16.39 LPDO: Linear Partial Differential Operators

Author: Thomas Sturm

16.39.1 Introduction

Consider the field $F = \mathbb{Q}(x_1, \ldots, x_n)$ of rational functions and a set $\Delta = \{\partial_{x_1}, \ldots, \partial_{x_n}\}$ of *commuting derivations* acting on F. That is, for all $\partial_{x_i}, \partial_{x_j} \in \Delta$ and all $f, g \in F$ the following properties are satisfied:

$$\begin{aligned} \partial_{x_i}(f+g) &= \partial_{x_i}(f) + \partial_{x_i}(g), \\ \partial_{x_i}(f \cdot g) &= f \cdot \partial_{x_i}(g) + \partial_{x_i}(f) \cdot g, && (16.87) \\ \partial_{x_i}(\partial_{x_j}(f)) &= \partial_{x_j}(\partial_{x_i}(f)). && (16.88) \end{aligned}$$

Consider now the set $F[\partial_{x_1}, \ldots, \partial_{x_n}]$, where the derivations are used as variables. This set forms a non-commutative *linear partial differential operator ring* with pointwise addition, and multiplication defined as follows: For $f \in F$ and ∂_{x_i}, $\partial_{x_j} \in \Delta$ we have for any $g \in F$ that

$$\begin{aligned} (f\partial_{x_i})(g) &= f \cdot \partial_{x_i}(g), \\ (\partial_{x_i} f)(g) &= \partial_{x_i}(f \cdot g), && (16.89) \\ (\partial_{x_i}\partial_{x_j})(g) &= \partial_{x_i}(\partial_{x_j}(g)). && (16.90) \end{aligned}$$

Here " $\cdot$ " denotes the multiplication in F. From (16.90) and (16.88) it follows that $\partial_{x_i}\partial_{x_j} = \partial_{x_j}\partial_{x_i}$, and using (16.89) and (16.87) the following *commutator* can be proved:

$$\partial_{x_i} f = f\partial_{x_i} + \partial_{x_i}(f).$$

A *linear partial differential operator* (LPDO) of order k is an element

$$D = \sum_{|j| \le k} a_j \partial^j \in F[\partial_{x_1}, \ldots, \partial_{x_n}]$$

in canonical form. Here the expression $|j| \le k$ specifies the set of all tuples of the form $j = (j_1, \ldots, j_n) \in \mathbb{N}^n$ with $\sum_{i=1}^n j_i \le k$, and we define $\partial^j = \partial_{x_1}^{j_1} \cdots \partial_{x_n}^{j_n}$.

A *factorization* of D is a non-trivial decomposition

$$D = D_1 \cdots D_r \in F[\partial_{x_1}, \ldots, \partial_{x_n}]$$

into multiplicative factors, each of which is an LPDO D_i of order greater than 0 and less than k. If such a factorization exists, then D is called *reducible* or *factorable*, else *irreducible*.

For the purpose of factorization it is helpful to temporarily consider as regular commutative polynomials certain summands of the LPDO under consideration. Consider a commutative polynomial ring over F in new indeterminates $y_1, \ldots, y_n$. Adopting the notational conventions above, for $m \leq k$ the *symbol of D of order m* is defined as

$$\mathrm{Sym}_m(D) \quad = \quad \sum_{|j|=m} a_j y^j \in F[y_1, \ldots, y_n].$$

For $m = k$ we obtain as a special case the *symbol* $\mathrm{Sym}(D)$ *of* D.

16.39.2 Operators

16.39.2.1 `partial`

There is a unary operator `partial(·)` denoting ∂.

⟨*partial-term*⟩ → **`partial (`** ⟨*id*⟩ **`)`**

16.39.2.2 `***`

There is a binary operator `***` for the non-commutative multiplication involving partials ∂_x. All expressions involving `***` are implicitly transformed into LPDOs, i.e., into the following normal form:

⟨*normalized-lpdo*⟩ → ⟨*normalized-mon*⟩ [**`+`** ⟨*normalized-lpdo*⟩]
⟨*normalized-mon*⟩ → ⟨*F-element*⟩ [**`***`** ⟨*partial-termprod*⟩]
⟨*partial-termprod*⟩ → ⟨*partial-term*⟩ [**`***`** ⟨*partial-termprod*⟩]

The summands of the *normalized-lpdo* are ordered in some canonical way. As an example consider

```
input: a()***partial(y)***b()***partial(x);

(a()*b()) *** partial(x) *** partial(y) + (a()*diff(b(),y,1)) *** partial(x)
```

Here the *F-elements* are polynomials, where the unknowns are of the type *constant-operator* denoting functions from F:

⟨*constant-operator*⟩ → ⟨*id*⟩ **`( )`**

We do not admit division of such constant operators since we cannot exclude that such a constant operator denotes 0.

The operator notation on the one hand emphasizes the fact that the denoted elements are functions. On the other hand it distinguishes `a()` from the variable `a` of a rational function, which specifically denotes the corresponding projection. Consider e.g.

```
input: (x+y)***partial(y)***(x-y)***partial(x);

  2    2
(x  - y ) *** partial(x) *** partial(y) + ( - x - y) *** partial(x)
```

Here we use as *F-elements* specific elements from $F = \mathbb{Q}(x, y)$.

16.39.2.3 `diff`

In our example with constant operators, the transformation into normal form introduces a formal derivative operation `diff(·,·,·)`, which cannot be evaluated. Notice that we do not use the Reduce operator `df(·,·,·)` here, which for technical reasons cannot smoothly handle our constant operators.

In our second example with rational functions as *F-elements*, derivative occurring with commutation can be computed such that `diff` does not occur in the output.

16.39.3 Shapes of F-elements

Besides the generic computations with constant operators, we provide a mechanism to globally fix a certain *shape* for *F-elements* and to expand constant operators according to that shape.

16.39.3.1 `lpdoset`

We give an example for a shape that fixes all constant operators to denote generic bivariate affine linear functions:

```
input: d := (a()+b())***partial(x1)***partial(x2)**2;

                                                    2
d := (a() + b()) *** partial(x1) *** partial(x2)

input: lpdoset {!#10*x1+!#01*x2+!#00,x1,x2};

{-1}

input: d;

2
```

```
(a00 + a01*x2 + a10*x1 + b00 + b01*x2 + b10*x1) *** partial(x1) *** partial(x2)
```

Notice that the placeholder # must be escaped with !, which is a general convention for Rlisp/Reduce. Notice that `lpdoset` returns the old shape and that `{-1}` denotes the default state that there is no shape selected.

16.39.3.2 lpdoweyl

The command `lpdoweyl {n,x1,x2,...}` creates a shape for generic polynomials of total degree `n` in variables `x1`, `x2`, ...

```
input: lpdoweyl(2,x1,x2);

                                2                                    2
{#_00_ + #_01_*x2 + #_02_*x2  + #_10_*x1 + #_11_*x1*x2 + #_20_*x1 ,x1,x2}

input: lpdoset ws;

{#10*x1 + #01*x2 + #00,x1,x2}

input: d;

                               2                                     2
(a_00_ + a_01_*x2 + a_02_*x2  + a_10_*x1 + a_11_*x1*x2 + a_20_*x1  + b_00_

                           2                                    2
  + b_01_*x2 + b_02_*x2  + b_10_*x1 + b_11_*x1*x2 + b_20_*x1 ) *** partial(x1)

                 2
 *** partial(x2)
```

16.39.4 Commands

16.39.4.1 General

lpdoord The *order* of an lpdo:

```
input: lpdoord((a()+b())***partial(x1)***partial(x2)**2+3***partial(x1));

3
```

lpdoptl Returns the list of derivations (partials) occurring in its argument LPDO *d*.

```
input: lpdoptl(a()***partial(x1)***partial(x2)+partial(x4)+diff(a(),x3,1));

{partial(x1),partial(x2),partial(x4)}
```

That is the smallest set $\{\ldots,\partial_{x_i},\ldots\}$ such that d is defined in $F[\ldots,\partial_{x_i},\ldots]$. Notice that formal derivatives are not derivations in that sense.

lpdogp Given a starting symbol a, a list of variables l, and a degree n, `lpdogp(a,l,n)` generates a generic (commutative) polynomial of degree n in variables l with coefficients generated from the starting symbol a:

```
input: lpdogp(a,{x1,x2},2);

                              2                                    2
a_00_ + a_01_*x2 + a_02_*x2  + a_10_*x1 + a_11_*x1*x2 + a_20_*x1
```

lpdogdp Given a starting symbol a, a list of variables l, and a degree n, `lpdogp(a,l,n)` generates a generic differential polynomial of degree n in variables l with coefficients generated from the starting symbol a:

```
input: lpdogdp(a,{x1,x2},2);

                       2                          2
a_20_ *** partial(x1)  + a_02_ *** partial(x2)

 + a_11_ *** partial(x1) *** partial(x2) + a_10_ *** partial(x1)

 + a_01_ *** partial(x2) + a_00_
```

16.39.4.2 Symbols

lpdosym The *symbol* of an lpdo. That is the differential monomial of highest order with the partials replaced by corresponding commutative variables:

```
input: lpdosym((a()+b())***partial(x1)***partial(x2)**2+3***partial(x1));

           2
y_x1_*y_x2_ *(a() + b())
```

More generally, one can use a second optional arguments to specify a the order of a different differential monomial to form the symbol of:

```
input: lpdosym((a()+b())***partial(x1)***partial(x2)**2+3***partial(x1),1);

3*y_x1_
```

Finally, a third optional argument can be used to specify an alternative starting symbol for the commutative variable, which is `y` by default. Altogether, the optional arguments default like `lpdosym(·)=lpdosym(·,lpdoord(·),y)`.

lpdosym2dp This converts a symbol obtained via `lpdosym` back into an LPDO resulting in the corresponding differential monomial of the original LPDO.

```
input: d := a()***partial(x1)***partial(x2)+partial(x3)$

input: s := lpdosym d;

s := a()*y_x1_*y_x2_

input: lpdosym2dp s;

a() *** partial(x1) *** partial(x2)
```

In analogy to `lpdosym` there is an optional argument for specifying an alternative starting symbol for the commutative variable, which is `y` by default.

lpdos Given LPDOs p, q and $m \in \mathbb{N}$ the function `lpdos(`p`,`q`,`m`)` computes the commutative polynomial

$$S_m = \sum_{\substack{|j|=m \\ |j|<k}} \left(\sum_{i=1}^{n} p_i \partial_i(q_j) + p_0 q_j \right) y^j.$$

This is useful for the factorization of LPDOs.

```
input: p := a()***partial(x1)+b()$

input: q := c()***partial(x1)+d()***partial(x2)$

input: lpdos(p,q,1);

a()*diff(c(),x1,1)*y_x1_ + a()*diff(d(),x1,1)*y_x2_ + b()*c()*y_x1_

 + b()*d()*y_x2_
```

16.39.4.3 Factorization

lpdofactorize Factorize the argument LPDO d. The ground field F must be fixed via `lpdoset`. The result is a list of lists $\{\ldots,(A_i,L_i),\ldots\}$. A_i is is genrally the identifiers `true`, which indicates reducibility. The respective L_i is a list of two differential polynomial factors, the first of which has order 1.

```
input: bk := (partial(x)+partial(y)+(a10-a01)/2) ***
       (partial(x)-partial(y)+(a10+a01)/2);

                 2             2
bk := partial(x)  - partial(y)  + a10 *** partial(x) + a01 *** partial(y)
```

```
          2      2
     - a01  + a10
 + -----------------
          4

input: lpdoset lpdoweyl(1,x,y);

{#_00_ + #_01_*y + #_10_*x,x,y}

input: lpdofactorize bk;

{{true,

                                    a01 - a10
  { - partial(x) - partial(y) + ------------,
                                       2

                                     - a01 - a10
    - partial(x) + partial(y) + ---------------}}}
                                       2
```

If the result is the empty list, then this guarantees that there is no approximate factorization possible. In general it is possible to obtain several sample factorizations. Note, however, that the result does not provide a complete list of possible factorizations with a left factor of order 1 but only at least one such sample factorization in case of reducibility.

Furthermore, the procedure might fail due to polynomial degrees exceeding certain bounds for the extended quantifier elimination by virtual substitution used internally. In this case there is the identifier `failed` returned. This must not be confused with the empty list indicating irreducibility as described above.

Besides

1. the LPDO d,

`lpdofactorizex` accepts several optional arguments:

2. An LPDO of order 1, which serves as a template for the left (linear) factor. The default is a generic linear LPDO with generic coefficient functions according from the ground field specified via `lpdoset`. The principle idea is to support the factorization by guessing that certain differential monomials are not present.

3. An LPDO of order $\mathrm{ord}(d)-1$, which serves as a template for the right factor. Similarly to the previous argument the default is fully generic.

lpdofac This is a low-level entry point to the factorization `lpdofactorize`. It accepts the same arguments as `lpdofactorize`. It generates factorization conditions as a quite large first-order formula over the reals. This can be passed to extended quantifier elimination. For example, consider `bk` as in the example for `lpdofactorize` above:

```
input: faccond := lpdofac bk$

input: rlqea faccond;

{{true,

               a01 - a10
  {p_00_00_ = -----------,
                   2

   p_00_01_ = 0, p_00_10_ = 0, p_01_00_ = -1, p_01_01_ = 0, p_01_10_ = 0,

   p_10_00_ = -1, p_10_01_ = 0, p_10_10_ = 0,

                - a01 - a10
   q_00_00_ = --------------,
                    2

   q_00_01_ = 0, q_00_10_ = 0, q_01_00_ = 1, q_01_01_ = 0, q_01_10_ = 0,

   q_10_00_ = -1, q_10_01_ = 0, q_10_10_ = 0}}}
```

The result of the extended quantifier elimination provides coefficient values for generic factor polynomials p and q. These are automatically interpreted and converted into differential polynomials by `lpdofactorize`.

16.39.4.4 Approximate Factorization

lpdofactorizex Approximately factorize the argument LPDO d. The ground field F must be fixed via `lpdoset`. The result is a list of lists $\{\ldots,(A_i,L_i),\ldots\}$. Each A_i is quantifier-free formula possibly containing a variable `epsilon`, which describes the precision of corresponding factorization L_i. L_i is a list containing two factors, the first of which is linear.

```
input: off lpdocoeffnorm$

input: lpdoset lpdoweyl(0,x1,x2)$

input: f2 := partial(x1)***partial(x2) + 1$

input: lpdofactorizex f2;
```

```
{{epsilon - 1 >= 0,{partial(x1),partial(x2)}},

 {epsilon - 1 >= 0,{partial(x2),partial(x1)}}}
```

If the result is the empty list, then this guarantees that there is no approximate factorization possible. In our example we happen to obtain two possible factorizations. Note, however, that the result in general does not provide a complete list of factorizations with a left factor of order 1 but only at least one such sample factorization.

Furthermore, the procedure might fail due to polynomial degrees exceeding certain bounds for the extended quantifier elimination by virtual substitution used internally. If this happens, the corresponding A_i will contain existential quantifiers `ex`, and L_i will be meaningless.
`Da sollte besser ein failed kommen ...`

The first of the two subresults above has the semantics that $\partial_{x_1}\partial_{x_2}$ is an approximate factorization of f_2 for all $\varepsilon \geq 1$. Formally, $||f_2 - \partial_{x_1}\partial_{x_2}|| \leq \varepsilon$ for all $\varepsilon \geq 1$, which is equivalent to $||f_2 - \partial_{x_1}\partial_{x_2}|| \leq 1$. That is, 1 is an upper bound for the approximation error over $\mathbb{R}^2$. Where there are two possible choices for the seminorm $||\cdot||$:

1. ...
2. ...

`explain switch lpdocoeffnorm ...`

Besides

1. the LPDO d,

`lpdofactorizex` accepts several optional arguments:

2. A Boolean combination ψ of equations, negated equations, and (possibly strict) ordering constraints. This ψ describes a (semialgebraic) region over which to factorize approximately. The default is `true` specifying the entire $\mathbb{R}^n$. It is possible to choose ψ parametrically. Then the parameters will in general occur in the conditions A_i in the result.

3., 4. An LPDO of order 1, which serves as a template for the left (linear) factor, and an LPDO of order $\mathrm{ord}(d) - 1$, which serves as a template for the right factor. See the documentation of `lpdofactorize` for defaults and details.

5. A bound ε for describing the desired precision for approximate factorization. The default is the symbol `epsilon`, i.e., a symbolic choice such that

the optimal choice (with respect to parameters in ψ) is obtained during factorization. It is possible to fix $\varepsilon \in \mathbb{Q}$. This does, however, not considerably simplify the factorization process in most cases.

```
input: f3 := partial(x1) *** partial(x2) + x1$

input: psi1 := 0<=x1<=1 and 0<=x2<=1$

input: lpdofactorizex(f3,psi1,a()***partial(x1),b()***partial(x2));

{{epsilon - 1 >= 0,{partial(x1),partial(x2)}}}
```

lpdofacx This is a low-level entry point to the factorization `lpdofactorizex`. It is analogous to `lpdofac` for `lpdofactorize`; see the documentation there for details.

lpdohrect

lpdohcirc