Prolog IV : langage et algorithmes

Frédéric Benhamou
Laboratoire d'Informatique Fondamentale d'Orléans
I.I.I.A., Université d'Orléans, Rue Léonard de Vinci
B.P. 6759 45067 ORLEANS Cedex 2 France
<benhamou@lifo.univ-orleans.fr>

Touraïvane
PrologIA
Parc Technologique de Luminy - case 919
13288 Marseille Cedex 09 France
<touraivane@prologianet.univ-mrs.fr>

Résumé

Dans ce papier nous présentons les principales caractéristiques du langage Prolog IV, qui succède au langage Prolog III. Prolog IV est un langage de programmation logique par contraintes qui introduit au sein d'un modèle unique des contraintes sur les arbres, les listes, les réels, les rationnels, les entiers et les Booléens. Les algorithmes qui servent de base à la résolution de contraintes s'appuient sur un algorithme naïf sur les listes différent de celui de Prolog III, des algorithmes de type Gauss et Simplex pour le traitement des équations, diséquations et inéquations linéaires sur les rationnels, et un algorithme d'approximation fondé sur l'arithmétique des intervalles pour les contraintes non-linéaires sur les réels, les contraintes sur les entiers bornés, et les contraintes booléennes. Enfin, Prolog IV est défini comme un Prolog aux normes ISO, les contraintes étant introduites grâce à des prédicats prédéfinis. Les prétentions de ce papier se limitent à présenter un premier apperçu informel mais complet du langage, de ses possibilités et des algorithmes de résolution de contraintes sous-jacents.

1 Introduction

Le langage Prolog III [8] fut avec CLP($\Re$) [14, 15] et CHIP [13, 28] l'un des pionniers des langages de programmation logique par contraintes[1] de ce qu'il convient d'appeler première génération.

L'utilisation de ces langages pour la réalisation d'applications, ainsi que l'émergence de nouveaux concepts, dont l'un des plus remarquable est peut être l'introduction de l'arithmétique des intervalles en CLP, ont permis de mieux cerner les domaines de contraintes

[1] Il n'est pas ici dans nos intentions de développer les concepts liés à l'introduction de la programmation logique par contraintes. Le lecteur pourra se reporter aux travaux fondateurs [14, 8, 13] ou à des présentations générales plus récentes (par exemple [16]).

(et les algorithmes pour les résoudre) à la fois utiles en résolution de problèmes continus et discrets, tout en restant efficaces. Le langage Prolog IV est une tentative pour intégrer ces considérations au sein d'un langage uniforme muni d'une sémantique clairement définie.

Le reste de cet article est organisé de la manière suivante : la section 2 présente rapidement un certain nombre de généralités sur le langage, le modèle théorique sous-jacent, sa syntaxe et sa sémantique. Dans les sections suivantes sont abordés successivement les contraintes sur les listes, les contraintes linéaires sur les nombres rationnels et les contraintes sur les réels, les entiers et les booléens traitées à l'aide de l'arithmétique des intervalles. Enfin, nous présentons quelques exemples de programmes avant de conclure.

2 Principes généraux

Nous ne rendrons pas compte ici dans les détails de la sémantique du langage, décrite par Alain Colmerauer dans [12]. Nous précisons simplement les quelques points fondamentaux qui donnent une idée générale du modèle théorique global et de la sémantique de Prolog IV.

2.1 Définitions

Le langage Prolog IV est défini à partir de trois ensembles de symboles, un ensemble de variables dont la syntaxe est définie par la norme Prolog ISO, un ensemble de symboles de fonctions et un ensemble de symboles de prédicats. Les termes sont définis de la manière habituelle, les contraintes sont définies à partir des symboles de prédicats, du symbole d'égalité, ainsi que des connecteurs logiques et des quantificateurs usuels. Une *structure* Σ est définie comme une application qui associe à chaque symbole fonctionnel ou de prédicat une fonction ou une relation sur un domaine noté D_Σ.

Soit Σ une structure et σ une application de l'ensemble des variables V dans D_Σ. Tout terme autorisé dans Σ peut alors être interprété comme un élément de D_Σ et toute contrainte p autorisée dans Σ peut alors être interprétée comme un élément de $\{true, false\}$ par une fonction d'*interprétation*.

Un programme p est une conjonction de la proposition $true$ et de propositions du type $\forall x_1 \ldots \forall x_n, p_1 \wedge \ldots \wedge p_m \Rightarrow p_0$, où p_0 ne doit pas être un symbole de prédicat réservé, appelées par abus de langage *clauses*. On a alors la propriété suivante :

Propriété 1 ([12]) *L'ensemble des structures qui satisfont p n'est pas vide et admet un plus petit élément.*

Sémantiquement, un programme p peut donc être vu comme la définition de la plus petite structure qui satisfait p et qui complète Σ.

2.2 Domaine

Le domaine de Prolog IV est l'ensemble des arbres rationnels étiquetés par des identificateurs (au sens de la norme ISO) et par des réels (au sens mathématique). Les étiquettes numériques sont interdites pour les nœuds ayant un ou plusieurs fils.

2.3 Opérations et relations

Les seules opérations définies en Prolog IV sont les *opérations de construction d'arbre* qui, pour tout symbole de fonction f d'arité n associent à toute suite d'arbre $(a_1, \ldots, a_n)$ l'arbre étiqueté par f dont la suite des fils est $(a_1, \ldots, a_n)$. Les relations de type $\Sigma(p)$ où p est un symbole de prédicat prédéfini sont définies extensivement dans [27]. Nous en donnons ici quelques exemples significatifs. Par convention, les a_i représentent des arbres, les l_i des listes, les q_i des nombres rationnels et les x_i des nombres réels. Pour tout intervalle à bornes rationnelles I, on note $F(I)$ le plus petit intervalle à bornes flottantes contenant I.

numeric : $\{(a_1) \mid a_1$ est un nombre$\}$
symbolic : $\{(a_1) \mid a_1$ n'est pas un nombre$\}$
dif : $\{(a_1, a_2) \mid a_1 \neq a_2\}$
eqbol : $\{(a_1, a_2, x_3) \mid$ si $a_1 = a_2$ alors $x_3 = 1$ sinon $x_3 = 0\}$
select : $\{(l_1, q_2, a_3) \mid q_2$ est entier, $0 \leq q_2 \leq |l_1|$ et a_3 est le $q_2$ème élément de $l_1\}$
geqrat : $\{(q_1, q_2) \mid q_1 \geq q_2\}$
geq : $\{(x_1, x_2) \mid x_1 \geq x_2\}$
plusrat : $\{(q_1, q_2, q_3) \mid q_3 = q_1 + q_2\}$
incc : $\{(x_1, q_1, q_2) \mid x_1 \in F([q_1, q_2])\}$
inccbol : $\{(x_1, q_2, q_3, x_2) \mid$ si $x_1 \in$ incc alors $x_2 = 1$ sinon $x_2 = 0\}$
integer : $\{(x_1) \mid x_1$ est entier$\}$
nprime : $\{(x_1) \mid x_1$ est un entier non premier$\}$
greater : $\{(x_1, x_2) \mid x_1 > x_2\}$
xor : $\{(x_1, x_2, x_3) \mid x_1, x_2, x_3 \in \{0, 1\}$ et $(x_3 = \overline{1}) \equiv (x_1 = \overline{1}) \oplus (x_2 = \overline{1})\}$
geqbol : $\{(x_1, x_2, x_3) \mid$ si $(x_1, x_2) \in$ geq alors $x_3 = 1$ sinon $x_3 = 0\}$
pi : $\{(x_1) \mid x_1 = \pi\}$
sqrt : $\{(x_1, x_2) \mid x_1 \geq 0$ et $x_2 = \sqrt{x_1}\}$
coth : $\{(x_1, x_2) \mid x_2 = \coth x_1\}$

Il est certainement notable que les fontions usuelles n-aires, utilisées comme telles dans la plupart des langages de type CLP (et notamment en Prolog III) sont ici définies comme des prédicats prédéfinis $n+1$-aires. Ce choix a le double avantage de respecter le caractère relationnel de Prolog (et de correspondre à la norme ISO) et d'être particulièrement adapté à l'arithmétique relationnelle d'intervalles sous-jacente à Prolog IV (voir sections suivantes). De plus, la coincidence des structures de base et des structures étendues peut être définie de façon naturelle sans introduire de transition entre fonctions partielles et relations. Pour une discussion détaillée sur les extensions de structures on pourra se reporter à [9, 12].

2.4 Syntaxe

La syntaxe de base est celle de la norme ISO (y compris pour les constantes numériques). Le traitement des contraintes est effectué grâce à l'utilisation de prédicats prédéfinis. On trouvera la liste de ces prédicats prédéfinis dans [27]. Cependant, pour alléger l'écriture des programmes une notation fonctionnelle des symboles de prédicats prédéfinis ainsi qu'un sucre syntaxique permettant une utilisation des opérateurs et des symboles de relations usuels (par exemple `+`, `*`, `>`) est autorisée en utilisant des symboles de réécriture différents pour les règles et les requêtes . Ainsi, trois types de règles sont syntaxiquement définis en Prolog IV différenciés par le symbole de réécriture (ou d'implication).
Le type principal est celui des règles et requêtes Prolog ISO :

```
P0 :- P1, ..., Pn.,
?- P1, ..., Pn.
```

où les `Pi` sont des prédicats, éventuellement prédéfinis si $i \neq 0$. Ces règles sont suffisantes, en utilisant les prédicats prédéfinis décrits plus haut pour utiliser toutes les fonctionnalités de Prolog IV. On notera cependant que les symboles de fonctions ne sont jamais interprétés hormis par les prédicats qui opèrent explicitement une évaluation (`is/2` par exemple).

Deux autres types permettent d'alléger l'écriture des programmes et des requêtes en autorisant les notations courantes pour les expressions et les contraintes. Il est important de noter qu'il ne s'agit là que de sucre syntaxique. La transformation de base est la suivante:

1. pour chaque terme t de la forme $f(t_1, \ldots, t_n)$, où f est un symbole fonctionnel correspondant à un prédicat prédéfini p, remplacer t par une nouvelle variable x, et ajouter en tête de la règle le but $p(t_1, \ldots, t_n, t)$,

2. répéter cette opération jusqu'à ce qu'elle ne puisse plus s'appliquer,

3. remplacer dans les buts chaque symbole de relation par le prédicat prédéfini correspondant.

On définit deux nouveau types de règles et requêtes pour régler le cas des symboles ambigus comme `+,*,>` qui peuvent correspondre soit aux prédicats prédéfinis correspondants sur les nombres rationnels (et donc être traités par les algorithmes de résolution de contraintes linéaires éventuellement retardés), soit aux prédicats prédéfinis correspondant sur les nombres réels (et donc être traités par approximation comme exposé plus avant).

Syntaxiquement ces deux types de règles sont définis comme suit :

1. Pour les contraintes sur les rationnels
```
P0 /:- P1, ..., Pn.
/?- P1, ..., Pn.
```

2. Pour les contraintes sur les réels
```
P0 .:- P1, ..., Pn.
.?- P1, ..., Pn.
```

Ces différents types de règles peuvent bien sûr être utilisées partout dans un programme, puisqu'un simple préprocesseur transforme tout programme en programme Prolog ISO.

Dans la seconde partie de cet article nous décrivons plus en détail le traitement des contraintes sur les listes, les contraintes linéaires sur les rationnels et les contraintes basées sur l'arithmétique des intervalles, qui recouvrent les contraintes linéaires, polynomiales et transcendantes sur les réels, les contraintes sur les domaines finis, sur les booléens, ainsi que les contraintes qui mixent ces divers domaines.

3 Contraintes sur les listes

En ce qui concerne le traitement des listes munies de la concaténation, la structure de liste est celle définie dans la norme Prolog ISO. Structurellement, ces listes sont donc construites sur la notion de paire pointée, à l'aide des deux symboles fonctionnels binaires usuels représentés par `.` et `[]`. Les deux relations principales sur les listes sont :

- `size(L,Q)`
- `conc(L1,L2,L3)`

La complexité des procédures de décision pour le problème général (problème du mot) rend celui-ci inaplicable à la définition d'un langage de programmation dans lequel la manipulation de listes est vraissemblablement l'opération la plus fréquente. Comme en Prolog III , des restrictions sont donc apportées aux ensembles de contraintes que Prolog IV résoud parfaitement (i.e. pour lesquels l'algorithme est complet). Ce sont les ensembles constitués de contraintes du type `conc(U,V,L1)` et `size(l2,N)` qui vérifient les conditions suivantes :

1. deux au moins des termes `L1,U` et `V` représentent des listes de taille connue,
2. `N` est un nombre entier positif connu ou `L2` est une liste de taille connue.

Toute contrainte ne satisfaisant pas les conditions exprimées précédemment est retardée, au sens de Prolog III , à savoir qu'elles n'est ajoutée au système de contraintes courant que lorsque suffisamment d'informations permettent de vérifier les conditions. Par exemple, aucun des deux systèmes suivants n'est reconnu comme incohérent par Prolog IV :

```
?- { size(X,N),  N <= 1, X = [1,2|Y] }

?- { L = U.V,
size(L,3),
U = [1,2|W],
V = [1,2|W] }
```

Par rapport à Prolog III , le traitement des listes a été remodelé entièrement pour s'adapter aux structures de listes standard. De plus, le traitement retardé sur les listes est simplifié par l'abandon de l'ajout systématique au système de contraintes numériques des *contraintes au longueurs* qui expriment, dans toute concaténation retardée que la somme des tailles des deux listes concaténées est égale à la taille de la liste résultat et que les tailles de deux listes égales sont égales. Cette simplification est motivée par le surcoût, en termes de performances causé par l'ajout de ces contraintes numériques inutiles dans les cas courant. Ces contraintes peuvent cependant être explicitement définies par le programmeur. Une description complète du traitement des listes en Prolog IV, on pourra se reporter à [9].

4 Contraintes linéaires

Comme dans Prolog III , la résolution des contraintes linéaires consiste à déterminer qu'un ensemble d'équations et d'inéquations linéaires est soluble. Outre ce problème de solubilité, le module de résolution de contraintes numériques de Prolog IV dispose d'un mécanisme de détection de variables contraintes à ne prendre qu'une seule valeur

(*variables figées*). Ce mécanisme est essentiel pour les communications entre les divers modules de résolution de contraintes ainsi que pour le traitement des diséquations et des buts retardés. Signalons enfin que ce module de résolution de contraintes linéaires manipule des nombres en précision infinie.

La résolution des équations linéaires utilise une version incrémentale de l'algorithme de Gauss.

```
?- { X + Y = 1, X - Y = 1 }
   X = 1/2, Y = -1/2
```

Quant à la résolution des inéquations, Prolog IV utilise une version modifiée de l'algorithme du Simplex [32, 33, 34, 25] pour d'une part l'étude de la satisfaisabilité et d'autre part la détection des variables figées.

```
?- { X + Y =< 0, X >= 0, Y >= 0 }
   X = 0, Y = 0
```

Le traitement des diséquations numériques et la détection des variables figées sont des problèmes similaires. En effet, résoudre $\{e \neq 0\}$ où e est une expression linéaire consiste à introduire une nouvelle variable x et à résoudre le système $\{x = e, x \neq 0\}$.

Comme signalé précédemment, l'algorithme de Gauss et celui du Simplex détectent les variables figées. La difficulté de la détection des variables figées survient lorsque l'expression linéaire e contient à la fois des variables contraintes à être non négatives et des variables non contraintes.

Pour des raison d'efficacité Prolog IV conserve dans le module de résolution des équations numériques, une copie du système des inéquations. Ce système dupliqué et codé sous la forme normale requise par l'algorithme de Gauss reste inerte aux pivots effectués par l'algorithme du Simplex et permet un gain important en efficacité et en consommation mémoire.

4.1 Implantation

Le mécanisme de base [24] du traitement des équations consiste à exprimer une variable comme une combinaison linéaire d'autres variables. Etant donné un sytème d'équations S et $e = 0$ une nouvelle équation, les variables figurant dans e sont remplacées par la combinaison linéaire associée à cette variable et ce tant qu'il est possible d'effectuer une substitution. Ce processus s'arrête avec l'une des trois forme suivantes:

1. $0 = 0$, l'équation est impliqué par le système S.

2. $k = 0$, où k est une constante non nulle; le système $S \cup \{e = 0\}$ est insoluble.

3. $x_0 - a_1x_1 - \ldots a_nx_n = 0$ où aucun des x_i ne s'exprime comme combinaison linéaire d'autres variables; la forme normale du sytème $S \cup \{e = 0\}$ est alors $S \cup \{x_0 = a_1x_1 + \ldots a_nx_n\}$. Etant donnée une relation d'ordre lexicographique $\prec$ sur les variables, la variable x_0 est telle que $x_0 \prec x_i$. La donnée de cet ordre permet

- l'optimisation du processus de substitution,
- l'optimisation de la détection des variables figées,
- la communication du module de résolution d'équations et celui des inéquations.

Pour le traitement des inéquations, la relation d'ordre sur l'ensemble des variables est telle que si x est une variable numérique non astreinte à être non négative et y une variable astreinte à être non négative alors on a $x \prec y$. Ainsi, lorque le processus de substitution décrit plus haut aboutit a une équation de la forme $x_0 + a_1x_1 + \cdots a_nx_n = 0$ où tous les x_i sont astreints à être non négatifs, cette équation est ajoutée dans le module de résolution des inéquations. Ce dernier vérifie que le système reste soluble et exhibe, s'il en existe, les variables figées induites par l'ajout de cette nouvelle contrainte.

5 Approximation par Intervalles

L'introduction de l'arithmétique des intervalles en programmation logique initialement présentés par John Cleary [6] et implantée par William Older et André Vellino dans leur système BNR-Prolog [21, 22] a ouvert la voie à l'étude des algorithmes d'approximation de contraintes par produits cartésiens d'intervalles. Ces résultats ont pu être étendus de manière naturelles au traitement efficace des domaines discrets et implantés dans le système CLP(BNR) [4, 22]. Ces travaux ont par la suite été généralisés par Alain Colmerauer [11]. Dans [5], il est également établi que les propriétés de cohérence partielle de type consistance d'arc développés en Intelligence Artificielle peuvent s'exprimer simplement en termes de points fixes d'opérateurs de réduction (constraint narrowing operators) définis à partir des fonctions d'approximation.

Pour une description complète des mécanismes d'approximation par intervalles on pourra se reporter par exemple à [4, 5]. Nous rappelons simplement ici les éléments théoriques essentiels.

5.1 Eléments théoriques

Le principe de base consiste à approximer toute relation définie sur $\mathbb{R}^n$ par le plus petit, au sens de l'inclusion) produit cartésien d'intervalles flottants (ou d'unions d'intervalles flottants). On appelle intervalle flottant tout intervalle (au sens mathématique usuel) dont les bornes sont des nombres flottants (en fait appartiennent à un sous-ensemble fini privilégié de $\mathbb{R}$) ou les symboles représentant les infinis. Ces fonctions d'approximation (notées $\underline{\text{apx}}$) sont monotones, englobantes et idempotentes.

On définit ensuite pour chaque relation n-aire ρ définie sur $\mathbb{R}$ un opérateur de réduction (constraint narrowing operator), noté $\overrightarrow{\rho}$, qui associe à toute relation $u = I_1 \times \ldots \times I_n$, où les I_i sont des intervalles flottants (ou des unions d'intervalles flottants) la relation définie par $\overrightarrow{\rho}(u) = \underline{\text{apx}}(\rho \cap u)$. Ces opérateurs sont contractants, monotones, idempotents et corrects.

Enfin, étant donné un ensemble de contraintes élémentaires sur des variables représentant des réels (ces relations correspondant exactement aux prédicats prédéfinis sur les réels

dans Prolog IV), et des domaines (intervalles flottants ou unions d'intervalles flottants) associés aux variables, on applique un algorithme de type AC3 (pour plus de précisions sur ces algorithmes de consistance d'arc on pourra se reporter à [18, 17]). Cet algorithme, appelé algorithme de point fixe, calcule un état stable pour lequel les produits cartésiens des domaines constituent un point fixe pour chacun des opérateurs de réduction correspondant aux contraintes du système. On montre que cet algorithme termine et calcule des résultats corrects, indépendamment de l'ordre des propagations.

Comme il est sous-entendu dans ces préliminaires, l'un des points importants concerne le choix de la fonction d'approximation. Les deux approximations naturelles sont l'approximation par intervalles flottants et l'approximation par unions d'intervalles flottants, notées ci-après respectivement hull et union. L'approximation union est bien entendu plus précise (intuitivement, on préserve les "trous" dans les domaines), mais présente le risque d'augmenter exponentiellement le nombre d'intervalles lorsque l'on introduit des contraintes qui ne sont pas intervalle-convexes[2]. Des tests ont cependant montré que cette approximation donnait d'excellents résultats sur des problèmes discrets [26]. En ce qui concerne Prolog IV les deux approximations sont disponibles et interchangeables à l'aide d'un meta-prédicat.

Enfin, nous terminerons ce préambule par une précision d'ordre terminologique. On parlera dans le reste de ce document de *résolution* de systèmes de contraintes. Or, comme nous l'avons évoqué, l'algorithme utilisé est incomplet (on rappelle à ce propos que le problème de la solubilité de contraintes sur les entiers ou de contraintes transcendantes est indécidable). On ne garantit jamais (si l'on excepte le cas où l'on peut utiliser des procédures d'énumération sur les domaines finis) l'existence de solutions dans les intervalles calculés. En revanche, la propriété de correction de l'algorithme permet d'affirmer la validité des preuves de non-existence de solutions. On appelera donc par abus de langage résolution d'un système de contraintes l'approximation de l'ensemble de ses solutions par les divers algorithmes de résolution utilisés.

La justification théorique de l'emploi de ce terme s'appuie sur la sémantique déclarative du langage (voir [12]) qui utilise une extension de la structure naturelle pour laquelle les algorithmes incomplets de Prolog IV sont prouvés complets. Ces travaux poursuivent ceux entrepris dans [8, 9] ou [1] et qui trouvent leurs racines dans la définition de l'algèbre des arbres rationnels pour justifier l'abandon du test d'occurence.

5.2 Appartenance à un intervalle

Dans la mesure où les prédicats qui définissent les relations d'appartenance d'une variable à un intervalle sont les plus couramment utilisés dans cette section, nous les rappelons brièvement ici :

incc : $\{(x_1, q_2, q_3) \mid x_1 \in F([q_2, q_3])\}$
inoc : $\{(x_1, q_2, q_3) \mid x_1 \in F((q_2, q_3])(\overline{q}_2..\overline{q}_3])\}$
inco : $\{(x_1, q_2, q_3) \mid x_1 \in F([q_2, q_3))\}$
inoo : $\{(x_1, q_2, q_3) \mid x_1 \in F((q_2, q_3))\}$

[2] Les relations intervalle-convexes (on doit le nom à J. Cleary [6]) sont les relations dont les projections de l'intersection avec un produit cartésien d'intervalles n'est pas toujours un intervalle. La relation ternaire mult ou la relation unaire integer sont des exemples de telles relations. Pour un discussion détaillée sur les relations intervalle-convexes on pourra se reporter à [4].

5.3 Contraintes non-linéaires sur les réels

A partir de l'algorithme décrit brièvement ci-dessus, on peut traiter en Prolog IV des ensembles de contraintes, bâties à partir des relations primitives construites à partir des opérations et relations usuelles en mathématiques, y compris les relations définies à partir des fonctions transcendantes. On donne ici quelques exemples très simples de la résolution de contraintes non-linéaires sur les réels.

```
.?- incc(Y,1,3), Y=X^2.
    incc(Y,1,3), incc(X,-1.73205089569091,1.73205089569091).

.?- 1 = X + 2Y, Y - 3X = 0, incc(X,10000,10000).
    incc(0.142857134342193603,0.14285714924335479736),
    incc(Y,0.428571403026580810,0.14285714924335479736).

.?- X >= 0, tan(X) = Y, X^2 + Y^2 = 5.
    inoc(X,1.096424102783,1.0972573757171630),
    inco(Y,1.94833934307098388,1.94880855083465576).
```

Lorsque les intervalles calculés doivent être affinés (spécialement lorsque l'on utilise la fonction d'approximation hull) les solutions peuvent être séparées grâce au prédicat prédéfini `enumerate/1` qui procède à une exploration dichotomique non-déterministe des intervalles. Ce prédicat est à rapprocher des procédures d'énumération utilisées couramment sur les domaines finis. En voici deux exemples :

```
.?- X^3 + Y^3 = 2Y, X^2+Y^2 = 1, X >= 0, enumerate(X).
    inoo(X,0.910759508609771728,0.910760223865509033),
    inoo(X,0.412935733795166,0.4129370152950286865).

.?- incc(X,0,1), 0 = 35X^(256) - 14X^(17) + X, enumerate(X).
    inoo(X,0,1.96181785005474389929 32e-44).

    inoo(X,0.847943603992462158,0.847943723201751708).

    inoo(X,0.995842456817626,0.995842516422271728).
```

5.4 Contraintes sur les entiers

Le traitement des contraintes sur les entiers bornés (classiquement appelées contraintes sur les domaines finis (voir par exemple [28]) consiste principalement à ajouter aux prédicats prédéfinis le prédicat `integer/1` qui correspond précisément à la relation unaire usuellement dénotée par l'ensemble $\mathbb{N}$. L'algorithme de point fixe défini plus haut appliqué à l'opérateur de réduction correspondant a un comportement équivalent aux algorithmes dits de "propagation d'intervalles" sur les domaines finis, dans le cas ou la fonction d'approximation est hull. Si on utilise l'approximation union, on retrouve l'implémentation des algorithmes de consistance d'arc.

Il faut également noter que, dans la mesure où `integer` dénote une relation (et non un type) la présence dans les mêmes contraintes de variables réelles et de variables

contraintes à représenter des valeurs entières se traite de manière très naturelle. Pour la même raison, toute variable numérique peut être dynamiquement contrainte à représenter un entier au cours de l'exécution d'un programme. Du point de vue des performances, de nombreux benchmarks réalisés à l'aide du premier prototype du langage montrent un comportement compétitif par rapport aux temps d'exécutions publiés dans la littérature.

5.5 Contraintes Booléennes

Les contraintes booléennes de Prolog IV sont, encore, un cas particulier des approximations par intervalles. Les variables booléennes sont des variables contraintes à être des entiers pris dans l'intervalle $[0, 1]$. Les relations formées à partir des fonctions usuelles $(\vee, \wedge, \neg, \Rightarrow)$ sont approximées de la même façon que les autres relations sur les réels. La procédure de décision implantée par l'algorithme de point fixe est bien entendu incomplête, ce qui se résoud classiquement par l'introduction de procédures d'énumération. Les résultats fournis par ce type de méthode sont, comme il à déjà été montré par exemple dans [1, 4, 7], particulièrement compétitifs avec les méthodes de résolution classiques. Voici deux exemples de ce type de contraintes :

```
.?- 1 = B & (C | ~D).
    B = 1, incc(C,0,1), incc(D,0,1).

.?- 1 = B & (C | ~D), 0 = B & C.
    B = 1, C = 0, D = 0.
```

5.6 Contraintes mixtes

Nous avons déjà remarqué que les contraintes sur les réels et les entiers sont intimement mélées du fait de la définition de la relation `integer`. On peut ajouter que les variables booléennes, qui sont aussi des variables numériques entières peuvent être utilisées dans des relations comme add, par exemple, ce qui permet d'exprimer naturellement, entre autres, les contraintes de cardinalité (comme définies dans [30]).

De plus, le simple fait de définir à la fois les contraintes sur les réels, sur les entiers et sur les booléens dans un même modèle permet d'introduire pour toute relation n-aire p, une relation n+1-aire p' dont le dernier argument est un booléen égal à 0 si p est vrai et égal à 1 dans le cas contraire. Toutes ces relations sont traitées par le même algorithme ce qui permet d'établir des liens explicites entre les relations numériques et les booléens, ce qui n'est pas le cas des langages tels que Prolog III ou CHIP.

5.7 Liens entre les contraintes rationnelles et réelles

Tout d'abord, il faut noter qu'en mode rationnel aussi bien qu'en mode réel, toutes les constantes numériques sont lues en précision infinie. Elles sont ensuite codées sous forme d'intervalles flottants lorsque c'est nécessaire.

Les liens entre les contraintes portant sur les rationnels et celles portant sur les réels sont définis opérationnelement de la manière suivante :

1. Pour toute variable `X` figée[3] à une valeur `A` par l'un des résolveurs en précision infinie, il existe une contrainte `incc(X,A,A)` dans l'ensemble de contraintes courant si `A` est une valeur exactement représentable par un nombre flottant, ou de la forme `inoo(X,A-,A+)` dans le cas contraire. On note `A-` (respectivement `A+`) le nombre flottant immédiatement inférieur (respectivement supérieur) à `A`.

2. Pour toute contrainte de la forme `incc(X,A,A)` présente dans le système courant, il existe également une contrainte de la forme `X=A`.

3. La détection de l'incohérence de tout système contenant une contrainte `X=A`, avec `X` irrationnel est assurée.

Ce dernier point mérite quelques explications. Considérons les deux exemples suivants :

1. Soit un système contenant la contrainte `pi(X)` et la contrainte `X=3.14159...`

2. Soit un système contenant les contraintes `mult(X,X,2)` et `X= 1,414....`

Supposons que, dans les deux cas, la variable `X` soit instanciée avec un nombre rationnel en précision infinie suffisamment précis pour que le plus petit intervalle flottant contenant `X` soit un intervalle ouvert dont les bornes sont deux flottants consécutifs. Si l'on se contente d'appliquer les règles (1) et (2), on ne peut pas détecter l'incohérence. Or, au moins dans le premier cas, une simple vérification en précision infinie suffit à décider que le système est insoluble. Dans le second cas, cette vérification ne suffit pas, mais le calcul de l'opérateur de réduction appliqué à la contrainte `irrational(X)`, avec `incc(X,1,414...,1,414...)` suffit à détecter l'incohérence.

Pour terminer, notons que les traductions de flottant en précision infinie, et de précision infinie dans l'un des deux flottants immédiatement supérieur ou inférieur sont accessibles par le biais de prédicats prédéfinis.

6 Exemples de programmes

Nous terminons ce très bref apperçu du langage par quelques exemples de programmes très simples et bien connus dans la communauté Programmation logique avec Contraintes, le manque de place ne nous permettant pas de présenter autant d'exemples que nous aurions voulu.

6.1 Concaténations

Voici Le programme qui inverse une liste ainsi que deux exécutions :

```
reverse([],[]).
reverse({[A]@X},{Y@[A]}) \:-
   reverse(X,Y).
```

[3] On rappele qu'une variable figée est une variable dont la valeur est identique pour chacune des solutions du système de contraintes courant.

```
?- reverse([1,2,3,4,5,6,7,8,9,10],L)
    L = [10,9,8,7,6,5,4,3,2,1].
?- reverse(L,[1,2,3,4,5,6,7,8,9,10])
    L = [10,9,8,7,6,5,4,3,2,1].
```

6.2 Contraintes non-linéaires

L'un des benchmarks classiques dans la communauté d'analyse numérique par intervalle est le polynôme de Wilkinson proposé initialement en [31]. Le problème consiste à résoudre l'équation suivante :

$$\prod_{i=1}^{20}(X+i)+EX^{19}=0.$$

Cette équation admet trivialement 20 solutions réelles lorsque $E = 0$. Ce qui est ici mis en avant est qu'une perturbation minuscule peut avoir des effets spectaculaires sur les solutions. Dans ce cas, l'ensemble des solutions réelles de cette équation lorsque $E = 2^{-23}$ est réduit à 10 valeurs, ce que la plupart des codes de calcul numérique calculant en flottant est incapable de vérifier. Voici le programme en Prolog IV et les résultats obtenus :

```
wilkinson(X) .:-
   wilkinson(X,20,P),
   P + 2^(-23)*X^(19) = 0.

wilkinson(X,0,1)) .:- .
wilkinson(X,I,(X+I)*P) .:-
   X # 0,
   wilkinson(X,I-1,P).

.?- wilkinson(X), enumerate(X).
    inoo(X,-20.846910476684571,-20.846906661987305).
    inoo(X,-8.9172506332397461,-8.9172496795654297).
    inoo(X,-8.0072679519653321,-8.0072669982910157).
    inoo(X,-6.9996976852416993,-6.9996972084045411).
    inoo(X,-6.0000071525573731,-6.0000066757202149).
    inoo(X,-5,-4.9999995231628418).
    inoo(X,-4.0000004768371583,-4).
    inoo(X,-3,-2.9999997615814209).
    inoo(X,-2.0000002384185792,-2).
    inoo(X,-1,-0.99999994039535523).
```

6.3 Suites magiques

Les contraintes mixtes peuvent être utilisées pour résoudre des problèmes du type de celui des suites magiques proposé dans [8, 28]. Le problème consiste à trouver une suite d'entiers non-négatifs $(x_0, ...x_{n-1})$ qui vérifient que pour tout i appartenant à $\{0, ..., n-1\}$,

x_i est le nombre d'occurences de l'entier i dans la suite. En d'autres termes, pour tout $i \in \{0, ..., n-1\}$

$$x_i = \sum_{j=0}^{n-1} (x_j = i),$$

où la valeur de $(x = y)$ est l'entier 1 si $(x = y)$ est vrai et l'entier 0 si $(x \neq y)$ est vrai. De plus, on établit les deux propriétés suivantes :

$$\sum_{i=0}^{n-1} x_i = n, \sum_{i=0}^{n-1} i\, x_i = n$$

Pour des raisons de lisibilité, le programme suivant exprime les contraintes principales sans ajouter les contraintes redondantes. Pour les mêmes raisons, le prédicat `non_neg_int` n'est pas détaillé:

```
magic(N,L) .:-
   size(L,N),
   non_neg_int(L),
   main_constraints(L,0),
   enumerate(L).

main_constraints(L,N) .:-
   size(L,N).
main_constraints(L,I) .:-
   size(L1,I),
   L = L1 @ [X] @ L2,
   sum(L,I,X),
   main_constraints(L,I+1).

sum([],I,0).
sum([X|L],I,S) .:-
   S = eqbol(X,I) + S1,
   sum(L,I,S1).
```

On notera l'utilisation fonctionnelle du prédicat `eqbol/3`. Voici un exemple d'exécution du programme :

```
.?- magic(7,L).
    L = [3, 2, 1, 1, 0, 0, 0].
```

7 Conclusion

Nous avons tenté dans ce bref apperçu de Prolog IV de décrire dans les grandes lignes les fonctionnalités du langage ainsi que les principaux algorithmes qui sont utilisés pour y résoudre des contraintes.

Nous terminerons en présentant l'état de développement du système. Un premier prototype de compilateur est d'ores et déjà opérationnel et intègre tous les solveurs

présentés ici. La phase actuelle concerne les tests et la validation de la syntaxe et du système en général, le développements de certains prédicats ISO qui ne sont pas encore définis ainsi que le développement de l'environnement de programmation et du débogueur.

Historiquement, les premiers travaux autour de Prolog IV ont débutés en 1991, et le prototype actuel est le fruit de nombreuses investigations et de collaborations fructueuses. Pour terminer, nous voudrions insister sur le fait que le travail présenté ici est bien entendu un travail d'équipe qui dépasse largement la participation des auteurs de cet article. Les principales décisions en matière de conception, ainsi que la définition du modèle théorique et des sémantiques opérationelles et déclaratives du langage sont dues à Alain Colmerauer. La partie approximation par intervalles a été développée par Michel Van Canheghem et Stéphane N'Dong. Les travaux d'implantation du compilateur ont été menés Jean-François Pique, Eric Vetillard et Jean Luc Massat ainsi que par l'un des auteurs. Enfin, l'environnement et la syntaxe doivent beaucoup à Pascal Bouvier.

Remerciements

Certains des travaux présentés dans cet article ont été en partie financés par les projets Européens Esprit PRINCE et Basic Research Action ACCLAIM. Les auteurs voudraient enfin remercier chaleureusement Alain Colmerauer qui les a guidés depuis bien des années au travers de ses propres découvertes.

Références

[1] F. Benhamou and J.L. Massat. Boolean Pseudo-Equations in Constraint Logic Programming, *Proceedings of ICLP'93,* MIT Press, pp 517–531, Budapest, Hungary, 1993

[2] F. Benhamou. Boolean Constraints in Prolog III in *Constraint Logic Programming: Selected Research,* F. Benhamou and A. Colmerauer (eds.), MIT Press, 1994.

[3] F. Benhamou, D. MacAllester and P. Van Hentenryck. CLP(Intervals) revisited. *Proceedings of ILPS'94*, Ithaca, NY, USA, 1994.

[4] F. Benhamou and W. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *Journal of Logic Programming*, 1995. (A paraître).

[5] F. Benhamou. Interval Constraint Logic Programming. in *Constraint Programming : Basics and Trends*, A. Podelski (ed.), LNCS, Springer, 1995 (A paraître).

[6] J.G. Cleary. Logical Arithmetic. *Future Generation Computing Systems,* 2(2), 1987.

[7] P. Codognet and D. Diaz A Simple and Efficient Boolean Solver for CLP. *Journal of Automated Reasoning*, (A paraître).

[8] A. Colmerauer. An Introduction to Prolog III. *CACM*, 33(7):69, 1990.

[9] A. Colmerauer. Naive Solving of Non-linear Constraints, *Constraint Logic Programming: Selected Research,* F. Benhamou and A. Colmerauer (eds.), MIT Press, pages 89–112, 1993.

[10] A. Colmerauer. Naive Solving of Constraints on Trees and Lists, invited talk in *PPCP'94*, Newport, RI (USA), 1993. aussi Deliverable projet ACCLAIM, 1993.

[11] A. Colmerauer. A legal framework for discussing approximate solving of Constraints. *INTERVALS '94, collection of abstracts*, St Petersburgh, Russia, 1994.

[12] A. Colmerauer. Spécifications de Prolog IV. *Draft*, 1995.

[13] M. Dincbas, H. Simonis and P. Van Hentenryck Extending Equation Solving and Constraints Handling in Logic Programming. *Proc. Colloquium CREAS MCC,* Austin, Texas, 1987.

[14] J. Jaffar and J.L. Lassez, Constraint Logic Programming. *Proc. POPL*, ACM, 1987.
[15] J. Jaffar, S. Michaylov, P. J. Stuckey and R. H. C. Yap, The CLP($\Re$) Language and System *ACM Tr. on Programming Languages and Systems*, Vol. 14, no 3, p 339–395, 1992.
[16] J. Jaffar and M. Maher, Constraint Logic Programming: a Survey *Journal of Logic Programming*, Vol. 19/20, pages 503–581, 1994.
[17] A.K. Mackworth. Consistency in Networks of Relations. *Art. Int.*, 8(1), 1977.
[18] U. Montanari. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Information Science*, 7(2):95–132,1974.
[19] R.E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.
[20] W. Older and A. Vellino, "Extending Prolog with Constraint Arithmetic on Real Intervals", in *Proc. of the Canadian Conference on Electrical and Computer Engineering*, 1990.
[21] W. Older and A. Vellino. *Constraint Arithmetic on Real Intervals*. In *Constraint Logic Programming: Selected Papers*, F. Benhamou & A. Colmerauer eds., The MIT Press, Cambridge, MA, 1993.
[22] W. Older and F. Benhamou. Programming in CLP(BNR). In *PPCP'94*, Newport, 1993.
[23] Prince Deliverable WP 2.1 / R4. Constraint solvers in Prince Prolog: analysis of algorithms,1992.
[24] Prince Deliverable WP 1.3 / R3. Final Results with the demonstration system, 1992.
[25] Prince Deliverable WP 2.1 / R5. Results on compilation of constraints, 1993.
[26] Prince Deliverable WP 2.3 / R6 Performance Evaluation of the Prince Prototype, 1994.
[27] Projet ESPRIT PRINCE *Final report*. 1995.
[28] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming* MIT Press, 1989.
[29] P. Van Hentenryck, V. Saraswat, and Y. Deville. The Design, Implementation, and Evaluation of the Constraint Language `cc(FD)`. Technical Report, Brown University, 1992.
[30] P. Van Hentenryck and Yves Deville. The Cardinality Operator: A new Logical Connective for Constraint Logic Programming *Constraint Logic Programming: Selected Research*, F. Benhamou and A. Colmerauer (eds.), MIT Press, pages 383–403, 1993.
[31] J.H. Wilkinson. *The algebraic Eigenvalue Problem*, Oxford University Press, 1965.
[32] George B. Dantzig. Linear Programming and Extensions. Princeton University Press (1963).
[33] R. S. Garfinkel and G. Nemhausser. Integer Programming. John Wiley and sons. 1972.
[34] Pascal Van Hentenryck and Thomas Graf. Standard Forms for Ration al linear Arithmetic in Constraint Logic Programming. Proc. of International Sympos ium on Mathematics and Artificial Intelligence, 1990.