1

Tutoriel et Survol de Prolog IV

1.1 Une session prolog sous Prolog IV

PROLOG IV est un langage de programmation. En tant que tel, il est doté des différents attributs d'un environnement de programmation : un compilateur et une bibliothèque. Le compilateur est utilisé pour transformer des programmes en une forme directement utilisable par la machine, et la bibliothèque contient tout le code prédéfini utilisé dans Prolog IV. Contrairement aux autres langages de programmation comme C ou C++, Prolog IV propose un mode interactif sous la forme d'un dialogue homme/machine de questions/réponses.

Pour démarrer une session Prolog IV vous devrez taper la commande `prolog4` sur votre ligne de commande, ou bien cliquer sur l'icone appropriée, ceci dépendant de la plate-forme que vous utilisez. Si vous avez des problèmes pour démarrer Prolog IV, faites-vous aider par votre administrateur système ou référez-vous au guide d'installation, fourni avec votre kit Prolog IV.

Prolog IV, c'est aussi prolog, c'est le but de cette mini-session

Si Prolog IV démarre normalement, vous verrez apparaître ce qui suit (aux date et numéro de version près) :

```
machine% prolog4

Prolog IV (beta III (204), cp4.8), Juin 1996 (C)PrologIA 1995,1996
[ Mon Jun 17 10:48:57 MET DST 1996 ]

>>
```

Le symbole >> est appelé l'*invite* (ou encore *prompt*). Elle indique que Prolog IV attend une requête. Tapons par exemple la requête « write(hello),nl. ».

```
>> write(hello), nl.

hello
true.

>>
```

Vous ne devez bien sûr taper que la requête, c.à.d. ce qui est en gras, le reste est affiché en réponse par Prolog IV) . Dans notre exemple, notre requête est constituée de deux littéraux séparés par une virgule. Le premier littéral, `write`, est une commande prédéfinie de Prolog IV qui peut donc être utilisée à tout moment. C'est une commande générale d'impression. Le littéral `nl` est aussi une commande prédéfinie, et imprime tout simplement un retour-chariot. On met toujours un point en fin de requête. Le mot `true` imprimé ensuite par Prolog IV indique que la commande a réussi, du moins d'un point de vue logique.

Maintenant, après avoir vu une requête, voici un programme. Il est composé de trois règles qui expriment que Jacques et Edouard vivent à Paris, et que Bernard habite Marseille.

```
>> consult.

Consulting ...
   habite(jacques,paris).
   habite(edouard,paris).
   habite(bernard,marseille).
   end_of_file.
true.

>>
```

La commande `consult` est une des commandes possibles permettant de rentrer un programme dans le système (on verra plus tard qu'il existe une commande `compile`, qui accepte en argument un nom de fichier). La ligne « `end_of_file.` », qu'il faut taper [1], indique au système la fin du programme et la terminaison du mode de consultation ; on revient donc au prompt Prolog IV.

Prolog IV enregistre les règles dans une *base de règles* et ce, dans l'ordre où elles sont tapées.

Essayons notre programme : Est-ce que Bernard habite Marseille ?

```
>> habite(bernard, marseille).

true.

>>
```

Dans cette requête, nous n'avons fait que vérifier ce que nous supposions déjà ; essayons maintenant d'interroger notre base de règle avec des questions du genre « Où habite Jacques ? ».

Il est temps d'introduire les notions d'*identificateur* et de *variable*. Un identificateur est un nom ; il commence par une lettre minuscule [2]. Une variable est représentée par une suite de lettres qui commence par une lettre majuscule [3]. A toute variable est associé un domaine de valeurs possibles à un instant donné. Parmi ces valeurs possibles, il y a bien sûr les nombres, les identificateurs mais aussi les constructions arborescentes utilisant ces nombres et ces identificateurs ; on appelle arbres ces constructions.

---

1. Dans un shell unix, on peut se contenter de taper les touches `Control-D` .
2. Vous comprenez maintenant pourquoi nous avons mis des noms propres en minuscules dans notre programme ; ceci peut etre contourné, mais nous n'en parlerons que dans le chapitre décrivant la syntaxe.
3. Le chapitre syntaxe décrit précisement la syntaxe d'une variable.

Lors de la création d'une variable pendant l'exécution du programme, celle-ci peut prendre pour valeur chacun des arbres possibles du domaine sur lequel travaille Prolog IV. Le déroulement d'un programme Prolog IV réduit le domaine des variables qui participent à son exécution.

Posons maintenant nos questions qui comportent une ou plusieurs inconnues :

Où habite Jacques ?

```
>> habite(jacques, X).

X = paris.

>>
```

Qui habite Paris ?

```
>> habite(X,paris).

X = jacques;

X = edouard.

>>
```

Nous avons obtenu deux réponses à cette question, c.à.d. deux valeurs possibles pour la variable `X`. En Prolog IV, les solutions sont séparées par un point-virgule (`;`) et la dernière solution est suivie d'un point, tout comme les requêtes.

Qui habite où ?

```
>> habite(X,Y).

X = jacques,
Y = paris;

X = edouard,
Y = paris;

X = bernard,
Y = marseille.

>>
```

A chaque fois, le système répond par un ensemble de valeurs données aux variables de la requête (qui sont `X` et `Y`), valeurs qui satisfont à la relation `habite`.

Pour terminer une session Prolog IV, tapez la commande suivante :

```
>> halt.

machine%
```

et vous retournerez à l'interpréteur de commande de votre système d'exploitation. Si Prolog IV était utilisé depuis son envionnement graphique, il faudrait actionner le bouton « Quit » de la palette principale.

1.2 D'autres exemples, avec des contraintes !

Jusqu'ici, nous n'avons vu qu'un petit programme Prolog IV (et même prolog) très simple, ne comportant d'autres contraintes que les égalites (l'unification). Prolog IV peut utiliser des programmes beaucoup plus complexes, et nous allons donner maintenant quelques exemples qui montrent comment manipuler les différents objets Prolog IV.

1.2.1 Les contraintes numériques

La partie la plus novatrice de Prolog IV est son traitement des nombres. Ses traitements de nombres devrait-on dire, car il y en a deux.

Quand le problème numérique posé est composé d'équations ou d'inéquations linéaires comme $\frac{3}{4}x - \frac{1}{5}y \geq 7z - 1$, c'est le solveur linéaire qui doit être utilisé et Prolog IV trouve les solutions exactes au problème.

Quand le problème numérique posé est composé d'éléments plus complexes, comme des mélanges d'opérations algébriques, de fonctions trigonométriques ou transcendentales (par exemple $x = 2sin^2x$), on utilisera le solveur dit «approché».

Commençons par examiner le solveur linéaire.

1.2.2 Contraintes numériques linéaires

Débutons par un petit exemple :

```
>> X+Y = 3 , X-Y = 1.

Y = 1,
X = 2.

>>
```

Ici, Prolog IV a été utilisé pour résoudre le système d'équations linéaires :

$$\begin{cases} x + y = 3 \\ x - y = 1 \end{cases}$$

Examinons maintenant un autre petit problème : nous avons un certain nombre de chats et d'oiseaux, ainsi que le nombre de leurs têtes et de leurs pattes, et nous nous demandons d'exprimer les relations qui lient ces nombres. Si nous appelons c le nombre de chats, v le nombre d'oiseaux (v pour volatile, o ressemblant trop à un zéro), t le nombre de têtes et p le nombre de pattes, nous avons les équations suivantes :

$$\begin{cases} t = c + v \\ p = 4c + 2v \end{cases}$$

qui sont bien linéaires (pas de produit ou de rapport de variables) et qui s'expriment naturellement en Prolog IV (en mettant bien les variables en majuscule !) :

```
>> T = C+V , P = 4*C+2*V .

P ~ real,
V ~ real,
C ~ real,
T ~ real.

>>
```

Prolog IV répond par une suite de contraintes qui indiquent que chacune des variables appartient à l'ensemble des nombres réels, (ce dont on se doutait). L'existence de cette réponse, dans le cadre du solveur linéaire, indique que le système est soluble. Allons un peu plus loin en donnant des valeurs à quelques unes des variables. Puisque le système est composé de deux équations à quatre inconnues, fixons la valeur de deux d'entre elles pour obtenir les valeurs des deux autres. Fixons le nombre de pattes à 14 et le nombre de têtes à 5 :

```
>> T = C+V , P = 4*C+2*V , P=14 , T=5 .

P = 14,
V = 3,
C = 2,
T = 5.

>>
```

Cette fois, Prolog IV nous dit que la solution est unique en exhibant les valeurs des variables pour lesquelles l'ensemble de contraintes est vérifié. Avant de poursuivre avec cet exemple, nous allons déclarer une règle qui lie les quatre variables et les équations, de sorte que nous n'ayons plus à écrire les équations à chaque fois :

```
>> consult.

Consulting ...
   chatsoiseaux(C,V,P,T) :- T = C+V , P = 4*C+2*V .
   end_of_file.
true.

>>
```

Nous venons d'introduire une nouvelle règle dont la signification est : *sous reserve que les deux contraintes (deux équations dans ce cas) soient vérifiées, la relation chatsoiseaux(c,v,p,t) est vraie.* Nous pouvons maintenant taper les requêtes suivantes :

```
>> chatsoiseaux(C,V,14,5).

C = 2,
V = 3.

>> chatsoiseaux(1,V,P,5).

V = 4,
P = 12.

>> chatsoiseaux(1,1,2,4).

false.

>> chatsoiseaux(C,V,6,4).

C = -1,
V = 5.

>>
```

Prolog IV nous repond `false` à la troisième requête, car l'ensemble des contraintes n'est pas soluble. Pour ce qui est de la quatrième, une des variables de la solution est négative. Il est vrai que lors de la formalisation de notre problème, nous n'avons jamais spécifié que les solutions devaient être positives. Nous devons donc réécrire notre relation `chatsoiseaux` pour contraindre les variables de notre relation à être positives. Ecrivons la nouvelle relation :

```
>> reconsult.

Reconsulting ...
   chatsoiseaux(C,V,P,T) :-
       T=C+V , P=4*C+2*V ,
       gelin(C,0), gelin(V,0), gelin(P,0), gelin(T,0).
   end_of_file.
true.

>>
```

La commande `reconsult` est utilisée pour remplacer un programme déjà défini avec une nouvelle version de celui-ci. Utiliser `consult` avec un programme déjà défini aurait amené à un message d'erreur. La contrainte `gelin(X,Y)` pose tout simplement la contrainte[4] $X \geq Y$. On a donc posé pour chacune des variables la contrainte $X \geq 0$.

4. Le nom $gelin$ vient de la juxtaposition de ge, qui est un raccourci de « greater or equal » (plus grand ou égal), et du suffixe lin qui rappelle que cette contrainte ne travaille que dans le linéaire. On peut trouver cette notation un peu lourde alors qu'existe le symbole conventionnel $\geq$ pour désigner cette relation d'ordre, mais d'une part la norme prolog ISO a reservé ce symbole à un usage précis (comparaison de nombres), ensuite il faut distinguer cette contrainte d'une autre contrainte de comparaison travaillant dans l'autre solveur numérique et enfin, c'est un nom officiel de Prolog IV, nom auquel on peut toujours se référer en toutes circonstances sans dépendre en aucune façon du contexte. Il sera fourni plus tard des « raccourcis » agréables qui altèrent la syntaxe…, pour le meilleur ou pour le pire de la lisibilité. Notons au passage l'existence des autres relations d'ordre fonctionnant dans le linéaire, qui sont $lelin$ « less or equal » (plus petit ou égal), $ltlin$ « less than » (plus petit strict) et $gtlin$ « greater than » (plus grand strict).

```
>> chatsoiseaux(C,V,6,4).

false.

>>
```

Prolog IV nous dit qu'il n'y a pas de solution à cette requête, ce que nous voulions. Mais maintenant, qu'arrive-t-il si nous donnons un nombre de pattes et un nombre de têtes tels qu'il n'y ait pas de solution entière, en mettant par exemple un nombre impair[5] de pattes ?

```
>> chatsoiseaux(C,V,7,3).

V = 5/2,
C = 1/2.

>>
```

Il nous est retourné un demi-chat et cinq-moitiés d'oiseaux... (une boucherie !) Il est vrai que nous n'avions pas la non plus specifié que les solutions devaient être entières. Nous avons utilisé de Prolog IV le solveur d'équations linéaires, qui raisonne sur l'ensemble des nombres rationnels (et donc pas seulement sur les entiers). Bien qu'il soit possible de contraindre une variable à être entière et d'affiner notre petit programme, nous nous arrêterons là avec cet exemple.

1.2.3 Contraintes sur les réels (solveur approché)

Prolog IV fournit un jeu de contraintes qui nous permet de raisonner sur les nombres réels. Nous prendrons bien soin de ne pas confondre les contraintes linéaires expliquées dans la section précédente avec les contraintes sur les réels que nous allons rapidement introduire ici. Il est difficile de raisonner précisement sur les nombres réels. Un nombre aussi simple que $\sqrt{2}$ n'est pas représenté exactement en machine avec Prolog IV.

Bien sûr, on peut calculer autant de chiffres que l'on veut et aller bien au delà de la précision de 1.414, mais nous ne pourrons jamais trouver (et donc à plus forte raison faire des opérations sur) toutes les décimales qui composent ce nombre.

Une alternative pourrait être d'implanter un solveur qui fait du calcul formel, mais il faut bien reconnaître que ces systêmes sont lourds et d'une certaine lenteur.

Ce qui nous reste donc est un solveur implantant un compromis entre performances (vitesse et consommation mémoire) et utilité de ce qui est déduit pour la résolution de problèmes.

Pour cette raison, ces contraintes sur les réels sont prises en compte par un solveur tout à fait différent du solveur complet des équations et inéquations linéaires.

Ceci nous amène à formuler les deux propriétés les plus importantes sur les contraintes sur les réels en Prolog IV :

– *La résolution d'*une contrainte de base *est* complète.

– *La résolution d'*un ensemble de contraintes *est* incomplète.

5. Les vertébrés ont normalement un nombre pair de pattes, et cette parité se transmet à toute collection de vertébrés. Mais c'est de la méta-connaissance et ceci déborde largement du sujet.

Par incomplet, nous voulons dire que Prolog IV peut ne pas détecter l'insolubilité de certains systèmes de contraintes qui n'ont pas de solution. Nous montrerons plus loin le pourquoi du passage de la complétude à l'incomplétude lorsqu'on passe de la résolution d'une contrainte à la résolution de plusieurs contraintes.

Bien sûr, l'incomplétude est *a priori* ennuyeuse, puisque décider si un système de contraintes est soluble ou pas est plus difficile. En fait, la façon dont ces contraintes sont traitées par Prolog IV nous garantit certains résultats.

Dans ce qui suit, on différencie le problème dit *exact*, qui est le problème originel (mathématique, physique, ...) du problème dit *approché*, qui est sa traduction naturelle en Prolog IV.

- Il est garanti que si un problème (système de contrainte) approché est déclaré insoluble par Prolog IV, alors le problème exact est tout aussi insoluble.
- Il est garanti que si le problème exact a une solution, alors elle se trouve parmi les solutions du problème approché (elle est donc trouvée par Prolog IV).
- Si Prolog IV trouve une solution au problème approché dans laquelle toutes les variables prennent une valeur unique, alors cette solution *est* aussi solution du problème exact.

Voyons quelques exemples. Essayons de calculer la valeur de $\sqrt{2}$. Il suffit de poser assez naturellement sous Prolog IV la contrainte $X = sqrt(2)$:

```
>> X = sqrt(2).

X ~ cc(`>1.4142135`,`>1.4142136`)

>>
```

L'expression $sqrt(2)$ est un *terme ensembliste* dit aussi *pseudo-terme*. Un pseudo-terme ne se distingue d'un terme que par le fait qu'il fait intervenir, à quelque niveau que ce soit, au moins un symbole de relation dont le nom et l'arité+1 sont réservés[6]. La définition est donc récursive : en effet soit le nœud a un nom de relation réservée, soit l'un des fils au moins du nœud est lui-même un pseudo-terme. Bien sûr tout ceci n'est pas exclusif et on peut fabriquer des imbrications assez diaboliques, avec plusieurs relations. Il faut noter qu'il n'y a pas de différences syntaxiques entre termes et pseudo-termes.

Les pseudo-termes numériques représentent des ensembles quelconques de réels. Parfois cet ensemble se réduit à un seul élément, nombre rationnel ou pas.

Pour en revenir à notre exemple, voyons qu'on peut tout aussi bien poser la contrainte à l'aide de la notation relationnelle plutôt que fonctionnelle, on obtiendra la même réponse (on voit mieux maintenant pourquoi la relation $sqrt$ est d'arité 2) :

6. Ils figurent dans une liste spéciale qu'on donnera plus loin dans le manuel. La relation $sqrt_2$ en fait partie.

```
>> sqrt(X, 2).

X ~ cc('>1.4142135','>1.4142136')

>>
```

Prolog IV répond en nous donnant le domaine de valeurs possibles pour la variable X, c'est à dire un sous-ensemble de **R**. Il ne s'agit toutefois pas de n'importe quel sous-ensemble de réels. Seulement une partie d'entre eux est utilisée par Prolog IV :

– les singletons formés d'un nombre rationnel (on parle alors dans ce cas-là de valeur plutôt que de domaine),

– les intervalles construits à l'aide des seuls nombres flottants simples de la norme IEEE (auxquels on se réfèrera plus tard sous le vocable de nombres flottants) [7] . Ces ensembles-là sont par ailleurs en nombre fini puisque formés à l'aide des seuls nombres flottants, qui sont en quantité finie [8] .

D'après nos principes, la solution (nous savons indépendamment de Prolog IV qu'elle existe) est à l'intérieur. Prolog IV nous a donné une réponse à l'aide des relations $\sim$ et cc. La contrainte $X \sim E$, avec X une variable signifie «X est élément de l'ensemble décrit par E». Par exemple $X \sim cc(1,2)$ signifie que X est élément de l'intervalle fermé [9] $[1,2]$.

Les valeurs des bornes sont des nombres rationnels [10] et la borne «`` `>1.4142135` ``» avec ces étranges décorations[11] vaut très précisément «le premier nombre flottant arrivant après le nombre décimal 1.4142135 .»

Comme on l'a vu précédemment, il est normal d'avoir un encadrement au lieu d'une simple valeur puisqu'il n'est pas possible de calculer (et à plus forte raison d'afficher) la valeur de $\sqrt{2}$. Prolog IV ne peut que retourner l'intervalle

$$[1.41421353816986083984375, 1.414213657379150390625]$$

qui est le plus petit intervalle possible qui contienne $\sqrt{2}$ et qui est tel que les deux bornes soient des nombres flottants en simple précision [12] . En fait, lors du traitement des contraintes sur les réels, Prolog IV effectue tous ses calculs de sous-domaines avec de tels intervalles.

Toutefois, Prolog IV peut calculer des valeurs exactes (qui sont donc des nombres rationnels), même quand il traite de telles contraintes :

```
>> X = sqrt(4).

X = 2.

>>
```

7. On peut travailler avec des unions d'intervalles en se plaçant dans un mode spécial.
8. L'ensemble des parties d'un ensemble fini est fini. Comme il y a environ 2^{32} nombres flottants, il y a donc environ $2^{2^{32}}$ sous-ensembles de nombres flottants.
9. cc est le mnémonique raccourci de *closed-closed* (fermé-fermé), désignant le type des bornes gauche et droite de l'intervalle. Il existe aussi bien sûr co, oc et oo, le o signifiant *open* (ouvert).
10. Les seuls nombres qui soient exprimables par Prolog IV ou le programmeur.
11. On aurait de même «`` `<1.4142135` ``» qui dénote le flottant précédant le nombre décimal 1.4142135 .
12. Prolog IV utilise une notation qui allège la suite de chiffres qu'il faudrait imprimer, sans perdre l'exactitude du nombre ainsi raccourci.

Dans ce cas, la valeur est précise, et nous sommes *sûrs* que c'est la solution de notre contrainte. Bien entendu, même dans l'exemple plus haut, la réponse était juste, quoique moins précisément exprimée car irrationnelle.

Regardons maintenant l'exemple suivant :

```
>> X = sqrt(2) , X = sqrt(2.0000001).

X ~ oo(`>1.4142135`,`>1.4142136`).

>>
```

La réponse est (presque) la même qu'avant et le résultat montre un succès bien qu'il soit évident pour nous que $\sqrt{2} \neq \sqrt{2.0000001}$. La raison en est qu'avec la précision donnée par les intervalles bornés par les nombres flottants, il n'est pas possible de faire la différence entre ces deux nombres, le pouvoir de résolution des nombres flottants [13] ne permettant pas de les distinguer. Toutefois, ce qui semble être une faiblesse respecte parfaitement un de nos principes «*s'il y a une solution du problème réel, elle se trouve dans les sous-domaines retournés par la résolution du problème approché.*»

Par acquis de conscience, donnons-nous le même exemple avec des nombres flottants séparables (il nous suffit d'enlever un 0) :

```
>> X = sqrt(2) , X = sqrt(2.000001).

false.

>>
```

Ouf !

Il faut toujours garder à l'esprit que le solveur des nombres réels ne travaille que sur le problème approché, pas sur le problème exact.

Voici maintenant la relation $square$. Elle ressemble assez à $sqrt$, mais ce n'est pas tout à fait la même chose puisque $sqrt(X)$ est le nombre positif dont le carré vaut X.

Jouons un peu avec toutes deux, en les utilisant de façon imbriquée pour changer :

```
>> square(sqrt(X)) = 2.

X ~ cc(`>1.9999998`,`>2.0000004`).

>> sqrt(square(X)) = 2, ge(X,0).

X = 2.

>>
```

La réponse à la première question donne une bonne approximation (bien lisible) de l'équation $(\sqrt{X})^2 = 2$ dont la solution est mathématiquement $X = 2$. La contrainte dans cette question n'est pas atomique, puisque formée de relations imbriquées.

13. Ce sont des nombres flottants IEEE en simple précision, celle-ci étant de l'ordre de 10^{-6} (relativement au nombre sur lequel elle s'applique, c.à.d. que la précision est de l'ordre de 10^{10} pour un nombre comme 10^{16}). L'exemple utilise des nombres séparés d'environ 10^{-7}.

La seconde question contient une nouvelle contrainte basée sur la relation ge [14] qui nous permet d'imposer à X d'être positif ($X \geq 0$). Bien que cette question comporte plusieurs contraintes approchées, sa réponse ne comporte que des valeurs (pas de domaine) ; elle est donc exacte, d'après l'un de nos principes (*Si Prolog IV trouve une solution au problème approché dans laquelle toutes les variables prennent une valeur, alors cette solution* est *aussi solution du problème exact*).

1.2.4 Le pourquoi de l'incomplétude

Voyons informellement comment se passe la résolution d'*une* contrainte :

1. A partir de la contrainte, on considère les domaines des arguments de sa relation (qu'on note par exemple $\mathcal{R}$) ; supposons la relation d'arité n, et appelons les domaines $D_1, \ldots, D_n$.
2. La contrainte est résolue complètement, c.à.d. on cherche le sous-ensemble E de $\mathbb{R}^n$ tel que $E = \mathcal{R} \cap D_1 \times \cdots \times D_n$. On supposera que E est non-vide (il y a sinon un échec.) Pour fixer les idées, imaginons que l'on est dans $\mathbb{R}^3$ et que notre ensemble ressemble à une patate.
3. On effectue une approximation de cet ensemble par le plus petit pavé de $\mathbb{R}^n$ qui l'englobe. Avec notre exemple, on obtient un parallèlépipède contenant la patate au plus près, c.à.d. la touchant sur chaque face [15].
4. Le pavé en question est bien sûr un produit cartésien d'intervalles de $\mathbb{R}$; on va approcher ceux-ci par des sous-domaines réels de Prolog IV. Chacune de ses dimensions fait l'objet d'une approximation par un ensemble :

   Si l'intervalle est réduit à un nombre, on le transforme en le plus petit intervalle flottant qui le contient.

   Sinon c'est un véritable intervalle et on arrondi chacune des bornes par le flottant le plus proche [16] par défaut ou par excès selon qu'il s'agit de la borne inférieure ou supérieure.

   Bref, quand les deux bornes d'un intervalle ne sont pas des nombres flottants, on approche celui-ci par un intervalle de nombres flottants, pris un peu plus large (juste ce qu'il faut) afin ne pas perdre de valeurs.
5. Ces domaines sont attribués aux variables qui apparaissaient en tant qu'argument. Le domaine de chacune des variables n'a pu que diminuer.

En plus de cette résolution existe un mécanisme prenant en compte exclusivement les nombres rationnels. Quand suffisamment d'arguments de la contrainte sont des nombres rationnels, il peut être déduit les valeurs (elles aussi rationnelles) des autres arguments.

La résolution d'*un ensemble* de contraintes se passe très simplement en traitant chaque contrainte (c'est une relation et des variables ou constantes en

14. ge est le raccourci de *greater or equal*. Il ne faut pas la confondre avec $gelin$!!! La relation ge pose une contrainte sur les réels, alors que $gelin$ permet de poser une contrainte linéaire qui n'est prise en compte que dans un autre solveur.

15. Donc le plus petit parallèlépipède possible.

16. Il peut s'agir du même borne si celle-ci est déjà un nombre flottant.

argument) comme indiqué ci-dessus. Les variables servent seules d'interface entre les contraintes, aussi un sous-domaine « surévalué » (qui approche trop grossièrement l'ensemble exact) peut éventuellement cacher une absence de solution mathématique. En effet, l'intersection de l'approximation de deux ensembles disjoints peut être non-vide.

Prenons par exemple la requête avec les contraintes suivantes :

```
>> X = oo(-1,1), Y = oo(-1,1), gt(X,Y).

Y ~ oo(-1,1),
X ~ oo(-1,1).

>> X = oo(-1,1), Y = oo(-1,1), lt(X,Y).

Y ~ oo(-1,1),
X ~ oo(-1,1).

>>
```

On obtient en réponse aux deux requêtes que les sous-domaines pour X et Y sont inchangés [17]. Ceci signifie que l'ensemble approximant la conjonction des trois contraintes de chaque requête est le pavé ouvert $(-1,1) \times (-1,1)$.

Et quand on met ensemble les quatres contraintes, on obtient. . .

```
>> X = oo(-1,1), Y = oo(-1,1), gt(X,Y), lt(X,Y).

Y ~ oo(-1,1),
X ~ oo(-1,1).
```

. . . un succès ! Alors que cet ensemble de contraintes est mathématiquement insoluble.

Nos ennuis viennent de la conjonction des deux contraintes $gt(X,Y)$ et $lt(X,Y)$. Chacune est traitée parfaitement, mais le traitement de l'ensemble des deux passe par l'utilisation de domaines grossiers (parce que se sont des pavés) qui servent d'interface (par le biais de variables communes). C'est cette faiblesse de représentation des domaines (voulue pour une implantation efficace) qui rend notre solveur incomplet.

Notes :

– Les contraintes de domaines (utilisant des relations comme cc, oo,. . .) ne posent pas de problèmes dans la mesure où ce qui est passé en argument est représentable par des nombres flottants.

– Il y a une autre cause d'incomplétude car il y a deux moments où il est fait une approximation pendant le traitement d'une contrainte (les points 3 et 4). Celle décrite au point 4 se présente moins fréquemment que celle que l'on vient de montrer : il s'agit de l'exemple avec $\sqrt{2}$ et $\sqrt{2.0000001}$, vu dans une section antérieure.

– Une contrainte de la forme $rel(X,X)$ n'est pas autre chose que la conjonction de contraintes $(\exists A)\ rel(X,A), eq(A,X)$ qui comporte donc *deux* contraintes et est donc soumise aux problèmes d'incomplétude précités.

17. On ne s'expliquera pas sur le choix d'intervalles ouverts dans ces exemples. Il permet de simplifier l'explication et de ne pas cumuler plusieurs effets.

1.2.5 Union d'intervalles

Oublions temporairement l'incomplétude en examinant d'autres requêtes et leurs réponses dans deux modes de fonctionnement du solveur de contraintes sur les réels.

```
>> X = sqrt(2).

X ~ cc('>1.4142135','>1.4142136').

>> square(X) = 2.

X ~ cc(-'>1.4142136','>1.4142136')

>>
```

La première requête (qu'on connait déjà) rend un petit domaine alors que la dernière requête rend pour X un domaine de taille environ 2.8 d'amplitude (le premier nombre est négatif !). Nous savons que l'équation $X^2 = 2$ a deux racines $-\sqrt{2}$ et $\sqrt{2}$, et le plus petit intervalle qui les contient toutes deux est mathématiquement $[-\sqrt{2}, \sqrt{2}]$, intervalle dont Prolog IV donne une bonne approximation à travers le sous-domaine de la variable X.

Passons dans le mode Prolog IV *union d'intervalles* pour voir comment les choses se passent [18] :

```
>> set_prolog_flag(interval_mode, union).

true.

>> sqrt(square(X)) = 2.

X ~ -2 u 2.

>>
```

Nous rend X appartenant au sous-domaine formé des deux nombres -2 et 2, qui est une forme allégée de $\{-2\} \cup \{2\}$. Au passage, nous découvrons l'existence du pseudo-terme t_1 `u` t_2 (avec `u` pour union) qui est l'ensemble des arbres qui appartiennent à t_1 ou à t_2 (de façon non-exclusive).

En mode « intervalles simples » cette requête aurait pour réponse `X ~ cc(-2,2)`, et donc l'intervalle $[-2, 2]$.

La question suivante avait déjà été posée plus haut, en mode « intervalles simples». Il y est maintenant répondu, en mode «union d'intervalles»:

```
>> square(X) = 2.

X ~ cc(-'>1.4142136',-'>1.4142135')u cc('>1.4142135','>1.4142136').

>>
```

Cette fois-ci, on obtient une union de deux petits intervalles respectivement autour de $-\sqrt{2}$ et de $\sqrt{2}$, au lieu de l'intervalle $[-\sqrt{2}, \sqrt{2}]$, obtenu en mode «intervalles simples». Dans certains cas, on a donc une meilleure séparation des solutions à l'aide du mode «union d'intervalles».

18. Pour en sortir, il suffira de taper le but « `set_prolog_flag(interval_mode, simple).` ».

1.2.6 Le quantificateur existentiel

Le quantificateur existentiel sert à construire une contrainte de la forme $X\ ex\ P$ dans laquelle X est une variable muette (ou existentielle) et P une contrainte (ou une conjonction de contraintes comme une requête). Elle se lit «*il existe X tel que P*» et a même valeur de vérité que P. Cette construction est une extension qui ne fait pas partie de la norme ISO. Pragmatiquement, cette construction signale à Prolog IV que la variable X n'a pas d'intérêt aux yeux du programmeur et que l'on n'est pas intéressé par sa valeur ou son sous-domaine, qui ne seront donc pas affichés en réponse quand on utilise cette construction dans une requête. Par exemple :

```
>> X = Y.

X = Y,
Y ~ tree.

>> X ex X = Y.

Y ~ tree.

>> X ex X = f(g(2),g(2)).

true.

>>
```

En réponse à la seconde requête, rien n'est affiché pour X. Dans la troisième, Prolog IV se contente de répondre que la requête est soluble.

Les variables existentielles permettent d'avoir à sa disposition des sortes de « variables locales », complètement déconnectées du contexte, elles ne sont donc pas liées avec l'extérieur de la construction ex.

Dans la requête suivante, la variable X joue deux rôles, dont un rôle tout à fait local dans la contrainte parenthésée. Il ne faut pas être étonné que X puisse valoir à la fois 1 et 3, puisqu'en fait il ne s'agit pas du tout du même X.

```
>> X = Y, (X ex X = 1), Y = 3.

Y = 3,
X = 3.

>>
```

Voici encore trois exemples utilisant la quantification existentielle.

```
>> X ex Y = f(X,X)

A ex
Y = f(A,A),
A ~ tree.

>> X ex Y = f(X).

Y ~ f(tree).

>> X ex Y ex Z = f(X,Y).

Z ~ f(tree,tree).

>>
```

Dans la première requête, on crée une contrainte comportant deux occurrences d'une même variable existentielle. Dans la réponse, une variable existentielle est créée (avec pour domaine l'ensemble des arbres) puisqu'elle est nécessaire pour refléter le fait que deux sous-arbres identiques inconnus figurent dans la valeur de Y.

Dans la seconde requête au contraire, il n'est pas nécessaire de créer une variable existentielle puisque l'arbre complètement inconnu n'apparait qu'une fois, Prolog IV s'est contenté de mettre à cette place le pseudo-terme $tree$, qui représente l'ensemble de tous les arbres.

La troisième requête ne sert ici qu'à montrer une imbrication de quantificateurs, et se lit : *il existe X tel que : il existe Y tel que : Z égale …*, l'associativité se faisant de la droite vers la gauche.

1.2.7 Utiliser *compatible* ou *egal* ?

Quand doit-on mettre « $\sim$» ou «$=$» dans les contraintes ?

Ces deux relations $\sim$ (compatible) et $=$ (égal) sont très similaires et il faut bien avouer qu'elles semblent faire la même chose, dans le sens où le programmeur peut se tromper et écrire l'un pour l'autre, Prolog IV rectifiant de lui-même [19]. Par contre, Prolog IV ne se permet pas en sortie cet abus de notation et utilise toujours la bonne relation pour les réponses, en utilisant $=$ le plus souvent possible, et $\sim$ quand c'est nécessaire. La relation $=$ ne s'emploie qu'avec des termes, alors que $\sim$ seule accepte les pseudo-termes en membre gauche ou droit. Bref $=$ est l'égalité (unification dans les prologs standard) ordinaire, alors que $\sim$ se comporte selon le contexte de l'une des façons suivantes :

- $terme \sim terme$ est l'égalité (c'est exactement «$=$»).
- $terme \sim pseudoterme$ est vraie si l'individu représenté par $terme$ appartient à l'ensemble représenté par $pseudoterme$, fausse sinon.
- $pseudoterme \sim terme$ est vraie si l'individu représenté par $terme$ appartient à l'ensemble représenté par $pseudoterme$, fausse sinon.

19. La raison pour laquelle Prolog IV arrive à « rectifier » les $=$ abusifs en $\sim$ est qu'il n'y a pas d'unification sur les ensembles en Prolog IV, mais seulement des calculs d'intersection, internes et *a priori* inaccessibles. Il n'y a donc pas d'ambiguïté pour lui.

– $pseudoterme \sim pseudoterme$ est vraie si l'intersection des deux ensembles est non-vide, fausse sinon.

1.2.8 Les variables muettes *underscore*

Les variables *underscore* sont des variables muettes qui s'écrivent à l'aide du seul caractère «_». Chacune des occurrences représente une variable différente des autres. Ces variables sont tout simplement des variables existentielles qu'il n'est pas nécessaire de nommer et de quantifier.

```
>> X = f(_,_).

X ~ f(tree,tree).

>>
```

1.2.9 Relations et pseudo-termes

On a parfois montré à travers les exemples précédents des formes fonctionnelles (pseudo-terme) ou relationnelles (littéral) d'une même contrainte. Il faut savoir que toute contrainte peut être notée sous forme de relation ou de pseudo-terme (ensemble), la seule différence étant la position syntaxique et, bien sûr, l'arité.

Voici le rapport entre les deux concepts (et notations) que sont la relation et l'opération :

$$nom(X_0, X_1, \ldots, X_n) \;\equiv\; X_0 \sim nom(X_1, \ldots, X_n)$$

Comme on le voit, l'argument qui sert de «résultat» à l'opération est le premier de la relation. La formule précédente conserve l'ordre des arguments quand on lit chaque membre de l'équivalence de gauche à droite. Par exemple, les contraintes :

$ge(X, 0) \;\equiv\; X \sim ge(0)$ avec $ge(0)$ l'ensemble des nombres supérieurs ou égaux à zéro.

$u(X, 1, 2) \;\equiv\; X \sim u(1, 2)$ avec $u(1, 2)$ [20] l'ensemble des deux nombres $\{1, 2\}$.

La présence de l'argument «résultat» en première position peut surprendre les habitués de prolog pour qui l'usage veut que les arguments de sortie viennent après ceux d'entrées, donc vers la fin du littéral, mais il faut se rappeler les points suivants :

– *ce qu'on appelle* «résultat» *n'est pas plus un argument de sortie qu'un autre, c'est tout au plus un argument privilegié quant à son extraction.*

– *Les relations sont rarement utilisées telles quelles, il faut leur préférer les notations fonctionnelles (pseudo-termes) qui sont la plupart du temps plus lisibles* [21].

– *L'argument* «résultat» *se trouve toujours au même endroit*[22].

20. u existant en tant qu'opérateur Prolog IV infixé, on peut faire aussi $1\ u\ 2$.

21. C'est particulèrement vrai lorsque les relations ont une forte connotation fonctionnelle, comme $plus$ ou cos.

22. Cette propriété n'est intéressante que pour la méta-programmation, où on manipule des règles et littéraux.

1.2.10 Des requêtes aussi puissantes qu'étonnantes

Dans tout ce qui suit, le programme consiste à chaque fois en une requête, sans avoir à définir de règles. On supposera sauf indication contraire qu'on est dans le mode «union d'intervalles».

L'ensemble des X tels que leur carré est strictement plus grand que 1 : (Ce qui suit un caractère `%` jusqu'à la fin de ligne est un commentaire prolog.)

```
>> gt(square(X),1).  % ou encore » square(X) ~ gt(1).

X ~ lt(-1)u gt(1).
```

X a pour domaine l'ensemble des nombres strictement plus petits[23] que -1 ou strictement plus grands que 1.

Une déduction élégante

Si Y est un carré ...

```
>> X ex square(X) = Y.

Y ~ ge(0).
```

... alors il est positif ou nul !

Il a été utilisé ici un quantificateur existentiel afin de ne pas voir apparaître inutilement X dans la réponse.

Quelques requêtes sur les nombres entiers

Voici le pseudo-terme $X n Y$ dont le n rappele le symbole $\cap$ de l'intersection. Il construit l'ensemble dont les éléments appartiennent à chacun des deux ensembles donnés en arguments[24]. Le pseudo-terme int représente l'ensemble des entiers relatifs. Quels sont les réels entre 1 et 9 qui sont des entiers ? (autrement dit quel est l'ensemble : $\{X|X \in [1,9] \cap \mathbb{N}\}$)

```
>> X ~ cc(1,9) n int.

X ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9.
```

Comme on est en mode «union», on obtient la liste en extension[25, 26].

---

23. lt provient de *less than* (plus petit strict). gt provient de *greater than* (plus grand strict). Il existe de même le pour *less or equal* (plus petit ou égal) et ge pour *greater or equal* (plus grand ou égal).
24. Quand l'un des arguments de n n'est pas un pseudo-terme (et donc pas un ensemble), alors l'argument est transformé en singleton contenant le terme (*idem* pour u (union))
25. en mode «simple», on obtient rien de plus comme domaine que `X ~ cc(1,9)`. En effet, int n'est pas un domaine, mais une contrainte, et Prolog IV n'affiche que les domaines et constructions dans ses réponses.
26. Si vous voulez savoir ce qui se passe quand on demande la liste en extension des entiers, allez directement à la section «**Problèmes**».

```
>> X  ~  cc(1,9) n times(2,int).      % les entiers pairs entre 1 et 9

X ~ 2 u 4 u 6 u 8.

>> X  ~  cc(1,9) n int n times(2,nint).  % les entiers impairs de 1..9

X ~ 1 u 3 u 5 u 7 u 9.
```

Le pseudo-terme $nint$ [27] représente l'ensemble des réels qui ne sont pas des entiers. Le pseudo-terme $times(X,Y)$ construit l'ensemble des produits possibles de ses deux arguments. Pour exprimer que P est un nombre pair, il suffit de contraindre P a être le double d'un entier. Pour avoir un nombre impair, il faut le contraindre a être un entier qui est le double d'un nombre non-entier.

Les nombres premiers entre 10 et 100 (les diviseurs possibles à tester ne peuvent être que les nombres premiers de 2 à 7) :

```
>> X  ~  cc(10,100) n int n times(2,nint) n times(3,nint)
            n times(5,nint) n times(7,nint).

X ~ 11 u 13 u 17 u 19 u 23 u 29 u 31 u 37 u 41 u 43 u 47 u 53 u 59 u 61
u 67 u 71 u 73 u 79 u 83 u 89 u 97
```

Nombres rationnels et irrationnels

Le pseudo-terme pi représente l'ensemble des nombres égaux à π. Dans le monde réel, il n'y a bien sûr qu'une valeur possible, laquelle n'est pas représentable en Prolog IV, puisque irrationnelle. La seule déduction possible pour Prolog IV est le domaine de la relation, un petit intervalle ouvert.

```
>> X ~ pi.

X ~ oo(`>3.1415925`,`>3.1415927`).
```

Il ne faut pas croire pour autant que Prolog IV se laisse abuser facilement en lui donnant un nombre rationnel «humainement» proche de π [28] :

```
>> pi ~ 3.141592653589793238462643383279502884197169399375105820974944.

false.
```

Bien que l'on ait une réponse affirmative à la question :

```
>> oo(`>3.1415925`,`>3.1415927`) ~
        3.141592653589793238462643383279502884197169399375105820974944.
```

27. Le nom $nint$ provient de *not integer* (non-entier).

28. Faites-nous confiance, il s'agit bien des véritables soixante premières décimales de π.

En effet, une relation numérique dont tous les arguments sont des constantes est complètement vérifiée par Prolog IV. Dans le cas très simple de pi, les seules constantes numériques de Prolog IV étant les nombres rationnels et π étant irrationnel, la relation `pi(`$nombre$`)` est toujours fausse, puisque $nombre$ ne peut être qu'un rationnel.

Un peu de trigonométrie avec cosinus.

Quels sont les cosinus entiers d'angles compris entre 1 et 100 ? On peut formaliser cet ensemble par $\{y \mid \exists x, x \in [1, 100],\ y = \cos x,\ y \in \mathbb{N}\}$, et le traduire immédiatement en la requête :

```
>> X ex X ~ cc(1,100), Y ~ cos(X) n int.

Y ~ -1 u 0 u 1
```

On pouvait tout aussi bien taper la requête suivante :

«`Y ~ cos(cc(1,100)) n int`».

Quels sont les x tels que $\cos x = 1$ *avec* $-10 \leq x \leq 10$ *?*

```
>> X ~ cc(-10,10), cos(X) ~ 1.

X ~ oo(-'>6.2831854',-'>6.283185')u 0 u oo('>6.283185','>6.2831854')
```

qui se traduirait mathématiquement par l'ensemble :

$$]-(2\pi+\varepsilon),\ -(2\pi-\varepsilon)[\ \cup \{0\} \cup\ ]2\pi-\varepsilon,\ 2\pi+\varepsilon[$$

On y reconnait l'approximation de l'ensemble $\{-2\pi, 0, 2\pi\}$.

Il y a bien entendu d'autres fonctions trigonométriques. Citons en vrac, outre cos, les fonctions sin, tan, cot, leurs réciproques $arcsin$, $arccos$, $arctan$ et les fonctions hyperboliques $cosh$, $sinh$, $tanh$ et $coth$.

Pour clôturer la liste des fonctions transcendentales, on citera aussi exp, ln et log (logarithme décimal).

1.2.11 Fonctions algébriques et autres

Sans grandes explications, voici quelques fonctions et opérations sur les réels. Les quatre opérations classiques sur les réels ($+$, $-$, $*$, $/$) deviennent quatre relations $plus$, $minus$, $times$, div) sur les arbres. Le plus et moins unaires deviennent quant à eux respectivement les relations $uplus$ et $uminus$.

Les notations avec pseudo-termes permettent de voir ces relations comme étant des opérations partielles sur les réels.

Pour des raisons pratiques, certaines relations ont un raccourci synonyme. En voici quelques-uns :

– La relation de symbole `.+.` est un synonyme[29] de la relation $plus$, l'ensemble des sommes de ses deux arguments ; c'est aussi un raccourci pour la relation $uplus$[30].

29. L'ajout de points permet de le distinguer du symbole +, qui lui travaille dans le solveur linéaire. Cette remarque vaut pour les quatres opérations.

– La relation `.-.` est synonyme de la relation $minus$ ou $uminus$.

– La relation `.*.` est synonyme de $times$.

– La relation `./.` est synonyme de div.

De plus, ces raccourcis peuvent être utilisés syntaxiquement comme opérateurs, par exemple `X.+.Y` .

Posons une question en utilisant ces raccourcis :

```
>> X ~ cc(-2,2), Y ~ cc(-1,1), Z ~ .-.sqrt(X.+.Z)./.2 .+. 3.*.Y .

Z ~ cc(-2,3),
Y ~ cc(-'<0.6666667',1),
X ~ cc(-2,2).
```

Il existe d'autres pseudo-termes sur les réels, $floor$ et $ceil$, qui « rendent » respectivement le plus grand entier inférieur et le plus petit entier supérieur, à leur argument. On trouve aussi abs qui est bien sûr la valeur absolue de son argument, min et max qui « rendent » respectivement le plus petit et le plus grand de leurs deux arguments.

1.2.12 Deux mots sur la réduction des sous-domaines numériques

Voyons comment se propage la réduction des intervalles au travers de ce petit exemple mettant en œuvre l'addition. Soit l'équation $z = x+y$, avec $x \in [1, 5]$ et $y \in [-1, 8]$.

Quel est le domaine de z ?

```
>> Z ~ X.+.Y, X ~ cc(1,5), Y ~ cc(-1,8).

Y ~ cc(-1,8),
X ~ cc(1,5),
Z ~ cc(0,13).
```

On obtient comme réponse que $z \in [0, 13]$, le domaine des variables x et y étant inchangé .

Même question, mais en imposant à z d'appartenir à l'intervalle $[-1, 1]$:

```
>> Z ~ X.+.Y, X ~ cc(1,2), Y ~ cc(-1,1), Z ~ cc(0,1).

Y ~ cc(-1,0),
X ~ cc(1,2),
Z ~ cc(0,1).
```

On obtient une réduction de domaine des variables x et y qui appartiennent respectivement à $[1, 2]$ et $[-1, 0]$, ainsi que z, qui ne peut plus qu'appartenir qu'à $[0, 1]$.

Si on impose plutôt à z d'être négatif ou nul,

```
>> Z ~ X.+.Y, X ~ cc(1,5), Y ~ cc(-1,8), Z = le(0).

Y = -1,
X = 1,
Z = 0.
```

30. En réalité, le nom de relation est un couple $(symbole, arit\acute{e})$ qu'on note aussi $symbole/arit\acute{e}$, ou encore $symbole_{arit\acute{e}}$; on devrait donc parler des relations $.+._3$ et $.+._2$, ce qui lève toute ambiguïté.

on obtient ici des domaines restreints à une seule valeur pour chacune des variables. On sait alors en vertu d'un de nos principes que ce n'est plus une approximation du problème réel, mais que c'est sa solution exacte.

Remarques :

- L'ordre des contraintes n'a aucune importance.
- Le fait que les nombres soient des entiers ou non-entiers est sans importance.

1.2.13 Raccourcis et opérateurs

Voici une table donnant les raccourcis disponibles en Prolog IV, tout au moins en ce qui concerne les pseudo-termes. Ces raccourcis sont tous des opérateurs infixes ou préfixes.

Pseudo-termes	Raccourci
$uplus(t)$	`.+.` t
$uminus(t)$	`.-.` t
$plus(t_1, t_2)$	t_1 `.+.` t_2
$minus(t_1, t_2)$	t_1 `.-.` t_2
$times(t_1, t_2)$	t_1 `.*.` t_2
$div(t_1, t_2)$	t_1 `./.` t_2
$upluslin(t)$	$+\,t$
$uminuslin(t)$	$-\,t$
$pluslin(t_1, t_2)$	$t_1 + t_2$
$minuslin(t_1, t_2)$	$t_1 - t_2$
$timeslin(t_1, t_2)$	t_1 `*` t_2
$divlin(t_1, t_2)$	t_1 `/` t_2
$conc(t_1, t_2)$	t_1 `o` t_2
$n(t_1, t_2)$	t_1 `n` t_2
$u(t_1, t_2)$	t_1 `u` t_2
$index(t_1, t_2)$	t_1 `:` t_2

A savoir sur les priorités de ces opérateurs :

- Les opérateurs unaires (comme `.-.`) sont prioritaires sur les binaires.
- Les opérateurs multiplicatifs (comme `.*.` ou `./.`) sont prioritaires sur les opérateurs additifs (comme + ou `.+.`).
- L'associativité de ces opérateurs binaires op est gauche-droite, c.à.d. qu'une expression $t_1\ op\ t_2\ op\ t_3$ est interprétée $(t_1\ op\ t_2)\ op\ t_3$.
- L'intersection (n) est prioritaire sur l'union (u).

1.2.14 Enumération

A travers cet exemple sur les triangles pythagoriciens [31], nous allons effleurer l'énumération des entiers. Afin de limiter le nombre de solutions, on limitera volontairement les côtés des triangles à des valeurs comprises entre 1 et 10. On dit que X, Y et Z appartiennent à $[1, 10] \cap \mathbb{N}$ et on pose $Z^2 = X^2 + Y^2$.

31. Triangles rectangles de côtés entiers.

(le symbole `.+.` est un raccourci de la relation $plus$, qui est l'ensemble des sommes de ses deux arguments.)

```
% en mode "intervalles simples"
>> set_prolog_flag(interval_mode, simple).

true.

>> X ~ cc(1,10) n int, Y ~ cc(1,10) n int, Z ~ cc(1,10) n int,
      square(Z) ~ square(X).+.square(Y).

Z ~ cc(2,10),
Y ~ cc(1,9),
X ~ cc(1,9).

% en mode "union d'intervalles"
>> set_prolog_flag(interval_mode, union).

true.

>> X ~ cc(1,10) n int, Y ~ cc(1,10) n int, Z ~ cc(1,10) n int,
      square(Z) ~ square(X).+.square(Y).

Z ~ 5 u 10,
Y ~ 3 u 4 u 6 u 8,
X ~ 3 u 4 u 6 u 8.
```

On obtient dans les deux cas une solution unique qui contient les triplets qui nous intéressent. Ce qui nous est rendu est un pavé de $\mathbb{R}^3$ dans le premier cas, et un produit cartésien discret dans le cas des unions d'intervalles. On remarquera au passage que le sous-domaine de X est aussi celui de Y, ce qui n'est pas étonnant puisque ces variables jouent un rôle symétrique.

Ecrivons une petite relation comportant les contraintes relatives à un côté, afin de pouvoir taper des requêtes plus courtes :

```
>> consult.

Consulting ...
   cote(X,B) :- X ~ cc(1,B) n int.
   end_of_file.
true.

>> cote(X,10), cote(Y,10), cote(Z,10),
      square(Z) ~ square(X).+.square(Y).

Z ~ 5 u 10,
Y ~ 3 u 4 u 6 u 8,
X ~ 3 u 4 u 6 u 8

>>
```

Effectuons une énumération sur les variable X, Y et Z avec la primitive prédéfinie `intsplit` : on donne en argument à cette primitive une liste de variables dont on cherche des valeurs entières (qui respectent bien sûr le domaine de ces variables). On obtient alors par backtracking les quatre solutions :

```
>> X = cote(X,10), cote(Y,10), cote(Z,10),
      square(Z) ~ square(X).+.square(Y),
      intsplit([X,Y,Z]).

Z = 5,
Y = 4,
X = 3;

Z = 5,
Y = 3,
X = 4;

Z = 10,
Y = 8,
X = 6;

Z = 10,
Y = 6,
X = 8.

>>
```

1.2.15 Partitionnement (ou comment énumérer sur $\mathbb{R}$)

Soit à chercher les solutions (numériques) de l'équation $x = 2 \sin x$, on écrit simplement en Prolog IV :

```
>> X = 2.*.sin(X).

X ~ cc(-2,2).
```

On obtient que l'ensemble des racines de l'équation, si elles existent, se situent dans l'intervalle $[-2, 2]$. C'est intéressant, mais comment obtenir quelque chose de plus précis ? Tous simplement en partitionnant le domaine de la variable X, en deux sous-intervalles par exemple, et pour chacun, voir si les contraintes sont toujours valides ou pas. Quand un intervalle est valide, on le coupe en deux, etc.... Bref, on espère par un partitionnement (dichotomique dans notre cas, mais ce n'est pas requis), élaguer des portions de domaines pour lesquelles il n'y a pas de solution ; on sait alors, par l'un de nos principes[32], qu'il n'y a pas de solution au problème réel, pour ces portions de domaine de X.

Le parallèle avec l'énumération est tout à fait justifié par le fait que le partitionnement de $\mathbb{R}$ s'effectue en suivant tout au plus la grille des nombres flottants, base de calcul des contraintes sur les réels. Ces nombres flottants étant en quantité finie, les intervalles constitués de deux nombres flottants sont tout aussi finis. Le partitionnement se fait donc en temps limité (même s'il peut être très long.)

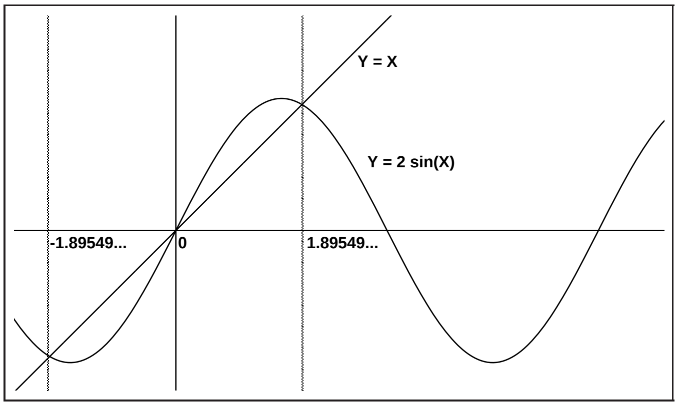
La primitive $realsplit$ se charge d'exhiber par backtracking les domaines où les contraintes restent satisfaites. Cette primitive accepte en argument une liste de variables v_i pour lesquelles on souhaite partitionner le domaine $D_{v_1} \times \cdots \times D_{v_i} \times \cdots \times D_{v_n}$.

32. *« Si un problème (système de contraintes) approché est déclaré insoluble par Prolog IV, alors le problème exact est tout aussi insoluble. »*

Améliorons nos résultats avec cette primitive : dans notre cas nous n'avons qu'une liste d'une seule variable :

```
>> X = 2.*.sin(X), realsplit([X]).

X ~ oo(-`>1.8954951`,-`>1.8954933`);

X ~ oc(-`>3.5762786e-7`,0);

X ~ oo(0,`<2.384186e-7`);

X ~ oo(`<1.895494`,`>1.8954945`).

>>
```

Il nous est rendu pour x quatre domaines disjoints assez fins, où peuvent se trouver les solutions du problème réel. Il n'est pas du ressort du solveur sur les réels d'en dire plus, nous avons obtenu là le maximum.[33]



Regardons les solutions obtenues et la figure ci-dessus, fruit d'une construction analytique sans rapport avec le raisonnement du solveur Prolog IV sur les réels ; repérons les principaux acteurs : les fonctions $y = x$, $y = 2\sin x$ et l'intersection de leurs courbes, en trois points. On remarque sur notre dessin que les solutions sont symétriques[34] par rapport à 0 .

La première et la quatrième réponse correspondent aux approximations des solutions opposés[35] $-1.89549\ldots$ et $1.89549\ldots$.

Quand aux deux autres solutions (la seconde et la troisième), elles forment en fait une seule solution séparée par les hasards du partitionnement dichotomique en deux bouts connexes[36]. C'est en fait un seul (petit) intervalle contenant 0.

33. Tout ce que nous pourrions essayer de faire est de substituer à x, dans l'equation, chacun (?!) des rationnels r qui appartiennent aux domaines, c.à.d. de faire $x = r, x = 2\sin x$ et voir s'il y a un succès ou un échec. Comme ce n'est guère praticable (il y a une infinité de nombres rationnels pour Prolog IV), on se contente tout au plus de tester les bornes fermées de ces domaines. On trouverait dans notre exemple la valeur 0 comme solution *certaine* de notre problème exact, en vertu d'un de nos principes sur les constantes.

34. Ne soyons pas étonnés, toutes les fonctions de notre problème sont impaires.

35. Nous seuls savons qu'elles sont opposées (pas Prolog IV). Les remarques sur les symétries du problème sont de la méta-connaissance.

36. Il n'y a strictement aucun réel entre ces intervalles.

1.2.16 Arbres et listes

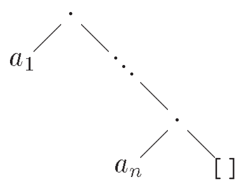
Le domaine de travail de Prolog IV est l'ensemble des arbres dont les nœuds sont étiquetés :

– soit par un nombre ; il n'y a alors pas de fils sous ce nœud.

– soit par un identificateur ; il peut y avoir alors zéro ou plusieurs fils.

On appelle *feuille* un arbre sans fils. Ces arbres se caractérisent également par le fait qu'ils ne comportent qu'un seul nœud.

Une *liste* est un arbre a de la forme



,

où les a_i sont des arbres quelconques. Dans notre texte, une telle liste est notée $[a_1, \ldots, a_n]$ et on désigne par $|a|$ la *longueur* n éventuellement nulle de la liste a. La concaténation de deux listes éventuellement vides est définie et notée par $[a_1, \ldots, a_m] \circ [a_{m+1}, \ldots, a_n] = [a_1, \ldots, a_n]$.

La liste vide est notée [ ] et c'est aussi une feuille et un identificateur. Elle a bien sûr pour longueur 0.

Le pseudo-terme $list$ représente l'ensemble de ces listes. Le pseudo-terme $X\,oY$ [37] (ou encore $conc(X,Y)$) est la concaténation de deux listes.

Voici quelques exemples sur les listes et concaténations.

```
>> X ~ [1,2] o [3,4].

X = [1,2,3,4].

>> [1,2,3,4] ~ X o [3,4].

X = [1,2].

>> X ~ X o [].

X ~ list.

>> X ~ Y o [].

Y ~ list,
X ~ list.
```

La troisième requête ne donne pas de réponse pour X sinon qu'il appartient à l'ensemble des listes. La quatrième requête ne déduit rien sur X et Y [38] sinon leur domaine, car l'approximation de la concaténation utilisée dans le solveur des arbres, suffisante dans les cas courants, n'a pas assez d'informations pour conclure. En effet, dans le cas de la concaténation, il faut en général que deux des trois arguments soient des listes de longueurs connues pour déduire le

37. C'est la lettre o minuscule.

38. On pouvait s'attendre à obtenir $X = Y$, mais ce serait une déduction formelle, pas une déduction sur les domaines.

troisième argument, quel qu'il soit ; sinon, la contrainte $conc$ est retardée et seuls sont attribués les sous-domaines $list$ sur chacun des arguments.

Le pseudo-terme $size(L)$ apparaissant dans $N \sim size(L)$ est l'ensemble des tailles de la liste L. On peut aussi bien s'en servir pour determiner la taille d'une liste que pour la contraindre.

```
>> 10 ~ size(L).

L ~ [tree,tree,tree,tree,tree,tree,tree,tree,tree,tree].

>> N ~ size([1,2,3,4] o [5,6]).

N = 6.

>> N = size(L).

L ~ list,
N ~ ge(0).

>> lt(0) = size(L).

false.

>> N = le(0), N =size(L).

L = [],
N = 0.
```

La première requête a construit une liste de taille 10, garnie de dix éléments appartenant à l'ensemble des arbres. La seconde réponse nous donne la taille d'une liste de taille connue, ici 6. La troisième requête nous apprend que les listes ont une taille positive ou nulle. La quatrième nous confirme le fait précédent, la taille d'une liste ne peut être un nombre inférieur à 0. La cinquième nous indique qu'il n'y a qu'une seule liste de taille 0, c'est la liste vide [ ].

Encore plus loin avec les listes :

```
>> [a] o X  ~  X o [a] ,  10 ~ size(X).

X = [a,a,a,a,a,a,a,a,a,a].

>> [tree] o X  ~  X o [tree] ,  10 ~ size(X).

A ex
X = [A,A,A,A,A,A,A,A,A,A],
A ~ tree.
```

Les deux requêtes construisent des listes de tailles 10. La première est formée de dix occurrences de l'identificateur « a », et la seconde crée une liste de dix variables identiques (une variable existentielle de type $tree$ est donc créée pour pouvoir afficher la liste X.)

Apres avoir vu les deux contraintes de base sur les listes, tout au moins en ce qui concerne leur construction, nous allons maintenant introduire les relations $index$ et $inlist$.

La relation $index$, comme son nom le suggère, permet d'indexer[39] une liste comme on le ferait d'un tableau . Elle permet de poser la contrainte qu'un

39. Pour un i donné, prendre la $i$ème composante de ...

arbre A est le Nième élément de la liste L. Nous ne considérerons dans les exemples qui suivent que des listes de nombres, c'est là que les déductions de Prolog IV sont les plus fécondes.

Première requête : Où est 5 (à quel rang dans la liste) ?

```
>> 5 ~ index([1,3,2,4,6,5,8,10], N).

N = 6.
```

Où est 5 (à quel(s) rang(s) dans la liste) ?

```
>> 5 ~ index([1,3,2,4,6,5,5,8,10,5,13], N).

N ~ cc(6,10).
```

Il nous est repondu que l'indice N est entre 6 et 10 (en effet, 5 apparait plusieurs fois entre ces indices. Essayons d'obtenir mieux en mode «union».

```
>> set_prolog_flag(interval_mode, union).

true.

>> 5 ~ index([1,3,2,4,6,5,5,8,10,5,13], N).

N ~ 6 u 7 u 10.
```

On obtient cette fois-ci les valeurs exactes des rangs où se trouvent nos 5.

Un petit essai d'indexation avec le rang connu.

Quel est le neuvième élément de la liste ?

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], 9).

X = 10.
```

Un petit essai d'indexation avec le rang partiellement connu (entre 1 et 5).

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], N), N ~ cc(1,5).

N ~ cc(1,5),
X ~ cc(1,6).
```

Il est déduit que X est un réel entre 1 et 6.

Voyons maintenant le petit problème suivant : étant donnée une liste de nombres, nous voulons déterminer quels sont les éléments d'une liste L dont la valeur est leur rang dans cette liste, c.à.d. les x tels que $x = L_x$. Allons-y :

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], X).

X ~ cc(1,10).
```

Il nous est repondu que X peut être l'un des dix premiers éléments de la liste, qu'il vaut donc un entier entre 1 et 10 (13 a été éliminé). On aurait préféré en savoir plus sur X. Plaçons nous en mode «union» pour voir :

```
>> set_prolog_flag(interval_mode, union).

true.

>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], X).

X ~ 1 u 2 u 3 u 4 u 5 u 6 u 8.
```

On obtient un meilleur résultat, puisque on n'a plus que sept valeurs possibles pour X au lieu de dix précédemment. Pour en savoir plus (en fait la réponse complète et définitive), effectuons une énumération sur X [40] :

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], X), intsplit([X]).

X = 1;

X = 4;

X = 8.
```

On a cette fois-ci les valeurs exactes (et non plus un sous-domaine) de X, le problème est donc complètement résolu puisqu'on a obtenu des constantes.

Pour voir ce qui se passe quand peu de choses sont connues :

X est le 3ème élément de la liste L :

```
>> X ~ index(L, 3).

L ~ [tree,tree,X|list],
X ~ tree.
```

Prolog IV crée le début de liste L, place X en troisième position puis termine avec quelque chose qui doit être une liste (il s'agit du pseudo-terme $list$.)

La relation $inlist$ ressemble beaucoup à $index$, mais en plus simple, car elle ne se préoccupe pas d'indexation, juste d'appartenance à la liste. Essayons $inlist$ avec, pour changer, une liste d'arbres, contraignons X à appartenir à la liste $[a, b, c]$ puis imposons à X une valeur toute autre :

```
>> X ~ inlist([a,b,c]), X = d.

false.
```

La relation if permet d'avoir un *si-alors-sinon* numérique piloté par la valeurs d'un argument (le premier). Voici sans autres commentaires quelques requêtes utilisant if :

40. Repasser en mode «intervalles simples» n'influe en rien sur la réponse.

```
>> X ~ if(B, 3,4).

B ~ cc(0,1),
X ~ cc(3,4).

>> X ~ if(B, 3,4), X = 5.

false.

>> X ~ if(B, 3,4), X ~ cc(0,3.5).

B = 1,
X = 3.
```

1.2.17 Booléens

Les booléens ne sont rien d'autre en Prolog IV que des entiers contraints à ne valoir que 0 (pour « faux ») ou 1 (pour « vrai »). Le premier avantage de ce choix est que ce sont des nombres réels comme les autres et sont donc utilisables dans n'importe quelle contrainte numérique (ou mixte, comme $index$). Il existe bien entendu toute une série de relations travaillant avec ces booléens, comme $impl$, and, or, xor et d'autres encore.

Traduisons en Prolog IV le fait que $a \Rightarrow b$, $b \Rightarrow c$ et $c \Rightarrow a$ [41] :

```
>> impl(A,B), impl(B,C), impl(C,A).

C ~ cc(0,1),
B ~ cc(0,1),
A ~ cc(0,1).
```

On n'obtient pas d'autres déductions sur nos variables sinon qu'elles sont à valeurs dans $[0, 1]$. Fixons l'une d'entre elles à « vrai » (1) :

```
>> impl(A,B), impl(B,C), impl(C,A), C = 1.

C = 1,
B = 1,
A = 1.
```

On a cette fois-ci une déduction maximale sur nos variables.

Insistons sur le fait que les booléens sont astreints à ne valoir que 0 ou 1, et non pas seulement un réel entre 0 et 1. Ceci se voit immédiatement en passant simplement en mode «union», mais se voit aussi en mode «intervalles simples» de la façon suivante :

```
>> impl(B,_).

B ~ cc(0,1).

>> impl(B,_), B = co(-1,1).

B = 0.
```

La première requête montre un moyen correct de fabriquer un booléen. Dans la seconde requête, il est imposé à B d'appartenir à un l'intervalle contenant

41. Ceci traduit le fait que les variables a, b et c sont équivalentes, c.à.d. vraies en même temps, ou fausses en même temps.

0 mais *ouvert en 1*. Le booléen B prend alors instantanément la valeur 0 (la seule valeur booléenne qui appartienne au domaine donné).

Il existe toute une famille de contraintes faisant intervenir les booléens sous la forme d'une valeur de vérité. Leur nom commence par la lettre b et le pseudo-terme associé retourne une valeur booléenne qui représente la valeur de vérité de la contrainte obtenue en enlevant la première lettre (le b).

Par exemple, il existe la relation ble, qui rend le booléen associé à la valeur de vérité de la contrainte $le(X, Y)$, si on la posait. Un exemple sera plus clair.

B vaut «X est plus petit ou égal à 0 ». Que vaut B ?

```
>> B = ble(X, 0), X = 3.

X = 3,
B = 0.
```

La réponse est «faux» ($B = 0$).

Quels sont les x tels que $x \leq -1$ ET $1 \leq x$:

```
>> and(ble(X,-1), ble(1,X)).

false.
```

Il n'y en a bien sûr aucun.

Toutes ces relations permettent

– de retarder la pose d'une contrainte : la contrainte $le(X, -1)$ est posée dès que B est fixé à «vrai» (1)

```
>> B = ble(X,-1), B = 1.

B = 1,
X ~ le(-1).
```

– de contrôler la pose d'une contrainte (ou de son contraire) ; ici, selon la valeur de B, on pose la contrainte $le(X, -1)$ ou son contraire $lt(-1, X)$

```
>> B = ble(X,-1), B = 0.

B = 1,
X ~ gt(-1).
```

– d'élaborer des expressions booléennes complexes

```
>> and(binlist(X, [3,4,5]), bcc(X,4.5, 10)).

X = 5.
```

Citons pêle-mêle quelques unes de ces relations : $bimpl$, bor, $band$,..., bcc, bco,..., ble, blt, bge, bgt, beq, $bdif$[42] , $binlist$, $bint$, $bnint$, ...

42. $beq(X, Y)$ et $bdif(X, Y)$ sont respectivement les relations rendant la valeur de vérité des contraintes $eq(X, Y)$ ($X = Y$) et $dif(X, Y)$, si elles étaient posées.

1.2.18 Contraintes sur les arbres

Pour poser les contraintes portant sur les arbres (pour les tester ou les contraindre), on a à sa disposition un certain nombre de relations, liées aux sous-domaines de Prolog IV.

On trouvera la plupart du temps une relation d'appartenance à un sous-domaine nommé de Prolog IV, ainsi qu'une relation de non-appartenance à ce sous-domaine. Le pseudo-terme $nxxx(X)$ contraint X à ne pas vérifier xxx, donc à ne pas appartenir au sous-domaine xxx.

On trouve aussi les relations (qui commence par b) dont le pseudo-terme associé retourne une valeur booléenne qui represente la valeur de vérité de la contrainte obtenue en enlevant la première lettre (le b).

Appartenance aux arbres finis : $finite$, $infinite$, $bfinite$, $binfinite$.

Appartenance aux listes : $list$, $nlist$, $blist$, $bnlist$.

Appartenance aux feuilles : $leaf$, $nleaf$, $bleaf$, $bnleaf$.

Appartenance aux identificateurs : $identifier$, $nidentifier$, $bidentifier$, $bnidentifier$.

Appartenance aux nombres réels : $real$, $nreal$, $breal$, $bnreal$.

Quelques requêtes

```
>> X = f(f(X)).

X = f(X).

>> X ~ finite, X = f(f(X)).

false.

>> X = [A] n finite.

X = [A],
A ~ finite.

>> B = breal(B).

B = 1.

>> B = leaf n list.

B = [].
```

1.2.19 Contraintes et sous-domaines

Certaines relations permettent de contraindre leur argument à appartenir à un sous-domaine privilégié de Prolog IV, de même nom. Ceci nous amène à citer ces sous-domaines :

$tree$, $finite$, $list$, $nlist$, $leaf$, $nleaf$, $identifier$, $nidentifier$, $real$, $nreal$, $cc(r_1, r_2)$, $oo(r_1, r_2)$, $\ldots$, ou les r_i sont des nombres représentables par des nombres flottants.

Ce sont aussi bien des noms de relations que des noms de sous-domaines, elles peuvent donc apparaître dans les réponses de Prolog IV, avec la relation *compatible* ($X \sim sousDomaine$). Certaines d'entres-elles sont des négations d'autres ($nxxx(X)$ indique que X n'appartient pas au sous-domaine xxx .)

Une intersection de sous-domaine étant un sous-domaine, la relation n peut intervenir dans les sorties.

Dans le mode «unions d'intervalles», Prolog IV est susceptible d'afficher des unions d'ensembles (la relation u). C'est notamment le cas pour les résultats provenant de contraintes sur les réels.

NOTES :

- int et $nint$ ne sont pas des sous-domaines, seulement des contraintes.
- $infinite$ n'est pas un sous-domaine, seulement une contrainte.

1.3 L'environnement de programmation

On décrit dans cette section diverses possibilités du logiciel qui ne font pas partie du langage proprement dit. Il peut s'agir aussi bien de primitives que de modes de fonctionnement et de paramétrage.

1.3.1 Quelques options de la ligne de commande

La plupart des options de la ligne de commande concerne le paramétrage des piles et espace de travail utilisés par Prolog IV. D'autres options positionnent des modes de fonctionnement de Prolog IV.

Ces options peuvent tout aussi bien être fournies à Prolog IV lorsqu'il démarre d'un terminal que lancé sous son interface graphique (où ces options peuvent être données au travers d'un dialogue de préférences).

Dans ce qui suit, on appelle cellule un double mot-machine[43].

Voici quelques unes de ces options, avec entre crochet la valeur par défaut donnée par PrologIA, lorsque cette information est appropriée. Nb est un entier positif à fournir.

`-help`	Affiche la liste des options disponibles.
`-heap` Nb	Attribue à la pile $heap$ la place pour nb cellules [700000].
`-trail` Nb	Nb double-cellules pour la pile $trail$ [400000].
`-env` Nb	Nb cellules pour la pile env [50000].
`-choice` Nb	Nb cell. pour la pile $choice$ [50000].
`-global` Nb	Nb cell. pour la zone statique (consult,record,...) [500000].
`-local` Nb	Nb cell. pour la pile locale (unification, solveur de contraintes) [5000].
`-work` Nb	Nb cell. pour la zone temporaire (calculs rationnels, Gauss) [50000].
`-nbpcindex` Nb	Nb maximum de littéraux pour les programmes compilés (Pcode) [10000].
`-union`	Démarre Prolog IV en mode « unions d'intervalle » [intervalles «simples»].
`-iso`	Démarre Prolog IV en mode «iso étendu» [mode «prolog4»].

43. Une cellule occupe donc huit octets sur une machine 32 bits.

1.3.2 Prolog IV et la norme ISO

Prolog IV respecte la norme ISO, tout en l'étendant, les primitives de la norme sont implantées[44] et les mécanisme de base sont conformes à ce qu'édicte le standard.

Toutefois, afin de ne pas pénaliser lourdement par une syntaxe rigide l'usage des innovations qui caractérisent davantage Prolog IV que sa seule adhérence à la norme, deux modes de fonctionnement de Prolog IV sont disponibles, et on choisira son mode en fonction de ses propres goûts, selon que l'on souhaite suivre le standard de près ou au contraire choisir le mode «naturel» de Prolog IV.

Depuis le début de ce tutorial, le mode de fonctionnement est le mode « `prolog4` »[45] que nous ne saurons jamais assez vous conseiller d'adopter (c'est ce mode qui est positionné par défaut au démarrage de Prolog IV). Passer de l'un à l'autre est toutefois facile :

```
>> iso.

true.

?-
```

On remarque que l'invite est maintenant devenue «`?-`».

Pour revenir dans le mode naturel de Prolog IV dit mode `prolog4` :

```
?- prolog4.

true.

>>
```

Quelles sont les différences entre ces deux modes ? Les voici, en désordre :

– Les pseudo-termes ne sont pas compris par `consult` (ou `compile`) lorsque l'on est en mode iso. Ceci impose la notation relationnelle comme la seule permettant d'écrire des contraintes sous ce mode. La notation fonctionnelle ne construit que des termes. Les pseudo-termes restent donc de vrais termes, non évalués. Par exemple :

  ```
  ?- X = 3 + X, functor(X, N,A).

  A = 2,
  N = (+),
  X = 3+X.
  ```

  Outre la surprise que procurerait le succès d'une telle équation numérique si c'en était une, on trouve à l'aide de la primitive `functor`[46] que X est un arbre dont le nœud principal est étiqueté par + et d'arité deux, et pas du tout un nombre ni une variable numérique.

– Les nombres ayant la syntaxe d'un nombre flottant sont codés comme des nombres flottants IEEE lorsque l'on est en mode « iso », et non

44. Celles qui ne le sont pas encore dans une première version le seront rapidement.

45. Par opposition au mode «`iso`».

46. functor est une primitive de la norme qui dans notre cas retourne le nom et l'arité du nœud principal du terme, donné en premier argument.

pas comme des nombres rationnels en précision parfaite du mode `prolog4`.

```
?- X  = 1.00000000000000000000000000001.

X = 1.0.
```

– La forme $nom/arite$, fréquente lors de la gestion de règles dans la norme ISO, n'est pas disponible dans le mode `prolog4` (cette forme est interprétée comme une division retardée dans le solveur linéaire). Dans cet exemple, la contrainte $divlin(X, toto, 4)$ (de raccourci /) est fausse car $toto$ n'est pas un nombre.

```
>> X = toto/4.

false.
```

1.3.3 Les entrées/sorties de Prolog IV

Un ensemble de primitives qui gèrent les entrées/sorties est disponible en Prolog IV. Ces primitives permettent :

– l'ouverture de fichiers, que ce soit en lecture ou en écriture (`open`),

– la lecture de terme dans un fichier ouvert en lecture (`read`),

– l'écriture de caractères ou de termes dans un fichier ouvert en écriture (`write` et `nl`),

– la fermeture de fichiers (`close`).

– d'obtenir des informations sur les fichiers ouverts.

Introduisons approximativement la notion de descripteur de flot[47]. Un descripteur de flot est une entitée prolog qui permet de référencer, dans un programme, tout fichier ouvert. C'est bien sûr un terme prolog[48].

Montrons ceci sur un exemple. Supposons qu'il existe un fichier nommé `monfichier.p4` dans le répertoire courant, contenant le texte suivant :

```
toto(X,h(u),X).
[a,b,3333, 44444444444444,
555555555555, X].
```

Essayons de lire les deux termes (se terminant chacun par un point) de ce fichier ; on s'y prend de la manière suivante :

```
>> open('monfichier.p4', read, S), read(S,T1), read(S, T2), close(S).

A ex
T2 ~ [a,b,3333,44444444444444,555555555555,tree],
T1 = toto(A,h(u),A),
S = 4443940,
A ~ tree.

>>
```

47. Un flot d'entrée est un «endroit» (fichier par exemple) ou sont disponibles un certain nombre de caractères. Un flot de sortie est quant à lui un «endroit» ou on peut déposer des caractères.

48. C'est un nombre dans l'implémentation actuelle de Prolog IV.

la requête contient un appel à `open`, qui est la primitive générale d'ouverture de flot à qui on précise qu'on veut lire (par le mode *read*) dans le fichier `monfichier.p4`[49], suivie de deux appels à la primitive `read` qui lit un terme[50], on ferme enfin le descripteur de flot à l'aide de la primitive `close` de fermeture de flot. Nous avons récupéré nos deux termes dans les variables `T1` et `T2`. La variable `S` contient le descripteur de flot, et la valeur qu'elle possède après l'appel à la primitive `open` n'est guère destinée qu'à être transmise aux autres primitives d'entrée/sortie. Elle est sans doute différente dans votre session.

Essayons maintenant d'écrire des caractères ou des termes dans un fichier `ecr.p4`. On peut faire :

```
>> open('ecr.p4', write, S),
      write(S,'Salut tout le monde, voici un terme :'), nl(S),
      write(S, toto(h,10/20)), nl(S),
      close(S).

S = 4443940.

>>
```

On reconnait la primitive `open`, et on lui précise cette fois-ci que le fichier doit être ouvert pour y écrire[51] (par le mode *write*), ensuite nous retrouvons de vieilles connaissances en les primitives `write` et `nl` rencontrées au tout début de ce chapitre. Elles sont utilisées de façon légèrement différentes, avec un argument supplémentaire au début qui est le descripteur de flot retourné par `open`. On termine encore une fois par un appel à `close`.

Le fichier `ecr.p4` visualisé par ailleurs, contient maintenant les deux lignes :

```
Salut tout le monde, voici un terme :
toto(h,1/2)
```

1.3.4 L'entrée des règles

Nous avons tout au long de ce tutorial abordé de façon quelque peu légère l'entrée des règles sous Prolog IV, puisque l'accent a davantage été porté sur les possibilités du langage par le biais des contraintes prédéfinies. Nous avons rapidement vu les primitives `consult` et `reconsult` permettant de « donner » des règles à Prolog IV. Nous les rappelons ici, avec leurs différentes formes.

Auparavant, quelques notions élémentaires sur les règles prolog.

Paquets de règles

On appelle nom et arité d'une règle le nom et l'arité de la tête de cette règle.

On appelle paquet de règles un ensemble de règles de même nom et arité. Un paquet est donc parfaitement repéré par ce couple $(nom, arit\acute{e})$. L'usage veut

49. Le nom de fichier doit être un atome dans l'implantation actuelle de Prolog IV. Comme le nom `monfichier.p4` ne contient pas que des lettres ou chiffres, il faut le mettre entre apostrophes (`'...'`).

50. `read` ne peut lire qu'un terme suivi d'un point, suivi d'un espace ou retour-chariot. Ceci est du à la richesse (complexité ?) syntaxique des termes prolog, qui impose une convention pour arrêter l'analyse d'un terme prolog en cours de lecture.

51. Le fichier est créé s'il n'existe pas déjà, préalablement effacé sinon.

que les règles de même nom et arité soit regroupées dans un paquet, et non pas éparpillées dans un ou plusieurs fichiers [52].

Un nom de paquet sera souvent noté $nom/arité$.

Compilation, consultation

Il existe en Prolog IV trois façons d'entrer des règles :

– en compilant un fichier,

– en consultant un fichier (ou la console, qui n'est autre chose qu'un fichier spécial toujours ouvert),

– en assertant des règles (à partir de termes) pendant l'exécution d'un programme Prolog IV.

Il existe deux formes de codage de règle en Prolog IV, dues à la présence d'impératifs techniques contradictoires :

– la forme compilée (les primitives de la famille `compile`). Elle a pour avantage la vitesse d'exécution de ces règles, une fois appelées. Son inconvénient est l'impossibilité de traduire ces règles en termes, et de modifier dynamiquement les paquets de règles ainsi compilés (autrement qu'en les détruisant).

– la forme interprétée (ou assertée) (les primitives des familles `consult` ou `assert`). Son avantage est la possibilité de traduire ces règles en termes, et de modifier dynamiquement les paquets de règles (en ajoutant ou supprimant une règle donnée du paquet). Son inconvénient est une plus grande lenteur d'exécution, ainsi qu'une plus grande consommation mémoire.

Malgré ce qui a été montré tout au long de ce chapitre, c'est le mode d'entrée de règles par compilation qui est fortement préconisé. Il est en effet beaucoup plus rare d'avoir besoin d'une gestion dynamique de règles.

Les primitives de lecture de règles

`consult` : lit une suite de règles dans l'entrée courante. Si un paquet de règles lu avait préalablement été entré dans Prolog IV, une erreur indiquant une tentative de redéfinition surviendrait. Si on souhaite vraiment redéfinir ce paquet, il faut utiliser la primitive `reconsult`.

`consult(`*`fichier`*`)` : se comporte de même que `consult` sans argument, mais les paquets de règles sont lus dans un fichier de nom *fichier*.

`reconsult` : lit une suite de paquets de règles dans l'entrée courante. Ceux qui existaient déjà avec ce même nom et arité sont détruits si ce sont des règles utilisateurs [53] seulement.

`reconsult(`*`fichier`*`)` : se comporte de même que `reconsult` sans argument, mais les paquets de règles sont lus dans un fichier de nom *fichier*.

`compile(`*`fichier`*`)` : se comporte un peu comme consult, mais utilise la compilation des règles comme forme de codage interne.

52. Il est possible pour les rares cas ou c'est indispensable d'éparpiller des règles au moyen de déclarations par des directives. Ce n'est peut-être pas implanté dans la version actuelle. . . .

53. Ce sont toutes les règles qui ne font pas partie du système Prolog IV.

`recompile(`*`fichier`*`)` **:** se comporte un peu comme reconsult, mais utilise la compilation des règles comme forme de codage interne.

Dans tous les cas :

- La fin de fichier ou le terme «`end_of_file.`» arrêtent le lecteur de règles.
- Les règles sont codées en mémoire.
- Tenter de redéfinir des règles prédéfinies amène à une erreur.

Note : On ne peut pas compiler de règle dans la console, il faut passer par un fichier[54].

1.3.5 Factorisation des sorties

En positionnant cette option, l'affichage des sorties est davantage condensé en factorisant les sous-termes, quitte à introduire des variables existentielles (qui n'étaient donc pas dans la requête).

```
>> record(q__factorize, 1).

true.

>> X = f(Y), Y = f(X).

X = Y,
Y = f(Y).

>> X = f(g(2),g(2)).

A ex
X = f(A,A),
A = g(2).

>>
```

Au lieu des réponses «non factorisées», ce qu'on obtient par défaut :

```
>> record(q__factorize, 0).

true.

>> X = f(Y), Y = f(X).

X = f(X),
Y = f(Y).

>> X = f(g(2),g(2)).

X = f(g(2),g(2)).

>>
```

54. On peut cependant, lorsqu'on est sous unix et dans un terminal `tty` (comme une fenêtre `xterm` ou `cmdtool`) utiliser le pseudo-fichier `/dev/tty`, comme dans `compile('/dev/tty')`, qui attendra des buts dans le terminal, et ce jusqu'à la fin de fichier (généralement obtenue en tapant `CTRL-D`)

1.3.6 Problèmes

Il est décrit dans cette section quelques pièges et problèmes se présentant de temps à autres, le plus souvent par inattention.

Mélange de solveurs numériques

On cherche à calculer les valeurs possibles pour X à partir d'une formule numérique.

```
>> X = cc(1,2).*.3 - exp(1).

X ~ real.
```

La réponse n'est pas du tout satisfaisante. On pouvait s'attendre à trouver pour X un domaine numérique plus fin, plutôt que l'ensemble des réels. En fait, en regardant plus attentivement la requête, on peut se rendre compte que le signe $-$, qui est le raccourci de la relation $minuslin$ fonctionnant dans le solveur linéaire, a été utilisé au sein d'une contrainte sur les réels, ce qui est fortement déconseillé. Bref, **il ne faut pas mélanger des contraintes linéaires et des contraintes sur les réels.**

La correction consiste simplement à utiliser le signe `.-.` qui est le raccourci de la relation $minus$, la soustraction dans le solveur sur les réels.

```
>> X = cc(1,2).*.3.-.exp(1).

X ~ oo('>0.281718','>3.2817182').
```

La même faute courante :

```
>> X = -pi.

X ~ real.
```

donne un résultat vraiment peu informatif. Il a été utilisé le moins unaire du solveur linéaire au lieu du moins unaire du solveur sur les réels. Il fallait faire :

```
>> X = .-.pi.

X ~ oo(-'>3.1415927',-'>3.1415925').
```

Qui est bien l'approximation de $-\pi$.

Débordement en mode «union»

Le but de cette partie est essentiellement de montrer que les contraintes en rapport avec les entiers ainsi que les fonctions périodiques peuvent faire déborder Prolog IV en mode «union».

Tapons les requêtes suivantes :

```
>> set_prolog_flag(interval_mode, union).

true

>> X ~ int, X ~ cc(0,10).

error: heap_overflow
>>
```

On a demandé l'ensemble des entiers compris entre 1 et 10, et on obtient en fin de compte une erreur de débordement de la pile principale. Que c'est-il passé ?

Et bien, Prolog IV a tenté de construire l'union constituée par l'ensemble des entiers, tout au moins ceux qui sont dans l'approximation de l'ensemble des réels, c.à.d. les entiers qui sont représentables par des nombres flottants. Ces entiers sont en nombre fini, mais il n'est pas possible de représenter cet ensemble en mémoire, tout au moins avec les configurations matérielles qu'on trouve fréquemment sur le marché (de quelques dizaines de mégaoctets de mémoire vive).

Il faut donc, pour éviter de tomber sur ce débordement de mémoire, penser à réduire *avant* le domaine de la variable que l'on souhaite contraindre à être entière. Dans la plupart des cas, il suffira de réordonner des contraintes afin que les contraintes « gloutonnes » soient posées après celles qui effectuent de grosses coupes dans le domaine des variables à contraindre. On a maintenant ce qu'on veut :

```
>> X ~ cc(0,10), X ~ int.

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>>
```

Quelles sont les contraintes « gloutonnes » ?

Ce sont les contraintes faisant intervenir les entiers, ainsi que celles en rapport avec les fonctions périodiques[55] . La liste explicite est celle-ci : int, $nint$, $bint$, $bnint$, $floor$, $ceil$, sin, cos, tan et cot.

Il faut tout de même préciser que le compilateur réordonne tout seul les contraintes au sein d'un pseudo-terme. On peut donc sans crainte tout aussi bien écrire les deux requêtes :

```
>> X ~ cc(0,10) n int.

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>> X ~ int n cc(0,10).

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>>
```

Les requêtes sont transformées avec le compilateur. Les programmes le sont si les primitives de la famille `compile` sont utilisées. Les primitives de la famille `consult` ne réordonnent pas les contraintes.

Insistons sur le fait que ces débordements ne peuvent se produire avec ces contraintes qu'en mode « union d'intervalles ».

1.3.7 La mise au point de programme (Déboguage)

Prolog IV dispose d'un débogueur facilitant la mise au point des programmes. Celui-ci implémente le modèle des boîtes que l'on trouve dans certains

55. C'est de façon générale les contraintes dont la fonction associée ou sa réciproque est non-continue.

systèmes prolog, avec quelques améliorations en rapport avec les nouveautés de Prolog IV. Bien qu'il puisse fonctionner dans un environnement textuel pauvre comme les fenêtres `tty` (du genre `xterm` ou `cmdtool`), c'est sous l'environnement graphique de Prolog IV que le débogueur acquiert sa pleine puissance et sa simplicité d'utilisation. On a alors sous cet environnement la possibilité de suivre le déroulement de l'exécution dans les fichiers sources.

Tout ceci est décrit dans les chapitres « Environnement » et « Environnement Graphique ».

1.4 De Prolog III à Prolog IV

La présente section est une rapide introduction à Prolog IV, plus particulièrement destinée aux utilisateurs de Prolog III. Nous y décrivons donc les différences majeures entre Prolog III et Prolog IV.

1.4.1 Prolog IV est Prolog

La première affirmation importante concernant Prolog IV est :

"Prolog IV est Prolog"

Bien entendu, cela est vrai d'un point de vue historique puisque les concepts à la base de Prolog IV découlent directement des principes de la première version de Prolog, au début des années 1970. Toutefois, l'importance de notre affirmation initiale provient principalement du fait que Prolog IV est une extension de Prolog qui est conforme à la norme ISO pour Prolog de 1995. Pour les utilisateurs de Prolog, cela signifie qu'un programme développé en Prolog standard peut être compilé et exécuté en Prolog IV sans changement. Par exemple, le programme suivant est un programme Prolog IV et son exécution donnera le résultat défini par la norme ISO :

```
foo([X|L]) :-
    X =.. [f,1,Y] ,
    ( Y = 1 -> bar1(Y) ; bar2(Y) ) ,
    foo(L).
```

Notre affirmation a un sens complètement différent pour les utilisateurs de Prolog III. Puisque Prolog III n'est pas une extension de la norme ISO, le programme Prolog III suivant n'est donc *PAS* un programme Prolog IV :

```
Sumlist(<>,x) -> { x = 0 } ;
Sumlist(<x>.l,x+s) ->
  Sumlist(l,s) ;
```

En fait, il y a bien des différences entre cet échantillon de code et le standard ISO. Les quatre différences principales sont :

– Les symboles -> et `;` doivent être remplacés par les symboles `:-` et `.` et la syntaxe des identificateurs et des variables a changé (nous retrouvons donc une syntaxe très proche de la syntaxe Edimbourg de Prolog III).

– Il n'est plus possible dans le standard d'utiliser les accolades pour entourer le système de contraintes, qui devra donc être intégré dans le programme.

– Les listes Prolog III (tuples) ne faisant pas partie du standard, les listes Edimbourg (utilisant `[` et `]`) doivent être utilisées systématiquement.

– Le symbole d'addition + n'est pas interprété en Prolog standard, ce qui signifie que le terme `x+s` est *a priori* interprété comme l'arbre `+(x,s)` .

En Prolog IV, toutes ces caractéristiques ont changé, de manière à être conforme à la norme ISO. Le programme qui calcule la somme des éléments d'une liste peut donc s'écrire de la manière suivante en Prolog IV(en mode ISO) :

```
sumlist( [] , 0 ).
  sumlist( [X|L] , S1 ) :-
    pluslin( S1 , X , S ) ,
    sumlist( L , S ).
```

Ici, nous avons utilisé le *prédicat prédéfini* `pluslin`, qui ajoute au système courant de contraintes la contrainte $S1 = X + S$. La principale raison de ce changement est que le standard ISO-Prolog indique clairement que les termes présents dans les littéraux doivent être interprétés comme des arbres. Avec cette restriction, la seule solution pour étendre le langage est donc d'ajouter un ensemble de prédicats prédéfinis comme `pluslin`, qui ont une sémantique particulière définie dans le langage. Ainsi, les contraintes de Prolog IV sont définies comme des prédicats prédéfinis dans le système (il y en a plus de 120).

Les utilisateurs de Prolog III sont en droit de se plaindre du fait qu'un tel programme est beaucoup moins lisible que la version Prolog III du même programme. Nous sommes d'accord avec eux, et c'est pourquoi nous avons ajouté au langage une syntaxe alternative, qui nous permet de calculer la somme des éléments d'une liste de la manière suivante :

```
sumlist( [] , 0 ).
  sumlist( [X|L] , X + S ) :-
    sumlist( L , S ).
```

La différence majeure avec le programme précédent réside dans le traitement de l'addition, qui ici ressemble fort à l'addition de Prolog III. En fait, l'argument `X + S` est un *pseudo-terme* Prolog IV ; pour arriver à cette écriture élégante (au sens Prolog III), il y a en fait une version intermédiaire qui est la suivante :

```
sumlist( [] , 0 ).
  sumlist( [X|L] , pluslin(X,S) ) :-
    sumlist( L , S ).
```

Dans cette version, nous voyons que l'addition est écrite de manière fonctionnelle, mais en conservant l'identificateur initial. En mode Prolog IV, cet identificateur (`pluslin`) est interprété de manière spéciale quand il apparaît en tant que foncteur à deux arguments : une transformation de programme est appliquée automatiquement, dans laquelle l'expression `pluslin(X,S)` est remplacée par une nouvelle variable[56] `New`, et le littéral `pluslin(New,X,S)` est ajouté en début de la queue de règle, ce qui nous donne le programme initial.

Ces pesudo-termes peuvent être utilisés pour toutes les contraintes, ce qui permet de simplifier de nombreuses expressions. Enfin, le fait de remplacer le

56. Soit une variable qui n'apparaît pas dans la règle courante.

foncteur `pluslin` par l'opérateur + n'est que du "sucre syntaxique" permettant de rendre l'expression plus simple à lire. De nombreux exemples d'utilisation des pseudo-termes apparaissent tout au long du présent manuel.

Il existe donc deux modes distincts d'interprétation des programmes, un qui est compatible avec la syntaxe Prolog définie dans la norme ISO, et un autre qui autorise l'utilisation de pseudo-termes, facilitant l'écriture de programmes avec contraintes. Il existe donc des moyens de passer d'un mode à l'autre. Sur la ligne de commande, pour passer en mode ISO, il faut taper :

```
>> iso.

   true.

  ?-
```

On note ici que l'invite de commande change de >> en mode Prolog IV à `?-` en mode ISO. Pour revenir en mode Prolog IV, il faut taper :

```
?- prolog4.

   true.

  >>
```

Pour conclure, nous pouvons résumer les points essentiels de cette section :

– En Prolog IV, les contraintes sont ajoutées par le biais de prédicats prédéfinis.

– En plus du mode ISO, il existe un mode spécifique Prolog IV, qui permet d'utiliser des pseudo-termes, et donc d'écrire les contraintes de manière plus élégante.

1.4.2 Prolog IV hérite de Prolog III

Une deuxième affirmation importante à propos de Prolog IV est :

"Prolog IV hérite de Prolog III"

De nombreux programmes Prolog III peuvent être portés en Prolog IV en effectuant un effort minimal, souvent d'ordre syntaxique. Par exemple, un solveur linéaire est inclus dans Prolog IV, qui peut manipuler les mêmes contraintes que le solveur linéaire de Prolog III. La requête suivante permet donc de résoudre le système d'équations linéaires $\{X + Y = 3, X - Y = 1\}$:

```
>> X + Y = 3 , X - Y = 1 .

   Y = 1,
   X = 2.

  >>
```

Il existe bien d'autres caractéristiques de Prolog III qui existent toujours en Prolog IV. Par exemple, les prédicats de gestion des contraintes linéaires, comme `enum` et `lower_bound` existent toujours. A l'inverse, certaines caractéristiques ont dû être modifiées, généralement pour assurer la compatibilité avec la norme ISO. Par exemple, les opérateurs $>$ et $<$ sont prédéfinis dans la norme ISO, comme étant des comparaisons de nombres après évaluation des arguments par la primitive `is` ; on peut donc avoir l'échange suivant :

```
>> X > 1 .

   error: error(instantiation_error,(is)/2)
  >>
```

A la place de $>$, il faut donc utiliser la contrainte `gtlin`, et les autres contraintes de la famille (`gelin`, `ltlin` et `lelin`). L'échange normal serait donc le suivant :

```
>> gtlin(X,1).

   X ~ real.

  >>
```

On note ici une autre différence entre Prolog III et Prolog IV, qui est que Prolog IV n'affiche en sortie que les sous-domaines associés aux variables, et pas les contraintes du système [57].

Le solveur linéaire n'est pas le seul solveur de Prolog III que l'on retrouve en Prolog IV. Une opération de concaténation de listes est également disponible en Prolog IV, et elle permet d'écrire un programme élégant pour renverser le contenu d'une liste de la manière suivante :

```
reverse([] , []).
  reverse([X] o L1 , L2 o [X]) :-
    reverse(L1,L2).
```

L'opération de concaténation est maintenant noté `o` (o minuscule), puisque le point (`.`) est réservé dans la norme ISO. Le programme ci-dessus se comporte exactement comme le programme Prolog III équivalent tant que la longueur d'un de ses arguments est connue. Par contre, il peut avoir un comportement différent dans les autres cas.

En fait, comme dans tout héritage, certaines des fonctions héritées ont été modifiées, dégradées, voire perdues. Le solveur de listes en est un parfait exemple. Il existe toujours des contraintes permettant de contraindre la longueur d'une liste ou de concaténer deux listes, mais leur sémantique a été modifiée. Ici, le but était d'unifier les listes Prolog III avec les listes classiques de la norme ISO, et de rendre les opérations sur les listes symétriques. Cela nous a amené à diminuer de manière significative la puissance du raisonnement sur les listes [58]. Veuillez consulter les exemples ou le reste du présent manuel pour de plus amples informations sur les contraintes sur les listes.

Enfin, certains changements sont encore plus importants. Le plus important concerne le solveur booléen, qui n'existe plus en tant que tel dans Prolog IV. En effet, il perd son aspect complet et formel en étant plongé dans le solveur sur les réels, et gagne en puissance par les possibilités de mélange de contraintes numériques, ce qui n'était pas possible en Prolog III.

Il demeure donc possible de traiter des contraintes booléennes, un peu affaiblies, d'un côté et largement ouvertes d'un autre, en utilisant les contraintes sur les réels présentées dans la section suivante.

57. Ces différences sont discutées en détail à la fin du présent chapitre.

58. Par exemple, en Prolog III, les équations liant les longueurs des listes apparaissant dans des concaténations étaient traitées par le solveur linéaire ; ce n'est plus le cas en Prolog IV.

Pour conclure cette section, nous pouvons retenir les faits suivants :

– Il existe toujours en Prolog IV un solveur linéaire et un solveur sur les listes, mais le solveur booléen (un peu affaibli) est immergé dans le solveur numérique.

– Dans les solveurs qui ont été repris de Prolog III, des adaptations ont été nécessaires, et le comportement de certains programmes peut avoir changé.

1.4.3 Prolog IV est nouveau

Une troisième affirmation à propos de Prolog IV, et probablement la plus importante, est :

"Prolog IV est nouveau"

Prolog IV a de nombreuses caractéristiques qui peuvent être utilisées pour résoudre une large gamme de problèmes que Prolog III ne pouvait pas résoudre. De plus, comme nous l'avons déjà vu brièvement, certaines des fonctionnalités de Prolog III ont été entièrement révisées, de manière à donner à nos utilisateurs un système plus efficace et plus simple à utiliser.

Le changement le plus visible est l'ajout d'un nouveau solveur, basé sur l'arithmétique des intervalles, qui permet de manipuler toutes sortes de contraintes portant sur des nombres réels, sur des entiers et sur des booléens. Ce solveur est fondamentalement différent du solveur rationnel (ou linéaire) hérité de Prolog III, pour de nombreuses raisons :

– Ce solveur peut manipuler toutes sortes de contraintes non-linéaires, de la simple multiplication aux fonctions transcendentales comme `cos`.

– Des contraintes sur les réels, les entiers et les booléens peuvent être librement mélangées.

– Ce solveur est basé sur des *approximations*, ce qui implique que le solveur n'est pas complet [59] ; ses résultats doivent donc être analysés avec de plus grandes précautions que ceux du solveur linéaire.

Commençons par introduire de manière très informelle la façon de fonctionner de ce solveur. Il est destiné à raisonner sur les nombres réels, mais ces nombres ne peuvent pas être manipulés directement ; le solveur effectue donc ses calculs sur des intervalles de nombres réels qui ont la propriété suivante : leurs bornes doivent être représentables exactement par des nombres flottants [60]. Par exemple, Prolog IV contient un prédicat prédéfini `sqrt`, qui peut être utilisé pour calculer la racine carrée d'un nombre. La requête suivante a donc le résultat attendu :

```
>> X = sqrt(4).

   X = 2.

  >>
```

A l'opposé, la requête suivante a des résultats plus surprenants :

59. Un solveur est complet si pour tout système de contraintes, il est capable de déterminer si le système est soluble ou non.

60. On utilise ici les nombres flottants simples définis dans la norme IEEE 754.

```
>> X = sqrt(2).

   X ~ cc(`>1.4142135`,`>1.4142136`).

  >>
```

Ce résultat signifie que le système n'est pas capable de calculer un résultat exact, mais qu'il est par contre capable de déterminer que la solution est dans l'intervalle :

$[1.41421353816986083984375, 1.414213657379150390625]$.

Les deux nombres ci-dessus sont les valeurs exactes (avec toutes les décimales) des nombres flottants simples qui encadrent $\sqrt{2}$. L'écriture renvoyée par Prolog IV est une écriture simplifiée mais non équivoque. L'objet `` `>1.4142135` `` désigne en fait le plus petit nombre flottant supérieur à 1.4142135 ce qui permet de désigner un nombre flottant précisément sans avoir à donner toutes ses décimales.

On voit donc que le résultat n'est pas précis. Si on complique un peu notre requête, nous pouvons avoir :

```
>> X = sqrt(2), X = sqrt(2.0000001).

   X ~ oo(`>1.4142135`,`>1.4142136`).

  >>
```

Ce résultat est bien plus surprenant. En effet, il est clair pour nous que les nombres $\sqrt{2}$ et $\sqrt{2.0000001}$ sont des nombres différents. Toutefois, la faible précision des nombres flottants simples ne permet pas de les différencier, puisqu'ils font partie du même intervalle. Cette requête est donc un exemple de l'incomplétude du langage, dans la mesure le système n'a pas pu détecter l'incohérence d'un système de contraintes. Bien entendu, ce problème disparait dès que la précision de calcul est suffisante :

```
>> X = sqrt(2), X = sqrt(2.000001).

   false.

  >>
```

Ici, nous avons simplement changé la constante, en lui enlevant un 0. Ce changement suffit au système pour se rendre compte de l'incohérence du système.

Comme indiqué précédemment, le nouveau solveur permet également de manipuler des contraintes sur les entiers, sans recourir systématiquement à l'énumération comme le faisait Prolog III. En fait, une seule contrainte nous sert pour cela, la contrainte `int`. Considérons par exemple la requête suivante :

```
>> X ~ cc(1.5,2), Y ~ cc(2.25,2.75), Z = X .+. Y, int(Z), ge(Y,2.5).

 Z = 4,
 Y = 5/2,
 X = 3/2.

>>
```

Décrivons en détail la résolution de ce problème pour montrer comment fonctionne le solveur :

- La première contrainte indique que $X \in [1.5, 2]$.
- La deuxième contrainte indique que $Y \in [2.25, 2.75]$.
- Ensuite, en considérant la contrainte d'addition, on infère très simplement que $Z \in [3.75, 4.75]$. Cette inférence provient du fait qu'en additionnant n'importe quel nombre du domaine de X à un nombre du domaine de Y, nous devons obtenir un nombre du domaine de Z, et que le domaine $[3.75, 4.75]$ est le plus petit domaine vérifiant cette propriété.
- Ensuite, en considérant la contrainte `int`, le domaine de Z est réduit à $[4, 4]$ puisque 4 est le seul entier dans le domaine de départ de Z. En reconsidérant les contraintes accumulées et en particulier l'addition, le domaine de Y peut être réduit à $[2.25, 2.5]$ et celui de X à $[1.5, 1.75]$. Pour effectuer la réduction du domaine de Y, le raisonnement est le suivant: nous savons que $X + Y = 4$; Y est maximal quand X est minimal, soit quand $X = 1.5$. La valeur maximale de Y est donc 2.5. Le même raisonnement est appliqué pour chaque borne.
- La dernière étape consiste à considérer la contrainte $Y \geq 2.5$. Cette contrainte réduit le domaine de Y au singleton $[2.5, 2.5]$. Par propagation, le domaine de X se trouve également réduit et devient $[1.5, 1.5]$.
- Enfin, puisque les intervalles sont réduits à un singleton, ils sont traduits en nombres rationnels. Un mécanisme permet de déterminer les valeurs de certains arguments d'une contrainte lorsque suffisamment d'arguments prennent une valeur rationnelle.

Ce court exemple nous a permis de mettre en évidence les mécanismes de base de l'algorithme de résolution :

- Il est basé sur un algorithme de réduction d'intervalles, qui essaie continuellement de réduire les domaines de variables.
- Il est basé sur des calculs de point fixe, dans lequel les contraintes sont reconsidérées pour essayer de réduire les domaines jusqu'à avoir atteint la stabilité.
- Quand un intervalle se réduit à un singleton, une interface avec les autres solveurs permet de vérifier l'exactitude des calculs.

En fait, l'incomplétude de cet algorithme est quelque peu similaire à la situation qui se produisait en Prolog III avec les multiplications retardées. Quand une multiplication restait retardée (et donc pas prise en compte par le solveur numérique), il était impossible d'être sûr que le système courant était soluble. Par contre, quand la solution d'un système de contraintes contenant des multiplications était entièrement figée, nous étions alors certains que la multiplication était vérifiée. En utilisant les contraintes sur les réels de Prolog IV, il faut continuellement faire attention à la même chose. Considérons l'exemple classique où on cherche à résoudre le système insoluble $X \times X = -1$:

```
>> X .*. X = -1 .

   X ~ real.

  >>
```

On voit ici que l'incohérence n'est pas détectée. Pour la détecter, nous pouvons, comme en Prolog III, utiliser une énumération. Cela nous permet de mettre en évidence deux avantages importants du raisonnement sur les réels en utilisant des intervalles :

- Le nombre de "valeurs" possibles (c'est-à-dire d'intervalles de réels à bornes flottantes) est fini, ce qui permet de faire une énumération totale.
- Il est possible d'énumérer sur les nombres réels, c'est-à-dire de considérer tous les intervalles, même ceux qui n'ont pas de bornes entières.

Voyons tout de suite un exemple, en utilisant le prédicat `realsplit`, qui cherche des intervalles de réels :

```
>> X .*. X = -1 , realsplit([X]).

   false.

  >>
```

Ici, l'incohérence est détectée, dans la mesure où le système n'est pas capable de trouver une valeur de X qui vérifie l'équation, ce qui est le résultat attendu.

L'algorithme d'énumération, qui est en fait ici un algorithme de "découpage" des intervalles, est également beaucoup plus puissant que l'énumération de Prolog III, dans la mesure où il offre de nombreuses options permettant de contrôler la manière dont s'effectue le découpage, et permet de travailler simultanément sur plusieurs variables.

Nous n'irons pas plus loin dans cette section, et vous êtes invités à lire le reste du manuel pour de plus amples informations concernant les nouveaux solveurs de Prolog IV. Les informations les plus importantes de cette section sont :

- Un nouveau solveur approximé est disponible pour traiter les contraintes numériques ; il utilise l'arithmétique des intervalles et n'est pas complet.
- Ce solveur permet de mélanger des contraintes sur les nombres réels, les entiers et les booléens.
- Des utilitaires puissants sont mis à votre disposition pour les opérations d'énumération.

1.4.4 Prolog IV ne ressemble pas à Prolog III

La quatrième et dernière affirmation concernant Prolog IV est donc :

"Prolog IV ne ressemble pas à Prolog III"

Par cela, nous voulons dire que l'environnement de Prolog IV ne ressemble pas du tout à l'environnement de Prolog III. Voici les principaux changements :

- Prolog IV est un langage compilé, et non un langage interprété comme Prolog III.
- Prolog IV est livré avec un environnement complet, incluant un éditeur dédié, une console spécifique et un débogueur source interactif.

Nous n'allons pas dans la présente section discuter de l'environnement graphique, mais nous allons insister sur les quelques commandes qui vous permettent de contrôler le compilateur, et donner quelques informations concernant la console.

Ecrivons par exemple un programme `animaux` qui établit un lien entre un nombre de chats, un nombre d'oiseaux, un nombre de pattes et un nombre de têtes. Nous devons faire l'échange suivant :

```
>> consult.

   Consulting ...
   animaux(Chats,Oiseaux,Pattes,Tetes) :-
     Pattes = 4 .*. Chats .+. 2 .*. Oiseaux,
     Tetes = Chats + Oiseaux ,
     int(Chats), int(Oiseaux),
     ge(Chats,0), ge(Oiseaux,0),
     ge(Pattes,0), ge(Tetes,0),
     intsplit([Chats,Oiseaux]).
   end_of_file.
   true.

  >>
```

Le prédicat `consult` est utilisé pour rentrer le programme. Nous avons ici utilisé sa forme interactive (sans argument). Si un argument est fourni, ce doit être un atome qui contient le nom d'un fichier contenant le programme source. ATTENTION, le code inséré par `consult` n'est pas compilé. Pour obtenir du code compilé, il faut placer le code dans un fichier, et utiliser la commande `compile`, qui prend nécessairement un argument (le nom du fichier).

Le programme `animaux` se contente ici de poser les contraintes liant les variables, d'indiquer que nous voulons toujours avoir un nombre entier de chats et d'oiseaux, et que toutes les variables sont positives, avant de faire une énumération[61]. Pour finir la saisie, il faut taper la séquence "`end_of_file.`"[62] . Il n'est pas permis comme en Prolog III de terminer la saisie en entrant un point sur une ligne isolée. Nous pouvons maintenant tester notre programme :

61. Vous pouvez noter ici l'utilisation de `intsplit` qui effectue un découpage en considérant que les variables passées en arguments sont des variables entières.

62. On peut taper Ctrl-D si on est dans un terminal unix, hors de la console Prolog IV.

```
>> Pattes = 10, animaux(Chats,Oiseaux,Pattes,Tetes).

   Tetes = 5,
   Oiseaux = 5,
   Chats = 0,
   Pattes = 10;

   Tetes = 4,
   Oiseaux = 3,
   Chats = 1,
   Pattes = 10;

   Tetes = 3,
   Oiseaux = 1,
   Chats = 2,
   Pattes = 10.

 >>
```

La requête demande toutes les solutions pour lesquelles le nombre de pattes est 10. On obtient donc trois solutions. On note ici une différence avec Prolog III au niveau de l'interface utilisateur. En Prolog III, chacune des solutions était exprimée sous la forme d'un système de contraintes qui pouvait en retour être donné en entrée à Prolog III. Ici, c'est la solution entière du système qui est exprimée sous la forme d'une requête. Les solutions sont donc séparées par des points-virgules, et la dernière est terminée par un point. En recopiant cette solution sur la console, on obtient de nouveau la même solution, à une permutation des variables près.

Prolog IV contient également un mécanisme de *quantification existentielle* qui permet de simplifier des résultats. Nous pouvons par exemple écrire la même requête que précédemment, en stipulant qu'on ne souhaite connaitre les résultats que par rapport à `Chats` et `Oiseaux` :

```
>> Pattes ex Tetes ex Pattes = 10, animaux(Chats,Oiseaux,Pattes,Tetes).

 Oiseaux = 5,
 Chats = 0;

 Oiseaux = 3,
 Chats = 1;

 Oiseaux = 1,
 Chats = 2.

>>
```

L'opérateur `ex` permet donc d'indiquer que son opérande de droite doit être évalué sans chercher une valeur particulière pour son opérande de gauche (qui doit être une variable). Ce mécanisme permet donc d'éliminer des variables de la sortie, mais également d'utiliser des variables “locales” dans des littéraux Prolog.

Enfin au niveau de l'interaction, il est à noter que Prolog IV n'affiche pas comme résultat le système de contraintes résultant, mais seulement les domaines des variables non existentielles apparaissant dans la requête. Ces domaines sont des sous-ensembles de l'ensemble des arbres, dont les types principaux apparaissent dans l'exemple ci-dessous :

```
>> L = [X,Y,Z] o C, Y = f(Y), cc(X,1,pi), int(Z).

   Y = f(Y),
   C ~ list,
   Z ~ real,
   L ~ list,
   X ~ co(1,'>3.1415927').

 >>
```

Ici, les variables L et C sont typées comme des listes, puisqu'elles apparaissent dans une concaténation. La variable *Z* est typée comme étant numérique ; cela paraît plus faible que ce qu'implique la contrainte `int`, mais `int` n'est qu'une contrainte, et non un domaine, et n'apparaît donc pas dans les sorties. Le domaine de la variable X est correct (la valeur de π est bien entendu approximée). Pour Y, nous avons l'équation définissant un arbre infini, non pas parce que c'est une contrainte importante, mais parce que c'est une construction, et que de plus cet arbre est unique, et la valeur de Y est donc entièrement connue. Ces notations demandent un peu d'habitude, mais elles deviennent assez rapidement très informatives sur le déroulement du programme.

Pour terminer ce chapitre destiné aux utilisateurs de Prolog III, voici un pot-pourri de requêtes dont le résultats devrait surprendre, émerveiller ou inquiéter de nombreux utilisateurs Prolog III :

– Une contrainte vraiment pas linéaire :

```
>> X = cos(X) .

   X ~ oo('>0.739085','>0.7390852').

 >>
```

– Les contraintes sont traitées de manière approximée :

```
>> cc(X,0,pi), cos(X) = sin(X).

   X ~ cc(0,'<1.570797').

 >>
```

– Mais un petit "split" règle les problèmes :

```
>> cc(X,0,pi), cos(X) = sin(X), realsplit([X]).

   X ~ oc('>0.7853977','<0.7853985').

 >>
```

– Les variables existentielles servent à simplifier le résultat :

```
>> X ex Y ex cc(X,0,pi), Y = cos(X), Y = sin(X),
      Z = square(Y) .*. 2, int(Z), realsplit([X]).

   Z = 1.

 >>
```