

rec.spad

Martin Rubey

August 3, 2008

Abstract

The package defined in this file provide an operator for the n^{th} term of a recurrence and an operator for the coefficient of x^n in a function specified by a functional equation.

Contents

1	package RECOP RecurrenceOperator	1
1.1	Defining new operators	3
1.2	Recurrences	5
1.2.1	Extracting some information from the recurrence	5
1.2.2	Evaluating a recurrence	7
1.2.3	Setting the evaluation property of <code>oprecur</code>	8
1.2.4	Displaying a recurrence relation	9
1.3	Functional Equations	9
1.3.1	Extracting some information from the functional equation	9
1.3.2	Extracting a coefficient given a functional equation	10
1.3.3	Displaying a functional equation	12
2	License	12

1 package RECOP RecurrenceOperator

```
<package RECOP RecurrenceOperator>≡  
)abbrev package RECOP RecurrenceOperator  
++ Author: Martin Rubey  
++ Description:  
++ This package provides an operator for the n-th term of a recurrence and an  
++ operator for the coefficient of  $x^n$  in a function specified by a functional  
++ equation.  
RecurrenceOperator(R, F): Exports == Implementation where  
  R: Join(OrderedSet, IntegralDomain, ConvertibleTo InputForm)  
  F: Join(FunctionSpace R, AbelianMonoid, RetractableTo Integer, _  
         RetractableTo Symbol, PartialDifferentialRing Symbol)
```

```

--RecurrenceOperator(F): Exports == Implementation where
-- F: Join(ExpressionSpace, AbelianMonoid, RetractableTo Integer,
--      RetractableTo Symbol, PartialDifferentialRing Symbol)

Exports == with

evalRec: (BasicOperator, Symbol, F, F, F, List F) -> F
  ++ \spad{evalRec(u, dummy, n, n0, eq, values)} creates an expression that
  ++ stands for  $u(n_0)$ , where  $u(n)$  is given by the equation eq. However, for
  ++ technical reasons the variable n has to be replaced by a dummy
  ++ variable dummy in eq. The argument values specifies the initial values
  ++ of the recurrence  $u(0)$ ,  $u(1)$ ,...
  ++ For the moment we don't allow recursions that contain u inside of
  ++ another operator.

evalADE: (BasicOperator, Symbol, F, F, F, List F) -> F
  ++ \spad{evalADE(f, dummy, x, n, eq, values)} creates an expression that
  ++ stands for the coefficient of  $x^n$  in  $f(x)$ , where  $f(x)$  is given by the
  ++ functional equation eq. However, for technical reasons the variable x
  ++ has to be replaced by a dummy variable dummy in eq. The argument
  ++ values specifies  $f(0)$ ,  $f'(0)$ ,  $f''(0)$ ,...

getADE: F -> F
  ++ \spad{getADE f} returns the defining equation, if f stands for the
  ++ coefficient of an ADE.

-- should be local
  numberOfValuesNeeded: (Integer, BasicOperator, Symbol, F) -> Integer

-- should be local
  if R has Ring then
    getShiftRec: (BasicOperator, Kernel F, Symbol) -> Union(Integer, "failed")

    shiftInfoRec: (BasicOperator, Symbol, F) ->
      Record(max: Union(Integer, "failed"),
            ord: Union(Integer, "failed"),
            ker: Kernel F)

Implementation == add
<implementation: RecurrenceOperator>

```

1.1 Defining new operators

We define two new operators, one for recurrences, the other for functional equations. The operators for recurrences represents the n^{th} term of the corresponding sequence, the other the coefficient of x^n in the Taylor series expansion.

```
<implementation: RecurrenceOperator>≡
  oprecur := operator("rootOfRec"::Symbol)$BasicOperator

  opADE := operator("rootOfADE"::Symbol)$BasicOperator

  setProperty(oprecur, "%dummyVar", 2 pretend None)
  setProperty(opADE, "%dummyVar", 2 pretend None)
```

Setting these properties implies that the second and third arguments of `oprecur` are dummy variables and affects `tower$ES`: the second argument will not appear in `tower$ES`, if it does not appear in any argument but the first and second. The third argument will not appear in `tower$ES`, if it does not appear in any other argument. (`%defsum` is a good example)

The arguments of the two operators are as follows:

1. `eq`, i.e. the vanishing expression

```
<implementation: RecurrenceOperator>+≡
  eqAsF: List F -> F
  eqAsF 1 == 1.1
```

2. `dummy`, a dummy variable to make substitutions possible

```
<implementation: RecurrenceOperator>+≡
  dummy: List F -> Symbol
  dummy 1 == retract(1.2)@Symbol

  dummyAsF: List F -> F
  dummyAsF 1 == 1.2
```

3. the variable for display

```
<implementation: RecurrenceOperator>+≡
  displayVariable: List F -> F
  displayVariable 1 == 1.3
```

4. `operatorName(argument)`

```
<implementation: RecurrenceOperator>+≡  
  operatorName: List F -> BasicOperator  
  operatorName l == operator(kernels(1.4).1)  
  
  operatorNameAsF: List F -> F  
  operatorNameAsF l == 1.4  
  
  operatorArgument: List F -> F  
  operatorArgument l == argument(kernels(1.4).1).1
```

Concerning `rootOfADE`, note that although we have `arg` as argument of the operator, it is intended to indicate the coefficient, not the argument of the power series.

5. `values` in reversed order.

- `rootOfRec`: maybe `values` should be preceded by the index of the first given value. Currently, the last value is interpreted as $f(0)$.
- `rootOfADE`: values are the first few coefficients of the power series expansion in order.

```
<implementation: RecurrenceOperator>+≡  
  initialValues: List F -> List F  
  initialValues l == rest(1, 4)
```

1.2 Recurrences

1.2.1 Extracting some information from the recurrence

We need to find out whether we can determine the next term of the sequence, and how many initial values are necessary.

```
<implementation: RecurrenceOperator>+≡
  if R has Ring then
    getShiftRec(op: BasicOperator, f: Kernel F, n: Symbol)
      : Union(Integer, "failed") ==
      a := argument f
      if every?(freeOf?(#1, n::F), a) then return 0

      if #a ~ 1 then error "RECOP: operator should have only one argument"

      p := univariate(a.1, retract(n::F)@Kernel(F))
      if denominator p ~ 1 then return "failed"

      num := numer p

      if degree num = 1 and coefficient(num, 1) = 1
        and every?(freeOf?(#1, n::F), coefficients num)
        then return retractIfCan(coefficient(num, 0))
        else return "failed"

-- if the recurrence is of the form
-- $p(n, f(n+m-o), f(n+m-o+1), \dots, f(n+m)) = 0$
-- in which case shiftInfoRec returns [m, o, f(n+m)].

    shiftInfoRec(op: BasicOperator, argsym: Symbol, eq: F):
      Record(max: Union(Integer, "failed"),
            ord: Union(Integer, "failed"),
            ker: Kernel F) ==

-- ord and ker are valid only if all shifts are Integers
-- ker is the kernel of the maximal shift.
      maxShift: Integer
      minShift: Integer
      nextKernel: Kernel F

-- We consider only those kernels that have op as operator. If there is none,
-- we raise an error. For the moment we don't allow recursions that contain op
-- inside of another operator.

      error? := true
```

```

for f in kernels eq repeat
  if is?(f, op) then
    shift := getShiftRec(op, f, argsym)
    if error? then
      error? := false
      nextKernel := f
      if shift case Integer then
        maxShift := shift
        minShift := shift
      else return ["failed", "failed", nextKernel]
    else
      if shift case Integer then
        if maxShift < shift then
          maxShift := shift
          nextKernel := f
        if minShift > shift then
          minShift := shift
        else return ["failed", "failed", nextKernel]

if error? then error "evalRec: equation does not contain operator"

[maxShift, maxShift - minShift, nextKernel]

```

1.2.2 Evaluating a recurrence

```
<implementation: RecurrenceOperator>+≡
evalRec(op, argsym, argdisp, arg, eq, values) ==
  if ((n := retractIfCan(arg)@Union(Integer, "failed")) case "failed")
    or (n < 0)
  then
    shiftInfo := shiftInfoRec(op, argsym, eq)

    if (shiftInfo.ord case "failed") or ((shiftInfo.ord)::Integer > 0)
    then
      kernel(oprecur,
        append([eq, argsym::F, argdisp, op(arg)], values))
    else
      p := univariate(eq, shiftInfo.ker)
      num := numer p

-- If the degree is 1, we can return the function explicitly.

    if degree num = 1 then
      eval(-coefficient(num, 0)/coefficient(num, 1), argsym::F,
        arg::F-(shiftInfo.max)::Integer::F)
    else
      kernel(oprecur,
        append([eq, argsym::F, argdisp, op(arg)], values))
  else
    len: Integer := #values
    if n < len
    then values.(len-n)
    else
      shiftInfo := shiftInfoRec(op, argsym, eq)

      if shiftInfo.max case Integer then
        p := univariate(eq, shiftInfo.ker)

        num := numer p

      if degree num = 1 then

        next := -coefficient(num, 0)/coefficient(num, 1)
        nextval := eval(next, argsym::F,
          (len-(shiftInfo.max)::Integer)::F)
        newval := eval(nextval, op,
          evalRec(op, argsym, argdisp, #1, eq, values))
        evalRec(op, argsym, argdisp, arg, eq, cons(newval, values))
      else
```

```

        kernel(oprecur,
              append([eq, argsym::F, argdisp, op(arg)], values))
    else
        kernel(oprecur,
              append([eq, argsym::F, argdisp, op(arg)], values))

numberOfValuesNeeded(numberOfValues: Integer,
                    op: BasicOperator, argsym: Symbol, eq: F): Integer ==
    order := shiftInfoRec(op, argsym, eq).ord
    if order case Integer
    then min(numberOfValues, retract(order)@Integer)
    else numberOfValues

else
    evalRec(op, argsym, argdisp, arg, eq, values) ==
        kernel(oprecur,
              append([eq, argsym::F, argdisp, op(arg)], values))

numberOfValuesNeeded(numberOfValues: Integer,
                    op: BasicOperator, argsym: Symbol, eq: F): Integer ==
    numberOfValues

```

1.2.3 Setting the evaluation property of oprecur

irecur is just a wrapper that allows us to write a recurrence relation as an operator.

```

<implementation: RecurrenceOperator>+≡
    irecur: List F -> F
    irecur l ==
        evalRec(operatorName l,
                dummy l, displayVariable l,
                operatorArgument l, eqAsF l, initialValues l)

evaluate(oprecur, irecur)$BasicOperatorFunctions1(F)

```

1.2.4 Displaying a recurrence relation

```
<implementation: RecurrenceOperator>+≡
ddrec: List F -> OutputForm
ddrec l ==
  op := operatorName l
  values := reverse l
  eq := eqAsF l

  numberOfValues := numberOfValuesNeeded(#values-4, op, dummy l, eq)

  vals: List OutputForm
    := cons(eval(eq, dummyAsF l, displayVariable l)::OutputForm = _
            0::OutputForm,
            [elt(op::OutputForm, [(i-1)::OutputForm]) = _
             (values.i)::OutputForm
             for i in 1..numberOfValues])

  bracket(hconcat([(operatorNameAsF l)::OutputForm,
                  ": ",
                  commaSeparate vals]))

setProperty(oprecur, "%specialDisp",
            ddrec@(List F -> OutputForm) pretend None)
```

1.3 Functional Equations

1.3.1 Extracting some information from the functional equation

getOrder returns the maximum derivative of op occurring in f.

```
<implementation: RecurrenceOperator>+≡
getOrder(op: BasicOperator, f: Kernel F): NonNegativeInteger ==
  res: NonNegativeInteger := 0
  g := f
  while is?(g, %diff) repeat
    g := kernels(argument(g).1).1
    res := res+1

  if is?(g, op) then res else 0
```

1.3.2 Extracting a coefficient given a functional equation

```

<implementation: RecurrenceOperator>+≡
  evalADE(op, argsym, argdisp, arg, eq, values) ==
    if not freeOf?(eq, retract(argdisp)@Symbol)
      then error "RECOP: The argument should not be used in the equation of the_
ADE"

    if ((n := retractIfCan(arg)@Union(Integer, "failed")) case "failed")
      then
--
-- The following would need yet another operator, namely "coefficient of".
--
--      keq := kernels eq
--      order := getOrder(op, keq.1)
--      for k in rest keq repeat order := max(order, getOrder(op, k))
--
--      if zero? order then
-- -- in this case, we have an algebraic equation
--
--          p: Fraction SparseUnivariatePolynomial F
--          := univariate(eq, kernels(D(op(argsym::F), argsym, order)).1)$F
--
--          num := numer p
--
--          if one? degree num then
-- -- the equation is in fact linear
--              return eval(-coefficient(num, 0)/coefficient(num, 1), argsym::F, arg::F)
--
--          kernel(opADE,
--              append([eq, argsym::F, argdisp, op(arg)], values))
else
  if n < 0
  then 0
  else
    keq := kernels eq
    order := getOrder(op, keq.1)
    output(hconcat("The order is ", order::OutputForm))$OutputPackage
    for k in rest keq repeat order := max(order, getOrder(op, k))

    p: Fraction SparseUnivariatePolynomial F
      := univariate(eq, kernels(D(op(argsym::F), argsym, order)).1)$F

    output(hconcat("p: ", p::OutputForm))$OutputPackage
    if degree numer p > 1
    then

```

```

        kernel(opADE,
              append([eq, argsym::F, argdisp, op(arg)], values))
else
  s := seriesSolve(eq, op, argsym, first(reverse values, order))
      $ExpressionSolve(R, F,
                      UnivariateFormalPowerSeries F,
                      UnivariateFormalPowerSeries
                      SparseUnivariatePolynomialExpressions F)

elt(s, n::Integer::NonNegativeInteger)

iADE: List F -> F
-- This is just a wrapper that allows us to write a recurrence relation as an
-- operator.
iADE l ==
  evalADE(operatorName l,
          dummy l, displayVariable l,
          operatorArgument l, eqAsF l, initialValues l)

evaluate(opADE, iADE)$BasicOperatorFunctions1(F)

getADE(f: F): F ==
  ker := kernels f
  if one?(#ker) and is?(operator(ker.1), "rootOfADE"::Symbol) then
    l := argument(ker.1)
    eval(eqAsF l, dummyAsF l, displayVariable l)
  else
    error "getADE: argument should be a single rootOfADE object"

```

1.3.3 Displaying a functional equation

```
<implementation: RecurrenceOperator>+≡
  ddADE: List F -> OutputForm
  ddADE l ==
    op := operatorName l
    values := reverse l

  vals: List OutputForm
    := cons(eval(eqAsF l, dummyAsF l, displayVariable l)::OutputForm = _
            0::OutputForm,
            [eval(D(op(dummyAsF l), dummy l, i), _
                  dummyAsF l=0)::OutputForm = (values.(i+1))::OutputForm
              for i in 0..min(4,#values-5)])

  bracket(hconcat([bracket((displayVariable l)::OutputForm ** _
                           (operatorArgument l)::OutputForm),
                        (op(displayVariable l))::OutputForm, ": ",
                        commaSeparate vals])))

  setProperty(opADE, "%specialDisp",
              ddADE@(List F -> OutputForm) pretend None)
```

2 License

```
<license>≡
-- Copyright (C) 2006 Martin Rubey <Martin.Rubey@univie.ac.at>
--
-- This program is free software; you can redistribute it and/or
-- modify it under the terms of the GNU General Public License as
-- published by the Free Software Foundation; either version 2 of
-- the License, or (at your option) any later version.
--
-- This program is distributed in the hope that it will be
-- useful, but WITHOUT ANY WARRANTY; without even the implied
-- warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
-- PURPOSE. See the GNU General Public License for more details.
```

```
<*>≡
<license>
```

```
<package RECOP RecurrenceOperator>
```