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

Ralf Treinen

Université Paris Cité
UFR Informatique
Institut de Recherche en Informatique Fondamentale

treinen@irif.fr

24 janvier 2024

Contenu cours 3

Rappel : le modèle d'exécution d'Oz

- La mémoire (store) : un système résolu d'équations
- À exécuter : une pile de paires (environnement, instruction)
- L'environnement 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.

Attention aux motifs dans les entêtes de fonctions

- On a le droit de mettre un motif dans l'entête d'une fonction :

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

- Se comporte comme

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

- Suspension (ou échec) de l'appel quand le paramètre actuel n'est pas de la bonne forme.

Exemples (motif.oz)

```
declare
fun {F f(Y Z)} Y+Z end

{Browse {F (f(2 3))}}

declare X
{Browse {F X}}

X=f(5 7)
```

La mémoire

La mémoire contient (modèle simplifié) :

- Des liaisons de variables à des variables :

$$x = y$$

- Des liaisons de variables à des structures :

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

- C'est à dire les termes sont *plats*.
- 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 plusieurs techniques différentes pour l'implémentation de la fonction ν, 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.

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.

Exemple 1 Ask

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

- réponse : incohérence !

Exemples (ask1.oz)

```
declare X X1 Y Y1 in
X=f(X1)
Y=f(Y1)
X1=a
Y1=b

case X
of !Y then {Browse ok} % Y must be the global Y
else {Browse ko}
end
```

Exemple 2 Ask

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

- Réponse : oui !

Exemples (ask2.oz)

```
declare X Y Z in
X=f(Y Z)
Y=f(X Z)

case X
of !Y then {Browse ok} % Y must be the global Y
else {Browse ko}
end
```

Exemple 3 Ask

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

- Réponse : on ne sait pas !
- Il faut attendre plus d'information sur z avant de décider.

Exemples (ask3.oz)

```
declare X Y Z in
X=f(Y Z)
Y=f(X Y)

case X
of !Y then {Browse ok} % Y must be the global Y
else {Browse ko}
end

Z = 42
```

Monotonie de la mémoire

- Pendant le calcul, la mémoire croît de façon *monotone* !
- Si la mémoire est σ_1 à un certain moment, et plus tard σ_2, alors $\sigma_2 \models \sigma_1$: toute conséquence logique de σ_1 est aussi une conséquence de σ_2.
- L'échec d'un tell ou une réponse (affirmative ou négative) d'un ask sont dûs à des implications de la mémoire :
  - Échec 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 σ_1, alors c'est aussi le cas pour σ_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 procédures 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 observable si on se restreint au modèle d'exécution vu au cours 2 (il est bien sûr observable quand les threads ont un effet de bord). 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 !

L'ordre d'exécution des threads n'est pas observable 1

- Il y a 3 instructions en mini-Oz qui agissent sur la mémoire :
  - $x = t$ (tell)
  - `if` $< x >$ `then` $< s >_1$ `else` $< s >_2$ `end`
  - `case` $< x >$ `of` $< p >$ `then` $< s >_1$ `else` $< s >_2$ `end`
- Pour faire simple nous supposons que les instructions $< s >_1$ et $< s >_2$ sont également des tells.
- Chacune des ces 3 instructions est de la forme suivante :
  - il y a un certains nombre de choix.
  - chaque cas est un tell sous la condition d'un ask.
- Pour l'instruction $x = t$: il y a un seul choix, et la condition est `True`.

L'ordre d'exécution des threads n'est pas observable 2

- `if` $< x >$ `then` $< s >_1$ `else` $< s >_2$ `end`

ask	tell
$x = true$	$< s >_1$
$x = false$	$< s >_2$
$x \neq true \wedge x \neq false$	$\bot$

- `case` $< x >$ `of` $< p >$ `then` $< s >_1$ `else` $< s >_2$ `end`

ask	tell
$\exists \bar{X}.x = p$	$< s >_1$
$\neg\exists \bar{X}.x = p$	$< s >_2$

où $\bar{X}$ est l'ensemble des variables de p.

L'ordre d'exécution des threads n'est pas observable 3

- Imaginons un store σ, et deux threads qui sont en mesure d'avancer avec une instruction qui agit sur le store.
- Dans un thread il y a donc ask ϕ_1 tell ψ_1, et $\sigma \models \phi_1$
- Dans l'autre il y a ask ϕ_2 tell ψ_2, et $\sigma \models \phi_2$.
- Si le premier thread exécute d'abord on passe à un store logiquement équivalent à $\sigma \wedge \psi_1$. On a $\sigma \wedge \psi_1 \models \phi_2$, donc on obtient un store équivalent à $\sigma \wedge \psi_1 \wedge \psi_2$.
- Si le deuxième thread exécute d'abord on passe à un store logiquement équivalent à $\sigma \wedge \psi_2$. On a $\sigma \wedge \psi_2 \models \phi_1$, donc on obtient un store équivalent à $\sigma \wedge \psi_2 \wedge \psi_1$.
- Les deux stores sont logiquement équivalents.

Presque pas de non-déterminisme observable

- En vérité : on peut observer l'entrelacement quand on permet des effets de bords comme des entrées/sorties, ou la capture des exceptions.
- La réalité est plus riche que le langage noyaux utilisé au chapitre 2.

Exemples (observ2.oz)

```
local X in
   {Browse X}
   thread {Delay 1} X=a end
   thread {Delay 2} X=b end
end
```

Est-ce qu'il y a un test si une valeur est une variable ?

- Non !
- (en vérité si, mais dans une bibliothèque pour la programmation bas niveau - en verra dans quelques semaines pourquoi)
- Raison : si c'était possible alors le comportement ne serait plus monotone !

```
thread if IsVar(X) then Y=1 else Y=2 fi end
thread X=42 end
```

- La conditionnelle prendrait, selon l'ordonnancement, soit la branche positive, soit la branche négative.

Un “test” qu’une valeur n’est *pas* une variable

- C’est facile !
- ```
  fun {IsNotVar X} if X==42 then true else true end
  ```
- Envoie **`true`** quand l’argument n’est pas une variable.
- N’envoie jamais **`false`**.
- Le calcul suspend quand l’argument est une variable.
  ```

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

```
% 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.

Exemples (lazy1.oz) I

```
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}
```