16.21 DUMMY: Canonical form of expressions with dummy variables

This package allows a user to find the canonical form of expressions involving dummy variables. In that way, the simplification of polynomial expressions can be fully done. The indeterminates are general operator objects endowed with as few properties as possible. In that way the package may be used in a large spectrum of applications.

Author: Alain Dresse.

16.21.1 Introduction

The possibility to handle dummy variables and to manipulate dummy summations are important features in many applications. In particular, in theoretical physics, the possibility to represent complicated expressions concisely and to realize simplifications efficiently depend on both capabilities. However, when dummy variables are used, there are many more ways to express a given mathematical objects since the names of dummy variables may be chosen almost arbitrarily. Therefore, from the point of view of computer algebra the simplification problem is much more difficult. Given a definite ordering, one is, at least, to find a representation which is independent of the names chosen for the dummy variables otherwise, simplifications are impossible. The package does handle any number of dummy variables and summations present in expressions which are arbitrary multivariate polynomials and which have operator objects eventually dependent on one (or several) dummy variable(s) as some of their indeterminates. These operators have the same generality as the one existing in REDUCE. They can be noncommutative, anticommutative or commutative. They can have any kind of symmetry property. Such polynomials will be called in the following *dummy* polynomials. Any monomial of this kind will be called *dummy* monomial. For any such object, the package allows to find a well defined *normal form* in one-to-one correspondance with it.

In section 2, the convention for writing dummy summations is explained and the available declarations to introduce or suppress dummy variables are given.

In section 3, the commands allowing to give various algebraic properties to the operators are described.

In section 4, the use of the function `CANONICAL` is explained and illustrated.

In section 5, a fairly complete set of references is given.

The use of DUMMY requires that the package `ASSIST` version 2.2 be available. This is the case when `REDUCE 3.6` is used. When loaded, ASSIST is automatically loaded.

16.21.2 Dummy variables and dummy summations

A dummy variable (let us name it dv) is an identifier which runs from the integer i_1 to another integer i_2. To the extent that no definite space is defined, i_1 and i_2 are assumed to be some integers which are the *same* for all dummy variables.

If f is any `REDUCE` operator, then the simplest dummy summation associated to dv is the sum

$$\sum_{dv=i_1}^{i_2} f(dv)$$

and is simply written as

$$f(dv).$$

No other rules govern the implicit summations. dv can appear as many times we want since the operator f may depend on an arbitrary number of variables. So, the package is potentially applicable to many contexts. For instance, it is possible to add rules of the kind one encounters in tensor calculus.

Obviously, there are as many ways we want to express the *same* quantity. If the name of another dummy variable is dum then the previous expression is written as

$$\sum_{dum=i_1}^{i_2} f(dum)$$

and the computer algebra system should be able to find that the expression

$$f(dv) - f(dum);$$

is equal to 0. A very special case which is *allowed* is when f is the identity operator. So, a generic dummy polynomial will be a sum of dummy monomials of the kind

$$\prod_i c_i * f_i(dv_1, \ldots, dv_{k_i}, fr_1, \ldots, fr_{l_i})$$

where $dv_1, \ldots,$ are dummy variables while $fr_1, \ldots,$ are ordinary or free variables.

To declare dummy variables, two commands are available:

- i.

  ```
  dummy_base <idp>;
  ```

  where `idp` is the name of any unassigned identifier.

- ii.

  ```
  dummy_names <d>,<dp>,<dpp> ....;
  ```

The first one declares $idp_1, \ldots, idp_n$ as dummy variables i.e. all variables of the form idp_{xxx} where xxx is a number will be dummy variables, such as $idp_1, idp_2, \ldots, idp_{23}$. The second one gives special names for dummy variables. All other identifiers which may appear are assumed to be *free*. However, there is a restriction: named and base dummy variables cannot be declared *simultaneously*. The above declarations are mutually *exclusive*. Here is an example showing that:

```
dummy_base dv; ==> dv

          % dummy indices are dv1, dv2, dv3, ...

dummy_names i,j,k; ==>

          ***** The created dummy base dv must be cleared
```

When this is done, an expression like

```
op(dv1)*sin(dv2)*abs(i)*op(dv2)$
```

means a sum over dv_1, dv_2. To clear the dummy base, and to create the dummy names i, j, k one is to do

```
clear_dummy_base; ==> t

dummy_names i,j,k; ==> t

           % dummy indices are i,j,k.
```

When this is done, an expression like

```
op(dv1)*sin(dv2)*abs(x)*op(i)^3*op(dv2)$
```

means a sum over i. One should keep in mind that every application of the above commands erases the previous ones. It is also possible to display the declared dummy names using `SHOW_DUMMY_NAMES`:

```
show_dummy_names(); ==> {i,j,k}
```

To suppress *all* dummy variables one can enter

```
clear_dummy_names; clear_dummy_base;
```

16.21.3 The Operators and their Properties

All dummy variables *should appear at first level* as arguments of operators. For instance, if i and j are dummy variables, the expression

```
rr:=  op(i,j)-op(j,j)
```

is allowed but the expression

```
op(i,op(j)) - op(j,op(j))
```

is *not* allowed. This is because dummy variables are not detected if they appear at a level larger than 1. Apart from that there is no restrictions. Operators may be commutative, noncommutative or even anticommutative. Therefore they may be elements of an algebra, they may be tensors, spinors, grassman variables, etc. ... By default they are assumed to be *commutative* and without symmetry properties. The `REDUCE` command `NONCOM` is taken into account and, in addition, the command

```
anticom at1, at2;
```

makes the operators at_1 and at_2 anticommutative.

One can also give symmetry properties to them. The usual declarations `SYMMETRIC and ANTISYMMETRIC` are taken into account. Moreover and most important they can be endowed with a *partial* symmetry through the command `SYMTREE`. Here are three illustrative examples for the r operator:

```
symtree (r,{!+, 1, 2, 3, 4});
symtree (r,{!*, 1, {!-, 2, 3, 4}});
symtree (r, {!+, {!-, 1, 2}, {!-, 3, 4}});
```

The first one makes the operator (fully) symmetric. The second one declares it antisymmetric with respect to the three last indices. The symbols !*, !+ and !- at the beginning of each list mean that the operator has no symmetry, is symmetric or is antisymmetric with respect to the indices inside the list. Notice that the indices are not denoted by their names but merely by their natural order of appearance. 1 means the first written argument of r, 2 its second argument etc. The first command is equivalent to the declaration `symmetric` except that the number of indices of r is *restricted* to 4 i.e. to the number declared in `SYMTREE`. In the second example r is stated to have no symmetry with respect to the first index and is declared to be antisymmetric with respect to the three last indices. In the third example, r is made symmetric with respect to the interchange of the pairs of indices 1,2 and 3,4 respectively and is made antisymmetric separately within the pairs $(1, 2)$ and $(3, 4)$. It is the symmetry of the Riemann tensor. The anticommutation property and the

various symmetry properties may be suppressed by the commands REMANTICOM and REMSYM. To eliminate partial symmetry properties one can also use SYMTREE itself. For example, assuming that r has the Riemann symmetry, to eliminate it do

```
symtree (r,{!*, 1, 2, 3, 4});
```

However, notice that the number of indices remains fixed and equal to 4 while with REMSYM it becomes again arbitrary.

16.21.4 The Function CANONICAL

CANONICAL is the most important functionality of the package. It can be applied on any polynomial whether it is a dummy polynommial or not. It returns a normal form uniquely determined from the current ordering of the system. If the polynomial does not contain any dummy index, it is rewriten taking into account the various operator properties or symmetries described above. For instance,

```
symtree (r, {!+, {!-, 1, 2}, {!-, 3, 4}});

aa:=r(x3,x4,x2,x1)$

canonical aa; ==>  - r(x1,x2,x3,x4).
```

If it contains dummy indices, CANONICAL takes also into account the various dummy summations, makes the relevant simplifications, eventually rename the dummy indices and returns the resulting normal form. Here is a simple example:

```
operator at1,at2;
anticom at1,at2;

dummy_names i,j,k; ==> t

show_dummy_names(); ==> {i,j,k}

rr:=at1(i)*at2(k) -at2(k)*at1(i)$


canonical rr; => 2*at1(i)*at2(j)
```

It is important to notice, in the above example, that in addition to the summations over indices i and k, and of the anticommutativity property of the operators, canonical has replaced the index k by the index j. This substitution is essential to get full simplification. Several other examples are given in the test file and,

there, the output of CANONICAL is explained.

As stated in the previous section, the dependence of operators on dummy indices is limited to *first* level. An erroneous result will be generated if it is not the case as the subsequent example illustrates:

```
operator op;

dummy_names i,j;

rr:=op(i,op(j))-op(j,op(j))$

canonical rr; ==> 0
```

Zero is obtained because, in the second term, CANONICAL has replaced j by i but has left $op(j)$ unchanged because it *does not see* the index j which is inside. This fact has also the consequence that it is unable to simplify correctly (or at all) expressions which contain some derivatives. For instance (i and j are dummy indices):

```
aa:=df(op(x,i),x) -df(op(x,j),x)$

canonical aa; ==> df(op(x,i),x) - df(op(x,j),x)
```

instead of zero. A second limitation is that CANONICAL does not add anything to the problem of simplifications when side relations (like Bianchi identities) are present.

16.21.5 Bibliography

- **Butler, G. and Lam, C. W. H.**, "A general backtrack algorithm for the isomorphism problem of combinatorial objects", J. Symb. Comput. vol.1, (1985) p.363-381.
- **Butler, G. and Cannon, J. J.**, "Computing in Permutation and Matrix Groups I: Normal Closure, Commutator Subgroups, Series", Math. Comp. vol.39, number 60, (1982), p. 663-670.
- **Butler, G.**, "Computing in Permutation and Matrix Groups II: Backtrack Algorithm", Math. Comp. vol.39, number 160, (1982), p.671-680.
- **Leon, J.S.**, "On an Algorithm for Finding a Base and a Strong Generating Setfor a Group Given by Generating Permutations", Math. Comp. vol.35, (1980), p941-974.
- **Leon, J. S.**, "Computing Automorphism Groups of Combinatorial Objects", Proc. LMS Symp. on Computational Group Theory, Durham, England, editor: Atkinson, M. D., Academic Press, London, (1984).

- **Leon, J. S.**, "Permutation Group Algorithms Based on Partitions, I: Theory and Algorithms", J.Symb. Comput.vol.12, (1991) p. 533-583.
- **Linton, Stephen Q.**, "Double Coset Enumeration", J. Symb. Comput., vol.12, (1991) p. 415-426.
- **McKay, B. D.**, "Computing Automorphism Groups and Canonical Labellings of Graphs", Proc. Internat. Conf. on Combinatorial Theory, Lecture Notes in Mathematics" vol. 686, (1977), p.223-232, Springer-Verlag, Berlin.
- **Rodionov, A. Ya. and Taranov, A. Yu.**, "Combinatorial Aspects of Simplification of Algebraic Expression", Proceedings of Eurocal 87, Lecture Notes in Comp. Sci., vol. 378, (1989), p. 192.
- **Sims, C. C.**, "Determining the Conjugacy Classes of a Permutation Group", Computers in Algebra and Number Theory, SIAM-AMS Proceedings, vol. 4, (1971), p. 191-195, editor G. Birckhoff and M. Hall Jr., Amer. Math. Soc..
- **Sims, C. C.**, "Computation with Permutation Groups", Proc. of the Second Symposium on Symbolic and Algebraic Manipulation, (1971), p. 23-28, editor S. R. Petrick, Assoc. Comput. Mach., New York.
- **Burnel A., Caprasse H., Dresse A.**, " Computing the BRST operator used in Quantization of Gauge Theories" IJMPC vol. 3, (1993) p.321-35.
- **Caprasse H.**, "BRST charge and Poisson Algebras", Discrete Mathematics and Theoretical Computer Science, Special Issue: Lie Computations papers, http://dmtcs.thomsonscience.com, (1997).