UCLouvain

Compilation de code LINFO1104

compilation du 18 juin 2023

Thomas Debelle

Juin 2023

Table des matières

Préface

Bonjour à toi !

Cette synthèse recueille toutes les informations importantes données au cours, pendant les séances de tp et est améliorée grâce au note du Syllabus. Elle ne remplace pas le cours donc écoutez bien les conseils et potentielles astuces que les professeurs peuvent vous donner. Notre synthèse est plus une aide qui, on l'espère, vous sera à toutes et tous utile.

Elle a été réalisée par toutes les personnes que tu vois mentionnées. Si jamais cette synthèse a une faute, manque de précision, typo ou n'est pas à jour par rapport à la matière actuelle ou bien que tu veux simplement contribuer en y apportant tes connaissances ? Rien de plus simple ! Améliore la en te rendant ici où tu trouveras toutes les infos pour mettre ce document à jour. (*en plus tu auras ton nom en gros ici et sur la page du github*)

Nous espérons que cette synthèse te sera utile d'une quelconque manière ! Bonne lecture et bonne étude.

Chapitre 1

Code

1.1 Les différents paradigmes

1.1.1 Scoping

```
local
  X
in
  X = 42 {Browse X}    % Imprime 42
  local X in
    X = 11 {Browse X} % Imprime 11
  end
  {Browse X}           % Imprime 42
end
```

1.2 Programmation symbolique

1.2.1 Pattern matching

```
fun {Sum L}
  case L
  of nil then 0
  [] H|T then H+{Sum T}
  end
end
```

1.2.2 Tuples et Records

Tuples

```
X = state(1 b 2)
{Browse {Label X}}
{Browse {Width X}}
```

1.2.3 Arbre avec Tuples

```
declare
Y = left(1 2) Z = right(3 4)
X = mid(Y Z)
```

1.2.4 Similitude Tuples et listes

```
List1 = [1 2 3]
List2 = (1:1 2:(1:2 2:(1:3 2:nil)))
List1 == List2 //Vrai
```

1.2.5 Sémantique formelle

```
local P Q in
  {Browse 'do something'}
  proc {Q}
    {P}
  end
  {Browse 'another something'}
end
```

1.2.6 Sémantique opérationnelle

Langage Kernel complet

```
<s> ::= skip
  | < s >1< s >2
  | local <x> in <s> end
  | < x >1=< x >2
  | <x>=<v>
  | if <x> then < s >1 else < s >2 end
  | {<x> < y >1,...,< y >n}
  | case <x> of <p> then < s >1 else < s >2 end

<v> ::= <number> | <procedure> | <record>
<number> ::= <int> | <float>
<procedure> ::= proc {$ < x >1,...,< x >n} <s> end
<record>, <p> ::= <lit> | <lit>(< f >1:< x >1,...,< f >n:< x >n)
```

La machine abstraite

```
local X in
  local B in
    B=true
    if B then X=1
    else skip end
  end
end
```

Listing 1.1 – Programme en Oz

```
([(local X in
     local B in
       B=true
       if B then x=1 else skip end
     end
   end, {})],
 {})
```

Listing 1.2 – État initial

```
([(local B in
     B=true
     if B then X=1 else skip end
   end, {X → x})],
 {x})
```

```
([(X=1, {B → b, X → x})],
 {b=true, x})
```

```
([(B=true
   if B then X=1 else skip end),
 {B → b, X → x})],

 {b, x})
```

```
([],
 {b=true, x=1})
```

1.2.7 Procédures

Adjonction

```
local X in
  (E1) X=1
  local X in
    (E2) X=2
    {Browse X}
  end
end
E1 = {Browse → b, X → x}
E2 = E1 + {X → y} = {Browse → b, X → y}
```

Restriction

```
local A B C AddB in
  A=1 B=2 C=3 (E)
  fun {AddB X} (EC: contextual environment)
    X+B
  end
end
E = {A → a, B → b, C → c, AddB → a' }
E_C = E_|{B}  = {B → b }
```

1.2.8 Rappel procédure sémantique

```
{Browse {Inc 10}}   #Langage pratique (classique)
```

```
local M in      #Langage Kernel
  local N in
    M=10
    {Inc M N}
    {Browse N}
  end
end
```

1.3 Programmation d'ordre supérieur

1.3.1 FoldL

```
declare
fun {FoldL L F U}
  case L
  of nil then U
  [] H|T then {FoldL T F {F U H}}
  end
end

{FoldL LIST Function Acc}
```

1.4 Lambda calcul

1.4.1 Fonctionnement

```
declare
fun {FoldL L F U}
   case L
   of nil then U
   [] H|T then {FoldL T F {F U H}}
   end
end

{FoldL LIST Function Acc}
```

Fonctionnement

```
t ::= x | (λx.t) | t1  t2
```

1.4.2 En Oz

Définition fonction

```
fun {$ X} T end
```

```
{T1  T2}
```

Currying

```
F = fun {$ X} fun {$ Y} T end end
```

```
{{F X} Y}
```

1.4.3 Sémantique des expressions

α-renaming

```
λ x. x →α λy. y
```

β-renaming

```
(λx.t1)t2 → t1[::= t2]
(λx.(x  x))y → (y  y)
```

η-renaming

```
λx.(t  x) → t if x ∉ FV(t)
```

1.4.4 Eager and Lazy evaluation

```
{Double {Average 5 7}} →
{Double ((5 + 7)/2)} →
{Double (12/2)} →
{Double 6} →
6 + 6 →
12
12
12
12
```

```
{Double {Average 5 7}} →
{Average 5 7} + {Average 5 7} →
((5 + 7)/2) + {Average 5 7} →
(12/2) + {Average 5 7} →
6 + {Average 5 7} →
6 + ((5 + 7)/2) →
6 + ((12)/2) →
6 + 6 →
12
```

1.5 État mutable et abstraction des données

1.5.1 State

```
fun {Sum Xs A}
   case Xs
   of nil then A
   [] X|Xr then {Sum Xr A+X}
   end
end
{Browse {Sum [1 2 3 4] 0}}
```

```
Xs          A
--------------
[1 2 3 4]   0
[2 3 4]     1
[3 4]       3
[4]         6
nil         10
```

1.5.2 Cellule

```
A=5; B=6
C={NewCell A} // on crée une nouvelle cellule
{Browse @C}   // on montre le contenu avec @. Imprime 5
C:=B          // on change la valeur pointée par C en celle de B
{Browse @C}   // Imprime 6
```

1.5.3 Langage Kernel extension Cellule

```
<s> ::= skip
   | < s >1 < s >2
   | local <x> in <s> end
   | < x >1=< x >2
   | <x>=<v>
   | if <x> then < s >1 else < s >2 end
   | {< x >, < y >1, ..., < y >n}
   | case <x> of <p> then < s >1 else < s >2 end
   | {NewCell <y> <x>}
   | <x>:=<y> ____ {Exchange <x> <y> <z>}
   | <y>=@<x> __|
<v> ::= <number> | <procedure> | <record>
<number> ::= <int> | <float>
<procedure> ::= proc {$ < x >1, ..., < x >n} <s> end
<record>, <p> ::= <lit> | <lit>(< f >1:< x >1 ... < f >n:< x >n)
```

1.5.4 Fairy tale

```
fun {MF}
  X = {NewCell 0}
  fun {F ...}
    X:=@X+1 % Definition of F
  end
  fun {G ...}
    % Definition of G
  end
  fun {Count} @X end
in 'export'(f:F g:G c:Count)
end
M = {MF}
```

1.6 Type de données abstraites

1.6.1 ADT avec encapsulation

Wrapper

```
{NewWrapper Wrap Unwrap}  //crée une nouvelle clée d'encryption
W={Wrap X}                //encrypte
X={Unwrap W}              //décrypte
```

ADT

```
local Wrap Unwrap in
  {NewWrapper Wrap Unwrap}

  fun {NewStack} {Wrap nil} end
  fun {Push W X} {Wrap X|{Unwrap W}} end
  fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
  fun {IsEmpty W} {Unwrap W}==nil end
end
```

1.6.2 Objets stateful

Utilisation

```
S={NewStack}
{S push(X)}
{S pop(X)}
{S isEmpty(B)}
```

Objet

```
fun {NewStack}
  C={NewCell nil}
  proc {Push X} C:=X|@C end
  proc {Pop X} S=@C in C:=S.2 X=S.1 end
  proc {IsEmpty B} B=(@C==nil) end
in
  proc {$ M}
    case M of push(X) then {Push X}
    [] pop(X) then {Pop X}
    [] isEmpty(B) then {IsEmpty B} end
  end
```

```
end
```

Objet avec un Record

```
fun {NewStack}
  C={NewCell nil}
  proc {Push X} C:=X|@C end
  proc {Pop X} S=@C in X=S.1 C:=S.2 end
  fun {IsEmpty} @C==nil end
in
  stack(push:Push pop:Pop isEmpty:IsEmpty)
end
```

1.6.3 Functional object (stateless)

```
local
  fun {StackObject S}
    fun {Push E} {StackObject E|S} end
    fun {Pop S1}
      case S of X|T then S1={StackObject T} X end end
    fun {IsEmpty} S==nil end
  in
    stack(push:Push pop:Pop isEmpty:IsEmpty)
  end
  in
    fun {NewStack} {StackObject nil} end
end
```

1.6.4 Stateful ADT

```
local Wrap Unwrap
  {NewWrapper Wrap Unwrap}
  fun {NewStack} {Wrap {NewCell nil}} end
  proc {Push S E} C={Unwrap S} in C:=E|@C end
  fun {Pop S} C={Unwrap S} in
    case @C of X|S1 then C:=S1 X end end
  fun {IsEmpty S} @{Unwrap S}==nil end
in
  Stack=stack(new:NewStack push:Push pop:Pop isEmpty:IsEmpty)
end
```

1.7 Les exceptions

1.7.1 En Oz

```
try <s1> catch <y> then <s2> end  //On crée l'intercepteur
raise <x> end                     //On établit une erreur
```

Exemple typique

```
fun {Eval E}
  if {IsNumber E} then E
  else
    case E of plus(X Y) then {Eval X}+{Eval Y}
```

```
    [] times(X Y) then {Eval X}*{Eval Y}
    else raise badExpression(E) end
    end
  end
end

try
  {Browse {Eval plus(23 times(5 5))}}
  {Browse {Eval plus(23 minus(4 3))}}
catch X then {Browse X} end
```

Finally

```
FH={OpenFile 'foobar'}
try
  {ProcessFile FH}
catch X then
  {Show '*** Exception during execution ***'}
finally {CloseFile FH} end % Always close the file
```

1.8 Programmation Simultanée

1.8.1 Thread

```
thread <s> end
```

1.8.2 Multi-agent

Sieve

```
fun {Sieve Xs}
  case Xs
  of nil then nil
  [] X|Xr then X|{Sieve thread {Filter Xr X} end}
  end
end
declare Xs Ys in
thread Xs={Prod 2} end
thread Ys={Sieve Xs} end
{Browse Ys}
```

```
fun {CMap Xs F}
  case Xs of nil then nil
  [] X|Xr then
  thread {F X} end | {CMap Xr F}
  end
end
```

Compteur thread

```
C={NewCell 0}
proc {Inc C}
  {Exchange C X Y} Y=X+1
end

fun {Fib X}
```

```
  if X==0 then 0
  elseif X==1 then 1
  else
    thread {Inc C} {Fib X-1} end + {Fib X-2}
  end
end
```

1.8.3 Porte digitale

And

```
fun {And A B} if A==1 andthen B==1 then 1 else 0 end end
fun {Loop S1 S2}
  case S1#S2 of (A|T1)#(B|T2) then {And A B}|{Loop T1 T2} end
end
thread Sc={Loop Sa Sb} end
```

Gatemaker

```
fun {GateMaker F}
  fun {$ Xs Ys}
    fun {GateLoop Xs Ys}
      case Xs#Ys of (X|Xr)#(Y|Yr) then
        {F X Y}|{GateLoop Xr Yr}
      end
    end
  in
    thread {GateLoop Xs Ys} end
  end
end
```

```
% Creation d'un interrupteur

proc {Latch C Di Do}
  A B E F
in
  F={DelayG Do}
  A={AndG C F}
  E={NotG C}
  B={AndG E Di}
  Do={OrG A B}
end
```

1.8.4 Serveur non fonctionnel et fonctionnel

```
proc {Server S1 S2}
  case S1|S2 of (M1|T1)|S2
    then (handle M1) {Server T1 S2}
    [] S1|(M2|T2) then
    (handle M2) {Server S1 T2}
  end
end
```

Listing 1.3 – non fonctionnel

```
proc {Server S}
  case S of M|T then
    (handle M)
    {Server T}

  end
end
```

Listing 1.4 – fonctionnel

1.8.5 Port

En Oz

```
P = {NewPort S}    // Cree un port P avec son stream S
{Send P X}         // Envoie X a la fin du stream du port P
```

Langage Kernel

```
P={NewPort S}, {P → p, S → s} //p, s ∈ σ1 crée p = ξ et paire p : s à σ2
{Send P S}, {P → p, X → x}       //p = ξ que s ∈ σ1, p : s ∈ σ2 crée s = x|s' et update p : s
```

1.8.6 Message-passing concurrency

MathProcess

```
proc {Math M}
  case M of
  add(N M A) then A=N+M
  [] mul(N M A) then A=N*M
  ...
  end
end
```

```
MP={NewPort S}
proc {MathProcess Ms}
  case Ms of M|Mr then
    {Math M} {MathProcess Mr}
  end
end
thread {MathProcess S} end
```

ForAll

```
proc {ForAll Xs P}
  case Xs of nil then skip
  [] X|Xr then {P X} {ForAll Xr P}
  end
end
```

```
proc {MathProcess Ms}

  {ForAll Ms Math}

end
```

1.8.7 Objets ports stateless générique

```
fun {NewPortObject0 Process} Port Stream in
  Port={NewPort Stream}
  thread {ForAll Stream Process} end
  Port
end
```

```
fun {NewPortObject0 Process} Port Stream in
  Port={NewPort Stream}
  thread for M in Stream do {Process M} end end
  Port
end
```

1.8.8 Stateful port objects

```
fun {NewPortObject Init F}
  proc {Loop S State}
    case S of M|T then
      {Loop T {F State M}} end
  end
    P
in
  thread S in P={NewPort S} {Loop S Init} end
  P
end
```

```
proc {Loop S State}
  case S of M|T then {Loop T {F State M}} end
end
```

1.8.9 Autres Exemples

Updated NewPortObject

```
fun {NewPortObject Init F}
   P Out
in
   thread S in P={NewPort S} Out={FoldL S F Init} end
   P
end
```

Cell Agent

```
fun {CellProcess S M}
   case M
   of assign(New) then New
   [] access(Old) then Old=S S
   end
end
```

Uniform Interface

```
// On crée et utilise un cell agent
declare Cell
Cell={NewPortObject CellProcess 0}
{Send Cell assign(1)}
local X in {Send Cell access(X)} {Browse X} end

// On veut avoir les mêmes interfaces en tant qu'objet
{Cell assign(1)}
local X in {Cell access(X)} {Browse X} end

// On change la sortie pour être une procédure
fun {NewPortObject Init F}
   P Out
in
   thread S in P={NewPort S} Out={FoldL S F Init} end
   proc {$ M} {Send P M} end
end
```

1.9 Classe et Objet

1.9.1 Définition Classe et Objet

```
class Counter
   attr i
   meth init(X)
      i := X
   end
   meth inc(X)
      i := @i + X
   end
   meth get(X)
      X=@i
   end
end
```

```
{Ctr inc(10)}
{Ctr inc(5)}
local X in
  {Ctr get(X)}
  {Browse X}
end
```

Active Object

```
fun {NewActive Class Init}
  Obj={New Class Init}
  P
in
  thread S in
    {NewPort S P}
    for M in S do {Obj M} end
end
  proc {$ M} {Send P M} end
end
```

1.10 Deterministic dataflow with ports

Définition

```
(< s >1 || < s >2)      // Crée deux threads et attend que les deux se terminent
< s >3                  // S'exécute que quand les deux se terminent
```

Implémentation

```
local X1 X2 in
  thread < s >1 X1 = unit end
  thread < s >2 X2 = unit end
  {Wait X1}
  {Wait X2}
end
```

Higher-order abstraction

```
proc {Barrier Ps}
  Xs={Map Ps fun {$ P} X in thread {P} X=unit end X end}
in
  for X in Xs do
    {Wait X}
  end
end
```

1.10.1 Composition concurrente

```
proc {NewThread P SubThread} //SubThread is an output
  S Pt={NewPort S}
in
  proc {SubThread P}
    {Send Pt 1}
```

```
    thread
      {P} {Send Pt ~1} //Minus sign is tilde
    end
  end
  {SubThread P} //Main computation
  {ZeroExit 0 S} //Keep running sum on S and stop when 0
end
```

Compteur de thread

```
proc {ZeroExit N S}
  case S of X|S2 then
    if N+X==0 then skip
    else {ZeroExit N+X S2} end
  end
end
```

1.11 Erlang

1.11.1 Organisation

```
-module(math).
-export([areas/1]).
-import(lists, [map/2]).

areas(L) -> lists:sum(map(fun(I) -> area(I) end, L)).

area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y.
```

Fonction

```
Pid = spawn(fun) % Pour spawn une fonction et avoir son numero de processus

fun(args) -> expr end % anonyme ou nomme
fun name/arity
```

Receive

```
receive
  pattern1 when guard1 -> expr1;
  pattern2 when guard2 -> expr2;
  ...
  patternN when guardN -> exprN
end
```

1.11.2 Send & Receive

```
Pid ! Message,
...

receive
  Message1 ->
    Actions1;
```

```
  Message2 ->
    Actions2;
  ...
  after Time ->
    TimeOutActions
end
```

1.11.3 Process Linking

```
start() -> spawn(fun go/0).

go() ->
  process_flag(trap_exit, true),
  loop().

loop() ->
  receive
    {'EXIT',Pid,Why} -> ...
    ... -> ..., loop()
  end.
```

1.11.4 Dynamic code change

```
-module(m).

loop(Data, F) ->
  receive
  (From,Q} ->
    {Reply,Data1}=F(Q,Data),
    m:loop(Data1, F)
end.
```

Listing 1.5 – use new version

```
-module(m).

loop(Data, F) ->
  receive
  {From,Q} ->
     {Reply,Data1}=F(Q,Data),
     loop(Data1, F)
end.
```

Listing 1.6 – use old version

1.11.5 Client server hotswapable

```
server(Fun, Data) ->
  receive
    {new_fun,Fun1} ->
      server(Fun1,Data);
    {rpc,From,ReplyAs,Q} ->
      {Reply,Data1} = Fun{Q,Data},
      From!{ReplyAs,Reply},
      server(Fun, Data1)
end.
```

```
% Fonction RPC


rpc(A,B) ->
  Tag=new_ref(),
  A!{rpc,self(),Tag,B},
  receive
    {Tag,Val} -> Val
end.
```