Writing Algorithms in Cadabra2

Dominic Price

Cadabra is a powerful computer algebra system which is specialised for handling tensor field calculations and is built on top of Python. By using Python as a foundation, it is very easy to extend the functionality of Cadabra with custom algorithms to perform specific tasks as well as very generic routines which can use the incredibly flexible and reflective language features of Python to dynamically respond to different inputs.

This is a brief introduction to writing algorithms in Cadabra, introducing some of the typical idioms which allow user algorithms to behave like the inbuilt routines as well as the various language features and technical considerations of writing algorithms.

The first section is a purely Python centred guide and assumes a basic familiarity with both Cadabra and Python with any more advanced language features being explained and external resources to provide more details linked. In the second section, the infrastructure of the underlying C++ codebase is explained and directions on how to extend Cadabra using C++ is covered. This is a slightly more advanced discussion although it is aimed to be accessible for casual users of C++.

1 What is an algorithm in Cadabra?

Before diving into the ins and outs of *how* to write an algorithm for Cadabra, it is first worth discussing exactly what specifically we are talking about as the term 'algorithm' is quite general. For the purposes of this tutorial, when talking about an algorithm we are specifically talking about a function which satisfies some criteria:

1. The number of argument it takes can vary, but its first argument must be a `Ex` object
2. It should apply transformations on this expression
3. It then returns a reference to this expression

The functions in the Cadabra core all satisfy this format, and most of the functions found in the `cdb.*` libraries are too although occasionally they will instead create a copy of the input expression instead of modifying it. An example of an algorithm which acts like this is `get_component` from the `cdb.core.component` library which resolves the free indices of an expression into specified components.

While it is of course possible to write functions which modify expression and which *don't* follow these criteria, if you are planning to release your algorithm to the wider community or even are collaborating on a project then by making sure that you follow these requirements your code will be far more intuitive for others to follow and will be able to be used alongside the inbuilt Cadabra algorithms seamlessly. To illustrate, the following function

```
def distribute_excluding(subex, ex):
    substitute(ex, $@(subex) -> qqq$)
    distribute(ex)
    substitute(ex, $qqq -> @(subex)$)
    return True
```

which takes the 'acting' expression as a second parameter and returns a `bool` is almost certain to be a common source of bugs and look out of place in a list of algorithm calls:

```
...
unwrap(ex)
substitute(ex, $\partial_{\mu}{p} -> 0$)
distribute_excluding($t - s$, ex)
...
```

and also prevents users from immediately viewing the results of the algorithm:

```
ex := (x + y)*(a + b)*(c + d)
substitute(ex, $a + b -> 1$); # Displays $(x+y)(c+d)$
distribute_excluding($x + y$, ex); # Displays 'True'
```

2 Writing Cadabra algorithms in Python

Having covered what an algorithm is, we can now start looking at ways to implement them. Using pure Python, there are two basic constructions which can be used for interfacing an algorithm: the simplest method is by using a Python function definition which satisfies the criteria given above. The second method is by using the special `@algo` decorator defined in `cdb.util.develop` which allows *tree-traversal* algorithms to be easily created.

2.1 Iterating through the expression tree

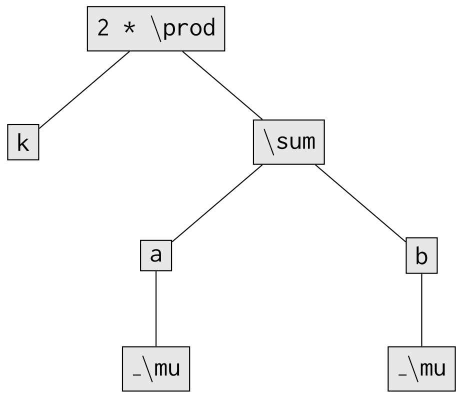
Often, we may want to write a routine which does something completely different to what the inbuilt functions provide and will require a lower-level look at the expression. In order to handle this, Cadabra provides methods for iterating over the *expression tree*, which is the internal representation of the expression which Cadabra keeps. You can get a basic visualisation of this tree, for example typing

```
ex := 2k * (a_{\mu} + b_{\mu}).
tree(ex);
```

will show you the output

```
{\prod}  2  (4 000001E82DF51F60)
  1:{k}  (4 000001E82DF510B0)
  2:{\sum}  (4 000001E82DF51560)
  3:  {a}  (4 000001E82DF51650)
  4:    _{\mu}  (4 000001E82DF51790)
  5:  {b}  (4 000001E82DF518D0)
  6:    _{\mu}  (4 000001E82DF51920)
```

The `tree` command is mainly for debugging purposes and so isn't the most friendly of layouts — the name of the node is shown inside brackets (usually curly, but this reflects the bracket type of the node) and is optionally preceded by a symbol (usually _ or ^) reflecting its relation to the parent node, this is followed by a number representing the rational multiplier of the node if it is non-unitary, and then in brackets is some extra information about the pointer location. Nodes representing operators are not denoted by symbols but are given names, thus in this diagram we can see `*` $\longrightarrow$ \prod and + $\longrightarrow$ \sum. A list of these special node names is given in Appendix A. Lets look at a diagrammatic representation to make it clearer what is being shown:



If the relation between this and the expression $2k(a_{\mu} + b_{\mu})$ is unclear then there is a good explanation of how to build up these trees on the Mathematics StackExchange[1] and the Wikipedia article for binary trees[2] although it should be remembered that operator nodes in the Cadabra expression tree are not limited to two children.

[1]https://math.stackexchange.com/a/369807
[2]https://en.wikipedia.org/wiki/Binary_expression_tree

Access to this tree in Python allows arbitrary manipulation of expressions. The tree can be traversed in numerous ways, with ExNode objects providing an interface to the nodes in the tree. These both expose various properties of the tree as well as methods which allow the insertion, modification and removal of nodes: these methods are enumerated in Appendix B.

There are three main ways of obtaining references to nodes in the tree:

1. **`Ex.top()`**

   The node representing the top node of a tree can be obtained by using the top method of a Ex object. From this, the tree can be further explored by using one of the ExNode methods which returns an iterator, for example

   ```
   for node in ex.top().children():
       print(node.name)
   ```

   which for the expression above prints

   ```
   k
   \sum
   ```

2. **Name search**

   It is possible to iterate over nodes based on their name as in the following example:

   ```
   ex := v_{\mu} v^{\mu} \gamma_{\rho \lambda} + T_{\rho \lambda};
   for node in ex[r'\mu']:
       node.name = r'\nu'
   ```

   where the expression now reads $v_\nu v^\nu \gamma_{\rho\lambda} + T_{\rho\lambda}$

3. **Whole tree traversal**

   The simplest traversal method is to visit the whole tree using a for loop:

   ```
   for node in ex:
       print(node.name)
   ```

   For the expression above this prints

   ```
   \prod
   k
   \sum
   a
   \mu
   b
   \mu
   ```

   This is known as *preorder* iteration, i.e. each node is returned before its children are visited.

2.2 Accessing property information

In many cases, in order for a algorithm to work correctly we need to be able to query subexpressions to see which properties they have assigned. This can be done using the .get(ex) method of a property which returns the corresponding property object if it assigned to the expression or None otherwise. In the following example this lets us check whether various objects are defined as coordinates:

```
{t, x, y, z}::Coordinate.
Coordinate.get($t$);
Coordinate.get($\phi$);
for node in $x, r$.top().children():
    Coordinate.get(node);
```

```
Property Coordinate attached to t.
None
Property Coordinate attached to x
None
```

Note that some properties, such as Coordinate, can be dependent upon their parent relationship, for example the t in v_t is still a coordinate but the following returns None:

```
ex := v_{t}.
for node in ex['t']:
    Coordinate.get(node);
```

In this instance, we can pass the optional argument ignore_parent_rel=True to allow Cadabra to make the match:

```
for node in ex['t']:
    Coordinate.get(node, ignore_parent_rel=True);
```

Many property object have extra methods which provide extra information, usually these are from attributes passed when the property was declared. In the following example we can use this to see which values various indices run over:

```
{i, j, k, l}::Indices(cartesian, values={x,y,z}).
{\mu, \nu, \rho, \lambda}::Indices(polar, values={r, \phi, \theta}).
ex := x^{i}x^{i} + p_{\mu}p^{\mu}.
for node in ex:
    prop = Indices.get(node)
    if prop is not None:
        print(node.name, prop.values)
```

```
i [x, y, z]
i [x, y, z]
\mu [r, \phi, \theta]
\mu [r, \phi, \theta]
```

To get a list of which extra methods a property defines you can use dir or help to list all the methods it defines.

2.3 Algorithms using def

For almost all situations you are likely to come across, a simple definition of an algorithm using a Python function will be sufficient. We already saw a bad example of this in the previous section with distribute_excluding; lets rewrite that now to make it more conventional:

```
def distribute_excluding(ex, subex):
    substitute(ex, $@(subex) -> qqq$)
    distribute(ex)
    substitute(ex, $qqq -> @(subex)$)
    return ex
```

This is a simple routine which allows us to limit the effects of distribute on an expression which can be useful if there are some common factors containing sums which we want to preserve, for example in Schwarzchild space-time where factors of $2M - r$ are common. Of course, this is a slightly naïve way of writing this (it will cause unexpected behaviour in the event someone uses qqq as a variable name and is limited to only allowing one expression to be excluded) but illustrates a common technique for writing algorithms quite nicely: more complicated manipulations can often be produced simply by combining the inbuilt algorithms like this to suit specific needs.

2.4 Algorithms using @algo

A common technique for writing algorithms is to traverse the expression tree checking each node against a criteria and applying a function to it if the criteria is satisfied. The entire collection of core algorithms is built in this manner and the pattern is implemented in the Algorithm class. This is a abstract base class with two pure virtual methods: can_apply which implements the predicate checking if an operation can be applied to a tree node, and apply which then carries out the actual operation.

The class itself is, however, usually not directly visible to users — instead its behaviour is wrapped inside of a function which takes care of iterating through the tree, calling can_apply and apply, and ensuring that the tree is in a normalised state after the operation. In Python this wrapping is implemented with the @algo decorator which automatically converts the class into an appropriate function call. A very simple example is the following x_to_y algorithm. We first need to import the relevant objects from the cdb.utils.develop library:

```
from cdb.utils.develop import algo, Algorithm
```

We then define a decorated class which inherits from `Algorithm` and overrides the two virtual functions:

```
@algo
class x_to_y(Algorithm):
    def can_apply(self, node):
        return node.name == 'x'
    def apply(self, node):
        node.name = 'y'
        return result_t.changed
```

The predicate in this instance simply checks the name of the node, but in all cases it should return a boolean type. Any node for which the result is `False` is skipped and so the apply method can make assumptions about node it receives as a parameter based on this. In this case, it knows that it can change the name to `y`. The return value of the apply method is of type `result_t` which should take on the value `changed` if some transformation was applied, `error` if some error occurred inside the method or `unchanged` if apply ended up making no modifications to the tree. apply *must* return a `result_t` or an exception will be raised when the algorithm is called that the function signature is incorrect.

The `@algo` decorator converts this into a function so we call this algorithm as any other algorithm in Cadabra:

```
x_to_y($x + y$)
```

```
2y
```

Notice that various cleanup routines are automatically called after the algorithm is applied which simplifies the expression from $y + y$ into $2y$.

The function is also given the optional keyword arguments `deep`, `repeat` and `depth` with the same behaviour as the core algorithms.

Stateful algorithms

Sometimes it is necessary for an algorithm to be stateful, that is either be passed extra information through parameters when it is called or to retain some information between various calls to `apply`. This can be done by overriding the `__init__` method of the class; any arguments passed to the function call (other than `deep`, `repeat` and `depth`) are forwarded onto this constructor. Let's use this to extend our `x_to_y` algorithm into one where we can specify other values for `x` and `y`:

```
@algo
class x_to_y(Algorithm):
    def __init__(self, ex, x='x', y='y'):
        Algorithm.__init__(self, ex)
        self.x = x
        self.y = y
    def can_apply(self, node):
        return node.name == self.x
    def apply(self, node):
        node.name = self.y
        return result_t.changed
```

We can now use this as a 'y-to-x' algorithm too:

```
x_to_y($x + y$, x='y', y='x');
```

```
2x
```

Its important to ensure that the first argument to the `__init__` method is the `Ex` object on which the algorithm is called, and to remember to call the base `Algorithm` constructor too.

Functional shortcut

For simple algorithms such as our `x_to_y` example here, instead of defining an entire class we can also just supply `@algo` with a function and it will automatically generate the class definition using the function body for the `apply` method:

```python
@algo
def x_to_y(node):
    if node.name == 'x':
        node.name = 'y'
        return result_t.changed
    return result_t.unchanged
```

Note that it is not possible to create a stateful algorithm using this method, the definition using a class as shown above must be used if this is desired. Anoher drawback to this approach is that the predicate is now inside the body of apply which results in slightly less efficient code being produced, but this is negligible for simple algorithms such as this for which the shorthand is intended. If this does become a concern, it is also possible to supply algo with a verb—pred— parameter which it will use as the body for can_apply eliminating the need to perform this check inside the function body:

```python
@algo(pred=lambda node: node.name == 'x')
def x_to_y(node):
    node.name = 'y'
    return result_t.changed
```

This produces code which is identical to the first class definition we discussed above.

2.5 Documenting algorithms

As notebooks can be imported from other notebooks using Python's `import` statement, many Cadabra packages are written in notebooks which allows the documentation to be provided in accompanying LaTeX cells. In order to format this documentation the `algorithm` command is provided which takes two arguments: the signature of the algorithm's function call and a brief description of its use and application.

The signature should contain the types of all the arguments and the return value using Python's type hinting style, for example a function which takes a Ex object and a int and returns a Ex might be documented like

```latex
\algorithm{my_algo(ex: Ex, n: int) -> Ex}{Does some transformation to \texttt{ex} which
    involves \texttt{n}}
```

`my_algo(ex: Ex, n: int) -> Ex`

Does some transformation to `ex` *which involves* `n`

3 Writing Cadabra algorithms in C++

Although one of the greatest strengths of Cadabra is how easy it is to extend using Python, efficiency can sometimes cause issues either on very large expressions or when algorithms reach a certain level of complexity and are hindered by the Python interpreted runtime. In these instances, writing the algorithm in C++ is almost certainly the necessary solution. It is not a common occurrence for users to contribute C++ algorithms and so no specific functionality has been developed in order to facilitate the writing of external libraries and so it is necessary to build Cadabra from source and develop the algorithm inside the core codebase, however Cadabra always welcomes community contributions and if the algorithm proves to be useful then it may well be included in a future release of Cadabra.

There are two ways of placing a C++ algorithm into Cadabra: either by extending the list of core algorithms, or by placing the code in a library package. In either instance, the first step is to get the Cadabra code base from github [3] and follow the build instructions for your OS. The next optional (but recommended especially if you are planning of submitting a pull request) is to set up a new git branch to perform your development on.

3.1 Writing a core algorithm

As with the `@algo` style of algorithm definition in Python, the core algorithms in C++ all derive from the C++ `Algorithm` class. First, you should create a header and source file where the code for you algorithm

[3] `https://github.com/kpeeters/cadabra2`

will live. As with the previous example, we will create the simple x_to_y algorithm and so we create the files core/algorithms/x_to_y.hh and core/algorithms/x_to_.cc. We also need to tell CMake about the new source file by adding a line to the ALGORITHM_SRC_FILES variable in core/CMakeLists.txt:

```
SET(ALGORITHM_SRC_FILES
   algorithms/canonicalise.cc
   algorithms/collect_components.cc
   algorithms/collect_factors.cc
   ...
   algorithms/unwrap.cc
   algorithms/unzoom.cc
   algorithms/untrace.cc
   algorithms/x_to_y.cc
   algorithms/vary.cc
   algorithms/young_project.cc
   algorithms/young_project_product.cc
)
```

Next we need to define the x_to_y class in the header file core/algorithms/x_to_y.hh we just created:

```
#pragma once
#include <Algorithm.hh>

namespace cadabra {
    class x_to_y : public Algorithm {
        public:
            x_to_y(const Kernel& kernel, Ex& ex);

            bool can_apply(Ex::iterator it) override;
            result_t apply(Ex::iterator& it) override;
        };
    }
```

This should be fairly understandable, the signatures of all the class methods must be the same as in this example but you may add more parameters to the constructor after the Ex reference (as we will do later). We can now move on to the implementation which goes into core/algorithms/x_to_y.cc. We start by importing the header and for convenience pulling cadabra into the global namespace:

```
#include "algorithms/x_to_y.hh"

using namespace cadabra;
```

For the simple algorithm we are writing, we will simply forward the constructor arguments onto the Algorithm constructor:

```
x_to_y::x_to_y(const Kernel& kernel, Ex& ex)
    : Algorithm(kernel, ex)
    {

    }
```

Next we write in our implementation of can_apply. This is a very straightforward function, the only thing to keep in mind is that node names in Cadabra are interned and so to access the std::string representation of them they must be dereferenced:

```
bool x_to_y::can_apply(Ex::iterator it)
    {
    return *it->name == "x";
    }
```

The last bit of code we need to write in this file is the implementation of apply. Again, as the strings are interned instead of assigning a string we must query the global name_set for an iterator and store the value this returns to us:

```
Algorithm::result_t x_to_y::apply(Ex::iterator& it)
    {
    it->name = name_set.insert("y").first;
    return result_t::l_applied;
    }
```

It is also worth mentioning that the `result_t` names differ between the Python and C++ interfaces; whereas before we used changed and unchanged in C++ we use `l_applied` and `l_no_action` respectively.

The final step we need to perform is the Python interfacing code in `core/pythoncdb/py_algorithms.cc`. We first add our header file to the list of includes at the top:

```
...
#include "../algorithms/unwrap.hh"
#include "../algorithms/unzoom.hh"
#include "../algorithms/untrace.hh"
#include "../algorithms/x_to_y.hh"
#include "../algorithms/vary.hh"
#include "../algorithms/young_project.hh"
#include "../algorithms/young_project_product.hh"
#include "../algorithms/young_project_tensor.hh"
...
```

The final step before building is then the actual binding code. This is the least intuitive part of the process, but most of the technical details are hidden and all that needs to be done is to call `def_algo` templated against the `x_to_y` class. You can add this line to the end of the list of `def_algo` declarations in the `init_algorithms` function:

```
void init_algorithms(py::module& m)
    {
    pybind11::enum_<Algorithm::result_t>(m, "result_t")
    .value("checkpointed", Algorithm::result_t::l_checkpointed)
    ...
    def_algo_preorder<meld, bool>(m, "meld", true, false, 0, py::arg("project_as_sum") =
        false);
    def_algo<x_to_y>(m, "x_to_y", true, false, 0);
    }
```

The first argument is a handle to the cadabra2 Python module and the second is the name of the function on the Python side, this should be the same as the C++ name to avoid confusion. The next three parameters are the default values of the deep, repeat and depth parameters but these can be changed in Python when calling the algorithm by using the optional keyword arguments.

It's now time to build Cadabra again and try out the new algorithm. The algorithm will be imported into the global scope at launch:

```
x_to_y($x + y$);
```

```
2y
```

Adding extra constructor parameters

As with our `x_to_y` function from the Python section, a natural extension would be to allow users to provide values of x and y. This only requires a few alterations to the code we have written: first the constructor must be altered to take the new parameters and the class definition extended to include the two new variables x and y. We will store these as interned strings and so we declare them as `nset_t::iterator` variables. The new class definition in `x_to_y.hh` now looks as follows

```
class x_to_y : public Algorithm {
    public:
        x_to_y(const Kernel& kernel, Ex& ex, const std::string& x, const std::string& y);

        bool can_apply(Ex::iterator it) override;
        result_t apply(Ex::iterator& it) override;

    private:
        nset_t::iterator x, y;
    };
```

Moving onto `x_to_y.cc`, we now also change the constructor signature here and initialise x and y:

```
x_to_y::x_to_y(const Kernel& kernel, Ex& ex, const std::string& x, const std::string& y)
    : Algorithm(kernel, ex)
    , x(name_set.insert(x).first)
    , y(name_set.insert(y).first)
    {

    }
```

The can_apply and apply function definitions also need to use the stored values instead of the hardcoded values of x and y:

```
bool x_to_y::can_apply(Ex::iterator it)
{
    return it->name == x;
}

Algorithm::result_t x_to_y::apply(Ex::iterator& it)
{
    it->name = y;
    return result_t::l_applied;
}
```

Almost done: just one more line to change, the Python binding code. In order to make Python aware that the function now takes two extra arguments, we need to add the types of these these extra arguments to the list of template parameters of `def_algo`. We also need to give these extra arguments Python names, which is done by adding a `py::arg` object parameter onto the end of the list of `def_algo`'s parameters for each argument. The final binding code should look like this:

```
def_algo<x_to_y, const std::string&, const std::string&>(
    m, "x_to_y", true, false, 0,
    py::arg("x"), py::arg("y"));
```

After building, this is now accessible from Cadabra:

```
x_to_y($x + y$, 'y', 'x')
```

```
2x
```

3.2 Writing a package algorithm

Recently, additions to the binding code have allowed the possibility of creating Cadabra packages in C++. There is currently only support in the build system for creating libraries for the builtin cdb.* libraries. This can be used to refactor algorithms contained in these libraries which are currently implemented in Python into C++ routines, or to provide 'non-core' functionality in an easily accessible location without polluting the global namespace.

It is recommended that the C++ library is prefixed with an underscore, and the relevant functions, classes or other definitions which it provides imported into a Cadabra notebook with the documentation attached there. Therefore, if we wanted to create a `cdb.core.subs` module for our `x_to_y` algorithm, we would create the C++ module `cdb.core._subs` and then have a notebook `cdb.core.subs` with the contents

```
\algorithm{x_to_y(ex: Ex) -> Ex}{Renames all nodes names x to y}
```

```
from cdb.core._subs import x_to_y
```

To create the package, we first need to make a file `core/packages/cdb/core/_subs.cc`. To make the build system aware of the package, we need to add it to the `COMPILED_PACKAGES` variable in the build script located at `core/packages/CMakeLists.txt`:

```
set(COMPILED_PACKAGES
  core/_component.cc
  utils/_algorithm.cc
  core/_subs.cc
  )
```

The contents of the file `_subs.cc` is very similar to the code used for implementing a core algorithm. We begin by importing the functionality from the `py_algorithms.hh` header which will also give us access to the `Algorithm` class, and we will then bring it into scope by importing all the names from the cadabra namespace:

```
#include "pythoncdb/py_algorithms.cc"

using namespace cadabra;
```

We can then implement the x_to_y class as before, here we use inline function definitions for brevity:

```
class x_to_y : public Algorithm {
    public:
        x_to_y(const Kernel& kernel, Ex& ex)
            : Algorithm(kernel, ex)
            {

            }

        bool can_apply(Ex::iterator it) override
            {
            return *it->name == "x";
            }

        result_t apply(Ex::iterator& it) override
            {
            it->name = name_set.insert("y").first;
            }
    };
```

Finally we create the Python module itself using the PYBIND11_MODULE macro. The first argument must match the filename as this will be the name of the module, the second argument can be any valid identifier. We can then call def_algo just as before:

```
PYBIND11_MODULE(_subs, m)
{
    def_algo<x_to_y>(m, "x_to_y", true, false, 0);
}
```

After building, this will now be available to import into Cadabra. Assuming we have also created the subs notebook as described above, we can now call the x_to_y from there:

```
x + y
```

```
2y
```

Extending this to the alternate form of x_to_y with extra arguments is done exactly as for the case when writing a core algorithm was discussed.

Circling back to the very beginning of this tutorial, it is of course also possible to create a function which does this by accepting and returning a Ex_ptr (a convenience typedef for the Ex pointer type exposed to Python):

```
Ex_ptr x_to_y(Ex_ptr ex)
    {
    for (Ex::iterator beg = ex->begin(), end = ex->end(); beg != end; ++beg) {
        if (*beg->name == "x")
            beg->name = name_set.insert("y").first;
        }
    return ex;
    }
```

with the binding code

```
m.def("x_to_y", x_to_y);
```

A Reserved node names

More information on the behaviour of these nodes can be found in the Cadabra documentation.

Symbol	Node name	Brief description
$\times$	\prod	Product of children
$+$	\sum	Sum of children
$\oplus$	\oplus	Tensor sum of children
$\frac{x}{y}$	\frac	Quotient of children (internal use only)
$,$	\comma	List of children as independent expressions
$\rightarrow$	\arrow	Substitution rule from first child to second
$\cdot$	\inner	Inner product of children
x^y	\pow	First child to power of second child
$\int$	\int	Integral of first child w.r.t second child
$=$	\equals	First child equals second child
$\neq$	\unequals	First child does not equal second child
$[a,b]$	\commutator	Commutator between two children
$\{a,b\}$	\anticommutator	Anticommutator between two children
$\begin{cases} \Box_t = \cdots \\ \cdots \end{cases}$	\components	Component values of free indices
$\wedge$	\wedge	Wedge product of children
x with y	\conditional	First child as pattern which holds under second child
$<$	\less	First child is less than second child
$>$	\greater	First child is greater than second child
$(A+B+C)_\mu$	\indexbracket	First child is an object with indices given by the remaining children
$\cdots$	\ldots	Suppressed terms

B ExNode methods and properties

B.1 Properties

name The name of the current node

parent_rel The relationship of the current node to its parent

multiplier The rational multiplier attached to the current node

B.2 Methods

args() Return an iterator over all arguments of the current node

append_child(other) Append a Ex or ExNode as a child of the current node

children() Return an iterator over all children of the current node

compare(other, use_props, ignore_parent_rel Compare to another ExNode. Returns a match_t object.

erase() Erase the current node

ex() Return a copy of the current node as a Ex object

factors() Return an iterator over all factors at the level of the current node

free_indices() Return an iterator over all free indices of the current node

indices() Return an iterator over all indices of the current node

insert(other) Insert a Ex or ExNode object in front of the current node

own_indices() Return an iterator over all indices of the current node which are not inherited from child nodes

replace(ex) Replace the current node with a Ex object

terms() Return an iterator over all terms at the level of the current node