

Programmation Logique et Par Contraintes  
Avancée  
Cours 2 – Le modèle d'exécution d'Oz

Ralf Treinen

Université Paris Diderot  
UFR Informatique  
Laboratoire Preuves, Programmes et Systèmes  
treinen@pps.jussieu.fr

Contenu cours 2

Machine abstraite pour l'exécution de Oz

Exécution d'un programme mini-Oz

Les règles d'exécution

Exemple

Appels terminaux

Syntaxe de *mini-Oz*

Langage noyau, utilisé seulement pour définir la sémantique.

Une instruction  $\langle s \rangle$  peut être :

- ▶ skip
- ▶  $\langle s \rangle_1 \langle s \rangle_2$
- ▶ local  $\langle x \rangle$  in  $\langle s \rangle$  end
- ▶  $\langle x \rangle = \langle t \rangle$  ( $\langle t \rangle$  peut être une procédure)
- ▶ if  $\langle x \rangle$  then  $\langle s \rangle_1$  else  $\langle s \rangle_2$  end
- ▶ case  $\langle x \rangle$  of  $\langle p \rangle$  then  $\langle s \rangle_1$  else  $\langle s \rangle_2$  end
- ▶  $\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$

Raccourcies syntaxiques (1)

Le langage Oz complet peut être traduit en mini-Oz :

local avec plusieurs variables

```
local X Y Z in <i> end
```

peut être traduit en

```
local X in
  local Y in
    local Z in
      <i>
    end
  end
end
```

## Raccourcis syntaxiques (2)

### declare

```
declare X in <i>
```

peut être traduit (pour un programme complet) en

```
local X in <i> end
```

## Raccourcis syntaxiques (3)

### local avec affectation d'une valeur

```
local proc {P X} <i1> end in <i2> end
```

peut être traduit en

```
local P in
  P = proc {$ X} <i1> end
  <i2>
end
```

## Raccourcis syntaxiques (4)

### utiliser des termes à la place de variables

```
if <t> then <i1> else <i2> end
```

peut être traduit en

```
local X in
  X = <t>
  if X then <i1> else <i2> end
end
```

(où *X* est un nouvel identificateur)

## Modèles de mémoire

- ▶ à *valeur* : toute variable a une valeur, et cette valeur ne change pas pendant l'exécution du programme (langages fonctionnels : OCaml etc.)
- ▶ à *cellule* : la valeur d'une variable peut changer pendant l'exécution d'un programme (langages impératifs : C, Pascal, C++, etc.)
- ▶ à *affectation unique* : Une variable peut être non liée, ou liée à une valeur. Une fois créée, la liaison d'une variable ne change plus (langages logiques et/ou à contraintes : Prolog, Oz, etc.)

## Valeurs partielles

Si on a à la fois

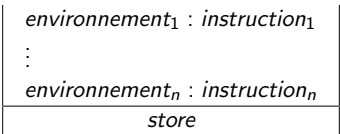
- ▶ une mémoire à affectation unique
- ▶ des valeurs hiérarchiques (arbres, etc.)

alors on obtient un langage à affectation unique avec des valeurs *partielles* : une variable peut être liée à une valeur qui contient des variables, par exemple  $x = f(y, g(a, z))$ .

## Composantes de la machine abstraite

- ▶ Une mémoire (angl. : *store*). En général, la mémoire contient une contrainte en forme résolue (voir plus tard).
- ▶ Une pile de paires (environnement, instruction).
- ▶ Un environnement lie des *identificateurs* (qui paraissent dans le programme) à des *variables* (qui existent dans la mémoire)
- ▶ l'environnement est nécessaire pour la gestion des variables locales et de la liaison statique.
- ▶ Les procédures sont représentées dans la mémoire comme des clôtures (liaison statique!)

## Représentation graphique



## Exécution d'un programme

- ▶ État initial pour l'exécution d'un programme  $s$  :



- ▶ État terminal :



### La mémoire

Dans un premier temps, la mémoire est un système d'équations résolues :

$$\begin{aligned}
 x_1 &= t_1 \\
 &\vdots \\
 x_n &= t_n
 \end{aligned}$$

où toutes  $x_i \neq x_j$  pour  $i \neq j$ . On peut le voir comme

- ▶ liaison des variables à des valeurs partielles
- ▶ les cycles sont permises!

### Exécution : le cas du skip

$$\frac{\begin{array}{|l} E : \text{skip} \\ \text{reste} \end{array}}{\sigma} \Rightarrow \frac{\text{reste}}{\sigma}$$

### Exécution : le cas d'une composition

$$\frac{\begin{array}{|l} E : s_1 \ s_2 \\ \text{reste} \end{array}}{\sigma} \Rightarrow \frac{\begin{array}{|l} E : s_1 \\ E : s_2 \\ \text{reste} \end{array}}{\sigma}$$

### Exécution : le cas d'une portée locale

$$\frac{\begin{array}{|l} E : \text{local } X \text{ in } s \text{ end} \\ \text{reste} \end{array}}{\sigma} \Rightarrow \frac{\begin{array}{|l} E \cup \{X \mapsto x\} : s \\ \text{reste} \end{array}}{\sigma}$$

- ▶ où  $x$  est une nouvelle variable.
- ▶  $E \cup F$  est la mise à jour de l'environnement  $E$  par l'environnement  $F$ . Si l'identificateur  $X$  est lié à la fois par  $E$  et par  $F$  alors  $E \cup F$  lie  $X$  à  $F(X)$ .

Exécution : le cas d'une équation  $X = t$  (1)

$$\frac{\left| \begin{array}{l} E : X = t \\ \text{reste} \\ \hline \sigma \end{array} \right.}{\Rightarrow} \frac{\left| \begin{array}{l} \text{reste} \\ \hline \sigma' \end{array} \right.}{}$$

- ▶ Soit  $E(X = t)$  obtenu par remplaçant tout identificateur  $Y$  par  $E(Y)$ .
- ▶ Si  $\sigma'$  est la forme résolue de  $\sigma \cup \{E(X = t)\}$ .
- ▶  $\sigma'$  peut lier des variables à des nouvelles clôtures (avec environnement  $E$ )

Exécution : le cas d'une équation  $X = t$  (2)

$$\frac{\left| \begin{array}{l} E : X = t \\ \text{reste} \\ \hline \sigma \end{array} \right.}{\Rightarrow} \perp$$

- ▶ Soit  $E(X = t)$  obtenu par remplaçant tout identificateur  $Y$  par  $E(Y)$ .
- ▶ Si  $\sigma \cup \{E(X = t)\}$  n'a pas de solution.

Exécution : le cas du if (1)

$$\frac{\left| \begin{array}{l} E : \text{if } X \text{ then } s_1 \text{ else } s_2 \text{ end} \\ \text{reste} \\ \hline \sigma \end{array} \right.}{\Rightarrow} \frac{\left| \begin{array}{l} E : s_1 \\ \text{reste} \\ \hline \sigma \end{array} \right.}{}$$

- ▶ Si  $\sigma(E(X)) = \text{true}$

Exécution : le cas du if (2)

$$\frac{\left| \begin{array}{l} E : \text{if } X \text{ then } s_1 \text{ else } s_2 \text{ end} \\ \text{reste} \\ \hline \sigma \end{array} \right.}{\Rightarrow} \frac{\left| \begin{array}{l} E : s_2 \\ \text{reste} \\ \hline \sigma \end{array} \right.}{}$$

- ▶ Si  $\sigma(E(X)) = \text{false}$

Exécution : le cas du if (3)

$$\frac{\left| \begin{array}{l} E : \text{if } X \text{ then } s_1 \text{ else } s_2 \text{ end} \\ \text{reste} \end{array} \right|}{\sigma} \Rightarrow \perp$$

- ▶ Si  $\sigma(E(X))$  est une valeur  $\notin \{\text{true}, \text{false}\}$

Exécution : le cas du if (4)

- ▶ Pourquoi un 4ème cas??
- ▶  $\sigma(E(X))$  peut être une variable!
- ▶ Pas de règle de transformation pour ce cas : suspension du fil d'exécution!

Exécution : le cas du case

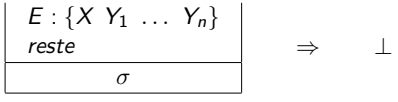
- ▶ Similaire au if, mais peut étendre l'environnement (par les identificateurs du motif) et la mémoire (pour les variables liées aux identificateurs du motif).
- ▶ Le calcul avance si on peut conclure que *toutes* les « valeurs possibles » de  $E(X)$  sont filtrées par le motif (cas then), ou si on peut conclure qu'*aucune* « valeur possible » de  $E(X)$  est filtré par  $p$  (cas else).
- ▶ Sinon : suspension du fil.
- ▶ Voir la semaine prochaine!

Exécution : Appel d'une procédure (1)

$$\frac{\left| \begin{array}{l} E : \{X \ Y_1 \ \dots \ Y_n\} \\ \text{reste} \end{array} \right|}{\sigma} \Rightarrow \frac{\left| \begin{array}{l} F \cup \{Z_1 \mapsto E(Y_1), \dots, Z_n \mapsto E(Y_n)\} : s \\ \text{reste} \end{array} \right|}{\sigma}$$

- ▶ Si  $\sigma(E(X))$  est la clôture  $(F, s)$  avec les arguments formels  $Z_1, \dots, Z_n$

Exécution : Appel d'une procédure (2)



- ▶ Si  $\sigma(E(X))$  est une valeur qui n'est pas une clôture à  $n$  arguments

Exécution : Appel d'une procédure (3)

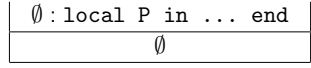
- ▶ Un appel de procédure  $\{ X \ Y_1 \ \dots \ Y_n \}$  suspend quand  $\sigma(E(X))$  est une variable.

Un Exemple

```

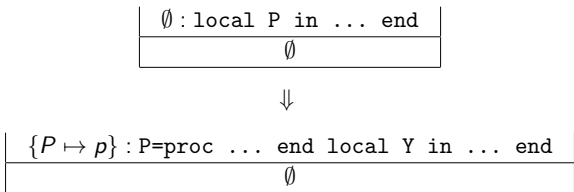
local P in
  P = proc {$ X} X=1 end
  local Y in
    {P Y}
    Y=2
  end
end
end
    
```

Configuration Initiale



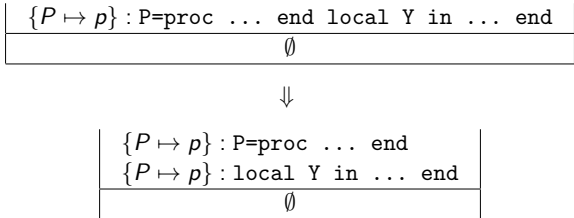
- ▶ sur la pile : le programme départ dans un environnement vide
- ▶ mémoire vide

Exécution du local P

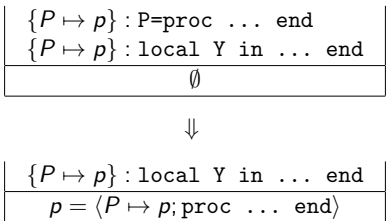


► *p* est une nouvelle variable

Décomposition de l'instruction composée

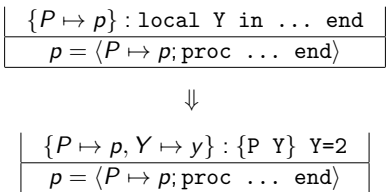


Traiter l'équation pour P



► La valeur de la variable *p* est une clôture

Exécution du local Y



► *y* est une nouvelle variable

### Décomposition de l'instruction composée

$$\frac{\{P \mapsto p, Y \mapsto y\} : \{P \ Y\} \ Y=2}{p = \langle P \mapsto p; \text{proc } \dots \text{ end} \rangle} \Downarrow \frac{\{P \mapsto p, Y \mapsto y\} : \{P \ Y\} \quad \{P \mapsto p, Y \mapsto y\} : Y=2}{p = \langle P \mapsto p; \text{proc } \dots \text{ end} \rangle}$$

### Appel de la procédure P

$$\frac{\{P \mapsto p, Y \mapsto y\} : \{P \ Y\} \quad \{P \mapsto p, Y \mapsto y\} : Y=2}{p = \langle P \mapsto p; \text{proc } \dots \text{ end} \rangle} \Downarrow \frac{\{P \mapsto p, X \mapsto y\} : X=1 \quad \{P \mapsto p, Y \mapsto y\} : Y=2}{p = \langle P \mapsto p; \text{proc } \dots \text{ end} \rangle}$$

- ▶ L'environnement lie P à p
- ▶ La mémoire donne pour p une clôture
- ▶ On remplace l'appel par le corps de la procédure
- ▶ Environnement du corps : environnemnt de la clôture plus liaison pour les paramètres formels

### Traiter l'équation

$$\frac{\{P \mapsto p, X \mapsto y\} : X=1 \quad \{P \mapsto p, Y \mapsto y\} : Y=2}{p = \langle P \mapsto p; \text{proc } \dots \text{ end} \rangle} \Downarrow \frac{\{P \mapsto p, Y \mapsto y\} : Y=2}{p = \langle P \mapsto p; \text{proc } \dots \text{ end} \rangle \quad y = 1}$$

### Traiter l'équation

$$\frac{\{P \mapsto p, Y \mapsto y\} : Y=2}{p = \langle P \mapsto p; \text{proc } \dots \text{ end} \rangle \quad y = 1} \Downarrow \perp$$

- ▶ l'environnement lie Y à la variable y
- ▶  $y = 1 \wedge y = 2$  est contradictoire

## Appels terminaux de fonctions

### La fonction *Append* en OCaml

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

- ▶ L'appel récursif de la fonction *append* n'est pas terminal.
- ▶ Toutes les instances de la variable *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
```

L'appel récursif de la fonction *Append* est terminal ! Pourquoi ?

## 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.