

Programmation Logique et Par Contraintes Avancée

Cours 3 – Programmation concurrente dataflow en Oz

Ralf Treinen

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

Contenu cours 3

La technique des listes de différence

Ask et Tell

Programmation concurrente dataflow en Oz

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)

Application : concaténer deux listes

```
declare
% append of two extensible difference lists
% yields a difference list
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)}}}
```

Application : inverser une liste

```
% list reversal using difference lists
declare
fun {Reverse Xs}
  % ReverseD reverses a (normal) list
  % Result is a difference list
  proc {ReverseD Xs (Rs#RsRest)}
    case Xs
    of nil then Rs=RsRest
    [] X|Xr then {ReverseD Xr (Rs#(X|RsRest))}
    end
  end
in
  local Y in {ReverseD Xs (Y#nil)} Y end
end

{Browse {Reverse [a b c d e]}}
```

Application : liste d'attente

- ▶ aussi appelée FIFO (first in, first out)
- ▶ Programmation impérative (avec pointeurs) : solution efficace
- ▶ Programmation fonctionnelle : solution avec complexité *accumulée* linéaire (due à Okasaki, utilisant deux piles)
- ▶ Programmation avec valeurs partielles ?

Queues en Oz (1)

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

Une queue est représentée par

- ▶ sa longueur
- ▶ une liste de différence pour le contenu

Queues en Oz (2)

```
declare
fun {Remove Q}
  case Q of q(N _|S1 E) then
    q(N-1 S1 E)
  end
end
```

Queues en Oz (3)

```
declare
fun {Add Q X}
  case Q of q(N S E) then E1 in
    E=X|E1
    q(N+1 S E1)
  end
end
```

Pourquoi est-ce qu'on ne peut pas simplement utiliser un motif $q(N S X|E1)$, comme pour Remove?

Queues en Oz (4)

```
declare
fun {First Q}
  case Q of q(_ X|_ _) then X end
end
fun {Length Q}
  case Q of q(N _ _) then N end
end
```

La mémoire

La mémoire contient (modèle simplifié) :

- ▶ Des liaisons de variables à des variables :

$$x = y$$

- ▶ Des liaison de variables à des structures :

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

(simplification : tuples, pas d'enregistrements)

Les liaisons variable - variable

- ▶ Les liaisons entre variables ne doivent pas être cycliques.
- ▶ En suivant toutes les liaisons 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, 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 ($x = \nu(x)$)
- ▶ Toutes les variables sur la *gauche* sont différentes
- ▶ Les cycles sont permis.

Exemple de mémoire

$x_1 = x_3$
 $x_2 = x_3$
 $y_1 = y_2$
 $x_3 = f(y_2, a)$
 $y_2 = g(b, z)$

Tell

- ▶ *tell* est l'opération qui ajoute une équation à la mémoire
- ▶ correspond à une instruction $X = t$
- ▶ L'opération échoue quand les équations sont incohérentes.
- ▶ On peut se ramener à trois cas :
 - ▶ ajouter $x = y$
 - ▶ ajouter $x = f(x_1 \dots x_n)$
 - ▶ ajouter $f(x_1 \dots x_n) = g(y_1 \dots y_m)$

Tell $x = y$

- ▶ Si $\nu(x) = \nu(y)$: rien à faire
- ▶ Sinon : Ajouter à la mémoire l'équation $\nu(x) = \nu(y)$ (ou $\nu(y) = \nu(x)$), et :

- ▶ Normaliser les liaisons variable - structure (maintenir l'invariant des liaisons variable - structure)
- ▶ Si la mémoire contient

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

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

Alors 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 liée à une structure : ajouter

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

- ▶ S'il existe une liaison

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

Alors 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 $x_1 = y_1, \dots, x_n = y_n$

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 $x = y$

- ▶ 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 un « tell tentative » de $x = y$
- ▶ si échec : pas d'implication
- ▶ si pas d'échec :
 - ▶ si pour toute nouvelle liaison entre variables $z_1 = z_2$ les deux variables étaient, avant le tell tentative, 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 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 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($s=t$) : $\sigma \models \exists x_1, \dots, x_n s = t$
 - ▶ Réponse neg. de ask($s=t$) : $\sigma \models \neg \exists x_1, \dots, x_n s = t$
- ▶ 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.

Synchronisation de threads

- ▶ Certaines instructions peuvent bloquer quand la mémoire ne contient pas suffisamment d'information pour conclure :
 - ▶ teste `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.

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 threads ne devrait 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 le mécanisme de contrôle (`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.

Observer par les erreurs

```
local X in
  thread X=a end
  thread X=b end
  thread X=c end
end
```

Observer en récupérant les erreurs

```
local X Y in
  try
    thread X=a end
    thread X=b end
  catch failure then case X
    of a then Y=1
    [] b then Y=2
    end
  end
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.