distro.spad

FRANZ LEHNER

Abstract. The domains and packages in this file provide an environment for combinatorial manipulation of probability distributions, moments, cumulants, orthogonal polynomials and convolutions. The central object is the domain `Distribution`, which represents probability distributions as streams of moments and cumulants.

Contents

1. category SEQUCAT SequenceCategory

Sequences form an infinite dimensional vector space (implemented as streams below) over a commutative ring with appropriate vector space arithmetic.

1 ⟨*category SEQUCAT SequenceCategory* 1⟩≡ (32)

```
)abbrev category SEQUCAT SequenceCategory
++ Author: Franz Lehner lehner@math.tugraz.at
++ Date Created: 2008
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   A category for infinite sequences over a commutative ring.
++   It is a stream with arithmetics.
SequenceCategory(R : CommutativeRing) : Category == Exports where
  Exports ==> Join(LazyStreamAggregate(R), Module(R)) with
```

Date: Rev.2600, August 29, 2019.

```
elt : (%, Integer) -> R
  ++ \spad{elt(mm, n)} returns the nth element of a sequence.
apply : (%, Partition) -> R
  ++ \spad{elt(mm, pi)} returns the product of the entries indexed
  ++ by the integer partition pi (as in partitionend moments)
cons : (R, %) -> %
  ++ \spad{cons(r, s)} prepends \spad{r} to the stream \spad{s}
coerce : Stream R -> %
  ++ \spad{coerce(x)} creation of elements
sequence : Stream R -> %
  ++ \spad{sequence(x)} turns the stream x into a sequence
stream : % -> Stream R
  ++ \spad{stream(x)} returns stream of entries
first : (%, NonNegativeInteger) -> %
  ++ \spad{first(x, n)} returns the sequence of the first n entries
firstn : (%, NonNegativeInteger) -> List R
  ++ \spad{firstn(x, n)} returns a list of the first n entries
dilate : (R, %) -> %
  ++ \spad{dilate(a, x)} returns the sequence $a^n x_n$
  ++ (starting at $n=1$)
sequence : (R -> R, R) -> %
  ++ sequence(f, s0) generates a stream recursively
  ++ by applying the function f: S -> S to the last
  ++ computed value.
```

2. domain SEQU Sequence

3 ⟨*domain SEQU Sequence* 3⟩≡ (32)

```
)abbrev domain SEQU Sequence
++ Author: Franz Lehner lehner@math.tugraz.at
++ Date Created: 2008
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   A domain for infinite sequences over a commutative ring.
++   It is implemented as stream with arithmetics.
Sequence(R : CommutativeRing) : SequenceCategory(R) == Implementation where
  Implementation ==> Stream(R) add
    Rep := Stream R
    rep(x : %) : Rep == x :: Rep
    per(r : Rep) : % == r :: %

    0 == per repeating([0$R])$Rep

    elt(mm : %, n : Integer) : R ==
        n < 1 => error "no such element"
        (elt$Rep)(rep mm, n)

    apply(mm : %, pi : Partition) : R ==
        pil : List Integer := convert pi
        import from List R
        reduce(_*, [mm k for k in pil])

    cons(r : R, s : %) : % ==
        per cons(r, rep s)$Rep

  -- module operations

    _+(x : %, y : %) : % ==
        per map((s : R, t : R) : R +-> s+t, rep x, rep y)

    _-(x : %) : % ==
        per map((s : R) : R +-> -s, rep x)

    multiply : (%, %) -> %

    multiply(x : %, y : %) : % ==
            per map((s : R, t : R) : R +-> s*t, rep x, rep y)

    _*(x : R, y : %) : % ==
        per map((t : R) : R +-> x*t, rep y)

    _=(x : %, y : %) : Boolean ==
        rep x = rep y

    coerce(x : Stream R) : % == per x

    sequence(x : Stream R) : % == per x

    stream(x : %) : Stream R == rep x

    first(x : %, n : NonNegativeInteger) : % == per(first(rep x, n)$Rep)
```

```
    firstn(x : %, n : NonNegativeInteger) : List R ==
        entries complete first(rep x, n)$Rep

    -- generator
    sequence(f : R -> R, r : R) : % ==
             per (stream(f, r)$Rep )

    -- output
    coerce(mm : %) : OutputForm ==
            coerce(rep mm)$Rep --$

    dilate(a : R, mm : %) : % ==
            apow : % := sequence( (x : R) : R +-> x*a, a)
            multiply(apow, mm)
```

3. package SEQU2 SequenceFunctions2

4 ⟨*package SEQU2 SequenceFunctions2* 4⟩≡ (32)

```
)abbrev package SEQU2 SequenceFunctions2
++ Author: Franz Lehner lehner@math.tugraz.at
++ Date Created: 16.09.2011
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   A domain for mappings between sequences.
SequenceFunctions2(R1 : CommutativeRing, R2 : CommutativeRing) : Exports
     == Implementation where
  Exports ==> with
    map : (R1 -> R2, Sequence R1) -> Sequence R2
      ++ \spad{map(f, x)} maps the function f on the entries of x

  Implementation ==> add
    map(f, x) == sequence map(f, stream x)$StreamFunctions2(R1, R2)
```

4. package HANKP HankelPackage

The Hankel matrix of a sequence $[a_0, a_1, \ldots, a_{2n}]$ of odd length is the matrix

$$A = \begin{bmatrix} a_0 & a_1 & \ldots & a_n \\ a_1 & a_2 & \ldots & a_{n-1} \\ \ldots & \ldots & \ldots & \ldots \\ a_n & a_2 & \ldots & a_{2n} \end{bmatrix}$$

5 ⟨*package HANKP HankelPackage* 5⟩≡ (32)

```
)abbrev package HANKP HankelPackage
++ Author: Franz Lehner lehner@math.tugraz.at
++ Date Created: 2008
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   Package to generate Hankel matrices.
HankelPackage(R : Ring) : Exports == Implementation where
  Exports ==> with
    HankelMatrix : List R -> Matrix(R)
      ++ \spad{HankelMatrix([a_0, a_1, a_2, ..., a_{2n}])} builds
      ++ the Hankel matrix
      ++ [ a0, a1, ..., an ]
      ++ [ a1, a2, ...     ]
      ++ [ ...             ]
      ++ [ an, ...,     a2n]

  Implementation ==> add
    HankelMatrix(l : List R) : Matrix(R) ==
        not odd? ((#l)::Integer) => error "n must be odd"
        n : Integer := ((#l + 1) exquo 2)::Integer
        lloc : List R := cons(0@R, l)
        import from List Matrix R, List List R
        x : R
        i, j : Integer
        reduce(vertConcat$(Matrix R),
               [ (lloc := rest lloc;
                 matrix [[x for x in lloc for i in 1..n]]) for j in 1..n ])
```

5. package PARRPKG PathArrayPackage

This package counts weighted Motzkin paths. A Motzkin path is a sequence $0 = x_0, x_1, x_2, \ldots, x_n = 0$ where $x_i \geqslant 0$ and $|x_i - x_{i-1}| \leqslant 1$, i.e., only steps of size $-1, 0, 1$ are allowed.

The formula of Flajolet [1] expresses the moments as a sum of Motzkin paths weighted by the Jacobi parameters

$$m_n = \sum_{\pi} v(\pi)$$

as follows. Given sequences $(a_n)_{n\geqslant 0}$, $(b_n)_{n\geqslant 0}$, $(c_n)_{n\geqslant 1}$, the weight of the steps are

$$k \to \begin{cases} k+1 & a_k \\ k & b_k \\ k-1 & c_k \end{cases}$$

and the weight of a path is the product of the weights of the steps. In other words,

$$\sum_{x_1, x_2, \ldots, x_n - 1} J(0, x_1)\, J(x_1, x_2) \cdots J(x_{n-1}, 0)$$

where

$$J = \begin{bmatrix} a_0 & b_0 & & & \\ c_1 & a_1 & b_1 & & \\ & c_2 & a_2 & b_2 & \\ & & \ddots & \ddots & \ddots \end{bmatrix}$$

is the associated Jacobi matrix.

These numbers are computed recursively by storing the entries $J^n(0, k)$ of partial sums for $k = 1, 2, \ldots, n$ in an array. Given sequences $(a_n)_{n\geqslant 0}$, $(b_n)_{n\geqslant 0}$, $(c_n)_{n\geqslant 1}$ and the sums of paths of length n in a vector $v = (v_0, v_1, \ldots, v_n)$, the path sums of length $n+1$ are given by the sum

$$\begin{bmatrix} v_n a_n \\ v_{n-1} a_{n-1} \\ v_{n-2} a_{n-2} \\ \vdots \\ v_0 a_0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ v_n b_n \\ v_{n-1} b_{n-1} \\ \vdots \\ v_1 b_1 \\ v_0 b_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ v_n c_n \\ \vdots \\ v_2 c_2 \\ v_1 c_1 \end{bmatrix}$$

6 ⟨*package PARRPKG PathArrayPackage* 6⟩≡ (32)

```
)abbrev package PARRPKG PathArrayPackage
++ Author: Franz Lehner lehner@math.tugraz.at
++ Date Created: 15 April 2011
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   A package for weighted Motzkin paths over a ring.
PathArrayPackage(R : Ring) : Exports == Implementation where
  S==> List R
  Exports ==> with
    motzkinPathArray : (Stream R, Stream R, Stream R) -> Stream S
      ++ \spad{motzkinPathArray([a0, a1, ...], [b0, b1, ...], [c1, c2, ...])}
      ++ computes Flajolet's Motzkin
      ++ path array from the sequences ai, bi, ci.
    jacobiPathArray : (Stream R, Stream R) -> Stream S
      ++ \spad{jacobiPathArray([b0, b1, ...], [c1, c2, ...])}
      ++ computes Flajolet's Motzkin
      ++ path array from the sequences [1, 1, ...], bi, ci.
    bottom : Stream S -> Stream R
      ++ bottom(x) returns the bottom entry of each column.

  Implementation ==> add
    -- function to compute next row
```

```
nextMotzkinRow(vv : S, aa : Stream R, bb : Stream R, cc : Stream R
             ) : S ==
    -- build a new list
    vvnew : S := empty()
    vva := cons(0, vv)
    aa := cons(0, aa)
    vvb := vv
    vvc := rest vv
    while not empty? vvc repeat
        vi : R := first vva*first aa + first vvb*first bb +
                  first vvc*first cc
        vvnew := cons(vi, vvnew)
        vva := rest vva
        vvb := rest vvb
        vvc := rest vvc
        aa := rest aa
        bb := rest bb
        cc := rest cc
    vi := first vva*first aa + first vvb*first bb
    vvnew := cons(vi, vvnew)
    vva := rest vva
    aa := rest aa
    vvnew := cons(first vva*first aa, vvnew)
    vvnew := reverse! vvnew

nextJacobiRow(vv : S, bb : Stream R, cc : Stream R) : S ==
    -- build a new list
    vvnew : S := empty()
    vva := cons(0, vv)
    vvb := vv
    vvc := rest vv
    while not empty? vvc repeat
        vi : R := first vva + first vvb*first bb +first vvc*first cc
        vvnew := cons(vi, vvnew)
        vva := rest vva
        vvb := rest vvb
        vvc := rest vvc
        bb := rest bb
        cc := rest cc
    vi := first vva + first vvb*first bb
    vvnew := cons(vi, vvnew)
    vva := rest vva
    vvnew := cons(first vva, vvnew)
    vvnew := reverse! vvnew

motzkinPathArray(aa : Stream R, bb : Stream R, cc : Stream R
               ) : Stream S ==
    start : S := [1]
    stream( (vv : S) : S +-> nextMotzkinRow(vv, aa, bb, cc), start)

jacobiPathArray(bb : Stream R, cc : Stream R) : Stream S ==
    start : S := [1]
    stream( (vv : S) : S +-> nextJacobiRow(vv, bb, cc), start)

bottom(x : Stream S) : Stream R ==
    map(first$S, x)$StreamFunctions2(S, R)
```

6. package MOMPKG MomentPackage

This package contains the core transformations between moments and various cumulants. When possible, the power series stream operations from the packages `StreamTaylorSeriesOperations` and `StreamExponentialSeriesOperatic` (e.g., for log, exp and Lagrange inversion) are used.

8 ⟨*package MOMPKG MomentPackage* 8⟩≡ (32)

```
)abbrev package MOMPKG MomentPackage
++ Author: Franz Lehner lehner@math.tugraz.at
++ Date Created: 2008
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   An auxiliary package for various moment and cumulant transformations
++   used in \spad{Distribution}.
MomentPackage(R : CommutativeRing) : Exports == Implementation where
  JAC2 ==> Record(an : R, bn : R)
  JAC2SEQ ==> Stream JAC2
  JACSEQ2 ==> Record(an : Stream R, bn : Stream R)
  JACLIST ==> Record(an : List R, bn : List R)
  STSOR ==> StreamTaylorSeriesOperations R
  STESOR ==> StreamExponentialSeriesOperations R
  STRREC ==> Record(puiseux : Fraction Integer, laurent : Fraction Integer,
                    coef : Sequence R)
  SUPR ==> SparseUnivariatePolynomial R
  SUPRpq ==> Record(first : SUPR, second : SUPR)
  RECSRM ==> Record(momt : List SUPR, cum : List R, mom : Stream R)
  Exports ==> with
    cumulant2moment : Sequence R -> Sequence R
      ++ \spad{cumulant2moment(cc)} computes the sequence of moments
      ++ from the sequence of classical cumulants cc
    moment2cumulant : Sequence R -> Sequence R
      ++ \spad{moment2cumulant(mm)} computes the sequence of classical
      ++ cumulants from the sequence of moments mm

    if R has Field then
      moment2nthJacobi : List R -> JACLIST
        ++ \spad{moment2nthJacobi(mm)} computes the list of Jacobi parameters
        ++ from the list of moments mm as far as possible.
      moment2jacobi : Sequence R -> JACSEQ2
        ++ \spad{moment2jacobi(mm)} computes the Jacobi parameters
        ++ as pair of streams.
      moment2jacobi2 : Sequence R -> JAC2SEQ
        ++ \spad{moment2jacobi2(mm)} computes the Jacobi parameters
        ++ as stream of pairs $(an, bn)$.

    freeCumulant2moment : Sequence R -> Sequence R
      ++ \spad{freeCumulant2moment(cc)} computes the sequence of moments
      ++ from the sequence of free cumulants cc
    moment2freeCumulant : Sequence R -> Sequence R
      ++ \spad{moment2freeCumulant(mm)} computes the sequence of free
      ++ cumulants from the sequence of moments mm
    booleanCumulant2moment : Sequence R -> Sequence R
      ++ \spad{booleanCumulant2moment(cc)} computes the sequence of moments
      ++ from the sequence of boolean cumulants cc
    moment2booleanCumulant : Sequence R -> Sequence R
      ++ \spad{moment2booleanCumulant(mm)} computes the sequence of boolean
      ++ cumulants from the sequence of moments mm
```

```
    monotoneCumulant2moment : Sequence R -> Sequence R
      ++ \spad{monotoneCumulant2moment(hh)} computes the sequence of moments
      ++ from the sequence of monotone cumulants hh
    monotoneCumulant2momentPoly : Sequence R -> Sequence SUPR
      ++ \spad{monotoneCumulant2momentPoly(hh)} computes the sequence
      ++ of moment polynomials $m_n(t)$ from the sequence of monotone
      ++ cumulants hh
    moment2monotoneCumulant : Sequence R -> Sequence R
      ++ \spad{moment2monotoneCumulant(mm)} computes the sequence of
      ++ monotone cumulants from the sequence of moments mm
--    qcumulant2nthmoment : (R, Sequence R, NonNegativeInteger) -> R
--      ++ \spad{qcumulant2nthmoment(q, cc, n)} computes the nth moment
--      ++ from the sequence of reduced q-cumulants cc

    hankelDeterminant : (Sequence R, NonNegativeInteger) -> R
      ++ \spad{hankelDeterminant(x, n)} returns the nth Hankel determinant
      ++ of the sequence \spad{x}.
    jacobi2poly : (Stream R, Stream R) -> Stream SUPR
      ++ \spad{jacobi2poly(aa, bb)} returns the stream
      ++ of orthogonal polynomials corresponding to the
      ++ Jacobi parameters \spad{a_n} and \spad{b_n}.

    if R has Algebra Fraction Integer then
      moment2Stransform : Sequence R -> STRREC
        ++ \spad{moment2Stransform(x)} returns the Puiseux and Laurent order
        ++ and coefficients of the S transform of x
      moment2monotoneCumulant : Sequence R -> Sequence R
        ++ \spad{moment2monotoneCumulant(x)} returns the sequence
        ++ of monotone cumulants of the moment sequence x

  Implementation ==> add
```

⟨*MomentPackage implementation* 9a⟩

6.1. MomentPackage Classical cumulants. Moments and cumulants are related via their exponential generating functions

$$\sum_{n=0}^{\infty} \frac{m_n}{n!} z^n = \exp \sum_{n=1}^{\infty} \frac{\kappa_n}{n!} z^n.$$

This is more convenient than the recursion

$$\kappa_n = \mu_n - \sum_{k=1}^{n-1} \binom{n-1}{k-1} \kappa_k \mu_{n-k}.$$

9a ⟨*MomentPackage implementation* 9a⟩≡ (8) 9b ▷

```
  cumulant2moment(cum : Sequence R) : Sequence R ==
      sequence rest exp0$STESOR stream cum
```

The equivalent form of the recursion is

$$\mu_n = \kappa_n + \sum_{k=1}^{n-1} \binom{n-1}{k} \mu_k \kappa_{n-k}$$

but we rather use the logarithm of exponential power series.

9b ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁ 9a 10a ▷

```
  moment2cumulant(mom : Sequence R) : Sequence R ==
      sequence rest log1$STESOR stream mom
```

6.2. MomentPackage Free Cumulants. Free cumulants can be defined via their generating function. Given the ordinary moment generating function

$$M(z) = 1 + \sum_{n=1}^{\infty} m_n z^n$$

the free cumulant series

$$C(z) = 1 + \sum_{n=1}^{\infty} c_n z^n.$$

satisfies the equations

$$C(zM(z)) = M(z) \qquad M(z/C(z)) = C(z).$$

In the first case the function $f(z) = zM(z)$ satisfies

$$C(f(z)) = f(z)/z$$

and can be directly translated into a call to `lagrange` from `StreamTaylorSeriesOperations`, which can solve solve the general equation

$$f(z) = z\, g(f(z))$$

with function $g(z) = C(z)$.

10a ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁9b 10b▷

```
freeCumulant2moment(cum : Sequence R) : Sequence R ==
    import from STSOR
    coerce(rest rest lagrange concat(1@R, stream cum))
```

In the other direction, we can write

$$\frac{z}{M(z/C(z))} = \frac{z}{C(z)}$$

and take $f(z) = \frac{z}{C(z)}$ and $g(z) = \frac{1}{M(z)}$.

10b ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁10a 10c▷

```
moment2freeCumulant(mom : Sequence R) : Sequence R ==
    -- first divide
    g : Stream R := cons(1$R, stream mom)
    g1:Union(Stream R, "failed") := recip(g)$STSOR -- value1 --$
    -- g1 case "failed" => "failed"
    f : Stream R := lagrange(g1::Stream R)
    -- now we have f=z/C(z)
    f1:Union(Stream R, "failed") := recip rest f
    -- f1 case "failed" => "failed"
    coerce rest (f1::Stream R)
```

6.3. MomentPackage Boolean cumulants. The boolean cumulant generating function $B(z)$ and the moment generating function $M(z)$ are related by

$$M(z) = \frac{1}{1 - B(z)} \qquad B(z) = 1 - \frac{1}{M(z)}$$

10c ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁10b 11a▷

```
booleanCumulant2moment(x : Sequence R) : Sequence R ==
    rec : Stream R := cons(1$R, stream(-x))
    boo:Union(Stream R, "failed") :=
        recip(rec)$STSOR
    rest sequence (boo::Stream R)

moment2booleanCumulant(x : Sequence R) : Sequence R ==
    boo : Stream R := cons(1$R, stream x)
    mom : Union(Stream R, "failed") :=
        recip(boo)$STSOR
    rest (- sequence (mom::Stream R))
```

6.4. MomentPackage Jacobi Parameters and Orthogonal Polynomials.

11a ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁10c 11b ▷

```
hankelDeterminant(x : Sequence R, n : NonNegativeInteger) : R ==
    import from HankelPackage R
    determinant HankelMatrix cons(1, firstn(x, 2*n))
```

The Jacobi parameters are only available in the case of a Field.

11b ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁11a 12b ▷

```
if R has Field then
    ⟨MomentPackage moment2jacobi 11c⟩
```

The Jacobi parameters are defined by

$$xP_n(x) = P_{n+1}(x) + a_n P_n(x) + b_n P_{n-1}(x)$$

they are computed via the recursion...

First a finite version using lists.

11c ⟨*MomentPackage moment2jacobi* 11c⟩≡ (11b) 12a ▷

```
moment2nthJacobi(mom : List R) : JACLIST ==
    N : Integer := (#mom)::Integer
    G : List R := mom -- add 1 to the moments
    aa : List R := empty()
    bb : List R := empty()
    while N>1 repeat
        -- assign the new coefficients
        ak : R := first G
        bk : R := second G - ak*ak
        aa := concat(aa, ak)
        bb := concat(bb, bk)
        -- invert G and store it in H.
        -- the first three entries are clear
        H : List R := [-ak, -bk]
        k, l : Integer
        for k in 3..N repeat
            Hk : R := -G k
            for l in 1..k-1 repeat
                Hk := Hk - (G l) * H(k-l)
            H := concat(H, Hk)
        if R has Field then
            h : R
            G := [-h/bk for h in  H ]
            G := rest rest G
        N := N-2
    [aa, bb]
```

Here is a recursive version. The algorithm is very simple: if

$$F(z) = 1 + \sum_{n=1}^{\infty} c_n z^n = \cfrac{1}{1 - a_1 z - \cfrac{b_1 z^2}{1 - a_2 z - \ldots}}$$

then

$$\frac{1}{F(z)} = 1 - a_1 z - b_1 z^2 G_2(z)$$

and this can be recursively applied to $G_2(z)$.

12a ⟨*MomentPackage moment2jacobi* 11c⟩+≡ (11b) ◁ 11c

```
moment2jacobi2(mom : Sequence R) : JAC2SEQ == delay
    gseries : Stream R := cons(1, (stream$(Sequence R)) mom)
    -- first divide
    gseries1 := (recip(gseries)$STSOR)::Stream R
    gseries1 := rest gseries1
    a : R := - first gseries1
    gseries1 := rest gseries1
    b : R := - first gseries1
    b1 : R := - inv b
    cons([a, b]$JAC2, moment2jacobi2( - inv b * sequence rst gseries1)
        )$JAC2SEQ

moment2jacobi(mom : Sequence R) : JACSEQ2 ==
    res : JAC2SEQ := moment2jacobi2 mom
    [map(y+->y an, res)$(StreamFunctions2(JAC2, R)),
        map(y+->y bn, res)$(StreamFunctions2(JAC2, R))]$JACSEQ2
```

The orthogonal polynomials satisfy the recursion

$$xP_k(x) = P_{k+1}(x) + a_k P_k(x) + b_{k-1} P_{k-1}(x)$$

where per definitionem we set $b_{-1} = 0$ and we assume that the sequences of Jacobi coefficients both start with index 0. In an auxiliary function we carry pairs of subsequent polynomials P_n, P_{n-1} from which we obtain the pair P_{n+1}

$$P_{n+1} = (x - a_n)P_n - b_{n-1}P_{n-1}$$

12b ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁ 11b 12c ▷

```
xsup := monomial(1, 1)$SUPR

jacobi2polypq(aa : Stream R, bb : Stream R, p : SUPR, q : SUPR
             ) : Stream SUPRpq == delay
    q1 : SUPR := (xsup - (first aa)*1$SUPR)*q - first bb*p
    pq1 : SUPRpq := [q, q1]
    cons(pq1, jacobi2polypq(rest aa, rest bb, q, q1))
```

From the stream of pairs we extract the first elements.

12c ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁ 12b 13 ▷

```
jacobi2poly(aa : Stream R, bb : Stream R) : Stream SUPR ==
    p0 : SUPR := 1
    p1 : SUPR := xsup - (first aa)*p0
    res : Stream SUPRpq := jacobi2polypq(rest aa, bb, p0, p1)
    map( (pp : SUPRpq) : SUPR +-> pp.first, res
       )$StreamFunctions2(SUPRpq, SUPR)
```

6.5. MomentPackage S-transform. Let $\psi(z) = \sum_{n=1}^{\infty} m_n z^n$ be the moment generating function and $\chi(z)$ its compositional inverse, then $S(z) = \chi(z)\frac{1+z}{z}$ is called *S-transform*. If X and Y are free with nonvanishing first moment, then $S_{XY}(z) = S_X(z)S_Y(z)$. If both have vanishing first moments, then the resulting distribution is trivial.

If one of them has vanishing first moment, we have the following modification [5]. Assume that X has vanishing first moment and $\psi(z) = \sum_{n=2}^{\infty} m_n z^n$, then $\psi(z)$ has two inverses (Puiseux series in $\pm\sqrt{z}$). Let $\tilde{\psi}(z) = \sqrt{\psi(z)} = z\sqrt{m_2}\sqrt{1 + \frac{m_3}{m_2}z + \ldots}$, then $\tilde{\psi}$ has a compositional inverse $\tilde{\chi}(z)$, i.e., $\tilde{\psi}(\tilde{\chi}(z)) = z$ and this means that $\psi(\tilde{\chi}(z)) = z^2$, therefore $\tilde{\chi}(z) = \chi(z^2)$. The S-transform leads to Puiseux series and therefore the ring underlies restrictions. The return value is a record containing the Puiseux exponent, the Laurent exponent and is suitable for Puiseux series operations in the S-transform package in section 7.

13 ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁12c 14 ▷

```
if R has Algebra Fraction Integer then
  moment2Stransform(x : Sequence R) : STRREC ==
      mom := stream x
      if zero? first mom then
          -- case of zero mean: take square root
          mom2 : Stream R := cons(0, powern(1/2, rest mom)$STSOR)
          -- invert
          chi2 : Stream R := revert(mom2)$STSOR
          chi2s : Sequence R := sequence chi2
          -- S2 = chi2+chi2/z^2
          S2 : Sequence R := cons(0, chi2s) + rest chi2s
          [1/2, -1, S2]$STRREC
        else
          -- case of nonzero mean
          mom := cons(0, mom)
          chi : Stream R := revert(mom)$STSOR
          -- S = chi + chi/z
          S : Sequence R := sequence chi + sequence rest chi
          [1, 0, S]$STRREC
```

6.6. Monotone cumulants. The monotone cumulants were defined by Hasebe and Saigo [2] and are determined by the infinite system of differential equations

$$\begin{aligned} m_n(0) &= 0 \\ \frac{dm_n(t)}{dt} &= \sum_{k=1}^{n} k r_{n-k+1} m_{k-1}(t) \\ &= r_n + \sum_{k=2}^{n} k r_{n-k+1} m_{k-1}(t) \end{aligned}$$

for $n \geqslant 1$, where $m_0(t) = 1$.

6.6.1. *From moments to monotone cumulants.* Given a moment sequence m_n, the monotone cumulants can be computed inductively as follows. Given r_k and the polynomials $m_k(t)$ for $k \leqslant n-1$, we integrate the equation on the interval $[0,1]$ and obtain

$$r_n = m_n - \int_0^1 \sum_{k=2}^{n} k r_{n-k+1} m_{k-1}(t)\, dt$$

and then

$$m_n(t) = t r_n + \int_0^t \sum_{k=2}^{n} k r_{n-k+1} m_{k-1}(s)\, ds. \tag{1}$$

The ring must be an `Algebra Fraction Integer` in order for integration to work. We keep the information in a record `RECSRM` with entries

momt: the list of moment polynomials computed so far in reverse order
cum: the list of cumulants computed so far in reverse order
mom: the stream of remaining moments

The function `moment2monotoneCumulantGenerator` then implements the two steps above.

14 ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁13 15▷

```
-- monotone cumulants
  t := monomial(1, 1)$SUPR
  moment2monotoneCumulantGenerator(srm : RECSRM) : RECSRM ==
      mt : List SUPR := srm.momt
      rr : List R := srm.cum
      mm : Stream R := srm.mom
      n : Integer := #rr + 1
      tmp : SUPR := 0
      for k in 2..n for r in rr for m in reverse mt repeat
          tmp := tmp + k*m*r
      mnt : SUPR := integrate tmp
      rn : R := first mm - mnt 1
      mnt := mnt + t*rn
      rrnew : List R := cons(rn, rr)
      mtnew : List SUPR := cons(mnt, mt)
      return [mtnew, rrnew, rest mm]

  moment2monotoneCumulant(mm : Sequence R) : Sequence R ==
      m1t : SUPR := monomial(first mm, 1)$SUPR
      r1 := first mm
      mtr1 : RECSRM := [[m1t], [r1], rest stream mm]$RECSRM
      res : Stream RECSRM :=
          stream(moment2monotoneCumulantGenerator, mtr1)
      res1 : Stream R := map( (s : RECSRM) : R +-> first (s.cum), res
                            )$StreamFunctions2(RECSRM, R)
      sequence res1
```

6.6.2. *From monotone cumulants to moments.* For the converse direction we simply iterate the integral formula (1). We compute the stream of $m_n(t)$ and finally evaluate it at $t = 1$.

15 ⟨*MomentPackage implementation* 9a⟩+≡ (8) ◁14

```
-- monotone cumulants
  RECMM ==> Record(cum : Stream R, momt : List SUPR)
  t := monomial(1, 1)$SUPR
  monotoneCumulant2momentGenerator(srm : RECMM) : RECMM ==
      mt : List SUPR := srm.momt
      rr : Stream R := srm.cum
      n : Integer := #mt + 1
      NNI==> NonNegativeInteger
      tmp : SUPR := 0
      for k in 1..n-1 for m in mt repeat
          tmp := tmp + (n+1-k)*m*first rr
          rr := rest rr
      mnt : SUPR := t*first rr + integrate tmp
      mtnew : List SUPR := cons(mnt, mt)
      return [srm.cum, mtnew]

  monotoneCumulant2moment(rr : Sequence R) : Sequence R ==
      m1t : SUPR := monomial(first rr, 1)$SUPR
      mtr1: RECMM := [stream rr, [m1t]]$RECMM
      res : Stream RECMM := stream(monotoneCumulant2momentGenerator, mtr1)
      res1 : Stream R := map( (s : RECMM) : R +-> (first (s.momt)).(1),
                              res)$StreamFunctions2(RECMM, R)
      sequence res1

  monotoneCumulant2momentPoly(rr : Sequence R) : Sequence SUPR ==
      m1t : SUPR := monomial(first rr, 1)$SUPR
      mtr1 : RECMM := [stream rr, [m1t]]$RECMM
      res : Stream RECMM := stream(monotoneCumulant2momentGenerator, mtr1)
      res1 : Stream SUPR := map( (s : RECMM) : SUPR +-> (first (s.momt)),
                                 res)$StreamFunctions2(RECMM, SUPR)
      sequence res1
```

7. package STRANS STransformPackage

In the case of a field we can also do the S-transform. As discussed above (Section 6.5, there are cases where the S-transform is a Puiseux series.

16 ⟨*package STRANS STransformPackage* 16⟩≡ (32) 18 ▷

```
)abbrev package STRANS STransformPackage
++ Author: Franz Lehner lehner@math.tugraz.at, Waldek Hebisch
++ Date Created: 2010
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   A package to computes Taylor and Puiseux series of S-transforms.
STransformPackage(R, UTSR, ULSR, UPSR) : Exports == Implementation where
  R : Join(CommutativeRing, Algebra Fraction Integer)
  UTSR : UnivariateTaylorSeriesCategory R
  ULSR : UnivariateLaurentSeriesConstructorCategory(R, UTSR)
  UPSR : UnivariatePuiseuxSeriesConstructorCategory(R, ULSR)
  DISTR ==> Distribution R
  STSOR ==> StreamTaylorSeriesOperations R
  TERM ==> Record(k : Fraction Integer, c : Coef)
  STRREC ==> Record(order : Fraction Integer, coeff : Stream R)
  Exports ==> with
    STransform1 : (DISTR) -> UTSR
      ++ \spad{STransform(x)} returns the Taylor expansion the S-transform
      ++ \spad{S(z)} of the distribution \spad{x} if the mean is nonzero.
    STransform2 : (DISTR) -> UPSR
      ++ \spad{STransform(x)} returns the Puiseux expansion of the
      ++ S-transform \spad{S(z)} if the mean is zero.
    STransform : DISTR -> UPSR
      ++ \spad{STransform(x)} returns the Puiseux expansion of the
      ++ S-transform \spad{S(z)}.
    distributionBySTransform : UPSR -> DISTR
      ++ of the distributions \spad{x} if the mean is zero
      ++ \spad{distributionBySTransform(x)} returns the distribution
      ++ with S-transform x.
    freeMultiplicativeConvolution : (DISTR, DISTR) -> DISTR
      ++ \spad{freeMultiplicativeConvolution(x, y)} returns the free
      ++ multiplicative convolution of the distributions
      ++ \spad{x} and \spad{y}.

  Implementation ==> add
    STransform1(x : DISTR) : UTSR ==
        mom : Stream R := stream moments x
        zero? first mom => error "mean is zero!"
        mom := cons(0, mom)
        chi : Stream R := revert(mom)$STSOR
        -- S = chi + chi/z
        res := sequence chi + sequence rest chi
        series stream res

    STransform2(x : DISTR) : UPSR ==
        mom : Stream R := stream moments x
        not zero? first mom => error "mean is nonzero!"
        -- divide by z^2, take square root and multiply with z
        mom2 := cons(0, powern(1/2, rest mom)$STSOR)
        chi2 := revert(mom2)$STSOR
        res2 : UTSR := series chi2
```

```
    -- S2 = chi2+chi2/z^2
    S2 : ULSR := laurent(-2, res2)+laurent(0, res2)
    puiseux(1/2, S2)$UPSR

-- two in one
STransform(x : DISTR) : UPSR ==
    mom : Stream R := stream moments x
    if zero? first mom then
        -- case of mean zero
        mom2 := cons(0, powern(1/2, rest mom)$STSOR)
        chi2 := revert(mom2)$STSOR
        res2 : UTSR := series chi2
        -- S2 = chi2+chi2/z^2
        S2 : ULSR := laurent(-2, res2)+laurent(0, res2)
        puiseux(1/2, S2)$UPSR
      else
        -- case of mean nonzero
        mom := cons(0, mom)
        chi : Stream R := revert(mom)$STSOR
        -- S = chi + chi/z
        res := sequence chi + sequence rest chi
        S : UTSR := series stream res
        puiseux(1, laurent(0, S))$UPSR --$
```

Getting back from the S-transform $S(z)$ to the distribution, the procedure is

$$\chi(z) = S(z)\frac{z}{1+z}$$

then $\psi(z) = \chi^{-1}(z) = \sum_{n=1}^{\infty} m_n z^n$.

There are two possibilities for $S(z)$: If the first moment is nonzero, then $S(z)$ is a Taylor series

$$\frac{1}{m_1} + \frac{-m_2 + {m_1}^2}{{m_1}^3}\, z + \frac{-m_1\, m_3 + 2\, {m_2}^2 - {m_1}^2\, m_2}{{m_1}^5}\, z^2 + O\left(z^3\right)$$

with nonzero constant term.

If the first moment vanishes, then $S(z)$ is a Puiseux series

$$\frac{1}{\sqrt{m_2}}\, z^{\left(-\frac{1}{2}\right)} - \frac{m_3}{2\, {m_2}^2} + \frac{-4\, m_2\, m_4 + 5\, {m_3}^2 + 8\, {m_2}^3}{8\, {m_2}^3\, \sqrt{m_2}}\, z^{\frac{1}{2}} + O\left(z^1\right)$$

otherwise the first and second moments vanish and it is not the S-transform of a nontrivial probability measure.

In the first case we must multiply with

$$\frac{z}{1+z} = z - z^2 + z^3 - z^4 + - \cdots$$

then invert and extract coefficients to get the moments. In the second case we must first evaluate at z^2, multiply with

$$\frac{z^2}{1+z^2} = z^2 - z^4 + z^6 - z^8 + - \cdots$$

invert and evaluate at $\sqrt{z}$, i.e., call `puiseux(1/2, f)$UPXS(EXPR INT, z, 0)`.

18 ⟨*package STRANS STransformPackage* 16⟩+≡ (32) ◁16

```
distributionBySTransform(S : UPSR) : DISTR ==
    if order(S) = 0 then
      -- Taylor case
      S1 : UTSR := retract retract S
      -- multiply with z/(1+z)
      z1z : UTSR := series(cons(0, repeating([1, -1])$(Stream R)))
      chi : Stream R := coefficients(S1*z1z)
      psi : Stream R := revert(chi)$STSOR
      return distributionByMoments sequence rest psi
    else if not order(S) = -1/2 then
      error "Not an S-transform"
    if not rationalPower S = 1/2 then
      error "Not an S-transform"
    -- Puiseux case. Evaluating at z^2 is the same
    -- as taking the Laurent representative
    S2 := laurentRep S
    -- multiply with z^2/(1+z^2)
    z1z : UTSR := series(cons(0, repeating([0, 1, 0, -1])$(Stream R)))
    chi2 : UTSR := retract(S2*laurent(0, z1z))
    psi2 := revert(coefficients chi2)$STSOR
    psi := powern(2::Fraction Integer, psi2)
    distributionByMoments sequence rest psi
```

8. category DISTCAT DistributionCategory

19 ⟨*category DISTCAT DistributionCategory* 19⟩≡ (32)

```
)abbrev category DISTCAT DistributionCategory
++ Author: Franz Lehner lehner@math.tugraz.at, Waldek Hebisch
++ Date Created: 2009
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   Category of distributions formally given by moments.
DistributionCategory(R : CommutativeRing) : Category == Exports where
  NNI ==> NonNegativeInteger
  PI ==> PositiveInteger
  STRREC ==> Record(puiseux : Fraction Integer, laurent : Fraction Integer,
                    coef : Sequence R)
  SUPR ==> SparseUnivariatePolynomial R
  SUPFR ==> SparseUnivariatePolynomial Fraction R
  Exports ==> SetCategory with
    0 : constant -> %
      ++ \spad{0} is the Dirac distribution
    moment : (%, NNI) -> R
      ++ \spad{moment(x, n)} returns the n-th moment
      ++ of the distribution \spad{x}
    classicalCumulant : (%, PI) -> R
      ++ \spad{classicalCumulant(x, n)} returns the n-th classical cumulant
      ++ of the distribution \spad{x}
    freeCumulant : (%, PI) -> R
      ++ \spad{freeCumulant(x, n)} returns the n-th free cumulant
      ++ of the distribution \spad{x}
    booleanCumulant : (%, PI) -> R
      ++ \spad{booleanCumulant(x, n)} returns the n-th boolean cumulant
      ++ of the distribution \spad{x}
    moments : % -> Sequence R
      ++ \spad{moments(x)} returns the sequence of moments
      ++ of the distribution \spad{x}
    classicalCumulants : % -> Sequence R
      ++ \spad{classicalCumulants(x)} returns sequence of classical cumulants
      ++ of the distribution \spad{x}
    freeCumulants : % -> Sequence R
      ++ \spad{freeCumulants(x)} returns the sequence of free cumulants
      ++ of the distribution \spad{x}.
    booleanCumulants : % -> Sequence R
      ++ \spad{booleanCumulants(x)} returns the sequence of boolean cumulants
      ++ of the distribution \spad{x}.
    hankelDeterminants : % -> Stream R
      ++ \spad{hankelDeterminants(x)} returns the stream of hankel
      ++ determinants of the distribution \spad{x}.

    if R has Algebra Fraction Integer then
--      monotoneCumulant : (%, PI) -> R
--        ++ \spad{monotoneCumulant(x, n)} returns the n-th monotone cumulant
--        ++ of the distribution \spad{x}
        monotoneCumulants : % -> Sequence R
          ++ \spad{monotoneCumulants(x)} returns the sequence of monotone
          ++ cumulants of the distribution \spad{x}.

    if R has Field then
```

```
    jacobiParameters : % -> Record(an : Stream R, bn : Stream R)
      ++ \spad{jacobiParameters(x)} returns the pair of streams
      ++ of Jacobi parameters of the distribution \spad{x}.
    orthogonalPolynomials : % -> Stream SUPR
      ++ \spad{orthogonalPolynomials(x)} returns the stream of
      ++ orthogonal polynomials.

  else if R has IntegralDomain then
    jacobiParameters : % -> Record(an : Stream Fraction R,
                                   bn : Stream Fraction R)
      ++ \spad{jacobiParameters(x)} returns the pair of streams
      ++ of Jacobi parameters of the distribution \spad{x}.
    orthogonalPolynomials : % -> Stream SUPFR
      ++ \spad{orthogonalPolynomials(x)} returns the stream of
      ++ orthogonal polynomials.

-- operations
classicalConvolution : (%, %) -> %
  ++ \spad{classicalConvolution(x, y)} returns the classical convolution
  ++ of the distributions \spad{x} and \spad{y}
freeConvolution : (%, %) -> %
  ++ \spad{freeConvolution(x, y)} returns the free convolution
  ++ of the distributions \spad{x} and \spad{y}
booleanConvolution : (%, %) -> %
  ++ \spad{booleanConvolution(x, y)} returns the boolean convolution
  ++ of the distributions \spad{x} and \spad{y}
monotoneConvolution : (%, %) -> %
  ++ \spad{monotoneConvolution(x, y)} returns the monotone convolution
  ++ of the distributions \spad{x} and \spad{y}
_^: (%, PositiveInteger) -> %
  ++ \spad{x^k} constructs the distribution of the \spad{k}th power
  ++ of the random variable with distribution \spad{X}
  ++ by picking every k-th moment.
orthogonalConvolution : (%, %) -> %
  ++ \spad{orthogonalConvolution(x, y)} returns the orthogonal convolution
  ++ of the distributions \spad{x} and \spad{y}

subordinationConvolution : (%, %) -> %
  ++ \spad{subordinationConvolution(x, y)} returns the subordination convolution
  ++ of the distributions \spad{x} and \spad{y}
```

9. domain DISTRO Distribution

21 ⟨*domain DISTRO Distribution* 21⟩≡ (32) 26a ▷

```
)abbrev domain DISTRO Distribution
++ Author: Franz Lehner lehner@math.tugraz.at, Waldek Hebisch
++ Date Created: 2009
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   Domain for distributions formally given by moments.
++   moments and different kinds of cumulants are
++   stored in streams and computed on demand.
Distribution(R : CommutativeRing) : Exports == Implementation where
  MPR ==> MomentPackage R
  NNI ==> NonNegativeInteger
  PI ==> PositiveInteger
  STSOR ==> StreamTaylorSeriesOperations R
  STRREC ==> Record(puiseux : Fraction Integer, laurent : Fraction Integer,
                    coef : Sequence R)
  SUPR ==> SparseUnivariatePolynomial R
  SUPFR ==> SparseUnivariatePolynomial Fraction R
  Exports ==> DistributionCategory(R) with
    distributionByMoments : Sequence R -> %
      ++ \spad{distributionByMoments(mm)} initiates a distribution with
      ++ given moments mm.
    distributionByMoments : Stream R -> %
      ++ \spad{distributionByMoments(mm)} initiates a distribution with
      ++ given moments \spad{mm}.
    distributionByEvenMoments : Sequence R -> %
      ++ \spad{distributionByEvenMoments(mm)} initiates a distribution with
      ++ given even moments \spad{mm} and odd moments zero.
    distributionByEvenMoments : Stream R -> %
      ++ \spad{distributionByEvenMoments(mm)} initiates a distribution with
      ++ given even moments \spad{mm} and odd moments zero.
    distributionByClassicalCumulants : Sequence R -> %
      ++ \spad{distributionByEvenMoments(kk)} initiates a distribution with
      ++ given classical cumulants \spad{kk}.
    distributionByClassicalCumulants : Stream R -> %
      ++ \spad{distributionByEvenMoments(kk)} initiates a distribution with
      ++ given classical cumulants \spad{kk}.
    distributionByFreeCumulants : Sequence R -> %
      ++ \spad{distributionByFreeCumulants(cc)} initiates a distribution with
      ++ given free cumulants \spad{cc}.
    distributionByFreeCumulants : Stream R -> %
      ++ \spad{distributionByFreeCumulants(cc)} initiates a distribution with
      ++ given free cumulants \spad{cc}.
    distributionByBooleanCumulants : Sequence R -> %
      ++ \spad{distributionByBooleanCumulants(bb)} initiates a distribution
      ++ with given Boolean cumulants \spad{bb}.
    distributionByBooleanCumulants : Stream R -> %
      ++ \spad{distributionByBooleanCumulants(bb)} initiates a distribution
      ++ with given Boolean cumulants \spad{bb}.
    distributionByJacobiParameters : (Sequence R, Sequence R) -> %
      ++ \spad{distributionByJacobiParameters(aa, bb)} initiates a
      ++ distribution with given Jacobi parameters \spad{[aa, bb]}.
    distributionByJacobiParameters : (Stream R, Stream R) -> %
      ++ \spad{distributionByJacobiParameters(aa, bb)} initiates a
```

```
    ++ distribution with given Jacobi parameters \spad{[aa, bb]}.

  booleanCumulantFromJacobi : (Integer, Sequence R, Sequence R) -> R
    ++ \spad{booleanCumulantFromJacobi(n, aa, bb)}
    ++ computes the \spad{n}th Boolean cumulant from
    ++ the given Jacobiparameters \spad{aa} and \spad{bb}.
  construct : (Sequence R, Sequence R, Sequence R, Sequence R) -> %
    ++ \spad{construct(mom, ccum, fcum, bcum)} constructs a distribution
    ++ with moments \spad{mom}, classical cumulants \spad{ccum},
    ++ free cumulants \spad{fcum} and boolean cumulants \spad{bcum}.
    ++ The user must make sure that these are consistent,
    ++ otherwise the results are unpredictable!

  if R has Algebra Fraction Integer then
    distributionByMonotoneCumulants : Sequence R -> %
      ++ \spad{distributionByMonotoneCumulants(hh)} initiates a
      ++ distribution with given monotone cumulants \spad{hh}.
    distributionByMonotoneCumulants : Stream R -> %
      ++ \spad{distributionByMonotoneCumulants(hh)} initiates a
      ++ distribution with given monotone cumulants \spad{hh}.
    distributionBySTransform : STRREC -> %
      ++ \spad{distributionBySTransform(series)} initiates a
      ++ distribution with given S-transform \spad{series}.
    distributionBySTransform : (Fraction Integer, Fraction Integer,
                                Sequence R) -> %
      ++ \spad{distributionBySTransform(series)} initiates a distribution
      ++ with given S-transform \spad{series}.
    freeMultiplicativeConvolution : (%, %) -> %
      ++ \spad{freeMultiplicativeConvolution(mu, nu)} computes
      ++ the free multiplicative convolution of the distributions
      ++ \spad{mu} and \spad{nu}.

Implementation ==> add
  Rep := Record(moments : Sequence R, ccumulants : Sequence R,
                fcumulants : Sequence R, bcumulants : Sequence R)
  rep(x : %) : Rep == x :: Rep
  per(r : Rep) : % == r :: %

  0 == distributionByMoments sequence repeating [0]

  distributionByMoments(mm : Sequence R) : % ==
      import from MomentPackage R
      per ([mm, moment2cumulant mm, moment2freeCumulant mm,
            moment2booleanCumulant mm]$Rep)

  distributionByMoments(mm : Stream R) : % ==
      distributionByMoments(sequence(mm)$(Sequence R))

  interlace : (Stream R, Stream R) -> Stream R
    -- interlace ([a1, a2, ...], [b2, b2, ...]) is the stream
    -- [a1, b1, a2, b2, ...]
  interlace(x : Stream R, y : Stream R) : Stream R == delay
      cons(first x, cons(first y, interlace(rest x, rest y)))

  distributionByEvenMoments(mm : Sequence R) : % ==
      distributionByMoments sequence interlace (repeating [0], stream mm)

  distributionByEvenMoments(mm : Stream R) : % ==
      distributionByEvenMoments(sequence(mm)$(Sequence R))

  distributionByClassicalCumulants(cc : Sequence R) : % ==
```

```
    import from MomentPackage R
    mm := cumulant2moment cc
    per ([mm, cc, moment2freeCumulant mm, moment2booleanCumulant mm]$Rep)

distributionByClassicalCumulants(mm : Stream R) : % ==
    distributionByClassicalCumulants(sequence(mm)$(Sequence R))

distributionByFreeCumulants(fc : Sequence R) : % ==
    import from MomentPackage R
    mm := freeCumulant2moment fc
    per ([mm, moment2cumulant mm, fc, moment2booleanCumulant mm]$Rep)

distributionByFreeCumulants(mm : Stream R) : % ==
    distributionByFreeCumulants(sequence(mm)$(Sequence R))

distributionByBooleanCumulants(bc : Sequence R) : % ==
    import from MomentPackage R
    mm := booleanCumulant2moment bc
    per ([mm, moment2cumulant mm, moment2freeCumulant mm, bc]$Rep)

distributionByBooleanCumulants(mm : Stream R) : % ==
    distributionByBooleanCumulants(sequence(mm)$(Sequence R))

booleanCumulantFromJacobi(n : Integer, aa : Sequence R,
                          bb : Sequence R) : R ==
    zero? n => 1
    one? n => aa 1
    bb 1*booleanCumulantFromJacobi(n-2, rest aa, rest bb)

distributionByJacobiParameters(aa : Sequence R, bb : Sequence R) : % ==
    rior : Stream List R := jacobiPathArray(stream aa, stream bb
                                            )$(PathArrayPackage R)
    mom : Stream R := rest bottom rior
    distributionByMoments(sequence mom)

distributionByJacobiParameters(aa : Stream R, bb : Stream R) : % ==
    distributionByJacobiParameters(sequence(aa)$(Sequence R), _
        sequence(bb)$(Sequence R))

construct(mom : Sequence R, ccum : Sequence R, fcum : Sequence R,
          bcum : Sequence R) : % ==
    per([mom, ccum, fcum, bcum]$Rep)

if R has Algebra Fraction Integer then
  monotoneCumulants(x : %) : Sequence R ==
      moment2monotoneCumulant(moments x)$(MomentPackage R)

  distributionByMonotoneCumulants(mc : Sequence R) : % ==
      import from MomentPackage R
      mm := monotoneCumulant2moment mc
      per ([mm, moment2cumulant mm, moment2freeCumulant mm,
            moment2booleanCumulant mm]$Rep)

  distributionByMonotoneCumulants(mm : Stream R) : % ==
      distributionByMonotoneCumulants(sequence(mm)$(Sequence R))

  distributionBySTransform(puiseux : Fraction Integer, _
    laurent : Fraction Integer, coef : Sequence R) : % ==
      psi : Stream R
      if puiseux = 1 then
        -- the series z/(1+z)
```

```
        z1z : Stream R := cons(0, repeating([1, -1])$(Stream R))
        chi : Stream R := (stream(coef)*z1z)$STSOR
        psi := revert(chi)$STSOR
        return distributionByMoments sequence rest psi
      else if not puiseux = 1/2 then
        error "Not an S-transform"
      else if not laurent = -1 then
        error "Not an S-transform"
      else
        -- Puiseux case. Evaluating at z^2 is the same
        -- as taking the Laurent representative
        S2 := coef
        -- multiply with z^2
        chi2 : Stream R := cons(0, stream S2)
        -- multiply with 1/(1+z^2)
        z1z : Stream R := repeating([1, 0, -1, 0])$(Stream R)
        chi2 := (chi2 * z1z)$STSOR
        psi2 : Stream R := revert(chi2)$STSOR
        psi := powern(2::Fraction Integer, psi2)
        distributionByMoments sequence rest psi

  distributionBySTransform(S : STRREC) : % ==
      distributionBySTransform(S.puiseux, S.laurent, S.coef)

  freeMultiplicativeConvolution(x : %, y : %) : % ==
      import from Sequence R
      Sx : STRREC := moment2Stransform(moments(x))$MPR
      Sy : STRREC := moment2Stransform(moments(y))$MPR
      Sx.puiseux = 1/2 and Sy.puiseux = 1/2 => 0$%
      Sxc := stream Sx.coef
      Syc := stream Sy.coef
      Sxyc : Stream R
      if Sx.puiseux = 1 and Sx.puiseux = 1 then
        Sxyc := Sxc *$STSOR Syc
        return distributionBySTransform (1, 0, sequence Sxyc)
      else if Sx.puiseux = 1/2 then
      -- two cases left
        Sxyc := Sxc *$STSOR invmultisect(2, 0, Syc)
      else -- Sy.puiseux = 1/2 then
        Sxyc := invmultisect(2, 0, Sxc) *$STSOR Syc
      distributionBySTransform (1/2, -1, sequence Sxyc)

-- output
coerce(x : %) : OutputForm ==
    coerce((rep x) moments)$(Sequence R)

moment(x : %, n : NNI) : R ==
    zero? n => 1@R
    mm : Sequence R := (rep x) moments
    elt(mm, n)

classicalCumulant(x : %, n : PI) : R ==
    cc : Sequence R := (rep x) ccumulants
    elt(cc, n)

freeCumulant(x : %, n : PI) : R ==
    fc : Sequence R := (rep x) fcumulants
    elt(fc, n)

booleanCumulant(x : %, n : PI) : R ==
    fc : Sequence R := (rep x) bcumulants
```

```
    elt(fc, n)

moments(x : %) : Sequence R == (rep x) moments

classicalCumulants(x : %) : Sequence R == (rep x) ccumulants

freeCumulants(x : %) : Sequence R == (rep x) fcumulants

booleanCumulants(x : %) : Sequence R == (rep x) bcumulants

hankelDeterminants1(x : %, n : NonNegativeInteger) : Stream R == delay
    cons(hankelDeterminant(moments x, n)$(MomentPackage R),
         hankelDeterminants1(x, n+1))

hankelDeterminants(x : %) : Stream R == hankelDeterminants1(x, 1)

if R has Field then
    jacobiParameters(x : %) : Record(an : Stream R, bn : Stream R) ==
        moment2jacobi(moments x)$(MomentPackage R)

    orthogonalPolynomials(x : %) : Stream SUPR ==
        jac : Record(an : Stream R, bn : Stream R) := jacobiParameters x
        jacobi2poly(jac.an, jac.bn)$(MomentPackage R)

  else if R has IntegralDomain then
    jacobiParameters(x : %) : Record(an : Stream Fraction R,
                                     bn : Stream Fraction R) ==
        mm1 : Stream R := stream moments x
        mm : Stream Fraction R :=
            map( (t : R) : Fraction R +-> (coerce(t)@(Fraction R)),
                 mm1)$StreamFunctions2(R, Fraction R)
        moment2jacobi(sequence mm)$(MomentPackage Fraction R)

    orthogonalPolynomials(x : %) : Stream SUPFR ==
        jac : Record(an : Stream Fraction R, bn : Stream Fraction R) :=
            jacobiParameters x
        jacobi2poly(jac.an, jac.bn)$(MomentPackage Fraction R)

classicalConvolution(x : %, y : %) : % ==
    distributionByClassicalCumulants(classicalCumulants x +
                                     classicalCumulants y)

freeConvolution(x : %, y : %) : % ==
    distributionByFreeCumulants(freeCumulants x + freeCumulants y)

booleanConvolution(x : %, y : %) : % ==
    distributionByBooleanCumulants(booleanCumulants x +
                                   booleanCumulants y)
```

9.1. Monotone Convolution. Monotone convolution is done by composing the reciprocal Cauchy transforms: Let $H_\mu(z) = \frac{1}{G_\mu(z)}$, then $H_{\mu\rhd\nu}(z) = H_\mu(H_\nu(z))$. In other terms, let $F(z) = z\sum_{k=0}^{\infty} m_k z^k$, then $F_{\mu\rhd\nu}(z) = F_\mu(F_\nu(z))$. So we have to add the constant term 1 and multiply by z first and invert this operation at the end to obtain the moment sequence.

26a ⟨*domain DISTRO Distribution* 21⟩+≡ (32) ◁21 26b ▷

```
monotoneConvolution(x : %, y : %) : % ==
    distributionByMoments(
        sequence rest rest compose(cons(0, cons(1, stream moments x)),
            cons(0, cons(1, stream moments y)))$STSOR ) --$

x : %
n : PositiveInteger

nth : (Stream R, PositiveInteger) -> Stream R

nth(s : Stream R, n : PositiveInteger) : Stream R == delay
    res := s
    for k in 2..n repeat res := rest res
    cons(first res, nth (rest res, n))

_^(x, n) ==
    momn : Stream R := nth (stream  moments x, n)
    distributionByMoments sequence momn
```

9.2. Orthogonal Convolution. [3, Theorem 6.2] is defined by the relation

$$F_{\mu\vdash\nu}(z) = F_\mu(F_\nu(z)) - F_\nu(z) + z$$

From this it follows that

$$\tilde{B}_{\mu\vdash\nu}(z) = \tilde{B}_\mu(zM_\nu(z)) \tag{2}$$

where

$$\tilde{B}(z) = \frac{1}{z}B(z) = \sum_{n\geqslant 1} b_n z^{n-1}$$

26b ⟨*domain DISTRO Distribution* 21⟩+≡ (32) ◁26a 27a ▷

```
orthogonalConvolution(x, y) ==
    Bx:Stream R := stream booleanCumulants x
    zMy:Stream R :=  cons(0,cons(1, stream moments y))
    Bxy := compose(Bx, zMy)$STSOR
    distributionByBooleanCumulants Bxy
```

The convolution identity [4, p.2, (6)]

$$\mu \rhd \nu = \nu \uplus (\mu \vdash \nu) \tag{3}$$

26c ⟨*test.input* 26c⟩≡ 27b ▷

```
A:=distributionByMoments([a[k] for k in 1..])
B:=distributionByMoments([b[k] for k in 1..])
AmB := monotoneConvolution(A,B);
AtestB:= booleanConvolution(B, orthogonalConvolution(A,B));
moments AmB - moments AtestB
```

9.3. Subordination Convolution. Subordination convolution is defined implicitly by the relation [3, (1.6)]

$$\mu \boxplus \nu = \nu \rhd (\mu \boxplus \nu)$$

that is, is defined by the relation

$$F_{\mu \boxplus \nu}(z) = F_{\nu}(F_{\mu \boxplus \nu}(z))$$

Liu's formula [4, Lemma 7.7]:

$$R_{\mu \boxplus \nu}(z) = R_{\mu}(zM_{\nu}(z)) \tag{4}$$

27a ⟨*domain DISTRO Distribution* 21⟩+≡ (32) ◁ 26b

```
subordinationConvolution(x, y) ==
    Rx:Stream R := stream freeCumulants x
    zMy:Stream R :=  cons(0,cons(1, stream moments y))
    Rxy := compose(Rx, zMy)$STSOR
    distributionByFreeCumulants Rxy
```

The convolution identity

$$\mu \boxplus \nu = \nu \rhd (\mu \boxplus \nu)$$

27b ⟨*test.input* 26c⟩+≡ ◁ 26c

```
A:=distributionByFreeCumulants([a[k] for k in 1..])
B:=distributionByFreeCumulants([b[k] for k in 1..])
freeCumulants monotoneConvolution(B, subordinationConvolution(A,B))
```

10. package DISTPOL DistributionPolynomialPackage

28 ⟨*package DISTPOL DistributionPolynomialPackage* 28⟩≡ (32)

```
)abbrev package DISTPOL DistributionPolynomialPackage
++ Author: Franz Lehner lehner@math.tugraz.at, Waldek Hebisch
++ Date Created: 2010
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   A package to apply polynomial transformations to distributions
++   and integrate polynomials with respect to distributions.
DistributionPolynomialPackage(R : CommutativeRing,
                              S : Join(CommutativeRing, Module R),
                              UPS : UnivariatePolynomialCategory S
                             ) : Exports == Implementation   where
  DR ==> Distribution R
  DS ==> Distribution S
  FR==> Fraction R

  Exports ==> with
    eval : (DR, UPS) -> S
      ++ \spad{eval(d, p)} evaluates the distribution \spad{d}
      ++ as a linear functional on the polynomial \spad{p}.
      ++ Same as \spad{integrate(p, d)}.

    integrate : (UPS, DR) -> S
      ++ \spad{integrate(p, d)} integrates the polynomial \spad{p}
      ++ against the distribution \spad{d}. Same as \spad{eval(d, p)}.

    apply : (UPS, DR) -> DS
      ++ \spad{apply(p, d)} computes the distribution
      ++ of the random variable \spad{p(X)}$ where
      ++ \spad{X} has distribution \spad{d}.

  Implementation ==> add
    eval(x : DR, p : UPS) : S ==
        -- only 0th moment
        ground? p => (leadingCoefficient p)*(1$S)
        res : S := 0
        while not zero? p repeat
            res := res + moment(x, degree p)*leadingCoefficient p
            p := reductum p
        res

    integrate(p : UPS, x : DR) : S == eval(x, p)

    apply(p : UPS, x : DR) : DS ==
        -- evaluate x at powers of p
        IN : Stream Integer := expand([1..])$(UniversalSegment Integer)
        mompx : Stream S :=
             map( (k : Integer) : S +-> eval(x, p^(k::PositiveInteger)),
                  IN)$StreamFunctions2(Integer, S)
        distributionByMoments sequence mompx
```

11. package DSTCFPG DistributionContinuedFractionPackage

It is not possible to work with `UnivariatePolynomialCategory` because for the continued fraction we need access to `UnivariatePolynomial(Fraction R)`.

29 ⟨*package DSTCFPG DistributionContinuedFractionPackage* 29⟩≡ (32)

```
)abbrev package DSTCFPG DistributionContinuedFractionPackage
++ Author: Franz Lehner lehner@math.tugraz.at, Waldek Hebisch
++ Date Created: 2010
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   A package to compute Jacobi continued fractions of Cauchy transforms.
DistributionContinuedFractionPackage(R : CommutativeRing,
                                     z : Symbol
                                    ) : Exports == Implementation   where
  DR ==> Distribution R
  UPz ==> UnivariatePolynomial(z, R)
  FR==> Fraction R
  UPFz ==> UnivariatePolynomial(z, FR)

  Exports ==> with
    if R has Field then
        JContinuedFraction : (DR, UPz) -> ContinuedFraction UPz
          ++ \spad{JContinuedFraction(d, z)} returns the Cauchy
          ++ transform as a continued fraction at \spad{z}.

      else if R has IntegralDomain then
        JContinuedFraction : (DR, UPFz) -> ContinuedFraction UPFz
          ++ \spad{JContinuedFraction(d, z)} returns the Cauchy
          ++ transform as a continued fraction at \spad{z}.

  Implementation ==> add
    if R has Field then
        JContinuedFraction(x : DR, zz : UPz) : ContinuedFraction UPz ==
            jac : Record(an : Stream R, bn : Stream R) :=
                jacobiParameters(x)$DR
            import from StreamFunctions2(R, UPz)
            an1 : Stream UPz :=
                map((a : R) : UPz +-> zz-monomial(a, 0), jac an)
            bn1 : Stream UPz := map((b : R) : UPz +-> monomial(-b, 0), jac bn)
            continuedFraction(0, cons(1, bn1), cons(zz, an1))

      else if R has IntegralDomain then
        JContinuedFraction(x : DR, zz : UPFz) : ContinuedFraction UPFz ==
            jac : Record(an : Stream FR, bn : Stream FR) :=
                jacobiParameters(x)$DR
            import from StreamFunctions2(FR, UPFz)
            an1 : Stream UPFz :=
                map((a : FR) : UPFz +-> zz-monomial(a, 0), jac an)
            bn1 : Stream UPFz :=
                map((b : FR) : UPFz +-> monomial(-b, 0), jac bn)
            continuedFraction(0, cons(1, bn1), cons(zz, an1))
```

12. package DISTRO2 DistributionFunctions2

30 ⟨*package DISTRO2 DistributionFunctions2* 30⟩≡ (32)

```
)abbrev package DISTRO2 DistributionFunctions2
++ Author: Franz Lehner lehner@math.tugraz.at, Waldek Hebisch
++ Date Created: 16.09.2011
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   A package to map distributions between different rings.
DistributionFunctions2(R1 : CommutativeRing, R2 : CommutativeRing
                       ) : Exports == Implementation where
  DR1 ==> Distribution R1
  DR2 ==> Distribution R2
  SR1 ==> Sequence R1
  SR2 ==> Sequence R2
  SFR1R2 ==> SequenceFunctions2(R1, R2)
  Exports ==> with
    mapall : (R1 -> R2, Distribution R1) -> Distribution R2
      ++ \spad{map(f, x)} maps the moments and cumulants
      ++ from the ring R1 to R2.
    coerce : Distribution R1 -> Distribution R2
      ++ \spad{coerce(x)} coerces a distribution
      ++ from the ring R1 to R2.
    R1_to_R2_coercion : R1 -> R2
      ++ R1_to_R2_coercion should be local but conditional

  Implementation ==> add
    mapall(f : R1 -> R2, x : DR1) : DR2 ==
        mom2 : SR2 := map(f, moments x)$SFR1R2
        cum2 : SR2 := map(f, classicalCumulants(x))$SFR1R2
        fcum2 : SR2 := map(f, freeCumulants x)$SFR1R2
        bcum2 : SR2 := map(f, booleanCumulants x)$SFR1R2
        construct(mom2, cum2, fcum2, bcum2)$DR2

    if R2 has coerce : R1 -> R2 then
        R1_to_R2_coercion(x : R1) : R2 == coerce(x)$R2
    else if R1 has coerce : R1 -> R2 then
        R1_to_R2_coercion(x : R1) : R2 == coerce(x)$R1
    else
        R1_to_R2_coercion(x : R1) : R2 == error "unimplemented"

    coerce(x : Distribution R1) : Distribution R2 ==
        mapall(R1_to_R2_coercion, x)
```

13. package DISTEX DistributionPackage

31 ⟨*package DISTEX DistributionPackage* 31⟩≡ (32)

```
)abbrev package DISTEX DistributionPackage
++ Author: Franz Lehner lehner@math.tugraz.at, Waldek Hebisch
++ Date Created: 2009
++ Basic Functions:
++ Related Constructors:
++ Also See:
++ AMS Classifications:
++ Keywords:
++ References:
++ Description:
++   Various popular distributions.
DistributionPackage(R : CommutativeRing) : Exports == Implementation where
  DR ==> Distribution R

  Exports ==> with
    gaussianDistribution : R -> DR
      ++ \spad{gaussianDistribution(a)} produces
      ++ a gaussian distribution of variance \spad{a}.
    poissonDistribution : R -> DR
      ++ \spad{poissonDistribution(a)} produces
      ++ a Poisson distribution of variance \spad{a}.
    wignerDistribution : R -> DR
      ++ \spad{wignerDistribution(a)} produces
      ++ a Wigner distribution of variance \spad{a}.
    freePoissonDistribution : R -> DR
      ++ \spad{freePoissonDistribution(a)} produces
      ++ a free Poisson distribution of variance \spad{a}.
    bernoulliDistribution01 : R -> DR
      ++ \spad{bernoulliDistribution01(a)} produces
      ++ the distribution of a projection of trace a

    if R has Algebra Fraction Integer then
      arcsineDistribution : R -> DR
        ++ \spad{arcsineDistribution(a)} produces
        ++ the arcsine distribution of variance \spad{a}.

  Implementation ==> add
    gaussianDistribution(a : R) : DR ==
        cumu : Stream R := concat(construct [0, a], repeating [0])
        distributionByClassicalCumulants sequence cumu

    poissonDistribution(a : R) : DR ==
        cumu : Stream R := repeating [a]
        distributionByClassicalCumulants sequence cumu

    wignerDistribution(a : R) : DR ==
        cumu : Stream R := concat(construct [0, a], repeating [0])
        distributionByFreeCumulants sequence cumu

    freePoissonDistribution(a : R) : DR ==
        cumu : Stream R := repeating [a]
        distributionByFreeCumulants sequence cumu

    bernoulliDistribution01(a : R) : DR ==
        mom : Stream R := repeating [a]
        distributionByMoments sequence mom

    if R has Algebra Fraction Integer then
```

```
STSOR ==> StreamTaylorSeriesOperations R

arcsineDistribution(a : R) : DR ==
    -- mgf is 1/sqrt(1-2x^2)
    mgf : Stream R := cons(-2::R*a, repeating [0])
    mgf := cons(0, mgf)
    half : Fraction Integer := 1/ (2::Integer)
    mgf := powern(half, cons(1, mgf))$STSOR
    mgf := (recip(mgf)$STSOR)::Stream R
    distributionByMoments sequence rest mgf
```

32 ⟨* 32⟩≡

```
-- noweb source available at https://www.math.tugraz.at/~lehner/fricas/distro.spad.nw
```

⟨*category SEQUCAT SequenceCategory* 1⟩
⟨*domain SEQU Sequence* 3⟩
⟨*package SEQU2 SequenceFunctions2* 4⟩
⟨*package HANKP HankelPackage* 5⟩
⟨*package PARRPKG PathArrayPackage* 6⟩
⟨*package MOMPKG MomentPackage* 8⟩
⟨*package STRANS STransformPackage* 16⟩
⟨*category DISTCAT DistributionCategory* 19⟩
⟨*domain DISTRO Distribution* 21⟩
⟨*package DISTPOL DistributionPolynomialPackage* 28⟩
⟨*package DSTCFPG DistributionContinuedFractionPackage* 29⟩
⟨*package DISTRO2 DistributionFunctions2* 30⟩
⟨*package DISTEX DistributionPackage* 31⟩

References

[1] P. Flajolet. Combinatorial aspects of continued fractions. *Discrete Math.*, 32(2):125–161, 1980.
[2] Takahiro Hasebe and Hayato Saigo. The monotone cumulants. *Ann. Inst. Henri Poincaré Probab. Stat.*, 47(4):1160–1170, 2011.
[3] Romuald Lenczewski. Decompositions of the free additive convolution. *J. Funct. Anal.*, 246(2):330–365, 2007.
[4] Weihua Liu. Relations between convolutions and transforms in operator-valued free probability. arXiv:1809.05789, 2018.
[5] N. Raj Rao and Roland Speicher. Multiplication of free random variables and the S-transform: the case of vanishing mean. *Electron. Comm. Probab.*, 12:248–258, 2007.