // Element (right-value)

        void Add(pointer);                 // Ajout element
        void Delete(pointer);              // Retire un element
        void UniqAdd(pointer);             // Ajout element sans duplication
        void Append(const list &);         // Concatene une liste
        void UnAppend(const list &);       // Retire les elements
        void UniqAppend(const list &);     // Concatene sans duplication
        void CutString(char *s);           // Liste des champs d'une chaine
        int HasValue(pointer);             // Test d'appartenance

        friend ostream &operator<<(ostream &,const  list &);  // Impression

    protected:
                                // Impression d'un element
        virtual void outHandle(ostream & ,const pointer) const;

    /.../

    private:

    /.../

    };
```

Figure 3.5: list.h (2/2)

- C'est aussi à ce niveau que sont implémentés les calculs des domaines et intervalles d'un polynôme (sections 3.4.1 et 3.4.2).
- Remarquer la classe "liste d'expressions" dérivée de la classe liste.
- Petit problème rencontré en C++: on ne peut définir une union contenant une classe possédant un constructeur (ici, une liste). Ce qui se comprend, mais est embêtant (voir remarque dans le listing).

3.6.5 Les variables

Rien de spécial pour l'implémentation des variables. Elles sont désignées par un identificateur chaîne de caractères pour pouvoir les utiliser dans l'analyseur syntaxique, mais se manipulent bien comme des variable C++.

- Classe dérivée des domaines (puisque pour l'instant toutes les variables sont des variables de domaine).
- Un identificateur et une liste de contraintes à réveiller (voir figure 3.8).

```
/.../
#define tokenNb     8                                   // Nombre de mots-cles
#define TOKEN_CODES { tokMult, tokPlus, tokEqual ,      // Codes internes
                      tokLessOrEqual, tokLess, tokDiff,
                      tokCardinal, tokPento}
#define TOKEN_NAMES { "*", "+", "=" ,                   // Mots-cles
                      "<=", "<", "!=", "#" ,"Pento"}
/.../
/*
 *  Liste d'expression
 */
class expressionList:public list {

    friend class expression;    // No comment!

public:
    expressionList():list() /.../                   // Liste vide
    expressionList(expressionList &l):list(l) /.../ // Construc-assignation
    expressionList(expressionPointer x, ...);       // Constructeur-Liste

    ~expressionList();                              // Destructeur

    void operator=(const expressionList &);         // Assignation

    expressionPointer Ith(int i) const /.../        // Element (right-value)

    void Parse(variableList&,const string);         // Analyseur


    void outHandle(ostream &,const pointer) const;  // Impression element

private:
    /.../
};

enum exprToken TOKEN_CODES;
enum exprType  {exprValue, exprVariable, exprTree };    // Type d'expression

/*
 *  Expression
 */
class expression{

    friend class expressionList;    // No comment!

private:

    exprType type;              // Type d'expression
    // REM: une union sur les trois cas aurait ete bienvenue mais
    //      mais une union ne peut (et pour cause) contenir un type
    //      de classe avec constructeur (ici la liste des fils)
    integer val;                // Si exprValue: la variable
    variablePointer vr;         // Si exprVariable: la variable
    exprToken token;            // Si exprToken: le mot-cle
    expressionList l;           //               et les fils

public:
```

Figure 3.6: expr.h (1/2)

09/11/92 expr.h Page 2/2

```
        expression() /.../                      // Constructeur vide
        expression(integer i) /.../             // Constructeur valeur
        expression(variablePointer d) /.../     // Constructeur variable
        expression(const expression& e)/.../    // Constructeur-assignation

        expression(exprToken,expressionPointer, ...); // Constructeur-Liste

        virtual ~expression();          // Destructeur par defaut

        void Parse(variableList&,const string);                    // Analyseur

        void operator=(const expression &e) /.../                  // Assignation

        friend ostream &operator<<(ostream &,const expression &);  // Impression

        void FindVars(variableList &) const;            // Toutes les variables

        void Domain(domain&);                       // Resolution par les domaines
        void QRDomains(variablePointer, domain &,domain &);              // Idem
        void Interval(interval&);                   // Resolution par les intervalles
        void QRIntervals(variablePointer, interval &,interval &);        // Idem

        exprType Type() /.../                       // Renvoie le type
        integer Value() /.../                       // Renvoie la valeur
        variablePointer Variable() /.../            // Renvoie la variable
        expressionPointer Child(int i) const /.../  // Renvoie un fils
        int NbOfChild() const /.../                 // Nombre de fils
        constraint *BuildConstraint() const;        // Transforme en contrainte
        constraint *BuildNegatedConstraint() const; // Idem contrainte niee

    protected:
        /.../

    private:
        /.../
    };
```

Figure 3.7: expr.h (2/2)

3.6.6 Les contraintes

En ce qui concerne les contraintes (figure 3.9), une classe de base possède les comportements de preuve et d'action.

- Cette classe de base est elle même dérivée de la classe expression.
- Elle possède une méthode d'initialisation appelée par le solveur en début de recherche, et une méthode appelée à la fin de la résolution. Ces deux méthodes permettent à la contrainte de réaliser un travail interne qui lui est propre, puis de le défaire éventuellement si besoin est (désallocations, etc.) Elles sont utilisées par les contraintes de type opérateur cardinal (et par la contrainte pentomino).

Hiérarchie

Comme on peut le voir sur les figures 3.10 à 3.12:

09/11/92 var.h Page 1/1

```
/.../

/*
 *  Variable
 */

class variable:public domain {

private:
    constraintList constraints;         // Les contraintes a reveiller
    char ident[10];                      // Identificateur

public:
    variable():domain() /.../                       // Construc. vide
    variable(char *s):domain() /.../                // Construc. domaine vide
    variable(char *s, integer i):domain(i) /.../    // Construc. domaine [i,i]
    variable(char *s, integer l,integer h):domain(l,h)  // Cons. [l,h]
        /.../

    variable(const variable &v):domain(v) /.../     // Construc.-assignation

    void operator=(const variable& v) /.../         // Assignation

                                                    // Recherche les contraintes
    void UpdateConstraints(const constraintList &);

    char *Ident() const /.../                       // Renvoie l'identif.
    constraintList &Constraints() const             // Renvoie les contraintes

    friend ostream& operator<<(ostream &s,const variable &v) // Impression
        /.../
};

/.../
```

Figure 3.8: var.h (1/1)

- Les contraintes d'égalité, de non-égalité et d'inégalité sont dérivées d'une même classe "contrainte algébrique", elle même dérivée de la classe de base contrainte.
- La classe intermédiaire `algebricRelation` regroupe ce qui est commun aux quatres classes finales:
  - Une preuve se décomposant en deux méthodes: `vraie?` et `fausse?`
  - Une syntaxe similaire ($P \top Q$).
  - Une action du type réduction de domaines.
- La classe égalité est un peut plus complexe que les autres, sa réduction de domaines appelant une méthode spécifique, `SolveEquation`, correspondant à l'équation (3.1) de la page 34.
- La contrainte opérateur cardinal utilise les méthodes d'initialisation et de fermeture au cours desquelles elle crée les contraintes C_i (voir section 3.5) (et leurs négations, puisque c'est éventuellement elles qu'il faudra injecter au système). Pendant ses méthodes de preuve et d'action, elle gère ces listes de méthodes, et marque celles qui sont déjà vraies ou fausses:
  - Une contrainte peut appeler d'autres contraintes et chercher à les prouver.

09/11/92 constr.h Page 1/1

```
/.../

enum { activeState, asleepState,    // Etats d'un contrainte
       trueState , reservedState, trueReservedState, falseReservedState };
enum { activeOk, activeFailed,      // Retour de DoAct()
       activeTrue, activeAsleep };
enum { prooveFalse, prooveTrue,     // Retour de Proove()
       prooveDontKnow };

/.../

/*
 *  Contrainte de base
 */

class constraint:public expression{

protected:
    variableList vars;                    // Liste des variables
    char state;                           // Etat

public:
    constraint():expression() {}                        // Contructeur vide
    constraint(const constraint& c):expression(c) /.../ // Contruc-assign
            { vars=c.vars; }

    void operator=(const constraint &c) /.../           // Assignation

    constraint(const expression& e):expression(e) /.../ // Conversion(expr)

    variableList &Variables() const /.../               // Les variables
    void SetState(char s) /.../                         // Assigne l'etat
    char State() const /.../                            // Renvoie l'etat
    char *GetStatePointer() /.../           // Renvoie l'adresse de l'etat
                                            // (pour le trail)

    friend ostream&operator<<(ostream &,const constraint&); // Impression

    virtual void InitialPhase (clp&) /.../  // Phase initiale par defaut
    virtual void ClosePhase (clp&) {} /.../ // Phase finale par defaut
    virtual int DoProve(clp&) /.../         // Preuve par defaut
    virtual int DoAct(clp&) /.../           // Comportement actif

private:
    /.../
};

/.../
```

Figure 3.9: constr.h (1/1)

```
/.../

/*
 *  Classe de base
 */

class algebricRelation:public constraint {

protected:
    interval interv;    // Variable temporaire

public:
    algebricRelation(exprToken token,                   // Constructeur
        expressionPointer e1,expressionPointer e2)
        :constraint(token,e1,e2,NULL) /.../
    algebricRelation(const expression &e):constraint(e) // Construc-assign.
        /.../

    int DoAct(clp&);        // Comportement actif
    int DoProve(clp&);      // Comportement passif


private:

    virtual void ReduceDomain(variablePointer,domain&) /.../    // Reducteur
    virtual int IsTrue() /.../                                  // Vrai?
    virtual int IsFalse()/.../                                  // Faux?
    void VerifySyntax();            // Verifie la syntaxe si assignation
};

/*
 *  Egalite
 */

class equal:public algebricRelation {

public:
    equal(expressionPointer e1,expressionPointer e2)        // Contructeur
        :algebricRelation(tokEqual,e1,e2) /.../
    equal(const expression& e):algebricRelation(e) /.../    // Constr-Assign.

private:
    int IsTrue();                                   // Vrai?
    int IsFalse();                                  // Faux?
    void ReduceDomain(variablePointer,domain &);    // Reducteur

    int SolveEquation(domain&,domain&,domain&);     // Procedure privee
};

class diff:public algebricRelation {

public:
    diff(expressionPointer e1,expressionPointer e2)     // Constructeur
        :algebricRelation(tokDiff,e1,e2) /.../
    diff(const expression &e):algebricRelation(e) /.../ // Constr-Assign.

private:
    int IsTrue();                                   // Vrai?
```

Figure 3.10: algeb.h (1/2)

09/11/92 algeb.h Page 2/2

```
    int IsFalse();                                  // Faux?
    void ReduceDomain(variablePointer,domain &);    // Reducteur
};

/*
 *  Inferieur ou Egal
 */

class lessOrEqual:public algebricRelation {

public:
    lessOrEqual(expressionPointer e1,expressionPointer e2)  // Constructeur
        :algebricRelation(tokLessOrEqual,e1,e2) /.../
    lessOrEqual(const expression &e):algebricRelation(e)/.../// Constr.Assign.

private:
    int IsTrue();                                   // Vrai?
    int IsFalse();                                  // Faux?
    void ReduceDomain(variablePointer,domain &);    // Reducteur
};

/*
 *  Inferieur
 */

class less:public algebricRelation {

public:
    less(expressionPointer e1,expressionPointer e2)     // Constructeur
        :algebricRelation(tokLess,e1,e2) /.../
    less(const expression &e):algebricRelation(e) /.../ // Constr.Assign.

private:
    int IsTrue();                                   // Vrai?
    int IsFalse();                                  // Faux?
    void ReduceDomain(variablePointer,domain &);    // Reducteur
};
```

Figure 3.11: algeb.h (2/2)

– Une contrainte peut utiliser le trail pour mémoriser ce qu'elle veut. C'est ce que fait l'opérateur cardinal pour marquer les C_i vraies et fausses.

La figure 3.13 résume l'organisation des classes de contraintes.

3.6.7 Le solveur

Rien de spécial en ce qui concerne le solveur (figures 3.14 à 3.16):

- Une classe réalisant le trail. Rien de compliqué.

09/11/92 cardin.h Page 1/1

```
/.../

/*
 *  Operateur cardinal
 */

class cardinalOperator:public constraint {

private:
    int n;                              // Nombre de contraintes a verifier
    int p;                              // Nombre total de contraintes
    int trueOnes;                       // Nombre de contraintes deja verifiees
    int falseOnes;                      // Nombre de contraintes deja fausses
    constraintList constraints,         // Les contraintes annexes
                   negatedConstraints; // Et leur negation


public:
    cardinalOperator(const expression &e)          // Constructeur
        :constraint(e) /.../
    ~cardinalOperator() /.../                      // Destructeur

    void InitialPhase(clp&);        // Phase Initiale
    void ClosePhase(clp&);          // Phase de fermeture

    int DoProve(clp&);              // Comportement passif
    int DoAct(clp&);                // Comportement actif

private:
    void CreateConstraints();       // Cree les contraintes annexes
    void DeleteConstraints();       // Les detruit
};
```

Figure 3.12: cardin.h (1/1)

- Une classe contenant le problème à traiter (classe `clp`) possédant les méthodes de résolution, propagation, etc.[13]
- Une classe réalisant l'opération de labeling, choix des variables, etc.

3.7 Applications

Cette section présente quelques applications testées en CLP^{++} . Leur énoncé n'est pas toujours très "esthétique" puisque l'interface entre le langage et le solveur ne se fait que par l'intermédiaire du rudimentaire analyseur lexical. Ce qui signifie que la génération de contraintes par une procédure (ex: `AllDifferent()`) passe par la génération des chaînes de caractères. Encore une fois, le but de la maquette était de tester des algorithmes de résolution, et non de présenter un ensemble complet langage+solveur.

[13] Dont une méthode pour les problèmes d'ordonnancement: `DoPert()` (voir section 3.7.5).

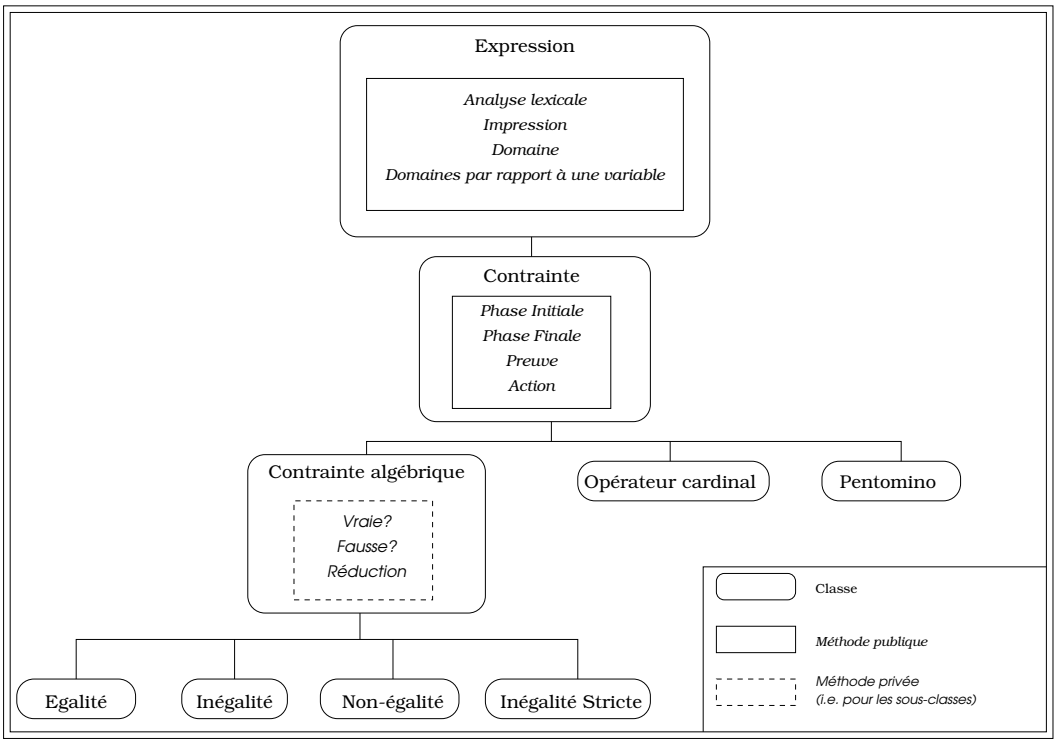

Figure 3.13: Organisation des contraintes

3.7.1 Send+More=Money

Le problème à résoudre est celui déjà présenté section 1.1.1. L'énoncé en CLP++ est présenté figure 3.17.

- Il est programmé sans les retenues: la contrainte

  $$(1000S+100E+10N+D)+(1000M+100O+10R+E) = 10000M+1000O+100N+10E+Y$$

  est directement injectée, ce qui ne gêne pas le solveur.

- La procédure `AllDifferent(pb,var_names)` ajoute au problème `pb` les contraintes imposant que toutes les variables de `var_names` sont différentes. Elle travaille comme annoncé avec des chaînes de caractères. Remarquer la procédure `CutString(s)` qui construit une liste dont les éléments sont les sous-chaînes de `s` séparées par des virgules.

Après une première propagation, les variables sont réduites à:

$$M = \{1\},\ O = \{0\},\ S = \{9\},\ E, N, D, R, Y = \{2, 3, 4, 5, 6, 7, 8\}$$

Le solveur crée un point de choix sur E. Pour chaque valeur de E, une nouvelle propagation suffit à découvrir l'échec ($E = 2, 3, 4, 6, 7, 8$) ou la solution $S, E, N, D, M, O, R, Y = 9, 5, 6, 7, 1, 0, 8, 2$ ($E = 5$). Soit au total un seul point de choix et six échecs. Rien n'est gagné sur CHIP. Réponse instantanée, évidemment.

09/11/92 trail.h Page 1/1

```
/*
 *  Trail
 */

class trail {

private:
    byte *stack;    // Pile
    long pc;        // Pointeur de pile
    long size;      // Taille de la pile (pour re-allocation)

public:
    trail()     /.../           // Contructeur
    ~trail()    /.../           // Destructeur

    void Init();                    // Initialisation
    void Close();                   // Fermeture et retablissement des valeurs
    void Memorize(domainPointer);   // Empile un domaine
    void Memorize(byte *);          // Empile un octet

private:
    /.../
    void RestoreDomain();           // Retablit un domaine
    void RestoreByte();             // Retablit un octet
};
```

Figure 3.14: trail.h (1/1)

3.7.2 Les huit reines

Le problème des huit reines se programme de façon analogue (figure 3.18). Toutes les solutions sont obtenues en une dizaine de secondes! Avec $\Gamma = 0$, 172 échecs. Avec $\Gamma \neq 0$, 155 échecs, ce qui n'est pas très inférieur...

3.7.3 Où est le zèbre?

Il s'agit du célèbre problème posé en ces termes:

Cinq hommes de nationalités différentes vivent dans cinq maisons voisines. Ils exercent cinq professions différentes, ont chacun un animal familier différent, une boisson préférée différente. Les maisons sont aussi de couleurs différentes. Sachant que:

1. L'Anglais vit dans la maison rouge.
2. L'Espagnol a un chien.
3. Le Japonais est peintre.
4. L'Italien boit du thé.
5. Le Norvégien vit dans la maison de gauche.
6. L'habitant de la maison verte boit du café.
7. La maison verte est à droite de la maison blanche.

09/11/92 solve.h Page 1/2

```
/*
 *  Classe pour generer les choix ("labeling")
 */

class label {

private:
    variablePointer labeled_var;    // La variable choisie
    domain labeledDomain;           // Son domaine explore'
    enumerator enm;                 // Pour enumerer le domaine
    /.../

public:
    void Init(variablePointer);     // Initialise
    int NextValue(clp&,int);        // Valeur suivante
};

/*
 *  Classe du probleme a resoudre
```

Figure 3.15: solve.h (1/2)

8. Le sculpteur élève des escargots.
9. Le diplomate vit dans la maison jaune.
10. On boit du lait dans la maison du milieu.
11. Le Norvégien habite à côté de la maison bleue.
12. Le violoniste boit du jus de fruits.
13. Il y a un renard à côté de la maison du docteur.
14. Il y a un cheval à côté de la maison du diplomate.

La question est:

"Qui boit de l'eau et qui possède un zèbre?"

La solution est:

	1	2	3	4	5
Nationalité	Norvégien	Italien	Anglais	Espagnol	Japonais
Animal	Renard	Cheval	Escargot	Chien	Zèbre
Couleur	Jaune	Bleu	Rouge	Blanc	Vert
Boisson	Eau	Thé	Lait	Jus	Café
Métier	Diplomate	Docteur	Sculpteur	Violoniste	Peintre

Simple à résoudre en Prolog[14], ce problème pose en général des difficultés aux systèmes de contraintes puisqu'il contient des disjonctions (la proposition 'voisin de' se traduit par un 'ou').

[14] Mais bien moins rapidement

```
 */

class clp {

    friend class label;
    // Les classes de contraintes doivent etre amies ...
    friend class constraint;
    friend class algebricRelation;
    friend class cardinalOperator;
    friend class pentomino;

private:
    constraintList constraints;     // Les contraintes
    variableList vars;              // Les variables
    int nbOfChoices;                // Compteurs
    int nbOfFails;                  //  ""
    int nbOfSols;                   //  ""
    trail *trl;                     // Le trail actuel
    /.../

public:
    clp() /.../         // Constructeur
    ~clp();             // Destructeur

    void AddVariables(char *,integer,integer);  // Ajoute des variables
    void AddConstraints(char *);                // Ajoute des contraintes

    void DoPert(char *);    // Resout en ordonnancement
    void DoSolve();         // Recherche les solutions

    variablePointer Variable(char*);            // Renvoie une variable
    void SetVariable(char *,integer);           // Assigne une variable [i,i]
    void SetVariable(char *,integer,integer);   // Assigne une variable [l,h]

    friend ostream &operator<<(ostream &,const clp &);  // Impression

private:
    void SetDomainTo(domainPointer d) /.../               // Change une variable
    void SetDomainTo(domainPointer d,integer i) /.../           // Idem
    void SetDomainTo(domainPointer d,integer l,integer h) /.../ // Idem
    void SetDomainTo(domainPointer d,const domain &e)     /.../ // Idem

    void SetState(constraintPointer c,char st)  // Change l'etat d'une contrainte
        /.../

    void InitConstraints();         // Initialise les contraintes
    void close_constraints();       // Referme les contraintes


    int DoPropagate();              // Propagation

    void InitVariableLabeling();            // Initialise le labeling
    int ChooseNextVar(variablePointer &);   // Choisit la variable a enumerer
    void Choose(int );                      // Enumere la variable

    /.../

};
```

Figure 3.16: solve.h (2/2)

09/11/92 sendmory.c Page 1/1

```
void AllDifferent(clp &pb, char *var_names)
{
    list vars;
    string dif;

    vars.CutString(var_names);

    for (int i=0; i<vars.Size()-1; i++)
        for (int j=i+1; j<vars.Size(); j++) {
            sprintf(dif,"!=(%s,%s)",vars[i],vars[j]);
            pb.AddConstraints(dif);
        }
}


void Sendmory()
{
    clp sendmory;

    sendmory.AddVariables("S,E,N,D,M,O,R,Y",0,9);

    sendmory.AddConstraints("!=(M,0),
                             !=(S,0)");

    AllDifferent(sendmory,"S,E,N,D,M,O,R,Y"));

    sendmory.AddConstraints(
         "=(
            +(*(1000,S),*(100,E),*(10,N),D,*(1000,M),*(100,O),*(10,R),E),
            +(*(10000,M),*(1000,O),*(100,N),*(10,E),Y))");
                      );

    sendmory.DoSolve();
}
```

Figure 3.17: sendmory.c (1/1)

Avec l'opérateur cardinal, la disjonction[15] est un problème résolu (voir figure 3.19).

$$C_1 \text{ ou } C_2 \;\Leftrightarrow\; \#(1, \langle C_1, C_2 \rangle)$$

La réponse est encore instantanée. Avec $\Gamma = 0$, 11 suppositions sont faites. Avec $\Gamma \neq 0$, 8 seulement.

3.7.4 Multiplication

Ce problème est considéré comme le cryptogramme arithmétique le plus difficile[16] par [Lau78]. Il s'agit de résoudre la multiplication:

[15] Il s'agit ici d'un 'ou' exclusif. Mais un 'ou' inclusif se traite avec l'opérateur cardinal un peu plus général: "Entre n et p contraintes sont vraies".

[16] Il faut bien plusieurs jours à la main...

09/11/92 queens.c Page 1/1

```
void NoDiagonalAttack(clp &pb, char *var_names,int inc)
{
    list vars;
    string dif;

    vars.CutString(var_names);

    for (int i=0; i<vars.Size()-1; i++)
        for (int j=i+1; j<vars.Size(); j++) {
            sprintf(dif,"!=(+(%s,%d),%s)",vars[i],inc*(i-j),vars[j]);
            pb.AddContraints(dif);
        }
}

void Queens()
{
    clp queens;
    char *vars="X1,X2,X3,X4,X5,X6,X7,X8";

    queens.AddVariables(vars,1,8);

    AllDifferent(queens,vars);
    NoDiagonalAttack(queens,vars,1);
    NoDiagonalAttack(queens,vars,-1));

    queens.DoSolve();
}
```

Figure 3.18: queens.c (1/1)

$$
\begin{array}{cccccc}
 & & . & . & . & \\
 & \times & . & . & . & \\
\hline
 & & . & . & . & \\
 & . & . & . & & \\
. & . & . & & & \\
\hline
. & . & . & . & . &
\end{array}
$$

où chacun des points représente un chiffre de 0 à 9, et où au total il apparaît exactement chaque chiffre deux fois.

Initialement impossible à programmer en CHIP du fait de l'impossibilité d'exprimer la contrainte: "exactement deux fois chaque chiffre", cette multiplication est très simple à énoncer en ALICE (nombre d'antécédents = 2). L'opérateur de cardinalité sauve CHIP pour ce qui est de la possibilité de poser le problème. Mais les contraintes contiennent des multiplications de variables et il est probable que CHIP ne s'en sorte pas très bien.

En CLP^{++} , le programme de la figure 3.20 trouve la solution:

$$
\begin{array}{cccccc}
 & & 1 & 7 & 9 & \\
 & \times & 2 & 2 & 4 & \\
\hline
 & & 7 & 1 & 6 & \\
 & 3 & 5 & 8 & & \\
3 & 5 & 8 & & & \\
\hline
4 & 0 & 0 & 9 & 6 &
\end{array}
$$

09/11/92 zebra.c Page 1/1

```
void Zebra()
{
    clp zebra;

    zebra.AddVariables("N1,N2,N3,N4,N5,
                        P1,P2,P3,P4,P5,
                        C1,C2,C3,C4,C5,
                        D1,D2,D3,D4,D5,
                        A1,A2,A3,A4,A5",1,5);

    AllDifferent(zebra,"N1,N2,N3,N4,N5"));
    AllDifferent(zebra,"P1,P2,P3,P4,P5"));
    AllDifferent(zebra,"C1,C2,C3,C4,C5"));
    AllDifferent(zebra,"D1,D2,D3,D4,D5"));
    AllDifferent(zebra,"A1,A2,A3,A4,A5"));

    zebra.AddConstraints("=(N1,C2),
                          =(N2,A1),
                          =(P1,N3),
                          =(D3,N4),
                          =(N5,1),
                          =(C1,D4),
                          =(C1,+(C5,1)),
                          =(P5,A4),
                          =(P2,C3),
                          =(D5,3),
                          #(1,=(N5,+(C4,-1)),=(N5,+(C4,1))),
                          =(P3,D1),
                          #(1,=(P4,+(A3,-1)),=(P4,+(A3,1))),
                          #(1,=(P2,+(A5,-1)),=(P2,+(A5,1)))");

    zebra.DoSolve();
}
```

Figure 3.19: zebra.c (1/1)

en moins d'une minute[17], et prouve son unicité en douze minutes. (On a ici une idée de la puissance d'ALICE qui annonce sur une machine de 1978 un temps de quatre minutes!).

3.7.5 Ordonnancement

Cet exemple montre comment on peut utiliser le schéma de CLP++ , et notamment le mécanisme de propagation des contraintes pour résoudre certains problèmes d'ordonnancement. Ici, il s'agit

[17] sur une Sun SparcStation 2

09/11/92 multiply.c Page 1/1

```
char *AllEqualTo(char *var_names,integer x)
{
    static string allEq;
    string eq;
    list vars;

    vars.CutString(var_names);
    for (int i=0; i<vars.Size(); i++) {
        sprintf(eq,",=(%s,%ld)",vars[i],x);
        strcat(allEq,eq);
    }
    return(allEq);
}

void N_EqualTo(clp &pb,char *var_names,integer n,integer x)
{
    string card;

    sprintf(card,"#(%ld %s)",n,AllEqualTo(var_names,x));
    pb.AddConstraints(card);
}

void Multiply()
{
    clp mult;
    char *vars="A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T";

    mult.AddVariables(vars,0,9);

    mult.AddConstraints("!=(A,0),
                         !=(D,0),
                         !=(G,0),
                         !=(J,0),
                         !=(M,0)");

    for (integer i=0;i<=9;i++)
        N_EqualTo(mult,vars,2,i));

    mult.AddConstraints("=(+(*(100,A,F),*(10,B,F),*(C,F)),
                           +(*(100,G),*(10,H),I)),
                        =(+(*(100,A,E),*(10,B,E),*(C,E)),
                           +(*(100,J),*(10,K),L)),
                        =(+(*(100,A,D),*(10,B,D),*(C,D)),
                           +(*(100,M),*(10,N),O)),
                        =(+(*(10000,M),*(1000,N),*(100,O),
                            *(1000,J),*(100,K),*(10,L),*(100,G),*(10,H),I),
                           +(*(10000,P),*(1000,Q),*(100,R),*(10,S),T))");
    mult.DoSolve();
}
```

Figure 3.20: multiply.c (1/1)

09/11/92 pert.c Page 1/1

```
void Pert()
{
    clp pert;

    pert.AddVariables("A,B,C,D,E,F,G,H,J,K,S",0,28);

    pert.AddConstraints("<=(+(A,7),B),
                         <=(+(A,7),D),
                         <=(+(B,3),C),
                         <=(+(C,1),E),
                         <=(+(D,8),E),
                         <=(+(C,1),G),
                         <=(+(D,8),G),
                         <=(+(D,8),F),
                         <=(+(C,1),F),
                         <=(+(F,1),H),
                         <=(+(H,2),J),
                         <=(+(G,1),K),
                         <=(+(E,2),K),
                         <=(+(J,1),K),
                         <=(+(K,1),S)");
    pb.DoPert("S");
}
```

Figure 3.21: pert.c (1/1)

d'un petit problème se ramenant à:

$$min(S) \text{ pour } \begin{cases} A+7 \leq B \\ A+7 \leq D \\ B+3 \leq C \\ C+1 \leq E \\ D+8 \leq E \\ C+1 \leq G \\ D+8 \leq G \\ C+1 \leq F \\ F+1 \leq H \\ H+2 \leq J \\ G+1 \leq K \\ E+2 \leq K \\ J+1 \leq K \\ K+1 \leq S \end{cases}$$

La méthode `DoPert(EndVar)` de la classe `clp` résout ce type de problèmes. Son principe est le suivant[18]:

- ◇ Propager les contraintes.
- ◇ Assigner `EndVar` à sa plus petite valeur possible

[18] En fait, une méthode plus complexe permettant de résoudre des problèmes un peu plus généraux (avec disjonctions) a été implémentée.

◇ Repropager les contraintes.
◇ Afficher les variables et les contraintes non encore résolues.

Pour notre exemple, programmé figure 3.21, la première propagation donne:

$$\begin{cases} A = [0,8],\ B = [7,19],\ C = [10,22],\ D = [7,15],\ E = [15,27],\ F = [15,23] \\ G = [15,28],\ H = [16,24],\ J = [19,27],\ K = [21,29],\ S = [22,30] \end{cases}$$

puis, après avoir forcé $S = 22$, la deuxième propagation aboutit à:

$$\begin{cases} A = 0,\ B = [7,11],\ C = [10,14],\ D = 7,\ E = [15,19],\ F = 15 \\ G = [15,20],\ H = 16,\ J = 19,\ K = 21,\ S = 22 \\ B + 3 \leq C \end{cases}$$

qui est bien la solution. Certaines variables peuvent encore varier librement dans un domaine restreint. D'autres, comme B et Cne peuvent varier qu'en respectant une certaine contrainte: $B + 3 \leq C$

3.7.6 Pentominos

Enfin, CLP^{++} a été utilisé pour résoudre le problème consistant à placer dans une boite (20×3 par exemple) les douze pentominos (assemblages plans de cinq cubes). Le problème a d'abord été traduit en contraintes algébriques. Le résultat obtenu a été catastrophique (trop de contraintes, arbre de recherche trop profond), ce qui n'a pas été une surprise. Puis la création d'une contrainte spécialisée a résolu le problème. Cette nouvelle contrainte:

- travaille avec une variable par pièce sur un domaine $[0, 60 \times 4 \times 2 - 1]_{\mathbb{N}}$ codant:
  - La position de la pièce (60 cases)
  - Son orientation dans le plan (4 choix)
  - Son retournement dans l'espace (2 choix)
- Réduit trivialement les domaines des pièces possédant des symétries.
- Crée ses propres structures de raisonnement (tableau représentant la boite, etc.)
- Possède une phase active de réduction de domaine.
- Possède une phase passive de vérification utilisant les critères suivants:
  - Les pièces ne débordent pas de la boite.
  - Elles ne se chevauchent pas.
  - La taille des composantes connexes de l'ensemble des cases encore vides doit être multiple de cinq. (difficile à coder avec des contraintes usuelles!) Ce critère est évidemment essentiel.

Si CLP^{++} est vu comme une bibliothèque de mécanismes et de contraintes mettant à la disposition du programmeur le moyen de créer de nouvelles contraintes, l'ajout d'une contrainte *pentominos* est cohérent avec la philosophie du système. Finalement, dans ce cas extrême, le programmeur profite de CLP^{++} pour ses mécanismes de propagation et de recherche avec backtrack.

On trouve rapidement les deux solutions figure 3.22, plus les symétriques.

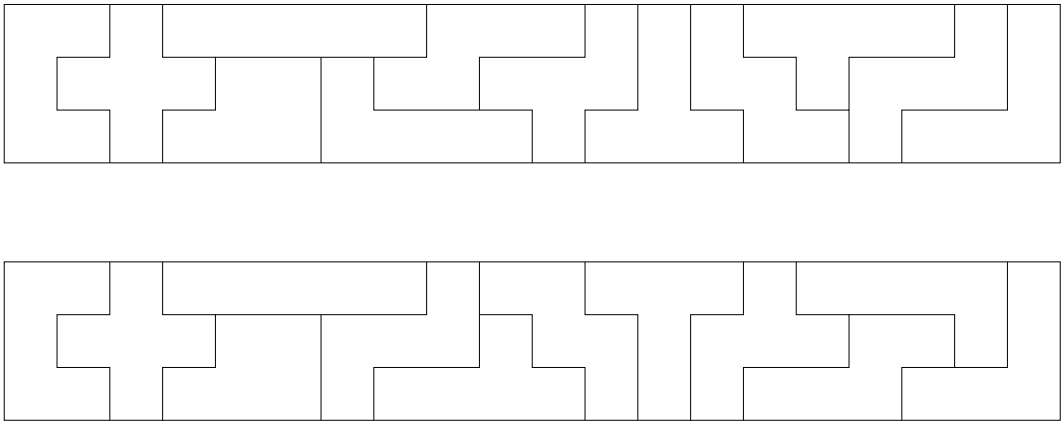

Figure 3.22: Solutions du problème des pentominos

3.8 Conclusion

Les défauts de CLP++ sont nombreux, et le but n'était pas de rivaliser avec ALICE et CHIP. On peut tirer plusieurs enseignements de la réalisation de cette maquette:

- Le traitement des contraintes algébriques peut être vu sous un jour différent en introduisant des opérateur sur les domaines. Les réductions de domaines sont alors plus fines, mais parfois trop coûteuses. Il en reste que c'est sans perte d'efficacité que l'on peut traiter les contraintes entre polynômes de $\mathcal{Q}_1(\mathbb{Z})$, sans se limiter comme CHIP aux polynômes de $\mathcal{P}_1(\mathbb{Z})$.
- Si la Programmation Logique intègre naturellement la Programmation par Contraintes dans son schéma ([JL87]), on peut très bien utiliser les contraintes comme un simple outil de résolution de problèmes. Dans cette optique, la réalisation d'une bibliothèque C++ de résolution de problèmes peut très bien se justifier, tant du point de vue de l'utilité que du point de vue théorique. Il n'est d'ailleurs pas exclus qu'un langage tel que CHIP ne soit en pratique utilisé que comme un solveur autour duquel on développe des applications, c'est-à-dire ni plus ni moins qu'un Prolog muni d'une bibliothèque de résolution. Pourquoi alors ne pas disposer d'un langage impératif similaire?
- Il reste encore du travail si l'on veut pouvoir décrire facilement les problèmes en C++. Il n'est pas impossible que la syntaxe de C++ ne suffise pas. Il faudrait alors, soit concevoir CLP++ comme une couche supplémentaire et réaliser un traducteur de CLP++ vers C++, soit retoucher directement le compilateur C++. C'est le même choix qui s'est posé lors du passage de C à C++.
- Bien des problèmes usuels nécessitent l'utilisation d'algorithmes spécifiques et ne peuvent se traiter à l'aide de contraintes.

En résumé, nous avons:

1. introduit le concept d'opération sur les domaines permettant de traiter simplement les réductions de domaines pour des contraintes algébriques, notamment des opérations dites faibles assorties d'un seuil réglant leur efficacité.
2. constaté que les contraintes entre polynômes de $\mathcal{Q}_1(\mathbb{Z})$ se traitaient avec notre méthode aussi simplement que dans $\mathcal{P}_1(\mathbb{Z})$ — cas auquel CHIP se restreint.

3. étendu la notion de rôle actif des contraintes qui ne consiste plus uniquement en une diminution du domaine des variables. Dans sa phase active, une contrainte peut aussi générer de nouvelles contraintes.

4. introduit — sans toutefois implémenter — une notion de rôle stratégique pour une contrainte. Une contrainte peut proposer un ensemble de contraintes possibles dont une seule est vraie. Un point de choix de l'arbre de recherche consiste alors:

   - soit à énumérer une variable de domaine (stratégie habituelle)
   - soit à supposer vraie une des contraintes possibles (et fausses les autres).

5. implémenté les contraintes comme une bibliothèque d'objets à partir desquels il est aisé de programmer de nouvelles classes de contraintes. En particulier, une nouvelle classe:

   - profite des comportements de la classe dont elle dérive.
   - peut faire appel à d'autres contraintes. Ainsi l'opérateur cardinal interroge-t-il ses sous contraintes en utilisant des méthodes a priori conçues pour être appelées par le solveur!
   - peut créer s'il le faut son propre trail mais aussi utiliser le trail courant du solveur pour mémoriser un état qui sera automatiquement rétabli au backtrack.

Table des Matières

Références

[JL87] J. Jaffar and J. L. Lassez. Constraint logic programming. In *14th ACM Symp. on Principles of Programming Languages*, pages 111–119, Munich, January 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In *4th International Conference on Logic Programming*, pages 197–217, Melbourne, Australia, May 1987.

[Lau78] J.L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, September 1978.

[Van89] Pascal VanHentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[VY91] P. VanHentenryck and Y.Deville. The cardinality operator: a new logical connective for constraint logic programming. In *8th International Conference on Logic Programming*, pages 745–759, Paris, France, June 1991.