# Programmation Logique et Par Contraintes Avancée
# Cours 6 – Blocages de Propagateurs et Recherche Multidimensionelle

Ralf Treinen

Université Paris Cité
UFR Informatique
Institut de Recherche en Informatique Fondamentale


treinen@irif.fr

14 février 2024

# Types

- Dans la documentation de Oz, les noms utilisés pour des arguments de procédures indiquent leur *type* et leur *mode* attendus.
- Le type est en vérité le type attendu de la valeur (car Oz n'est pas statiquement typé), notament

| lettre | type | lettre | type |
|---|---|---|---|
| I | entier | B | booléen |
| R | record | P | procédure |
| X | n'importe | | |
| s final | liste | v final | vecteur |

- Exemple :

```
{List.length +Xs ?I}
```

# Modes

- Le mode peut être `?`, `+`, `*`, `$`, ou rien
- Les modes sont seulement documentation, ce n'est pas dans la syntaxe du langage Oz.
- Le mode indique le comportement de *synchronisation* de la procédure :
  - Le mode `?` indique qu'il s'agit d'un *résultat*
  - Le mode `+` indique que l'argument doit être suffisament déterminée, sinon la procédure suspend.
  - Le mode `*` indique que l'argument doit avoir un domaine fini (ca inclut le cas d'être une valeur entière)
  - Tous les arguments avec mode `$`, sauf éventuellement un, doivent avoir un domaine fini.

# Exemples (mod1.oz)

```
% Problème : pas de propagateur !
declare
proc {D L}
   X Y Z
in
   L=X#Y
   [X Y]:::1#9
   Z::2#4
   X mod Z = Y
   {FD.distribute naive [X Y Z]}
end

{Browse {SearchAll D}}
% donne simplement une paire de deux domaines
```

# Analyse : `mod1.oz`

- On a donné tous les domaines initiaux qu'il faut.
- On a mis le bon `FD.distribute`
- Mais évidemment la distribution n'a pas lieu. Pourquoi ?
- Le problème est l'appel à `mod`. Regardons ce qui dit la doc (Oz Base Environment, 4.2 Integers) :

```
{Int.'mod' +I1 +I2 ?I3}
```

- Donc l'appel à `mod` suspend car les deux premiers arguments sont des domaines et pas des entiers.
- Tentative de solution : utiliser `FD.modI`

# Exemples (mod2.oz)

```
% Toujours Blocage!

declare
proc {D L}
   X Y Z
in
   L=X#Y
   [X Y]:::1#9
   Z::2#4
   {FD.modI X Z Y}
   {FD.distribute naive [X Y Z]}
end

{Browse {SearchAll D}}
```

- Ca ne distribue toujours pas !
- Essayons d'instrumenter le code pour voir ou il bloque.

# Exemples (mod3.oz)

```
% Toujours Blocage!

declare
proc {D L}
   X Y Z
in
   L=X#Y
   {Show 'apres liaison L'}
   [X Y]:::1#9
   {Show 'apres domaine pour X et Y'}
   Z::2#4
   {Show 'apres domaine pour Z'}
   {FD.modI X Z Y}
   {Show 'apres lancement propagateur'}
   {FD.distribute naive [X Y Z]}
end

{Browse {SearchAll D}}
```

# Propagateurs qui peuvent suspendre

Voir la documentation Oz : *System Modules*, Section *5.11 Miscellaneous Propagators* :

- le propagateur pour + : `{FD.plus $D1 $D2 $D3}`
- mais le propagateur pour *modI* : `{FD.modI $D1 +I $D2}` suspend quand le diviseur n'est pas une valeur déterminée !
- Solutions possibles :
  - énumérer avant de tester avec div et mod
  - utiliser des propagateur pour la multiplication pour exprimer la division avec reste.

# Exemples (mod4.oz) I

```
% Énumerer pour éviter le blocage ?
declare
proc {D L}
   X Y Z
in
   L=X#Y
   [X Y]:::1#9
   Z::2#4
   {FD.distribute naive [Z]}
   X mod Z = Y
   {FD.distribute naive [X Y]}
end

{Browse {SearchAll D}}
% donne trois paires de domaines
```

# Exemples (mod5.oz) I

```
% Il faut quand même un propagateur !
declare
proc {D L}
   X Y Z
in
   L=X#Y
   [X Y]:::1#9
   Z::2#4
   {FD.distribute naive [Z]}
   {FD.modI X Z Y}
   {FD.distribute naive [X Y]}
end

{Browse {SearchAll D}}
% marche !
```

# Le bon endroit pour distribute

- On a le droit de mettre plusieures `FD.distribute`
- On a le droit de les intercaler avec le reste (progateurs, ...)
- Cela permet de controler plus finement la construction de l'arbre de recherche
- Mettre les distributeurs aussi tard que possible, sinon on pert l'avantage des propagateurs.

# Exemples (mod6.oz) I

```
% Solution alternative avec propagateurs pour addition et multi
declare
proc {D L}
   X Y Z
in
   L=X#Y [X Y]:::1#9 Z::2#4
   local P in
      {FD.decl P}
      X =: Z * P + Y
      Y <: Z
   end
   {FD.distribute naive [X Y Z]}
end

{Browse {SearchAll D}}
```

# Exemples (domain.oz)

```
declare X Y
{Browse [X Y]}

X :: 1#10

Y :: X#10 % bloque, attend la valeur de X

X = 5     % maintenant Y a son domaine
```

# Propagateurs pour la diségalité

- Reference Manual *System Modules*, chapitre 5 *Finite Domain Constraints*, section 5.8 *Symbolic Propagators*.
- `{FD.distinct *Dv}`
  All elements in Dv are pairwise distinct.
- `{FD.distinctOffset *Dv +Iv}`
  All sums $\mathtt{D}_i + \mathtt{I}_i$ are pairwise distinct.

# Exemples (queens.oz) I

```
% N Queens
declare
fun {Queens N}
   proc {$ Row}
      L1N ={MakeTuple c N}
      LM1N={MakeTuple c N}
   in
      for I in 1..N do L1N.I=I LM1N.I=~I end
          % L1N = [1 2 3 ... N]
          % LM1N =[~1 ~2 ... ~N]
      {FD.tuple queens N 1#N Row}
      {FD.distinct Row}
      {FD.distinctOffset Row LM1N}
      {FD.distinctOffset Row L1N}
      {FD.distribute naive Row}
   end
end
```

# Stratégies d'énumération

- Jusqu'à maintenant nous avons utilisé avec `FD.distribute` la stratégie `naive`. Qu'est-ce qu'il y a d'autre ?
- On donne à `FD.distribute` un vecteur $V$ de variables à domaine fini. Il y a deux choix à faire dans une stratégie :
  1. Choisir la variable pour laquelle on va créer 2 alternatives, parmi celles de $V$ dont le domaine a au moins 2 éléments.
  2. Une fois la variable trouvée, choisir comment couper son domaine en deux.
- Voir le document `System Modules`, Section 5.12

# La stratégie `naive`

- La stratégie `naive` choisit la première variable $X$ (dans l'ordre du vecteur $V$) qui a un domaine non trivial.
- Si le domaine actuel de $X$ est $D$, et $d$ est l'élément le plus petit de $D$, alors cette stratégie crée les deux alternatives :
  1. `X=d`
  2. `X\\=: d`

# La stratégie `ff`

- `ff` est pour *first fail*. Il s'agit d'une heuristique avec le but de trouver des échecs aussi rapidement que possible.
- La stratégie `ff` choisit la variable $X$ avec un domaine de taille minimale, parmi toutes les variables avec un domaine non-trivial.
- Si le domaine actuel de $X$ est $D$, et $d$ est l'élément le plus petit de $D$, alors cette stratégie crée les deux alternatives :
  1. `X=d`
  2. `X\\=: d`

# La stratégie `split`

- La stratégie `split` choisit la variable $X$ avec un domaine de taille minimale, parmi toutes les variables avec un domaine non-trivial.
- Si le domaine actuel de $X$ est $D$, et $d$ est l'élément médian de $D$, alors cette stratégie crée les deux alternatives :
  1. `X <: d`
  2. `X >=: d`

# La stratégie `generic`

- La stratégie `generic` permet de contrôler finement le choix de la variable et le découpage de son domaine.
- Voir la documentation.

# AtMost, AtLeast, Exactly

- Reference Manual *System Modules*, chapitre 5 *Finite Domain Constraints*, section 5.8 *Symbolic Propagators*.
- ```
  {FD.atMost *D *Dv +I}
  {FD.atLeast *D *Dv +I}
  {FD.exactly *D *Dv +I}
  ```
- - le premier argument doit être un domaine fini (ou un entier) ;
  - le deuxième argument doit être un vecteur de domaines finis ;
  - le troisième argument doit être un entier.
- At most, at least, exactly D elements of Dv are equal to I.
  ```

# Exemples (exactly1.oz)

```
% propagation de Dv vers D
declare L N in
{FD.list 5 1#10 L}
N::0#20
{Browse L}
{Browse N}

{FD.exactly N L 8}

{Nth L 1} <: 5
{Nth L 2} <: 5
```

# Exemples (exactly2.oz)

```
% propagation de D vers Dv
declare L N in
{FD.list 5 1#10 L}
N::0#20
{Browse L}
{Browse N}

{FD.exactly N L 8}

N >=: 4

{Nth L 3} =: 7
```

# Exemples (atleast.oz)

```
% Propagators FD.atLeast

declare L in
{FD.list 5 1#10 L}
{Browse L}

{Nth L 1} <: 5
{Nth L 2} <: 5

{FD.atLeast 3 L 8}
```

# Exemple : Emploi du temps d'un colloque

- On veut faire l'emploi du temps pour un colloque.
- L'emploi du temps va consister en plusieurs créneaux. Une session occupe toute la durée d'un créneau, mais on peut a priori avoir plusieurs sessions en parallèle.
- Donné :
  1. nombre de sessions, nombre de salles.
  2. contrainte possible pour une session $n$ : doit être avant une autre session $m$, ou ne doit pas être en parallèle avec une autre session $m$.
- On cherche une solution qui occupe un nombre minimal de créneaux.

# Exemples (colloque1.oz) I

```
% exemple de donnees

declare
Colloque =
  data(nbSessions:11  nbSalles:3
       constraints: [ before(4 11)  before(5 10)
                      before(6 11)
                      disjoint(1 [2 3 5 7 8 10])
                      disjoint(2 [3 4 7 8 9 11])
                      disjoint(3 [5 6 8])
                      disjoint(4 [6 8 10])
                      disjoint(6 [7 10])
                      disjoint(7 [8 9])
                      disjoint(8 [10]) ] )
```

# Solution pour le colloque

- Une variable par session, le domaine initial est l'ensemble des créneaux.
- Problème : le nombre de créneaux n'est pas connu d'avance.
- Recherche en deux dimensions : nombre de créneaux d'abord, puis affectation des créneaux aux sessions
- Il faut éviter de mettre plus d'exposés sur le même créneau qu'on a de salles : utiliser `FD.atMost`.

# L'exemple Colloque

Cet exemple montre plusieurs choses :

- Le script engendre les propagateurs à partir des données.
- Recherche multi-dimensionnelle. Faire attention à l'ordre dans lequel on distribue.
- En occurrence : `FD.int` suspend quand les bornes inférieurs et supérieures ne sont pas des entiers !

# Exemples (colloque2.oz) I

```
declare
fun {Conference Data}
   NbSessions   = Data.nbSessions
   NbSalles     = Data.nbSalles
   Constraints  = Data.constraints
   MinNbSlots   = NbSessions div NbSalles
in
   proc {$ Plan}
      NbSlots = {FD.int MinNbSlots#NbSessions}
   in
      {FD.distribute naive [NbSlots]}
      {Browse NbSlots}
      %% Plan: Session --> Slot
      {FD.tuple plan NbSessions 1#NbSlots Plan}
      %% at most NbSalles sessions per slot
      for Slot in 1..NbSlots do
         {FD.atMost NbSalles Plan Slot}
```

# Exemples (colloque2.oz) II

```
      end
      %% impose constraints
      {ForAll Constraints
       proc {$ C}
          case C
          of before(X Y) then
             Plan.X <: Plan.Y
          [] disjoint(X Ys) then
             {ForAll Ys proc {$ Y} Plan.X \=: Plan.Y end}
          end
       end}
      {FD.distribute naive Plan}
   end
end

{Browse {SearchOne {Conference Colloque}}}
```