LINGI1131 : Computer Language Concepts

Ce cours est une introduction aux concepts de la programmation avec un accent mis sur la programmation concurrente.

Ce cours se base sur le livre "Concepts, Techniques, and Models of the Computer Programming" et couvre les sections : * Section 2.1-2.6 : Declarative Model and Kernel language * Section 3.2-3.4 : Declarative programming * Section 4.1-4.6, 4.8 : Declarative Concurrency * Section 5.1-5.6 : Message passing concurrency and multi-agent systems * Section 6.1-6.3 : State * Section 7.2, 7.7, 7.8 : Classes and Active Objects * Section 8.1-8.6 : Shared-State Concurrency (Locks, Monitors, Transactions)

1. Declarative Model and Kernel language

1.1 Kernel language VS practical language

The **practical language** is an easy way to code. It provides usefull abstractions for the programmer and it can be extended with linguistic abstractions (and syntaxic sugar). The *practical language* can always be translated into **kernel Language**. **Kernel Language** is a minimal set of intuitive concepts that are easy for the programmer to understand and that has a formal semantic (not always easy to work with). Exmple : This exemple in practical language :

```
fun {Sqr X} X*X end
B={Sqr {Sqr A}}
```

can be translated into this in Kernel language

```
proc {Sqr X Y}
  {'*' X X Y}
end

Local T in
  {Sqr A T}
  {Sqr T B}
end
```

1.2 Single assignement

A variable in the single assignement store is unbound or a value. if it's a value, this value can only be bound one time

1.3 Kernel language

::= | X = f(l1:Y1 ... ln:Yn) | X = | X = | X = | {NewName X} | X = Y | local X1 ... Xn in S1 end | proc {X Y1 ... Yn} S1 end | {X Y1 ... Yn} | {NewCell Y X} | Y=@X | X:=Y | {Exchange X Y Z} | if B then S1 else S2 end | thread S1 end | try S1 catch X then S2 end | raise X end

1.4 Partial values

ex:

```
declare A B C L in
   L=[ABC]
   {Browse L} //display [_ _ _]
```

1.5 Dataflow principle

Concept of dataflow : the availibility drives the execution `declare Y=X+1 // X is unbound and the program will wait for X` if we wait, someone else need to bound this variable. so we introduce the concept of thread (State of the execution). They can be multiple threads inside the program. The execution continue when the data becomes available

2. Declarative programming

La programmation déclarative est un type de programmation qui ne se programme pas en instructions mais en indiquant au programme à quel résultat celui-ci doit arriver. Ce qui revient à dire à un chauffeur de taxi où vous voulez aller et non lui indiquer rue par rue toutes les indications pour rentrer jusque chez vous. Le programme est ainsi plus simple. Cependant, dans certains cas, il arrive que le programme fasse des choix surprenants. Il est aussi important de noté que la programmation déclarative est une programmation composée (= elle se compose de différentes parties pour former des parties plus grandes)

2.1 Réccursion

Le language OZ est un language qui utilise la réccusrion comme fondement. (ex: listes, trees, ...). Il est donc toujours important de penser en réccusrion

2.2 Pattern Matching

Quand on programme, il arrive que l'on ai besoin d'accéder à un élément d'une structure. Dans l'exemple suivant, il s'agit d'une liste.

```
case Xs
  of nil then <premier code>
  [] X|Xr then <second code>
end
```

On analyse la liste Xs, si elle est nulle, c'est le premier code qui sera lancé. dans le second cas, le premier élément de la liste pourra facilement être accédé via le X et le suivant Xr dans le second code. Ceci s'appelle le pattern Matching, il est très utile en programmation en OZ

2.3 Time and space efficiency

L'efficacité temporelle est spatiale peut être calculée sur base du kernel language, elle décrit l'évolution de la consomation de mémoire ou le temps d'exécution si le problème de taille n venait à changer. (ex: O(1) temps constant, O(nˆ{2}) évolution quadratique) Il existe plusieurs notations pour décrire l'efficience : theta(borne minimum), Grand O (temps moyen), Omega (temps maximal). qui représentent généralement respectivement les cas, optimal, moyen et le pire cas.

2.4 Higher-order Programming

La programmation de haut degré est une programmation qui accepte une fonction comme argument d'une autre fonction.

2.5 Genericity

La généricité est une des conséquences du High-order programming, elle permet l'utilisation de différentes fonctions placés en argument comme input de celle-ci (Ex: La fonction map prends une fonction et une liste en argument. la fonction map applique la fonction en argument à chaque élément de la liste)

2.6 Embedding

Les procédures peuvent être mises dans une structure de données, ce qui leur donne plusieurs usages : * Explicit lazy evaluation (=exécution différée) : construit une petite partie de la structure et le reste en fonction des besoins sur le translated * Modules : un Record qui groupe ensemble un set d'opérations ensemble * Software Components : Un ensemble de modules en input et output d'un nouveau modules. On importe ainsi seulement ce dont on a besoin de chaque module

2.7 Data techniques (pattern matching)

List

```
case L
  of nil then ...
  []X|Xr then ...
  else ...
end
```

Trees

```
case T
  of nil then ...
  [] ??????
end
```

3. Declarative Concurrency (Sections: 4.1-4.6 + 4.8)

The concurreny extend the declarative model. Putting a program in a thread does not change the final result but the result of an program can be calculated incrementaly. A thread is a piece of the program tha is independant and can be computed separately. With threads we have multiple semantics stacks ("threads") but only one Single-assignement Store.

Interleaving : chaque thread est exécuté mais il peut n'être exécuté que d'un ou deux instructions, puis exécuter quelques instructions d'un autre thread.

3.0.1 Concurrency and thread

be carefull whith this because you create a *activity* inside the system (thread) each time you unbound a varaible **Thread** : sequence in execution

`thread < S > end`

create a thread.

We can have multiple threads simultaniously(+- because only one cpu) the execution uses 'interleaving semantics' : one cpu, multiple threads that are sharing this calculation power at the same time. interleaving : the system will perform only one sequence of steps in the execution. concurrency does not apply parallelism. so no need of multiple cores. parallelism : small number (4-8 cores so 4-8 processes at the same time) concurrency : hundreds at the same time

each step is the thread is choose by the scheduller (ordonnanceur).

3.0.2 the execution tree

it shows all possibile exécutions (all possible ways in the execution)

```
declare A B C in
thread A=1 end
thread B=2 end
thread C=A+B end
```

3.1 Non-Determinism

Une exécution est appelée non déterministe si il y a une étape dans l'exécution ou l'on à la possibilité de choisir quoi faire ensuite (ex: 2 threads en même temps, lequel doit avancer?). Le non-déterminisme apparaît dès l'apparition d'activités concurrentes. Ce choix sera effectué par l'ordonnanceur (Scheduller)

3.2 Browse et threads

Le code suivant :

```
thread {Browse 111} end
thread {Browse 222} end
```

peut donner des résultats comme 112122 ou des erreurs car tous les caractères ne sont pas imprimés en même temps.

Exemple programmation avec threads (Fibonnacci avec thread)

```
fun {Fib X}
  if X =<2 then 1
  else thread {Fib X-1} end  + {Fib X-2} end
end
```

3.3 Stream

En programmation concurrente, la technique la plus utile pour communiquer sont les stream. **un Stream** : une liste potentiellement non finie de messages. Envoyer un message au stream allonge celui-ci d'un élément. Le stream est une sort d'objet actif. Il ne nécéssite pas de mécanisme de blocage ou d'exclusion mutuelle tant que chaque variable est reliée à un seul thread

3.3.1 Basic Producer/Consumer

Avec des threads on peut créer un programme de producteur/consommateur asynchrone

```
declare
fun {Generate N Limit}
   if N<Limit then
      N|{Generate N+1 Limit}
   else
      nil
   end
end
fun {Sum Xs A}
   case Xs
   of X|Xr then {Sum Xr A+X}
   [] nil then A
   end
end

local Xs S in
   thread Xs = {Generate 0 15000000000} end
   thread S = {Sum Xs 0} end
   {Browse S}
end
```

Dans le consomateur, le case agit comme un bloquage si il n'y a pas de données à calculer

High-Order version

```
declare
fun {Generate N Limit}
   if N<Limit then
      N|{Generate N+1 Limit}
   else
      nil
   end
end
fun {Map Xs F A}
   case Xs
   of X|Xr then {Map Xr F {F X A}}
   [] nil then A
   end
end

local Xs S in
   thread Xs = {Generate 0 150000} end
   thread S = {Map Xs fun {$ X A} X+A end 0} end
   {Browse S}
end
```

Multiple readers

Buffer bounded

```
declare
proc {DGenerate N Xs}
   case Xs of X|Xr then
      X=N
      {DGenerate N+1 Xr}
   end
end
declare
fun {Dsum ?Xs A Limit}
   if Limit>0 then
      X|Xr = Xs
   in
      {Dsum Xr A+X Limit-1}
   else
      A
   end
end
declare
proc {Buffer N ?Xs Ys}
   fun {Startup N ?Xs}
      if N==0 then Xs
      else
     Xr in Xs=_|Xr {Startup N-1 Xr} end
   end
   proc {AskLoop Ys ?Xs ?End}
      case Ys of Y|Yr then Xr End2 in
     Xs=Y|Xr
     End = _|End2
     {AskLoop Yr Xr End2}
      end
   end
   End = {Startup N Xs}
in
   {AskLoop Ys Xs End}
end
local Xs Ys S in
   thread {DGenerate 0 Xs} end
   thread {Buffer 4 Xs Ys} end
   thread S = {Dsum Ys 0 150000} end
   {Browse Xs}{Browse Ys}
   {Browse S}
end
```

ping-pong

a thread need to display ping pong, in alternance with thread1 saying ping and the other pong. how should threads do this ? threads should communicate

```
declare
fun {Ping S}
  case S of ok|T then
  {browse ping}
  ok|{Ping T}
  end
end
fun {Pong S}
  case S of ok|T then
  {browse Pong}
  ok|{Pong T}
  end
end
declare S1 S2 in
thread S2={Ping ok|S1} end //ok| is used to initialise the ping pong
thread S1={Pong S2} end
```

S5 Lazy Programming

S5.1 Lazy suspension

Hamming problem

La théorie des producteurs-consommateurs s'articule autour d'un thread qui génère les données à calculer et un consommateur qui les calcule. On ne connait pas à l'avance la vitesse du producteur et du consommateur. Cette méthode de résomution est très utile lors de problèmes ou l'on ne connait pas en avance la taille des données à calculer. * Générer un **stream** * Commence à générer un calcul dans un thread (Producteur) * Commence à calculer (Consommateur)

La programation fénéante (lazy programming), est une programation ou l'on ne calcule un élément uniquement lorsque celui-ci est nécéssaire. On évite ainsi de calculer des élément inutilement.

```
H=1|{Merge {Times H 2} {Merge {Times H 3} {Times H 5}} }

declare

fun lazy{Times H N}
  case H of E|H2 then E*N|{Times H2 N} end
end
```

```
declare
fun lazy{Merge S1 S2}
  case S1|S2
    of (E1|T1)|(E1|T1) then
      if E1<E2 then E1|{Merge T1 S2}
      elseif E1>E2 then E2|{Merge S1 T2}
      else  % (E1==E2)
        E1|{Merge T1 T2}
      end
  end
end

declare
proc{touch H N}
  if N==0 then skip
  else {touch H2 N-1} end
end
```

La programmation fénéante est la meilleure solution pour résoudre des problèmes ou la taille n'est pas connue à l'avance. On évite donc les calculs inutiles (ex fibonnacci) et on ne calcule le resultat seulement quand celui-ci est nécessaire. Dans le cas de Quicksort, la méthode fénéante est même plus rapide que son homologue.

```
declare
proc {Partition L X L1 L2}
  case L of Y|M then
    if Y<X then M1 in
      L1 = Y|M1 {Partition M X M1 L2}
    else M2 in
      L2 = Y|M2 {Partition M X L1 M2}
    end
  []nil then
    L1 = nil
    L2 = nil
  end
end

declare
fun {Append L1 L2}
  case L1 of X|M1 then X|{Append M1 L2}
  []nil then L2 end
end

declare
fun {Append L1 L2}
  case L1 of X|M1 then X|{Append M1 L2}
```

```
  []nil then L2 end
end

declare
fun  {Quicksort L}
  case L of X|M then
    {Partition M X L1 L2}
    S1 = {Quicksort L1}
    S2 = {Quicksort X|L2}
  []nil then nil
  end
end

declare L  L1 L2 in
L = [1 2 3 4 5 ...]
{Partition L 4 L1 L2}
{Browse L1}
{Browse L2}

{Browse {Quicksort [....]}}

% -----Lazy version -------

declare
fun lazy {LAppend L1 L2}
    case L1 of X|M1 then X|{LAppend M1 L2}
    []nil then L2 end
end

declare
fun lazy {LQuicksort L}
    case L of X|M then L1 L2 S1 S2 in
      {Partition M X L1 L2}
      S1 = {LQuicksort L1}
      S2 = {LQuicksort L2}
      {LAppend S1 X|S2}
    []nil then nil
    end
end
```

On passe donc de O(nlog(n)) à O(n)

S7

Message passing concurrency (ch5)

Ports and port Objects

Ports

Port Objects

Port object = port + thread + function port object has a internal State F : msg x state -> State I : initial State

```
declare
fun {NewPortObject F I}
  proc {Loop S State}
    case S of M|T then
      {Loop T {F M State}}
    end
  end
  P
in
  thread S in P = {NewPort S} {Loop S I}end
  P
end
//counter Objects

declare
Ctr = {NewPortObject
  fun{$ Msg State}
    case Msg
    of inc(X) then State+X
    [] dec(X) then State-X
    [] get(X) then X = State State
    end
  end
  0}
  ‘‘‘
###Playball exemple
exemple of non deterministic execution
here it comes from random numbers but it can comes from an other source
```

declare fun{Player Others} {NewPortObject fun {$ Msg State} case Msg of ball then Ran in Ran = ({OS.rand} mod {Width Others})+1 {Send Others.Ran ball} State+1 [] get(X) then X= State State end end 0} end

//init the game declare P1 P2 P3 in P1 = {Player others(P2 P3)} P2 = {Player others(P1 P3)} P3 = {Player others(P1 P2)}

local X in {Send P1 get(X)} {Browse X} end local X in {Send P2 get(X)} {Browse X} end local X in {Send P3 get(X)} {Browse X} end {Send P1 ball}

```
###Message Protocols
Rules for agents talking to each others
####RMI
simple port with no internal State
Will run forever ( but not bad because accept always new messages )
```

fun {NexPortObject2 Proc} P in thread S in P ={NewPort S} for M in S do {Proc M} end end P end

```
Rmi :
```

declare proc{ServerProc Msg} case Msg of calc(X Y) then Y = X*X+2.0*X+234.3 end end Server = {NewPortObject2 ServerProc}

local Y in {Send Server calc(1.0 Y)} {Browse Y} end

// Client declare proc{ClientProc Msg} case Msg of work(Y) then Y1 Y2 in {Send Server calc(1.0 Y1)} {wait Y1} {Send Server calc{2.0 Y2}} {wait Y2} Y=Y1+Y2 end end Client = {NewPortObject2 ClientProc}

// DO the work declare Y in {Send Client work(Y)} {Browse Y}

```
#### Async Rmi
twice as fast
better in online and real world application
```

// Client Async declare proc{ClientProc Msg} case Msg of work(Y) then Y1 Y2 in {Send Server calc(1.0 Y1)} {Send Server calc{2.0 Y2}} {wait Y1} % MOVE THIS <———————— {wait Y2} Y=Y1+Y2 end end Client2 = {NewPortObject2 ClientProc}

// DO the work declare Y in {Send Client2 work(Y)} {Browse Y}

```
#### RMI with Callback (thread)
good only with cheap threads (ex: ERLANG)
```

declare proc{ServerProc Msg} case Msg of calc(X Y Client) then D in {Send Client delta(D)} Y = X*X+2.0*D*X+D*D+23.0 end end Server = {NewPortObject2 ServerProc}

declare proc{ClientProc Msg} case Msg of work(Z) then Y in {Send Server calc(10.0 Y Client)} thread Z = Y+10 end <—- avoid DeaDlock1 by threading [] delta(D) then

```
D = 0.1
```

end end Client = {NewPortObject2 ClientProc}

declare Z in {Send Client work(Z)} {Browse Z}

```
**DeaDlock** : Circular wait
client wait Server and Server wait Client

restart server -> if he is deadlocked

#### RMI with Callback (Record Continuation)
If threads are expensives (ex: JAVA)

Split method  into pieces that don't wait

cont(Y Z) is a record (data structure)
Need to pas y to server which is passed back
```

declare proc{ServerProc Msg} case Msg of calc(X Client Y) then {Send Client delta(D)} <–get info from Server Y = X*X+2.0*D*X+D*D+23.0 {Send Client cont(Y Z)} <– gives result to client end end Server = {NewPortObject2 Server-Proc}

declare proc{ClientProc Msg} case Msg of work(Z) then Y in {Send Server calc(10.0 Client Y)} [] cont(Y Z) then Z = Y+10.0 [] delta(D) then D = 0.1 end end Client = {NewPortObject2 ClientProc}

declare Z in {Send Client work(Z)} {Browse Z}

```
#### RMI with Callback (procedure Continuation)
Use high-order programming
```

declare proc{ServerProc Msg} case Msg of calc(X Client Y contProc) then Z D in {Send Client delta(D)} Y = X*X+2.0*D*X+D*D+23.0 {Send Client cont(Y Z)} end end Server = {NewPortObject2 ServerProc}

declare proc{ClientProc Msg} case Msg of work(Y) then Z in {Send Server calc(10.0 Client Z proc{$} Y = Z+10.0 end)} [] p(ContProc) then {ContProc} [] delta(D) then D = 0.1 end end Client = {NewPortObject2 ClientProc}

declare Z in {Send Client work(Z)} {Browse Z} “‘

Async RMI with Callback (Do it yourself)

Active objects and passive objects (Classes)

Port Objects -> tran?? sem??

F : Msg x State -> state #### Active Objects -> Classes c : i = i+1 (read implicit) OZ : i:=@i + 1 Pascal modua and @ do the read of the value “‘ declare class Counter attr i meth init(X) i:=X end meth inc(X) i:=@i + X end meth get(X) X=@i end end

declare Ctr = {Newcounter init(0)}

local X in {Ctr get(X)} {Browse X} end {Ctr inc(10)} “‘

Passive objets

{ctr inc(0)} i :=@i +1 executed in the caller thread feeds the thread like a parasite