November 16, 2020

The Cadabra Book

A field-theory motivated approach to symbolic computer algebra

Kasper Peeters

kasper.peeters@cadabra.science

0 Contents

1 Introduction and overview

1.1 Cadabra's design philosophy

Cadabra is built around the fact that many computations do not have one single and unique path between the starting point and the end result. When we do computations on paper, we often take bits of an expression apart, do some manipulations on them, stick them back into the main expression, and so on. Often, the manipulations that we do are far from uniquely determined by the problem, and often there is no way even in principle for a computer to figure out what is 'the best' thing to do.

What we need the computer to do, in such a case, is to be good at performing simple but tedious steps, without enforcing on the user how to do a particular computation. In other words, we want the computer algebra system to be a scratchpad, leaving us in control of which steps to take, not forcing us to return to a 'canonical' expression at every stage.

Most existing computer algebra systems allow for this kind of work flow only by requiring to stick clumsy 'inert' or 'hold' arguments onto expressions, by default always 'simplifying' every input to some form they think is best. Cadabra starts from the other end of the spectrum, and as a general rule keeps your expression untouched, unless you explicitly ask for something to be done to it.

Another key issue in the design of symbolic computer algebra systems has always been whether or not there should be a distinction between the 'data language' (the language used to write down mathematical expressions), the 'manipulation language' (the language used to write down what you want to do with those expressions) and the 'implementation language' (the language used to implement algorithms which act on mathematical expressions). Many computer algebra systems take the approach in which these languages are the same (Axiom, Reduce, Sympy) or mostly the same apart from a small core which uses

a different implementation language (Mathematica, Maple). The Cadabra project is rooted in the idea that for many applications, it is better to keep a clean distinction between these three languages. Cadabra writes mathematics using LaTeX, is programmable in Python, and is under the hood largely written in C++.

1.2 History

Cadabra was originally written around 2001 to solve a number of problems related to higher-derivative supergravity [1, 2]. It was then expanded and polished, and first saw its public release in 2007 [3]. During the years that followed, it became clear that several design decisions were not ideal, such as the use of a custom programming language and the lack of functionality for component computations. Over the course of 2015-2016 a large rewrite took place, which resulted in Cadabra 2.x [4]. This new version is programmable in Python and does both abstract and component computations.

2 The input format

2.1 Input format

2.1.1 Mathematical expressions

The input format of Cadabra is closely related to the notation used by LaTeX to denote tensorial expressions. That is, one can use not only bracketed notation to denote child objects, like in

```
object[child,child]
```

but also the usual sub- and superscript notation like

```
object^{child child}_{child}
```

One can use backslashes in the names of objects as well, just as in LaTeX. All of the symbols that one enters this way are considered "passive", that is, they will go into the expression tree just like one has entered them.

Expressions are entered by using the ':=' operator, as in

```
ex:=A+B+C_{m} C^{m};
```

$$A + B + C_m C^m$$

Expressions (the 'ex' above) are ordinary Python objects (of type `cadabra2.Ex`), and their names can thus only contain normal alphanumeric symbols.

```
type(ex);
```

```
<class 'cadabra2.Ex'>
```

Lines always have to be terminated with either a ";" or a ":". These delimiting symbols act in the same way as in Maple: the second form suppresses the output of the entered expression. Long expressions can, because of these delimiters, be spread over many subsequent input lines. Any line starting with a "#" sign is considered to be a comment (even when it appears within a multi-line expression). Comments are always ignored completely (they do not end up in the expression tree. When entering maths as above (using the ':=' assignment operator) you do not need to indicate that the right-hand side is mathematics. There are situations, however, when you do need to give Cadabra a hint that what you type is maths, not Python. In this case, you add dollar symbols, just as in LaTeX,

```
substitute($A + B + C$, $C -> D$);
```

$$A + B + D$$

As you can see, this uses an 'inline' definition of a mathematical expression, without giving it a name.

2.1.2 Algorithms

Algorithms are ordinary Python functions, which act on `cadabra2.Ex` objects.

2.2 Printing expressions in various formats

2.2.1 Basic usage

With basic use of Cadabra, you will typically display your expressions by postfixing them with a semi-colon, as in

```
ex:=A_{m n} ( B^{n} + 3 C^{n} );
```

$$A_{mn}\left(B^{n} + 3C^{n}\right)$$

What happens behind the scenes is that the semi-colon gets translated to a call of `display` on the last-entered expression. It is therefore equivalent to

```
display(ex)
```

$$A_{mn}\left(B^{n} + 3C^{n}\right)$$

If you do not want to display the expression, post-fix with a colon, as in

```
ex:=A_{m n} ( B^{n} + 3 C^{n} ):
```

If you want to display an expression again later, you can just write the name of the expression followed by a semi-colon, or use the `display` function again,

```
ex;
display(ex)
```

$$A_{mn}\,(B^n + 3C^n\,)$$

$$A_{mn}\,(B^n + 3C^n\,)$$

Note that while it may be tempting to use `print(ex)`, the `display` function is better because it knows about the capabilities of the interface used, and it will automatically select a text output when you use Cadabra from the terminal, or LaTeX output when you use it in the graphical notebook.

2.2.2 Other output formats

Cadabra expressions are standard Python objects, and as such they have a `__str__` method which converts them into a printable expression, and a `__repr__` method to produce a machine readable form. These are called by the standard `str` and `repr` Python functions, as the examples below show.

```
print(str(ex))
```

```
\begin{verbatim}A_{m n} (B^{n} + 3C^{n})
\end{verbatim}
```

```
print(repr(ex))
```

```
\begin{verbatim}\prod(A_{m n})(\sum(B^{n})(3C^{n}))
\end{verbatim}
```

In addition there are some methods to obtain output useful in other software: Mathematica, LaTeX and Sympy:

```
print(ex.mma_form())
```

```
\begin{verbatim}A[DNm, DNn]*(B[UPn]+3*C[UPn])
\end{verbatim}
```

```
print(ex._latex_())
```

```
\begin{verbatim}A_{m n} \brwrap{(}{B^{n}+3C^{n}}{)}
\end{verbatim}
```

```
print(ex.sympy_form())
```

```
\begin{verbatim}A(DNm, DNn)*(B(UPn)+3*C(UPn))
\end{verbatim}
```

2.3 Object properties and declaration

2.3.1 Generic properties

Symbols in Cadabra have no a-priori "meaning". If you write `\Gamma`, the program will not know that it is supposed to be, for instance, a Clifford algebra generator. You will have to declare the properties of symbols, i.e. you have to tell Cadabra explicitly that if you write `\Gamma`, you actually mean a Clifford algebra generator. This indirect way of attaching a meaning to a symbol has the advantage that you can use whatever notation you like; if you prefer to write `\gamma`, or perhaps even `\rho` if your paper uses that, then this is perfectly possible (object properties are a bit like "attributes" in `Mathematica` or "domains" in `Axiom` and `MuPAD`).

Properties are all written using capitals to separate words, as in `AntiSymmetric`. This makes it easier to distinguish them from commands. Properties of objects are declared by using the "::" characters. This can be done "at first use", i.e. by just adding the property to the object when it first appears in the input. As an example, one can write

```
F_{m n p}::AntiSymmetric;
```

Attached property AntiSymmetric to F_{mnp}.

This declares the object to be anti-symmetric in its indices. The property information is stored separately, so that further appearances of the "`F_{m n p}`" object will automatically share this property. A list of all properties is available from the manual pages on the web site.

Note that properties are attached to patterns, Therefore, you can have

```
R_{m n}::Symmetric;
R_{m n p q}::RiemannTensor;
```

Attached property Symmetric to R_{mn}.

Attached property TableauSymmetry to R_{mnpq}.

at the same time. The program will not warn you if you use incompatible properties, so if you make a declaration like above and then later on do

```
R_{m n}::AntiSymmetric;
```

Attached property AntiSymmetric to R_{mn}.

this may lead to undefined results. The fact that objects are attached to patterns also means that you can use something like wildcards. In the following declaration,

```
{m#, n#}::Indices(vector);
```

Attached property Indices(position=free) to $[m\#,\ n\#]$.

the entire infinite set of objects $m1, m2, m3, \ldots$ and $n1, n2, n3, \ldots$ are declared to be in the dummy index set "vector" (this way of declaring ranges of objects is similar to the "autodeclare" declaration method of FORM). Properties can be assigned to an entire list of symbols with one command, namely by attaching the property to the list. For example,

```
{n, m, p, q}::Integer(1..d);
```

Attached property Integer to $[n,\ m,\ p,\ q]$.

This associates the property "Integer" to each and every symbol in the list. However, there is also a concept of "list properties", which are properties which are associated to the list as a whole. Examples of list properties are "AntiCommuting" or "Indices". Objects can have more than one property attached to them, and one should therefore not confuse properties with the "type" of the object. Consider for instance

```
x::Coordinate;
W_{m n p q}::WeylTensor;
W_{m n p q}::Depends(x);
```

Attached property Coordinate to x.

Attached property TableauSymmetryWeylTensor to W_{mnpq}.

Attached property Depends to W_{mnpq}.

This attaches two completely independent properties to the pattern W_{mnpq}. In the examples above, several properties had arguments (e.g. "`vector`" or "`1..d`"). The general form of these arguments is a set of key-value pairs, as in

```
T_{m n p q}::TableauSymmetry(shape={2,1}, indices={0,2,1});
```

Attached property TableauSymmetry to T_{mnpq}.

In the simple cases discussed so far, the key and the equal sign was suppressed. This is allowed because one of the keys plays the role of the default key. Therefore, the following two are equivalent,

```
{ m, n }::Integer(range=0..d);
{ m, n }::Integer(0..d);
```

Attached property Integer to $[m,\ n]$.

Attached property Integer to $[m,\ n]$.

See the detailed documentation of the individual properties for allowed keys and the one which is taken as the default. Finally, there is a concept of "inherited properties". Consider e.g. a sum of spinors, declared as

```
{\psi_1, \psi_2, \psi_3}::Spinor;
ex:= \psi_1 + \psi_2 + \psi_3;
```

Attached property Spinor to $[\psi_1,\ \psi_2,\ \psi_3]$.

$\psi_1 + \psi_2 + \psi_3$

Here the sum has inherited the property "Spinor", even though it does not have normal or intrinsic property of this type. Properties can also inherit from each other, e.g.

```
\Gamma_{#}::GammaMatrix.
ex:=\Gamma_{p o i u y};
```

Γ_{poiuy}

```
canonicalise(_);
```

$-\Gamma_{iopuy}$

The `GammaMatrix` property inherits from `AntiSymmetric` property, and therefore the `\Gamma` object is automatically anti-symmetric in its indices. indices.

2.3.2 List properties and symbol groups

Some properties are not naturally associated to a single symbol or object, but have to do with collections of them. A simple example of such a property is `AntiCommuting`. Although it sometimes makes sense to say that "ψ_m is anticommuting" (meaning that $\psi_m\psi_n = -\psi_n\psi_m$), it happens just as often that you want to say "ψ and χ anticommute" (meaning that $\psi\chi = -\chi\psi$). The latter property is clearly relating two different objects.

Another example is dummy indices. While it may make sense to say that "m is a dummy index", this does not allow the program to substitute m with another index when a clash of dummy index names occurs (e.g. upon substitution of one expression into another). More useful is to say that "m, n, and p are dummy indices of the same type", so that the program can relabel a pair of m's into a pair of p's when necessary.

In Cadabra such properties are called "list properties". You can associate a list property to a list of symbols by simply writing, e.g. for the first example above,

```
{ \psi, \chi }::AntiCommuting;
```

Attached property AntiCommuting to $[\psi,\ \chi]$.

Note that you can also attach normal properties to multiple symbols in one go using this notation. The program will figure out automatically whether you want to associate a normal property or a list property to the symbols in the list.

Lists are ordered, although the ordering does not necessarily mean anything for all list properties (it is relevant for e.g. `SortOrder` but irrelevant for e.g. `AntiCommuting`).

2.3.3 Querying properties

For many built-in algorithms, assigning properties to the objects which appear in your expressions will be enough to make them work. However, sometimes you may want to explicitly query whether a particular symbol has a particular property. The following example shows how this works.

```
A_{m n}::Symmetric;
if Symmetric.get($A_{m n}$):
   print("A_{m␣n}␣is␣symmetric.")
if AntiSymmetric.get($A_{m n}$):
   print("A_{m␣n}␣is␣anti-symmetric.")
```

Property Symmetric attached to A_{mn}.

```
\begin{verbatim}A_{m n} is symmetric.
\end{verbatim}
```

The object returned by `Property.get(...)` is either `None` or the property which you asked about. It is possible to do something with that property, e.g. attach it to another symbol. In the example below, we start off with one tensor with a symmetry, and then attach it to another symbol.

```
A_{m n p}::TableauSymmetry(shape={1,1}, indices={1,2});
p = TableauSymmetry.get($A_{m n p}$)
p.attach($B_{m n p}$)
ex:= B_{m n p} - B_{m p n};
canonicalise(_);
```

Property TableauSymmetry attached to A_{mnp}.

$B_{mnp} - B_{mpn}$

$2B_{mnp}$

2.4 Indices, dummy indices and automatic index renaming

In Cadabra, all objects which occur as subscripts or superscripts are considered to be "indices". The names of indices are understood to be irrelevant when they occur in a pair, and automatic relabelling will take place whenever necessary in order to avoid index clashes.

Cadabra knows about the differences between free and dummy indices. It checks the input for consistency and displays a warning when the index structure does not make sense. Thus, the input

```
ex:= A_{m n} + B_{m} = 0;
```

will results in an error message. The location of indices is, by default, not considered to be relevant. That is, you can write

```
{m, n}::Indices(name="free");
ex:=A_{m} + A^{m};
```

Attached property Indices(position=free) to $[m,\ n]$.

$A_m + A^m$

as input and these are considered to be consistent expressions. You can collect such terms by using `lower_free_indices` or `raise_free_indices`,

```
lower_free_indices(ex);
```

$2A_m$

If, however, the position of an index means something (like in general relativity, where index lowering and raising implies contraction with a metric), then you can declare index positions to be "fixed". This is done using

```
{a,b,c,d,e,f}::Indices(name="fixed", position=fixed);
```

Attached property Indices(position=fixed) to $[a,\ b,\ c,\ d,\ e,\ f]$.

Cadabra will raise or lower indices on such expressions to a canonical form when the `canonicalise` algorithm is used,

```
ex:= G_{a b} F^{a b} + G^{a b} F_{a b};
canonicalise(_);
```

$G_{ab}F^{ab} + G^{ab}F_{ab}$

$2G^{ab}F_{ab}$

If upper and lower indices should remain untouched at all times, there is a third index position type, called 'independent',

```
{q,r,s}::Indices(name="independent", position=independent);
ex:= G_{q r} F^{q r} + G^{q r} F_{q r};
canonicalise(_);
```

Attached property Indices(position=independent) to $[q,\ r,\ s]$.

$G_{qr}F^{qr} + G^{qr}F_{qr}$

$G_{qr}F^{qr} + G^{qr}F_{qr}$

As the last line shows, the index positions have remained unchanged. When substituting an expression into another one, dummy indices will automatically be relabelled when necessary. To see this in action, consider the following example:

```
ex:= G_{a b} Q;
rl:= Q-> F_{a b} F^{a b};
substitute(ex, rl);
```

$G_{ab}Q$

$Q \to F_{ab}F^{ab}$

$G_{ab}F_{cd}F^{cd}$

The a and b indices have automatically been relabelled to c and d in order to avoid a conflict with the free indices on the G_{ab} object. You may have noticed that when you write `T_{a b}` the '`a b`' in the subscript is not interpreted as a product, but rather as two different indices to the tensor T.

2.5 Implicit versus explicit indices

When writing expressions which involves vectors, spinors and matrices, one often employs an implicit notation in which some or all of the indices are suppressed. Examples are

$$a = Mb\,, \quad \overline{\psi}\gamma^m\chi\,,$$

where a and b are vectors, ψ and χ are spinors and M and γ^m are matrices. Clearly, the computer cannot know this without some further information. In Cadabra objects can carry implicit indices, through the `ImplicitIndex` property. There are derived forms of this, e.g. `Matrix` and `Spinor`. The following example shows how implicit indices ensure that objects do not get moved through each other when sorting expressions.

```
{a,b}::ImplicitIndex;
M::Matrix;
ex:= a = M b;
sort_product(_);
```

Attached property ImplicitIndex to $[a,\ b]$.

Attached property Matrix to M.

$a = Mb$

$a = Mb$

If you had not made the property assignment in the first two lines, the `sort_product` would have incorrectly swapped the matrix and vector, leading to a meaningless expression.

If you have more than one set of implicit indices, it is best to use a form of `ImplicitIndex` which makes explicit which indices are suppressed. In the following example, we write consider the expression Mcb in which M is a matrix acting on the vector b, while c is a

different matrix which does not act on the same vector space. In other words, we consider $M^i{}_j c^m{}_n b^j$. Clearly we can also write this as Mbc, which is indeed what `sort_product` converts it to.

```
{i,j}::Indices(vector);
{m,n}::Indices(spinor);
M::ImplicitIndex(M^{i}_{j});
b::ImplicitIndex(b^{i});
c::ImplicitIndex(c^{m}_{n});
```

Attached property Indices(position=free) to $[i,\ j]$.

Attached property Indices(position=free) to $[m,\ n]$.

Attached property ImplicitIndex to M.

Attached property ImplicitIndex to b.

Attached property ImplicitIndex to c.

```
ex:= M c b;
```

Mcb

```
sort_product(_);
```

Mbc

Such explicit property information is also respected by operators like `Trace`. The following example shows how to remove objects from traces when they do not carry any indices on which the trace acts.

```
Tr{#}::Trace(indices=vector);
ex:= Tr( M c M );
untrace(_);
```

Attached property Trace to $Tr\,(\#)$.

$Tr\,(McM)$

$cTr\,(MM)$

2.5.1 Converting between implicit and explicit

It is possible to convert from implicit indices to explicit indices, that is, make Cadabra write out all implicit indices explicitly. For this to work you need to have declared an `ImplicitIndex` property which lists the explicit indices of the object. Cadabra will then take care of creating index lines.

```
{i,j,k}::Indices;
a::ImplicitIndex(a^{i});
```

```
M::ImplicitIndex(M^{i}_{j});
ex:= M a;
```

Attached property Indices(position=free) to $[i,\ j,\ k]$.

Attached property ImplicitIndex to a.

Attached property ImplicitIndex to M.

Ma

```
explicit_indices(ex);
```

$M^{i}{}_{j}a^{j}$

Note how dummy indices were introduced automatically.

3 Mathematical properties

3.1 Derivatives and implicit dependence on coordinates

There is no fixed notation for derivatives; as with all other objects you have to declare derivatives by associating a property to them, in this case the `Derivative` property.

```
\nabla{#}::Derivative;
```

Attached property Derivative to $\nabla\#$.

Derivative objects can be used in various ways. You can just write the derivative symbol, as in

```
ex:=\nabla{ A_{\mu} };
```

∇A_{μ}

Or you can write the coordinate with respect to which the derivative is taken,

```
s::Coordinate;
A_{\mu}::Depends(s);
ex:=\nabla_{s}{ A_{\mu} };
```

Attached property Coordinate to s.

Attached property Depends to A_{μ}.

$\nabla_{s} A_{\mu}$

Finally, you can use an index as the subscript argument, as in

```
{ \mu, \nu }::Indices(vector);
ex:=\nabla_{\nu}{ A_{\mu} };
```

Attached property Indices(position=free) to $[\mu,\ \nu]$.

$\nabla_\nu A_\mu$

(in which case the first line is, for the purpose of using the derivative operator, actually unnecessary). The main point of associating the `Derivative` property to an object is to make the object obey the Leibnitz or product rule, as illustrated by the following example,

```
\nabla{#}::Derivative;
ex:= \nabla{ A_{\mu} * B_{\nu} };
product_rule(_);
```

Attached property Derivative to $\nabla\#$.

$\nabla\left(A_\mu B_\nu\right)$

$\nabla A_\mu B_\nu + A_\mu \nabla B_\nu$

This behaviour is a consequence of the fact that `Derivative` derives from `Distributable`. Note that the `Derivative` property does not automatically give you commuting derivatives, so that you can e.g. use it to write covariant derivatives. More specific derivative types exist too. An example are partial derivatives, declared using the `PartialDerivative` property. Partial derivatives are commuting and therefore automatically symmetric in their indices,

```
\partial{#}::PartialDerivative;
{a,b,m,n}::Indices(vector);
C_{m n}::Symmetric;
ex:=T^{b a} \partial_{a b}( C_{m n} D_{n m} );
```

Attached property PartialDerivative to $\partial\#$.

Attached property Indices(position=free) to $[a,\ b,\ m,\ n]$.

Attached property Symmetric to C_{mn}.

$T^{ba}\partial_{ab}\left(C_{mn}D_{nm}\right)$

```
canonicalise(_);
```

$T^{ab}\partial_{ab}\left(C_{mn}D_{mn}\right)$

4 Manipulating expressions

4.1 Selecting parts of expressions

In many other computer algebra systems, you can select parts of results using the mouse, paste them into a new input cell, and then continue the computation. Naively this sounds like a nice feature to have, and it is indeed quite useful for quick computations. However, for larger projects, this feature quickly becomes a major source of trouble. Once you use the cut-n-paste technique, you are no longer able to make any changes in cells before the one with pasted content. Or rather, you can make changes, but they will not automatically propagate to into the pasted cell. Any effect of the change at the top of the notebook will have to be evaluated until the point of the cut-n-paste, and then you have to do the cut-n-paste again by hand.

Now this is fine if you just do a quick computation, as you will probably know precisely what you want to cut-n-paste. But if you give your notebook to someone else, this may no longer be clear. Worse, if you do not look at your notebook for some time, and then return after a few months (or years), you will most likely have forgotten completely what was the logic for the particular cut.

For this reason, Cadabra does not support cut-n-paste of output. But that does not mean that you cannot select parts of expressions for subsequent computation. For that, Cadabra has a more systematic logic, which is built around the `zoom` and `unzoom` commands.

4.1.1 Zooming into an expression

If you have a large expression, and want to select a part of it for further manipulation, while temporarily ignoring the rest, use the `zoom` command. It takes an expression and a

pattern, and then suppresses all terms in the expression which do not match the pattern. An example:

```
ex:= \int{ c A + c**2 B + c D + c**2 A }{x};
```

$$\int \left(cA + c^2B + cD + c^2A\right)\ \mathrm{d}x$$

```
zoom(_, $c Q??$);
```

$$\int \left(cA + \ldots + cD + \ldots\right)\ \mathrm{d}x$$

This has selected all terms with a single factor of c, and suppressed the other ones (but keeping a reminder that those terms are still there, in the form of the dots). You can now work on the visible terms as usual, e.g. doing a substitution,

```
substitute(_, $A -> E$);
```

$$\int \left(cE + \ldots + cD + \ldots\right)\ \mathrm{d}x$$

In order to get back to the full expression, use `unzoom`,

```
unzoom(_);
```

$$\int \left(cE + c^2B + cD + c^2A\right)\ \mathrm{d}x$$

As you can see, the substitution has only changed the terms which were visible at the time.

4.2 Using multiple files and notebooks

At some point, you will encounter computations which are best separated out into their own notebook. Or you will do a computation which takes a long time, and you want to write an intermediate result into a file so that you can read it back later easily. There are two options for this in Cadabra: importing notebooks into other notebooks, or writing individual expressions to a file and reading them back.

4.2.1 Importing a notebook into another one

The simplest way to separate functionality is to simply write a separate notebook with the properties and expressions which you want to re-use elsewhere. In this way, writing a

'package' for Cadabra is nothing else but writing a separate notebook. You can import any notebook into another one by using the standard Python import logic.

Example

Let us say we have a notebook `library.cnb`, which contains a single cell with the following content:

```
{m,n,p,q,r}::Indices;
ex:=A_{q r} A_{q r};
```

You can now import this into another notebook by simply using

```
from library import *
```

Cadabra looks for the `library.cnb` notebook in your `PYTHONPATH` (just as in ordinary Python programs), as well as in the current directory. You can see that this worked by e.g. the following:

```
ex;
rename_dummies(ex);
```

$$A_{qr}A_{qr}$$

$$A_{mn}A_{mn}$$

Note that the import has thus not only imported the `ex` expression, but also the property information about the index set, which enabled the `rename_dummies` to work. Behind the scenes, what happens is that the import statement looks for a file `library.cnb`. If it finds this, it will first convert that file to a proper Python file (remember the `library.cnb` file is a Cadabra notebook, not a Python file). It then uses the standard Python logic to do the import.

4.2.2 Writing expressions to a file and reading them back

A somewhat more difficult way to re-use expressions is to write them to a file using standard Python methods, and then read them back elsewhere. This method is best used for long computations of which you want to write an intermediate result out to disk, to be read in later (instead of doing a re-computation). Be aware that if you write an expression to disk, you do not write the property information of any of the symbols in that expression to disk.

Example

The following example declares two expressions and writes them to disk. It then reads the expressions back in again and assigns them to different Python names.

```
ex1:= A_{m n} \sin{x};
ex2:= B_{m n};
with open("output.cdb", "w") as file:
   file.write( ex1.input_form()+"\n" )
   file.write( ex2.input_form()+"\n" )
```

$A_{mn} \sin x$

B_{mn}

```
with open("output.cdb", "r") as file:
   ex3=Ex( file.readline() )
   ex4=Ex( file.readline() )
```

```
ex3;
ex4;
```

$A_{mn} \sin x$

B_{mn}

Note that when written in this way, the file `output.cdb` only contains the expressions, not their names (`ex1` and `ex2` in the example above). Cadabra's expressions can also be written to disk using Python's pickle functionality. This makes the code slightly less messy, but note that the file will no longer be human-readable. If you use the pickle module, the example above would read:

```
import pickle

ex1:= A_{m n} \sin{x};
ex2:= B_{m n};
with open("output.pkl", "wb") as file:
   pickle.dump(ex1, file)
   pickle.dump(ex2, file)
```

$A_{mn} \sin x$

B_{mn}

```
with open("output.pkl", "rb") as file:
   ex3=pickle.load(file)
   ex4=pickle.load(file)
```

```
ex3;
ex4;
```

$A_{mn} \sin x$

B_{mn}

4.3 Default simplification

By default, Cadabra does very few things "by itself" with your expressions. It only collects equal terms, but even that can be turned off if you want to. The logic is that all simplification steps are contained in a function `post_process`, which is executed on every new input and on every completion of an algorithm. It can contain arbitrary code, but by default it reads

```
def post_process(ex):
    collect_terms(ex)
```

which simply collects equal terms. You can for instance apply a substitution on every input automatically,

```
def post_process(ex):
    distribute(ex)
    substitute(ex, $A_{m n} -> B_{m q} B_{q n}$)
    collect_terms(ex)
```

```
{m,n,p,q}::Indices(vector);
A_{m n}::Symmetric;
ex:=A_{m n} ( A_{n m} + C_{n m});
```

Attached property Indices(position=free) to $[m,\ n,\ p,\ q]$.

Attached property Symmetric to $B_{mq}B_{qn}$.

$B_{mq}B_{qn}B_{np}B_{pm} + B_{mq}B_{qn}C_{nm}$

As usual dummy indices have been relabelled appropriately. The `post_process` function can be redefined on-the-fly in the middle of a notebook.

4.4 Patterns, conditionals and regular expressions

Patterns in Cadabra are quite a bit different from those in other computer algebra systems, because they are more tuned towards the pattern matching of objects common in tensorial expressions, rather than generic tree structures. Cadabra knows about three different pattern types: *name patterns* (for single names), *object patterns* (for things which include indices and arguments) and *dummy patterns* (things for which the name is irrelevant, like indices).

Name patterns are things which match a single name in an object, without indices or arguments. They are constructed by writing a single question mark behind the name, as in

```
ex:= Q + R;
substitute(_, $A? + B? -> 0$);
```

$$Q + R$$

$$0$$

which matches all sums with two terms, each of which is a single symbol without indices or arguments. If you want to match instead any object, with or without indices or arguments, use the double question mark instead. To see the difference more explicitly, compare the two substitute commands in the following example

```
ex:=A_{m n} + B_{m n};
substitute(_, $A? + B? -> 0$ );
substitute(_, $A?? + B?? -> 0$ );
```

$$A_{mn} + B_{mn}$$

$$A_{mn} + B_{mn}$$

$$0$$

Note that it does not make sense to add arguments or indices to object patterns; a construction of the type `A??_{\mu}(x)` is meaningless and will be flagged as an error.

There is a special handling of objects which are dummy objects. Objects of this type do not need the question mark, as their explicit name is never relevant. You can therefore write

```
ex:= A_{m n};
substitute(_, $A_{p q}->0$);
```

$$A_{mn}$$

$$0$$

to set all occurrances of the tensor A with two subscript indices to zero, regardless of the names of the indices (as you can see, this command sets A_{pq} to zero). When index sets are declared using the `Indices` property, these will be taken into account when matching. When replacing object wildcards with something else that involves these objects, use the question mark notation also on the right-hand side of the rule. For instance,

```
ex:= C + D + E + F;
substitute(_, $A? + B? -> A? A?$, repeat=True);
```

$$C + D + E + F$$

$$CC + EE$$

replaces every consecutive two terms in a sum by the square of the first term. The following example shows the difference between the presence or absence of question marks on the right-hand side:

```
ex:= C + D;
substitute(_, $A? + B? -> A?$);
```

C

```
ex:= C + D;
substitute(_, $A? + B? -> A A$);
```

AA

So be aware that the full pattern symbol needs to be used on the right-hand side (in contrast to many other computer algebra systems).

Note that you can also use patterns to remove or add indices or arguments, as in

```
{\mu, \nu, \rho, \sigma}::Indices(vector);
ex:= A_{\mu} B_{\nu} C_{\nu} D_{\mu};
substitute(_, $A?_{\rho} B?_{\rho} -> \dot{A?}{B?}$, repeat=True);
```

Attached property Indices(position=free) to $[\mu,\ \nu,\ \rho,\ \sigma]$.

$A_{\mu} B_{\nu} C_{\nu} D_{\mu}$

$A \cdot DB \cdot C$

4.4.1 Conditionals

In many algorithms, patterns can be supplemented by so-called conditionals. These are constraints on the objects that appear in the pattern. In the example below, the substitution is not carried out, as the rule applies only to patterns where the n and p indices are not equal,

```
ex:= A_{m n} B_{n q};
substitute(_, $ A_{m n} B_{p q} | n != p -> 0$);
```

$A_{mn} B_{nq}$

$A_{mn} B_{nq}$

$A_{mn} B_{nq}$

Without the conditional, the substitution does apply,

```
ex:= A_{m n} B_{n q};
substitute(_, $ A_{m n} B_{p q} -> 0$);
```

$A_{mn} B_{nq}$

0

Note that the conditional follows directly after the pattern, not after the full substitution rule. A way to think about this is that the conditional is part of the pattern, not of the rule. The reason why the conditional follows the full pattern, and not directly the symbol to which it relates, is clear from the example above: the conditional is a "global" constraint on the pattern, not a local one on a single index.

These conditions can be used to match names of objects using regular expressions. In the following example, the pattern `M?` will match against `C7`,

```
ex:= A + B3 + C7;
substitute(_, $A + M? + N? | \regex{M?}{"[A-Z]7"} -> \sin(M? N?)/N?$);
```

$$A + B3 + C7$$

$$\sin\left(C7B3\right) B3^{-1}$$

Without the condition, the first match of `M?` would be against `B3`,

```
ex:= A + B3 + C7;
substitute(_, $A + M? + N? -> \sin(M? N?)/N?$);
```

$$A + B3 + C7$$

$$\sin\left(B3C7\right) C7^{-1}$$

5 Writing your own packages

5.1 Programming in Cadabra

Cadabra is fully programmable in Python. At the most basic level this means that you can make functions which combine various Cadabra algorithms together, or write loops which repeat certain Cadabra algorithms. At a more advanced level, you can inspect the expression tree and manipulate individual subexpressions, or construct expressions from elementary building blocks.

5.1.1 Fundamental Cadabra objects: Ex and ExNode

The two fundamental Cadabra objects are the `Ex` and the `ExNode`. An object of type `Ex` represents a mathematical expression, and is what is generated if you type a line containing `:=`, as in

```
ex:=A+B;
type(ex);
```

$$A + B$$

```
<class 'cadabra2.Ex'>
```

An object of type `ExNode` is best thought of as an iterator. It can be used to walk an expression tree, and modify it in place (which is somewhat different from normal Python iterators; a point we will return to shortly). The most trivial way to get an iterator is to call the `top` member of an `Ex` object; think of this as returning a pointer to the topmost node of an expression,

```
ex.top();
type(ex.top());
```

```
A + B
```

```
<class 'cadabra2.ExNode'>
```

You will also encounter `ExNodes` when you do a standard Python iteration over the elements of an `Ex`, as in

```
for n in ex:
   type(n);
   display(n)
```

```
<class 'cadabra2.ExNode'>
```

```
A + B
```

```
<class 'cadabra2.ExNode'>
```

```
A
```

```
<class 'cadabra2.ExNode'>
```

```
B
```

As you can see, this 'iterates' over the elements of the expression, but in a perhaps somewhat unexpected way. We will discuss this in more detail in the next section. Important to remember from the example above is that the 'pointers' to the individual elements of the expression are `ExNode` objects. There are various other ways to obtain such pointers, using various types of 'filtering', more on that below as well.

Once you have an `ExNode` pointing to a subexpression in an expression, you can query it further for details about that subexpression.

```
ex:= A_{m n};
for i in ex.top().free_indices():
   display(i)
```

$$A_{mn}$$

```
m
```

```
n
```

The example above shows how, starting from an iterator which points to the top of the expression, you can get a new iterator which can iterate over all free indices.

5.1.2 ExNode and Python iterators

Before we continue, we should make a comment on how `ExNode` objects relate to Python iterators. For many purposes, `ExNode` objects behave as you expect from Python iterators: they allow you to loop over nodes of an `Ex` expression, you can call `next(...)` on them,

and so on. However, there are some slight differences, which have to do with the fact that Cadabra wants to give you access to the nodes of the original Ex, so that you can modify this original Ex in place. Consider for instance this example with a Python list of integers, with standard iterators:

```
q=[1,2,3,4,5];
for element in q:
   element=0
q;
```

[1, 2, 3, 4, 5] [1, 2, 3, 4, 5] It still produces the original list at the end of the day, because each element is a *copy* of the element in the list. With ExNodes you can actually modify the original Ex, as this example shows:

```
ex:=A + B + C + D;
for element in ex.top().terms():
   element.replace($Q$)
ex;
```

$$A + B + C + D$$

$$Q + Q + Q + Q$$

In this case, element is not an Ex corresponding to each of the 5 terms, but rather an ExNode, which is more like a pointer into the Ex object. The replace member function allows you to replace the building blocks of the original ex expression.

If you want to get a proper Ex object (so a *copy* of the element in the expression over which you are iterating), more like what you would get if iteration over Cadabra's expressions was an ordinary Python iteration, then you can use ExNode.ex():

```
ex:= A + 2 B + 3 C + 4 D;
lst=[]
for element in ex.top().terms():
   lst.append( element.ex() )
ex;
lst[2];
```

$$A + 2B + 3C + 4D$$

$$A + 2B + 3C + 4D$$

$$3C$$

Here the list lst contains copies of the individual terms of the ex expression.

A good way to remember about this is to keep in mind that Cadabra tries its best to allow you to modify expressions *in-place*. The ExNode iterators provide that functionality.

5.1.3 Traversing the expression tree

The `ExNode` iterator can be instructed to traverse expressions in various ways. The most basic iterator is obtained by using standard Python iteration with a `for` loop,

```
ex:= A + B + C_{m} D^{m};
```

$A + B + C_m D^m$

```
for n in ex:
   print(str(n))
```

```
\begin{verbatim}A + B + C_{m} D^{m}
A
B
C_{m} D^{m}
C_{m}
m
D^{m}
m
\end{verbatim}
```

The iterator obtained in this way traverses the expression tree node by node, and when you ask it to print what it is pointing to, it prints the entire subtree of the node it is currently visiting. If you are only interested in the name of the node, not the entire expression below it, you can use the `.name` member of the iterator:

```
for n in ex:
   print(str(n.name))
```

```
\begin{verbatim}\sum
A
B
\prod
C
m
D
m
\end{verbatim}
```

Often, this kind of 'brute force' iteration over expression elements is not very useful. A more powerful iterator is obtained by asking for all nodes in the subtree which have a certain name. This can be the name of a tensor, or the name of a special node, such as a product or sum,

```
for n in ex["C"]:
   display(n)
```

`C_{m}`

```
for n in ex["\\prod"]:
   display(n)
```

```
C_{m} D^{m}
```

The above two examples used an iterator obtained directly from an `Ex` object. Various ways of obtaining iterators over special nodes can be obtained by using member functions of `ExNode` objects themselves. So one often uses a construction in which one first asks for an iterator to the top of an expression, and then requests from that iterator a new one which can iterate over various special nodes. The example below obtains an iterator over all top-level terms in an expression, and then loops over its values.

```
for n in ex.top().terms():
   display(n)
```

```
A
```

```
B
```

```
C_{m} D^{m}
```

Two special types of iterators are those which iterate only over all arguments or only over all indices of a sub-expression. These are discussed in the next section.

5.1.4 Arguments and indices

There are various ways to obtain iterators which iterate over all arguments or all indices of an expression. The following example, with a derivative acting on a product, prints the argument of the derivative as well as all free indices.

```
\nabla{#}::Derivative;
ex:= \nabla_{m}{ A^{n}_{p} V^{p} };
```

Attached property Derivative to $\nabla\#$.

$\nabla_{m}\left(A^{n}{}_{p}V^{p}\right)$

```
for nabla in ex[r'\nabla']:
   for arg in nabla.args():
      print(str(arg))
   for i in nabla.free_indices():
      print(str(i))
```

```
\begin{verbatim}A^{n}_{p} V^{p}
m
n
\end{verbatim}
```

5.1.5 Example: covariant derivatives

The following example shows how you might implement the expansion of a covariant derivative into partial derivatives and connection terms.

```
def expand_nabla(ex):
    for nabla in ex[r'\nabla']:
        nabla.name=r'\partial'
        dindex = nabla.indices().__next__()
        for arg in nabla.args():
            ret:=0;
            for index in arg.free_indices():
                t2:= @(arg);
                if index.parent_rel==sub:
                    t1:= -\Gamma^{p}_{@(dindex) @(index)};
                    t2[index]:= _{p};
                else:
                    t1:= \Gamma^{@(index)}_{@(dindex) p};
                    t2[index]:= ^{p};
                ret += Ex(str(nabla.multiplier)) * t1 * t2
            nabla += ret
    return ex
```

The sample expressions below show how this automatically takes care of not introducing connections for dummy indices, and how it automatically handles indices which are more complicated than single symbols.

```
\nabla{#}::Derivative;
ex:= 1/2 \nabla_{a}{ h^{b}_{c} };
expand_nabla(ex);
```

Attached property Derivative to $\nabla\#$.

$$\frac{1}{2}\nabla_{a}h^{b}{}_{c}$$

$$\frac{1}{2}\partial_{a}\left(h^{b}{}_{c}\right)+\frac{1}{2}\Gamma^{b}{}_{ap}h^{p}{}_{c}-\frac{1}{2}\Gamma^{p}{}_{ac}h^{b}{}_{p}$$

```
ex:= 1/4 \nabla_{a}{ v_{b} w^{b} };
expand_nabla(ex);
```

$$\frac{1}{4}\nabla_{a}\left(v_{b}w^{b}\right)$$

$$\frac{1}{4}\partial_{a}\left(v_{b}w^{b}\right)$$

```
ex:= \nabla_{\hat{a}}{ h_{b c} v^{c} };
expand_nabla(ex);
```

$$\nabla_{\widehat{a}} \left(h_{bc} v^{c} \right)$$

$$\partial_{\widehat{a}} \left(h_{bc} v^{c} \right) - \Gamma^{p}{}_{\widehat{a}b} h_{pc} v^{c}$$

6
Algorithms

distribute

Distribute factors over sums.

Rewrite a product of sums as a sum of products, as in

$$a\,(b+c) \to a\,b + a\,c\,.$$

This would read

```
ex:=a (b+c);
distribute(_);
```

$a\,(b+c)$

$ab + ac$

The algorithm in fact works on all objects which carry the `Distributable` property,

```
Op{#}::Distributable;
ex:=Op(A+B);
distribute(_);
```

Attached property Distributable to $Op\,(\#)\,.$

$Op\,(A+B)$

$Op\,(A) + Op\,(B)$

The primary example of a property which inherits the `Distributable` property is `PartialDerivative`. The `distribute` algorithm thus also automatically writes out partial derivatives of sums as sums of partial derivatives,

```
\partial{#}::PartialDerivative;
ex:=\partial_{m}{A + B + C};
distribute(_);
```

Attached property PartialDerivative to $\partial\#$.

$$\partial_m \left(A + B + C \right)$$

$$\partial_m A + \partial_m B + \partial_m C$$

6
Bibliography

[1] Kasper Peeters, Pierre Vanhove, and Anders Westerberg. “Supersymmetric higher-derivative actions in ten and eleven dimensions, the associated superalgebras and their formulation in superspace”. In: *Class. Quant. Grav* 18 (2001), pp. 843–889. eprint: hep-th/0010167.

[2] Kasper Peeters and Anders Westerberg. “The Ramond-Ramond sector of string theory beyond leading order”. In: *Class. Quant. Grav.* 21 (2004), pp. 1643–1666. eprint: hep-th/0307298.

[3] Kasper Peeters. “Introducing Cadabra: a symbolic computer algebra system for field theory problems”. In: (2007). eprint: hep-th/0701238.

[4] Kasper Peeters. “Cadabra2: computer algebra for field theory revisited”. In: *J. Open Source Softw.* 3.32 (2018), p. 1118. DOI: 10.21105/joss.01118.