APPENDICE B

Code source

On trouvera ici le code source des packages ou domaines évoqués dans le texte. On se reportera à [Scr] pour la syntaxe de Scratchpad II.

§ 1. IDENTIFIABILITÉ

1. CONVPACK

Ce package réalise quelques coercions utiles pour IDPACK : la fonction *mkeqQ* prend une fraction $f_i = P_i/Q_i$ et retourne $Q_i(y)P_i(x) - P_i(y)Q_i(x) \in \mathbf{Q}(y)[x]$. Une variante *mkeqP* traite le cas où f_i est un polynôme. La fonction *mkeqDiv* retourne $Q_i(x)u_i - 1$.

```
)abb package CONP ConvPack

ConvPack(le:List Expression):Public == Prive where
   E ==> Expression
   SE ==> SortedExpressions
   L ==> List
   LE ==> L E
   I ==> Integer
   PI  ==> Polynomial I
   RFI ==> RationalFunction I
   NDMP ==> NewDistributedMultivariatePolynomial(le,RFI)
   OV ==> OrderedVarlist le
   Public == with
     convertP : PI          - > NDMP
     mkeqP    : (PI,PI)      - > NDMP
     mkeqQ    : (RFI,RFI)    - > NDMP
     mkeqP    : PI          - > NDMP
     mkeqQ    : RFI          - > NDMP
     mkeqP    : PI          - > NDMP
     mkeqQ    : RFI          - > NDMP
     mkeqDiv  : (PI,E)       - > NDMP
   Prive == add
```

```
-- fonctions locales
   ov       : E   - > OV
   convertP1:(PI,L E) - > NDMP

-- implantation
   mkeqDiv(pol,e) ==
      convertP(pol) * varPol(ov(e))$NDMP - 1
   ov(ex) == (coerce(ex)$OV):OV
   convertP1(pol,l) ==
      null(l) => pol::RFI::NDMP
      ex:=l.0
      se:=ex::SE
      ¬ member(se,varlist pol) => convertP1(pol,rest l)
      var:=varPol(ov ex)$NDMP
      upol:=likeUniv(pol,se)
      res:NDMP:=0
      while upol ¬ = 0 repeat
         res:= res + var**(degree upol)*convertP1(lc upol,rest l)
         upol:=red upol
      res

   convertP(pol:PI) ==
      l:=le
      convertP1(pol,l)

   mkeqP(pol1:PI,pol2:PI) ==
      convertP(pol2)-pol1::RFI::NDMP

   mkeqQ(rat1:RFI,rat2:RFI) ==
      pol1:PI:=numer(rat2)
      pol2:PI:=denom(rat2)
      convertP(pol1)-convertP(pol2)*rat1::NDMP

   mkeqP(pol:PI) ==
      convertP(pol)-pol::RFI::NDMP

   mkeqQ(rat:RFI) ==
      pol1:PI:=numer(rat)
      pol2:PI:=denom(rat)
      convertP(pol1)-convertP(pol2)*rat::NDMP
```

2. IDPACK

Ce package teste l'identifiabilité par la méthode de l'idéal Δ. Il traite aussi la discernabilité en vérifiant qu l'idéal obtenu est trivial (voir chap. V § 5 n° 2). La fonction *identifiable* prend comme premier argument un résumé exhaustif, comme second la liste des paramètres a priori inconnus et comme troisième argument la liste des paramètres dont on veut tester l'identifiabilité. Par défaut, s'il n'y a pas de troisième argument, on teste l'identifiabilité de tous les paramètres.

```
)abb package IDPACK IdentifiabilitePackage

IdentifiabilitePackage(): Public == Prive where
   L     ==> List
   E     ==> Expression
```

```
SE    ==> SortedExpressions
I     ==> Integer
NNI   ==> NonNegativeInteger
PI    ==> Polynomial I
RFI   ==> RationalFunction I
NDMP  ==> NewDistributedMultivariatePolynomial
NDP   ==> NewDirectProduct
GB    ==> GroebnerPackage
CONP  ==> ConvPack
Public == with
  distingable? : (L PI,L PI,L E)       -> Boolean
  distingable? : (L RFI,L RFI,L E)     -> Boolean
  identifiable? : (L PI,L E)           -> Boolean
  identifiable? : (L RFI,L E)          -> Boolean
  identifiable? : (L PI,L E,L E)       -> Boolean
  identifiable? : (L RFI,L E,L E)      -> Boolean
  lpP          : ()                    -> L PI
  putlpP       : L PI                  -> L PI
  stdP         : ()                    -> L E
  leP          : ()                    -> L E
  putleP       : L E                   -> L E
  putstdP      : L E                   -> L E
  lpQ          : ()                    -> L RFI
  putlpQ       : L RFI                 -> L RFI
  stdQ         : ()                    -> L E
  leQ          : ()                    -> L E
  putleQ       : L E                   -> L E
  putstdQ      : L E                   -> L E
Prive == add
-- On garde en memoire les arguments du dernier calcul de base standard
-- pour eviter de la recalculer inutilement
  memo:Record(LPP:L PI,LPQ:L RFI,LEP:L E,LEQ:L E,STDP:L E,_
              STDQ:L E) := [[],[],[],[],[],[]]

-- fonctions locales

-- implantation

-- cas particulier ou le resume est polynomial
  distingable?(lp1:L PI,lp2:L PI,le2:L E):Boolean ==
    ndmp:= NDMP(le2,RFI)
    lndmp:= L ndmp
    conp:=CONP(le2)
    ndp:=NDP(£ le2,NNI)
    systeme:L ndmp:=
      [mkeqP(pol1,pol2)$conp for pol1 in lp1 for pol2 in lp2]
    systeme:=removeDuplicates systeme
    systeme:=delete(0$ndmp,systeme)$L(ndmp)
    gb:=GB(RFI,ndp,ndmp)
    resu:=groebner(systeme)$gb
    resu.0 =1$ndmp => true
    false

-- cas general rationnel
  distingable?(lp1:L RFI,lp2:L RFI,le2:L E):Boolean ==
    lden:L PI:=[]
    for rat in lp2 repeat
      if ¬ isconst?(pol:=denom(rat)) then lden:=cons(pol,lden)
    lden:=removeDuplicates lden
```

```
      lvDiv:L E:=[i::E for i in 1..(£ lden)]
      leSys:=append(le2,lvDiv)
      ndmp:=NDMP(leSys,RFI)
      lndmp:=L ndmp
      ndp:=NDP(£ leSys,NNI)
      conp:=CONP(leSys)
      -- on fabrique le systeme engendrant l'ideal J (chap. V.5.2 p. 112)
      systeme:L ndmp:=
        [mkeqQ(pol1,pol2)$conp for pol1 in lp1 for pol2 in lp2]
      systeme:=removeDuplicates systeme
      systeme:=delete(0$ndmp,systeme)$L(ndmp)
      sysDiv:L ndmp:=
        [mkeqDiv(pol,e) for pol in lden for e in lvDiv]
      systeme:=append(systeme,sysDiv)
      gb:=GB(RFI,ndp,ndmp)
      resu:=groebner(systeme)$gb
      -- on teste que l'ideal est trivial
      resu.0 =1$ndmp => true
      false

  -- cas polynomial
    identifiable?(lp:L PI,le:L E,lVar:L E):Boolean ==
      ndmp:=NDMP(le,RFI)
      lndmp:= L ndmp
      conp:=CONP(le)
      sol:lndmp:= if lp=lpP() and le=leP() then stdP():lndmp else
        ndp:=NDP(£ le,NNI)
        systeme:L ndmp:=[mkeqP(pol)$conp for pol in lp]
        systeme:=removeDuplicates systeme
        systeme:=delete(0$ndmp,systeme)$L(ndmp)
        gb:=GB(RFI,ndp,ndmp)
        resu:=groebner(systeme)$gb
        putlpP(lp)
        putstdP(resu:L(E))
        putleP(le)
        resu:lndmp
      etalon:L ndmp:=[mkeqP(varPol(e::SE)$PI)$conp for e in lVar]
      for np in etalon repeat
        ¬ member(np,sol) => return false
      true

  -- cas rationnel
    identifiable?(lp:L PI,le:L E):Boolean  ==
      identifiable?(lp,le,le)

    identifiable?(lp:L RFI,le:L E,lVar:L E):Boolean ==
      lden:L PI:=[]
      for rat in lp repeat
        if ¬ isconst?(pol:=denom(rat)) then lden:=cons(pol,lden)
      lden:=removeDuplicates lden
      lvDiv:L E:=[i::E for i in 1..(£ lden)]
      leSys:=append(le,lvDiv)
      ndmp:=NDMP(leSys,RFI)
      lndmp:=L ndmp
      conp:=CONP(leSys)
      sol:lndmp:= if lp=lpQ() and le=leQ() then stdQ():lndmp else
        ndp:=NDP(£ leSys,NNI)
        -- on fabrique le systeme engendrant l'ideal J du th. III.2.3.3 p.51
        systeme:L ndmp:=[mkeqQ(rat)$conp for rat in lp]
```

```
        systeme:=removeDuplicates systeme
        systeme:=delete(0$ndmp,systeme)$L(ndmp)
        sysDiv:L ndmp:=
          [mkeqDiv(pol,e) for pol in lden for e in lvDiv]
        systeme:=append(systeme,sysDiv)
        gb:=GB(RFI,ndp,ndmp)
        resu:=groebner(systeme)$gb
        putlpQ(lp)
        putstdQ(resu:L(E))
        putleQ(le)
        resu:lndmp
      etalon:L ndmp:=[mkeqP(varPol(e::SE)$PI)$conp for e in lVar]
      for np in etalon repeat
        -- on teste que xi - yi est dans l'ideal
        ¬ member(np,sol) => return false
      true

    identifiable?(lp:L RFI,le:L E):Boolean ==
      identifiable?(lp,le,le)

    stdP() == memo.STDP
    putstdP(le)  == memo.STDP := le
    stdQ() == memo.STDQ
    putstdQ(le)  == memo.STDQ := le
    lpP() == memo.LPP
    lpQ() == memo.LPQ
    putlpP(lp) == memo.LPP:=lp
    putlpQ(lr) == memo.LPQ:=lr
    leP()==memo.LEP
    leQ()==memo.LEQ
    putleP(le)==memo.LEP:=le
    putleQ(le)==memo.LEQ:=le
```

3. STRUCTLS

Ce domaine modélise les structures linéaines stationnaires avec conditions initiales nulles.

```
-- Address comments and questions to
--   François OLLIVIER
--   Laboratoire d'Informatique de L'X (LIX)
--   Ecole Polytechnique
--   91 128 Palaiseau cedex (FRANCE)
--   BITNET: CFFOLL@FRPOLY11

)abb domain STRUCTLS StructureLineaireStationnaire

StructureLineaireStationnaire():Public == Prive where
  I    ==> Integer
  PI   ==> PositiveInteger
  L    ==> List
  E    ==> Expression
  S    ==> String
  Pol  ==> Polynomial I
  M    ==> Matrix Pol
  SUPol ==> SparseUnivariatePolynomial Pol
```

```
Pol2  ==> QuotientField SUPol
M2    ==> Matrix Pol2
ID    ==> IdentifiabilitePackage

Public == Set with
   new              :  (I,I,I,L E)      -> $
   changeA          :  ($,I,I,Pol)      -> $
   changeB          :  ($,I,I,Pol)      -> $
   changeC          :  ($,I,I,Pol)      -> $
   changeLE         :  ($,L E)          -> $
   etat             :  ($,I,I,Pol)      -> $
   etat             :  ($,I,Pol)        -> $
   com              :  ($,I,I,Pol)      -> $
   obs              :  ($,I,I,Pol)      -> $
   matriceDeTransfert :  $              -> M2
   resumExhMarkov   :  $                -> L Pol
   resumExhTransfert :  $               -> L Pol
   resumExhSpecial  :  $                -> L Pol
   resumExhSpecial  :  ($,L Pol)        -> L Pol
   detXidMoinsA     :  $                -> L Pol
   minDetXidMoinsA  :  ($,I,I)          -> L Pol
   get              :  (S,S)            -> Union($,"failed")
   save             :  ($,S,S)          -> $
   liste            :  S                -> L S
   distingable?     :  ($,$,S)          -> Union(Boolean,"failed")
   identifiable?    :  ($,S)            -> Boolean
   identifiable?    :  ($,L E,S)        -> Boolean
   a                :  $                -> M
   b                :  $                -> M
   c                :  $                -> M
   stdM             :  $                -> L E
   stdT             :  $                -> L E
   stdS             :  $                -> L E
   lParms           :  $                -> L E

Prive == add
-- La representation interne contient le trois matrices,
-- la liste des coefficients inconnus, les resumes deja
-- calcules, ainsi que les bases standard correspondant
-- aux tests d'identifiabilite, pour eviter de repeter
-- les calculs. coefficient
   Rep:=Record(A:M,B:M,C:M,LE:L E,REM:L Pol,RET:L Pol,RES:L Pol,_
               STDM:L E,STDT:L E,STDS:L E)

   KAF:= KeyedAccessFile $
   LIB:= LibraryName
   lvP:L Pol:=[]
   lvE:L E:=[]



-- fonctions locales
   liscoe   : SUPo  -> L Pol
   move     : Pol   -> Pol2
   move2    : M     -> M2
   createKaf : S    -> KAF
-- pour oublier les calculs deja faits sur une structure
   reset    : $     -> $
```

```
-- implantation

   liscoe(supol:SUPol):L Pol ==
      resu:L Pol:=[0$Pol for i in 0..(degree supol)]
      while supol ¬ = 0 repeat
         resu.(degree supol):=lc supol
         supol:=reductum supol
      resu

   move(pol:Pol):Pol2 == pol::SUPol::Pol2

   move2(mat:M):M2 ==
      nr:=nrows(mat)
      nc:=ncols(mat)
      ir:=nr::I -1
      ic:=nc::I -1
      mat2:=zero(nr,nc)$M2
      for i in 0..ir repeat
         for j in 0..ic repeat
            mat2.i.j:=move(mat.i.j)
      mat2

   reset(struct:$):$ ==
      struct.REM:=lvP
      struct.RET:=lvP
      struct.RES:=lvP
      struct.STDM:=lvE
      struct.STDT:=lvE
      struct.STDS:=lvE
      struct

   createKaf(fich:S):KAF ==
      libf:=find(fich)$LIB
      lib:LIB:=
         libf case "failed" => new(fich)$LIB
         libf::LIB
      open(lib)$KAF

-- fonction de sortie
   coerce(s:$):E ==
      eA:=(s.A)::E
      eB:=(s.B)::E
      eC:=(s.C)::E
      ligne1:=mkBinary(_=::E,mkBinary(_/::E,dX::E,dt::E),
      mkBinary(_+::E,mkBinary(_*::E,eA,X::E),mkBinary(_*::E,eB,U::E)))
      ligne2:=mkBinary(_=::E,"Y "::E,mkBinary(_*::E,eC,X::E))
      ligne3:=mkBinary(_=::E,"Parametres internes "::E,(s.LE)::E)
      mkBinary(SC::E,ligne1,mkBinary(SC::E,ligne2,ligne3))

-- fonction d'ecriture sur disque
   save(s:$,fich:S,nom:S):$ ==
      kaf:=createKaf(fich)
      setelt(kaf,nom,s)$KAF

-- fonction de lecture sur disque
   get(fich:S,nom:S):Union($,"failed") ==
      kaf:=createKaf(fich)
      search(kaf,nom)$KAF
```

```
    liste(fich:S):L S == keys createKaf fich

 -- fonctions d'entree

    new(r,p,q,le) ==
       [zero(r,r)$M,zero(r,p)$M,zero(q,r)$M,le,_
        lvP,lvP,lvP,lvE,lvE,lvE]$Rep

-- passage du compartiment i au compartiement j avec la constante pol
    etat(struct,i,j,pol) ==
       i:=i-1
       j:=j-1
       A1:=struct.A
       A1.i.i:=A1.i.i - pol
       A1.j.i:=A1.j.i + pol
       reset struct
-- evacuation du compartiment i avec la constante pol
    etat(struct,i,pol) ==
       i:=i-1
       A1:=struct.A
       A1.i.i:=A1.i.i - pol
       reset struct
-- commande du compartiment i par uj avec la constante pol
    com(struct,i,j,pol) ==
       i:=i-1
       j:=j-1
       B1:=struct.B
       B1.i.j:= pol
       reset struct
   -- la mesure du compartiment j avec la constante pol est yi
    obs(struct,i,j,pol) ==
       i:=i-1
       j:=j-1
       C1:=struct.C
       C1.i.j:=pol
       reset struct

-- pour changer une des trois matrices A, B ou C
-- (voir def. V.1.3.2 p. 105)
    changeA(s,i,j,pol) ==
       i:=i-1
       j:=j-1
       s.A.i.j:=pol
       reset s
    changeB(s,i,j,pol) ==
       i:=i-1
       j:=j-1
       s.B.i.j:=pol
       reset s
    changeC(s,i,j,pol) ==
       i:=i-1
       j:=j-1
       s.C.i.j:=pol
       reset s

-- pour changer la liste des paramètres inconnus
    changeLE(s:$,le:L E):$ ==
       s.STDM:=lvE
       s.STDT:=lvE
```

```
    s.STDS:=lvE
    s.LE:=le
    s

-- fonctions de recuperation des valeurs de la representation interne
  a(s) == s.A
  b(s) == s.B
  c(s) == s.C
  lParms(s) == s.LE
  stdM(s) == s.STDM
  stdT(s) == s.STDT
  stdS(s) == s.STDS

-- Resumes exhaustifs
  resumExhMarkov(s) ==
    ¬ null(re:=s.REM) => re
    A2:=s.A
    B2:=s.B
    C2:=s.C
    r:=nrows(A2)
    SM:=SquareMatrix(r:PI,Pol)
    A3:=A2:SM
    lp:L Pol:=[]
    for i in 1..(2*r-1) repeat
      mat:M:=((C2 * (A3**i):M)::M * B2)::M
      lp:=append(ravel mat,lp)
    s.REM:=lp

  matriceDeTransfert(s) ==
    r:=nrows(s.A)
    SM:= SquareMatrix(r::PI,Pol2)
    A2:=move2(s.A):SM
    B2:=move2(s.B)
    C2:=move2(s.C)
    XX:=varPol()$SUPol::Pol2
    MX:=XX::SM
    A2:=MX-A2
    A3:=recip(A2):M2
    ((C2*A3)::M2 * B2)::M2


  resumExhTransfert(s) ==
    ¬ null(re:=s.RET) => re
    A2:=matriceDeTransfert(s)
    ir:=nrows(A2)::I - 1
    ic:=ncols(A2)::I - 1
    lp:L Pol:=[]
    for i in 0..ir repeat
      for j in 0..ic repeat
        frac:Pol2:=A2.i.j
        pd:SUPol:=denom frac
        lp:=append(listCoef pd,lp)
        pn:SUPol:=numer frac
        lp:=append(liscoe pn,lp)
    s.RET:=lp

  detXidMoinsA(s:$):L Pol ==
    A1:=s.A
    nr:=nrows A1
```

```
    SM:=SquareMatrix(nr:PI,Pol2)
    A2:=move2(A1):SM
    XX:=varPol()$SUPol::Pol2
    MX:=XX::SM
    A2:=MX-A2
    det:Pol2:=determinant A2
    listCoef(numer det)

  minDetXidMoinsA(s:$,i:I,j:I): L Pol ==
    i:=i-1
    j:=j-1
    A1:=s.A
    nr:=(nrows A1)-1
    ir:=nr-1
    SM:=SquareMatrix(nr:PI,Pol2)
    A2:=move2(A1)
    A3:=0$SM
    for ii in 0..ir repeat
      for jj in 0..ir repeat
        iii:=if ii < i then ii else ii+1
        jjj:=if jj < j then jj else jj+1
        A3.ii.jj:= A2.iii.jjj
    XX:=varPol()$SUPol::Pol2
    MX:=XX::SM
    A3:=MX-A3
    det:Pol2:=determinant A3
    listCoef(numer det)


-- pour donner explicitement un resume calcule par une autre methode
  resumExhSpecial(s:$,lp:L Pol):L Pol ==
    s.RES:=lp
  resumExhSpecial(s) == s.RES

 -- identifiabilite (utilise IDPACK)
  inpu ==> READSPADEXPR$Lisp

  distingable?(s1:$,s2:$,str:S):Union(Boolean,"failed") ==
    nrows a s1 ¬ = nrows a s2 => "failed"
    nrows b s1 ¬ = nrows b s2 => "failed"
    ncols c s1 ¬ = ncols c s2 => "failed"
    while(str¬ ="Markov" and str¬ ="Transfert") repeat
      SAY("Les options sont Markov et Transfert.")$Lisp
      SAY("Entrez votre option.")$Lisp
      str:=inpu()
    if str="Markov" then
      resu1:L Pol:=resumExhMarkov s1
      resu2:L Pol:=resumExhMarkov s2
    if str="Transfert" then
      resu1:L Pol:=resumExhTransfert s1
      resu2:L Pol:=resumExhTransfert s2
   -- appelle IDPACK
    distingable?(resu1,resu2,lParms s2)$ID

  identifiable?(s:$,lVar:L E,str:S) ==
    while(str¬ ="Markov" and str¬ ="Transfert" and str¬ ="Special")_
      repeat
        SAY("Les options sont Markov, Transfert et Special.")$Lisp
        SAY("Entrez votre option.")$Lisp
```

```
        str:=inpu()
    le:=lParms s
    lp:L Pol:=lvP
    boo1:Boolean:=true
    if str="Markov" then
        lp:=resumExhMarkov s
        std:=stdM s
        ¬ null std =>
          boo1:=false
          putleP(le)$ID
          putlpP(lp)$ID
          putstdP(std)$ID
      else if str="Transfert" then
        lp:=resumExhTransfert s
        std:=stdT s
        ¬ null std =>
          boo1:=false
          putleP(le)$ID
          putlpP(lp)$ID
          putstdP(std)$ID
      else
        lp:=resumExhSpecial s
        std:=stdS s
        ¬ null std =>
          boo1:=false
          putleP(le)$ID
          putlpP(lp)$ID
          putstdP(std)$ID
   -- appelle IDPACK
    boo:Boolean:=identifiable?(lp,le,lVar)$ID
    std:=stdP()$ID
   -- stocke la base standard, si ce n'est deja fait
    if boo1 then
      if str="Markov" then s.STDM:=std else
        if str="Transfert" then s.STDT:=std else s.STDS:=std
    boo

  identifiable?(s,str) == identifiable?(s,lParms s,str)$$
```

§ 2. BASES CANONIQUES

1. MOFAM

Ce domaine implante le produit du monoïde des monômes en x et du monoïde libre engendré par l'ensemble des monoômes de tête, avec un ordre correspondant aux hypothèses de la prop. III.3.2.2.5 p. 57.

```
-- % MutableOrderedFreeAbelianMonoid
)abbrev domain MOFAM MutableOrderedFreeAbelianMonoid

MutableOrderedFreeAbelianMonoid(Mon): Exportation == Production where
```

```
Mon   : OrderedAbelianMonoidSup
I    ==> Integer
NNI  ==> NonNegativeInteger
E    ==> Expression
L    ==> List
SEX  ==> SExpression
OUT  ==> OutputPackage
Term ==> Record(gen: I, exp: NNI)
Tag  ==> List Term

Exportation ==> OrderedAbelianMonoidSup with
   coet       : I       -> $
   member?    : ($, I)  -> Boolean
   coev       : Mon     -> $
   new        : (I,Mon) -> $
   delete     : I       -> Void
   clearAll   : ()      -> Void
   lisTag     : ()      -> L I
   purTag?    : $       -> Boolean
   getTag     : $       -> L Record(gen: I, exp: NNI)
   eval       : $       -> Mon
   eval1      : Tag     -> Mon

Production ==> add
-- representation
-- Une partie "en x" val et une autre correspondant au monoide libre tag
   Rep:= Record(val: Mon,tag: Tag)

-- variable globale
   Rec:= Record(index:I,deg:Mon)
   LRec:= L Rec
-- On garde la liste des monomes de tete, car ils servent a definir l'ordre
   lTag:LRec:=[[0,0]]

   i: I
   f, g: $
   n: NNI
   m: I
   mon:Mon

 -- local
   getVal(m:I):Mon ==
      for cc in lTag.rest repeat
         cc.index = m => return cc.deg
      0$Mon

   sub(t:Term):E ==
      t.exp = 1 => mkBinary("SUB"::E,"p"::E,(t.gen)::E)
      mkBinary("*"::E,(t.exp)::E, mkBinary("SUB"::E,"p"::E,(t.gen)::E))

   plus(t1:Tag, t2:Tag):Tag ==
      t1 = [] => t2
      t2 = [] => t1
      term1:=t1.first
      term2:=t2.first
      term1.gen > term2.gen =>
         cons(term1,plus(t1.rest,t2))
      term1.gen < term2.gen =>
         cons(term2,plus(t2.rest,t1))
```

```
      cons([term1.gen,term1.exp + term2.exp],plus(t1.rest,t2.rest))

    moins(t1:Tag, t2:Tag):Union(Tag,"failed") ==
      t2 = [] => t1
      t1 = [] => "failed"
      term1:=t1.first
      term2:=t2.first
      term1.gen > term2.gen =>
        (bb:= moins(t1.rest,t2)) case "failed" => return "failed"
        cons(term1,bb::Tag)
      term1.gen < term2.gen => "failed"
      (aa:= term1.exp - term2.exp) case "failed" => return "failed"
      (bb := moins(t1.rest,t2.rest)) case "failed" => return "failed"
      cc:= bb::Tag
      aa = 0 => cc
      cons([term1.gen,aa::NNI],cc)

    sup1(t1:Tag, t2:Tag):Tag ==
      t1 = [] => t2
      t2 = [] => t1
      term1:=t1.first
      term2:=t2.first
      term1.gen > term2.gen =>
        cons(term1,sup1(t1.rest,t2))
      term1.gen < term2.gen =>
        cons(term2,sup1(t2.rest,t1))
      cons([term1.gen,sup(term1.exp,term2.exp)],sup1(t1.rest,t2.rest))

    mult(n:NNI, t1:Tag):Tag ==
      n = 0 => []
      [[term.gen,n * term.exp] for term in t1]

-- Ordre lexicographique inverse
    less(t1:Tag,t2:Tag):Boolean ==
      t1 = [] => false
      t2 = [] => true
      term1:=t1.first
      term2:=t2.first
      ind1:=term1.gen
      ind2:=term2.gen
      ind1 > ind2 => true
      ind1 < ind2 => false
      (exp1:=term1.exp) > (exp2:=term2.exp) => true
      exp1 < exp2 => false
      less(t1.rest,t2.rest)

-- fonctions exportees
    coet(m:I):$ == [0$Mon,[[m,1$NNI]]$Tag]$Rep

    member?(f,m) ==
      ft:=f.tag
      while ft ¬ = [] repeat
        (m2:=ft.first.gen) < m => return false
        m2 = m => return true
        ft:=ft.rest
      false


    coev(mon) == [mon,[]]
```

```
getTag(f) == ((f:Rep).tag):L Record(gen:I,exp:NNI)

new(m:I,mon:Mon):$ ==
  lTag.rest:=cons([m,mon]$Rec,lTag.rest)
  coet(m)

delete(m:I):Void ==
  l1:=lTag
  l2:=lTag.rest
  while l2 ¬ = [] repeat
    (ii:=l2.first.index) < m => return void()
    ii = m =>
      l1.rest:=l2.rest
      return void()
    l1:=l1.rest
    l2:=l2.rest
  void()

-- Pour tout remettre a zero avant un nouveau calcul
clearAll == lTag.rest:=nil()$LRec

lisTag() == [bouzins.index for bouzins in lTag.rest]

coerce(f): E ==
  f.tag = [] => (f.val)::E
  f.val = 0 =>
    mkNary("+"::E, [sub(t) for t in f.tag])
  mkBinary("+"::E,(f.val)::E,
    mkNary("+"::E, [sub(t) for t in f.tag]))

f = g ==
  f.val = g.val =>
    f.tag =$Tag g.tag => true
    false
  false

0 == [0$Mon,[]]

purTag?(f) == f.val = 0

eval1(lrec:Tag):Mon ==
  resu:Mon:= 0
  for term in lrec repeat
    resu:=resu + term.exp * getVal(term.gen)
  resu

eval(f:$):Mon ==
  f.val + eval1(f.tag)

f + g ==
  [f.val + g.val,plus(f.tag,g.tag)]

f - g ==
  (momo:= f.val - g.val) case "failed" => "failed"
  (tata:= moins(f.tag, g.tag)) case "failed" => "failed"
  [momo::Mon,tata::Tag]

n * f ==
```

```
      [n * f.val, mult(n, f.tag)]

  -- On commence par trier en evaluant,
  -- puis en comparant les parties "en x",
  -- enfin on rafine par l'ordre lexicographique inverse.
    f < g ==
      eval(f) < eval(g) => true
      eval(f) > eval(g) => false
      f.val < g.val => true
      f.val > g.val => false
      less(f.tag, g.tag) => true
      false

    min(f,g) ==
      f < g => f
      g


    max(f,g) ==
      g < f => f
      g

    sup(f,g) == [sup(f.val,g.val),sup1(f.tag,g.tag)]
```

2. STANDMON

Ce package implante le calcul de la base standard d'un idéal monomial, qui est une sorte de "base standard de monoïde". L'algorithme utilisé est celui de Gebauer et Moeller du domaine public de Scratchpad II, simplifié pour la circonstance.

```
)abb package STANDMON StandardBasisForMonoid

StandardBasisForMonoid(Mon): Exportation == Production where

  Mon: OrderedAbelianMonoidSup
  NNI ==> NonNegativeInteger
  I ==> Integer
  E ==> Expression
  L ==> List
  MOFAM ==> MutableOrderedFreeAbelianMonoid(Mon)
  Prod ==> Record(fir:MOFAM,sec:MOFAM)
  Syz ==> Record(ltij:MOFAM,eli:Prod,elj:Prod)
  LREC ==> L Record(gen: I, exp: NNI)
  OUT ==> OutputPackage

  Exportation == with
  -- fonctions principales
    begin: (L Mon,Mon) - > L LREC
    continue: (Mon, I) - > L LREC
    exprime: Mon - > LREC
  -- fonctions annexes
    cont: () - > L LREC
    isRed1: L LREC - > I
    isRed: Prod - > Boolean
```

```
    evSyz: Syz - > Prod
    red1: (MOFAM,Prod) - > MOFAM
    red: MOFAM - > MOFAM
    reduire: Prod - > Prod
    genListSyz: () - > Void
    actualiseListSyz: Prod - > L Syz
    genBaseStan: () - > Void
    actualiseBaseStan: Prod - > L Prod
    nettoieBaseStan: L I - > L Prod
    redMemberP: (Prod, Prod) - > MOFAM
    redMemberS: (Syz, Prod) - > MOFAM
    nettoieListSyz: L I - > L Syz
    abs: (Prod,L Prod) - > L Prod
    abs2: (MOFAM, L Prod) - > L Prod
    mkProd: (Mon, I) - > Prod
    crit1: (Prod, Prod) - > Boolean
    crit2: (Syz, Syz) - > Boolean
    crit3: (Syz, Prod) - > Boolean
    appliqueCrit2:(Syz, L Syz) - > L Syz
    appliqueCrit3:(L Syz, Prod) - > L Syz
    merge: (L Syz, L Syz) - > L Syz
    nEv: () - > I

  Production == add
  -- Types
  -- Variables globales
    zerProd:Prod:=[0$MOFAM,0$MOFAM]$Prod
    LProd:= L Prod
    baseStan: LProd:=[zerProd] -- avec une fausse tete
    LSyz:= L Syz
    listSyz:LSyz:=
      [[0$MOFAM,zerProd,zerProd]$Syz] -- avec une fausse tete
    nEvit:L I:=[0]
    monMax:L Mon:=[0]

  -- Pour commencer un calcul de base standard
    begin(lmon,mon) ==
      monMax.0:= mon
      nEvit.0:=0
      clearAll()$MOFAM
      leq:L Prod:= [ ]
      num:I:=£ lmon
      for mon in lmon repeat
        leq:=cons(mkProd(mon,num),leq)
        num:=num - 1
      leq:= sort(leq, £ 1.fir > £ 2.fir)
      baseStan.rest:=[leq.first]
      listSyz.rest:=nil()$LSyz
      for eq in leq.rest repeat
        actualiseListSyz(eq)
        actualiseBaseStan(eq)
      cont()


  -- Utilise par BASECAN pour reprendre le calcul de la base standard
  -- en rajoutant eventuellement un generateur
    continue(mon, i) ==
      if i ¬ = 0 then
        eq:= mkProd(mon, i)
```

```
      actualiseListSyz(eq)
      actualiseBaseStan(eq)
    cont()

  mkProd(mon, i) == [coev(mon)$MOFAM, new(i, mon)$MOFAM]$Prod

  redMemberS(syz,redu) ==
    syz.eli.sec:= red1(syz.eli.sec,redu)
    syz.elj.sec:= red1(syz.elj.sec,redu)

  redMemberP(eq,redu) ==
    eq.sec:= red1(eq.sec,redu)

-- Pour continuer le calcul de base standard
  cont() ==
    while (tls:=listSyz.rest) ¬ = [] repeat
      tlsf:=tls.first
      if monMax.0 ¬ = 0 then
        -- s'il n'y a plus de superposition
        eval tlsf.ltij > monMax.0 => return []
      eq1:= evSyz tlsf
      listSyz.rest:= tls.rest
      (redu:=reduire eq1).fir = 0 => "au suivant"
      actualiseListSyz(redu)
      actualiseBaseStan(redu)
      stan:=baseStan.rest
      purTag? redu.fir =>
        purTag? tlsf.ltij =>
          nEvit.0:= nEvit.0 + 1
        -- retourne une superpositio a BASECAN
        return [getTag redu.fir,getTag redu.sec]
    []

  isRed1(sup) ==
    l1:=sup.0
    £ l1 = 1 and l1.first.exp = 1 => l1.first.gen
    0

  isRed(redu) ==
    purTag? redu.fir and _
    isRed1 [getTag redu.fir, getTag redu.sec] ¬ = 0

  evSyz(syz) ==
    ele1:= (syz.ltij - syz.eli.fir)::MOFAM + syz.eli.sec
    ele2:= (syz.ltij - syz.elj.fir)::MOFAM + syz.elj.sec
    ele1 = ele2 => zerProd
    ele1 < ele2 => [ele2, ele1]$Prod
    [ele1, ele2]$Prod

  red1(mof,eq1) ==
    mof2:= eq1.fir
    mof3:= eq1.sec
    while (diff:= mof - mof2) case MOFAM repeat
      mof:= mof3 + diff::MOFAM
    mof

  red(mof) ==
    st:=baseStan.rest
    omof:=0$MOFAM
```

```
    while mof ¬ = omof repeat
      omof:=mof
      for eq1 in st repeat  mof:= red1(mof,eq1)
    mof

reduire(eq) ==
    (mof:= eq.fir) = (mof2:= eq.sec) => zerProd
    (nmof:=red mof) = mof2 => zerProd
    nmof > mof2 => [nmof, red mof2]
    reduire [mof2, nmof]

actualiseBaseStan(redu) ==
    stan:=baseStan.rest
    baseStan.rest:=abs(redu,baseStan.rest)

abs(redu,std) ==
    std = [] => [redu]
    std.first.fir < redu.fir => cons(std.first,abs(redu,std.rest))
    cons(redu,abs2(redu.fir,std))

abs2(firredu,std) ==
    std = [] => std
    std.first.fir - firredu case MOFAM  => abs2(firredu,std.rest)
    cons(std.first,abs2(firredu,std.rest))


actualiseListSyz(redu) ==
    lS:=listSyz.rest
    std:=baseStan.rest
    newSyz:L Syz:=
      [[sup(x.fir,redu.fir),x,redu]$Syz_
                              for x in std — crit1(x,redu)]
    if newSyz ¬ = [] then
      sort(newSyz,£ 1.ltij < £ 2.ltij)
      newSyz:=appliqueCrit2(newSyz.first,newSyz.rest)
    lS:=appliqueCrit3(lS,redu)
    listSyz.rest:=merge(newSyz,lS)

crit1(eq1,eq2) ==
    sup(eq1.fir,eq2.fir) = eq1.fir + eq2.fir  => false
    true

crit2(syz1,syz2) == syz1.ltij - syz2.ltij case "failed"

crit3(syz,eq) ==
    (eq1:= syz.ltij) - eq.fir case "failed" or _
      sup(syz.eli.fir, eq.fir) = eq1 or sup(syz.elj.fir, eq.fir) = eq1

appliqueCrit2(syz,lS) ==
    lS:=[sy for sy in lS — crit2(sy,syz)]
    lS = [] => [syz]
    cons(syz,appliqueCrit2(lS.first,lS.rest))

appliqueCrit3(lS,redu) == [sy for sy in lS — crit3(sy,redu)]

merge(lS1,lS2) ==
    lS1 = [] => lS2
    lS2 = [] => lS1
    (s1:= lS1.first).ltij < (s2:= lS2.first).ltij =>
```

```
            cons(s1,merge(lS1.rest, lS2))
          cons(s2,merge(lS1, lS2.rest))

    -- utilise par BASECAN pour la reduction
    -- decompose un monome en un produit des monomes de tete
    -- si c'est possible, sinon retourne la liste vide
      exprime(mon) ==
        mof:= coev mon
        purTag? (mof:= red mof) => getTag mof
        []

      nEv() == nEvit.0
```

3. BASECAN

Ce package implante le calcul de la base canonique.

```
-- Address comments and questions to
--   François OLLIVIER
--   Laboratoire d'Informatique de L'X (LIX)
--   Ecole Polytechnique
--   91 128 Palaiseau cedex (FRANCE)
--   BITNET: CFFOLL@FRPOLY11

-- This domain implements a completion procedure for building the
-- canonical basis of a k-subalgebra of k[x1,...xn]. A canonical basis
-- plays the same role for k-subalgebras as standard basis for ideals.
-- The degrees (according to the degree function of spad) of polynomials
-- in the C.B. form a minimal set of generators for the submonoid of
-- N**n consisting of the degrees of all polynomials in the subalgebra.
-- Unfortunately, the C.B. could be infinite, so that the completion
-- procedure never stops. It must be said that, in our implementation,
-- the completion procedure may never stop, even if the C. B. is finite
-- except for degree orderings (which means all orderings sorting
-- polynomials at first according to their total degree). This is
-- no great limitation, for the degree ordering (implemented in NDMP)
-- works well.

-- BaseCanonique takes 1, 2 or 3 arg. The first is a list of polynomials
-- the second a list of string which can be "trace", "info" or
-- "relations".
-- "info" displays the set of polynomials calculated after each
-- reduction.
-- "trace" allows the user to obtain the trace of calculations, and so
-- the expression of pols in the C.B. in function of generating pols.
-- For that issue playBack() after having computed the C.B.
-- "relations" allows the user to get relations between pols in the C.B.
-- For that issue baseStandard after calculating the C.B. (the given
-- polynomials form a standard basis for some order).
-- The third arg is a pol, no superposition will be computed if its
-- degree is greater that the degree of that pol, exept if it is 0
-- We consider the multidegree with current ordering (not totalDegree
-- which is unfortunately not implemented!?).
-- reduire gives the reduction of a pol with respect to the C.B. It is 0
-- iff the pols belongs to the subalgebra.
-- reduireExpr gives the reduction and its expression from pols in the
```

```
-- C.B.
-- Nota: pols in the C.B. are numered according to their order of
-- appearance. Issue getPol(n) to know which has number n.
--
-- For using this you need MOFAM, which represent a monoid, and
-- STANDMON, which calculates Standard basis on a monoid in order to
-- find superpositions.

-- % BaseCanonique
)abb package BASECAN BaseCanonique

BaseCanonique(k,kAlg,Mon): Exportation == Production where
   k: Field
   Mon: OrderedAbelianMonoidSup
   kAlg: GeneralPolynomial(k,Mon)
   Pol ==> Polynomial k
   I ==> Integer
   NNI ==> NonNegativeInteger
   E ==> Expression
   SE ==> SortedExpressions
   S ==> String
   L ==> List
   MOFAM ==> MutableOrderedFreeAbelianMonoid Mon
   STAND ==> StandardBasisForMonoid Mon
   REC ==> Record(gen: I, exp: NNI)
   LREC ==> L Record(gen: I, exp: NNI)
   RecTrace ==> Record(index: I, sup: L LREC)
   RecExpr ==> Record(index:I, exp: Pol)
   RecRed ==> Record(po: kAlg, exp: Pol)
   OUT ==> OutputPackage


   Exportation == with
      baseCanonique: L kAlg - > L kAlg
      baseCanonique: (L kAlg, L S) - > L kAlg
      baseCanonique: (L kAlg, L S, kAlg) - > L kAlg
      reduire: kAlg - > kAlg
      reduireExpr: kAlg - > RecRed
      baseStandard: () - > L Pol
      playBack: () - > L RecExpr
   -- fonctions annexes (destinees a devenir locales)
      conforme: (L I, L LREC) - > Boolean
      reduire2: RecRed - > RecRed
      getPol: I - > kAlg
      mkMonic: kAlg - > I
      mkMonic2: RecRed - > Pol
      delConst: kAlg - > kAlg
      eval: LREC - > kAlg
      calcSup: L LREC - > I
      calcSup2: L LREC - > Pol
      calcSup3: L LREC - > Pol
      num: () - > I
      delPol: I - > I
      redReductum: kAlg - > kAlg
      redReductum2: RecExpr - > RecExpr
      formExpr: LREC - > Pol
      findRecExpr: (L RecExpr, I) - > RecExpr
```

```
Production == add
-- variables globales
   numPols:L I:=[0]
   nR0:L I:=[0]
   registre:= Record(index: I,member: kAlg)
   LReg:= L registre
   listPols:L registre:= [[0,0]$registre] -- avec une fausse tete
   saveBaseCan:L kAlg:= [0$kAlg] -- idem
   listTrace:L RecTrace:= [[0,nil()$L(LREC)]$RecTrace] -- idem
   listRelat:L L LREC:= [nil()$L(LREC)] -- idem
   savePols:L kAlg:= [0] -- idem
   string1:S:= "Nombre de superpositions reduites a 0 :"
   string2:S:= "Nombre de superpositions evitees par le critere 3 :"
   sTr:S:= "trace"
   sInf:S:= "info"
   sRel:S:= "relations"
   flagI:L Boolean:= [false]
   flagT:L Boolean:= [false]
   flagR:L Boolean:= [false]
   maxDeg:L Mon:= [0]
   monMax:L Mon:= [0]


-- implantation des fonctions principales
   baseCanonique(lpol) == baseCanonique(lpol,[],0)

   baseCanonique(lpol,lst) == baseCanonique(lpol,lst,0)

   baseCanonique(lpol,lst,pdm) ==
     monMax.0:= (mon:= degree pdm)
     nR0.0:= 0
     numPols.0:=0
     listPols.rest:=nil()$LReg
     listTrace.rest:=nil()$L(RecTrace)
     listRelat.rest:=nil()$L(L(LREC))
     if member(sInf,lst) then flagI.0:= true
       else flagI.0:= false
     if member(sTr,lst) then flagT.0:= true
       else flagT.0:= false
     if member(sRel,lst) then flagR.0:= true
       else flagR.0:= false
   -- et non pas   :=[] comme on penserait pouvoir l'ecrire...
     lpol:=sort(lpol,degree £ 1 > degree £ 2)
     if flagT.0 then savePols.rest:= lpol
     for pol in lpol repeat mkMonic delConst pol
     -- initialise le calcul de la base standard
     superposition:L LREC:=
       begin([degree reg.member for reg in listPols.rest], mon)$STAND
     -- boucle principale
     while superposition ¬ = [] repeat
       -- a chaque superposition,
       i:= calcSup superposition
       if i = 0 then
            nR0.0:= nR0.0 + 1
            if flagR.0 then listRelat.rest:=
              cons(superposition,listRelat.rest)
          else if flagT.0 then
            listTrace.rest:=
              cons([i, superposition]$RecTrace, listTrace.rest)
```

```
        -- si c'est la reduction d'un polynome, on le supprime
        delPol isRed1(superposition)$STAND
        if flagI.0 then
          baseCan:=[reg.member for reg in listPols.rest]
          output(baseCan::E)$OUT
        -- on itere le calcul de la base standard
        -- pour chercher une nouvelle superposition
        superposition:=continue(degree getPol i, i)$STAND
      baseCan:=
        [(reg.member:=redReductum reg.member) for reg in listPols.rest]
      if flagT.0 then listTrace.rest:= nreverse listTrace.rest
      if flagI.0 then
        output(string1, (nR0.0)::E)$OUT
        output(string2, (nEv()$STAND)::E)$OUT
        if flagT.0 then output((listTrace.rest)::E)$OUT
      baseCan:= sort(baseCan, degree £ 1 < degree £ 2)
      if flagR.0 then saveBaseCan.rest:= baseCan
      baseCan


-- Pour reduire un polynome par la base canonique
    reduire pol ==
      pol = 0 => 0
      lrec:= exprime(degree pol)$STAND
      lrec ¬ = [] => reduire (pol - lc pol * eval lrec)
      monom(degree pol, lc pol) + reduire red pol

-- pour le reduire en calculant son expression en fonction des generateurs
    reduireExpr pol ==
      recred:=reduire2 [pol,0]$RecRed
      recred.exp:= -recred.exp
      recred

-- Calcul d'une base standard de l'ideal des relations
    baseStandard() ==
      ¬ flagR.0 => error "use baseCanonique with string:= relations—"
      listPols.rest:=nil()
      numPols.0:=0
      lpol:= reverse saveBaseCan.rest
      for pol in lpol repeat mkMonic pol
      superposition:L LREC:=
        begin([degree pol for pol in saveBaseCan.rest], monMax.0)$STAND
      lRel:L Pol:= []
      -- On reduit toutes les superpositions
      while superposition ¬ = [] repeat
        -- On recupere la liste des relations evidentes
        lRel:= cons(calcSup3 superposition, lRel)
        superposition:= continue(0$Mon,0$I)$STAND
      nreverse lRel

-- Reprend le calcul de la base canonique
-- et exprime les polynomes de la base en fonction des generateurs
    playBack() ==
      ¬ flagT.0 => error "Sorry boy, there's nothing to play back—"
      numPols.0:=0
      listPols.rest:=nil()$LReg
      pb:L(RecExpr):=[]
      lpol:= [pol for pol in savePols.rest — degree pol > 0]
      numer:=£ lpol
```

```
    for pol in lpol for i in 1..numer repeat
      expr:Pol:=
        expr1:= varPol(mkBinary("SUB","p",i::E)::SE)
        (expr1 - coef(pol,0)::Pol) / lc pol
      pb:=cons([i,expr],pb)
    for pol in lpol repeat mkMonic delConst pol
    lpol2:=reverse lpol
    superposition:L LREC:=
      begin([degree pol for pol in lpol2], monMax.0)$STAND
    for tra in listTrace.rest repeat
      j:I:= tra.index
      while superposition ¬ = tra.sup repeat
        delPol isRed1(superposition)$STAND
        superposition:=continue(0$Mon, 0$I)$STAND
      pb:=cons([j, calcSup2 tra.sup],pb)
      delPol isRed1(superposition)$STAND
      baseCan:=[reg.member for reg in listPols.rest]
      output(baseCan::E)$OUT
      superposition:=continue(degree getPol j, j)$STAND
    while superposition ¬ = [] repeat
      delPol isRed1(superposition)$STAND
      superposition:=continue(0$Mon, 0$I)$STAND
    for reg in listPols.rest repeat
      redReductum2 findRecExpr(pb, reg.index)
    nreverse pb

-- implantation des fonctions annexes
  conforme(lind,super) ==
    for lrec in super repeat
      for rec in lrec repeat
        ¬ member(rec.gen,lind) => return false
    true

  reduire2 polExpr  ==
    pol:= polExpr.po
    expr:= polExpr.exp
    pol = 0 => polExpr
    lrec:= exprime(degree pol)$STAND
    lrec ¬ = [] =>
      expr:= expr - lc pol * formExpr lrec
      reduire2 [pol - lc pol * eval lrec, expr]$RecRed
    [monom(degree pol, lc pol) +-
      (suite:= reduire2 [red pol,expr]$RecRed).po, suite.exp]$RecRed

-- Rend le reste de la reduction unitaire et complete la base
-- canonique s'il est non nul. Retourne l'indice du nouveau polynome.
  mkMonic(pol) ==
    pol = 0 => 0
    listPols.rest:=
      reg:=[numPols.0 := numPols.0 + 1, pol / lc pol]$registre
      cons(reg,listPols.rest)
    numPols.0

  mkMonic2(polexpr) ==
    pol:= polexpr.po
    expr:= polexpr.exp
    expr:= expr / lc pol
    listPols.rest:=
      reg:=[numPols.0 := numPols.0 + 1, pol / lc pol]$registre
```

```
        cons(reg,listPols.rest)
      expr

    getPol(i) ==
      i = 0 => 0
      listP:=listPols.rest
      for reg in listP repeat
        reg.index = i => return reg.member
      error "can't find it"

    delPol(j) ==
      j = 0 => j
      listP:=listPols
      while listP.rest ¬ = [] repeat
        if listP.rest.first.index = j then
          polreduit:=listP.rest.first.member
          listP.rest:= listP.rest.rest
          return j
        listP:=listP.rest
      error "can't find it"

    num() == numPols.0

    eval(lrec) ==
      pol:= 1$kAlg
      for rec in lrec repeat
        pol:= pol * getPol(rec.gen)**(rec.exp)
      pol

    delConst(pol) == pol - coef(pol,0$Mon)::kAlg

    formExpr lrec ==
      resExpr:= 1$Pol
      for rec in lrec repeat
        resExpr:= resExpr *_
            varPol(mkBinary("SUB"::E,"p"::E,(rec.gen)::E)::SE)**rec.exp
      resExpr

-- Reduit le S-polynome associe a une superposition.
    calcSup(llrec) ==
      pol1:= eval llrec.0
      pol2:= eval llrec.1
      mkMonic reduire (pol1 - pol2)

    calcSup2 llrec ==
      pol1:= eval llrec.0
      pol2:= eval llrec.1
      expr:Pol:= formExpr llrec.0 - formExpr llrec.1
      mkMonic2 reduire2 [pol1 - pol2, expr]$RecRed

-- Calcule la relation evidente associee a une superposition
    calcSup3 llrec ==
      pol1:= eval llrec.0
      pol2:= eval llrec.1
      expr:Pol:= formExpr llrec.0 - formExpr llrec.1
      (reduire2 [pol1 - pol2, expr]$RecRed).exp

    redReductum pol ==
      monom(degree pol, lc pol) + reduire red pol
```

```
redReductum2 recexpr ==
   expression:= recexpr.exp
   i:= recexpr.index
   recexpr.exp:= reduire2([red getPol(i),expression]$RecRed).exp
   recexpr

findRecExpr(pb,i) ==
   for recexpr in pb repeat
      recexpr.index = i => return recexpr
   error "can't find it"
```