

FSAB 1402: Informatique 2

Concurrence et Systèmes Multi-Agents



Peter Van Roy
Département d'Ingénierie Informatique, UCL

pvr@info.ucl.ac.be



P. Van Roy

1

Ce qu'on va voir aujourd'hui

- La programmation concurrente
 - Motivation
 - L'exécution dataflow
 - Les threads
 - Les flots de données
- Les systèmes multi-agents
 - Producteur/consommateur
 - Pipeline (comme dans Unix!)
- Quelques réflexions sur les paradigmes de programmation
- Consignes et conseils pour l'examen

P. Van Roy

2

Parties du livre pour aujourd'hui



- Concurrence, dataflow, nondéterminisme
 - 1.10, 1.11, 1.15
- Threads, dataflow
 - 4.2.1, 4.2.2, 4.2.3
- Flots et agents
 - 4.3.1, 4.3.2

Motivation pour la concurrence



Le monde est concurrent!



- Le monde réel est concurrent
 - Il est fait d'activités qui évoluent de façon indépendante
- Le monde informatique est concurrent aussi
 - Système réparti: ordinateurs liés par un réseau
 - Une activité concurrente s'appelle un ordinateur
 - Système d'exploitation d'un ordinateur
 - Une activité concurrente s'appelle un processus
 - Parfois, il y a plusieurs activités dans un processus
 - Typiquement, dans les browsers Web chaque fenêtre correspond à une activité!
 - Une activité concurrente s'appelle un thread

La programmation concurrente



- La concurrence est naturelle
 - Deux activités qui sont indépendantes sont concurrentes!
 - Lien fort entre indépendance et concurrence
 - Qu'est-ce qu'on fait si on veut faire un programme qui fait deux activités indépendantes?
 - La concurrence doit être soutenu par les langages de programmation
- Un programme concurrent
 - Plusieurs activités s'exécutent simultanément
 - Les activités peuvent communiquer et synchroniser
 - Communiquer: les informations passent d'une activité à une autre
 - Synchroniser: une activité attend une autre

Exécution dataflow



P. Van Roy

7

Exécution dataflow



- Il y a trois manières principales de programmer avec la concurrence
- La manière la plus simple s'appelle **l'exécution dataflow** ou **la concurrence déclarative**
 - C'est ce qu'on va voir aujourd'hui
- Il y a deux autres manières principales (voir INGI2131)
 - **Concurrence par envoi de messages**
 - Toujours assez simple
 - Les activités s'envoient des messages comme par courrier
 - **Concurrence par état partagé**
 - Beaucoup plus compliquée!
 - Les activités se partagent des données, elles essaient de travailler ensemble sans se marcher sur les pattes
 - Malheureusement, c'est ce que fait Java :-)

P. Van Roy

8

Une variable libre (non-initialisée)



- Une variable libre est créée en mémoire mais n'est pas encore affectée à une valeur
- Qu'est-ce qui se passe si on essaie de faire une opération avec une variable libre?

```
local X Y in
  Y=X+1
  {Browse Y}
end
```

- Qu'est-ce qui se passe?
 - **Rien!** L'exécution attend juste avant l'addition.

P. Van Roy

9

Quoi faire avec une variable non-initialisée?



- Différents langages font des choses différentes
 - **En C**, l'addition continue mais X contient une valeur "au hasard" (= le contenu de la mémoire)
 - **En Java**, l'addition continue avec 0 comme valeur pour X (si X est l'attribut d'un objet)
 - **En Prolog**, l'exécution s'arrête avec une erreur
 - **En Java**, il y a une détection d'erreur par le compilateur (si X est une variable locale)
 - **En Oz**, l'exécution attend juste avant l'addition et peut continuer quand X est lié (exécution dataflow)
 - **Dans la programmation par contraintes**, l'addition "Y=X+1" est ajoutée à l'ensemble de contraintes et l'exécution continue! (voir cours "Programmation par Contraintes")

P. Van Roy

10



Faire continuer l'exécution

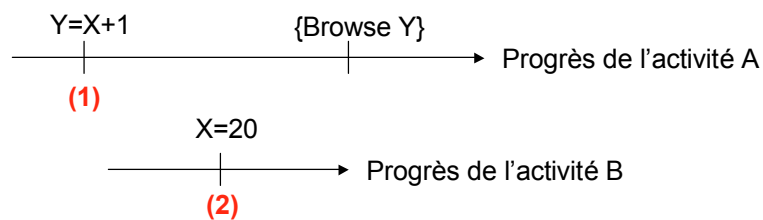
- L'instruction qui attend:
`declare X`
`local Y in`
 `Y=X+1`
 `{Browse Y}`
`end`
- Si quelqu'un d'autre pourrait lier X, alors l'exécution pourrait continuer!
- Mais qui peut le faire?
- Réponse: une autre activité concurrente!
- Si une autre activité fait:
 `X=20`
- Alors l'addition continuera et on affichera 21!
- Cela s'appelle de **l'exécution dataflow**

P. Van Roy

11



L'exécution dataflow



- L'activité A attend sagement au point **(1)** juste avant l'addition
- Quand l'activité B fait `X=20` au point **(2)**, alors l'activité A peut continuer
- Si l'activité B fait `X=20` **avant que** l'activité A n'arrive au point **(1)**, alors l'activité A **n'attendra pas du tout**

P. Van Roy

12

Threads



Threads



- Une “activité” est **une séquence d’instructions en exécution**
 - On appelle cela un **thread**
- Chaque thread est **indépendant** des autres
 - Entre deux threads il n’y a pas d’ordre
 - Le système garantit que chaque thread reçoit une partie équitable de la capacité de calcul du processeur
- Deux threads peuvent communiquer s’ils partagent des variables
 - Par exemple, la variable qui correspond à X dans l’exemple qu’on vient de voir

La sémantique des threads (1)



- Chaque thread correspond à une pile sémantique
- L'instruction **thread** $\langle s_1 \rangle$ **end** crée une nouvelle pile sémantique
- Tous les threads se partagent la même mémoire

La sémantique des threads (2)



Une pile sémantique avec une instruction pour créer un thread

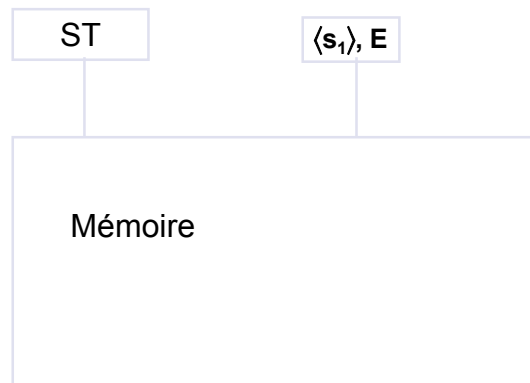
thread $\langle s_1 \rangle$ end , E
ST

Mémoire

La sémantique des threads (3)



Deux piles!



P. Van Roy

17

La création des threads



- En Oz, la création d'un thread est simple
- On peut exécuter n'importe quelle instruction $\langle s \rangle$ dans un nouveau thread:
`thread $\langle s \rangle$ end`
- Par exemple:
`declare X`
`thread {Browse X+1} end`
`thread X=1 end`
- Qu'est-ce que fait ce fragment de programme?
 - Il y a plusieurs exécutions possibles, mais elles arrivent toutes au même résultat: tôt ou tard, on affichera 2!

P. Van Roy

18



Un petit programme (1)

- Voici un petit programme avec des threads:
declare X0 X1 X2 X3
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Le Browser affiche [X0 X1 X2 X3]
 - Les variables ne sont pas encore affectées
 - Le Browser utilise aussi le dataflow: quand une variable est affectée, l'affichage est mis à jour



Un petit programme (2)

- Voici un petit programme avec des threads:
declare X0 X1 X2 X3
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Les deux threads attendent
 - X1=1+X0 attend (X0 n'est pas affectée)
 - X3=X1+X2 attend (X1 et X2 ne sont pas affectées)

Un petit programme (3)



- Voici un petit programme avec des threads:
declare X0 X1 X2 X3
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Nous faisons une affectation
 - Faites X0=4

Un petit programme (4)



- Voici un petit programme avec des threads:
declare X0 X1 X2 X3
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Nous faisons une affectation
 - Faites X0=4
 - Le premier thread peut exécuter, il fait X1=5
 - Le Browser montre [4 5 X2 X3]

Un petit programme (5)



- Voici un petit programme avec des threads:

```
declare X0 X1 X2 X3  
thread X1=1+X0 end % thread terminé  
thread X3=X1+X2 end  
{Browse [X0 X1 X2 X3]}
```
- Le second thread attend toujours
 - Parce que X2 n'est toujours pas affecté

Un petit programme (6)



- Voici un petit programme avec des threads:

```
declare X0 X1 X2 X3  
thread X1=1+X0 end % thread terminé  
thread X3=X1+X2 end  
{Browse [X0 X1 X2 X3]}
```
- Nous faisons une autre affectation
 - Faites X2=7
 - Le second thread peut exécuter, il fait X3=12
 - Le Browser montre [4 5 7 12]

Le Browser est concurrent



- Le Browser exécute avec ses propres threads
- Pour chaque variable libre, il y a un thread dans le Browser qui attend sur cette variable
 - Quand la variable est affectée, l'affichage est mis à jour
- Attention: cela ne marche pas avec les cellules!
 - Le Browser ne regarde pas le contenu d'une cellule

Ordre total, ordre partiel et nondéterminisme



Exécution "simultanée"



- Avec la concurrence, on peut avoir plusieurs activités qui s'exécutent "en même temps"
- Il faut imaginer que tous les threads exécutent vraiment en parallèle, chacun avec son propre processeur mais partageant la même mémoire
- Lire et écrire des variables et des cellules peuvent être fait simultanément dans différents threads
 - Lire la même variable peut être fait de façon simultanée
 - Ecrire la même variable est fait de façon séquentielle

Ordre des états d'exécution

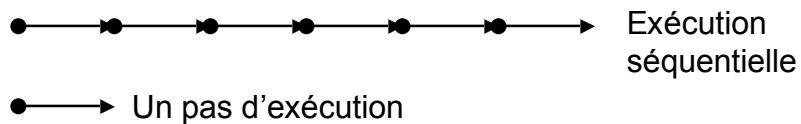


- Dans un programme séquentiel, tous les états d'exécution sont dans un **ordre total**
- Dans un programme concurrent, tous les états d'exécution du même thread sont dans un ordre total
 - Les états d'exécution du programme complet (avec plusieurs threads) sont alors dans un **ordre partiel**
- Ordre total = il y a un ordre défini entre chaque paire d'états
- Ordre partiel = il y a un ordre défini entre une partie des paires d'états (certaines paires n'ont pas d'ordre entre eux)

Ordre total dans un programme séquentiel



- Dans un programme séquentiel, tous les états d'exécution sont dans un **ordre total**



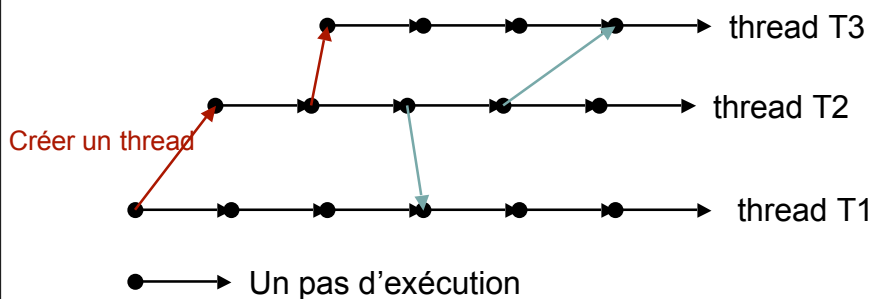
P. Van Roy

29

Ordre partiel dans un programme concurrent



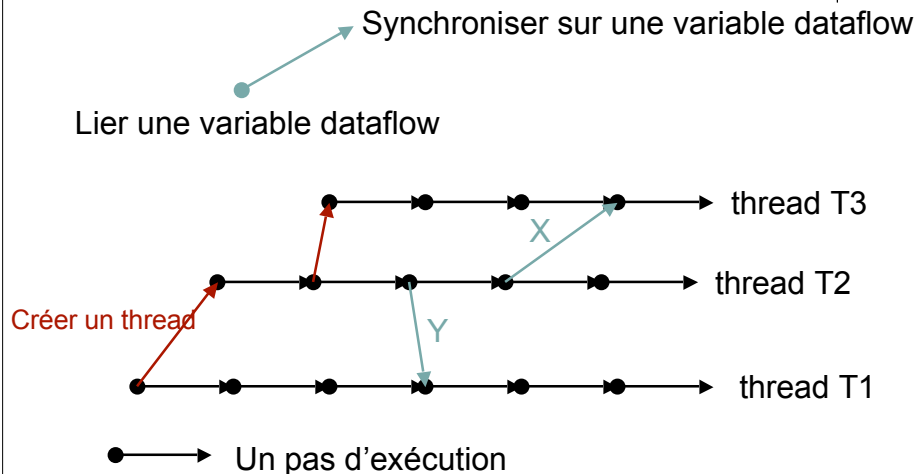
- Dans un programme concurrent, tous les états d'exécution du même thread sont dans un **ordre total**
- Les états d'exécution du programme complet (avec plusieurs threads) sont dans un **ordre partiel**



P. Van Roy

30

Ordre partiel dans un programme concurrent



P. Van Roy

31

Nondéterminisme (1)



- Qu'est-ce que fait le programme suivant?

```
declare X  
thread X=1 end  
thread X=2 end
```
- L'ordre d'exécution des deux threads n'est pas déterminé
 - X sera lié à 1 ou à 2, on ne sait pas à quelle valeur
 - L'autre thread **aura une erreur (une exception sera levée)**
 - On ne peut pas affecter une variable deux fois
- Cette incertitude s'appelle le **nondéterminisme**

P. Van Roy

32

Nondéterminisme (2)



- Qu'est-ce que fait le programme suivant?

```
declare X={NewCell 0}  
thread X:=1 end  
thread X:=2 end
```
- L'ordre d'exécution des deux threads n'est pas déterminé
 - La cellule X sera affectée à une valeur, puis à l'autre
 - Quand les deux threads sont terminés, X aura le contenu 1 ou 2, on ne sait pas quelle valeur
- Cette incertitude s'appelle le nondéterminisme

Nondéterminisme (3)



- En général, il faut **éviter le nondéterminisme**
 - Ce n'est pas facile en général
 - C'est assez compliqué si on mélange les threads et les cellules (concurrency par état partagé)
 - Malheureusement, beaucoup de langages utilisent la concurrence par état partagé :-)
- Le modèle déclaratif a un avantage
 - **Le modèle déclaratif n'a pas de nondéterminisme** (sauf s'il y a une erreur comme dans l'exemple précédent)

L'ordonnancement des threads (1)



- Si le nombre de threads est plus grand que le nombre de processeurs (souvent le cas), alors les threads se partagent les processeurs
 - Chaque thread est exécuté pendant des courtes périodes qui s'appellent des **tranches de temps** ("time slices")
- La choix de quel thread qui va être exécuté a chaque moment et pour combien de temps est fait par une partie du système qui s'appelle **l'ordonnanceur** (*scheduler*)
- Un thread est **exécutable** (*runnable*) si l'instruction au sommet de sa pile n'attend pas sur une variable dataflow. Sinon, le thread est **suspendu** (*suspended*) ou **bloqué sur une variable** (*blocked on a variable*)

P. Van Roy

35

L'ordonnancement des threads (2)



- Un ordonnanceur est **équitable** (*fair*) si chaque thread exécutable sera tôt ou tard exécuté
 - En général, on donne des garanties sur le pourcentage du processeur qui est donné à tous les threads de la même **priorité**
- Si l'ordonnancement est équitable, on peut raisonner sur l'exécution des programmes
- Sinon, un programme parfaitement bien écrit peut ne pas marcher

P. Van Roy

36

Flots et agents



P. Van Roy

37

Flot (stream)



- Un **flot** (*stream*) est une liste dont l'extrémité est une variable libre
 - $S = a|b|c|d|S2$
 - Un flot peut être étendu avec des nouveaux éléments indéfiniment
 - On peut fermer le flot en terminant la liste avec nil
- Un flot peut servir comme un **canal de communication** entre deux threads
 - Le premier thread ajoute des éléments au flot
 - Le second thread lit le flot

P. Van Roy

38



Exemple d'un flot

- Voici un programme qui affiche tous les éléments d'un flot:

```
proc {Disp S}  
  case S of X|S2 then {Browse X} {Disp S2} end  
end  
declare S  
thread {Disp S} end
```

- On ajoute des éléments au flot:
 declare S2 in S=a|b|c|S2
 declare S3 in S2=d|e|f|S3
- Essayez par vous-même!

P. Van Roy

39



Producteur/consommateur (1)

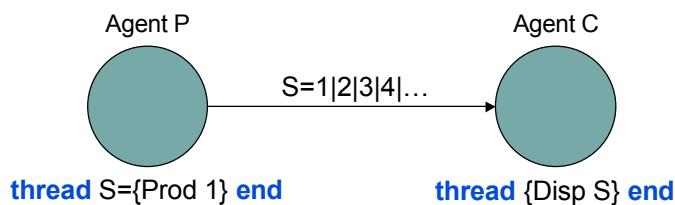
- Un **producteur** génère un flot de données
 fun {Prod N} {Delay 1000} N|{Prod N+1} end
 - Le {Delay 1000} ralentit l'exécution pour qu'on puisse voir!
- Un **consommateur** lit le flot et fait quelque chose (comme la procédure Disp)
- Un programme producteur/consommateur:

```
declare S  
thread S={Prod 1} end  
thread {Disp S} end
```

P. Van Roy

40

Producteur/consommateur (2)



- Chaque cercle est une **activité concurrente avec un (ou plusieurs) canaux de communication**
 - On appelle ça aussi un **agent**
- Les agents communiquent par le flot S
 - Le premier thread crée le flot, le second le lit

P. Van Roy

41

Pipeline (1)



- On peut ajouter d'autres agents entre P et C
- Voici un **transformateur** qui modifie le flot:

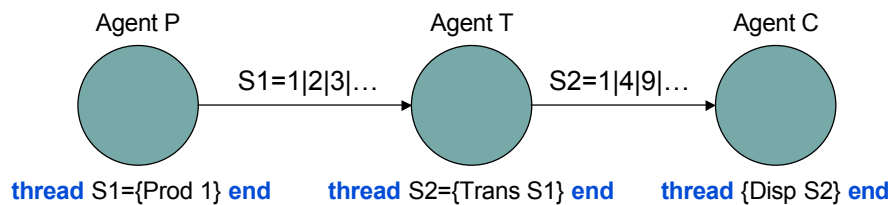
```
fun {Trans S}
  case S of X|S2 then X*X|{Trans S2} end
end
```
- Voici un programme avec trois agents:

```
declare S1 S2
thread S1={Prod 1} end
thread S2={Trans S1} end
thread {Disp S2} end
```

P. Van Roy

42

Pipeline (2)



- Nous avons maintenant créé trois agents
 - Le producteur (agent P) crée le flot S1
 - Le transformateur (agent T) lit S1 et crée S2
 - Le consommateur (agent C) lit S2
- La technique du pipeline est très utile!
 - Par exemple, il est **omniprésent en Unix**

P. Van Roy

43

Quelques réflexions sur les paradigmes de programmation

P. Van Roy

44

Les paradigmes de FSAB1402

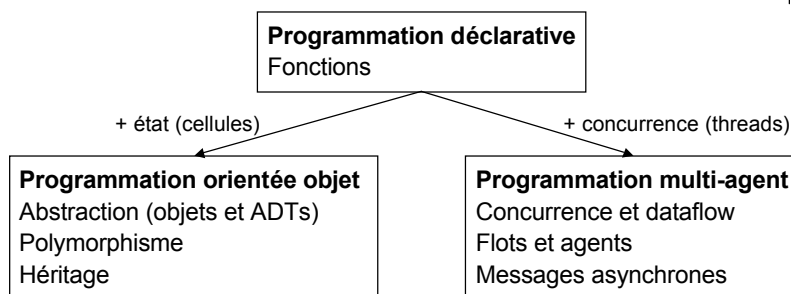


- Dans ce cours nous avons vu quelques des concepts les plus importants dans la programmation
- Nous avons aussi vu quelques paradigmes de programmation
 - Programmation déclarative (programmation fonctionnelle)
 - Programmation avec état
 - Programmation orientée objet
 - Programmation concurrente avec dataflow
 - Programmation multi-agent
- Il y a beaucoup d'autres paradigmes intéressants!
 - Programmation concurrente par envoi de messages
 - Programmation concurrente par état partagé
 - Programmation par composants logiciels
 - Programmation logique
 - Programmation par contraintes
 - ...

P. Van Roy

45

Les deux “mondes” du cours



- Dans ce cours, nous avons vu deux “mondes” très différents, chacun avec sa manière de penser
 - Pour voir comment les deux mondes se retrouvent il faut suivre le cours INGI2131!

P. Van Roy

46

Paradigmes de programmation



Programmation déclarative

Programmation fonctionnelle stricte, Scheme, ML
Programmation logique déterministe

+ concurrence
+ synchronisation selon besoin
Concurrence dataflow (déclarative)
Prog. fonctionnelle paresseuse, Haskell

+ choix nondéterministe
Programmation logique concurrente

+ traitement d'exceptions
+ état explicite

Programmation orientée objet (OO), Java, C++, Smalltalk

+ recherche
Prog. logique classique, Prolog

- Ce schéma donne un résumé des différents paradigmes avec les relations entre eux
- Chaque paradigme a ses avantages et désavantages et un domaine où il est le meilleur

Programmation OO concurrente
(envoi de messages, Erlang, E)
(état partagé, Java)

+ espaces de calcul
Programmation par contraintes

P. Van Roy

47

La coexistence des paradigmes



- Chaque paradigme a sa place
 - Avec plus de concepts on peut exprimer plus, mais le raisonnement devient plus compliqué
 - Avec moins de concepts on peut satisfaire des conditions d'utilisation plus stricte
- Les différents paradigmes ne sont pas meilleurs ou pires, mais simplement différents
 - Dans vos programmes, je vous conseille de bien réfléchir et de choisir le paradigme approprié
- Maintenant, je vous conseille de **relire le début du premier cours!**
 - Pourquoi on a organisé le cours autour des concepts

P. Van Roy

48

Consignes et conseils pour l'examen



L'examen



- L'examen sera de 3h, à livre fermé
- Il y aura une division égale entre théorie et pratique
 - Attention à la **précision** pour la théorie (voir le livre pour les définitions!)
 - Attention à la **syntaxe** pour la pratique!
- Il y aura certainement une question sur la **sémantique**
 - Un exercice où vous devez faire l'exécution d'un programme
 - Attention à ne pas sombrer dans les détails!
 - Sur le site Web il y a **une ancienne question d'examen avec solution** (faite par Damien Saucez et Anh Tuan Tang Mac)
- La matière est tout ce qui a été vu dans les cours magistraux et les séances pratiques
 - Une définition précise des concepts se trouve dans le livre du cours
- Les notes ne seront pas sur une courbe
 - J'espère pouvoir vous donner tous de bonnes notes

Le formulaire pour la syntaxe



- Vous pouvez venir à l'examen avec un formulaire pour la syntaxe
 - Le but est que vous ne faites pas d'erreurs de syntaxe dans vos réponses aux questions
 - Le formulaire doit être **écrit à la main** et **rendu avec l'examen**, mais il ne sera pas coté
- **Une page**, contenant uniquement des fragments de programme et des règles de grammaire
 - Fragments en Oz et en Java
 - Aucun mot écrit en langue naturelle
 - Aucune équation mathématique

Résumé





Résumé

- Concurrency
 - Activities that evolve independently
 - Execution dataflow with free variables
 - Nondeterminism
- Thread
 - To model a concurrent activity in the language
 - A sequence of instructions in execution
 - In semantics, it's a semantic stack
- Flot (stream)
 - A list whose end is a free variable
 - A communication channel between agents
- Agent
 - An activity concurrent with one or more communication channels
 - Multi-agent program: producer/consumer, pipeline
- Programming paradigms
 - Each paradigm has its place