# Programmation Logique et Par Contraintes Avancée
# Cours 3 – Programmation concurrente dataflow en Oz

Ralf Treinen

Université Paris Diderot
UFR Informatique
IRIF, équipe PPS

treinen@irif.fr

21 janvier 2019



## Contenu cours 3





## Rappel : le modèle d'exécution d'Oz

- La mémoire (store) : un système résolu d'équations
- À exécuter : une pile de paires (environement, instruction)
- L'environement lie des identificateurs à des variables de la mémoire.
- La mémoire lie des variables à des valeurs (éventuellement partielles).
- Liaison statique : les fonctions/procédures sont des clôtures.



## Appels terminaux de fonctions

### La fonction *Append* en OCaml

```
let rec append l1 l2 = match l1 with
| [] -> l2
| h1::r1 -> h1::(append r1 l2)
```

- En OCaml, l'appel récursif de la fonction *append* n'est pas terminal.
- En OCaml, toutes les instances de l'identificateur `h1` doivent être gardées sur la pile pour construire le résultat de la fonction.

## La fonction *Append* en Oz

```
declare
fun {Append L1 L2}
   case L1
   of nil    then L2
   [] H1|R1 then H1|{Append R1 L2}
   end
end
```

- En Oz, l'appel récursif de la fonction *Append* est terminal !



## La *Procédure Append* en Oz

```
declare
proc {Append L1 L2 R}
   case L1
   of nil    then R=L2
   [] H1|R1 then
         local Rr in
            R=H1|Rr
            {Append R1 L2 Rr}
         end
   end
end
```

- C'est dû au fait qu'on a des valeurs partielles.



## Listes de différence

- C'est une *paire* de listes $(l, r)$, dont
  - $r$ est un reste de $l$ ;
  - les deux peuvent être des listes partielles (qui se terminent sur des variables).
- On peut voir une liste de différence comme deux pointeurs, le premier pointe sur le début de la liste, et le deuxième sur la fin de la liste (qui peut être extensible).
- Une liste de différence $(l, r)$ *représente* une liste, c'est la différence entre $l$ et $r$ ( « les éléments entre les positions des deux pointeurs » ).
- Technique empruntée de la Programmation Logique.



## Exemples de listes de différence

| liste de différence | représente : |
|---|---|
| `X#X` | liste vide |
| `nil#nil` | liste vide |
| `[a]#[a]` | liste vide |
| `(a\|b\|c\|X)#X` | `[a b c]` |
| `(a\|b\|c\|d\|X)#(d\|X)` | `[a b c]` |
| `[a b c d]#[d]` | `[a b c]` |

(Rappel : # est le constructeur de paires)

## Exemples (append.oz) I

```
declare
% append of difference lists
proc {AppendD L1 L2 R}
   local
      X1#Y1=L1
      X2#Y2=L2
   in
      X2=Y1
      R=X1#Y2
   end
end

declare X Y in
{Browse {AppendD ((1|2|3|X)#X) ((4|5|6|Y)#Y)}}
% the first list must be extensible!
{Browse {AppendD ((1|2|3|nil)#nil) ((4|5|6|Y)#Y)}}
```



## Exemples (append-fonction.oz) I

```
declare
% est-ce vraiment pareil ?
fun {AppendFD X1#Y1 X2#Y2}
   X2=Y1
   X1#Y2
end

declare X Y in {Browse {AppendFD ((1|2|X)#X) ((3|4|Y)#Y)}}

declare X Y in
{Browse {AppendFD X Y}} % suspend !

declare X Y in
{Browse {AppendD X Y}}  % affiche une paire de variables
```



## Attention aux motifs dans les têtes de fonctions

- On a le droit de mettre un motif dans la tête d'une fonction :

```
fun {F f(Y Z)} .... end
```

- Se comporte comme

```
fun {F X}
   case X of f(Y Z) then ...
end
```

- Suspension de l'appel quand le paramêtre actuel n'est pas de la bonne forme.



## Application : liste d'attente

- aussi appelée FIFO (*first in, first out*)
- Programmation impérative (avec pointeurs) : solution efficace
- Programmation fonctionnelle : solution naïve inefficace, solution avec complexité linéaire due à Okasaki (voir le cours de *Programmation Fonctionnelle Avancée*)
- Programmation avec valeurs partielles ?

## Queues en Oz (1)

Une queue est représentée par

- sa longueur
- une liste de différence pour le contenu

```
declare
fun {NewQueue} X in q(0 X#X) end

declare
fun {First q(_ (X|_)#_)} X end

declare
fun {Length q(N _)} N end
```



## Queues en Oz (2)

Supprimer et ajouter un élément à la tête de la file :

```
declare
fun {Remove q(N (_|R)#E)}
   q(N-1 R#E)
end

declare
fun {Add q(N B#E) X}
   local E1 in
      E=X|E1
      q(N+1 B#E1)
   end
end
```



## La mémoire

La mémoire contient (modèle simplifié) :

- Des liaisons de variables à des variables :

$$x = y$$

- Des liaisons de variables à des structures :

$$x = f(x_1 \dots x_n)$$

- La simplification est : seulement tuples ; pas d'enregistrements, de procédures, ...



## Les liaisons variable - variable

- Les liaisons *entre variables* ne doivent pas être cycliques.
- En suivant toutes les liaisons variable-variable pour une variable $x$ on obtient son *représentant* $\nu(x)$ ($\nu(x) = x$ s'il n'y a pas de liaison de variable pour $x$).
- Cela définit une relation d'équivalence entre variables : $x$ et $y$ sont équivalentes quand $\nu(x) = \nu(y)$.
- Il y a plusieures techniques différentes pour l'implémentation de la fonction $\nu$, par exemple l'algorithme Union-Find dû à Tarjan.

## Les liaisons variable - structure

- Toutes les variables (la variable qui est liée, et toutes les variables qui paraissent dans la structure) sont des représentants (c.-à-d., $x = \nu(x)$)
- Toutes les variables sur la *gauche* sont différentes
- Les cycles sont permis.



## Exemple de mémoire

$$\begin{aligned} x_1 &= x_3 \\ x_2 &= x_3 \\ y_1 &= y_2 \\ x_3 &= f(y_2, a) \\ y_2 &= g(b, z) \end{aligned}$$

- $x_3$ est représentant de $\{x_1, x_2, x_3\}$
- $y_2$ est représentant de $\{y_1, y_2\}$



## Tell

- *tell* est l'opération qui ajoute une équation à la mémoire,
- Correspond à une instruction `X = t` : unification d'une équation avec la mémoire.
- L'opération échoue quand les équations sont incohérentes.
- On peut se ramener à trois cas :
  1. ajouter $x = y$
  2. ajouter $x = f(x_1 \dots x_n)$
  3. ajouter $f(x_1 \dots x_n) = g(y_1 \dots y_m)$
- On fait une *simplification relative* de l'équation.



## Tell $x = y$

- Si $\nu(x) = \nu(y)$ : rien à faire
- Sinon : Ajouter à la mémoire l'équation $\nu(x) = \nu(y)$, et puis :
  - Normaliser les liaisons variable - structure (maintenir l'invariant des liaisons variable - structure)
  - Si la mémoire contient

$$\begin{aligned} \nu(x) &= f(x_1 \dots x_n) \\ \nu(y) &= g(y_1 \dots y_m) \end{aligned}$$

Alors faire tell $f(x_1 \dots x_n) = g(y_1 \dots y_m)$

## Tell $x = f(x_1 \dots x_n)$

- Si $\nu(x)$ n'est pas encore liée à une structure : ajouter

$$\nu(x) = f(\nu(x_1) \dots \nu(x_n))$$

- S'il existe déjà une liaison

$$\nu(x) = g(y_1 \dots y_m)$$

Faire tell $f(\nu(x_1) \dots \nu(x_n)) = g(y_1 \dots y_m)$



## Tell $f(x_1 \dots x_n) = g(y_1 \dots y_m)$

- Si $f \neq g$ ou $n \neq m$ alors échec
- Sinon : tell $x_1 = y_1$, ..., $x_n = y_n$



## Terminaison du tell

- est-ce que ça termine même quand il y a des cycles dans les liaisons variable - structure ?
- Oui, car
  - toute récurrence passe par un tell $x = y$
  - si tell $x = y$ fait un appel récursif alors il a préalablement fait une nouvelle liaison variable - variable.
- Donc c'est le nombre de classes d'équivalences qui décroît !



## Ask

- Est-ce que une certaine équation (ou : filtrage par un motif) est logiquement impliquée par la mémoire ?
- Trois réponses possibles : oui, incohérent, inconnu.
- Cas de filtrage : il suffit de suivre les liaisons des variables dans la mémoire.
- Implication d'une équation entre variables : un peu plus compliquée, due au fait qu'on a des arbres potentiellement infinis.

## Ask : le cas simple

- Le cas simple est : est-ce que $\sigma \models \exists \bar{y}.x = t(\bar{y})$ où $t$ est un terme linéaire, avec les variables $\bar{y}$.
- Utilisé par example dans du code comme

```
match X with f(g(Y1 Y2 a) Z) then ... end
```

- On entre dans la branche si et dès qu'on a que

$$\sigma \models \exists y_1, y_2, z.x = f(y_1, y_2, a), z)$$

- Il suffit de suivre les liaisons pour $x$ dans la mémoire $\sigma$ pour le savoir.



## Ask $x = y$

- Utilisé par example dans du code comme

```
if X == Y then ... end
if g(X Y) == g(Y a) then ... end
```

- Exemple :

$$x_1 = f(y_1), y_1 = a, x_2 = f(y_2), y_2 = a \models x_1 = x_2$$

- Exemple avec arbres infinis :

$$x = f(y, z), y = f(x, z) \models x = y$$



## Ask $x = y$

- faire une « tentative de tell » $x = y$
- si échec : répondre *incohérence*
- si pas d'échec :
  - si pour toute nouvelle liaison entre variables $z_1 = z_2$ les deux variables étaient, avant le tell, liées à des structures : implication !
  - sinon : on ne sait pas, suspension du thread.



## Exemple 1 Ask

Est-ce que $x = f(x_1), y = f(y_1), x_1 = a, y_1 = b \models x = y$ ?

- tell $x = y$ échoue.
- réponse : incohérence !

## Exemple 2 Ask

Est-ce que $x = f(y,z), y = f(x,z) \models x = y$ ?

- tell $x = y$ n'échoue pas, et ajoute seulement $x = y$.
- $x$ et $y$ liées à des structures dans la mémoire de départ
- Réponse : oui !



## Exemple 3 Ask

Est-ce que $x = f(y,z), y = f(x,y) \models x = y$ ?

- tell $x = y$ n'échoue pas, et ajoute les équations $x = y$ et $y = z$
- $z$ n'est pas liée à une structure.
- Réponse : on ne sait pas !
- Il faut attendre plus d'information sur $z$ avant de décider.



## Monotonie de la mémoire

- Pendant le calcul, la mémoire croît de façon *monotone* !
- Si la mémoire est $\sigma_1$ à un certain moment, et plus tard $\sigma_2$, alors $\sigma_2 \models \sigma_1$ : toute conséquence logique de $\sigma_1$ est aussi une conséquence de $\sigma_2$.
- L'echec d'un tell ou une réponse (affirmative ou négative) d'un ask sont dûs à des implications de la mémoire :
  - Echec de tell(s=t) : $\sigma \models \neg(s = t)$
  - Réponse aff. de ask(s=t) : $\sigma \models \forall x_1, \ldots x_n . s = t$
  - Réponse neg. de ask(s=t) : $\sigma \models \forall x_1, \ldots x_n . s \neq t$
- Si un tell échoue, ou un ask répond « oui » ou « incohérent » pour $\sigma_1$, alors c'est aussi le cas pour $\sigma_2$.



## Des threads déclaratifs

Avec plusieurs threads,

- Quand un programme séquentiel (sans threads) donne un résultat, l'ajout des thread ne change pas ce résultat.
- Un programme avec threads peut éventuellement avancer là où le programme séquentiel bloque.
- Le calcul d'un programme avec threads peut être *incrémental*.

## L'instruction `thread` en Oz

- Syntaxe : `thread <s> end`
- Sémantique :
  - il y a plusieurs piles d'exécution ;
  - les threads différents utilisent la même mémoire !
- Le *Browser* est toujours exécuté dans son propre thread.
- Toute requête donnée à l'interpréteur est exécutée dans son propre thread.



## Exemples (browser.oz)

```
% the browser runs its own threads
declare X Y Z in
{Browse X}
```

X=a | Y

Y=b | Z



## Exemples (thread.oz)

```
% thread .. end est transparent pour les expressions
declare
fun {Fib X}
   if X =< 2 then 1
   else thread {Fib X-1} end + {Fib X-2}
   end
end
in
{Browse {Fib 10}}
```



## Synchronisation de threads

- Certaines instructions peuvent bloquer quand la mémoire ne contient pas suffisamment d'information pour conclure :
  - test e1 == e2 ;
  - instructions **case** et **if** ;
  - certaines procédure de la bibliothèque standard, comme Label ou Arity ;
  - l'appel d'une procédure (ou fonction) quand l'identificateur à la position de la procédure n'est pas liée à une valeur.
- Dans ce cas le thread est suspendu.

## Exemples (synch1.oz)

```
% data-flow synchronisation
declare X Y in
if X==Y then {Browse X} end
```

```
X=f(a)
```

```
Y=f(a)
```



## Exemples (synch2.oz)

```
% infinite trees
declare X Y in
if X==Y then {Browse X} end
```

```
X=f(Y)
```

```
Y=f(X)
```



## Ramassage de miettes

- Angl. : *garbage collector*
- Récupération de la mémoire qui n'est pas accessible (comme dans des langages fonctionnels, Java, ...)
- Peut aussi détruire un thread qui bloque et attend des informations sur une variable qui n'est plus accessible par des autres threads.



## Pas de non-déterminisme observable

- L'entrelacement de l'exécution des thread n'est pas être observable si on se restreint au modèle d'exécution vu au cours 2 (il est bien sûr observable quand les threads font un affichage). Ici, la seule observation permise est le contenu de la mémoire finale.
- C'est une conséquence du fait que les structures de contrôle du langage (if, case) utilisent ask avec une sémantique logique, et de la monotonie de la mémoire !

## Presque pas de non-déterminisme observable

- En vérité : on peut observer l'entrelacement dans un modèle dans lequel on peut distinguer des erreurs d'exécution différentes, ou dans lequel on peut les capturer !
- Il y a un écart entre le modèle de la sémantique et la réalité quand le modèle de la sémantique ne modélise pas la capture des exception.



## Exemples (observ2.oz)

```
local X in
   try
      thread {Delay 1} X=a end
      thread {Delay 2} X=b end
   catch failure then skip
   end
   {Browse X}
end
```



## Synchronisation par demande ou par offre

- Synchronisation par offre (*supply-driven concurrency*) est le modèle de Oz : le consommateur doit attendre le producteur.
- Synchronisation par demande (*demand-driven concurrency*) est le modèle de Haskell (lazy evaluation) : le producteur doit attendre le consommateur.



## Produire un flot d'entiers

Écrire un prédicat {Generate X} qui lie la variable X à la liste des nombres successives à partir de 2.

```
declare
fun {Generate}
% infinite list of integers staring from 2
   local
      fun {GenAux Curr}
         {Delay 1000}
         Curr|{GenAux Curr+1}
      end
   in
      {GenAux 2}
   end
end
```

## Exemples (inflist2.oz) I

```
% Don't do this !
% The browser starts only when {Generate} has finished
% {Browse {Generate}}

% Somewhat better
% still creates an unstopable thread
{Browse thread {Generate} end}

% Create a thread with a stop button
declare Handle in
{Browse thread {Thread.this Handle} {Generate} end}

{Thread.terminate Handle}
```



## Listes paresseuses (lazy lists)

- Ce sont des listes qui sont construites seulement tant que demandée.
- Intéressant pour la synchronisation par demande (en opposition à la synchronisation par offre).
- Il y a des langages qui réalisent les constructeurs de données (listes, par exemple) de façon paresseuse : Haskell, Miranda.
- En Ocaml, une expression peut être paresseuse.
- En Oz on peut définir des *fonctions* (ou procédures) paresseuses.



## Exécution paresseuse en Oz

- Il s'agit d'un mécanisme (extension du langage vu jusqu'à maintenant) qui permet de créer un thread « paresseux » qui est exécuté seulement quand nécessaire.
- La construction est ByNeed qui crée un thread, mais avec une indication à l'ordonnanceur que le thread est seulement à exécuter quand démandé.
- Nous n'entrons pas dans les détails (ça serait pour un cours sur la programmation concurrente en Oz)
- Utilisé pour réaliser les *listes paresseuses* en Oz.



## Exemples (lazy1.oz)

```
declare
fun {LGenerate}
   local
      fun lazy {GenAux Curr}
         {Show Curr} % pour observer l'avancement
         Curr|{GenAux Curr+1}
      end
   in
      {GenAux 2}
   end
end
```

# Exemples (lazy2.oz) I

```
declare X in
{LGenerate X}
{Browse X}

declare
fun {Tail L} L.2 end

declare
proc {Touch N L}
   if N>0 then {Touch N-1 {Tail L}} else skip end
end

{Touch 3 X}
{Touch 7 X}
```