$SPAD/src/algebra efstruc.spad

Manuel Bronstein, Waldek Hebisch

September 13, 2014

Abstract

Contents

1 package SYMFUNC SymmetricFunctions

⟨*package SYMFUNC SymmetricFunctions*⟩≡

```
)abbrev package SYMFUNC SymmetricFunctions
++ The elementary symmetric functions
++ Author: Manuel Bronstein
++ Date Created: 13 Feb 1989
++ Date Last Updated: 28 Jun 1990
++ Description: Computes all the symmetric functions in n variables.
SymmetricFunctions(R : Ring) : Exports == Implementation where
  UP  ==> SparseUnivariatePolynomial R

  Exports ==> with
    symFunc : List R  -> Vector R
      ++ symFunc([r1, ..., rn]) returns the vector of the
      ++ elementary symmetric functions in the \spad{ri's}:
      ++ \spad{[r1 + ... + rn, r1 r2 + ... + r(n-1) rn, ..., r1 r2 ... rn]}.
    symFunc : (R, PositiveInteger) -> Vector R
      ++ symFunc(r, n) returns the vector of the elementary
      ++ symmetric functions in \spad{[r, r, ..., r]} \spad{n} times.

  Implementation ==> add
    signFix : (UP, NonNegativeInteger) -> Vector R

    symFunc(x, n) == signFix((monomial(1, 1)$UP - x::UP) ^ n, 1 + n)

    symFunc l ==
      signFix(*/[monomial(1, 1)$UP - a::UP for a in l], 1 + #l)

    signFix(p, n) ==
      m := minIndex(v := vectorise(p, n)) + 1
      for i in 0..((#v quo 2) - 1)::NonNegativeInteger repeat
        qsetelt!(v, 2*i + m, - qelt(v, 2*i + m))
      reverse! v
```

2 package TANEXP TangentExpansions

⟨*package TANEXP TangentExpansions*⟩≡

```
)abbrev package TANEXP TangentExpansions
++ Expansions of tangents of sums and quotients
++ Author: Manuel Bronstein
++ Date Created: 13 Feb 1989
++ Date Last Updated: 20 Apr 1990
++ Description: Expands tangents of sums and scalar products.
TangentExpansions(R : Field) : Exports == Implementation where
  PI ==> PositiveInteger
  Z  ==> Integer
  UP ==> SparseUnivariatePolynomial R
  QF ==> Fraction UP

  Exports ==> with
    tanSum : List R -> R
      ++ tanSum([a1, ..., an]) returns \spad{f(a1, ..., an)} such that
      ++ if \spad{ai = tan(ui)} then \spad{f(a1, ..., an) = tan(u1 + ... + un)}.
    tanAn : (R, PI) -> UP
      ++ tanAn(a, n) returns \spad{P(x)} such that
      ++ if \spad{a = tan(u)} then \spad{P(tan(u/n)) = 0}.
    tanNa : (R,  Z) -> R
      ++ tanNa(a, n) returns \spad{f(a)} such that
      ++ if \spad{a = tan(u)} then \spad{f(a) = tan(n * u)}.

  Implementation ==> add
    import from SymmetricFunctions(R)
    import from SymmetricFunctions(UP)

    m1toN : Integer -> Integer
    tanPIa : PI -> QF

    m1toN n      == (odd? n => -1; 1)
    tanAn(a, n) == a * denom(q := tanPIa n) - numer q

    tanNa(a, n) ==
      zero? n => 0
      negative? n => - tanNa(a, -n)
      (numer(t := tanPIa(n::PI)) a) / ((denom t) a)

    tanSum l ==
      m := minIndex(v := symFunc l)
      +/[m1toN(i+1) * v(2*i - 1 + m) for i in 1..(#v quo 2)]
        / +/[m1toN(i) * v(2*i + m) for i in 0..((#v - 1) quo 2)]
```

```
-- tanPIa(n) returns P(a)/Q(a) such that
-- if a = tan(u) then P(a)/Q(a) = tan(n * u);
    tanPIa n ==
      m := minIndex(v := symFunc(monomial(1, 1)$UP, n))
      +/[m1toN(i+1) * v(2*i - 1 + m) for i in 1..(#v quo 2)]
        / +/[m1toN(i) * v(2*i + m) for i in 0..((#v - 1) quo 2)]
```

3 package EFSTRUC ElementaryFunctionStructurePackage

⟨*package EFSTRUC ElementaryFunctionStructurePackage*⟩≡
```
)abbrev package EFSTRUC ElementaryFunctionStructurePackage
++ Risch structure theorem
++ Author: Manuel Bronstein, Waldek Hebisch
++ Date Created: 1987
++ Date Last Updated: 9 October 2006
++ Description:
++   ElementaryFunctionStructurePackage provides functions to test the
++   algebraic independence of various elementary functions, using the
++   Risch structure theorem (real and complex versions).
++   It also provides transformations on elementary functions
++   which are not considered simplifications.
++ Keywords: elementary, function, structure.
ElementaryFunctionStructurePackage(R, F) : Exports == Implementation where
  R : Join(IntegralDomain, Comparable, RetractableTo Integer,
           LinearlyExplicitRingOver Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace R)

  B   ==> Boolean
  N   ==> NonNegativeInteger
  Z   ==> Integer
  Q   ==> Fraction Z
  SY  ==> Symbol
  K   ==> Kernel F
  UP  ==> SparseUnivariatePolynomial F
  SMP ==> SparseMultivariatePolynomial(R, K)
  REC ==> Record(func : F, kers : List K, vals : List F)
  U   ==> Union(vec : Vector Q, func : F, fail : Boolean)
  POWER ==> '%power
  NTHR  ==> 'nthRoot

  Exports ==> with
    normalize : F -> F
      ++ normalize(f) rewrites \spad{f} using the least possible number of
      ++ real algebraically independent kernels.
    normalize : (F, SY) -> F
      ++ normalize(f, x) rewrites \spad{f} using the least possible number of
      ++ real algebraically independent kernels involving \spad{x}.
    rischNormalize : (F, SY) -> REC
      ++ rischNormalize(f, x) returns \spad{[g, [k1, ..., kn], [h1, ..., hn]]}
      ++ such that \spad{g = normalize(f, x)} and each \spad{ki} was
```

```
    ++ rewritten as \spad{hi} during the normalization.
  rischNormalize : (F, List SY) -> REC
    ++ rischNormalize(f, lx) returns \spad{[g, [k1, ..., kn], [h1, ..., hn]]}
    ++ such that \spad{g = normalize(f, lx)} and each \spad{ki} was
    ++ rewritten as \spad{hi} during the normalization.
  realElementary : F -> F
    ++ realElementary(f) rewrites \spad{f} in terms of the 4 fundamental real
    ++ transcendental elementary functions: \spad{log, exp, tan, atan}.
  realLiouvillian : F -> F
    ++ realLiouvillian(f) rewrites \spad{f} elementary kernels of f in
    ++ terms 4 fundamental real elementary functions: \spad{log, exp, tan,
    ++ atan}.  Additionally, it rewrites Liouvillian functions as
    ++ indefinite integrals to support better normalization.
  realLiouvillian : (F, SY) -> F
    ++ realLiouvillian(f, x) rewrites \spad{f} elementary kernels of f in
    ++ terms 4 fundamental real elementary functions: \spad{log, exp, tan,
    ++ atan}.  Additionally, it rewrites Liouvillian functions of x as
    ++ indefinite integrals to support better normalization.
  realElementary : (F, SY) -> F
    ++ realElementary(f, x) rewrites the kernels of \spad{f} involving \spad{x}
    ++ in terms of the 4 fundamental real
    ++ transcendental elementary functions: \spad{log, exp, tan, atan}.
  validExponential : (List K, F, SY) -> Union(F, "failed")
    ++ validExponential([k1, ..., kn], f, x) returns \spad{g} if \spad{exp(f)=g}
    ++ and \spad{g} involves only \spad{k1...kn}, and "failed" otherwise.
  rootNormalize : (F, K) -> F
    ++ rootNormalize(f, k) returns \spad{f} rewriting either \spad{k} which
    ++ must be an nth-root in terms of radicals already in \spad{f}, or some
    ++ radicals in \spad{f} in terms of \spad{k}.
  rmap : (K -> F, F) -> F
    ++ rmap(f, e) rewrites e replacing each kernel k in e by f(k)
  tanQ : (Q, F) -> F
    ++ tanQ(q, a) is a local function with a conditional implementation.
  irootDep : K -> U
    ++ irootDep(k) is a local function with a conditional implementation.

Implementation ==> add
  import from TangentExpansions F
  import from IntegrationTools(R, F)
  import from IntegerLinearDependence F
  import from AlgebraicManipulations(R, F)
  import from InnerCommonDenominator(Z, Q, Vector Z, Vector Q)
  P  ==> SparseMultivariatePolynomial(R, K)

  HTRIG := 'htrig
  TRIG := 'trig
```

```
k2Elem             : (K, List SY) -> F
realElem           : (F, List SY) -> F
rootDep            : (List K, K)     -> U
findQRelation      : (List SY, List F, F) -> U
findRelation       : (List SY, List SY, List K, K) -> U
factdeprel         : (List K, K)     -> U
toR                : (List K, F) -> List K
toY                : List K -> List F
toZ                : List K -> List F
toU                : List K -> List F
toV                : List K -> List F
ktoY               : K  -> F
ktoZ               : K  -> F
ktoU               : K  -> F
ktoV               : K  -> F
gdCoef?            : (Q, Vector Q) -> Boolean
goodCoef           : (Vector Q, List K, SY) ->
                             Union(Record(index:Z, ker:K), "failed")
tanRN              : (Q, K) -> F
localnorm          : F -> F
rooteval           : (F, List K, K, Q) -> REC
logeval            : (F, List K, K, Vector Q) -> REC
expeval            : (F, List K, K, Vector Q) -> REC
taneval            : (F, List K, K, Vector Q) -> REC
ataneval           : (F, List K, K, Vector Q) -> REC
depeval            : (F, List K, K, Vector Q) -> REC
expnosimp          : (F, List K, K, Vector Q, List F, F) -> REC
tannosimp          : (F, List K, K, Vector Q, List F, F) -> REC
rtNormalize        : F -> F
rootNormalize0     : F -> REC
rootKernelNormalize : (F, List K, K) -> Union(REC, "failed")
tanSum             : (F, List F) -> F

comb?     := F has CombinatorialOpsCategory
mpiover2 : F := pi()$F / (-2::F)

realElem(f, l) == rmap(k +-> k2Elem(k, l), f)
realElementary(f, x) == realElem(f, [x])
realElementary f     == realElem(f, variables f)
k_to_liou : K -> F
k_to_liou1 : (K, SY) -> F
realLiouvillian(f) == rmap(k_to_liou, f)
realLiouvillian(f, x) == rmap((k : K) : F +-> k_to_liou1(k, x), f)
toY ker              == [func for k in ker | (func := ktoY k) ~= 0]
toZ ker              == [func for k in ker | (func := ktoZ k) ~= 0]
```

```
toU ker              == [func for k in ker | (func := ktoU k) ~= 0]
toV ker              == [func for k in ker | (func := ktoV k) ~= 0]
rtNormalize f        == rootNormalize0(f).func
toR(ker, x) == select(s +-> is?(s, NTHR) and first argument(s) = x, ker)

if R has GcdDomain then
  tanQ(c, x) ==
    tanNa(rootSimp zeroOf tanAn(x, denom(c)::PositiveInteger), numer c)
else
  tanQ(c, x) ==
    tanNa(zeroOf tanAn(x, denom(c)::PositiveInteger), numer c)

-- tanSum(c, [a1, ..., an]) returns f(c, a1, ..., an) such that
-- if ai = tan(ui) then f(c, a1, ..., an) = tan(c + u1 + ... + un).
-- MUST BE CAREFUL FOR WHEN c IS AN ODD MULTIPLE of pi/2
tanSum(c, l) ==
  k := c / mpiover2        -- k = - 2 c / pi, check for odd integer
                           -- tan((2n+1) pi/2 x) = - 1 / tan x
  (r := retractIfCan(k)@Union(Z, "failed")) case Z and odd?(r::Z) =>
       - inv tanSum l
  tanSum concat(tan c, l)

rootNormalize0 f ==
  ker := select!(x +-> is?(x, NTHR) and empty? variables first argument x,
                 tower f)$List(K)
  empty? ker => [f, empty(), empty()]
  (n := (#ker)::Z - 1) < 1 => [f, empty(), empty()]
  for i in 1..n for kk in rest ker repeat
    (u := rootKernelNormalize(f, first(ker, i), kk)) case REC =>
      rec := u::REC
      rn  := rootNormalize0(rec.func)
      return [rn.func, concat(rec.kers, rn.kers), concat(rec.vals, rn.vals)]
  [f, empty(), empty()]

findQRelation(lv : List Symbol, lpar : List Symbol, lk : List F, _
         ker : F) : U ==

    null lk => [true]
    isconstant := true
    m := #lv
    lvv := lv
    n := #lk
    v := new(m, 0)$(Vector F)
    for i in 1..m for var in lv repeat
        v(i) := differentiate(ker, var)
        if isconstant then
```

```
            isconstant := v(i) = 0
    if isconstant then
        m := #lpar
        lvv := lpar
        v := new(m, 0)$(Vector F)
        for i in 1..m for var in lpar repeat
            v(i) := differentiate(ker, var)
            if isconstant then
                isconstant := v(i) = 0
    isconstant =>
        print(ker::OutputForm)$OutputForm
        error "Hidden constant detected"
    mat := new(m, n, 0)$(Matrix F)
    for i in 1..m for var in lvv repeat
        for j in 1..n for k in lk repeat
            mat(i, j) := differentiate(k, var)
    (u := particularSolutionOverQ(mat, v)) case Vector(Q) => [u::Vector(Q)]
    [true]

-- This is only correct if Schanuel Conjecture is true, otherwise
-- we may miss some relations.
findLinearRelation1(lk : List F, ker : F) : U ==
    null lk => [true]

    n := #lk
    mat := new(1, n, 0)$(Matrix F)
    v := new(1, ker)$(Vector F)
    for j in 1..n for k in lk repeat
        if null(variables(k)) then
            mat(1, j) := k
        else
            mat(1, j) := 0::F
    (u := particularSolutionOverQ(mat, v)) case Vector(Q) => [u::Vector(Q)]
    [true]

ALGOP  := '%alg
transkers(x : List K) : List K ==
    [k for k in x | not(has?(operator k, ALGOP))]

ktoQ(ker : K) : Q ==
   is?(ker, 'log) and F has RetractableTo Q =>
      z : F := argument(ker).1
      qu := retractIfCan(z)@Union(Q, "failed")
      qu case Q => qu::Q
      1
   1
```

```
toQ(lk : List K) : List Q ==
    [ktoQ(k) for k in lk | is?(k, 'log) or is?(k, 'exp)]

import from MultiplicativeDependence()

findLinearRelation2(lk : List K, lz : List F, ker : K) : U ==
    z : F := argument(ker).1
    zkers := transkers(kernels(z))
    empty?(zkers) =>
        -- Algebraic case, check for dependencies between logarithms
        -- of rational numbers (we should do better)
        q := ktoQ(ker)
        not(q = 1 or q = -1) =>
            (u := logDependenceQ([toQ (lk)], q)) case Vector(Q) => [u::Vector(Q)]
            [true]
        kerF := ker :: F
        F is Expression(R) and R has ConvertibleTo(Float) _
          and R has IntegralDomain and R has OrderedSet =>
            m := #lz
            for z1 in lz for i in 1..m repeat
                Fratio : F := kerF/log(z1)
                (fratio := numericIfCan(Fratio, 20)$Numeric(R) _
                     ) case Float =>
                    qratio := rationalApproximation(fratio::Float, 8)
                    nd : Integer
                    nq : Integer
                    (qratio = 0 or
                      abs(fratio/(qratio::Float)-1.0) > 1.0e-16) _
                      or (abs(nq := numer(qratio)) > 100) _
                      or (abs(nd := denom(qratio)) > 100) => "iterate"
                    kertond := (argument(ker).1)^nd
                    nq > 0 =>
                        lz1tonq := z1^nq
                        (kertond = lz1tonq) =>
                            vv := zero(m)$Vector(Q)
                            qsetelt!(vv, i, qratio)
                            return [vv]
                    lz1tonq := (z1)^(-nq)
                    kertond*lz1tonq = 1 =>
                        vv := zero(m)$Vector(Q)
                        qsetelt!(vv, i, qratio)
                        return [vv]
            [true]
        [true]
    lpars0 : List K := transkers(lk)
```

```
    lpars1 : List Symbol := [new()$Symbol for k in lpars0]
    lpars1f : List F := [kernel(s)::F for s in lpars1]
    ly : List F
    nz : F
    if is?(ker, 'log) then
        ly := [log(eval(x, lpars0, lpars1f)) for x in lz]
        nz := log(eval(z, lpars0, lpars1f))
    else
        not(is?(ker, 'atan)) =>
            error "findLinearRelation2: kernel should be log or atan"
        ly := [atan(eval(x, lpars0, lpars1f)) for x in lz]
        nz := atan(eval(z, lpars0, lpars1f))
    findQRelation([], lpars1, ly, nz)

findRelation(lv : List Symbol, lpar : List Symbol, lk : List K, _
          ker : K) : U ==
    is?(ker, 'log) or is?(ker, 'exp) =>
        null(variables(ker::F)) =>
            is?(ker, 'exp) => findLinearRelation1(toY lk, ktoY ker)
            findLinearRelation2(lk, toZ lk, ker)
        findQRelation(lv, lpar, toY lk, ktoY ker)
    is?(ker, 'atan) or is?(ker, 'tan) =>
        null(variables(ker::F)) =>
            is?(ker, 'tan) => findLinearRelation1(toU lk, ktoU ker)
            findLinearRelation2(lk, toV lk, ker)
        findQRelation(lv, lpar, toU lk, ktoU ker)
    is?(ker, NTHR) => rootDep(lk, ker)
    comb? and is?(ker, 'factorial) =>
        factdeprel([x for x in lk | is?(x, 'factorial) and x ~= ker],
                   ker)
    [true]

ktoY k ==
  is?(k, 'log) => k::F
  is?(k, 'exp) => first argument k
  0

ktoZ k ==
  is?(k, 'log) => first argument k
  is?(k, 'exp) => k::F
  0

ktoU k ==
  is?(k, 'atan) => k::F
  is?(k,  'tan) => first argument k
  0
```

```
ktoV k ==
  is?(k,  'tan) => k::F
  is?(k, 'atan) => first argument k
  0

smp_map(f : K -> F, p : SMP) : F ==
     map(f, y +-> y::F, p)$PolynomialCategoryLifting(
                                   IndexedExponents K, K, R, SMP, F)

rmap(f, e) == smp_map(f, numer e)/smp_map(f, denom e)

LF  ==> LiouvillianFunction(R, F)
opint : BasicOperator := operator(operator('%iint)$CommonOperators)$LF

k2Elem0(k : K, op : BasicOperator, args : List F) : F ==
  ez, iez, tz2 : F

  z := first args

  is?(op, POWER)       => (zero? z => 0; exp(last(args) * log z))
  is?(op, 'cot)   => inv tan z
  is?(op, 'acot)  => atan inv z
  is?(op, 'asin)  => atan(z / sqrt(1 - z^2))
  is?(op, 'acos)  => atan(sqrt(1 - z^2) / z)
  is?(op, 'asec)  => atan sqrt(z^2 - 1)
  is?(op, 'acsc)  => atan inv sqrt(z^2 - 1)
  is?(op, 'asinh) => log(sqrt(1 + z^2) + z)
  is?(op, 'acosh) => log(sqrt(z^2 - 1) + z)
  is?(op, 'atanh) => log((z + 1) / (1 - z)) / (2::F)
  is?(op, 'acoth) => log((z + 1) / (z - 1)) / (2::F)
  is?(op, 'asech) => log((inv z) + sqrt(inv(z^2) - 1))
  is?(op, 'acsch) => log((inv z) + sqrt(1 + inv(z^2)))
  is?(op, '%paren) or is?(op, '%box) =>
    empty? rest args => z
    k::F
  if has?(op, HTRIG) then iez  := inv(ez  := exp z)
  is?(op, 'sinh)  => (ez - iez) / (2::F)
  is?(op, 'cosh)  => (ez + iez) / (2::F)
  is?(op, 'tanh)  => (ez - iez) / (ez + iez)
  is?(op, 'coth)  => (ez + iez) / (ez - iez)
  is?(op, 'sech)  => 2 * inv(ez + iez)
  is?(op, 'csch)  => 2 * inv(ez - iez)
  if has?(op, TRIG) then tz2  := tan(z / (2::F))
  is?(op, 'sin)   => 2 * tz2 / (1 + tz2^2)
  is?(op, 'cos)   => (1 - tz2^2) / (1 + tz2^2)
```

```
  is?(op, 'sec)   => (1 + tz2^2) / (1 - tz2^2)
  is?(op, 'csc)   => (1 + tz2^2) / (2 * tz2)
  op args

do_int(op : BasicOperator, args : List(F)) : F ==
    kf1 := op args
    vars := variables(kf1)
    vfs := [v::F for v in vars]
    dvs := [realLiouvillian(D(kf1, v)) for v in vars]
    kernel(opint, concat(vfs, dvs))::F

k_to_liou(k) ==
    op := operator k
    args := [realLiouvillian(a) for a in argument(k)]
    empty?(args) => k::F
    has?(op, 'prim) and not(is?(op, '%iint)) => do_int(op, args)
    nm := name(op)
    nm = 'polylog and
      (iu := retractIfCan(first(args))@Union(Integer, "failed"))
             case Integer =>
        (i := iu::Integer) > 0 and i < 10 => do_int(op, args)
        k2Elem0(k, op, args)
    k2Elem0(k, op, args)

do_int1(op : BasicOperator, args : List(F), x : SY) : F ==
    kf1 := op args
    vars : List(SY) := [x]
    vfs := [v::F for v in vars]
    dvs := [realLiouvillian(D(kf1, v), x) for v in vars]
    kernel(opint, concat(vfs, dvs))::F

k_to_liou1(k, x) ==
    op := operator k
    args := [realLiouvillian(a, x) for a in argument(k)]
    empty?(args) => k::F
    has?(op, 'prim) and not(is?(op, '%iint)) => do_int1(op, args, x)
    nm := name(op)
    nm = 'Gamma2 and D(first(args), x) = 0 => do_int1(op, args, x)
    nm = 'polylog and
      (iu := retractIfCan(first(args))@Union(Integer, "failed"))
             case Integer =>
        (i := iu::Integer) > 0 and i < 10 => do_int(op, args)
        k2Elem0(k, op, args)
    (nm = 'ellipticE2 or nm = 'ellipticF) and D(args(2), x) = 0 =>
        do_int1(op, args, x)
    nm = 'ellipticPi and D(args(2), x) = 0 and D(args(3), x) = 0 =>
```

```
            do_int1(op, args, x)
        k2Elem0(k, op, args)

    k2Elem(k, l) ==
        op := operator k
        args := [realElem(a, l) for a in argument(k)]
        empty?(args) => k::F
        k2Elem0(k, op, args)

--The next 5 functions are used by normalize, once a relation is found
    depeval(f, lk, k, v) ==
      is?(k, 'log)  => logeval(f, lk, k, v)
      is?(k, 'exp)  => expeval(f, lk, k, v)
      is?(k, 'tan)  => taneval(f, lk, k, v)
      is?(k, 'atan) => ataneval(f, lk, k, v)
      is?(k, NTHR) => rooteval(f, lk, k, v(minIndex v))
      [f, empty(), empty()]

    rooteval(f, lk, k, n) ==
      nv := nthRoot(x := first argument k, m := retract(n)@Z)
      l  := [r for r in concat(k, toR(lk, x)) |
             retract(second argument r)@Z ~= m]
      lv := [nv ^ (n / (retract(second argument r)@Z::Q)) for r in l]
      [eval(f, l, lv), l, lv]

    ataneval(f, lk, k, v) ==
      w := first argument k
      s := tanSum [tanQ(qelt(v, i), x)
                   for i in minIndex v .. maxIndex v for x in toV lk]
      g := +/[qelt(v, i) * x for i in minIndex v .. maxIndex v for x in toU lk]
      h : F :=
        zero?(d := 1 + s * w) => mpiover2
        atan((w - s) / d)
      g := g + h
      [eval(f, [k], [g]), [k], [g]]

    gdCoef?(c, v) ==
      for i in minIndex v .. maxIndex v repeat
        retractIfCan(qelt(v, i) / c)@Union(Z, "failed") case "failed" =>
          return false
      true


    -- If k1 is part of k2 we should not express k1 in terms of k2
    -- (othewise we would get infinite recursion).
    -- Below we impose a stronger condition : we require
```

```
-- height(k1) to be maximal

goodCoef(v, l, s) ==
  h : NonNegativeInteger := 0
  j : Integer := 0
  ll : List K := []
  for k in l repeat
    if (is?(k, 'log) or is?(k, 'exp)
        or is?(k, 'tan) or is?(k, 'atan)) then
          ll := cons(k, ll)
          h := h + 1
  not (h = (maxIndex(v) - minIndex(v) + 1)) => "failed"
  h := 0
  ll := reverse(ll)
  for i in minIndex v .. maxIndex v for k in ll repeat
    h1 := height(k)
    if (h1 > h) then
      j := i
      h := h1
  for i in minIndex v .. maxIndex v for k in ll repeat
    is?(k, s) and (i >= j) and
       ((r := recip(qelt(v, i))) case Q) and
        (retractIfCan(r::Q)@Union(Z, "failed") case Z)
          and gdCoef?(qelt(v, i), v) => return([i, k])
  "failed"

taneval(f, lk, k, v) ==
  u := first argument k
  fns := toU lk
  c := u - +/[qelt(v, i) * x for i in minIndex v .. maxIndex v for x in fns]
  (rec := goodCoef(v, lk, 'tan)) case "failed" =>
      tannosimp(f, lk, k, v, fns, c)
  v0 := retract(inv qelt(v, rec.index))@Z
  lv := [qelt(v, i) for i in minIndex v .. maxIndex v |
                                             i ~= rec.index]$List(Q)
  l  := [kk for kk in lk | kk ~= rec.ker]
  g := tanSum(-v0 * c, concat(tanNa(k::F, v0),
       [tanNa(x, - retract(a * v0)@Z) for a in lv for x in toV l]))
  [eval(f, [rec.ker], [g]), [rec.ker], [g]]

tannosimp(f, lk, k, v, fns, c) ==
  n := maxIndex v
  lk := [x for x in lk | is?(x, 'tan) or is?(x, 'atan)]
  lk1 := [x for x in lk for i in 1..n | not(qelt(v, i) = 0)]
  every?(x +-> is?(x, 'tan), lk1) =>
    dd := (d := (cd := splitDenominator v).den)::F
```

```
        newt := [tan(u / dd) for u in fns
                    for i in 1..n | not(qelt(v, i) = 0)]$List(F)
        newtan := [tanNa(t, d) for t in newt]$List(F)
        li := [i for i in 1..n | not(qelt(v, i) = 0)]
        h := tanSum(c, [tanNa(t, qelt(cd.num, i))
                        for i in li for t in newt])
        newtan := concat(h, newtan)
        lk1 := concat(k, lk1)
        [eval(f, lk1, newtan), lk1, newtan]
      h := tanSum(c, [tanQ(qelt(v, i), x)
                      for i in 1..n for x in toV lk])
      [eval(f, [k], [h]), [k], [h]]

    expnosimp(f, lk, k, v, fns, g) ==
      n := maxIndex v
      lk := [x for x in lk | is?(x, 'exp) or is?(x, 'log)]
      lk1 := [x for x in lk for i in 1..n | not(qelt(v, i) = 0)]
      every?(x +-> is?(x, 'exp), lk1) =>
        dd := (d := (cd := splitDenominator v).den)::F
        newe := [exp(y / dd) for y in fns
                    for i in 1..n | not(qelt(v, i) = 0)]$List(F)
        newexp := [e ^ d for e in newe]$List(F)
        li := [i for i in 1..n | not(qelt(v, i) = 0)]
        h := */[e ^ qelt(cd.num, i)
                for i in li for e in newe] * g
        lk1 := concat(k, lk1)
        newexp := concat(h, newexp)
        [eval(f, lk1, newexp), lk1, newexp]
      h := */[exp(y) ^ qelt(v, i)
                for i in minIndex v .. maxIndex v for y in fns] * g
      [eval(f, [k], [h]), [k], [h]]

    logeval(f, lk, k, v) ==
      z := first argument k
      dd := lcm([denom(qelt(v, i))$Q for i in minIndex v .. maxIndex v
                ]$List(Z))
      c := z^dd / (*/[x^(dd*qelt(v, i))
                   for x in toZ lk for i in minIndex v .. maxIndex v])
-- CHANGED log ktoZ x TO ktoY x SINCE WE WANT log exp f TO BE REPLACED BY f.
      g := +/[qelt(v, i) * x
              for i in minIndex v .. maxIndex v for x in toY lk]
                + log(c)/(dd::R::F)
      [eval(f, [k], [g]), [k], [g]]

    rischNormalize(f : F, vars : List SY) : REC ==
      lk : List K := tower f
```

```
    funs := lk -- [k for k in lk | height k > 1]@(List K)
    pars := variables(f) -- [name(operator(k)) for k in lk | height k = 1]
    pars := setDifference(pars, vars)
    --  funs := [k for k in kers | height k > 1]
    empty?(funs) => [f, empty(), empty()]
    n := #funs

    for i in 1..n for kk in rest funs repeat
      klist := first(funs, i)
      -- NO EVALUATION ON AN EMPTY VECTOR, WILL CAUSE INFINITE LOOP
      (c := findRelation(vars, pars, klist, kk)) case vec and not empty?(c.vec) =>

        rec := depeval(f, klist, kk, c.vec)
        rn  := rischNormalize(rec.func, vars)
        return [rn.func,
                 concat(rec.kers, rn.kers), concat(rec.vals, rn.vals)]
      c case func =>
        rn := rischNormalize(eval(f, [kk], [c.func]), vars)
        return [rn.func, concat(kk, rn.kers), concat(c.func, rn.vals)]
    [f, empty(), empty()]

  rischNormalize(f : F, v : SY) : REC == rischNormalize(f, [v])

  rootNormalize(f, k) ==
    (u := rootKernelNormalize(f, toR(tower f, first argument k), k))
       case "failed" => f
    (u::REC).func

  rootKernelNormalize(f, l, k) ==
    (c := rootDep(l, k)) case vec =>
      rooteval(f, l, k, (c.vec)(minIndex(c.vec)))
    "failed"

  localnorm f == rischNormalize(f, []).func

  validExponential(twr, eta, x) ==
    (c := particularSolutionOverQ(construct([differentiate(g, x)
       for g in (fns := toY twr)]$List(F))@Vector(F),
         differentiate(eta, x))) case "failed" => "failed"
    v := c::Vector(Q)
    g := eta - +/[qelt(v, i) * yy
                     for i in minIndex v .. maxIndex v for yy in fns]
    */[exp(yy) ^ qelt(v, i)
              for i in minIndex v .. maxIndex v for yy in fns] * exp g

  if R has GcdDomain then
```

```
  import from PolynomialRoots(IndexedExponents K, K, R, P, F)
  irootDep(k : K) : U ==
     n : N := (retract(second argument k)@Z)::N
     pr := froot(first argument k, n)
     not(pr.coef = 1) or not(pr.exponent = n) =>
         pr.exponent = 1 => [pr.coef*pr.radicand]
         nf : F := (pr.exponent)::F
         nr : F := pr.radicand
         nk : F := kernel(operator k, [nr, nf])
         nv : F := pr.coef*nk
         [nv]
     [true]
else
  irootDep(k : K) : U == [true]

rootDep(ker, k) ==
  empty?(ker := toR(ker, first argument k)) => irootDep(k)
  [new(1, lcm(retract(second argument k)@Z,
   "lcm"/[retract(second argument r)@Z for r in ker])::Q)$Vector(Q)]

expeval(f, lk, k, v) ==
  y   := first argument k
  fns := toY lk
  g := y - +/[qelt(v, i) * z for i in minIndex v .. maxIndex v for z in fns]
  (rec := goodCoef(v, lk, 'exp)) case "failed" =>
    expnosimp(f, lk, k, v, fns, exp g)
  v0 := retract(inv qelt(v, rec.index))@Z
  lv := [qelt(v, i) for i in minIndex v .. maxIndex v |
                                              i ~= rec.index]$List(Q)
  l  := [kk for kk in lk | kk ~= rec.ker]
  h : F := */[exp(z) ^ (- retract(a * v0)@Z) for a in lv for z in toY l]
  h := h * exp(-v0 * g) * (k::F) ^ v0
  [eval(f, [rec.ker], [h]), [rec.ker], [h]]

if F has CombinatorialOpsCategory then
  normalize f == rtNormalize localnorm factorials realElementary f

  normalize(f, x) ==
    rtNormalize(rischNormalize(factorials(realElementary(f, x), x), x).func)

  factdeprel(l, k) ==
    ((r := retractIfCan(n := first argument k)@Union(Z, "failed"))
      case Z) and (r::Z > 0) => [factorial(r::Z)::F]
    for x in l repeat
      m := first argument x
      ((r := retractIfCan(n - m)@Union(Z, "failed")) case Z) =>
```

```
        (r::Z > 0) => return([*/[(m + i::F) for i in 1..r] * x::F])
        error "bad order of factorials"
    [true]

else
  normalize f      == rtNormalize localnorm realElementary f
  normalize(f, x) == rtNormalize(rischNormalize(realElementary(f, x), x).func)
```

4 package ITRIGMNP InnerTrigonometricManipulations

⟨*package ITRIGMNP InnerTrigonometricManipulations*⟩≡

```
)abbrev package ITRIGMNP InnerTrigonometricManipulations
)boot $tryRecompileArguments := nil
++ Trigs to/from exps and logs
++ Author: Manuel Bronstein
++ Date Created: 4 April 1988
++ Date Last Updated: 9 October 1993
++ Description:
++   This package provides transformations from trigonometric functions
++   to exponentials and logarithms, and back.
++   F and FG should be the same type of function space.
++ Keywords: trigonometric, function, manipulation.
InnerTrigonometricManipulations(R, F, FG) : Exports == Implementation where
  R  : Join(IntegralDomain, Comparable)
  F  : Join(FunctionSpace R, RadicalCategory,
            TranscendentalFunctionCategory)
  FG : Join(FunctionSpace Complex R, RadicalCategory,
            TranscendentalFunctionCategory)

  Z   ==> Integer
  SY  ==> Symbol
  OP  ==> BasicOperator
  GR  ==> Complex R
  GF  ==> Complex F
  KG  ==> Kernel FG
  PG  ==> SparseMultivariatePolynomial(GR, KG)
  UP  ==> SparseUnivariatePolynomial PG
  NTHR  ==> 'nthRoot

  Exports ==> with
    GF2FG        : GF -> FG
      ++ GF2FG(a + i b) returns \spad{a + i b} viewed as a function with
      ++ the \spad{i} pushed down into the coefficient domain.
    FG2F         : FG -> F
      ++ FG2F(a + i b) returns \spad{a + sqrt(-1) b}.
    F2FG         : F  -> FG
      ++ F2FG(a + sqrt(-1) b) returns \spad{a + i b}.
    explogs2trigs : FG -> GF
      ++ explogs2trigs(f) rewrites all the complex logs and
      ++ exponentials appearing in \spad{f} in terms of trigonometric
      ++ functions.
    trigs2explogs : (FG, List KG) -> FG
```

```
      ++ trigs2explogs(f, [k1, ..., kn]) rewrites
      ++ all the trigonometric functions appearing in \spad{f}
      ++ in terms of complex logarithms and
      ++ exponentials. A kernel of the form \spad{tan(u)} is expressed
      ++ using \spad{exp(u)^2} if it is one of the \spad{ki's}, in terms of
      ++ \spad{exp(2*u)} otherwise.
    do_liou : (OP, List GF) -> GF
      ++ do_liou(op, lf) should be local but conditional.

  Implementation ==> add
    ker2explogs : (KG, List KG) -> FG
    smp2explogs : (PG, List KG) -> FG
    supexp      : (UP, GF, GF, Z) -> GF
    GR2GF       : GR -> GF
    GR2F        : GR -> F
    KG2F        : KG -> F
    PG2F        : PG -> F
    ker2trigs   : (OP, List GF) -> GF
    smp2trigs   : PG -> GF
    sup2trigs   : (UP, GF) -> GF

    nth := R has RetractableTo(Integer) and F has RadicalCategory

    GR2F g        == real(g)::F + sqrt(-(1::F)) * imag(g)::F
    KG2F k        == map(FG2F, k)$ExpressionSpaceFunctions2(FG, F)
    FG2F f        == (PG2F numer f) / (PG2F denom f)
    F2FG f        == map(x +-> x::GR, f)$FunctionSpaceFunctions2(R, F, GR, FG)
    GF2FG f       == (F2FG real f) + complex(0, 1)$GR ::FG * F2FG imag f
    GR2GF gr      == complex(real(gr)::F, imag(gr)::F)

    im := complex(0, 1)$GR :: FG

-- This expects the argument to have only tan and atans left.
-- Does a half-angle correction if k is not in the initial kernel list.
    ker2explogs(k, l) ==
      kf := k::FG
      empty?(args := [trigs2explogs(a, l)
                              for a in argument k]$List(FG)) => kf
      z  := first args
      is?(k, 'tan)  =>
        e := (member?(k, l) => exp(im * z) ^ 2;  exp(2 * im * z))
        - im * (e - 1) /$FG (e + 1)
      is?(k, 'atan) =>
        im * log((1 -$FG im *$FG z)/$FG (1 +$FG im *$FG z))$FG / (2::FG)
      (operator k) args
```

```
trigs2explogs(f, l) ==
  smp2explogs(numer f, l) / smp2explogs(denom f, l)


ker2trigs_error(op : OP, arg : List GF) : GF ==
    print(op::OutputForm)$OutputForm
    print(arg::OutputForm)$OutputForm
    error "ker2trigs: cannot convert kernel to gaussian function"


if F has LiouvillianFunctionCategory then
    do_liou(op, arg) ==
        a := first(arg)
        is?(op, 'Ei) and real(a) = 0 =>
            complex(Ci(imag(a)), Si(imag(a)))
        is?(op, 'erf) and real(a) = 0 => complex(0, erfi(imag(a)))
        is?(op, 'erf) and real(a) = imag(a) =>
            a1 := 2*real(a)/sqrt(pi())
            complex(1, 1)*complex(fresnelC(a1), -fresnelS(a1))
        is?(op, 'erf) and real(a) = -imag(a) =>
            a1 := 2*real(a)/sqrt(pi())
            complex(1, -1)*complex(fresnelC(a1), fresnelS(a1))
        ker2trigs_error(op, arg)

else
    do_liou(op, args) == ker2trigs_error(op, args)

-- return op(arg) as f + %i g
-- op is already an operator with semantics over R, not GR
ker2trigs(op, arg) ==
  "and"/[zero? imag x for x in arg] =>
    complex(op [real x for x in arg]$List(F), 0)
  a := first arg
  is?(op, 'exp)  => exp a
  is?(op, 'log)  => log a
  is?(op, 'sin)  => sin a
  is?(op, 'cos)  => cos a
  is?(op, 'tan)  => tan a
  is?(op, 'cot)  => cot a
  is?(op, 'sec)  => sec a
  is?(op, 'csc)  => csc a
  is?(op, 'asin)  => asin a
  is?(op, 'acos)  => acos a
  is?(op, 'atan)  => atan a
  is?(op, 'acot)  => acot a
  is?(op, 'asec)  => asec a
```

```
  is?(op, 'acsc)  => acsc a
  is?(op, 'sinh)  => sinh a
  is?(op, 'cosh)  => cosh a
  is?(op, 'tanh)  => tanh a
  is?(op, 'coth)  => coth a
  is?(op, 'sech)  => sech a
  is?(op, 'csch)  => csch a
  is?(op, 'asinh)  => asinh a
  is?(op, 'acosh)  => acosh a
  is?(op, 'atanh)  => atanh a
  is?(op, 'acoth)  => acoth a
  is?(op, 'asech)  => asech a
  is?(op, 'acsch)  => acsch a
  is?(op, 'abs)    => sqrt(norm a)::GF
  nth and is?(op, NTHR) => nthRoot(a, retract(second arg)@Z)
  is?(op, '%iint) =>
      complex(op [real x for x in arg]$List(F),
              op [imag x for x in arg]$List(F))
  do_liou(op, arg)

sup2trigs(p, f) ==
  map(smp2trigs, p)$SparseUnivariatePolynomialFunctions2(PG, GF) f

smp2trigs p ==
  map(x +-> explogs2trigs(x::FG), GR2GF, p)$PolynomialCategoryLifting(
                              IndexedExponents KG, KG, GR, PG, GF)

explogs2trigs f ==
  nf := numer f
  df := denom f
  (m := mainKernel f) case "failed" =>
    GR2GF(retract(nf)@GR) / GR2GF(retract(df)@GR)
  op  := operator(operator(k := m::KG))$F
  arg := [explogs2trigs x for x in argument k]
  num := univariate(nf, k)
  den := univariate(df, k)
  is?(op, 'exp) =>
    e  := exp real first arg
    y  := imag first arg
    g  := complex(e *  cos y, e * sin y)$GF
    gi := complex(cos(y) / e, - sin(y) / e)$GF
    supexp(num, g, gi, b := (degree num)::Z quo 2)/supexp(den, g, gi, b)
  sup2trigs(num, g := ker2trigs(op, arg)) / sup2trigs(den, g)

supexp(p, f1, f2, bse) ==
  ans : GF := 0
```

```
  while p ~= 0 repeat
    g := explogs2trigs(leadingCoefficient(p)::FG)
    if ((d := degree(p)::Z - bse) >= 0) then
         ans := ans + g * f1 ^ d
    else ans := ans + g * f2 ^ (-d)
    p := reductum p
  ans

PG2F p ==
  map(KG2F, GR2F, p)$PolynomialCategoryLifting(IndexedExponents KG,
                                                          KG, GR, PG, F)

smp2explogs(p, l) ==
  map(x +-> ker2explogs(x, l), y +-> y::FG,
       p)$PolynomialCategoryLifting(IndexedExponents KG, KG, GR, PG, FG)
```

5 package TRIGMNIP TrigonometricManipulations

⟨*package TRIGMNIP TrigonometricManipulations*⟩≡

```
)abbrev package TRIGMNIP TrigonometricManipulations
++ Trigs to/from exps and logs
++ Author: Manuel Bronstein
++ Date Created: 4 April 1988
++ Date Last Updated: 14 February 1994
++ Description:
++   \spadtype{TrigonometricManipulations} provides transformations from
++   trigonometric functions to complex exponentials and logarithms, and back.
++ Keywords: trigonometric, function, manipulation.
TrigonometricManipulations(R, F) : Exports == Implementation where
  R : Join(GcdDomain, Comparable, RetractableTo Integer,
           LinearlyExplicitRingOver Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace R)

  Z   ==> Integer
  SY  ==> Symbol
  K   ==> Kernel F
  FG  ==> Expression Complex R

  Exports ==> with
    complexNormalize : F -> F
      ++ complexNormalize(f) rewrites \spad{f} using the least possible number
      ++ of complex independent kernels.
    complexNormalize : (F, SY) -> F
      ++ complexNormalize(f, x) rewrites \spad{f} using the least possible
      ++ number of complex independent kernels involving \spad{x}.
    complexElementary : F -> F
      ++ complexElementary(f) rewrites \spad{f} in terms of the 2 fundamental
      ++ complex transcendental elementary functions: \spad{log, exp}.
    complexElementary : (F, SY) -> F
      ++ complexElementary(f, x) rewrites the kernels of \spad{f} involving
      ++ \spad{x} in terms of the 2 fundamental complex
      ++ transcendental elementary functions: \spad{log, exp}.
    trigs  : F -> F
      ++ trigs(f) rewrites all the complex logs and exponentials
      ++ appearing in \spad{f} in terms of trigonometric functions.
    real   : F -> F
      ++ real(f) returns the real part of \spad{f} where \spad{f} is a complex
      ++ function.
    imag   : F -> F
```

```
      ++ imag(f) returns the imaginary part of \spad{f} where \spad{f}
      ++ is a complex function.
    real?  : F -> Boolean
      ++ real?(f) returns \spad{true} if \spad{f = real f}.
    complexForm : F -> Complex F
      ++ complexForm(f) returns \spad{[real f, imag f]}.

  Implementation ==> add
    import from ElementaryFunctionSign(R, F)
    import from InnerTrigonometricManipulations(R, F, FG)
    import from ElementaryFunctionStructurePackage(R, F)
    import from ElementaryFunctionStructurePackage(Complex R, FG)

    RTRIG := 'rtrig

    s1  := sqrt(-1::F)
    ipi := pi()$F * s1

    K2KG          : K -> Kernel FG
    kcomplex      : K -> Union(F, "failed")
    locexplogs    : F -> FG
    localexplogs  : (F, F) -> FG
    complexKernels : F -> Record(ker : List K, val : List F)

    K2KG k           == retract(tan F2FG first argument k)@Kernel(FG)
    real? f          == empty?(complexKernels(f).ker)
    real f           == real complexForm f
    imag f           == imag complexForm f

-- returns [[k1, ..., kn], [v1, ..., vn]] such that ki should be replaced by vi
    complexKernels f ==
      lk : List(K) := empty()
      lv : List(F) := empty()
      for k in tower f repeat
        if (u := kcomplex k) case F then
           lk := concat(k, lk)
           lv := concat(u::F, lv)
      [lk, lv]

-- returns f if it is certain that k is not a real kernel and k = f,
-- "failed" otherwise
    kcomplex k ==
      op := operator k
      is?(k, 'nthRoot) =>
        arg := argument k
        even?(retract(n := second arg)@Z) and ((u := sign(first arg)) case Z)
```

```
      and (u::Z < 0) => op(s1, n / 2::F) * op(- first arg, n)
    "failed"
  is?(k, 'log) and ((u := sign(a := first argument k)) case Z)
      and (u::Z < 0) => op(- a) + ipi
  "failed"

complexForm f ==
  empty?((l := complexKernels f).ker) => complex(f, 0)
  explogs2trigs locexplogs eval(f, l.ker, l.val)

locexplogs f ==
  any?(x +-> has?(x, RTRIG),
       operators(g := realElementary f))$List(BasicOperator) =>
          localexplogs(f, g)
  F2FG g

complexNormalize(f, x) ==
  any?(y +-> has?(operator y, RTRIG),
   [k for k in tower(g := realElementary(f, x))
           | member?(x, variables(k::F))]$List(K))$List(K) =>
               FG2F(rischNormalize(localexplogs(f, g), x).func)
  rischNormalize(g, x).func

complexNormalize f ==
  l := variables(g := realElementary f)
  any?(x +-> has?(x, RTRIG), operators g)$List(BasicOperator) =>
    h := localexplogs(f, g)
    h := rischNormalize(h, empty()).func
    FG2F h
  g := rischNormalize(g, empty()).func
  g

complexElementary(f, x) ==
  any?(y +-> has?(operator y, RTRIG),
   [k for k in tower(g := realElementary(f, x))
             | member?(x, variables(k::F))]$List(K))$List(K) =>
                 FG2F localexplogs(f, g)
  g

complexElementary f ==
  any?(x +-> has?(x, RTRIG),
    operators(g := realElementary f))$List(BasicOperator) =>
      FG2F localexplogs(f, g)
  g

localexplogs(f, g) ==
```

```
  trigs2explogs(F2FG g, [K2KG k for k in tower f
                      | is?(k, 'tan) or is?(k, 'cot)])

trigs f ==
  real? f => f
  g := explogs2trigs F2FG f
  real g + s1 * imag g
```

6 package CTRIGMNP ComplexTrigonometricManipulations

⟨*package CTRIGMNP ComplexTrigonometricManipulations*⟩≡

```
)abbrev package CTRIGMNP ComplexTrigonometricManipulations
++ Real and Imaginary parts of complex functions
++ Author: Manuel Bronstein
++ Date Created: 11 June 1993
++ Date Last Updated: 14 June 1993
++ Description:
++   \spadtype{ComplexTrigonometricManipulations} provides function that
++   compute the real and imaginary parts of complex functions.
++ Keywords: complex, function, manipulation.
ComplexTrigonometricManipulations(R, F) : Exports == Implementation where
  R : Join(IntegralDomain, Comparable, RetractableTo Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace Complex R)


  SY  ==> Symbol
  FR  ==> Expression R
  K   ==> Kernel F


  Exports ==> with
    complexNormalize : F -> F
      ++ complexNormalize(f) rewrites \spad{f} using the least possible number
      ++ of complex independent kernels.
    complexNormalize : (F, SY) -> F
      ++ complexNormalize(f, x) rewrites \spad{f} using the least possible
      ++ number of complex independent kernels involving \spad{x}.
    complexElementary : F -> F
      ++ complexElementary(f) rewrites \spad{f} in terms of the 2 fundamental
      ++ complex transcendental elementary functions: \spad{log, exp}.
    complexElementary : (F, SY) -> F
      ++ complexElementary(f, x) rewrites the kernels of \spad{f} involving
      ++ \spad{x} in terms of the 2 fundamental complex
      ++ transcendental elementary functions: \spad{log, exp}.
    real   : F -> FR
      ++ real(f) returns the real part of \spad{f} where \spad{f} is a complex
      ++ function.
    imag   : F -> FR
      ++ imag(f) returns the imaginary part of \spad{f} where \spad{f}
      ++ is a complex function.
    real?  : F -> Boolean
```

```
    ++ real?(f) returns \spad{true} if \spad{f = real f}.
  trigs  : F -> F
    ++ trigs(f) rewrites all the complex logs and exponentials
    ++ appearing in \spad{f} in terms of trigonometric functions.
  complexForm : F -> Complex FR
    ++ complexForm(f) returns \spad{[real f, imag f]}.

Implementation ==> add
  import from InnerTrigonometricManipulations(R, FR, F)
  import from ElementaryFunctionStructurePackage(Complex R, F)

  RTRIG := 'rtrig

  rreal? : Complex R -> Boolean
  kreal? : Kernel F -> Boolean
  localexplogs  : (F, F) -> F

  real f        == real complexForm f
  imag f        == imag complexForm f
  rreal? r      == zero? imag r
  kreal? k      == every?(real?, argument k)$List(F)
  complexForm f == explogs2trigs f

  trigs f ==
    GF2FG explogs2trigs f

  real? f ==
    every?(rreal?, coefficients numer f)
      and every?(rreal?, coefficients denom f) and every?(kreal?, kernels f)

  localexplogs(f, g) ==
    trigs2explogs(g, [k for k in tower f
                      | is?(k, 'tan) or is?(k, 'cot)])

  complexElementary f ==
    any?(x +-> has?(x, RTRIG),
      operators(g := realElementary f))$List(BasicOperator) =>
        localexplogs(f, g)
    g

  complexElementary(f, x) ==
    any?(y +-> has?(operator y, RTRIG),
     [k for k in tower(g := realElementary(f, x))
              | member?(x, variables(k::F))]$List(K))$List(K) =>
                localexplogs(f, g)
    g
```

```
complexNormalize(f, x) ==
  any?(y +-> has?(operator y, RTRIG),
   [k for k in tower(g := realElementary(f, x))
           | member?(x, variables(k::F))]$List(K))$List(K) =>
               (rischNormalize(localexplogs(f, g), x).func)
  rischNormalize(g, x).func

complexNormalize f ==
  l := variables(g := realElementary f)
  any?(y +-> has?(y, RTRIG), operators g)$List(BasicOperator) =>
    h := localexplogs(f, g)
    rischNormalize(h, empty()).func
  rischNormalize(g, empty()).func
```

7 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 integration world should be compiled in the
-- following order:
--
--   intaux  rderf  intrf  curve  curvepkg  divisor  pfo
--   intalg  intaf  EFSTRUC  rdeef  intef  irexpand  integrat
```

⟨*package SYMFUNC SymmetricFunctions*⟩
⟨*package TANEXP TangentExpansions*⟩
⟨*package EFSTRUC ElementaryFunctionStructurePackage*⟩
⟨*package ITRIGMNP InnerTrigonometricManipulations*⟩
⟨*package TRIGMNIP TrigonometricManipulations*⟩
⟨*package CTRIGMNP ComplexTrigonometricManipulations*⟩

References

[1] nothing