Un Langage Logique Concurrent par Contraintes

Dimitrios Souflis

Septembre 1992

No 92-4

Un Langage Logique Concurrent par Contraintes

Dimitrios Souflis
CERMICS
La Courtine
93167 NOISY-LE-GRAND Cedex
Tél. : 43 04 40 98 - Fax : 43 05 83 06

Résumé

Les langages logiques concurrents par contraintes offrent un modèle général de communication et de synchronisation de processus, par moyen d'une base de contraintes.

Le langage décrit ici est une instanciation des langages logiques concurrents par contraintes. Le domaine d'application de nos contraintes est celui des *termes finis*. Il présente la nouveauté de traiter la déségalité dans la partie *Tell* et la quantification des variables des équations et des diséquations. En plus, les manipulations réflexives usuelles sur les termes peuvent être exprimées directement dans le langage, sans employer des primitives particulières, et un système de modules assure la séparation d'espaces de noms nécessaire pour le génie logiciel.

A Concurrent Constraint Logic Programming Language

Abstract

Concurrent Constraint Logic programming languages offer a general framework of communication and synchronization between processes using a shared Constraint Base.

The language described here is an instantiation of Concurrent Constraint Logic Languages. Our domain is that of *finite trees*. The novel aspects are: the treatment of disequalities in *Tell*, quantification of equalities and disequalities, and the possibility to describe the usual reflective manipulations on terms in the language itself, without need for special primitives. There is also a module system to make software engineering easier.

Ce rapport est issu d'un stage effectué par l'auteur dans le cadre du DEA IARFA.

Table des matières

1 Introduction **1**
- 1.1 La Programmation Logique Concurrente par Contraintes 1
- 1.2 Du point de vue du programmeur 1

2 Les Systèmes de Contraintes **3**
- 2.1 Problèmes Equationnels 3
- 2.2 Equations et Diséquations entre Termes Finis 4

3 La Sémantique Opérationnelle **5**
- 3.1 Clauses à Choix d'Engagement 5
- 3.2 Règles de Transition 5
- 3.3 ALPS 6
- 3.4 Arithmétique 7
- 3.5 Entrées / Sorties 8

4 La Syntaxe **9**
- 4.1 Les Modules 9
- 4.2 Les Identificateurs 11
  - 4.2.1 Les Constantes 11
  - 4.2.2 Les Variables 11
- 4.3 Les Termes 12
  - 4.3.1 Termes Incomplets 13
- 4.4 Clauses 14

5 Exemples de Programmation et de Techniques **15**
- 5.1 Exemples Communs avec autres Langages 15
  - 5.1.1 La Variable Logique 15
  - 5.1.2 Gardes à Ask uniquement 15
  - 5.1.3 Gardes à Ask & Tell 19
- 5.2 Exemples propres à notre langage 22

6 Implémentation **24**
- 6.1 Satisfaction des Contraintes 26

7 Conclusions **29**

1 Introduction

1.1 La Programmation Logique Concurrente par Contraintes

A partir de la fin des années soixante dix, une classe de langages a paru dans le monde de la Programmation Logique. Ces langages (en commençant peut-être par IC-Prolog) visaient à introduire de la concurrence dans Prolog. Cette recherche a donné naissance à une autre sorte de langages, appelés *langages logiques concurrents*, qui expriment des systèmes réactifs et des algorithmes parallèles. Selon E. Shapiro dans [4]:

> A concurrent logic program is a don't care nondeterministic logic program augmented with synchronization. A logic program thus augmented can realize the basic notions of concurrency: processes, communication, synchronization, and indeterminism. [1]

Une branche de la Programmation Logique Concurrente est la Programmation Logique Concurrente par Contraintes (*Concurrent Constraint Logic Programming*), issue du cadre de la Programmation Logique par Contraintes (*Constraint Logic Programming*). Maher [2] a donné une spécification logique des conditions de synchronisation, qui jusque là étaient définies de façon opérationnelle, en se basant sur les notions de la Programmation Logique par Contraintes. Saraswat (voir [3]), un chercheur actif dans le domaine, a développé un cadre de Programmation Logique Concurrente par Contraintes où le calcul se fait par des agents qui communiquent en plaçant des contraintes dans une base commune, et qui se synchronisent en vérifiant la validité de contraintes dans la base de contraintes. Les agents correspondent à des atomes logiques – buts, et les contraintes s'expriment dans une théorie quelconque (termes, $\Re$, booléens). Les contraintes qu'on a utilisées portent sur les *termes finis* comme dans FCP(:), mais avec l'adjonction de la déségalité, qui, d'habitude, n'est pas traitée et une forme limitée de quantification.

1.2 Du point de vue du programmeur

Prolog, en tant que langage de programmation pratique, offre de nombreux avantages:

- Abstraction des données de leur représentation (la raison pour laquelle D. Warren a appelé Prolog "Pointers made easy").

[1] L'*indéterminisme* (le choix arbitraire entre plusieurs voies) est différent du non-déterminisme habituel en Prolog, caractérisé comme "non-déterminisme par ignorance" (le choix exploratoire entre plusieurs voies dont on ignore la valeur).

- Uniformité : il n'y a qu'un mécanisme de traitement de données, l'unification, qui sert pour toute manipulation (passage de paramètres par valeur et par référence, assignation de valeur, test d'égalité, accès aux données composées, construction de données composées ...).
- Décomposition facile d'un problème en sous-problèmes.
- Stratégie de contrôle implicite.

C'est naturel de demander à un langage qui hérite de Prolog les mêmes avantages, ou équivalents. En fait, les langages logiques concurrents sont par rapport à Prolog des langages de plus bas niveau. Un programme concurrent n'est plus une spécification plus ou moins déclarative qui peut être exploitée de diverses façons par l'interpréteur Prolog. On n'a plus droit au *non-déterminisme par ignorance* de Prolog, en laissant l'interpréteur trouver le bon chemin de preuve. Au contraire, on écrit des programmes qui n'ont plus la possibilité d'échouer, car un échec signifie la fin (anormale) du calcul. En général il faut mettre plus de soin dans les programmes concurrents.

Mais on ne perd pas tout. On garde la manipulation facile des données, la possibilité d'engendrer facilement de nouveaux buts, et on gagne par rapport à Prolog le traitement naturel des entrées-sorties. Comme on verra par la suite, les entrées-sorties sont traitées de façon totalement logique en tant que flots. Les langages logiques concurrents sont particulièrement adaptés pour traiter des flots, un mécanisme aussi puissant que les objets à état mutable pour modéliser (et interagir avec) le monde réel (voir [1], *Flots*).

2 Les Systèmes de Contraintes

2.1 Problèmes Equationnels

Une proposition dans notre langage est soit une *contrainte élémentaire*

$$\exists(x, y \ldots)\, N = M$$

ou

$$\forall(x, y \ldots)\, N \neq M$$

(N et M termes), soit un *atome*. On suppose qu'un atome *a(M, N, ...)* représente une relation entre ses arguments, définie à l'aide de règles de réécriture. La résolution d'un but se sert d'un *problème équationnel* pour tester et imposer des contraintes afin de remplacer le but par une suite d'autres buts. Un problème équationnel est une conjonction de contraintes élémentaires. On peut représenter un problème équationnel par deux composants:

1. un système résolu d'équations
2. un système résolu de diséquations. Ce système correspond aux valeurs *non* permises aux variables, et contient des disjonctions implicites.

La forme résolue du système d'équations est un ensemble de couples *(x, M)* où x est une variable et M un terme qui ne contient aucun des x, et aucun des x n'apparaît plus d'une fois. Un couple *(x, M)* correspond à l'équation $\exists e_M\ x = M$ où e_M sont les variables quantifiées existentiellement dans M (variables locales). En outre, M ne peut pas être une variable existentiellement quantifié, $\exists x\, x = M$ étant une tautologie.

La forme résolue du système de diséquations est un ensemble de *négations* des systèmes d'équations sous forme résolue, qu'on note d'habitude comme un ensemble de déségalités de tuples $<$x, y, ...$>\neq<$N, M, ...$>$. Etant la négation d'un système résolu, la déségalité des tuples hérite de toutes les propriétés des systèmes résolus.

Dans notre cadre, on considère un problème équationnel comme une *base de contraintes*. Vis-a-vis d'une base de contraintes Γ, on dit qu' une formule A dans la théorie $\mathcal{T}$

- est *validée* quand $\mathcal{T} \models (\forall)(\Gamma \Rightarrow \exists v_A\ A)$
- est *insatisfaisable* quand $\mathcal{T} \models \neg(\exists)(\Gamma \wedge A)$
- est *satisfaisable* quand $\mathcal{T} \models (\exists)(\Gamma \wedge A)$

où v_A sont les variables dans A. Il reste à définir la théorie $\mathcal{T}$.

2.2 Equations et Diséquations entre Termes Finis

On va travailler dans le cadre du sous-ensemble de la théorie de Clark utilisée dans **Prolog**, en ajoutant des désegalités universellement quantifiées, des égalités existentiellement quantifiées et l'*unification syntaxique*.

Un *terme* dans notre langage est soit un *terme simple* soit un *terme composé*. Un terme simple est une variable ou une constante. Un terme composé est soit un *tuple* soit une *application*. Un *tuple* est un ensemble ordonné de termes dont le premier terme (le *foncteur*) est un terme simple. A la différence de ce qui est usuel en logique, les foncteurs n'ont pas d'arité fixe; tout terme simple peut être utilisée comme foncteur. Les *variables* dans notre théorie peuvent être utilisées même en tant que foncteurs. La *liste*, connue et appréciée dans **LISP** et **Prolog** comme une structure d'aggrégation commode, est un terme particulier qui utilise comme foncteur la constante **cons**, et qui a une deuxième écriture, particulièrement adaptée à son rôle, connue en Prolog. Une liste est soit une variable, soit la constante prédéfinie [], soit **cons(***terme*,*liste***)**.

Notre langage est dotée d'une puissante capacité reflexive: un terme peut être écrit comme une *application*: la juxtaposition d'un foncteur et d'une *liste* d'arguments, elle même pouvant être une variable, ou contenir des variables [2]. *(voir "La Syntaxe - Termes Incomplets").* L'unification qu'on utilise se définit à l'aide des axiomes suivants.

1. $t = t$ Un terme est égal à lui-même.
2. $a \neq b$ Une constante est différente de toute autre constante.
3. $f(t_1, t_2, \ldots t_n) = g(s_1, s_2, \ldots s_n) \Leftrightarrow f = g \wedge t_1 = s_1 \wedge t_2 = s_2 \wedge \ldots \wedge t_n = s_n$
4. $f(t_1, \ldots t_n) \neq g(s_1, \ldots s_m)$ si $n \neq m$
5. $fL_1 = gL_2 \Leftrightarrow f = g \wedge L_1 = L_2$
6. $f(t_1, t_2, \ldots t_n) = gL \Leftrightarrow f = g \wedge [t_1, t_2, \ldots t_n] = L$

[2] La considération derrière cette définition radicale des termes, c'est l'utilisation qu'on veut en faire. L'expérience dans le monde Prolog a montré que beaucoup de programmes pratiques ont besoin de manipuler syntaxiquement les termes, un besoin auquel a repondu par l'introduction des prédicats extra-logiques `arg`, `functor` et `=..`, qui restent totalement en dehors la définition formelle du langage.

3 La Sémantique Opérationnelle

3.1 Clauses à Choix d'Engagement

Chaque "procédure" correspond a (une ou) plusieurs règles logiques de la forme des clauses de Horn. On distingue quatre parties, dont la signification sera expliquée par la suite *(voir aussi La Syntaxe)*.

$$Head \texttt{ :- } Ask : \ Tell \mid Body \ .$$

La lecture logique d'une clause est celle d'une clause de Horn. Les séparateurs spéciaux ':' et '|' sont à cet effet assimilés à des conjonctions logiques. *Head* est un seul atome. *Ask* et *Tell* sont des conjonctions de contraintes élémentaires, et *Body* est une conjonction d'atomes (buts). On se place alors dans le cadre des langages qui offrent la possibilité de *Atomic Test Unification* (essai d'unification atomique), mais dans une forme plus étendue.

Le pas élémentaire d'évolution du système est le choix d'une clause pour éliminer un but, en le remplaçant éventuellement par d'autres et en modifiant la base de contraintes. Le remplacement se fait de façon analogue a Prolog : après l'élimination du but B par moyen de la clause $A \texttt{ :- } Ask : \ Tell \mid Body$, on est sûr d'avoir $(A = B) \wedge Ask \wedge Tell \wedge Body$. La façon exacte dont ça se fait est donnée par les règles de transition.

3.2 Règles de Transition

On présente maintenant les règles de transition qui décrivent le comportement d'un programme. D'abord on donne quelques définitions dont on a besoin:

- **try(Γ, B, A:-Ask:Tell|Body)** (Γ=base des contraintes, B=but) **=**
  - **Si** $((A = B) \wedge Ask)$ est validée et $(A = B) \wedge Ask \wedge Tell$ est satisfaisable, **alors** $[\Gamma \wedge (A = B) \wedge Ask \wedge Tell, succeed]$
  - **Si** $((A = B) \wedge Ask)$ est validée et $\Gamma \wedge (A = B) \wedge Ask \wedge Tell$ est insatisfaisable,
    ou $((A = B) \wedge Ask)$ est insatisfaisable,
    alors $[\Gamma, fail]$
  - **Sinon** $[\Gamma, suspend]$
- **fst([a, b]) = a**
- **snd([a, b]) = b**

Les règles de transition (on omet ici la règle de transition ALPS, présentée dans la section correspondante) opèrent sur l'état du système réactif, représenté par le couple [Z, Γ] ou Z est la conjonction des buts (considérée comme ensemble) et Γ est la base de contraintes.

- **Succès(** $[\emptyset, \Gamma]$ **)** $\rightarrow$ le système s'arrête.
- **Echec(** $[Z, \Gamma]$ **)** $\rightarrow$
  - **Si** il existe but B tel que pour toute clause R
    snd(**try(** Γ, B, R)) = *fail*
    alors le système s'arrête avec échec.
- **Réduction(** $[Z, \Gamma]$ **)** $\rightarrow$
  - **Si** il existe but B et clause R telle que **snd(** **try(** Γ, B, R)) = *succeed*
    alors [Z-B+Body, **fst(** **try(** Γ, B, R))]

Notons que le système se trouve en impasse quand aucune règle de transition ne peut s'appliquer (*voir section ALPS qui suit*).

3.3 ALPS

On a vu que le système de transitions peut se bloquer d'une seule façon sans s'arrêter: quand pour tous les buts, toutes les clauses posent des Ask trop forts pour la base de contraintes courante. Pour éviter dans la mesure du possible cette éventuallité, on va emprunter une idée à M. J. Maher [2]. La règle de sélection *ALPS* est formée de deux cas. L'un c'est celui qu'on a vu, partagé par la plupart des langages logiques concurrents : que certaines variables doivent en quelque sorte être suffisamment "définies" pour déclencher une clause. L'autre est décrit par la suite.

Quand un but suspend une seule clause, A :- Ask : Tell | Body, de toute façon c'est la seule alternative qui lui reste. On peut alors, sous certaines conditions, *forcer* la prise en compte de cette clause en la traitant comme si elle était A' :- true : A'=A, Ask, Tell | Body (où A' est A avec comme arguments rien que des variables). Cette transformation effectivement est un "ultimatum", car un Tell soit échoue soit réussit mais n'est jamais suspendu. On fait de cette manière un choix forcé.

Si le nouveau Tell échoue, l'ancien Ask (But=A, Ask) allait échouer aussi. On détecte alors un échec qui serait masqué par la suspension. Un Tell qui réussit, par contre, produit des nouvelles contraintes qui n'ont peut-être pas toujours de sens opérationnel. En particulier, produire une valeur d'*entrée* n'a pas de sens du tout. Un autre problème est posé par la présence des instructions arithmétiques *(voir section suivante)*. Le passage en Tell d'une contrainte arithmétique exigerait une approche du genre CLP($\mathcal{R}$).

Un exemple tiré de [2] est le suivant:

```
append([], Y, Y).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
```

Ce petit programme, étant donné le but `append([], [a, b], A)` se trouve, sous la règle de sélection normale, en impasse. En fait, le programme correct serait:

```
append([], Y, Z) :- true : Y=Z | true.
append([A|X], Y, Z) :- true : Z=[A|W] | append(X, Y, W).
```

en indiquant bien les variables d'entrée et de sortie. Pourtant, sous la règle de sélection *ALPS*, la première clause qui est la seule qui reste (le test `[]=[A|X]` échouant) est choisie et produit le résultat A=[a, b]. Ce n'est pas, quand même, une facilité purement syntaxique, car le but `append([a, b], Y, [a, b, c])` réussit aussi et produit Y=[c] au lieu d'être suspendu. La règle ALPS se formule de la façon suivante:

ALPS($[Z, \Gamma]$ **)** $\rightarrow$

- **Si** il existe but B pour lequel il existe une seule clause R = A :- Ask : Tell | Body telle que **snd(** **try(** Γ, B, R)) = *suspend* et pour toute autre clause R', **snd(** **try(** Γ, B, R')) = *fail*
  alors
  if snd(**try(** Γ, B, A':-true:(A'=A),Ask,Tell|Body)) = *succeed*
  alors [Z-B+Body, **fst(** **try(** Γ, B, A':-true:(A'=A),Ask,Tell|Body))]
  sinon le système s'arrête avec échec.

3.4 Arithmétique

Pour des raisons pratiques, on augmente le langage avec des prédicats et des instructions arithmétiques. Les termes arithmétiques font partie du langage *(voir Syntaxe)* et les prédicats `==`, `=/=`, `<`, `<=`, `>` et `>=` sont acceptés dans la partie *Ask*. L'atome spécial *X* `is` *Exp* où *X* est une variable qui doit être non instanciée au moment de l'appel et *Exp* une expression arithmétique, est reconnu dans *Body*. Un appel à *X* `is` *E* est suspendu tant que l'expression n'est pas totalement instanciée, et échoue si jamais *X* est instancié ailleurs ou *E* contient des constantes non-numériques. La règle ALPS n'est pas applicable dans le cas de contraintes arithmétiques dans Ask. Exemple d'arithmétique:

```
fibo(N, L) :- fibo0 (N, 1, 1, L).

fibo0(0, _, _, L) :- true : L=[].
fibo0(N, A, B, L) :-
        N>0 :
        L=[A | L1] |
        C is A+B, M is N-1, fibo0(M, B, C, L1).
```

L'appel `fibo(`*N, L*`)` produit la liste *L* des *N* premiers nombres de Fibonacci.

3.5 Entrées / Sorties

Les entrées / sorties dans notre langage s'intègrent uniformément dans le même cadre logique que tout le reste. Une variable d'entrée qui n'a pas encore reçu une valeur, suspend, tandis qu'une variable de sortie produit des résultats au fur et à mesure qu'ils sont générés. L'association d'une variable à un flot (*stream*) d'entrée où de sortie, se fait dans la requête, avec les commandes `instream(`S, V`)` et `outstream(`S, V`)`. Une variable liée à un flot de sortie n'a pas de statut spécial. Une variable liée à un flot d'entrée jouit, par contre, d'une certaine protection: on restreint l'application de ALPS pour qu'elle n'instancie pas un flot d'entrée (mais aucune protection n'est imposée contre une instanciation explicite voulue par l'utilisateur). Exemple de requête : `instream(stdin, Xs), squares(Xs, Ys), outstream(stdout, Ys).`

4 La Syntaxe

Conventions Par *espace blanc* on entend toute suite des caractères *espacement*, *tabulation* et *fin de ligne* (même si ce dernier peut être lui-même une suite de caractères). Un *caractère alphanumérique* est une lettre (sans distinction entre minuscules et majuscules), un chiffre, où le caractère '_' (souligné).

4.1 Les Modules

Les *modules* définissent les espaces de noms dans le langage. La qualification (*voir plus bas pour sa syntaxe*) est la désignation d'un certain nom dans un certain module. Il y a un module qui s'appelle **principal**, vers lequel on peut *importer* des noms d'un module pour qu'ils soient accessibles partout **sans qualification**. Le nom sous lequel on importe un nom peut être différent, on parle dans ce cas-là d'un *alias*. Un nom importé qui existe déjà dans le module principal ne peut pas être *réimporté* sous le même nom. Une telle tentative provoque une erreur. La justification logique de la séparation des espaces de noms, est que tout nom en conflit avec un nom du module principal est *renommé* dans le module, et grâce à l'utilisation du nom qualifié / alias on fait référence au nouveau nom. A l'intérieur d'un module, les noms des prédicats *non qualifiés* font référence aux prédicats définis dans le même module ou présents dans le module principal, une définition locale ayant la priorité. Pareil pour les constantes et foncteurs utilisés dans les clauses. En tant que constructeurs des données, ils peuvent être importés dans le module principal sous un alias, pour faire utiliser les mêmes structures de données par différents modules. Si, par contre, ils ne sont pas renommés, ils sont importés **automatiquement** sous le même nom. Ceci est fait pour rendre le langage plus facile.

Les noms des modules doivent être des *constantes non qualifiées* distinctes *(voir section)*.

Les modules correspondent alors à des sous-programmes et le module principal au programme (`main` en C, session de Prolog).

Un module contient une collection de clauses *(voir section)*, éventuellement accompagnées par une définition
`module(`*M*`).`
où *M* est un nom de module. L'ordre n'est pas significatif.

Le module principal (interactif ou lu à partir d'un fichier) contient une série de
`import(`*F*`=`*M*`,`*id1*`=`*nouvid1*`,`*id2*`=`*nouvid2*`, ...).`
où *F* est un nom de fichier, *M* est un nom de module, *id*x est un identificateur du module et *nouvid*x son alias. Si un "= *nouvid*x" manque, le nom est importé tel quel. Si le "= *M*" manque, le nom du module est celui déclaré dans le module ou, par défaut, le nom du fichier.

Exemple: On a le fichier `fich1` qui a le contenu suivant:

```
module(toto).

lenght (L, N) :- length0( L, N, 0).

length0 ([], N, I) :- N=I.
length0 ([_|L], N, I) :- J is I+1, length(L, N, J).
```

Après l'exécution de la commande:

```
import(fich1).
```

les clauses retenues en mémoire sont:

```
toto.length( L, N) :-
 toto.length0( L, N, 0).
toto.length0( [], N, I) :-
 true :
 N=I |
 stop.
toto.length0( [_|L], N, I) :-
 J is (I+1), toto.length0( L, N, J).
```

Après exécution de la commande:

```
import(fich1,length).
```

les clauses retenues en mémoire sont:

```
length( L, N) :-
 toto.length0( L, N, 0).
toto.length0( [], N, I) :-
 true :
 N=I |
 stop.
toto.length0( [_|L], N, I) :-
 J is (I+1), toto.length0( L, N, J).
```

Après l'exécution de la commande:

```
import(fich1, length=longueur, []=fin_stream, cons=stream).
```

les clauses retenues en mémoire sont:

```
longueur( L, N) :-
 toto.length0( L, N, 0).
toto.length0( fin_stream, N, I) :-
```

```
 true :
 N=I |
 stop.
toto.length0( stream( _, L), N, I) :-
 J is (I+1), toto.length0( L, N, J).
```

4.2 Les Identificateurs

Les *identificateurs* servent pour nommer des entités. Ils se séparent en *constantes* et *variables*.

4.2.1 Les Constantes

Une *constante non qualifiée* est une suite de *caractères alphanumériques* qui ne commence pas par '_' ou une majuscule. `asterix`, `1482` et `3sur4` sont des constantes. Une constante qui ne comporte que des chiffres est une *constante numérique*.

Une *constante qualifiée* est une constante non qualifiée précédée par le nom d'un module et le caractère '.' **sans espace entre eux**. `module1.faire_qch`, `3.1et2` sont des constantes qualifiées. Une constante numérique ne peut pas être qualifiée.

4.2.2 Les Variables

Une variable de clause ou de but, est une suite de *caractères alphanumériques* qui commence par une majuscule ou un souligné. Le souligné est censé capitaliser le caractère qui suit, donc les deux formes *Xyyy* est *_xyyy* sont équivalentes (CAVEAT!). Ainsi `X`, `Variable` et `_alpha`, `_12` sont des noms des variables légaux. Le format *_v9999* ou *V9999* (où 9999 est un nombre quelconque) est réservé pour les variables internes du système et il est déconseillé. Une variable est toujours **non qualifiée**, car sa **portée** est la clause ou le but mais pas le module: elle est censée être quantifiée universellement dans une clause et existentiellement dans un but. On ne différencie pas syntaxiquement entre ces deux utilisations, le contexte étant suffisant pour les distinguer. C'est l'habitude, d'ailleurs, dans le monde **Prolog**.

A ces variables locales dans une clause ou dans un but, on ajoute des variables *locales* dans *une* équation ou *une* diséquation. Elles sont des suites de caractères alphanumériques qui commencent par une majuscule ou un chiffre, précédées par le caractère '?'. `?Z`, `?A_variable` et `?2` sont des variables locales. Elles sont censées être quantifiées universellement dans des diséquations et existentiellement dans les équations. Le contexte suffit pour les distinguer et, en plus, cela rappelle qu'une diséquation est la négation d'une équation. Ainsi, $\exists x\ y = f(x, z)$ sera écrit : `Y = f( ?X, Z)` et $\exists x\ y \neq f(x, z)$ sera écrit `Y /= f(?X, Z)` $\equiv$ `not Y = f(?X, Z)` (voir [5]).

La variable *?* s'appelle *variable anonyme* et chaque occurrence de cette variable doit être vue comme désignant une nouvelle variable, utilisée nulle part ailleurs. La notation _ est supportée aussi, pour raisons de compatibilité avec **Prolog**. En fait, la variable anonyme **Prolog** est vraiment une variable locale existentiellement quantifiée.

4.3 Les Termes

Les termes sont utilisés de diverses façons : pour désigner des valeurs (de l'algèbre de termes), des atomes logiques et tout objet structuré utilisé dans les commandes du système. Un terme peut être un *terme simple*, un *terme composé* où un *terme incomplet*.

- Si N est un terme, *(N)* l'est aussi. De l'espace blanc peut séparer les éléments du terme.

- Une variable ou constante est un *terme simple*.

- Si f est un terme simple, et X, Y, ... Z sont des termes, alors $f(X, Y, \ldots Z)$ est un terme *composé*. Tous les éléments peuvent être séparés par de l'espace blanc. Notons que le *foncteur* d'un terme composé peut être une variable *(voir section termes incomplets)*.

  Une catégorie de termes composés, ce sont les *expressions arithmétiques*. Ces termes ont une deuxième écriture infixe, celle connue en mathématiques. Par exemple `_x + 3` et `2+4*6`. Les règles de précédence sont celles attendues.

- Une *liste* est une sorte particulière de terme composé, un terme à deux arguments à foncteur `cons`. C'est la structure principale d'aggregation et en tant que telle elle a une syntaxe particulière, en plus de la syntaxe préfixée normale.

  La *liste vide* `[]` est une liste. `[`$M|N$ `]` est une autre façon d'écrire `cons(`M`,`N`)`.

  Si M, N, ... L sont des termes (il doit y avoir au moins un) et K est un terme, [M, N, ... L | K] est une autre façon d'écrire `cons(`M`, cons(`N`,` ...`cons(`L`,`K`)` ...`))`. Le terme K est appelé *la fin de la liste*. Une liste peut s'écrire de plusieurs façons : `[a, b, c, d | e] = [a, b, c | [d | e]] = [a, b | [c, d | e]] = [a, b | [c | [d | e]]]` etc. si on choisit différentes fins de liste. La *forme normale* d'une liste est celle dont la fin de liste n'est pas une liste non vide. Cette forme est utilisée en sortie du système, bien qu'en entrée on puisse donner une liste de toute manière syntaxiquement correcte. La liste vide est utilisé pour marquer la fin d'une liste habituelle. Dans ce cas, la liste est marquée $[X, Y, \ldots]$ = $[X, Y, \ldots | []]$. Un autre cas intéressant est quand la fin d'une liste est une variable.

- Si f est un terme simple, et L est une variable ou une liste, alors f L est un terme *(voir section termes incomplets)*. Les deux termes peuvent être séparés par de l'espace blanc.

4.3.1 Termes Incomplets

On a vu qu'un terme dans notre langage ne correspond pas tout-à-fait à un terme de l'algèbre des termes. La raison est qu'on veut pouvoir raisonner sur la forme des termes de façon impossible avec les termes usuels en logique du premier ordre. A ce dessein, on a introduit deux constructions: `_f( x, y, ...z)` et `_f L` (ou `f L`) où `L` est une liste ou une variable désignant une liste. La première nous permet de laisser non instancié le foncteur d'un terme composé, ce qui n'est pas trop illogique. Il faut quand même s'assurer que la **valeur** du foncteur est une constante, et non pas un autre terme composé, sinon le terme produit n'a ni signification logique, ni signification opérationnelle. A part cela, on utilise le terme comme attendu : `_f( _x) = a( _y)` donne `_f = a` et `_x = _y`, etc. La deuxième construction, qui est le *terme incomplet*, est introduite pour éviter les commandes extra-logiques de Prolog **=..** (univ), **arg/3** et **functor/3**. En effet, on peut se servir de l'unification avec `F X` pour décomposer un terme; `X = F W` donne le même résultat que `X =.. [F | W]`. Cas limite: `F []` correspond à une constante. Là encore, la valeur du foncteur doit être une constante et la valeur de L doit être une liste. La différence avec les prédicats extra-logiques de Prolog est que l'unification avec un terme incomplet est défini directement dans le système de contraintes. Exemples d'utilisation :

- `X = ?[]` contraint X d'être une constante.
- `X = F[?, ?, ? | ?]` contraint X à être un terme composé de foncteur F et avec au moins trois arguments.
- En utilisant un terme incomplet, on peut dans certains cas remplacer le test extra-logique `var(X)` par une contrainte qui n'est pas définie de façon opérationnelle.

Assurer les limitations imposées par les termes incomplets (valeur de f une constante, valeur de L une liste) est au même niveau qu'assurer les limitations imposées par l'arithmétique *ad hoc* sans utiliser un système de contraintes du genre **CLP(*R*)**. Donc, on ne pense pas que ça introduise des nouveaux problèmes. On pense même que l'implémentation doit se faire de la même façon, c'est-à-dire tester les contraintes au moment de l'utilisation. De la même manière qu'avoir en suspension un `_x is _y+1` ne veut pas dire qu'on espionne _y pour assurer en tout moment que c'est un nombre.

4.4 Clauses

Une clause est construite par les quatre parties *Head*, *Ask*, *Tell* et *Body* séparées de la façon suivante:

$$Head \texttt{ :- } Ask \texttt{ : } Tell \mid Body.$$

- *Head* est un terme composé (un *atome*) à foncteur instancié différent de $=, \neq$.
- *Ask* est une suite de contraintes (et de prédicats arithmétiques, voir section 3.4) séparées par des virgules. Si elle est vide, elle consiste au seul mot `true`.
- *Tell* est une suite de contraintes. Si elle est vide, elle consiste au seul mot `true`.
- *Body* est une suite d'*atomes*. Si elle est vide, elle consiste au seul mot `true` ou `stop`.

Certaines conventions facilitent l'écriture des clauses.

- Quand *Ask* et Tell sont vides, on peut les omettre, ainsi que les signes ":" et "|".
- Quand *Tell* est vide, on peut l'omettre, ainsi que le signe ":".
- Quand *Body* est vide et un des *Ask*, *Tell* n'est pas vide, on peut omettre le séparateur et `true` et y mettre seulement le point.
- Quand *Ask*, *Tell* et *Body* sont tous vides, on peut ometre tout ce qui suit *Head* sauf le point.
- On peut écrire des contraintes du *Tell* dans *Body* pour avoir des clauses qui ressemblent à **FCP(|)** et autres langages. Ceci n'est qu'une convention syntaxique. La clause retenue en machine a toujours les quatre parties qu'on vient de décrire.

5 Exemples de Programmation et de Techniques

5.1 Exemples Communs avec autres Langages

Le langage exposé jusqu'ici est une généralisation d'autres langages logiques concurrents. En tant que tel, il hérite les techniques de programmation des ces langages-là. On base cette présentation sur [4], un "survey" des langages logiques concurrents, où le lecteur pourra trouver plus de détails.

5.1.1 La Variable Logique

La variable logique et sa puissance est connue en Programmation Logique, mais son utilisation dans les langages logiques lui donne encore plus de richesse d'expression. En effet, la variable logique correspond à un **canal de communication** dont la flexibilité lui permet de remplacer plusieurs protocoles de la programmation concurrente, comme:

- discours tête-à-tête (un locuteur, un auditeur)
- diffusion (un locuteur, plusieurs auditeurs)
- conversation (deux interlocuteurs)
- serveur (plusieurs locuteurs, un auditeur)
- tableau noir (plusieurs interlocuteurs)

Seuls les deux premiers protocoles peuvent être implantés dans tous les langages concurrents.

La variable logique sert pour implanter des *listes latentes* — *flots* dont les queues sont instanciées peu-à-peu.

5.1.2 Gardes à Ask uniquement

Une grande partie des langages "plats" (sans prédicats définis par l'utilisateur dans la garde) correspond à un sous-ensemble de notre langage, qui n'a pas de *Tell* (*Tell*=true). On commence alors par la présentation des méthodes de programmation dans ces langages, qui sont plus connues et répandues.

Listes Latentes — Flots Dans beaucoup de cas, un flot est unidirectionnel en ce qui concerne un processus c'est-à-dire le processus soit *lit* le flot, soit il *écrit* dessus. Si un processus lit un flot et produit un autre, il s'appelle un "transducteur". Comme exemples on donne un processus qui produit un flot de nombres consécutifs, et un processus qui double les nombres d'un flot en produisant un deuxième flot. (La syntaxe est celle de **FCP(|)** et correspond à la syntaxe "sucrée" de notre langage).

```
ints(From, To, Ns) :- From > To | Ns=[].
ints(From, To, Ns) :- From <= To |
        Ns=[From | Ns2], From2 is From+1, ints(From2, To, Ns2).

squares([], Out) :- Out=[].
squares([A | In], Out) :-
        B is A*A, Out=[B | Out2], squares(In, Out2).
```

Fusion Il arrive très souvent qu'on veut fusionner plusieurs flots à un seul. Ceci doit se faire de façon non déterministe, parce qu'on ne peut pas supposer un ordre fixe d'instanciation des flot. En plus, la fusion ne doit privilégier aucun flot par rapport aux autres. Les langages logiques concurrents peuvent décrire très facilement une telle manipulation. On présente comme exemple un "merger" de trois flots (on peut l'étendre facilement en ajoutant des nouveaux arguments et des clauses de choix correspondantes):

```
merge3([X|In1], In2, In3, Out) :-
        Out=[X|Out2], merge3(In1, In2, In3, Out2).
merge3(In1, [X|In2], In3, Out) :-
        Out=[X|Out2], merge3(In1, In2, In3, Out2).
merge3(In1, In2, [X|In3], Out) :-
        Out=[X|Out2], merge3(In1, In2, In3, Out2).
merge3([], In2, In3, Out) :-
        merge2(In2, In3, Out).
merge3(In1, [], In3, Out) :-
        merge2(In1, In3, Out).
merge3(In1, In2, [], Out) :-
        merge2(In1, In2, Out).
```

Remarque: quand on arrive à `merge1`, `merge1(In, Out)` correspond, bien sûr, à `In=Out`.

Ceci est un bon moment pour parler un peu de l'**équité** (*fairness*). Disons simplement qu'une exécution d'un langage concurrent est *équitable* quand aucun choix non déterministe n'est ni privilégié ni ignoré. Bien que simple à énoncer, c'est un prérequis très fort, difficile à formaliser et permet d'exprimer des calculs non exprimables dans un langage non équitable sans trop d'effort. Le langage qu'on a décrit est équitable.

Réseaux dynamiques de processus Un exemple qui montre la capacité de repartir un calcul en autant de processus que nécessaire, est la multiplication de deux matrices, représentées par des listes de listes.

```
% Produit scalaire de deux vecteurs
ip(Xs, Ys, S) :-
```

```
        ip1(Xs, Ys, 0, S).

ip1([], [], P, S) :- P=S.
ip1([X|Xs], [Y|Ys], P, S) :-
        P1 is P+X*Y,
        ip1(Xs, Ys, P1, S).

% Produit d'une matrice et d'un vecteur
vm(_, [], Zv) :- Zv=[].
vm(Xv, [Yv|Ym], Zv) :-
        Zv=[Z|Zv1],
        ip(Xv, Yv, Z),
        vm(Xv, Ym, Zv1).

% Produit d'une matrice et d'une matrice transpos\'ee
mm([], _, Zm) :- Zm=[].
mm([Xv|Xm], Ym, Zm) :-
        Zm=[Zv|Zm1],
        vm(Xv, Ym, Zv),
        mm(Xm, Ym, Zm1).
```

Messages Incomplets Les messages envoyés sur un flot peuvent ne pas être totalement instanciés. De cette façon, on peut effectivement *multiplier* les liaisons entre processus et *reconfigurer dynamiquement* le réseau de communication (un flot peut être communiqué à d'autres processus).

Un exemple de messages incomplets qui réalisent une liaison producteur–consommateur où c'est le consommateur qui demande des valeurs (eager consumer – lazy producer) est le suivant (un traitement plus élaboré de cette exemple est donné par la suite).

```
consume(0, L) :- L=[].
consume(N, L) :- N>0 |
        L=[X | Ls], M is N-1, consumer(M, Ls).

producer([], _).
producer([X|L], N) :- X=N, M is N+1, producer(L, M).
```

Ici, le producteur attend l'instanciation de son premier argument à une liste. Si c'est la liste vide, il arrête, sinon il unifie le premier élément avec la prochaine valeur à produire, et il itère sur le reste de la liste. Le producteur est piloté par le consommateur, qui envoie selon son gré des messages incomplets à instancier. Pour étendre ce protocole pour permettre au producteur d'arrêter l'interaction, il faut la fonctionalité de Tell (voir plus bas).

L'exemple suivant est un *moniteur* (données encapsulées avec un agent qui les gère) :

```
queue(In) :- queue0(In, Q-Q).

queue0([dequeue(X)|In], H-T) :- H=[X|H2], queue0(In, H2-T).
queue0([enqueue(X)|In], H-T) :- T=[X|T2], queue0(In, H-T2).
queue0([], _).
```

Le processus accepte un flot de messages `dequeue(`*Variable* `)` et `enqueue(`*Elément*`)` qu'il exécute en tant que directives qui gèrent une file (représentée par une d-liste). Le message `dequeue` contient une variable qui est unifiée avec l'élément à lire, effectivement lui donnant une valeur. Notons, en passant, que la communication dans les langages logiques concurrents est anonyme, et qu'on peut interpréter l'instanciation d'un message incomplet comme une sorte de **RPC** (Remote Procedure Call).

Exclusion Mutuelle Le processus `mutex` est un mécanisme qui donne la main successivement à des processus en exclusion mutuelle. L'alimentation de `mutex` avec des requêtes, dépend d'une fusion (voir section 5.1.2) dans un seul flot qu'il reçoit en entrée. Notez que la première clause de `mutex0` attend à la fois que le processus qui a la main, la lâche, et qu'une requête la demande. Le synchronisation se fait grâce à une variable `Done` qui est instanciée à `done` pour lâcher la main.

```
mutex(In) :- mutex0(In, done).

mutex0([lock(Reply)|In], done) :-
        Reply=granted(Done), mutex0(In, Done).
mutex0([], _).
```

Court-Circuit Un problème evident quand on lance plusieurs tâches en parallèle, c'est de détecter leur fin. Les langages logiques concurrents se prêtent à une solution qui leur est propre, et qui se sert des variables logiques pour surveiller des réseaux dynamiques distribués avec pas plus d'effort que pour les réseaux statiques. Ce protocole s'appelle **court circuit** à cause de sa similarité avec le circuit électrique. Un circuit ne se ferme que si tous les interrupteurs se sont fermés. On traite la création dynamique des tâches, tout simplement en leur attribuant des nouveaux interrupteurs qui leur appartiennent. On peut installer très facilement ce protocole, à condition de faire les modifications suivantes:

```
p(...)         ⇒ p(..., L–R) :- L=R.

p(...) :-      ⇒ p(..., L–R) :-
  p'(...)          p'(..., L–R).

p(...) :-      ⇒ p(..., L–R) :-
  p1(...),         p1(..., L–X1),
  p2(...),         p2(..., X1–X2),
    ⋮                ⋮
  pN(...).         pN(..., X(N-1)–R).
```

L'interrupteur est juste un couple de variables `Gauche-Droite` liées pour des raisons de facilité par un moins. La condition `Gauche = Droite` correspond à la fermeture de l'interrupteur. On en crée des nouveaux en les intercalant dans un existant, comme le montre le schéma précédent. Un processus qui veut détecter la fermeture du circuit, tient les deux bouts et teste leur égalité. Ce protocole peut s'étendre pour traiter des *phases* de synchronisation.

5.1.3 Gardes à Ask & Tell

Les langages à Ask et Tell (voir [4], [3]) offrent la possibilité de tester le succès des unifications. Non seulement ça permet d'écrire des méta-interpréteurs, mais ça permet la sélection d'une clause selon ce qui lui est possible de faire, et non seulement ce qui est fait.

Exclusion Mutuelle Un cas particulier de l'exclusion mutuelle, l'exclusion mutuelle à tour unique (single-round), a une solution très simple dans le cadre de ces langages. Il suffit de munir chaque processus avec un identificateur unique I_n et avec une variable ME. Quand un processus veut le lock, il essaie d'unifier son identificateur avec ME. S'il échoue, il agit convenablement.

```
p(ME, I, ...) :- true : ME=I |
                              ...lock donné ...
p(ME, I, ...) :- ME /= I |
                              ...lock refusé ...
```

Le Diner des Philosophes Le diner des philosophes est du à E. W. Dijkstra et les problèmes qu'il pose se trouvent à la base de tout système d'allocation des ressources distribué. Il y a n philosophes qui sont assis autour d'une table, avec une fourchette entre deux philosophes. Les philosophes aiment réflechir

et manger, mais il leur faut deux fourchettes pour manger et on doit écrire un système qui garantit que tous les philosophes arrivent à manger (sans s'empêcher les un les autres et sans qu'aucun philosophe ne crève de faim). Le programme suivant définit le comportement d'un philosophe:

```
phil(Id, [eating(LeftId, done)|Left], Right) :-  % attendre L
        phil(Id, Left, Right).
phil(Id, Left, [eating(RightId, done)|Right]) :- % attendre R
        phil(Id, Left, Right).
phil(Id, Left, Right) :-           % fourchettes
        true : Left  =[eating(Id, Done)|Left2],
               Right =[eating(Id, Done)|Right2] |

             ...manger, unifier Done=done, reflechir,
               puis lancer phil(Id, Left2, Right2).
```

Pour commencer un diner à n philosophes, il suffit de poser la requête: `phil(1, Fork1, Fork2)`, `phil(2, Fork2, Fork3)`, ..., `phil(`n`, Fork`n`, Fork1)`. Notons que le comportement d'un philosophe est fortement séquentiel; cette séquencialité exige une approche du genre "Court-circuit".

Duplex Stream Le programme suivant décrit un protocole de communication entre deux processus. L'un place des messages `write(Terme)` dans un flot et l'autre les lit. Le lecteur avertit le producteur quand il a lu tous les messages, en plaçant le message `read` dans le flot. Ce protocole est beaucoup plus souple qu'un protocole correspondant dans les langages sans Tell. Il permet une modification de la taille du "buffer" dynamique et facile.

```
produce(M, Ms, Ms2, Status) :- true :
        Ms=[write(M)|Ms2] | Status=old.
produce(M, [read|Ms], Ms2, Status) :-
        Ms=[write(M)|Ms2], Status=new.

consume([M|Ms], Ms2, Rs) :-
        consume0([M|Ms], Ms2, Rs).
consume0(Ms, Ms2, Rs) :-
        true : Ms=[read|Ms2], Rs=[] | stop.
consume0([write(M)|Ms], Ms2, Rs) :-
        true : Rs=[M|Rs2] | consume0(Ms, Ms2, Rs2).
```

Evaluation Paresseuse La procédure `consume` décrite dessous est un consommateur actif (*eager consumer*). La procedure `produce` est un producteur paresseux (*lazy producer*). On lie le premier argument du consommateur au premier argument du producteur avec une requête de la forme ..., `consume(X,`

`Out, ...), produce(X, ...), ....` Le producteur peut essayer d'arrêter la consommateur mais s'il échoue, il produit la valeur spéciale **overflow**.

```
consume( [], Out, ...) :-
  ...interruption d'entrée ... ...agir ...
consume( In, Out, ...) :-
  ...veut s'arrêter ... :
  In=[], Out=[], ... |
  ...arrêter ...
consume( In, Out, ...) :-
  ...veut continuer ... :
  In=[X|I], Out=[X|O], ... |
  ...continuer ... consume(I, O, ...).

produce( [], ...) :-
  ...arrêté ...
produce( [X|Xs], ...) :-
  ...veut continuer ... :
  X=valeur, ... |
  ...continuer ... produce(Xs, ...).
produce( Xs, ...) :-
  ...veut s'arrêter ...l'essaie ... :
  Xs=[], ... |
  ...a réussit d'interrompre ...
produce( [X|Xs], ...) :-
  ...pourtant il veut s'arrêter !... :
  X=overflow, ... |
  ...continuer sous force ... produce(Xs, ...).
```

Méta-interpréteurs La démonstration la plus évidente de la puissance de Tell, est le méta-interpréteur écrit dans le langage lui-même. Avec un tel méta-interpréteur (qui demande juste une transformation du programme) on peut exécuter un programme qui, au lieu d'échouer, indique d'une façon l'échec et on agit convenablement. Le point central d'un tel méta-interpréteur, c'est la méta-description de l'égalité:

```
tell_eq(X, Y, Outcome) :-
 true :
 X=Y, Outcome=ok |
 stop.
tell_eq(X, Y, Outcome) :-
 true :
 X/=Y, Outcome=fail |
 stop.
```

5.2 Exemples propres à notre langage

Notre langage offre en plus des langages à Ask & Tell, la possibilité d'utiliser des contraintes négatives. Prenons l'exemple du programme suivant. La procedure `accept` est un consommateur actif (*eager consumer*) piloté par un flot de directives. Une directive `any` accepte tout élément et une directive `not(`T`)` rejette l'élément T. La procedure `select_discard` est un producteur paresseux (*lazy producer*) qui fusionne de façon non-déterministe un nombre de flots d'entrée (ici, deux), en avançant **tous** les flots.

```
select_discard(_, _, []).
select_discard(In1, In2, [X|Out]) :-
 true :
 In1=[X|S1], In2=[_|S2] |
 select_discard(S1, S2, S).
select_discard(In1, In2, [X|Out]) :-
 true :
 In1=[_|S1], In2=[X|S2] |
 select_discard(S1, S2, S).

accept([], S) :-
 true :
 S=[] |
 stop.
accept([any|C], S) :-
 true :
 S=[_|So] |
 accept(C, So).
accept([not(T)|C], S) :-
 true :
 S=[X|So], X/=T |
 accept(C, So).
```

Si par exemple les flots d'entrée contiennent des commandes `dir(`Id, D`)`, un flot de directives `[any, not(`*Id1*`, ?), not(`*Id2*`, ?) ...]` refuse certains Id_x. On peut effectivement voir `select_discard` comme une sorte d'exclusion mutuelle. La procedure `accept` peut être étendue facilement pour traiter des directives plus complexes.

La requête `accept( [not(a), not(b)], Stream), select( [a, a, a|A], [b, b, b|B], Stream]).` dans notre langage a donné:

```
LOOP
Goals 2 = accept([not(b)], V122), select([a, a, a | A], [b, b, b | B],
[V120 | V122])
LOOP
```

```
Goals 2 = select([a, a, a | A], [b, b, b | B], [V120, V125 | V127]),
accept([], V127)
LOOP
Goals 2 = select([a, a | A], [b, b | B], [V125 | V127]), accept([], V127)
LOOP
Goals 2 = select([a | A], [b | B], V127), accept([], V127)
LOOP
Goals 1 = accept([], V127)
LOOP
Goals 0
 Stream = [b, a]

6 reductions
1 suspensions
3 reactivations
ok
```

Une contrainte négative peut être imposée sur toute une liste:

```
not_in_list(_, []).
not_in_list(X, [Y|L]) :-
 true :
 X/=Y |
 not_in_list(X, L).
```

ou un flot:

```
% to put between eager consumer and lazy producer
not_in_stream(_, [], S) :-
 true :
 S=[] |
 stop.
not_in_stream(X, [Y|L], S) :-
 true :
 X/=Y, S=[Y|So] |
 not_in_stream(X, L, So).
```

L'atomicité du Tell nous assure que le flot en sortie n'est alimenté qu'avec les messages incomplets contraints.

6 Implémentation

Le système de transitions qu'on a décrit a l'avantage d'être clair et logiquement simple. Il est évident, quand même, qu'une implémentation qui le traduirait directement en programme serait très inefficace. La solution adoptée généralement pour les mono-processeurs (et celle utilisée ici), est de mettre tous les buts dans une file d'attente, et de les traiter un par un, les buts suspendus étant mis dans une autre liste, appelée *liste de suspension*. La raison pour cela est de pouvoir leur communiquer facilement les modifications de la base de contraintes. Chaque ensemble de modifications (qui vient d'un Tell) est propagé et toutes ses conséquences sont déclenchées. Ainsi, le test des gardes n'est pas du tout indéterministe, car la diffusion des nouvelles contraintes se fait de façon algorithmique [3]. On assure tout de même l'indéterminisme parce que l'insertion des nouveaux buts dans la liste se fait à des endroits aléatoires. En plus, le choix entre plusieurs clauses qui réussissent se fait aléatoirement. Même avec ça, c'est sûr que le système de transitions qui résulte n'est pas strictement équivalent au système théorique, la raison étant que, à un instant donné, le choix entre les clauses qui réussissent se fait entre les clauses qui réussissent **à cet instant**, sans tenir compte des clauses qui **pourraient** réussir à un instant ultérieur. C'est-à-dire dès qu'on trouve au moins une clause validée on en choisit une et on la déclenche. Il paraît que cette considération est jugée mineure, par exemple elle n'apparaît pas dans [2]. On peut donner un exemple en utilisant la procédure `merge3` présentée plus haut. Si à un instant donné on traite le but `merge3(X,Y,Z,Out)` et les flots `X`, `Y` ont des éléments à lire tandis que `Z` n'en a pas, le choix sera, bien évidemment, entre ces deux flots. Si, par contre, le but est dans une position profonde et le flot `Z` a le temps d'avoir des éléments avant qu'on traite le but, le flot `Z` entre en considération aussi. Si tous les éléments de `Z` sont produits après que `X` et `Y` aient été fermés, ils sont tous mis à la fin. On voit, alors, que les transitions possibles sont conditionnées en partie par l'état du calcul. Cela est inévitable pour un langage pour le calcul parallèle.

La figure 1 montre quelle forme ont nos structures de données en mémoire. On associe à chaque variable qui suspend un but, des cellules appelées "sus rec" sur la figure (*suspension record*) qui pointent vers la clause et le but suspendus. Chaque fois qu'une contrainte vient d'être ajoutée, qui concerne la variable, on met à jour la clause suspendue. Si la clause réussit, on la déclenche et on retire le but de la liste de suspension, si elle échoue on la retire et si elle continue à suspendre, on la (re)met à jour pour des essais ultérieurs.

[3] On pourrait envisager d'autres stratégies, par exemple enregistrer les modifications et tester de nouveau les gardes concernés tous les n cycles. Cela ne change pas la validité de la remarque faite plus bas.

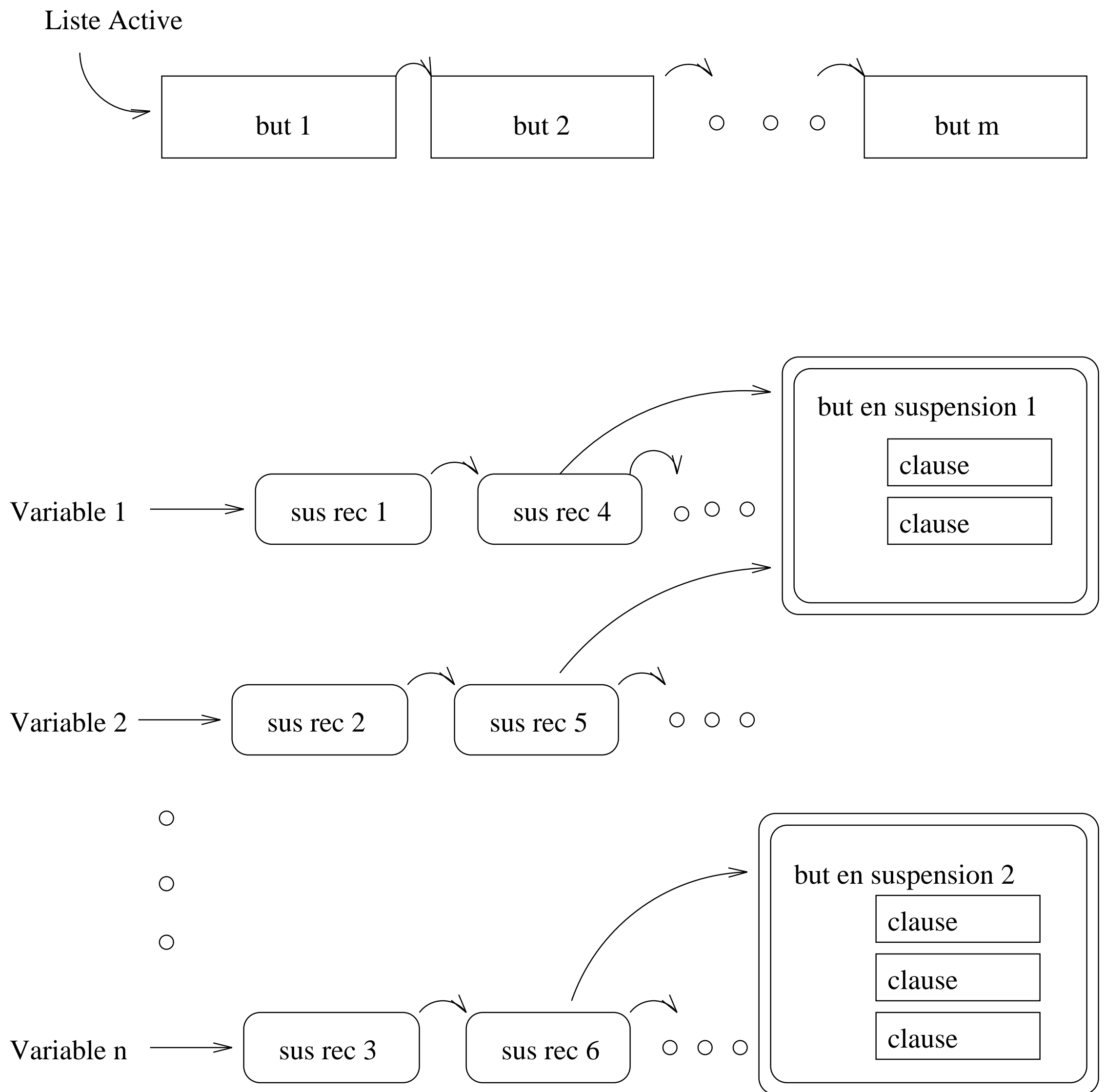

Figure 1: Liste de buts (Liste active), liste de suspension

6.1 Satisfaction des Contraintes

En décrivant le comportement opérationnel de notre langage, on s'est servi de formules logiques qui n'ont pas de signification opérationnelle évidente. Cette traduction en termes informatiques va être expliquée dans cette section.

Variables suspendues Le moyen principal d'évaluation de contraintes dans notre système est l'unification, étendue de la manière suivante: on distingue les variables auxquelles on peut donner librement une valeur, de celles auxquelles toute tentative d'affecter de valeur doit être suspendue. Typiquement, les variables du but qu'on cherche à effacer sont protégées. De telles variables, s'appellent "variables suspendues". L'unification qui est à la base des tests de garde, retourne une substitution faite de deux parties.

1. Une substitution qui affecte les variables suspendues.

2. Une substitution qui affecte les autres variables.

On écrira une telle substitution $\sigma = \{$ *partie suspendue* | *partie immédiate* $\}$. La propagation des substitutions qui sont suspendues nous permet une sorte d'évaluation partielle de la garde, et la détection des échecs inévitables. Exemple: Soit `proc([a|X], Y, Y, [X|X])` un but qu'on unifie avec la tête d'une clause `proc(V1, [V2|V3], V1, [?A|?A])`. Cette unification (qui fait partie de la garde) donne comme résultat : $\{$ `Y` $\mapsto$ `[V2|V3]` | `V2` $\mapsto$ `a`, `V3` $\mapsto$ `X`, `V1` $\mapsto$ `[a|X]` $\}$. On voit tout de suite que cette clause doit être suspendue. Pourtant, on continue à évaluer la partie Ask de la garde, dans le dessein de la simplifier et de détecter le plus tôt possible un échec.

Notez que le système équivalent de la substitution est toujours donné sous *forme normale* (voir *Les Systèmes de Contraintes*).

Compatibilité avec les Contraintes Le résultat d'une unification dépend des deux termes unifiés. Quand on teste une *égalité* par contre, l'unifiabilité ne suffit pas: on vérifie aussi la compatibilité avec les diséquations de la base de contraintes. (La compatibilité avec les équations est assurée par le fait qu'on propage toute substitution et, par conséquence, on n'a pas besoin de tester les équations).

La compatibilité avec les diséquations est testée par l'unification. On applique la substitution trouvée σ aux parties des diséquations $M \neq N$ une par une et on unifie $\sigma M = \sigma N$.

- Si l'unification réussit en produisant la substitution nulle (identité), on échoue.

- Si l'unification produit une substitution σ', la diséquation n'échoue pas et elle est simplifiée par supression de déségalités insatisfaisables.

- Si l'unification échoue, la diséquation est trivialement satisfaite.

Par exemple: soit la diséquation `(X,Y)/=(Z,W)`.

- Si on teste la substitution $\sigma = \{$ `X` $\mapsto$ `a`, `Y` $\mapsto$ `b`, `Z` $\mapsto$ `a`, `W` $\mapsto$ `b` $\}$, σ`(X,Y)=`σ`(Z,W)` donne la substitution identité $\{ \emptyset \}$ et la diséquation échoue.
- La substitution $\{$ `X` $\mapsto$ `a` $\}$ donne $\{$ `Z` $\mapsto$ `a`, `Y` $\mapsto$ `W` $\}$ ce qui correspond à la diséquation `(Z,Y)/=(a,W)`. La substitution $\{$ `X` $\mapsto$ `a`, `Z` $\mapsto$ `a` $\}$ donne $\{$ `Y` $\mapsto$ `W` $\}$ ce qui correspond à la diséquation `(Y)/=(W)`. Dans ces deux cas la déségalité n'échoue pas.
- La substitution $\{$ `X` $\mapsto$ `a`, `Z` $\mapsto$ `b` $\}$ échoue d'unifier les deux tuples, donc la diséquation est validée.

Notez que, quand on n'a pas besoin de distinguer une partie suspendue dans la substitution, on n'en met pas.

Ask – égalité Le test d'égalité dans le Ask $N = M$, correspond à unifier les deux termes M et N et puis:

- Si l'unification échoue ou produit une substitution incompatible avec les diséquations de la base de contraintes, le test (et la garde) échoue.
- Si l'unification réussit et la substitution n'a pas de partie suspendue, le test réussit.
- Si la substitution est suspendue, le test est suspendu aussi. Les équations sont remplacées par la partie suspendue de la substitution, transformée sous forme d'équations.

Le dernier cas ci-dessus correspond à une évaluation partielle de la clause. Par exemple, soit la clause `proc(V1, V2) :- V1=[V3|V4], V2=[V3|V5] | proc(V4, V5), do(V4, V3).` et le but `proc([a|X], [Y|Z])`. L'évaluation de la garde donne, si on propage chaque fois les substitutions:

1. `proc(V1, V2)=proc([a|X], [Y|Z])` $\Longrightarrow \{ \emptyset \mid$ `V1` $\mapsto$ `[a|X]`, `V2` $\mapsto$ `[Y|Z]` $\}$.
   Clause: [*Ask*=`[a|X]=[V3|V4]`, `[Y|Z]=[V3|V5]` *Tell*=$\emptyset$ *Body*=`proc(V4, V5)`, `do(V4, V3)`]
2. `[a|X]=[V3|V4]` $\Longrightarrow \{ \emptyset \mid$ `V3` $\mapsto$ `a`, `V4` $\mapsto$ `X` $\}$.
   Clause: [*Ask*=`[Y|Z]=[a|V5]` *Tell*=$\emptyset$ *Body*=`proc(X, V5)`, `do(X, a)`]
3. `[Y|Z]=[a|V5]` $\Longrightarrow \{$ `Y` $\mapsto$ `a` $\mid$ `V5` $\mapsto$ `Z` $\}$
   Clause: [*Ask*=$\emptyset$ *Tell*=$\emptyset$ *Body*=`proc(X, Z)`, `do(X, a)`]

La garde est suspendue et la clause suspendue est: [*Ask*=`Y=a` *Tell*=$\emptyset$ *Body*=`proc(X, Z),do(X, a)`].

L'égalité $but = tête$ est, en fait, le premier test de chaque garde.

Si le test suspend, on propage quand même la substitution pour évaluer partiellement la garde. Pour cette raison, la garde en mémoire est une liste où les ségalités se trouvent **après** les égalités (et tout autre test au milieu).

Ask – déségalité Le test de déségalité $N \neq M$ procède en testant l'égalité des deux termes, et puis:

- Si l'égalité échoue, la diséquation est satisfaite.
- Si l'unification produit la substitution nulle, la diséquation est invalidée.
- Si l'unification produit une autre substitution qui n'invalide pas la base de contraintes, la substitution donne une forme normale pour la nouvelle diséquation, partiellement évaluée.

Par exemple, `f(X,Y)/=f(X,Z)` est résolu en tant qu'égalité et produit la substitution { `Y` $\mapsto$ `Z` } qui correspond à la déségalité `(Y)/=(Z)`. On se sert des propriétés de l'unification pour trouver les formes normales des équations **et** des diséquations.

Tell Le Tell est en fait implémenté au-dessus de Ask. Les contraintes qui résultent sont considérées comme des contraintes à ajouter. Pour cette raison, les diséquations sont stockées après les équations (comme dans le Ask), afin de profiter le plus de la propagation des substitutions.

A la différence de Ask, un Tell ne peut pas contenir autre chose que des contraintes "pures" (pas de contraintes arithmétiques).

Réclamation d'Espace Pour une vraie implémentation réactive, il y a une autre considération dont il faut tenir compte: l'évolution des systèmes réactifs crée beaucoup de contraintes qui ne servent pas par la suite. Il faut prévoir un mécanisme qui détecte les contraintes "mortes" et en réclame l'espace.

7 Conclusions

Les langages à Ask et Tell intègrent toutes les notions de communication, de synchronisation et de répartition qui sont nécessaires au calcul concurrent. Ces langages se situent dans un cadre logique, à la fois de la programmation logique et du traitement logique des contraintes (à la Saraswat [3]). On peut espérer que cette situation permette le développement de méthodes de preuve et de transformation de programmes qui sont encore plus utiles au génie logiciel des systèmes concurrents qu'à celui des systèmes séquentiels. Notre travail a consisté en la conception, l'implémentation d'un langage à Ask et Tell opérant par contraintes sur les termes finis (égalité et deségalité), et l'étude de son expressivité. Ces contraintes ont déjà été utilisées dans des langages logiques non-concurrents, tandis que les langages à Ask et Tell ne traitent à notre connaissance que l'égalité des termes.

L'adjonction des deségalités dans la partie *Tell* a été faite sans trop de difficulté, et elle a produit un langage plus satisfaisant du point de vue théorique que les langages avec égalité seulement. Pourtant, cette nouvelle possibilité ne nous semble pas majeure. En effet, les contraintes d'égalité et de deségalité ne sont pas équivalentes de point de vue d'une application : comme partout en programmation, ce sont les valeurs des variables qui importent le plus. En programmation logique, l'ajout d'une égalité permet déjà d'interdire un ensemble de valeurs aux variables concernés ; ajouter des deségalités ne fait que réduire encore cet ensemble. L'intérêt des deségalités en partie Tell est de réduire cet ensemble de valeurs de façon moins contraignante, ce qui permet de prolonger le caractère non-déterministe de certains choix. Bien que d'un intérêt limité, cette possibilité est ainsi tout à fait dans l'esprit d'un langage concurrent.

L'adjonction de la réflexivité des termes répond à un besoin réel, comme le montre la littérature de la programmation logique. Elle a été faite de manière compatible avec le caractère logique du langage, pour éviter dans la mesure du possible l'emploi de prédicats non-logiques.

Le système de modules, finalement, est très prometteur à cause de sa simplicité et de sa souplesse. En fait, il peut être utilisé par **Prolog** ou n'importe quel autre langage similaire à espace de noms unique, en réécrivant simplement la procédure de consultation. Bien que moins rigide qu'un système de modules du genre **ML**, c'est assez puissant pour assurer la coopération et la protection des éléments logiciels.

Bibliographie

[1] Abelson et al. *Structures et interprétation des programmes informatiques.* Interéditions, 1989.

[2] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming — Proceedings of the Fourth Int'l Conference.* MIT Press, 1987.

[3] V. A. Saraswat. Semantic foundations of concurent constraint programming. In *Proceedings – Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Jan. 21-23, 1991.

[4] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing surveys*, 21(3), Septembre 1989.

[5] Donald A. Smith. Constraint operations for CLP($\mathcal{FT}$). In Koichi Furukawa, editor, *Logic Programming — Proceedings of the Eighth Int'l Conference.* MIT Press, 1991.