AEI-2006-037
cs.SC/0608005

A field-theory motivated approach to symbolic computer algebra

Kasper Peeters

Max-Planck-Institut für Gravitationsphysik, Albert-Einstein-Institut
Am Mühlenberg 1, 14476 Golm, GERMANY

Abstract

Field theory is an area in physics with a deceptively compact notation. Although general purpose computer algebra systems, built around generic list-based data structures, can be used to represent and manipulate field-theory expressions, this often leads to cumbersome input formats, unexpected side-effects, or the need for a lot of special-purpose code. This makes a direct translation of problems from paper to computer and back needlessly time-consuming and error-prone. A prototype computer algebra system is presented which features TEX-like input, graph data structures, lists with Young-tableaux symmetries and a multiple-inheritance property system. The usefulness of this approach is illustrated with a number of explicit field-theory problems.

1. Field theory versus general-purpose computer algebra

For good reasons, the area of general-purpose computer algebra programs has historically been dominated by what one could call "list-based" systems. These are systems which are centred on the idea that, at the lowest level, mathematical expressions are nothing else but nested lists (or equivalently: nested functions, trees, directed acyclic graphs, ...). There is no doubt that a lot of mathematics indeed maps elegantly to problems concerning the manipulation of nested lists, as the success of a large class of LISP-based computer algebra systems illustrates (either implemented in LISP itself or in another language with appropriate list data structures). However, there are certain problems for which a pure list-based approach may not be the most elegant, efficient or robust one.

That a pure list-based approach does not necessarily lead to the fastest algorithms is of course well-known. For e.g. polynomial manipulation, there exists a multitude of other representations which are often more appropriate for the problem at hand. An area for

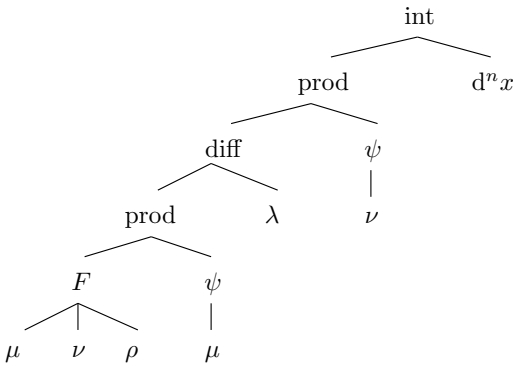
Email address: kasper.peeters@aei.mpg.de (Kasper Peeters).

which the limits of a pure list-based approach have received less attention consists of what one might call "field theory" problems. Without attempting to define this term rigorously, one can have in mind problems such as the manipulation of Lagrangians, field equations or symmetry algebras; the examples discussed later will define the class of problems more explicitly. The standard physics notation in this field is deceptively compact, and as a result it is easy to overlook the amount of information that is being manipulated when one handles these problems with pencil and paper. As a consequence, problems such as deriving the equations of motion from an action, or verifying supersymmetry or BRST invariance, often become a tedious transcription exercise when one attempts to do them with existing general-purpose computer algebra systems. Here, the inadequateness of simple lists is not so much that it leads to sub-optimal, slow solutions (although this certainly also plays a role at some stage), but rather that it prevents solutions from being developed at all.

To make the problems more concrete, let us consider a totally arbitrary example of the type of expressions which appear in field theory. A typical Lagrangian or Noether charge might contain terms of the type

$$\int \mathrm{d}^n x \, \frac{\partial}{\partial x^\lambda} \Big(F_{\mu\nu\rho} \, \psi^\mu \Big) \, \psi^\nu \, . \tag{1}$$

Let us take, purely as an example, $F_{\mu\nu\rho}$ to be a commuting field strength of some two-form field, ψ^μ an anti-commuting vector, and x^μ to label an n-dimensional space. Traditionally, one would represent (1) in the computer as a nested list, which in tree-form would take the form



The precise details do not matter here; the important point is that the expression takes the form of a multiply nested list. However, this list can only be the starting point, since expression (1) clearly contains much more information than just the tree structure. This leads to a number of "standard" problems which field-theory computer algebra systems have to face:

- The names of contracted indices do not matter, in fact, it is only the contraction which is relevant. The nested list structure does not capture the cyclic graph-structure inherent in the expression. How do we make a list-based program aware of this fact, especially when doing substitutions? (this problem is more colloquially known as the "dummy index" problem).

- The expression is, for the "outside world", an object with two free indices, λ and ρ. However, these indices, or graph edges, occur at different depths in the tree. How can we access the nested list by the free edges of the tree?
- The reason why F and ψ should stay as children of the diff node is that they depend on x. Where do we store this information? And how do we make algorithms aware of it?
- The diff and ψ child nodes of the prod node cannot be moved through each other, because the diff node contains a ψ. In other words, the diff node "inherits" the anti-commutativity from one of its child nodes. How do we handle this?
- The anti-symmetry of F relates several contraction patterns. However, there are more complicated symmetries present in the expression. The Bianchi identity on the field strength, for instance, is a multi-term relation involving the indices on F and the index on diff. How do we make the program aware of this identity, and how do we take it into account when reducing expressions to a canonical form?
- For physicists, the simplest way to write expressions such as (1) is to use TEX notation, for example

```
\int d^nx \partial_{\lambda} ( F_{\mu\nu\rho} \psi^{\mu} ) \psi^{\nu}
```

  Being able to input expressions like this would eliminate a large number of errors in transcribing physics problems to computer algebra systems, and make it much easier to use the computer as a scratch pad for tedious calculations (in particular, it would eliminate a good part of the learning curve of the program). Although TEX notation certainly lacks the semantics to be used as input language for a generic computer algebra system (see e.g. [1] for a discussion of this problem), it is not hard to come up with a subset of TEX notation which is both easily understandable and mathematically unambiguous. But how do we teach an existing general purpose system to deal with input of this type?

This collection of problems suggest that a general-purpose system based on "nested lists" is a rather bare-bones tool to describe field-theory problems. The nested list is just one of the many possible views or representations of the (rather heavily labelled) graph structure representing the expression. While it is perfectly possible to tackle many of the problems mentioned above in a list-based system (as several tensor algebra packages for general purpose computer algebra systems illustrate [2–5]), this may not be the most elegant, efficient or robust approach (the lack of a system which is able to solve all of the sample problems in section 3 in an elegant way exemplifies this point). By endowing the core of the computer algebra system with data structures which are more appropriate for the storage of field-theory expressions, it becomes much simpler to write a computer algebra system which can resolve the problems listed above. [1]

The remainder of this paper describes the key ingredients in an approach taken in the prototype computer algebra system "cadabra". Full details of this program, including source code, binaries and a reference manual, can be found at the web site [7].

[1] There are of course *other subclasses* of field-theory problems for which some of the points raised here are irrelevant and efficient computer algebra systems have been developed; see e.g. [6] for an example.

2. Design goals and implementation

This section describes in more detail the main features of the cadabra program: the internal graph views of the tree structure, its handling of node symmetries and the use of a multiple-inheritance property system. In addition to these features, cadabra also has a number of other characteristics which make it especially tuned to the manipulation of "field theory" problems. An important characteristic which should not remain unmentioned is the fact that cadabra accepts TEX-like notation for tensorial expressions, making it much easier to transcribe problems from and to paper. The program can be used both from a text-based command-line interface as well as from a graphical front-end or from within TEXmacs [8]. I will not discuss the TEX input/output in detail, but examples can be found in section 3.

2.1. Graph structure

Cadabra is a standalone program written in C++. As in most other computer algebra systems, the internal data storage used in cadabra is that of a tree. Concretely, this is implemented using a custom tree container class [9] based on STL ideas [10]. However, what sets the program apart from other systems is that a) the tree structure contains more data than usual, to make it easier to represent field-theory problems in a compact way, and b) the tree manipulation algorithms offer several ways of viewing and modifying this tree. In more detail:

- The nodes of the tree carry so-called "parent-relation" information, which determine how child nodes are related to parent nodes. As an example of such a relation, consider the expression $T^{\mu}{}_{\nu}(x)$. This is stored as a node T, with three children μ, ν and x, which have parent relations "superscript", "subscript" and "argument" respectively (more relations can easily be added in the future if necessary). A common way of storing this information is e.g. `T[mu, -nu, x]` or `T[up[mu], dn[nu], x]`, but both have their disadvantages: the first form does not allow us to store $T^{-\mu}{}_{\nu}(x)$, while the second form introduces an additional layer of overhead. A format similar to the second case is also used in Stensor [11], albeit using a different syntax; it has the disadvantage that it is neither a very convenient representation for the computer (especially not when the implementation is in C++), nor a representation which is convenient for the user, as it is quite distinct from TEX notation.
- The tree class not only provides a way to access the nodes of the graph by pre- or post-order traversal, but also provides iterators which access e.g. only all indices of a node. In the example (1), an index iterator acting on the `diff` node would return, in turn, the μ, ν, ρ, μ and λ indices. This makes it easy to write fast low-level routines which deal directly with the tensor structure of an expression.
- The tree manipulation algorithms are aware of the meaning of "contracted nodes" (contracted indices). Whenever one expression graph is inserted into another one, the algorithms automatically ensure that labels which are used to denote contracted indices (i.e. edges which connect two nodes) are relabelled appropriately. Names are chosen by using property lists (see section 2.3).
- The contraction detection mechanism can deal with sub- or superscripts which do not denote indices, as in e.g. $A^{\dagger}$. This is achieved by attaching a special property to the symbol (see section 2.3 for more details).

The enhanced tree structure can be modified at the level of the user interface through standard list manipulation commands, or at the level of custom modules written in C++.

A second issue which cadabra addresses differently from other computer algebra systems is that of node symmetries. It is common in computer algebra systems that there is a generic way to specify so-called *mono-term* symmetries. These are symmetries which relate one particular ordering of arguments to another one, up to a possible overall sign. Cadabra can of course find canonical representations for tensors with mono-term symmetries (using an external implementation [2] of the double-coset algorithm [12]; an alternative backtracking algorithm for mono-term symmetries is described in [13]).

However, mono-term symmetries do not exhaust the symmetries of tensors transforming in generic representations of the Lorentz group. They do not include Garnir symmetries of Young tableaux. Examples of such symmetries are the Ricci identity for Riemann tensors,

$$R_{mnpq} + R_{mpqn} + R_{mqnp} = 0\,, \tag{2}$$

or the Bianchi identity for field strengths. These identities relate more than two terms, and are therefore also called *multi-term* symmetries. Despite the clear importance of being able to take such identities into account, there are very few computer algebra systems which have implemented a solution. The implementation in [5] simply uses a large set of transformation rules for Riemann tensor monomials. These rules were constructed by hand, and are only available for Riemann tensor monomials up to third order (i.e. it would require tedious work to construct such rules for more general expressions, involving more than just the Riemann or Ricci tensors). An alternative approach is taken in [3, 11, 14], in which the set of all identities for a particular tensor is used to rewrite a given expression in canonical form. This idea of handling multi-term symmetries using a sum-substitution algorithm goes back to at least [15].

Cadabra, instead, uses Young projector methods internally for all index symmetry handling. The underlying idea is that by applying a Young projector to a tensor, its multi-term symmetries become manifest. This allows one to construct a basis of tensor monomials constructed from arbitrary tensors, and to decompose a given monomial on any preferred basis.[2] This method was first described in [17]. For e.g. Riemann tensors, the idea is to replace all tensors by their equivalent form

$$R_{abcd} \rightarrow \tfrac{1}{3}\big(2\,R_{abcd} - R_{adbc} + R_{acbd}\big)\,. \tag{3}$$

The expression on the right-hand side manifestly satisfies the cyclic Ricci identity, even if one only knows about the mono-term symmetries of the Riemann tensor. Using the projector (3) it is easy to show e.g. that $2\,R_{abcd}R_{acbd} = R_{abcd}R_{abcd}$. The monomial on the left-hand side maps to

$$R_{abcd}R_{acbd} \rightarrow \tfrac{1}{3}\big(R_{abcd}R_{acbd} + R_{abcd}R_{abcd}\big)\,, \tag{4}$$

while $R_{abcd}R_{abcd}$ maps to twice this expression, thereby proving the identity.

Writing each term in a sum in a canonical form by using (3) would typically lead to extremely large expressions, and not be very convenient for subsequent calculations.

[2] In order to determine the number of terms in a basis of monomials of tensors, cadabra relies on an external program for the computation of tensor product representations (using the LiE program [16]).

However, the same algorithm can also be used to write a sum in a "minimal" form.[3] That is, by projecting each term using (3) the program can perform the simplification

$$R_{abcd}R_{acbd} + R_{abcd}R_{abcd} \to 3\, R_{abcd}R_{acbd}\,, \tag{5}$$

i.e. express the second term in terms of the first one. This does not define a canonical form (the expression could equally well have been written using $R_{abcd}R_{abcd}$), but it does systematically eliminate terms which can be written as linear combinations of other terms.

2.3. Properties

A third problem for which cadabra takes a different approach from other systems is that of "typing" of symbols and expressions. In cadabra, the meaning of symbols or expressions is determined by associating *properties* to them. Properties can be simple, such as "being an integer", or "being anti-symmetric in all indices", or "being an index which is not to be summed over" (cf. the discussion in section 2.1). They can also be more complicated and composite, such as "being an anti-commuting spinor in the left-handed Weyl representation of the eight-dimensional Lorentz group".

The general problem of deducing properties of composite objects from the properties of their constituents is a hard (see e.g [18]). Cadabra takes a pragmatic approach, trying to provide a useful property system for concrete problems rather than trying to be complete or mathematically rigorous. Properties are implemented as a standard multiple-inheritance tree of C++ objects. The association to symbols is stored in a map, which relates patterns to pointers to property objects.[4] This makes it relatively easy to make properties inherit from each other. An example of an inherited property is the property `PartialDerivative`, which inherits from `TableauBase`, so that the symmetry information of objects on which a partial derivative acts are automatically propagated.

Nodes can inherit properties from child nodes. A simple situation in which this is useful is for instance when one uses accents to mark symbols, as in e.g. $\bar{\psi}\psi$. If `\psi` is declared to be self-anticommuting, we obviously want the `\bar{\psi}` tree to have this property as well. When scanning for properties of nodes, the internal algorithms take into account such inheritance of properties. Inheritance of a property is, itself, again implemented as a property (in the example above, the `\bar` node is declared to have the property `PropertyInherit`, while more fine-tuned inheritance is implemented by deriving from a templated `Inherit` class, as in e.g. `Inherit<Spinor>`).[5]

---

[3] "Minimal" here does not necessarily mean that the expression has been reduced to the shortest possible form, which is a problem which to the best of my knowledge remains unresolved. That is, while the algorithm removes dependent terms, as in $2\,R_{abcd} + 2\,R_{bcad} + R_{cabd} \to R_{abcd} + R_{bcad}$ (because the third term is found to be expressible as a linear combination of the first two), it does not reduce this further to $-R_{cabd}$ (typical cases are of course more complicated than this example).

[4] It is important that such properties are implemented at a low level. Most computer algebra systems would allow one to implement e.g. handling of sets of non-commuting objects using user-defined property testing functions and appropriate transformation rules. It is a much harder problem to make sure that all routines of the underlying system use these properties efficiently and correctly.

[5] This is similar to Macsyma's types and features: the property which is attached to a symbol is like a 'type', while all properties which the symbol inherits from child nodes are like 'features'. Property inheritance can also be found other systems, e.g. Axiom [19].

Not all property inheritance is, however, as simple as propagating the information up from a deeper lying node. More complicated property inheritance occurs when nodes have to "compute" their properties from properties of the child nodes. This occurs for instance when we want to know how products of symbols commute among each other. For such cases, there are more complicated property classes, for instance `CommutingAsProduct` or `CommutingAsSum`. Similarly, there is a property `IndexInherit` which indicate that nodes should make the indices of their child nodes visible to the outside world. Other composite property objects can easily be added to the system.

3. Typical examples

In this section, the three main points discussed in the previous section main text (enhanced tree data structures & algorithms, the use of representation theory to classify object symmetries, and the use of properties) will be illustrated with a number of explicit examples. These examples are meant to be readable without further information about the program language. As such, they also illustrate the ease with which tensorial expressions can be fed into the program. Full details of the input language and transformation algorithms can be found in the manual [7].

3.1. Index handling and substitution

When doing computations by hand, we do index relabelling almost automatically when a clash occurs. However, unless the computer program is aware of this problem at a low level, clashes are bound to occur frequently. Consider first the standard type of relabelling, illustrated by the expressions

$$C = A^2\,, \quad \text{with} \quad A = B_{mn}B_{mn} \quad \text{and} \quad B_{np} = T_{mn}T_{mp}\,. \tag{6}$$

In cadabra one can e.g. do [6]

```
{m,n,p,q#}::Indices(vector).
C:= A A;
@substitute!(%)( A = B_{m n} B_{m n} );
@substitute!(%)( B_{n p} = T_{m n} T_{m p} );
```

where the meaning of the hash symbol on the declaration of the q index (in the first line) will become clear soon. The result is

```
C:= T_{q2 m} T_{q2 n} T_{q3 m} T_{q3 n} T_{q4 p} T_{q4 q1} T_{q5 p} T_{q5 q1};
```

This type of relabelling and automatic index generation is not an entirely uncommon feature to find in tensor algebra systems, although it is often implemented in an add-on

[6] As alluded to in the first section, the notation used here is not generic TEX but rather a well-defined subset, with some additional conventions required to make the input unambiguous. An example of such a convention is the use of spaces to separate indices; further details about the input format conventions can be found in the reference manual [7].

package. The situation becomes more complicated when we have indices which do not occur at the same level, for instance

$$C = A^2\,, \quad \text{with} \quad A = \partial_m(B_n B_p + C_{np})B_{mnp} \quad \text{and} \quad B_n = T_{nm}S_m\,. \tag{7}$$

Few systems know how to deal with these types of expressions elegantly (i.e. without requiring a cumbersome input format). The reason is that the derivative carries an index, but the objects in the product on which it acts carry indices too, and these indices do not all occur at the same depth of the expression tree. The cadabra instructions, however, remain equally simple as in the previous example,

```
{m,n,p,q#}::Indices(vector).
\partial{#}::Derivative.
C:= A A;
@substitute!(%)( A = \partial_{m}( B_n B_p + C_{n p} ) B_{m n p} );
@substitute!(%)( B_n = T_{n m} S_{m} );
```

The result comes out as the expected

```
C:= \partial_{m}(T_{n q4} S_{q4} T_{p q5} S_{q5} + C_{n p}) B_{m n p}
    \partial_{q1}(T_{q2 q6} S_{q6} T_{q3 q7} S_{q7} + C_{q2 q3}) B_{q1 q2 q3};
```

Finally, it of course happens frequently that more than one type of index appears in an expression, labelling tensors in different spaces. Consider for instance,

$$C = A^2 \quad \text{with} \quad A_{m\mu} = \bar{\psi}\Gamma_{mp}\psi\, B_{p\mu}\,, \tag{8}$$

where the roman and Greek indices cannot be interchanged at will, because they refer to flat and curved spaces respectively. This example translates to

```
{\mu, \rho, \nu#}::Indices(curved).
{m, n, p, q#}::Indices(flat).
C:= A_{m \nu} A_{m \nu};
@substitute!(%)( A_{m \mu}
          =  \bar{\psi}\Gamma_{m p} \psi B_{p \mu \rho} C_{\rho});
```

with the expected result

```
C:= \bar{\psi} \Gamma_{m p} \psi B_{p \nu \rho} C_{\rho}
    \bar{\psi} \Gamma_{m n} \psi B_{n \nu \mu} C_{\mu};
```

All this type of relabelling is done by the internal tree manipulation algorithms, which ensures that no algorithm can lead to inconsistent expressions. New dummy indices are taken from the appropriate sets, using the property information associated to the various indices.

As long as one deals only with symmetric or antisymmetric tensors, many computer algebra systems are able to write tensor monomials in a canonical form (although efficient algorithms for very large numbers of indices or very large numbers of identical tensors have only surfaced relatively recently, see [2, 12, 20, 21]). Generic algorithms for multi-term Garnir symmetries, such as the Ricci or Bianchi identity, are much less widespread; see the discussion in section 2.2. Cadabra is the first system to label tensors by Young tableaux and to use Young projector methods to handle multi-term symmetries.

A common problem in which multi-term symmetries play an important role is the construction of a basis of all tensor monomials of a given length dimension. Determining the number of elements of such a basis is a relatively straightforward exercise in group theory [22]. In order to actually construct the basis, cadabra uses the Young projector method described in the appendix of [17]. As an example, let us construct a basis of monomials cubic in the Riemann tensor,

```
{m,n,p,q,r,s,t,u,v,w,a,b}::Indices(vector).
{m,n,p,q,r,s,t,u,v,w,a,b}::Integer(0..9).
R_{m n p q}::RiemannTensor.

basisR3:= R_{m n p q} R_{r s t u} R_{v w a b};
@all_contractions(%);

@canonicalise!(%):
@substitute!(%)( R_{m n m n} -> R ):
@substitute!(%)( R_{m n m p} -> R_{n p} );
```

After a declaration of the objects to be used, the program determines in one step all possible independent contractions of three Riemann tensors. The last two lines only serve to rewrite the result in terms of Ricci tensors and scalars, after which the output takes the form

```
basisR3:= \{ R_{m n p q} R_{m p r s} R_{n r q s},
             R R_{q r} R_{q r},
             R_{n p} R_{n q p r} R_{q r},
             R_{n p} R_{n q r s} R_{p r q s},
             R R_{p q r s} R_{p q r s},
             R_{n p} R_{n r} R_{p r},
             R_{m n p q} R_{m r p s} R_{n r q s},
             R R R \};
```

This result is equivalent to the basis given in the "$\mathcal{R}^0_{6,3}$" table on page 1184 of [22].

It is also possible to decompose any given tensor monomial on a previously constructed basis. Take for example the basis of Weyl tensor monomials of fourth order. This basis

can be read off from the tables of [22],

$$
\begin{aligned}
W_1 &= W_{mnab} W_{npbc} W_{pscd} W_{smda} \,, \\
W_2 &= W_{mnab} W_{npbc} W_{mscd} W_{spda} \,, \\
W_3 &= W_{mnab} W_{psba} W_{mncd} W_{psdc} \,, \\
W_4 &= W_{mnab} W_{mnba} W_{pscd} W_{psdc} \,, \\
W_5 &= W_{mnab} W_{npba} W_{pscd} W_{smdc} \,, \\
W_6 &= W_{mnab} W_{psba} W_{mpcd} W_{nsdc} \,, \\
W_7 &= W_{mn}{}^{[mn} W_{pq}{}^{pq} W_{rs}{}^{rs} W_{tu}{}^{tu]} \,.
\end{aligned}
\tag{9}
$$

If we want to find the decomposition

$$
W_{pqrs} W_{ptru} W_{tvqw} W_{uvsw} - W_{pqrs} W_{pqtu} W_{rvtw} W_{svuw} = W_2 - \tfrac{1}{4} W_6 \,, \tag{10}
$$

using "classical" methods, we would need to figure out the right way to repeatedly apply the Ricci cyclic identity to the left-hand side of this expression. The appropriate program to decompose the left-hand side on the seven-term basis and prove this identity is

```
{m,n,p,q,r,s,t,u,v,w,a,b,c,d,e,f}::Indices(vector).
W_{m n p q}::WeylTensor.

W1:= W_{m n a b} W_{n p b c} W_{p s c d} W_{s m d a};
W2:= W_{m n a b} W_{n p b c} W_{m s c d} W_{s p d a};
W3:= W_{m n a b} W_{p s b a} W_{m n c d} W_{p s d c};
W4:= W_{m n a b} W_{m n b a} W_{p s c d} W_{p s d c};
W5:= W_{m n a b} W_{n p b a} W_{p s c d} W_{s m d c};
W6:= W_{m n a b} W_{p s b a} W_{m p c d} W_{n s d c};
W7:= W_{m n}^{m n} W_{p q}^{p q} W_{r s}^{r s} W_{t u}^{t u};
@asym!(%){^{m},^{n},^{p},^{q},^{r},^{s},^{t},^{u}}:
@substitute!(%)( W_{a b}^{c d} -> W_{a b c d} ):
@indexsort!(%):
@collect_terms!(%):
@canonicalise!(%):
@collect_terms!(%);

basisW4:= { @(W1), @(W2), @(W3), @(W4), @(W5), @(W6), @(W7) };

W_{p q r s} W_{p t r u} W_{t v q w} W_{u v s w}
  - W_{p q r s} W_{p q t u} W_{r v t w} W_{s v u w};
@decompose!(%){ @(basisW4) };
@list_sum!(%);
@collect_terms!(%);
```

Most of this code is self-explanatory. The first two lines declare the symbols and objects to be used, the next block of lines declares the basis and performs the eight-fold anti-

symmetrisation for the last basis element.[7] The decomposition is done with the last three lines. The final output of this small program reads

```
{0, 1, 0, 0, 0, -1/4, 0 };
```

Internally, this involved a Young-projection of all tensors in the basis, a projection of the tensors in the expression which we want to decompose, and a solution of a system of linear equations [17]. The internal algorithm is completely generic and applies to tensor monomials with arbitrary symmetries.

3.3. Properties and property inheritance

A typical class of problems in which one handles tensors of both commuting and anti-commuting type is the construction of supersymmetric actions. This class of problems also shows the use of implicit dependence of tensors on coordinates, as well as inheritance of spinor and anti-commutativity properties.

Consider as a trivial example – which is nevertheless not easy to reproduce with other computer algebra systems – the invariance of the super-Maxwell action

$$S = \int \mathrm{d}^4x \left[-\frac{1}{4}(f_{ab})^2 - \frac{1}{2}\bar{\lambda}\gamma^a\partial_a\lambda \right], \tag{11}$$

(where $f_{ab} = \partial_a A_b - \partial_b A_a$) under the transformations

$$\delta A_a = \bar{\epsilon}\gamma_a\lambda\,, \quad \delta\lambda = -\frac{1}{2}\gamma^{ab}\epsilon\, f_{ab}\,. \tag{12}$$

The object properties for this problem are

```
{ a,b,c,d,e }::Indices(vector).
\bar{#}::DiracBar.
{ \partial{#}, \ppartial{#} }::PartialDerivative.
{ A_{a}, f_{a b} }::Depends(\partial, \ppartial).
{ \epsilon, \gamma_{#} }::Depends(\bar).
\lambda::Depends(\bar, \partial).
{ \lambda, \gamma_{#} }::NonCommuting.
{ \lambda, \epsilon }::Spinor(dimension=4, type=Majorana).
{ \epsilon, \lambda }::SortOrder.
{ \epsilon, \lambda }::AntiCommuting.
\lambda::SelfAntiCommuting.
\gamma_{#}::GammaMatrix.
\delta{#}::Accent.
f_{a b}::AntiSymmetric.
\delta_{a b}::KroneckerDelta.
```

Note the use of two types of properties: those which apply to a single object, like `Depends`, and those which are associated to a list of objects, like `AntiCommuting`. Clearly $\partial_a\lambda$ and

[7] Commands such as `@collect_terms` can be added to a list of default rules to be applied automatically; they have been included here so that all steps are explicit.

ϵ are anti-commuting too, but the program figures this out automatically from the fact that `\partial` has a `PartialDerivative` property associated to it.

The actual calculation is an almost direct transcription of the calculation one would do by hand.[8] First we define the supersymmetry transformation rules and the action, which can be entered as in TEX,

```
susy:= { \delta{A_{a}}   = \bar{\epsilon} \gamma_{a} \lambda,
         \delta{\lambda} = -(1/2) \gamma_{a b} \epsilon f_{a b} };

S:= -(1/4) f_{a b} f_{a b}
              - (1/2) \bar{\lambda} \gamma_{a} \partial_{a}{\lambda};
```

Showing invariance starts by applying a variational derivative,

```
@vary!(%)( f_{a b} -> \partial_{a}{\delta{A_{b}}} - \partial_{b}{\delta{A_{a}}},
           \lambda -> \delta{\lambda} );

@distribute!(%);
@substitute!(%)( @(susy) ): @prodrule!(%): @distribute!(%): @unwrap!(%);
```

After these steps, the result is (shown exactly as it appears in the graphical and the TEXmacs [8] front-ends)

$$S = \bar{\epsilon}\gamma_a\partial_b\lambda\, f_{ab} + \frac{1}{4}\overline{\gamma_{ab}\epsilon}\gamma_c\partial_c\lambda\, f_{ab} + \frac{1}{4}\bar{\lambda}\gamma_a\gamma_{bc}\epsilon\partial_a f_{bc}\,. \tag{13}$$

Since the program knows about the properties of gamma matrices it can rewrite the Dirac bar, and then we do one further partial integration,

```
@rewrite_diracbar!(%);
@substitute!(%)( \partial_{c}{f_{a b}} -> \ppartial_{c}{f_{a b}} ):
@pintegrate!(%){\ppartial}:
@rename!(%){"\ppartial"}{"\partial"}:
@prodrule!(%): @unwrap!(%);
```

What remains is the gamma matrix algebra, a rewriting of the derivative of the Dirac bar as the Dirac bar of a derivative, and sorting of spinors (which employs inheritance of the `Spinor` and `AntiCommuting` properties as already alluded to earlier),

```
@join!(%){expand}: @distribute!(%): @eliminate_kr!(%):
@substitute!(%)( \partial_{a}{\bar{\lambda}} -> \bar{\partial_{a}{\lambda}} );
@spinorsort!(%):
```

[8] This example makes use of a set of default rules, to wit "`::PostDefaultRules( @@prodsort!(%), @@rename_dummies!(%), @@canonicalise!(%), @@collect_terms!(%) )`", which mimick the automatic rewriting behaviour of many other computer algebra systems and get invoked automatically at each step. See [7] for more details.

The result is (after partial integration) a Bianchi identity on the field strength, and thus invariance of the action.

While this example is rather simple, and does not require a computer algebra system for its solution, it illustrates that the extended tree structure together with the property system make it possible to manipulate expressions in a way which closely resembles what one would do when solving the problem with pencil and paper. Several more complicated examples will be discussed in the upcoming [23].

4. Summary

I have presented a new prototype computer algebra system which is designed to be an easy-to-use scratch pad for problems encountered in field theory. The current library of algorithms include functionality to deal with bosonic and fermionic tensors, spinors, gamma matrices, differential operators and so on, all through the use of a multiple-inheritance property mechanism. Cadabra is the first system which handles generic multi-term tensor symmetries using a Young-projector based algorithm. It is also the first system which accepts input in TEX form, eliminating tedious translation steps and making programs much easier to read for new users. Finally, the source code of the system is freely available and the reference guide contains extensive documentation explaining how to add new algorithm modules to the program.

Acknowledgements

I am grateful to José Martin-Garcia for inspiring discussions and for help with the use of his `xPerm` code [2] for mono-term canonicalisation. I thank the anonymous referee for extensive comments which have substantially improved this paper.

References

[1] R. J. Fateman and E. Caspi, "Parsing TEX into mathematics", *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)* **33** (1999), no. 3, 26.

[2] J. Martin-Garcia, "xPerm and xAct", `http://metric.iem.csic.es/Martin-Garcia/xAct/index.html`.

[3] A. Balfagón, P. Castellví, and X. Jaén, "Tools of tensor calculus", `http://baldufa.upc.es/xjaen/ttc/`.

[4] S. A. Klioner, "EinS: A Mathematica package for computations with indexed objects", `gr-qc/0011012`.

[5] L. Parker and S. M. Christensen, "Mathtensor : A system for doing tensor analysis by computer", Addison-Wesley, 1994.

[6] J. A. M. Vermaseren, "New features of FORM", `math-ph/0010025`.

[7] K. Peeters, "Cadabra: tutorial and reference guide", 2006, `http://www.aei.mpg.de/~peekas/cadabra/`.

[8] J. van der Hoeven, "GNU TeXmacs: A free, structured, wysiwyg and technical text editor", in "Le document au XXI-ième siècle", D. Flipo, ed., vol. 39–40, pp. 39–50. Metz, 14–17 mai 2001. Actes du congrès GUTenberg. http://www.texmacs.org.

[9] K. Peeters, "Tree.hh", 2006, http://www.aei.mpg.de/~peekas/tree/.

[10] D. R. Musser and A. Saini, "STL tutorial and reference guide, C++ programming with the standard template library", Addison Wesley, 1996.

[11] M. A. H. MacCallum and J. E. F. Skea, "SHEEP: A computer algebra system for general relativity", in "Algebraic computing in general relativity", M. J. Rebouças and W. L. Roque, eds., pp. 1–172. Oxford, 1994.

[12] R. Portugal, "Algorithmic simplification of tensor expressions", *J. Phys.* **A32** (1999) 7779–7789.

[13] A. Dresse, "Polynomial poisson structures and dummy variables in computer algebra", PhD thesis, Université Libre de Bruxelles, 1993.

[14] V. A. Ilyin and A. P. Kryukov, "ATENSOR - REDUCE program for tensor simplification", *Comp. Phys. Commun.* **96** (1996) 36–52.

[15] L. Hornfeldt, "A system for automatic generation of tensor algorithms and indicial tensor calculus, including substitution of sums", in "Proceedings of EUROSAM 79", Ng, ed., vol. 72 of *Lecture Notes in Computer Science*, pp. 279–290. Springer, 1979.

[16] A. Cohen, M. van Leeuwen, and B. Lisser, "LiE v. 2.2", 1998, http://wwwmathlabo.univ-poitiers.fr/~maavl/LiE/.

[17] M. B. Green, K. Peeters, and C. Stahn, "Superfield integrals in high dimensions", *JHEP* **08** (2005) 093, hep-th/0506161.

[18] T. Weiberl and G. H. Gonnet, "An algebra of properties", in "Proceedings of the ISSAC-91 Conference, Bonn", pp. 352–359. 1991.

[19] T. Daly, "Axiom volume 1: tutorial", Lulu press, 2005.

[20] R. Portugal, "An algorithm to simplify tensor expressions", *Comp. Phys. Commun.* **115** (1998) 215–230, gr-qc/9803023.

[21] R. Portugal, "The Canon package", http://www.cbpf.br/~portugal/Canon.html.

[22] S. A. Fulling, R. C. King, B. G. Wybourne, and C. J. Cummins, "Normal forms for tensor polynomials. 1: The Riemann tensor", *Class. Quant. Grav.* **9** (1992) 1151.

[23] K. Peeters, "Introducing Cadabra: a symbolic computer algebra system for field theory problems", hep-th/0701238.