1 Lazy Evaluation

1.1 Semantique / kernel language Lazy function

```
fun lazy {F} <Exp> end

proc{F R}
  {WaitNeeded R} %%attent que R soit demander par un autre thread
  <Exp>
end
```

1.2 Touch function

Permet juste dans lancer les Lazy suspension

```
proc {Touch L N}
  if N == 0 then skip;
  else {Touch L.2 N-1} end %%L.2 lance la lazy susspension car il en a
     besion
end
```

1.3 Producer-Consumer Lazy et Eager

Eager : Le producteur controle car c'est lui qui décide combien d'element sont produit.

```
fun{Prod A N}
  if N == 1 then nil
  else A|{Prod A+1 N-1} end
end

fun{Cons L Acc}
  case L of H|T then
    Acc+H|{Cons T Acc+H}
  end
end
```

Lazy : Le consomateur controle et force Producteur a produire N élements

```
fun lazy {Prod A}
  A|{Prod A+1}
end

fun{Cons L N Acc}
  if N == 0 then nil
  case L of H|T then
    Acc+H|{Cons T N-1 Acc+H}
  end
end
```

1.4 Hamming Probleme

Souvent demander a l'exam, comprendre l'algo et savoir expliquer les lazy suspensions derriere

```
%Multiplie L par N
fun lazy {Times S N}
  case S of H|T then
    N*H|{Times T N}
  end
end
```

```

%Merge L1 et L2
fun lazy {Merge L1 L2}
   case L1|L2 of (H1|T1)|(H2|T2) then
      if H1 < H2 then
         H1|{Merge T1 L2}
      elseif H1 > H2 then
         H2|{Merge L1 T2}
      else
         H1|{Merge T1 T2}
      end
   end
end

%there is no ending, infinite stream
%Only element that are needed quand be computed
H = 1|{Merge {Time H 2} {Merge {Times H 3} {Times H 5}}}
```

1.5 Bounded Buffer

Savoir explique ce que c'est et a quoi sa sert, savoir implémenté
End permet de demander un nouvel element au producteur
Les threads permettent de ne pas bloquer
Combine Eager et lazy

```
%Can be inserted between prod-Cons pipeline whitout changing code
proc{BoundedBuffer S1 S2 N}
   fun lazy {Loop S1 End}
      case S1 of H|T then
         H|{Loop T thread End.2 end} %End.2 demande un elem au prod
      end
   end
   End
in
   thread {List.drop S1 N} end %Ask N element, ne doit pas etre lazy. le
      dernier element de la liste est S1 un unbound var qui sera utiliser
      pour rajouter des elements
   S2 = {Loop S1 End}
end

declare S1 S2 S3 in
   {Browse S1}
   {Browse S2}
   {Browse S3}
   S1 = {Prod 0}
   {BoundedBuffer S1 S2 3}
   S3 = Cons{S2 0}
```

1.6 Lazy QuickSort

La complexité est de $\mathcal{O}(n + k\log(k))$ ou k sont les k premier elements de la liste, vu que c'est du lazy quand demandé, calculé

```
%Divide L in sublist L1 L2
proc {Partition L X L1 L2}
   case L of H|T then
      if H < X then M1 in
```

```
      L1 = H|M1
      {Partition T X M1 L2}
    else M2 in % H>= X
      L2 = H|M2
      {Partition T X L1 M2}
    end
  [] nil then L1=nil L2=nil
  end
end

%To get the first element of L1, it must be needed. if l1 empty then get
   first element of L2
fun lazy{Lappend L1 L2}
  case L1 of H|T then H|{Lappend T L2}
  [] nil then L2
end


fun lazy {LQuickSort L}
  case L of H|T then L1 L2 S1 S2 in
    {Partition T H L1 L2}
    S1 = {LQuickSort L1}
    S2 = {LQuickSort L2}
    {Lappend S1 X|S2}
  [] nil then nil
  end
end

declare S in
S = {LQuickSort [2 73 283 1 384 ~3 482 21] % pour avoir les elements il faut
    soit fait S.1/S.2.1 ... ou {Touch S N}
```

2 Déclarative Programming

2.1 Amortized et Worst Case

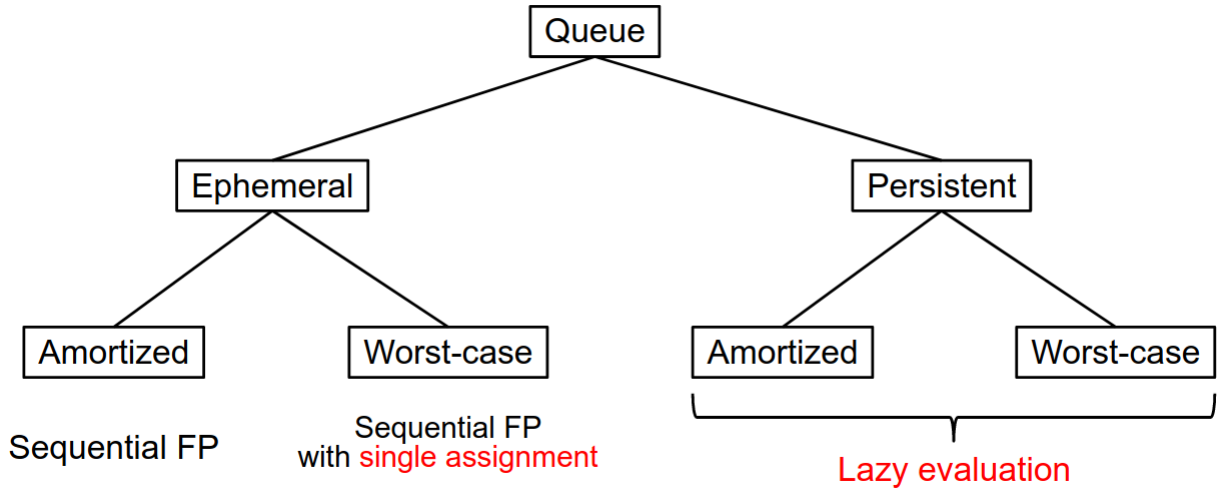
Amortized : si n opérations on un complexité **combiné** de $\mathcal{O}(f(n))$ alors chaque opérations a un complexité ammortie de $\mathcal{O}(\frac{f(n)}{n})$. Utile opérations individuelles sont expensive mais ensemble non. **Worst case** : Tu connais fait pas genre c'est le Big O

2.2 Ephemeral et persistent

Epehemral : Structure de donnée qui n'a qu'une seule version possible au meme moment. Une queue Q1 -¿ Q2 = Insert Q1 1 Q2 remplace Q1 qui n'est plus utilisable

Persistant : Le contraire

2.3 Queue



2.3.1 Amortzed Constant-tile Ephemeral Queue

Amortize O(1) car tous les insert se font en O(1) et le premier delete se fait en O(n) pour reverse la liste mais tant que elle est pas vide on reverse pas donc on peut delete encore n fois et éphemere car on change la liste a chauque fois et donc l'ancienne n'est plus utilisable apres

```
%Queue represented as a tuple q(F R)  where element of the liste is {Append
   F {Reverse R}}
%Inserting is made on R and deleting on F
%If F is empty then {Reverse R} to F

fun{NewQueue} q(nil nil) end
fun{Check Q}
  case Q of q(nil R) then
    q({Reverse R} nil)
  else Q end
end

fun {Insert Q X}
  case Q of q(F R) then
    {Check q(F X|R)}
  end
end

fun {Delete Q X}
  case Q of q(F R) then F1 in
    F = X|F1
    {Check q(F1 R)}
  end
end

Q = {Insert {Insert {Insert {NewQueue} 1} 2 } 3}
Q1 = {Delete Q X}
```

2.3.2 Worst-Case Constant Time Ephemeral Queue

Ici on crée des unbound variable au quelle on rajoute les élements ou retire exactement comme une difference liste

```
%Queue represented as q(N S E) N the size and (S, E) a difference list,
   inserting by updating E et remove by updating S
fun{NewQueue} X in q(0 X X) end

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

fun {Delete Q X}
  case Q of q(N S E) then S1 in
    S = X|S1
    q(N-1 S1 E)
  end
end
```

2.3.3 Amortized Constant Time Persistent Queue

Le fait de la Lazy suspension du Reverse fait que c'est persistant. mais une fois Reverse appeler il est fait d'un cout car Monolithic

```
fun{NewQueue} q(0 nil 0 nil) end

fun{Check Q}
  case Q of q(LenF F LenR R) then
    if LenF < LenR then
      q(LenF + LenR {Lappend F {fun lazy {$} {Reverse R} end}} 0 nil)
    else
      Q
    end
  end
end

fun {Insert Q X}
  Case Q of q(LenF F LenR R) then
    {Check LenF F LenR+1 X|R}
  end
end

fun {Delete Q X}
  case Q of q(LenF F LenR R) then F1 in
    F = X|F1
    {Check q(LenF-1 F1 LenR R)}
  end
end
```

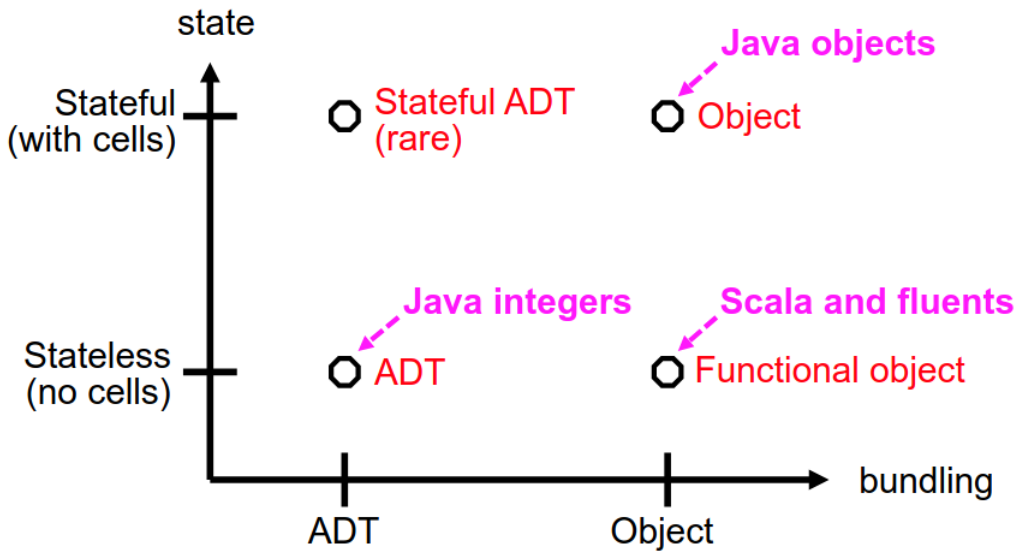
2.3.4 Worst-Case Constant-Time Persistent Queue

Le Append et reverse se font step par step et donc en O(1) donc pas de reverse direct en O(n)

```
fun{NewQueue} q(0 nil 0 nil) end

fun lazy {LAppRev F R B}
  case pair(F R) of pair(X|F2 Y|R2) then
    X|{LAppRev F2 R2 T|B}
  [] pair(nil [Y]) then Y|B
  end
end
```

```
fun {Check Q}
   case Q of q(LenF F LenR R) then
      if lenF < LenR then
         q(LenF + LenR {LAppRev F R nil} 0 nil)
      else Q end
   end
end

fun {Insert Q X}
   case Q of q(LenF F LenR R) then
      {Check q(LenF F LenR+1 X|R)}
   end
end

fun {Delete Q X}
   case Q of q(LenF F LenR R) then F1 in
   F = X|F1
   {Check q(LenF-1 F1 LenR R)}
   end
end
```

3 Declarative Programming



3.1 Abstract DataType (ADT)

Set de valeur et operation
Encapsulation avec des wrapper

```
proc{NewWrapper Wrap UnWrap}
   Key = {NewName}
in
   fun{Wrap X} {Chunk.new w(Key:X)} end
   fun{UnWrap W} W.key end
end

Local Wrap UnWrap in
   {NewWrapper Wrap UnWrap}
```

```
10
11    fun {NewStack} {Wrap nil} end
12    fun {Push W X} {Wrap x|{UnWrap W}} end
13    fun {Pop W X} S = {Unwrap W} in X = S.1 {Wrap S.2} end
14 end
```

3.2 Object

représente valeurs et les opérations

Pas déclaratif car state interne

```
1  Fun {NewStack}
2    C =  {NewCell nim}
3    proc{Push X} C:= X|@C end
4    proc{Pop X} S = @C in C:= S2 X = S.1 end
5  in
6    proc{$ M}
7      case M of push(x) then {Push X}
8      [] pop(x) then {Pop X}
9      end
10   end
11 end
```

3.3 Functional Object

Sans cellules, on utilise juste le static scope et le high order programming

```
1  local
2    fun {StackObject S}
3      fun {Push X} {StackObject X|S} end
4      fun {Pop X}
5        case S of H|T then X=H {StackObject T} end
6      end
7    in
8      stack(push:Push pop:Pop)
9    end
10 in
11   fun {NewStack} {StackObject nil} end
12 end
```

3.4 Stateful ADT

```
1      local Wrap UnWrap
2        {NewWrapper Wrap UnWrap}
3
4        fun {NewStack} {Wrap {NewCell nil}} end
5        fun {Push S X} C ={Unwrap S} in C:=X|@C end
6        fun {Pop S} C ={UnWrap S} in
7          case @C of H|T then C:=T H end
8        end
9      in
10       Stack = stack(new:NewStack push:Push pop:Pop)
11     end
```

4 Messages Passing

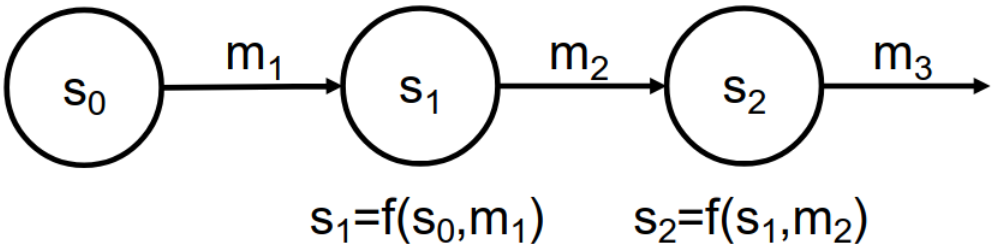
5 Server

Un server doit etre équitavle et le plus rapide possible, c'est pourquoi il est impossible de faire un server Non-déterministique car sinon les client serait dépendant de chaque un et cela ne serait plus dutout équitable si un client a le pouvoir sur d'autre

```
fun {Server S}
  case S of H|T then
    {Handle H} %gerere le message H
    {Server T}
  end
end
```

6 FOLDL !!!

Le truc le plus important




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

7 StateFull port Object

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

7.1 Active Object

```
fun {NewActive Class Init}
  Obj = {New Class Init}
  P
in
  thread S in
    P = {NewPort S}
```

```
    for M in S {Obj M} end
  end
  proc {$ M}{Send P M}
end
```

7.2 Flavius Problem

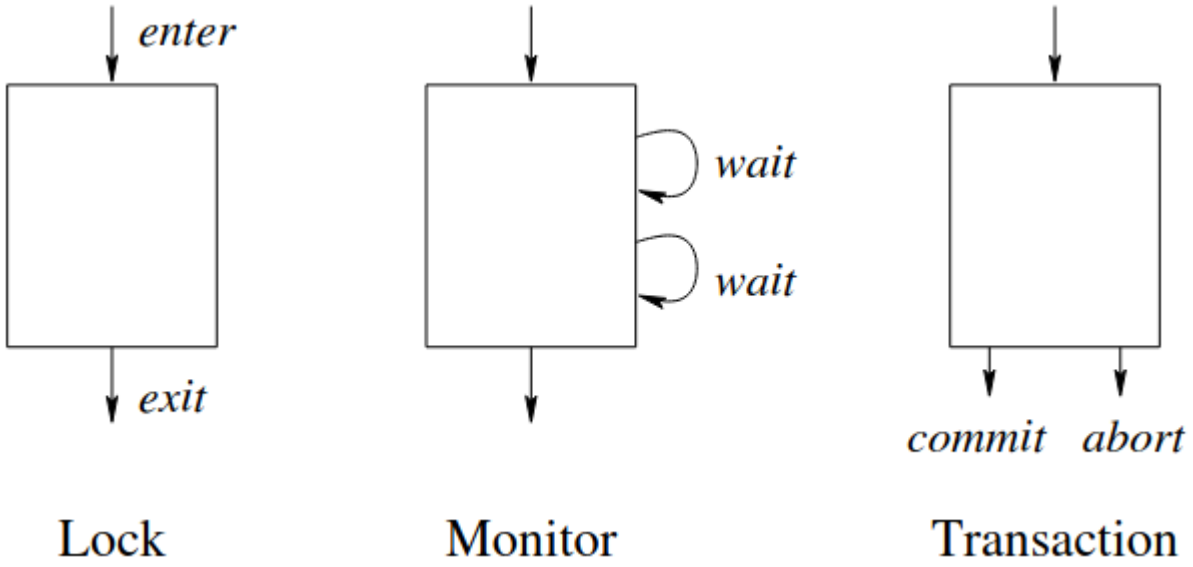
```
class Victim
  attr ident alive step last succ pred

  meth init(I K L)
    alive := true
    step := K
    last := L
    Ident := I
  end

  meth setSucc(S) succ := S end
  meth setPred(P) pred := P end

  meth kill(X, S)
    if @alive then
      if S == 1 then
        @last = ident
      elseif (X mod @step == 0) then
        alive := false
        {@pred setSucc(@succ)}
        {@succ setPred(@pred)}
        {@succ kill(X+1 S-1)}
      else
        {@succ kill(X+1 S)}
      end
    end
  end
end

fun{Josephus N K}
  A = {NewArray 1 N null}
  Last
in
  for I in 1..N do
    A.I := {New Active Victime init(I K Last)}
  end
  for I in 1..(N-1) do
    {A.I setSucc(A.I+1)}
  end
  {A.N setSucc(A.1)}

  for I in 2..(N) do
    {A.I setPred(A.I-1)}
  end
  {A.1 setPred(A.N)}
  {A.1 kill(K N)}

  Last
end
```

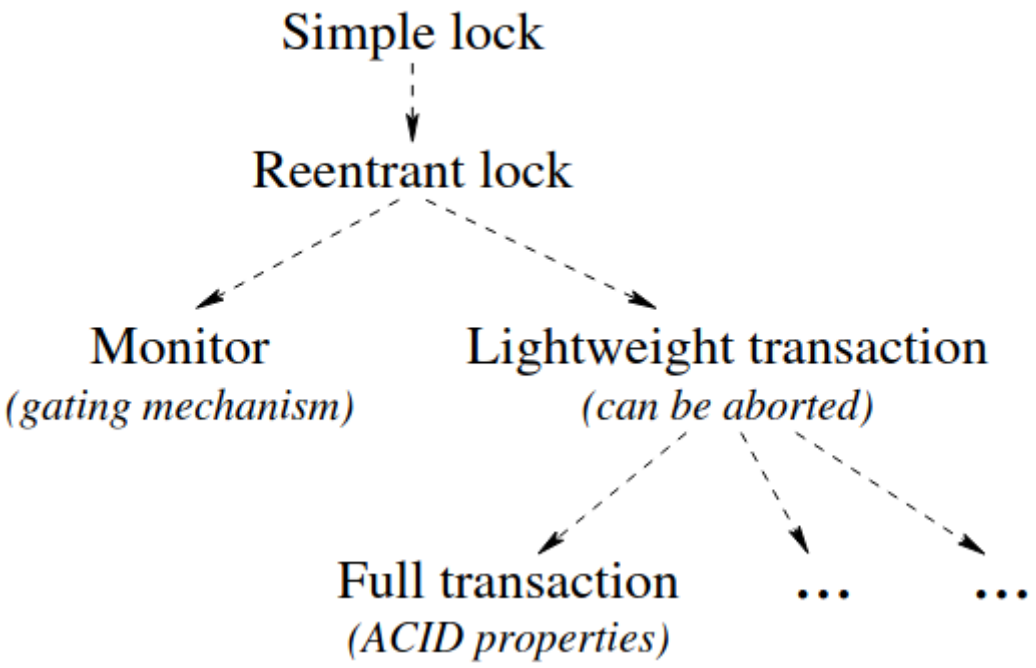
8 Multi-Agent Programming Erlang

8.1 La base

```erlang
%Cree un processus
Pid = spawn(Fun).


%Traiter les messages recus dans la mailBox
receive
  Pattern1 -> Action1
  ...
  PatternN -> ActionN
```

9 Shared State Concurency





9.1 Lock

9.1.1 Queue avec Lock

```

%% Avec le Lock
fun {NewQueue}
   X
   C = {NewCell q(0 nil nil)}
   L = {Lock}

   proc {Insert X} N F B2 in
      lock L then
         q(N F X|B) = @C
         C:= q(N+1 F B2)
      end
   end
   proc {Delete X} N F2 B in
      lock L then
         q(N X|F2 B) = @C
         C := q(N-1 F2 B)
      end
   end
in
   q(insert:Insert delete:Delete)
end

%Avec la fonction Exchange qui est atomique
fun {NewQueue}
   X
   C = {NewCell q(0 nil nil)}

   proc {Insert X} N F B2 M in
      {Exchange C q(N F X|B2) q(M F B2)} %Atomique aucun thread ne peut se
          glisser entre
      M = N+1
   end
   proc {Delete X} N F2 B M in
      {Exchange C q(N X|F2 B) q(M F2 B)}
      M = N+1
   end
in
   q(insert:Insert delete:Delete)
end
```

9.1.2 Implémentation Lock

```
%% Pas reetrant
fun {SimpleToken}
   Token = {NewCell unit}
   proc{Lock P} Old New in
      {Exchange Token Old New}
      {Wait Old}
      {P}
      New = unit
   end
in
   lock(lock:Lock)
end
```

```
13
14 %% Reetrant
15
16 fun {NewLock}
17   Token = {NewCell unit}
18   CurThr = {NewCell unit}
19
20   proc{Lock P}
21     if {Thread.this} == CurThr then
22       {P}
23     else Old New in
24       {Exchange Token Old New}
25       {Wait Old} %Entre section critique
26       CurThr := {Thread.this}
27       try {P} finally
28         CurThr := unit
29         New := unit % part section critique
30       end
31     end
32   end
33 end
34
35 %% Queue avec Tuple Space
36 fun {NewQueue}
37   X
38   TS = {New TupleSpace init}
39   proc {Insert X} N F B2 in
40     {TS read(q q(N F X|B2))}
41     {TS Write(q(N+1 F B2))}
42   end
43
44   proc{Delete X} N F2 B in
45     {TS read(q q(N X|F2 B))}
46     {TS write(q(N-1 F2 B))}
47   end
48 in
49   {TS write(q(0 X X))}
50   queue(insert:Insert delete:Delete)
51 end
```

9.2 Monitor

9.2.1 Buffer avec Monitor

```
1  class Buffer
2    attr m buf first last n i
3    meth init(N)
4      m := {NewMonitor}
5      buf := {NewArray 0 N-1 null}
6      first := 0
7      last := 0
8      n := N
9      i := 0
10   end
11
12   meth put(X)
13     {@m.lock proc{$}
14       if @i > @n then
15         {@m.wait}
```

```
        {self.put(X)} % Toujours relancer car sinon Buggy
      else
        @buf.@last := X
        last := (@last + 1) mod @n
        i := @i + 1
        {@m.notifyAll}
      end
    end}
  end

  meth get(X)
    {@m.lock proc{$}
      if @i == 0 then
        {@m.wait}
        {self.get(X)} % Toujours relancer car sinon Buggy
      else
        X := @buf.@first
        first := (@first + 1) mod @n
        i:= @i - 1;
        {@m.notifyAll}
      end
    end}
  end
end
```

9.2.2 Implémentation

```
fun {NewGRLock}
  Token1 = {Newcell unit}
  Token2 = {Newcell unit}
  CurThr = {Newcell unit}

  fun{GetLock}
    if {Thread.this} == 0 then
       Old New
    in
      {Exchange Token1 Old New}
      {Wait Old}
      Token2 = New
      CurThr = {Thread.this}
      true
    else
      false
    end
  end

  proc {ReleaseLock}
    CurThr := unit
    unit = @Token2 %Pass Token
  end
end

% Extended Queue
fun{NewQueue}
  X
  C = {NewCell q(0 X X)}
  L = {NewLock}

  proc {Insert X} N F B2 in
```

```
33      lock L then
34        q(N F X|B) = @C
35        C:= q(N+1 F B2)
36      end
37    end
38    proc {Delete X} N F2 B in
39      lock L then
40        q(N X|F2 B) = @C
41        C := q(N-1 F2 B)
42      end
43    end
44
45    fun {Size}
46      lock L then @C.1 end
47    end
48
49    fun {DeleteAll}
50      lock L then X S E in
51        q(_ S E) = @C
52        C := q(0 X X)
53        E = nil
54        S
55      end
56    end
57
58    fun {DeleteNonBlock}
59      lock l then
60        if {Size} > 0 then [{Delete}]
61        else nil end
62      end
63    end
64  in
65    queue(insert:Insert delete:Delete size:Size deleteAll:DeleteAll
          deleteNonBlock:DeleteNonBlock)
66  end
67
68  %Monitor
69
70  fun {NewMonitor}
71    Q = {NewQueue}
72    L = {NewGRLock}
73
74    proc {LockM P}
75      if{L.get} then
76        try {P} finally {L.release} end
77      else {P} end
78    end
79
80    proc {WaitM} X in
81      {Q.insert C}
82      {L.release}
83      {Wait X}
84      if {L.get} then skip end
85    end
86
87    proc {NotifyM}
88      U = {Q.deleteNonBlock}
89    in
90      case U of [X] then X=unit
91      else skip end
```

```
92    end
93
94    proc{NotifyAllM}
95       L={Q.deleteAll}
96    in
97       for X in L do X=unit end
98    end
99
100 end
```