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.univ-paris-diderot.fr

Contenu cours 2

Procédures et Fonctions

- Les fonctions sont simplement un raccourci pour des procédures.
- On peut définir une fonction et l'utiliser comme une procédure, ou définir une procédure et l'utiliser comme une fonction.

Définition d'une fonction

```
fun {P X1 ... Xn} i1 ... im expr end
```

est une abbréviation pour

```
proc {F X1 ... Xn R} i1 ... im R=exp end
```

Exemple : Définition d'une fonction

La définition de la fonction

```
fun {Max X Y} if X > Y then X else Y end end
```

est une abréviation pour

```
proc {Max X Y Z} if X>Y then Z=X else Z=Y end end
```

Appeler une procédure comme une fonction

Un appel d'une procédure dans un contexte où une expression est attendue :

```
context[{P X1 ... Xn}]
```

peut être vu comme une abbréviation pour

```
local R in
  {P X1 ... Xn R}
  context[R]
end
```

Exemple : Appeler une procédure comme une fonction

```
declare
proc {P X Y Z} Z=f(X Y) end
{Browse {P a b}}
```

affiche $f(a\ b)$.

Procédures et fonctions

- Une fonction est une abbréviation pour une procédure.
- Un appel de procédure est une instruction, un appel de fonctio n est une expression.
- Si la procédure P prend $n+1$ arguments alors la fonction P prend n arguments.
- Pas d'application partielle.

[Voir la Démo 1]

Déclaration avec Binding

- On peut mettre certaines instructions entre `declare`/`local` et `in` : Des équations et des définitions de procédures (ou fonctions)
- Effet : sont déclarées toutes les variables sur les côtés gauches des équations, et la variable directement après `proc`/`fun`.

[Voir la Démo 2]

Procédures et fonctions anonymes

- On peut utiliser le symbole $ à la première position. de l'entête d'un définition de procédure/fonction.
- Effet : on obtient une *expression* dont la valeur est la procédure/fonction.
- Cela correspond à des lambda-expressions :

`function x -> x+x`	OCaml
$\lambda x.x + x$	Math (lambda-calcul)
`fun {$ X} X+X end`	Oz

[Voir la Démo 3]

Syntaxe de *mini*-Oz

Langage noyau, utilisé seulement pour définir la sémantique.
Une instruction $< s >$ peut être :

- `skip`
- $< s >_1 < s >_2$
- `local` $< x >$ `in` $< s >$ `end`
- $< x > = < t >$ ($< t >$ peut être une procédure)
- `if` $< x >$ `then` $< s >_1$ `else` $< s >_2$ `end`
- `case` $< x >$ `of` $< p >$ `then` $< s >_1$ `else` $< s >_2$ `end`
- $\{ < x > < y >_1 \ldots < y >_n \}$

Syntaxe de mini-Oz

Un terme peut être :

- un identificateur x
- un terme composé $f(< t >_1 \ldots < t >_n)$
- une procédure `proc` $\{\$\ y_1 \ldots y_n\}$ $< s >$ `end`

Remarques :

- Ici : seulement termes classiques, pas d'enregistrements (mais l'algorithme d'unification se généralise facilement aux enregistrements)
- Les fonctions sont une abbréviation pour les procédures.

Raccourcies syntaxiques (1)

Le language 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
```

Raccourcies syntaxiques (2)

`declare`

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

peut être traduit (pour un programme complet) en

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

Raccourcies 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
```

Raccourcies 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)

Raccourcies syntaxiques (5)

Les instructions d'unification

```
s=t
```

peut être traduit en

```
local X in
  X = s
  X = t
end
```

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 *a ectation 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

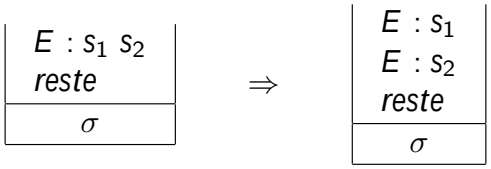
- une mémoire à affectation unique
- des valeurs hiérarchiques (arbres, etc.)

alors on obtient un langage à affection unique avec des valeurs *partielles* : une variable peut être liéée à une valeur qui contient des variables, par exemple $x = f(y, g(a, z))$.
[Voir la Démo 4]

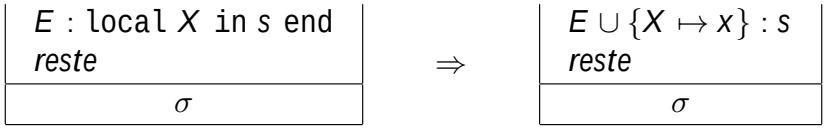
Composantes de la machine abstraite

- Une mémoire (angl. : *store*). En général, la mémoire contient une contrainte en forme résolue (voir le cours suivant).
- Une pile de paires (environnement, instruction).
- Un environnement lie des *identi cateurs* (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 !)

Exécution : le cas d'une composition

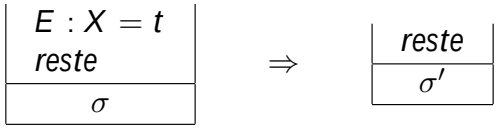


Exécution : le cas d'une portée locale



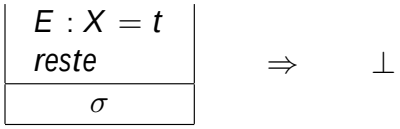
- où x est une nouvelle variable.
- $E \cup F$ est la mise à jour de l'environnement E par l'environnement F. Si l'identificateur X est lié à la fois par E et par F alors $E \cup F$ lie X à $F(X)$.

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



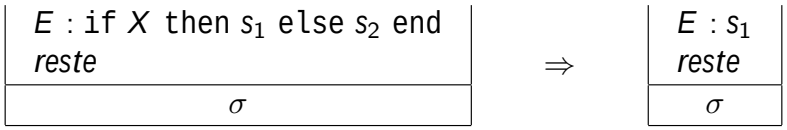
- Soit $E(X = t)$ obtenu par remplaçant tout identificateur Y par $E(Y)$.
- Si σ' est la forme résolue de $\sigma \cup \{E(X = t)\}$.
- σ' peut lier des variables à des nouvelles clôtures (avec environnement E)

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



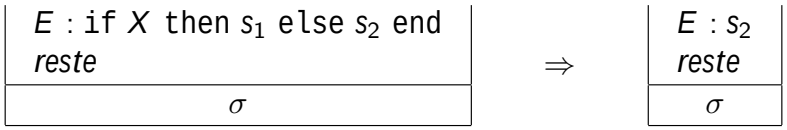
- Soit $E(X = t)$ obtenu par remplaçant tout identificateur Y par $E(Y)$.
- Si $\sigma \cup \{E(X = t)\}$ n'a pas de solution.

Exécution : le cas du `if` (1)



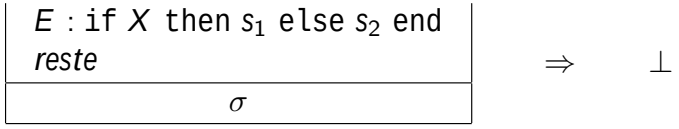
- Si $\sigma(E(X)) = \texttt{true}$

Exécution : le cas du `if` (2)



- Si $\sigma(E(X)) = \texttt{false}$

Exécution : le cas du `if` (3)



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

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 !

[Voir la Démo 4]

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)

$E : \{X\ Y_1\ \ldots\ Y_n\}$
reste
σ

$\Rightarrow$

$F \cup \{Z_1 \mapsto E(Y_1), \ldots, Z_n \mapsto E(Y_n)\} : s$
reste
σ

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

Exécution : Appel d'une procédure (2)

$E : \{X\ Y_1\ \ldots\ Y_n\}$
reste
σ

$\Rightarrow \quad \bot$

- 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 $\{\ X\ Y_1\ \ldots\ Y_n\ \}$ 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
```

Configuration Initiale

$\emptyset$: `local P in ... end`
$\emptyset$

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

Exécution du `local P`

$\emptyset$: `local P in ... end`
$\emptyset$

$\Downarrow$

$\{P \mapsto p\}$: `P=proc ... end local Y in ... end`
$\emptyset$

- p est une nouvelle variable

Décomposition de l'instruction composée

$\{P \mapsto p\}$: `P=proc ... end local Y in ... end`
$\emptyset$

$\Downarrow$

$\{P \mapsto p\}$: `P=proc ... end` $\{P \mapsto p\}$: `local Y in ... end`
$\emptyset$

Traiter l'équation pour P

$\{P \mapsto p\} : \texttt{P=proc ... end}$
$\{P \mapsto p\} : \texttt{local Y in ... end}$
$\emptyset$

$\Downarrow$

$\{P \mapsto p\} : \texttt{local Y in ... end}$
$p = \langle P \mapsto p; \texttt{proc ... end} \rangle$

- La valeur de la variable p est une clôture

Exécution du `local Y`

$\{P \mapsto p\} : \texttt{local Y in ... end}$
$p = \langle P \mapsto p; \texttt{proc ... end} \rangle$

$\Downarrow$

$\{P \mapsto p, Y \mapsto y\} : \{\texttt{P Y}\}\ \texttt{Y=2}$
$p = \langle P \mapsto p; \texttt{proc ... end} \rangle$

- y est une nouvelle variable

Décomposition de l'instruction composée

$\{P \mapsto p, Y \mapsto y\} : \{\texttt{P Y}\}\ \texttt{Y=2}$
$p = \langle P \mapsto p; \texttt{proc ... end} \rangle$

$\Downarrow$

$\{P \mapsto p, Y \mapsto y\} : \{\texttt{P Y}\}$
$\{P \mapsto p, Y \mapsto y\} : \texttt{Y=2}$
$p = \langle P \mapsto p; \texttt{proc ... end} \rangle$

Appel de la procédure P

$\{P \mapsto p, Y \mapsto y\} : \{\texttt{P Y}\}$
$\{P \mapsto p, Y \mapsto y\} : \texttt{Y=2}$
$p = \langle P \mapsto p; \texttt{proc ... end} \rangle$

$\Downarrow$

$\{P \mapsto p, X \mapsto y\} : \texttt{X=1}$
$\{P \mapsto p, Y \mapsto y\} : \texttt{Y=2}$
$p = \langle P \mapsto p; \texttt{proc ... end} \rangle$

- 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

Traiter l'équation

$\{P \mapsto p, X \mapsto y\}$: X=1
$\{P \mapsto p, Y \mapsto y\}$: Y=2
$p = \langle P \mapsto p; \texttt{proc } \ldots \texttt{ end} \rangle$

$\Downarrow$

$\{P \mapsto p, Y \mapsto y\}$: Y=2
$p = \langle P \mapsto p; \texttt{proc } \ldots \texttt{ end} \rangle$
$y = 1$

Traiter l'équation

$\{P \mapsto p, Y \mapsto y\}$: Y=2
$p = \langle P \mapsto p; \texttt{proc } \ldots \texttt{ end} \rangle$
$y = 1$

$\Downarrow$

$\bot$

- 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 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
```

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.