Terminaison du tell

- est-ce que ca termine même quand il y a des cycles dans les liaisons variable - structure ?
- Oui, car
  - toute recurrence passe par un tell $x = y$
  - si tell $x = y$ fait un appel recursif alors il a prealablement fait une nouvelle liaison variable - variable.

  Donc c'est le nombre de classes d'equivalences qui decro^t !

Ask

- Est-ce que une certaine equation (ou : ltrage par un motif) est logiquement impliquee par la memoire ?
- Trois reponses possibles : oui, incoherent, inconnu.
- Cas de ltrage : il su t de suivre les liaisons des variables dans la memoire.
- Implication d'une equation entre variables : un peu plus compliquee, due au fait qu'on a des arbres potentiellement in nis.

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 in nis :

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

Ask $x = y$

- faire une ≪ tentative de tell ≫ $x = y$
- si echec : repondre *incohérence*
- si pas d'echec :
  - si pour toute nouvelle liaison entre variables $z_1 = z_2$ les deux variables etaient, avant le tell, liees a 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 $x = y$ echoue.
- reponse : incoherence !

Exemple 2 Ask

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

- tell $x = y$ n'echoue pas, et ajoute seulement $x = y$.
- x et y liees a des structures dans la memoire de depart
- Reponse : oui !

Exemple 3 Ask

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

- tell $x = y$ n'echoue pas, et ajoute les equations $x = y$ et $y = z$
- z n'est pas liee a une structure.
- Reponse : on ne sait pas !
- Il faut attendre plus d'information sur z avant de decider.

Monotonie de la memoire

- Pendant le calcul, la memoire croît de facon *monotone* !
- Si la memoire est σ_1 a un certain moment, et plus tard σ_2, alors $\sigma_2 \models \sigma_1$: toute consequence logique de σ_1 est aussi une consequence de σ_2.
- L'echec d'un tell ou une reponse a rmative d'un ask sont dûs a des implications de la memoire :
  - Echec de tell(s=t) : $\sigma \models \neg(s = t)$
  - Reponse a . de ask(s=t) : $\sigma \models \forall x_1, \ldots x_n . s = t$
  - Reponse neg. de ask(s=t) : $\sigma \models \forall x_1, \ldots x_n . s \neq t$
- Si un tell echoue, ou un ask repond « oui » ou « incoherent » pour σ_1, alors c'est aussi le cas pour σ_2.

Des threads declaratifs

Avec plusieurs threads,

- Quand un programme sequentiel (sans threads) donne un resultat, l'ajout des thread ne change pas ce resultat.
- Un programme avec threads peut eventuellement avancer la ou le programme sequentiel bloque.
- Le calcul d'un programme avec threads peut être *incrémental*.

L'instruction `thread` en Oz

- Syntaxe : `thread <s> end`
- Semantique :
  - il y a plusieurs piles d'execution ;
  - les threads di erents utilisent la même memoire !
- Le *Browser* est toujours execute dans son propre thread.
- Toute requête donnee a l'interpreteur est executee dans son propre thread.

Synchronisation de threads

- Certaines instructions peuvent bloquer quand la memoire ne contient pas su samment d'information pour conclure :
  - test `e1 == e2` ;
  - instructions `case` et `if` ;
  - certaines procedure de la bibliotheque standard, comme `Label` ou `Arity` ;
  - l'appel d'une procedure (ou fonction) quand l'identi cateur a la position de la procedure n'est pas liee a une valeur.
- Dans ce cas le thread est suspendu.

Ramassage de miettes

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

Pas de non-determinisme observable

- L'entrelacement de l'execution des thread ne devrait pas être observable si on se restreint au modele d'execution vu au cours 2 (il est bien sûr observable quand les threads font un a chage). Ici, la seule observation permise est le contenu de la memoire nale.
- C'est une consequence du fait que le mecanisme de contrôle (if, case) utilisent ask avec une semantique logique, et de la monotonie de la memoire !

Presque pas de non-determinisme observable

- En verite : on peut observer l'entrelacement dans un modele dans lequel on peut distinguer des erreurs d'execution di erentes, ou dans lequel on peut les capturer !
- Il y a un ecart entre le modele de la semantique et la realite quand le modele de la semantique ne modelise 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 recuperant 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 o re

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

Produire un ot d'entiers

Ecrire un predicat {`Generate X`} qui lie la variable X a la liste des nombres successives a partir de 2.

```
fun {Generate}
% gives the list of numbers from 2
   local
      fun {GenAux Curr}
         Curr|{GenAux Curr+1}
      end
   in
      {GenAux 2}
   end
end
```


Listes paresseuses (lazy lists)

- Ce sont des listes qui sont construites seulement tant que demandee.
- Interessant pour la concurrence par demande (en opposition a la concurrence par o re).
- Il y a des langages qui realisent les constructeurs de donnees (listes, par exemple) de facon paresseuse : Haskell, Miranda.

Execution paresseuse en Oz

- Il s'agit d'un mecanisme (extension du langage vu jusqu'a maintenant) qui permet de creer un thread ≪ paresseux ≫ qui est execute seulement quand necessaire.
- La construction est `ByNeed` qui cree un thread, mais avec une indication a l'ordonnanceur que le thread est seulement a executer quand demande.
- Nous n'entrons pas dans les details (ca serait pour un cours sur la programmation concurrente en Oz)
- Utilise pour realiser les *listes paresseuses* en Oz.

Le generateur avec liste paresseuse

```
declare
fun {LGenerate}
   local
      fun lazy {GenAux Curr}
         Curr|{GenAux Curr+1}
      end
   in
      {GenAux 2}
   end
end
```

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

Forcer l'evaluation d'une liste paresseuse

```
proc {Touch N L}
   if N>0 then {Touch N-1 L.2} else skip end
end

{Touch 3 X}
```