$SPAD/src/algebra fspace.spad

Manuel Bronstein

October 2, 2018

Abstract

Contents

1 category ES ExpressionSpace **3**

2 ES.lsp BOOTSTRAP **10**

3 ES-.lsp BOOTSTRAP **13**

4 package ES1 ExpressionSpaceFunctions1 **30**

5 package ES2 ExpressionSpaceFunctions2 **31**

6 category FS FunctionSpace **32**

7 package FS2 FunctionSpaceFunctions2 **46**

8 License **47**

1 category ES ExpressionSpace

⟨*category ES ExpressionSpace*⟩≡

```
)abbrev category ES ExpressionSpace
++ Category for domains on which operators can be applied
++ Author: Manuel Bronstein
++ Date Created: 22 March 1988
++ Date Last Updated: 27 May 1994
++ Description:
++ An expression space is a set which is closed under certain operators;
++ Keywords: operator, kernel, expression, space.
ExpressionSpace(): Category == Defn where
  N   ==> NonNegativeInteger
  K   ==> Kernel %
  OP  ==> BasicOperator
  SY  ==> Symbol
  PAREN  ==> "%paren"::SY
  BOX    ==> "%box"::SY
  DUMMYVAR ==> "%dummyVar"

  Defn ==> Join(OrderedSet, RetractableTo K,
              InnerEvalable(K, %), Evalable %) with
    elt          : (OP, %) -> %
      ++ elt(op,x) or op(x) applies the unary operator op to x.
    elt          : (OP, %, %) -> %
      ++ elt(op,x,y) or op(x, y) applies the binary operator op to x and y.
    elt          : (OP, %, %, %) -> %
      ++ elt(op,x,y,z) or op(x, y, z) applies the ternary operator op to x, y and z.
    elt          : (OP, %, %, %, %) -> %
      ++ elt(op,x,y,z,t) or op(x, y, z, t) applies the 4-ary operator op to x, y, z and
    elt          : (OP, List %) -> %
      ++ elt(op,[x1,...,xn]) or op([x1,...,xn]) applies the n-ary operator op to x1,..
    subst        : (%, Equation %) -> %
      ++ subst(f, k = g) replaces the kernel k by g formally in f.
    subst        : (%, List Equation %) -> %
      ++ subst(f, [k1 = g1,...,kn = gn]) replaces the kernels k1,...,kn
      ++ by g1,...,gn formally in f.
    subst        : (%, List K, List %) -> %
      ++ subst(f, [k1...,kn], [g1,...,gn]) replaces the kernels k1,...,kn
      ++ by g1,...,gn formally in f.
    box          : % -> %
      ++ box(f) returns f with a 'box' around it that prevents f from
      ++ being evaluated when operators are applied to it. For example,
      ++ \spad{log(1)} returns 0, but \spad{log(box 1)}
      ++ returns the formal kernel log(1).
    box          : List % -> %
```

```
  ++ box([f1,...,fn]) returns \spad{(f1,...,fn)} with a 'box'
  ++ around them that
  ++ prevents the fi from being evaluated when operators are applied to
  ++ them, and makes them applicable to a unary operator. For example,
  ++ \spad{atan(box [x, 2])} returns the formal kernel \spad{atan(x, 2)}.
paren        : % -> %
  ++ paren(f) returns (f). This prevents f from
  ++ being evaluated when operators are applied to it. For example,
  ++ \spad{log(1)} returns 0, but \spad{log(paren 1)} returns the
  ++ formal kernel log((1)).
paren        : List % -> %
  ++ paren([f1,...,fn]) returns \spad{(f1,...,fn)}. This
  ++ prevents the fi from being evaluated when operators are applied to
  ++ them, and makes them applicable to a unary operator. For example,
  ++ \spad{atan(paren [x, 2])} returns the formal
  ++ kernel \spad{atan((x, 2))}.
distribute   : % -> %
  ++ distribute(f) expands all the kernels in f that are
  ++ formally enclosed by a \spadfunFrom{box}{ExpressionSpace}
  ++ or \spadfunFrom{paren}{ExpressionSpace} expression.
distribute   : (%, %) -> %
  ++ distribute(f, g) expands all the kernels in f that contain g in their
  ++ arguments and that are formally
  ++ enclosed by a \spadfunFrom{box}{ExpressionSpace}
  ++ or a \spadfunFrom{paren}{ExpressionSpace} expression.
height       : %  -> N
  ++ height(f) returns the highest nesting level appearing in f.
  ++ Constants have height 0. Symbols have height 1. For any
  ++ operator op and expressions f1,...,fn, \spad{op(f1,...,fn)} has
  ++ height equal to \spad{1 + max(height(f1),...,height(fn))}.
mainKernel   : %  -> Union(K, "failed")
  ++ mainKernel(f) returns a kernel of f with maximum nesting level, or
  ++ if f has no kernels (i.e. f is a constant).
kernels      : %  -> List K
  ++ kernels(f) returns the list of all the top-level kernels
  ++ appearing in f, but not the ones appearing in the arguments
  ++ of the top-level kernels.
tower        : %  -> List K
  ++ tower(f) returns all the kernels appearing in f, no matter
  ++ what their levels are.
operators    : %  -> List OP
  ++ operators(f) returns all the basic operators appearing in f,
  ++ no matter what their levels are.
operator     : OP -> OP
  ++ operator(op) returns a copy of op with the domain-dependent
  ++ properties appropriate for %.
```

```
belong?      : OP -> Boolean
  ++ belong?(op) tests if % accepts op as applicable to its
  ++ elements.
is?          : (%, OP)     -> Boolean
  ++ is?(x, op) tests if x is a kernel and is its operator is op.
is?          : (%, SY) -> Boolean
  ++ is?(x, s) tests if x is a kernel and is the name of its
  ++ operator is s.
kernel       : (OP, %) -> %
  ++ kernel(op, x) constructs op(x) without evaluating it.
kernel       : (OP, List %) -> %
  ++ kernel(op, [f1,...,fn]) constructs \spad{op(f1,...,fn)} without
  ++ evaluating it.
map          : (% -> %, K) -> %
  ++ map(f, k) returns \spad{op(f(x1),...,f(xn))} where
  ++ \spad{k = op(x1,...,xn)}.
freeOf?      : (%, %)  -> Boolean
  ++ freeOf?(x, y) tests if x does not contain any occurrence of y,
  ++ where y is a single kernel.
freeOf?      : (%, SY) -> Boolean
  ++ freeOf?(x, s) tests if x does not contain any operator
  ++ whose name is s.
eval         : (%, List SY, List(% -> %)) -> %
  ++ eval(x, [s1,...,sm], [f1,...,fm]) replaces
  ++ every \spad{si(a)} in x by \spad{fi(a)} for any \spad{a}.
eval         : (%, List SY, List(List % -> %)) -> %
  ++ eval(x, [s1,...,sm], [f1,...,fm]) replaces
  ++ every \spad{si(a1,...,an)} in x by
  ++ \spad{fi(a1,...,an)} for any \spad{a1},...,\spad{an}.
eval         : (%, SY, List % -> %) -> %
  ++ eval(x, s, f) replaces every \spad{s(a1,..,am)} in x
  ++ by \spad{f(a1,..,am)} for any \spad{a1},...,\spad{am}.
eval         : (%, SY, % -> %) -> %
  ++ eval(x, s, f) replaces every \spad{s(a)} in x by \spad{f(a)}
  ++ for any \spad{a}.
eval         : (%, List OP, List(% -> %)) -> %
  ++ eval(x, [s1,...,sm], [f1,...,fm]) replaces
  ++ every \spad{si(a)} in x by \spad{fi(a)} for any \spad{a}.
eval         : (%, List OP, List(List % -> %)) -> %
  ++ eval(x, [s1,...,sm], [f1,...,fm]) replaces
  ++ every \spad{si(a1,...,an)} in x by
  ++ \spad{fi(a1,...,an)} for any \spad{a1},...,\spad{an}.
eval         : (%, OP, List % -> %) -> %
  ++ eval(x, s, f) replaces every \spad{s(a1,..,am)} in x
  ++ by \spad{f(a1,..,am)} for any \spad{a1},...,\spad{am}.
eval         : (%, OP, % -> %) -> %
```

```
      ++ eval(x, s, f) replaces every \spad{s(a)} in x by \spad{f(a)}
      ++ for any \spad{a}.
    if % has Ring then
      minPoly: K -> SparseUnivariatePolynomial %
        ++ minPoly(k) returns p such that \spad{p(k) = 0}.
      definingPolynomial: % -> %
        ++ definingPolynomial(x) returns an expression p such that
        ++ \spad{p(x) = 0}.
    if % has RetractableTo Integer then
      even?: % -> Boolean
        ++ even? x is true if x is an even integer.
      odd? : % -> Boolean
        ++ odd? x is true if x is an odd integer.

   add

-- the 7 functions not provided are:
--        kernels   minPoly   definingPolynomial
--        coerce:K -> %  eval:(%, List K, List %) -> %
--        subst:(%, List K, List %) -> %
--        eval:(%, List Symbol, List(List % -> %)) -> %

    allKernels: %      -> Set K
    listk     : %      -> List K
    allk      : List % -> Set K
    unwrap    : (List K, %) -> %
    okkernel  : (OP, List %) -> %
    mkKerLists: List Equation % -> Record(lstk: List K, lstv:List %)

    oppren := operator(PAREN)$CommonOperators()
    opbox  := operator(BOX)$CommonOperators()

    box(x:%)     == box [x]
    paren(x:%)   == paren [x]
    belong? op   == op = oppren or op = opbox
    listk f      == parts allKernels f
    tower f      == sort_! listk f
    allk l       == reduce("union", [allKernels f for f in l], {})
    operators f  == [operator k for k in listk f]
    height f     == reduce("max", [height k for k in kernels f], 0)
    freeOf?(x:%, s:SY)       == not member?(s, [name k for k in listk x])
    distribute x == unwrap([k for k in listk x | is?(k, oppren)], x)
    box(l:List %)                  == opbox l
    paren(l:List %)                == oppren l
    freeOf?(x:%, k:%)              == not member?(retract k, listk x)
    kernel(op:OP, arg:%)           == kernel(op, [arg])
```

```
    elt(op:OP, x:%)                   == op [x]
    elt(op:OP, x:%, y:%)              == op [x, y]
    elt(op:OP, x:%, y:%, z:%)         == op [x, y, z]
    elt(op:OP, x:%, y:%, z:%, t:%)    == op [x, y, z, t]
    eval(x:%, s:SY, f:List % -> %)    == eval(x, [s], [f])
    eval(x:%, s:OP, f:List % -> %)    == eval(x, [name s], [f])
    eval(x:%, s:SY, f:% -> %)         == eval(x, [s], [f first #1])
    eval(x:%, s:OP, f:% -> %)         == eval(x, [s], [f first #1])
    subst(x:%, e:Equation %)          == subst(x, [e])

    eval(x:%, ls:List OP, lf:List(% -> %)) ==
      eval(x, ls, [f first #1 for f in lf]$List(List % -> %))

    eval(x:%, ls:List SY, lf:List(% -> %)) ==
      eval(x, ls, [f first #1 for f in lf]$List(List % -> %))

    eval(x:%, ls:List OP, lf:List(List % -> %)) ==
      eval(x, [name s for s in ls]$List(SY), lf)

    map(fn, k) ==
      (l := [fn x for x in argument k]$List(%)) = argument k => k::%
      (operator k) l

    operator op ==
      is?(op, PAREN) => oppren
      is?(op, BOX) => opbox
      error "Unknown operator"

    mainKernel x ==
      empty?(l := kernels x) => "failed"
      n := height(k := first l)
      for kk in rest l repeat
        if height(kk) > n then
          n := height kk
          k := kk
      k

-- takes all the kernels except for the dummy variables, which are second
-- arguments of rootOf's, integrals, sums and products which appear only in
-- their first arguments
    allKernels f ==
      s := brace(l := kernels f)
      for k in l repeat
          t :=
              (u := property(operator k, DUMMYVAR)) case None =>
                  arg := argument k
```

```
              s0  := remove_!(retract(second arg)@K, allKernels first arg)
              arg := rest rest arg
              n   := (u::None) pretend N
              if n > 1 then arg := rest arg
              union(s0, allk arg)
          allk argument k
      s := union(s, t)
  s

kernel(op:OP, args:List %) ==
  not belong? op => error "Unknown operator"
  okkernel(op, args)

okkernel(op, l) ==
  kernel(op, l, 1 + reduce("max", [height f for f in l], 0))$K :: %

elt(op:OP, args:List %) ==
  not belong? op => error "Unknown operator"
  ((u := arity op) case N) and (#args ^= u::N)
                                => error "Wrong number of arguments"
  (v := evaluate(op,args)$BasicOperatorFunctions1(%)) case % => v::%
  okkernel(op, args)

retract f ==
  (k := mainKernel f) case "failed" => error "not a kernel"
  k::K::% ^= f => error "not a kernel"
  k::K

retractIfCan f ==
  (k := mainKernel f) case "failed" => "failed"
  k::K::% ^= f => "failed"
  k

is?(f:%, s:SY) ==
  (k := retractIfCan f) case "failed" => false
  is?(k::K, s)

is?(f:%, op:OP) ==
  (k := retractIfCan f) case "failed" => false
  is?(k::K, op)

unwrap(l, x) ==
  for k in reverse_! l repeat
    x := eval(x, k, first argument k)
  x
```

```
distribute(x, y) ==
  ky := retract y
  unwrap([k for k in listk x |
          is?(k, "%paren"::SY) and member?(ky, listk(k::%))], x)

-- in case of conflicting substitutions e.g. [x = a, x = b],
-- the first one prevails.
-- this is not part of the semantics of the function, but just
-- a feature of this implementation.
eval(f:%, leq:List Equation %) ==
  rec := mkKerLists leq
  eval(f, rec.lstk, rec.lstv)

subst(f:%, leq:List Equation %) ==
  rec := mkKerLists leq
  subst(f, rec.lstk, rec.lstv)

mkKerLists leq ==
  lk := empty()$List(K)
  lv := empty()$List(%)
  for eq in leq repeat
    (k := retractIfCan(lhs eq)@Union(K, "failed")) case "failed" =>
                      error "left hand side must be a single kernel"
    if not member?(k::K, lk) then
      lk := concat(k::K, lk)
      lv := concat(rhs eq, lv)
  [lk, lv]

if % has RetractableTo Integer then
   intpred?: (%, Integer -> Boolean) -> Boolean

   even? x == intpred?(x, even?)
   odd? x  == intpred?(x, odd?)

   intpred?(x, pred?) ==
       (u := retractIfCan(x)@Union(Integer, "failed")) case Integer
              and pred?(u::Integer)
```

2 ES.lsp BOOTSTRAP

ES depends on a chain of files. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **ES** category which we can write into the **MID** directory. We compile the lisp code and copy the **ES.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

⟨*ES.lsp BOOTSTRAP*⟩≡

```
(|/VERSIONCHECK| 2)

(SETQ |ExpressionSpace;AL| (QUOTE NIL))

(DEFUN |ExpressionSpace| NIL
 (LET (#:G82344)
  (COND
   (|ExpressionSpace;AL|)
   (T
    (SETQ |ExpressionSpace;AL| (|ExpressionSpace;|))))))

(DEFUN |ExpressionSpace;| NIL
 (PROG (#1=#:G82342)
  (RETURN
   (PROG1
    (LETT #1#
     (|sublisV|
      (PAIR
       (QUOTE (#2=#:G82340 #3=#:G82341))
       (LIST (QUOTE (|Kernel| |$|)) (QUOTE (|Kernel| |$|))))
      (|Join|
       (|OrderedSet|)
       (|RetractableTo| (QUOTE #2#))
       (|InnerEvalable| (QUOTE #3#) (QUOTE |$|))
       (|Evalable| (QUOTE |$|))
       (|mkCategory|
        (QUOTE |domain|)
        (QUOTE (
         ((|elt| (|$| (|BasicOperator|) |$|)) T)
         ((|elt| (|$| (|BasicOperator|) |$| |$|)) T)
         ((|elt| (|$| (|BasicOperator|) |$| |$| |$|)) T)
         ((|elt| (|$| (|BasicOperator|) |$| |$| |$| |$|)) T)
         ((|elt| (|$| (|BasicOperator|) (|List| |$|))) T)
         ((|subst| (|$| |$| (|Equation| |$|))) T)
         ((|subst| (|$| |$| (|List| (|Equation| |$|)))) T)
         ((|subst| (|$| |$| (|List| (|Kernel| |$|)) (|List| |$|))) T)
```

```
  ((|box| (|$| |$|)) T)
  ((|box| (|$| (|List| |$|))) T)
  ((|paren| (|$| |$|)) T)
  ((|paren| (|$| (|List| |$|))) T)
  ((|distribute| (|$| |$|)) T)
  ((|distribute| (|$| |$| |$|)) T)
  ((|height| ((|NonNegativeInteger|) |$|)) T)
  ((|mainKernel| ((|Union| (|Kernel| |$|) "failed") |$|)) T)
  ((|kernels| ((|List| (|Kernel| |$|)) |$|)) T)
  ((|tower| ((|List| (|Kernel| |$|)) |$|)) T)
  ((|operators| ((|List| (|BasicOperator|)) |$|)) T)
  ((|operator| ((|BasicOperator|) (|BasicOperator|))) T)
  ((|belong?| ((|Boolean|) (|BasicOperator|))) T)
  ((|is?| ((|Boolean|) |$| (|BasicOperator|))) T)
  ((|is?| ((|Boolean|) |$| (|Symbol|))) T)
  ((|kernel| (|$| (|BasicOperator|) |$|)) T)
  ((|kernel| (|$| (|BasicOperator|) (|List| |$|))) T)
  ((|map| (|$| (|Mapping| |$| |$|) (|Kernel| |$|))) T)
  ((|freeOf?| ((|Boolean|) |$| |$|)) T)
  ((|freeOf?| ((|Boolean|) |$| (|Symbol|))) T)
  ((|eval| (|$| |$| (|List| (|Symbol|)) (|List| (|Mapping| |$| |$|))))
   T)
  ((|eval| (|$| |$| (|List| (|Symbol|))
                 (|List| (|Mapping| |$| (|List| |$|)))))
   T)
  ((|eval| (|$| |$| (|Symbol|) (|Mapping| |$| (|List| |$|)))) T)
  ((|eval| (|$| |$| (|Symbol|) (|Mapping| |$| |$|))) T)
  ((|eval| (|$| |$| (|List| (|BasicOperator|))
                         (|List| (|Mapping| |$| |$|))))
   T)
  ((|eval| (|$| |$| (|List| (|BasicOperator|))
                 (|List| (|Mapping| |$| (|List| |$|)))))
   T)
  ((|eval| (|$| |$| (|BasicOperator|) (|Mapping| |$| (|List| |$|)))) T)
  ((|eval| (|$| |$| (|BasicOperator|) (|Mapping| |$| |$|))) T)
  ((|minPoly| ((|SparseUnivariatePolynomial| |$|) (|Kernel| |$|)))
   (|has| |$| (|Ring|)))
  ((|definingPolynomial| (|$| |$|)) (|has| |$| (|Ring|)))
  ((|even?| ((|Boolean|) |$|))
   (|has| |$| (|RetractableTo| (|Integer|))))
  ((|odd?| ((|Boolean|) |$|))
   (|has| |$| (|RetractableTo| (|Integer|))))))
NIL
(QUOTE (
 (|Boolean|)
 (|SparseUnivariatePolynomial| |$|)
```

```
        (|Kernel| |$|)
        (|BasicOperator|)
        (|List| (|BasicOperator|))
        (|List| (|Mapping| |$| (|List| |$|)))
        (|List| (|Mapping| |$| |$|))
        (|Symbol|)
        (|List| (|Symbol|))
        (|List| |$|)
        (|List| (|Kernel| |$|))
        (|NonNegativeInteger|)
        (|List| (|Equation| |$|))
        (|Equation| |$|)))
       NIL)))
     |ExpressionSpace|)
   (SETELT #1# 0 (QUOTE (|ExpressionSpace|)))))))

(MAKEPROP (QUOTE |ExpressionSpace|) (QUOTE NILADIC) T)
```

3 ES-.lsp BOOTSTRAP

ES- depends on **ES**. We need to break this cycle to build the algebra. So we keep a cached copy of the translated **ES-** category which we can write into the **MID** directory. We compile the lisp code and copy the **ES-.o** file to the **OUT** directory. This is eventually forcibly replaced by a recompiled version.

Note that this code is not included in the generated catdef.spad file.

⟨*ES-.lsp BOOTSTRAP*⟩≡

```
(|/VERSIONCHECK| 2)

(DEFUN |ES-;box;2S;1| (|x| |$|) (SPADCALL (LIST |x|) (QREFELT |$| 16)))

(DEFUN |ES-;paren;2S;2| (|x| |$|) (SPADCALL (LIST |x|) (QREFELT |$| 18)))

(DEFUN |ES-;belong?;BoB;3| (|op| |$|)
 (COND
  ((SPADCALL |op| (QREFELT |$| 13) (QREFELT |$| 21)) (QUOTE T))
  ((QUOTE T) (SPADCALL |op| (QREFELT |$| 14) (QREFELT |$| 21)))))

(DEFUN |ES-;listk| (|f| |$|)
 (SPADCALL (|ES-;allKernels| |f| |$|) (QREFELT |$| 25)))

(DEFUN |ES-;tower;SL;5| (|f| |$|)
 (SPADCALL (|ES-;listk| |f| |$|) (QREFELT |$| 26)))

(DEFUN |ES-;allk| (|l| |$|)
 (PROG (#1=#:G82361 |f| #2=#:G82362)
  (RETURN
   (SEQ
    (SPADCALL
     (ELT |$| 30)
     (PROGN
      (LETT #1# NIL |ES-;allk|)
      (SEQ
       (LETT |f| NIL |ES-;allk|)
       (LETT #2# |l| |ES-;allk|)
      G190
       (COND
        ((OR (ATOM #2#)
             (PROGN (LETT |f| (CAR #2#) |ES-;allk|) NIL))
          (GO G191)))
       (SEQ
        (EXIT
         (LETT #1# (CONS (|ES-;allKernels| |f| |$|) #1#) |ES-;allk|)))
```

```
      (LETT #2# (CDR #2#) |ES-;allk|)
      (GO G190)
     G191
      (EXIT (NREVERSE0 #1#))))
    (SPADCALL NIL (QREFELT |$| 29))
    (QREFELT |$| 33))))))

(DEFUN |ES-;operators;SL;7| (|f| |$|)
 (PROG (#1=#:G82365 |k| #2=#:G82366)
  (RETURN
   (SEQ
    (PROGN
     (LETT #1# NIL |ES-;operators;SL;7|)
     (SEQ
      (LETT |k| NIL |ES-;operators;SL;7|)
      (LETT #2# (|ES-;listk| |f| |$|) |ES-;operators;SL;7|)
     G190
      (COND
       ((OR (ATOM #2#)
            (PROGN (LETT |k| (CAR #2#) |ES-;operators;SL;7|) NIL))
         (GO G191)))
      (SEQ
       (EXIT
        (LETT #1#
          (CONS (SPADCALL |k| (QREFELT |$| 35)) #1#)
          |ES-;operators;SL;7|)))
      (LETT #2# (CDR #2#) |ES-;operators;SL;7|)
      (GO G190)
     G191
      (EXIT (NREVERSE0 #1#))))))))

(DEFUN |ES-;height;SNni;8| (|f| |$|)
 (PROG (#1=#:G82371 |k| #2=#:G82372)
  (RETURN
   (SEQ
    (SPADCALL (ELT |$| 41)
     (PROGN
      (LETT #1# NIL |ES-;height;SNni;8|)
      (SEQ
       (LETT |k| NIL |ES-;height;SNni;8|)
       (LETT #2# (SPADCALL |f| (QREFELT |$| 38)) |ES-;height;SNni;8|)
      G190
       (COND
        ((OR (ATOM #2#)
             (PROGN (LETT |k| (CAR #2#) |ES-;height;SNni;8|) NIL))
          (GO G191)))
```

```
      (SEQ
       (EXIT
        (LETT #1#
         (CONS (SPADCALL |k| (QREFELT |$| 40)) #1#)
         |ES-;height;SNni;8|)))
      (LETT #2# (CDR #2#) |ES-;height;SNni;8|)
      (GO G190)
     G191
      (EXIT (NREVERSE0 #1#))))
    0
    (QREFELT |$| 44))))))

(DEFUN |ES-;freeOf?;SSB;9| (|x| |s| |$|)
 (PROG (#1=#:G82377 |k| #2=#:G82378)
  (RETURN
   (SEQ
    (COND
     ((SPADCALL |s|
       (PROGN
        (LETT #1# NIL |ES-;freeOf?;SSB;9|)
        (SEQ
         (LETT |k| NIL |ES-;freeOf?;SSB;9|)
         (LETT #2# (|ES-;listk| |x| |$|) |ES-;freeOf?;SSB;9|)
        G190
         (COND
          ((OR (ATOM #2#)
               (PROGN (LETT |k| (CAR #2#) |ES-;freeOf?;SSB;9|) NIL))
            (GO G191)))
         (SEQ
          (EXIT
           (LETT #1#
             (CONS (SPADCALL |k| (QREFELT |$| 46)) #1#)
             |ES-;freeOf?;SSB;9|)))
         (LETT #2# (CDR #2#) |ES-;freeOf?;SSB;9|)
         (GO G190)
        G191
         (EXIT (NREVERSE0 #1#))))
       (QREFELT |$| 48))
         (QUOTE NIL))
     ((QUOTE T) (QUOTE T)))))))

(DEFUN |ES-;distribute;2S;10| (|x| |$|)
 (PROG (#1=#:G82381 |k| #2=#:G82382)
  (RETURN
   (SEQ
    (|ES-;unwrap|
```

```
     (PROGN
      (LETT #1# NIL |ES-;distribute;2S;10|)
      (SEQ
       (LETT |k| NIL |ES-;distribute;2S;10|)
       (LETT #2# (|ES-;listk| |x| |$|) |ES-;distribute;2S;10|)
      G190
       (COND
        ((OR (ATOM #2#)
             (PROGN (LETT |k| (CAR #2#) |ES-;distribute;2S;10|) NIL))
         (GO G191)))
       (SEQ
        (EXIT
         (COND
          ((SPADCALL |k| (QREFELT |$| 13) (QREFELT |$| 50))
           (LETT #1# (CONS |k| #1#) |ES-;distribute;2S;10|)))))
       (LETT #2# (CDR #2#) |ES-;distribute;2S;10|)
       (GO G190)
       G191
       (EXIT (NREVERSE0 #1#))))
     |x|
     |$|)))))

(DEFUN |ES-;box;LS;11| (|l| |$|)
 (SPADCALL (QREFELT |$| 14) |l| (QREFELT |$| 52)))

(DEFUN |ES-;paren;LS;12| (|l| |$|)
 (SPADCALL (QREFELT |$| 13) |l| (QREFELT |$| 52)))

(DEFUN |ES-;freeOf?;2SB;13| (|x| |k| |$|)
 (COND
  ((SPADCALL
     (SPADCALL |k| (QREFELT |$| 56))
     (|ES-;listk| |x| |$|)
     (QREFELT |$| 57))
    (QUOTE NIL))
  ((QUOTE T) (QUOTE T))))

(DEFUN |ES-;kernel;Bo2S;14| (|op| |arg| |$|)
 (SPADCALL |op| (LIST |arg|) (QREFELT |$| 59)))

(DEFUN |ES-;elt;Bo2S;15| (|op| |x| |$|)
 (SPADCALL |op| (LIST |x|) (QREFELT |$| 52)))

(DEFUN |ES-;elt;Bo3S;16| (|op| |x| |y| |$|)
 (SPADCALL |op| (LIST |x| |y|) (QREFELT |$| 52)))
```

```
(DEFUN |ES-;elt;Bo4S;17| (|op| |x| |y| |z| |$|)
 (SPADCALL |op| (LIST |x| |y| |z|) (QREFELT |$| 52)))

(DEFUN |ES-;elt;Bo5S;18| (|op| |x| |y| |z| |t| |$|)
 (SPADCALL |op| (LIST |x| |y| |z| |t|) (QREFELT |$| 52)))

(DEFUN |ES-;eval;SSMS;19| (|x| |s| |f| |$|)
 (SPADCALL |x| (LIST |s|) (LIST |f|) (QREFELT |$| 67)))

(DEFUN |ES-;eval;SBoMS;20| (|x| |s| |f| |$|)
 (SPADCALL |x|
  (LIST (SPADCALL |s| (QREFELT |$| 69)))
  (LIST |f|)
  (QREFELT |$| 67)))

(DEFUN |ES-;eval;SSMS;21| (|x| |s| |f| |$|)
 (SPADCALL |x|
  (LIST |s|)
  (LIST (CONS (FUNCTION |ES-;eval;SSMS;21!0|) (VECTOR |f| |$|)))
  (QREFELT |$| 67)))

(DEFUN |ES-;eval;SSMS;21!0| (|#1| |$$|)
  (SPADCALL
   (SPADCALL |#1| (QREFELT (QREFELT |$$| 1) 72))
   (QREFELT |$$| 0)))

(DEFUN |ES-;eval;SBoMS;22| (|x| |s| |f| |$|)
 (SPADCALL |x|
  (LIST |s|)
  (LIST (CONS (FUNCTION |ES-;eval;SBoMS;22!0|) (VECTOR |f| |$|)))
  (QREFELT |$| 75)))

(DEFUN |ES-;eval;SBoMS;22!0| (|#1| |$$|)
 (SPADCALL
  (SPADCALL |#1| (QREFELT (QREFELT |$$| 1) 72))
  (QREFELT |$$| 0)))

(DEFUN |ES-;subst;SES;23| (|x| |e| |$|)
 (SPADCALL |x| (LIST |e|) (QREFELT |$| 78)))

(DEFUN |ES-;eval;SLLS;24| (|x| |ls| |lf| |$|)
 (PROG (#1=#:G82403 |f| #2=#:G82404)
  (RETURN
   (SEQ
    (SPADCALL |x| |ls|
     (PROGN
```

```
      (LETT #1# NIL |ES-;eval;SLLS;24|)
      (SEQ
       (LETT |f| NIL |ES-;eval;SLLS;24|)
       (LETT #2# |lf| |ES-;eval;SLLS;24|)
      G190
       (COND
        ((OR (ATOM #2#)
             (PROGN (LETT |f| (CAR #2#) |ES-;eval;SLLS;24|) NIL))
          (GO G191)))
       (SEQ
        (EXIT
         (LETT #1#
          (CONS (CONS (FUNCTION |ES-;eval;SLLS;24!0|) (VECTOR |f| |$|)) #1#)
          |ES-;eval;SLLS;24|)))
       (LETT #2# (CDR #2#) |ES-;eval;SLLS;24|)
       (GO G190)
      G191
       (EXIT (NREVERSE0 #1#))))
     (QREFELT |$| 75))))))

(DEFUN |ES-;eval;SLLS;24!0| (|#1| |$$|)
 (SPADCALL (SPADCALL |#1| (QREFELT (QREFELT |$$| 1) 72)) (QREFELT |$$| 0)))

(DEFUN |ES-;eval;SLLS;25| (|x| |ls| |lf| |$|)
 (PROG (#1=#:G82407 |f| #2=#:G82408)
  (RETURN
   (SEQ
    (SPADCALL |x| |ls|
     (PROGN
      (LETT #1# NIL |ES-;eval;SLLS;25|)
      (SEQ
       (LETT |f| NIL |ES-;eval;SLLS;25|)
       (LETT #2# |lf| |ES-;eval;SLLS;25|)
      G190
       (COND
        ((OR (ATOM #2#)
             (PROGN (LETT |f| (CAR #2#) |ES-;eval;SLLS;25|) NIL))
          (GO G191)))
       (SEQ
        (EXIT
         (LETT #1#
          (CONS (CONS (FUNCTION |ES-;eval;SLLS;25!0|) (VECTOR |f| |$|)) #1#)
          |ES-;eval;SLLS;25|)))
       (LETT #2# (CDR #2#) |ES-;eval;SLLS;25|)
       (GO G190)
      G191
```

```
      (EXIT (NREVERSE0 #1#))))
     (QREFELT |$| 67))))))

(DEFUN |ES-;eval;SLLS;25!0| (|#1| |$$|)
 (SPADCALL (SPADCALL |#1| (QREFELT (QREFELT |$$| 1) 72)) (QREFELT |$$| 0)))

(DEFUN |ES-;eval;SLLS;26| (|x| |ls| |lf| |$|)
 (PROG (#1=#:G82412 |s| #2=#:G82413)
  (RETURN
   (SEQ
    (SPADCALL |x|
     (PROGN
      (LETT #1# NIL |ES-;eval;SLLS;26|)
      (SEQ
       (LETT |s| NIL |ES-;eval;SLLS;26|)
       (LETT #2# |ls| |ES-;eval;SLLS;26|)
      G190
       (COND
        ((OR (ATOM #2#)
             (PROGN (LETT |s| (CAR #2#) |ES-;eval;SLLS;26|) NIL)
         ) (GO G191)))
       (SEQ
        (EXIT
         (LETT #1#
          (CONS (SPADCALL |s| (QREFELT |$| 69)) #1#)
          |ES-;eval;SLLS;26|)))
       (LETT #2# (CDR #2#) |ES-;eval;SLLS;26|)
       (GO G190)
      G191
       (EXIT (NREVERSE0 #1#))))
     |lf|
     (QREFELT |$| 67))))))

(DEFUN |ES-;map;MKS;27| (|fn| |k| |$|)
 (PROG (#1=#:G82428 |x| #2=#:G82429 |l|)
  (RETURN
   (SEQ
    (COND
     ((SPADCALL
       (LETT |l|
        (PROGN
         (LETT #1# NIL |ES-;map;MKS;27|)
         (SEQ
          (LETT |x| NIL |ES-;map;MKS;27|)
          (LETT #2# (SPADCALL |k| (QREFELT |$| 85)) |ES-;map;MKS;27|)
         G190
```

```
          (COND
           ((OR (ATOM #2#)
                (PROGN (LETT |x| (CAR #2#) |ES-;map;MKS;27|) NIL))
             (GO G191)))
          (SEQ
           (EXIT
            (LETT #1# (CONS (SPADCALL |x| |fn|) #1#) |ES-;map;MKS;27|)))
          (LETT #2# (CDR #2#) |ES-;map;MKS;27|)
          (GO G190)
         G191
          (EXIT (NREVERSE0 #1#))))
        |ES-;map;MKS;27|)
       (SPADCALL |k| (QREFELT |$| 85))
       (QREFELT |$| 86))
      (SPADCALL |k| (QREFELT |$| 87)))
      ((QUOTE T)
        (SPADCALL (SPADCALL |k| (QREFELT |$| 35)) |l| (QREFELT |$| 52))))))))

(DEFUN |ES-;operator;2Bo;28| (|op| |$|)
 (COND
  ((SPADCALL |op| (SPADCALL "%paren" (QREFELT |$| 9)) (QREFELT |$| 89))
    (QREFELT |$| 13))
  ((SPADCALL |op| (SPADCALL "%box" (QREFELT |$| 9)) (QREFELT |$| 89))
    (QREFELT |$| 14))
  ((QUOTE T) (|error| "Unknown operator"))))

(DEFUN |ES-;mainKernel;SU;29| (|x| |$|)
 (PROG (|l| |kk| #1=#:G82445 |n| |k|)
  (RETURN
   (SEQ
    (COND
     ((NULL (LETT |l| (SPADCALL |x| (QREFELT |$| 38)) |ES-;mainKernel;SU;29|))
       (CONS 1 "failed"))
     ((QUOTE T)
       (SEQ
        (LETT |n|
         (SPADCALL
          (LETT |k| (|SPADfirst| |l|) |ES-;mainKernel;SU;29|)
          (QREFELT |$| 40))
         |ES-;mainKernel;SU;29|)
        (SEQ
         (LETT |kk| NIL |ES-;mainKernel;SU;29|)
         (LETT #1# (CDR |l|) |ES-;mainKernel;SU;29|)
         G190
         (COND
          ((OR (ATOM #1#)
```

```
                (PROGN (LETT |kk| (CAR #1#) |ES-;mainKernel;SU;29|) NIL))
             (GO G191)))
          (SEQ
           (EXIT
            (COND
             ((|<| |n| (SPADCALL |kk| (QREFELT |$| 40)))
              (SEQ
               (LETT |n|
                (SPADCALL |kk| (QREFELT |$| 40))
                |ES-;mainKernel;SU;29|)
               (EXIT
                (LETT |k| |kk| |ES-;mainKernel;SU;29|)))))))
          (LETT #1# (CDR #1#) |ES-;mainKernel;SU;29|)
          (GO G190)
         G191
          (EXIT NIL))
         (EXIT (CONS 0 |k|)))))))))

(DEFUN |ES-;allKernels| (|f| |$|)
 (PROG (|l| |k| #1=#:G82458 |u| |s0| |n| |arg| |t| |s|)
  (RETURN
   (SEQ
    (LETT |s|
     (SPADCALL
      (LETT |l| (SPADCALL |f| (QREFELT |$| 38)) |ES-;allKernels|)
      (QREFELT |$| 29))
     |ES-;allKernels|)
    (SEQ
     (LETT |k| NIL |ES-;allKernels|)
     (LETT #1# |l| |ES-;allKernels|)
     G190
     (COND
      ((OR (ATOM #1#)
           (PROGN (LETT |k| (CAR #1#) |ES-;allKernels|) NIL))
         (GO G191)))
     (SEQ
      (LETT |t|
       (SEQ
        (LETT |u|
         (SPADCALL
          (SPADCALL |k| (QREFELT |$| 35))
          "%dummyVar"
          (QREFELT |$| 94))
         |ES-;allKernels|)
        (EXIT
         (COND
```

```
          ((QEQCAR |u| 0)
           (SEQ
            (LETT |arg| (SPADCALL |k| (QREFELT |$| 85)) |ES-;allKernels|)
            (LETT |s0|
             (SPADCALL
              (SPADCALL (SPADCALL |arg| (QREFELT |$| 95)) (QREFELT |$| 56))
              (|ES-;allKernels| (|SPADfirst| |arg|) |$|)
              (QREFELT |$| 96))
             |ES-;allKernels|)
            (LETT |arg| (CDR (CDR |arg|)) |ES-;allKernels|)
            (LETT |n| (QCDR |u|) |ES-;allKernels|)
            (COND ((|<| 1 |n|) (LETT |arg| (CDR |arg|) |ES-;allKernels|)))
            (EXIT (SPADCALL |s0| (|ES-;allk| |arg| |$|) (QREFELT |$| 30)))))
          ((QUOTE T)
            (|ES-;allk| (SPADCALL |k| (QREFELT |$| 85)) |$|)))))
       |ES-;allKernels|)
      (EXIT (LETT |s| (SPADCALL |s| |t| (QREFELT |$| 30)) |ES-;allKernels|)))
     (LETT #1# (CDR #1#) |ES-;allKernels|)
     (GO G190)
    G191
     (EXIT NIL))
    (EXIT |s|)))))

(DEFUN |ES-;kernel;BoLS;31| (|op| |args| |$|)
 (COND
  ((NULL (SPADCALL |op| (QREFELT |$| 97))) (|error| "Unknown operator"))
  ((QUOTE T) (|ES-;okkernel| |op| |args| |$|))))

(DEFUN |ES-;okkernel| (|op| |l| |$|)
 (PROG (#1=#:G82465 |f| #2=#:G82466)
  (RETURN
   (SEQ
    (SPADCALL
     (SPADCALL |op| |l|
      (|+| 1
       (SPADCALL
        (ELT |$| 41)
        (PROGN
         (LETT #1# NIL |ES-;okkernel|)
         (SEQ
          (LETT |f| NIL |ES-;okkernel|)
          (LETT #2# |l| |ES-;okkernel|)
         G190
          (COND
           ((OR (ATOM #2#)
                (PROGN (LETT |f| (CAR #2#) |ES-;okkernel|) NIL))
```

```
            (GO G191)))
          (SEQ
           (EXIT
            (LETT #1#
             (CONS (SPADCALL |f| (QREFELT |$| 99)) #1#) |ES-;okkernel|)))
          (LETT #2# (CDR #2#) |ES-;okkernel|)
          (GO G190)
         G191
          (EXIT (NREVERSE0 #1#))))
        0
        (QREFELT |$| 44)))
      (QREFELT |$| 100))
     (QREFELT |$| 87))))))

(DEFUN |ES-;elt;BoLS;33| (|op| |args| |$|)
 (PROG (|u| #1=#:G82482 |v|)
  (RETURN
   (SEQ
    (EXIT
     (COND
      ((NULL (SPADCALL |op| (QREFELT |$| 97))) (|error| "Unknown operator"))
      ((QUOTE T)
       (SEQ
        (SEQ
         (LETT |u| (SPADCALL |op| (QREFELT |$| 102)) |ES-;elt;BoLS;33|)
         (EXIT
          (COND
           ((QEQCAR |u| 0)
            (COND
             ((NULL (EQL (LENGTH |args|) (QCDR |u|)))
              (PROGN
               (LETT #1#
                (|error| "Wrong number of arguments")
                |ES-;elt;BoLS;33|)
               (GO #1#))))))))
        (LETT |v| (SPADCALL |op| |args| (QREFELT |$| 105)) |ES-;elt;BoLS;33|)
        (EXIT
         (COND
          ((QEQCAR |v| 0) (QCDR |v|))
          ((QUOTE T) (|ES-;okkernel| |op| |args| |$|))))))))
    #1#
    (EXIT #1#)))))

(DEFUN |ES-;retract;SK;34| (|f| |$|)
 (PROG (|k|)
  (RETURN
```

```
   (SEQ
    (LETT |k| (SPADCALL |f| (QREFELT |$| 107)) |ES-;retract;SK;34|)
    (EXIT
     (COND
      ((OR (QEQCAR |k| 1)
           (NULL
            (SPADCALL
             (SPADCALL (QCDR |k|) (QREFELT |$| 87))
             |f|
             (QREFELT |$| 108))))
        (|error| "not a kernel"))
      ((QUOTE T) (QCDR |k|))))))))

(DEFUN |ES-;retractIfCan;SU;35| (|f| |$|)
 (PROG (|k|)
  (RETURN
   (SEQ
    (LETT |k| (SPADCALL |f| (QREFELT |$| 107)) |ES-;retractIfCan;SU;35|)
    (EXIT
     (COND
      ((OR (QEQCAR |k| 1)
           (NULL
            (SPADCALL
             (SPADCALL (QCDR |k|) (QREFELT |$| 87))
             |f|
             (QREFELT |$| 108))))
        (CONS 1 "failed"))
      ((QUOTE T) |k|)))))))

(DEFUN |ES-;is?;SSB;36| (|f| |s| |$|)
 (PROG (|k|)
  (RETURN
   (SEQ
    (LETT |k| (SPADCALL |f| (QREFELT |$| 111)) |ES-;is?;SSB;36|)
    (EXIT
     (COND
      ((QEQCAR |k| 1) (QUOTE NIL))
      ((QUOTE T) (SPADCALL (QCDR |k|) |s| (QREFELT |$| 112)))))))))

(DEFUN |ES-;is?;SBoB;37| (|f| |op| |$|)
 (PROG (|k|)
  (RETURN
   (SEQ
    (LETT |k| (SPADCALL |f| (QREFELT |$| 111)) |ES-;is?;SBoB;37|)
    (EXIT
     (COND
```

```
      ((QEQCAR |k| 1) (QUOTE NIL))
      ((QUOTE T) (SPADCALL (QCDR |k|) |op| (QREFELT |$| 50))))))))))

(DEFUN |ES-;unwrap| (|l| |x| |$|)
 (PROG (|k| #1=#:G82507)
  (RETURN
   (SEQ
    (SEQ
     (LETT |k| NIL |ES-;unwrap|)
     (LETT #1# (NREVERSE |l|) |ES-;unwrap|)
    G190
     (COND
      ((OR (ATOM #1#)
           (PROGN (LETT |k| (CAR #1#) |ES-;unwrap|) NIL))
       (GO G191)))
     (SEQ
      (EXIT
       (LETT |x|
        (SPADCALL |x| |k|
         (|SPADfirst| (SPADCALL |k| (QREFELT |$| 85)))
         (QREFELT |$| 115))
        |ES-;unwrap|)))
     (LETT #1# (CDR #1#) |ES-;unwrap|)
     (GO G190)
    G191
    (EXIT NIL))
   (EXIT |x|)))))

(DEFUN |ES-;distribute;3S;39| (|x| |y| |$|)
 (PROG (|ky| #1=#:G82512 |k| #2=#:G82513)
  (RETURN
   (SEQ
    (LETT |ky| (SPADCALL |y| (QREFELT |$| 56)) |ES-;distribute;3S;39|)
    (EXIT
     (|ES-;unwrap|
      (PROGN
       (LETT #1# NIL |ES-;distribute;3S;39|)
       (SEQ
        (LETT |k| NIL |ES-;distribute;3S;39|)
        (LETT #2# (|ES-;listk| |x| |$|) |ES-;distribute;3S;39|)
       G190
        (COND
         ((OR (ATOM #2#)
              (PROGN (LETT |k| (CAR #2#) |ES-;distribute;3S;39|) NIL))
           (GO G191)))
        (SEQ
```

```
          (EXIT
           (COND
            ((COND
              ((SPADCALL |k|
                (SPADCALL "%paren" (QREFELT |$| 9))
                (QREFELT |$| 112))
               (SPADCALL |ky|
                (|ES-;listk| (SPADCALL |k| (QREFELT |$| 87)) |$|)
                (QREFELT |$| 57)))
              ((QUOTE T) (QUOTE NIL)))
              (LETT #1# (CONS |k| #1#) |ES-;distribute;3S;39|)))))
         (LETT #2# (CDR #2#) |ES-;distribute;3S;39|)
         (GO G190)
        G191
         (EXIT (NREVERSE0 #1#))))
       |x|
       |$|))))))

(DEFUN |ES-;eval;SLS;40| (|f| |leq| |$|)
 (PROG (|rec|)
  (RETURN
   (SEQ
    (LETT |rec| (|ES-;mkKerLists| |leq| |$|) |ES-;eval;SLS;40|)
    (EXIT (SPADCALL |f| (QCAR |rec|) (QCDR |rec|) (QREFELT |$| 117)))))))

(DEFUN |ES-;subst;SLS;41| (|f| |leq| |$|)
 (PROG (|rec|)
  (RETURN
   (SEQ
    (LETT |rec| (|ES-;mkKerLists| |leq| |$|) |ES-;subst;SLS;41|)
    (EXIT (SPADCALL |f| (QCAR |rec|) (QCDR |rec|) (QREFELT |$| 119)))))))

(DEFUN |ES-;mkKerLists| (|leq| |$|)
 (PROG (|eq| #1=#:G82530 |k| |lk| |lv|)
  (RETURN
   (SEQ
    (LETT |lk| NIL |ES-;mkKerLists|)
    (LETT |lv| NIL |ES-;mkKerLists|)
    (SEQ
     (LETT |eq| NIL |ES-;mkKerLists|)
     (LETT #1# |leq| |ES-;mkKerLists|)
    G190
     (COND
      ((OR (ATOM #1#)
           (PROGN (LETT |eq| (CAR #1#) |ES-;mkKerLists|) NIL))
        (GO G191)))
```

```
      (SEQ
       (LETT |k|
        (SPADCALL (SPADCALL |eq| (QREFELT |$| 122)) (QREFELT |$| 111))
        |ES-;mkKerLists|)
       (EXIT
        (COND
         ((QEQCAR |k| 1) (|error| "left hand side must be a single kernel"))
         ((NULL (SPADCALL (QCDR |k|) |lk| (QREFELT |$| 57)))
           (SEQ
            (LETT |lk| (CONS (QCDR |k|) |lk|) |ES-;mkKerLists|)
            (EXIT
             (LETT |lv|
               (CONS (SPADCALL |eq| (QREFELT |$| 123)) |lv|)
               |ES-;mkKerLists|)))))))
      (LETT #1# (CDR #1#) |ES-;mkKerLists|)
      (GO G190)
     G191
      (EXIT NIL))
     (EXIT (CONS |lk| |lv|))))))

(DEFUN |ES-;even?;SB;43| (|x| |$|) (|ES-;intpred?| |x| (ELT |$| 125) |$|))

(DEFUN |ES-;odd?;SB;44| (|x| |$|) (|ES-;intpred?| |x| (ELT |$| 127) |$|))

(DEFUN |ES-;intpred?| (|x| |pred?| |$|)
 (PROG (|u|)
  (RETURN
   (SEQ
    (LETT |u| (SPADCALL |x| (QREFELT |$| 130)) |ES-;intpred?|)
    (EXIT
     (COND
      ((QEQCAR |u| 0) (SPADCALL (QCDR |u|) |pred?|))
      ((QUOTE T) (QUOTE NIL))))))))

(DEFUN |ExpressionSpace&| (|#1|)
 (PROG (|DV$1| |dv$| |$| |pv$|)
  (RETURN
   (PROGN
    (LETT |DV$1| (|devaluate| |#1|) . #1=(|ExpressionSpace&|))
    (LETT |dv$| (LIST (QUOTE |ExpressionSpace&|) |DV$1|) . #1#)
    (LETT |$| (GETREFV 131) . #1#)
    (QSETREFV |$| 0 |dv$|)
    (QSETREFV |$| 3
     (LETT |pv$|
      (|buildPredVector| 0 0
       (LIST
```

```
        (|HasCategory| |#1| (QUOTE (|RetractableTo| (|Integer|))))
        (|HasCategory| |#1| (QUOTE (|Ring|)))))
      . #1#))
    (|stuffDomainSlots| |$|)
    (QSETREFV |$| 6 |#1|)
    (QSETREFV |$| 13
     (SPADCALL (SPADCALL "%paren" (QREFELT |$| 9)) (QREFELT |$| 12)))
    (QSETREFV |$| 14
     (SPADCALL (SPADCALL "%box" (QREFELT |$| 9)) (QREFELT |$| 12)))
    (COND
     ((|testBitVector| |pv$| 1)
       (PROGN
        (QSETREFV |$| 126 (CONS (|dispatchFunction| |ES-;even?;SB;43|) |$|))
        (QSETREFV |$| 128 (CONS (|dispatchFunction| |ES-;odd?;SB;44|) |$|)))))
    |$|))))

(MAKEPROP
 (QUOTE |ExpressionSpace&|)
 (QUOTE |infovec|)
 (LIST (QUOTE #(
   NIL NIL NIL NIL NIL NIL (|local| |#1|) (|String|) (|Symbol|) (0 . |coerce|)
   (|BasicOperator|) (|CommonOperators|) (5 . |operator|) (QUOTE |oppren|)
   (QUOTE |opbox|) (|List| |$|) (10 . |box|) |ES-;box;2S;1| (15 . |paren|)
   |ES-;paren;2S;2| (|Boolean|) (20 . |=|) |ES-;belong?;BoB;3| (|List| 34)
   (|Set| 34) (26 . |parts|) (31 . |sort!|) (|List| 55) |ES-;tower;SL;5|
   (36 . |brace|) (41 . |union|) (|Mapping| 24 24 24) (|List| 24)
   (47 . |reduce|) (|Kernel| 6) (54 . |operator|) (|List| 10)
   |ES-;operators;SL;7| (59 . |kernels|) (|NonNegativeInteger|)
   (64 . |height|) (69 . |max|) (|Mapping| 39 39 39) (|List| 39)
   (75 . |reduce|) |ES-;height;SNni;8| (82 . |name|) (|List| 8)
   (87 . |member?|) |ES-;freeOf?;SSB;9| (93 . |is?|) |ES-;distribute;2S;10|
   (99 . |elt|) |ES-;box;LS;11| |ES-;paren;LS;12| (|Kernel| |$|)
   (105 . |retract|) (110 . |member?|) |ES-;freeOf?;2SB;13| (116 . |kernel|)
   |ES-;kernel;Bo2S;14| |ES-;elt;Bo2S;15| |ES-;elt;Bo3S;16| |ES-;elt;Bo4S;17|
   |ES-;elt;Bo5S;18| (|Mapping| |$| 15) (|List| 65) (122 . |eval|)
   |ES-;eval;SSMS;19| (129 . |name|) |ES-;eval;SBoMS;20| (|List| 6)
   (134 . |first|) (|Mapping| |$| |$|) |ES-;eval;SSMS;21| (139 . |eval|)
   |ES-;eval;SBoMS;22| (|List| 79) (146 . |subst|) (|Equation| |$|)
   |ES-;subst;SES;23| (|List| 73) |ES-;eval;SLLS;24| |ES-;eval;SLLS;25|
   |ES-;eval;SLLS;26| (152 . |argument|) (157 . |=|) (163 . |coerce|)
   |ES-;map;MKS;27| (168 . |is?|) |ES-;operator;2Bo;28|
   (|Union| 55 (QUOTE "failed")) |ES-;mainKernel;SU;29|
   (|Union| (|None|) (QUOTE "failed")) (174 . |property|) (180 . |second|)
   (185 . |remove!|) (191 . |belong?|) |ES-;kernel;BoLS;31| (196 . |height|)
   (201 . |kernel|) (|Union| 39 (QUOTE "failed")) (208 . |arity|)
   (|Union| 6 (QUOTE "failed")) (|BasicOperatorFunctions1| 6)
```

```
  (213 . |evaluate|) |ES-;elt;BoLS;33| (219 . |mainKernel|) (224 . |=|)
  |ES-;retract;SK;34| |ES-;retractIfCan;SU;35| (230 . |retractIfCan|)
  (235 . |is?|) |ES-;is?;SSB;36| |ES-;is?;SBoB;37| (241 . |eval|)
  |ES-;distribute;3S;39| (248 . |eval|) |ES-;eval;SLS;40| (255 . |subst|)
  |ES-;subst;SLS;41| (|Equation| 6) (262 . |lhs|) (267 . |rhs|) (|Integer|)
  (272 . |even?|) (277 . |even?|) (282 . |odd?|) (287 . |odd?|)
  (|Union| 124 (QUOTE "failed")) (292 . |retractIfCan|)))
  (QUOTE #(
   |tower| 297 |subst| 302 |retractIfCan| 314 |retract| 319 |paren| 324
   |operators| 334 |operator| 339 |odd?| 344 |map| 349 |mainKernel| 355
   |kernel| 360 |is?| 372 |height| 384 |freeOf?| 389 |even?| 401 |eval| 406
   |elt| 461 |distribute| 497 |box| 508 |belong?| 518))
  (QUOTE NIL)
  (CONS
   (|makeByteWordVec2| 1 (QUOTE NIL))
   (CONS
    (QUOTE #())
    (CONS (QUOTE #())
     (|makeByteWordVec2| 130 (QUOTE
      (1 8 0 7 9 1 11 10 8 12 1 6 0 15 16 1 6 0 15 18 2 10 20 0 0 21 1 24
23 0 25 1 23 0 0 26 1 24 0 23 29 2 24 0 0 0 30 3 32 24 31 0 24 33 1 34 10
0 35 1 6 27 0 38 1 34 39 0 40 2 39 0 0 0 41 3 43 39 42 0 39 44 1 34 8 0 46
2 47 20 8 0 48 2 34 20 0 10 50 2 6 0 10 15 52 1 6 55 0 56 2 23 20 34 0 57 2
6 0 10 15 59 3 6 0 0 47 66 67 1 10 8 0 69 1 71 6 0 72 3 6 0 0 36 66 75 2 6
0 0 77 78 1 34 71 0 85 2 71 20 0 0 86 1 6 0 55 87 2 10 20 0 8 89 2 10 93 0
7 94 1 71 6 0 95 2 24 0 34 0 96 1 6 20 10 97 1 6 39 0 99 3 34 0 10 71 39 100
1 10 101 0 102 2 104 103 10 71 105 1 6 91 0 107 2 6 20 0 0 108 1 6 91 0 111
2 34 20 0 8 112 3 6 0 0 55 0 115 3 6 0 0 27 15 117 3 6 0 0 27 15 119 1 121
6 0 122 1 121 6 0 123 1 124 20 0 125 1 0 20 0 126 1 124 20 0 127 1 0 20 0
128 1 6 129 0 130 1 0 27 0 28 2 0 0 0 77 120 2 0 0 0 79 80 1 0 91 0 110 1
0 55 0 109 1 0 0 0 19 1 0 0 15 54 1 0 36 0 37 1 0 10 10 90 1 0 20 0 128 2
0 0 73 55 88 1 0 91 0 92 2 0 0 10 15 98 2 0 0 10 0 60 2 0 20 0 8 113 2 0 20
0 10 114 1 0 39 0 45 2 0 20 0 8 49 2 0 20 0 0 58 1 0 20 0 126 3 0 0 0 10 73
76 3 0 0 0 36 66 84 3 0 0 0 10 65 70 3 0 0 0 36 81 82 3 0 0 0 8 65 68 3 0 0
0 8 73 74 3 0 0 0 47 81 83 2 0 0 0 77 118 2 0 0 10 15 106 5 0 0 10 0 0 0 0
64 3 0 0 10 0 0 62 4 0 0 10 0 0 0 63 2 0 0 10 0 61 2 0 0 0 0 116 1 0 0 0 51
1 0 0 15 53 1 0 0 0 17 1 0 20 10 22)))))) (QUOTE |lookupComplete|)))
```

4 package ES1 ExpressionSpaceFunctions1

⟨*package ES1 ExpressionSpaceFunctions1*⟩≡

```
)abbrev package ES1 ExpressionSpaceFunctions1
++ Lifting of maps from expression spaces to kernels over them
++ Author: Manuel Bronstein
++ Date Created: 23 March 1988
++ Date Last Updated: 19 April 1991
++ Description:
++   This package allows a map from any expression space into any object
++   to be lifted to a kernel over the expression set, using a given
++   property of the operator of the kernel.
-- should not be exposed
ExpressionSpaceFunctions1(F:ExpressionSpace, S:Type): with
    map: (F -> S, String, Kernel F) -> S
      ++ map(f, p, k) uses the property p of the operator
      ++ of k, in order to lift f and apply it to k.

  == add
    --  prop  contains an evaluation function List S -> S
    map(F2S, prop, k) ==
      args := [F2S x for x in argument k]$List(S)
      (p := property(operator k, prop)) case None =>
                                   ((p::None) pretend (List S -> S)) args
      error "Operator does not have required property"
```

5 package ES2 ExpressionSpaceFunctions2

⟨*package ES2 ExpressionSpaceFunctions2*⟩≡

```
)abbrev package ES2 ExpressionSpaceFunctions2
++ Lifting of maps from expression spaces to kernels over them
++ Author: Manuel Bronstein
++ Date Created: 23 March 1988
++ Date Last Updated: 19 April 1991
++ Description:
++ This package allows a mapping E -> F to be lifted to a kernel over E;
++ This lifting can fail if the operator of the kernel cannot be applied
++ in F; Do not use this package with E = F, since this may
++ drop some properties of the operators.
ExpressionSpaceFunctions2(E:ExpressionSpace, F:ExpressionSpace): with
    map: (E -> F, Kernel E) -> F
      ++ map(f, k) returns \spad{g = op(f(a1),...,f(an))} where
      ++ \spad{k = op(a1,...,an)}.
  == add
    map(f, k) ==
      (operator(operator k)$F) [f x for x in argument k]$List(F)
```

6 category FS FunctionSpace

⟨*category FS FunctionSpace*⟩≡

```
)abbrev category FS FunctionSpace
++ Category for formal functions
++ Author: Manuel Bronstein
++ Date Created: 22 March 1988
++ Date Last Updated: 14 February 1994
++ Description:
++   A space of formal functions with arguments in an arbitrary
++   ordered set.
++ Keywords: operator, kernel, function.
FunctionSpace(R:OrderedSet): Category == Definition where
  OP ==> BasicOperator
  O  ==> OutputForm
  SY ==> Symbol
  N  ==> NonNegativeInteger
  Z  ==> Integer
  K  ==> Kernel %
  Q  ==> Fraction R
  PR ==> Polynomial R
  MP ==> SparseMultivariatePolynomial(R, K)
  QF==> PolynomialCategoryQuotientFunctions(IndexedExponents K,K,R,MP,%)

  ODD  ==> "odd"
  EVEN ==> "even"

  SPECIALDIFF  ==> "%specialDiff"
  SPECIALDISP  ==> "%specialDisp"
  SPECIALEQUAL ==> "%specialEqual"
  SPECIALINPUT ==> "%specialInput"

  Definition ==> Join(ExpressionSpace, RetractableTo SY, Patternable R,
                   FullyPatternMatchable R, FullyRetractableTo R) with
       ground?   : % -> Boolean
         ++ ground?(f) tests if f is an element of R.
       ground    : % -> R
         ++ ground(f) returns f as an element of R.
         ++ An error occurs if f is not an element of R.
       variables : %  -> List SY
         ++ variables(f) returns the list of all the variables of f.
       applyQuote: (SY, %) -> %
         ++ applyQuote(foo, x) returns \spad{'foo(x)}.
       applyQuote: (SY, %, %) -> %
         ++ applyQuote(foo, x, y) returns \spad{'foo(x,y)}.
       applyQuote: (SY, %, %, %) -> %
```

```
  ++ applyQuote(foo, x, y, z) returns \spad{'foo(x,y,z)}.
applyQuote: (SY, %, %, %, %) -> %
  ++ applyQuote(foo, x, y, z, t) returns \spad{'foo(x,y,z,t)}.
applyQuote: (SY, List %) -> %
  ++ applyQuote(foo, [x1,...,xn]) returns \spad{'foo(x1,...,xn)}.
if R has ConvertibleTo InputForm then
  ConvertibleTo InputForm
  eval     : (%, SY) -> %
    ++ eval(f, foo) unquotes all the foo's in f.
  eval     : (%, List SY) -> %
    ++ eval(f, [foo1,...,foon]) unquotes all the \spad{fooi}'s in f.
  eval     : % -> %
    ++ eval(f) unquotes all the quoted operators in f.
  eval     : (%, OP, %, SY) -> %
    ++ eval(x, s, f, y) replaces every \spad{s(a)} in x by \spad{f(y)}
    ++ with \spad{y} replaced by \spad{a} for any \spad{a}.
  eval     : (%, List OP, List %, SY) -> %
    ++ eval(x, [s1,...,sm], [f1,...,fm], y) replaces every
    ++ \spad{si(a)} in x by \spad{fi(y)}
    ++ with \spad{y} replaced by \spad{a} for any \spad{a}.
if R has SemiGroup then
  Monoid
  -- the following line is necessary because of a compiler bug
  "**"   : (%, N) -> %
    ++ x**n returns x * x * x * ... * x (n times).
  isTimes: % -> Union(List %, "failed")
    ++ isTimes(p) returns \spad{[a1,...,an]}
    ++ if \spad{p = a1*...*an} and \spad{n > 1}.
  isExpt : % -> Union(Record(var:K,exponent:Z),"failed")
    ++ isExpt(p) returns \spad{[x, n]} if \spad{p = x**n}
    ++ and \spad{n <> 0}.
if R has Group then Group
if R has AbelianSemiGroup then
  AbelianMonoid
  isPlus: % -> Union(List %, "failed")
    ++ isPlus(p) returns \spad{[m1,...,mn]}
    ++ if \spad{p = m1 +...+ mn} and \spad{n > 1}.
  isMult: % -> Union(Record(coef:Z, var:K),"failed")
    ++ isMult(p) returns \spad{[n, x]} if \spad{p = n * x}
    ++ and \spad{n <> 0}.
if R has AbelianGroup then AbelianGroup
if R has Ring then
  Ring
  RetractableTo PR
  PartialDifferentialRing SY
  FullyLinearlyExplicitRingOver R
```

```
  coerce    : MP -> %
    ++ coerce(p) returns p as an element of %.
  numer     : %  -> MP
    ++ numer(f) returns the
    ++ numerator of f viewed as a polynomial in the kernels over R
    ++ if R is an integral domain. If not, then numer(f) = f viewed
    ++ as a polynomial in the kernels over R.
    -- DO NOT change this meaning of numer!  MB 1/90
  numerator : % -> %
    ++ numerator(f) returns the numerator of \spad{f} converted to %.
  isExpt:(%,OP) -> Union(Record(var:K,exponent:Z),"failed")
    ++ isExpt(p,op) returns \spad{[x, n]} if \spad{p = x**n}
    ++ and \spad{n <> 0} and \spad{x = op(a)}.
  isExpt:(%,SY) -> Union(Record(var:K,exponent:Z),"failed")
    ++ isExpt(p,f) returns \spad{[x, n]} if \spad{p = x**n}
    ++ and \spad{n <> 0} and \spad{x = f(a)}.
  isPower   : % -> Union(Record(val:%,exponent:Z),"failed")
    ++ isPower(p) returns \spad{[x, n]} if \spad{p = x**n}
    ++ and \spad{n <> 0}.
  eval: (%, List SY, List N, List(% -> %)) -> %
    ++ eval(x, [s1,...,sm], [n1,...,nm], [f1,...,fm]) replaces
    ++ every \spad{si(a)**ni} in x by \spad{fi(a)} for any \spad{a}.
  eval: (%, List SY, List N, List(List % -> %)) -> %
    ++ eval(x, [s1,...,sm], [n1,...,nm], [f1,...,fm]) replaces
    ++ every \spad{si(a1,...,an)**ni} in x by \spad{fi(a1,...,an)}
    ++ for any a1,...,am.
  eval: (%, SY, N, List % -> %) -> %
    ++ eval(x, s, n, f) replaces every \spad{s(a1,...,am)**n} in x
    ++ by \spad{f(a1,...,am)} for any a1,...,am.
  eval: (%, SY, N, % -> %) -> %
    ++ eval(x, s, n, f) replaces every \spad{s(a)**n} in x
    ++ by \spad{f(a)} for any \spad{a}.
if R has CharacteristicZero then CharacteristicZero
if R has CharacteristicNonZero then CharacteristicNonZero
if R has CommutativeRing then
  Algebra R
if R has IntegralDomain then
  Field
  RetractableTo Fraction PR
  convert   : Factored % -> %
    ++ convert(f1\^e1 ... fm\^em) returns \spad{(f1)\^e1 ... (fm)\^em}
    ++ as an element of %, using formal kernels
    ++ created using a \spadfunFrom{paren}{ExpressionSpace}.
  denom     : %  -> MP
    ++ denom(f) returns the denominator of f viewed as a
    ++ polynomial in the kernels over R.
```

```
      denominator : % -> %
        ++ denominator(f) returns the denominator of \spad{f} converted to %.
      "/"        : (MP, MP) -> %
        ++ p1/p2 returns the quotient of p1 and p2 as an element of %.
      coerce     : Q  -> %
        ++ coerce(q) returns q as an element of %.
      coerce     : Polynomial Q -> %
        ++ coerce(p) returns p as an element of %.
      coerce     : Fraction Polynomial Q -> %
        ++ coerce(f) returns f as an element of %.
      univariate: (%, K) -> Fraction SparseUnivariatePolynomial %
        ++ univariate(f, k) returns f viewed as a univariate fraction in k.
      if R has RetractableTo Z then RetractableTo Fraction Z
add
 import BasicOperatorFunctions1(%)

 -- these are needed in Ring only, but need to be declared here
 -- because of compiler bug: if they are declared inside the Ring
 -- case, then they are not visible inside the IntegralDomain case.
 smpIsMult : MP -> Union(Record(coef:Z, var:K),"failed")
 smpret    : MP -> Union(PR, "failed")
 smpeval   : (MP, List K, List %) -> %
 smpsubst  : (MP, List K, List %) -> %
 smpderiv  : (MP, SY) -> %
 smpunq    : (MP, List SY, Boolean) -> %
 kerderiv  : (K, SY)  -> %
 kderiv    : K -> List %
 opderiv   : (OP, N) -> List(List % -> %)
 smp2O     : MP -> O
 bestKernel: List K -> K
 worse?    : (K, K) -> Boolean
 diffArg   : (List %, OP, N) -> List %
 substArg  : (OP, List %, Z, %) -> %
 dispdiff  : List % -> Record(name:O, sub:O, arg:List O, level:N)
 ddiff     : List % -> O
 diffEval  : List % -> %
 dfeval    : (List %, K) -> %
 smprep    : (List SY, List N, List(List % -> %), MP) -> %
 diffdiff  : (List %, SY) -> %
 diffdiff0 : (List %, SY, %, K, List %) -> %
 subs      : (% -> %, K) -> %
 symsub    : (SY, Z) -> SY
 kunq      : (K, List SY, Boolean) -> %
 pushunq   : (List SY, List %) -> List %
 notfound  : (K -> %, List K, K) -> %
```

```
equaldiff : (K,K)->Boolean
debugA: (List % ,List %,Boolean) -> Boolean
opdiff := operator("%diff"::SY)$CommonOperators()
opquote := operator("applyQuote"::SY)$CommonOperators

ground? x                == retractIfCan(x)@Union(R,"failed") case R
ground  x                == retract x
coerce(x:SY):%             == kernel(x)@K :: %
retract(x:%):SY            == symbolIfCan(retract(x)@K)::SY
applyQuote(s:SY, x:%)      == applyQuote(s, [x])
applyQuote(s, x, y)        == applyQuote(s, [x, y])
applyQuote(s, x, y, z)     == applyQuote(s, [x, y, z])
applyQuote(s, x, y, z, t)  == applyQuote(s, [x, y, z, t])
applyQuote(s:SY, l:List %) == opquote concat(s::%, l)
belong? op                 == op = opdiff or op = opquote
subs(fn, k) == kernel(operator k,[fn x for x in argument k]$List(%))

operator op ==
  is?(op, "%diff"::SY) => opdiff
  is?(op, "%quote"::SY) => opquote
  error "Unknown operator"

if R has ConvertibleTo InputForm then
  INP==>InputForm
  import MakeUnaryCompiledFunction(%, %, %)
  indiff: List % -> INP
  pint  : List INP-> INP
  differentiand: List % -> %

  differentiand l    == eval(first l, retract(second l)@K, third l)
  pint l  == convert concat(convert("D"::SY)@INP, l)
  indiff l ==
     r2:= convert([convert("::"::SY)@INP,convert(third l)@INP,convert("Symbol"::SY
     pint [convert(differentiand l)@INP, r2]
  eval(f:%, s:SY)             == eval(f, [s])
  eval(f:%, s:OP, g:%, x:SY) == eval(f, [s], [g], x)

  eval(f:%, ls:List OP, lg:List %, x:SY) ==
    eval(f, ls, [compiledFunction(g, x) for g in lg])

  setProperty(opdiff,SPECIALINPUT,indiff@(List % -> InputForm) pretend None)

variables x ==
  l := empty()$List(SY)
  for k in tower x repeat
    if ((s := symbolIfCan k) case SY) then l := concat(s::SY, l)
```

```
      reverse_! l

    retractIfCan(x:%):Union(SY, "failed") ==
      (k := retractIfCan(x)@Union(K,"failed")) case "failed" => "failed"
      symbolIfCan(k::K)

    if R has Ring then
      import UserDefinedPartialOrdering(SY)

-- cannot use new()$Symbol because of possible re-instantiation
      gendiff := "%%0"::SY

      characteristic()    == characteristic()$R
      coerce(k:K):%       == k::MP::%
      symsub(sy, i)       == concat(string sy, convert(i)@String)::SY
      numerator x         == numer(x)::%
      eval(x:%, s:SY, n:N, f:% -> %)     == eval(x,[s],[n],[f first #1])
      eval(x:%, s:SY, n:N, f:List % -> %) == eval(x, [s], [n], [f])
      eval(x:%, l:List SY, f:List(List % -> %)) == eval(x, l, new(#l, 1), f)

      elt(op:OP, args:List %) ==
        unary? op and ((od? := has?(op, ODD)) or has?(op, EVEN)) and
          leadingCoefficient(numer first args) < 0 =>
            x := op(- first args)
            od? => -x
            x
        elt(op, args)$ExpressionSpace_&(%)

      eval(x:%, s:List SY, n:List N, l:List(% -> %)) ==
        eval(x, s, n, [f first #1 for f in l]$List(List % -> %))

      -- op(arg)**m ==> func(arg)**(m quo n) * op(arg)**(m rem n)
      smprep(lop, lexp, lfunc, p) ==
        (v := mainVariable p) case "failed" => p::%
        symbolIfCan(k := v::K) case SY => p::%
        g := (op := operator k)
           (arg := [eval(a,lop,lexp,lfunc) for a in argument k]$List(%))
        q := map(eval(#1::%, lop, lexp, lfunc),
                 univariate(p, k))$SparseUnivariatePolynomialFunctions2(MP, %)
        (n := position(name op, lop)) < minIndex lop => q g
        a:%  := 0
        f    := eval((lfunc.n) arg, lop, lexp, lfunc)
        e    := lexp.n
        while q ^= 0 repeat
          m  := degree q
          qr := divide(m, e)
```

```
    t1 := f ** (qr.quotient)::N
    t2 := g ** (qr.remainder)::N
    a  := a + leadingCoefficient(q) * t1 * t2
    q  := reductum q
  a

dispdiff l ==
  s := second(l)::O
  t := third(l)::O
  a := argument(k := retract(first l)@K)
  is?(k, opdiff) =>
    rec := dispdiff a
    i   := position(s, rec.arg)
    rec.arg.i := t
    [rec.name,
       hconcat(rec.sub, hconcat(","::SY::O, (i+1-minIndex a)::O)),
                  rec.arg, (zero?(rec.level) => 0; rec.level + 1)]
  i   := position(second l, a)
  m   := [x::O for x in a]$List(O)
  m.i := t
  [name(operator k)::O, hconcat(","::SY::O, (i+1-minIndex a)::O),
                                       m, (empty? rest a => 1; 0)]

ddiff l ==
  rec := dispdiff l
  opname :=
    zero?(rec.level) => sub(rec.name, rec.sub)
    differentiate(rec.name, rec.level)
  prefix(opname, rec.arg)

substArg(op, l, i, g) ==
  z := copy l
  z.i := g
  kernel(op, z)


diffdiff(l, x) ==
  f := kernel(opdiff, l)
  diffdiff0(l, x, f, retract(f)@K, empty())

diffdiff0(l, x, expr, kd, done) ==
  op  := operator(k := retract(first l)@K)
  gg  := second l
  u   := third l
  arg := argument k
  ans:% := 0
```

```
    if (not member?(u,done)) and (ans := differentiate(u,x))^=0 then
      ans := ans * kernel(opdiff,
           [subst(expr, [kd], [kernel(opdiff, [first l, gg, gg])]),
                         gg, u])
    done := concat(gg, done)
    is?(k, opdiff) => ans + diffdiff0(arg, x, expr, k, done)
    for i in minIndex arg .. maxIndex arg for b in arg repeat
      if (not member?(b,done)) and (bp:=differentiate(b,x))^=0 then
        g   := symsub(gendiff, i)::%
        ans := ans + bp * kernel(opdiff, [subst(expr, [kd],
         [kernel(opdiff, [substArg(op, arg, i, g), gg, u])]), g, b])
    ans

  dfeval(l, g) ==
    eval(differentiate(first l, symbolIfCan(g)::SY), g, third l)

  diffEval l ==
    k:K
    g := retract(second l)@K
    ((u := retractIfCan(first l)@Union(K, "failed")) case "failed")
      or (u case K and symbolIfCan(k := u::K) case SY) => dfeval(l, g)
    op := operator k
    (ud := derivative op) case "failed" =>
         -- possible trouble
         -- make sure it is a dummy var
         dumm:%:=symsub(gendiff,1)::%
         ss:=subst(l.1,l.2=dumm)
         -- output(nl::OutputForm)$OutputPackage
         -- output("fixed"::OutputForm)$OutputPackage
         nl:=[ss,dumm,l.3]
         kernel(opdiff, nl)
    (n := position(second l,argument k)) < minIndex l =>
          dfeval(l,g)
    d := ud::List(List % -> %)
    eval((d.n)(argument k), g, third l)

  diffArg(l, op, i) ==
    n := i - 1 + minIndex l
    z := copy l
    z.n := g := symsub(gendiff, n)::%
    [kernel(op, z), g, l.n]

  opderiv(op, n) ==
--      one? n =>
    (n = 1) =>
      g := symsub(gendiff, n)::%
```

```
      [kernel(opdiff,[kernel(op, g), g, first #1])]
    [kernel(opdiff, diffArg(#1, op, i)) for i in 1..n]

  kderiv k ==
    zero?(n := #(args := argument k)) => empty()
    op := operator k
    grad :=
      (u := derivative op) case "failed" => opderiv(op, n)
      u::List(List % -> %)
    if #grad ^= n then grad := opderiv(op, n)
    [g args for g in grad]

-- SPECIALDIFF contains a map (List %, Symbol) -> %
-- it is used when the usual chain rule does not apply,
-- for instance with implicit algebraics.
  kerderiv(k, x) ==
    (v := symbolIfCan(k)) case SY =>
      v::SY = x => 1
      0
    (fn := property(operator k, SPECIALDIFF)) case None =>
       ((fn::None) pretend ((List %, SY) -> %)) (argument k, x)
    +/[g * differentiate(y,x) for g in kderiv k for y in argument k]

  smpderiv(p, x) ==
    map(retract differentiate(#1::PR, x), p)::% +
     +/[differentiate(p,k)::% * kerderiv(k, x) for k in variables p]

  coerce(p:PR):% ==
    map(#1::%, #1::%, p)$PolynomialCategoryLifting(
                              IndexedExponents SY, SY, R, PR, %)

  worse?(k1, k2) ==
    (u := less?(name operator k1,name operator k2)) case "failed" =>
      k1 < k2
    u::Boolean

  bestKernel l ==
    empty? rest l => first l
    a := bestKernel rest l
    worse?(first l, a) => a
    first l

  smp2O p ==
    (r:=retractIfCan(p)@Union(R,"failed")) case R =>r::R::OutputForm
    a :=
      userOrdered?() => bestKernel variables p
```

```
        mainVariable(p)::K
      outputForm(map(#1::%, univariate(p,
       a))$SparseUnivariatePolynomialFunctions2(MP, %), a::OutputForm)

    smpsubst(p, lk, lv) ==
      map(match(lk, lv, #1,
          notfound(subs(subst(#1, lk, lv), #1), lk, #1))$ListToMap(K,%),
           #1::%,p)$PolynomialCategoryLifting(IndexedExponents K,K,R,MP,%)

    smpeval(p, lk, lv) ==
      map(match(lk, lv, #1,
          notfound(map(eval(#1, lk, lv), #1), lk, #1))$ListToMap(K,%),
           #1::%,p)$PolynomialCategoryLifting(IndexedExponents K,K,R,MP,%)

-- this is called on k when k is not a member of lk
    notfound(fn, lk, k) ==
      empty? setIntersection(tower(f := k::%), lk) => f
      fn k

    if R has ConvertibleTo InputForm then
      pushunq(l, arg) ==
         empty? l => [eval a for a in arg]
         [eval(a, l) for a in arg]

      kunq(k, l, givenlist?) ==
        givenlist? and empty? l => k::%
        is?(k, opquote) and
          (member?(s:=retract(first argument k)@SY, l) or empty? l) =>
            interpret(convert(concat(convert(s)@InputForm,
              [convert a for a in pushunq(l, rest argument k)
                 ]@List(InputForm)))@InputForm)$InputFormFunctions1(%)
        (operator k) pushunq(l, argument k)

      smpunq(p, l, givenlist?) ==
        givenlist? and empty? l => p::%
        map(kunq(#1, l, givenlist?), #1::%,
          p)$PolynomialCategoryLifting(IndexedExponents K,K,R,MP,%)

    smpret p ==
      "or"/[symbolIfCan(k) case "failed" for k in variables p] =>
        "failed"
      map(symbolIfCan(#1)::SY::PR, #1::PR,
        p)$PolynomialCategoryLifting(IndexedExponents K, K, R, MP, PR)

    isExpt(x:%, op:OP) ==
      (u := isExpt x) case "failed" => "failed"
```

```
    is?((u::Record(var:K, exponent:Z)).var, op) => u
    "failed"

  isExpt(x:%, sy:SY) ==
    (u := isExpt x) case "failed" => "failed"
    is?((u::Record(var:K, exponent:Z)).var, sy) => u
    "failed"

  if R has RetractableTo Z then
      smpIsMult p ==
--          (u := mainVariable p) case K and one? degree(q:=univariate(p,u::K))
        (u := mainVariable p) case K and (degree(q:=univariate(p,u::K))=1)
          and zero?(leadingCoefficient reductum q)
            and ((r:=retractIfCan(leadingCoefficient q)@Union(R,"failed"))
               case R)
                 and (n := retractIfCan(r::R)@Union(Z, "failed")) case Z =>
                   [n::Z, u::K]
        "failed"

  evaluate(opdiff, diffEval)

  debugA(a1,a2,t) ==
     -- uncomment for debugging
     -- output(hconcat [a1::OutputForm,a2::OutputForm,t::OutputForm])$OutputPackag
     t

  equaldiff(k1,k2) ==
    a1:=argument k1
    a2:=argument k2
    -- check the operator
    res:=operator k1 = operator k2
    not res => debugA(a1,a2,res)
    -- check the evaluation point
    res:= (a1.3 = a2.3)
    not res => debugA(a1,a2,res)
    -- check all the arguments
    res:= (a1.1 = a2.1) and (a1.2 = a2.2)
    res => debugA(a1,a2,res)
    -- check the substituted arguments
    (subst(a1.1,[retract(a1.2)@K],[a2.2]) = a2.1) => debugA(a1,a2,true)
    debugA(a1,a2,false)
  setProperty(opdiff,SPECIALEQUAL,
                    equaldiff@((K,K) -> Boolean) pretend None)
  setProperty(opdiff, SPECIALDIFF,
                    diffdiff@((List %, SY) -> %) pretend None)
  setProperty(opdiff, SPECIALDISP,
```

```
                        ddiff@(List % -> OutputForm) pretend None)

if not(R has IntegralDomain) then
  mainKernel x          == mainVariable numer x
  kernels x             == variables numer x
  retract(x:%):R        == retract numer x
  retract(x:%):PR       == smpret(numer x)::PR
  retractIfCan(x:%):Union(R,  "failed") == retract numer x
  retractIfCan(x:%):Union(PR, "failed") == smpret numer x
  eval(x:%, lk:List K, lv:List %)  == smpeval(numer x, lk, lv)
  subst(x:%, lk:List K, lv:List %) == smpsubst(numer x, lk, lv)
  differentiate(x:%, s:SY)         == smpderiv(numer x, s)
  coerce(x:%):OutputForm           == smp2O numer x

  if R has ConvertibleTo InputForm then
    eval(f:%, l:List SY) == smpunq(numer f, l, true)
    eval f               == smpunq(numer f, empty(), false)

  eval(x:%, s:List SY, n:List N, f:List(List % -> %)) ==
    smprep(s, n, f, numer x)

  isPlus x ==
    (u := isPlus numer x) case "failed" => "failed"
    [p::% for p in u::List(MP)]

  isTimes x ==
    (u := isTimes numer x) case "failed" => "failed"
    [p::% for p in u::List(MP)]

  isExpt x ==
    (u := isExpt numer x) case "failed" => "failed"
    r := u::Record(var:K, exponent:NonNegativeInteger)
    [r.var, r.exponent::Z]

  isPower x ==
    (u := isExpt numer x) case "failed" => "failed"
    r := u::Record(var:K, exponent:NonNegativeInteger)
    [r.var::%, r.exponent::Z]

  if R has ConvertibleTo Pattern Z then
    convert(x:%):Pattern(Z) == convert numer x

  if R has ConvertibleTo Pattern Float then
    convert(x:%):Pattern(Float) == convert numer x

  if R has RetractableTo Z then
```

```
        isMult x == smpIsMult numer x

    if R has CommutativeRing then
      r:R * x:% == r::MP::% * x

    if R has IntegralDomain then
      par   : % -> %

      mainKernel x                    == mainVariable(x)$QF
      kernels x                       == variables(x)$QF
      univariate(x:%, k:K)            == univariate(x, k)$QF
      isPlus x                        == isPlus(x)$QF
      isTimes x                       == isTimes(x)$QF
      isExpt x                        == isExpt(x)$QF
      isPower x                       == isPower(x)$QF
      denominator x                   == denom(x)::%
      coerce(q:Q):%                   == (numer q)::MP / (denom q)::MP
      coerce(q:Fraction PR):%         == (numer q)::% / (denom q)::%
      coerce(q:Fraction Polynomial Q) == (numer q)::% / (denom q)::%
      retract(x:%):PR                == retract(retract(x)@Fraction(PR))
      retract(x:%):Fraction(PR) == smpret(numer x)::PR / smpret(denom x)::PR
      retract(x:%):R == (retract(numer x)@R exquo retract(denom x)@R)::R

      coerce(x:%):OutputForm ==
--        one?(denom x) => smp2O numer x
        ((denom x) = 1) => smp2O numer x
        smp2O(numer x) / smp2O(denom x)

      retractIfCan(x:%):Union(R, "failed") ==
        (n := retractIfCan(numer x)@Union(R, "failed")) case "failed" or
          (d := retractIfCan(denom x)@Union(R, "failed")) case "failed"
            or (r := n::R exquo d::R) case "failed" => "failed"
        r::R

      eval(f:%, l:List SY) ==
        smpunq(numer f, l, true) / smpunq(denom f, l, true)

      if R has ConvertibleTo InputForm then
        eval f ==
          smpunq(numer f, empty(), false) / smpunq(denom f, empty(), false)

        eval(x:%, s:List SY, n:List N, f:List(List % -> %)) ==
          smprep(s, n, f, numer x) / smprep(s, n, f, denom x)

      differentiate(f:%, x:SY) ==
        (smpderiv(numer f, x) * denom(f)::% -
```

```
    numer(f)::% * smpderiv(denom f, x))
      / (denom(f)::% ** 2)

eval(x:%, lk:List K, lv:List %) ==
  smpeval(numer x, lk, lv) / smpeval(denom x, lk, lv)

subst(x:%, lk:List K, lv:List %) ==
  smpsubst(numer x, lk, lv) / smpsubst(denom x, lk, lv)

par x ==
  (r := retractIfCan(x)@Union(R, "failed")) case R => x
  paren x

convert(x:Factored %):% ==
  par(unit x) * */[par(f.factor) ** f.exponent for f in factors x]

retractIfCan(x:%):Union(PR, "failed") ==
  (u := retractIfCan(x)@Union(Fraction PR,"failed")) case "failed"
    => "failed"
  retractIfCan(u::Fraction(PR))

retractIfCan(x:%):Union(Fraction PR, "failed") ==
  (n := smpret numer x) case "failed" => "failed"
  (d := smpret denom x) case "failed" => "failed"
  n::PR / d::PR

coerce(p:Polynomial Q):% ==
  map(#1::%, #1::%,
     p)$PolynomialCategoryLifting(IndexedExponents SY, SY,
                                              Q, Polynomial Q, %)

if R has RetractableTo Z then
  coerce(x:Fraction Z):% == numer(x)::MP / denom(x)::MP

  isMult x ==
     (u := smpIsMult numer x) case "failed"
        or (v := retractIfCan(denom x)@Union(R, "failed")) case "failed"
           or (w := retractIfCan(v::R)@Union(Z, "failed")) case "failed"
               => "failed"
     r := u::Record(coef:Z, var:K)
     (q := r.coef exquo w::Z) case "failed" => "failed"
     [q::Z, r.var]

if R has ConvertibleTo Pattern Z then
  convert(x:%):Pattern(Z) == convert(numer x) / convert(denom x)
```

```
    if R has ConvertibleTo Pattern Float then
      convert(x:%):Pattern(Float) ==
        convert(numer x) / convert(denom x)
```

7 package FS2 FunctionSpaceFunctions2

⟨*package FS2 FunctionSpaceFunctions2*⟩≡

```
)abbrev package FS2 FunctionSpaceFunctions2
++ Lifting of maps to function spaces
++ Author: Manuel Bronstein
++ Date Created: 22 March 1988
++ Date Last Updated: 3 May 1994
++ Description:
++   This package allows a mapping R -> S to be lifted to a mapping
++   from a function space over R to a function space over S;
FunctionSpaceFunctions2(R, A, S, B): Exports == Implementation where
  R, S: Join(Ring, OrderedSet)
  A   : FunctionSpace R
  B   : FunctionSpace S

  K  ==> Kernel A
  P  ==> SparseMultivariatePolynomial(R, K)

  Exports ==> with
    map: (R -> S, A) -> B
      ++ map(f, a) applies f to all the constants in R appearing in \spad{a}.

  Implementation ==> add
    smpmap: (R -> S, P) -> B

    smpmap(fn, p) ==
      map(map(map(fn, #1), #1)$ExpressionSpaceFunctions2(A,B),fn(#1)::B,
        p)$PolynomialCategoryLifting(IndexedExponents K, K, R, P, B)

    if R has IntegralDomain then
      if S has IntegralDomain then
        map(f, x) == smpmap(f, numer x) / smpmap(f, denom x)
      else
        map(f, x) == smpmap(f, numer x) * (recip(smpmap(f, denom x))::B)
    else
      map(f, x) == smpmap(f, numer x)
```

8 License

⟨*license*⟩≡

```
--Copyright (c) 1991-2002, The Numerical ALgorithms Group Ltd.
--All rights reserved.
--
--Redistribution and use in source and binary forms, with or without
--modification, are permitted provided that the following conditions are
--met:
--
--    - Redistributions of source code must retain the above copyright
--      notice, this list of conditions and the following disclaimer.
--
--    - Redistributions in binary form must reproduce the above copyright
--      notice, this list of conditions and the following disclaimer in
--      the documentation and/or other materials provided with the
--      distribution.
--
--    - Neither the name of The Numerical ALgorithms Group Ltd. nor the
--      names of its contributors may be used to endorse or promote products
--      derived from this software without specific prior written permission.
--
--THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
--IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
--TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
--PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
--OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
--EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
--PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
--PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
--LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
--NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
--SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

⟨*⟩≡
⟨*license*⟩

```
-- SPAD files for the functional world should be compiled in the
-- following order:
--
--   op  kl  FSPACE  expr funcpkgs
```

⟨*category ES ExpressionSpace*⟩
⟨*package ES1 ExpressionSpaceFunctions1*⟩
⟨*package ES2 ExpressionSpaceFunctions2*⟩
⟨*category FS FunctionSpace*⟩
⟨*package FS2 FunctionSpaceFunctions2*⟩

References

[1] nothing