rewalt

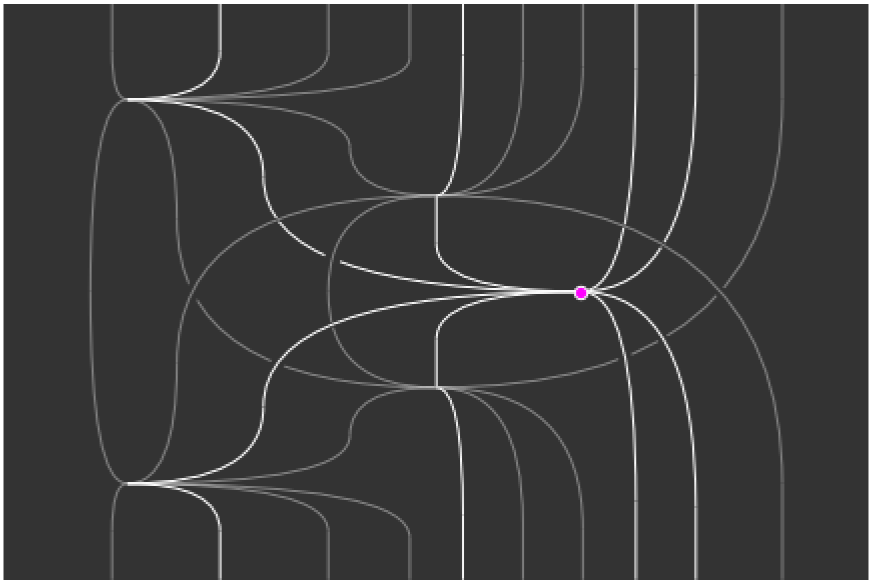
Release 0.1.0

Amar Hadzihasanovic

Sep 15, 2022

NOTEBOOKS

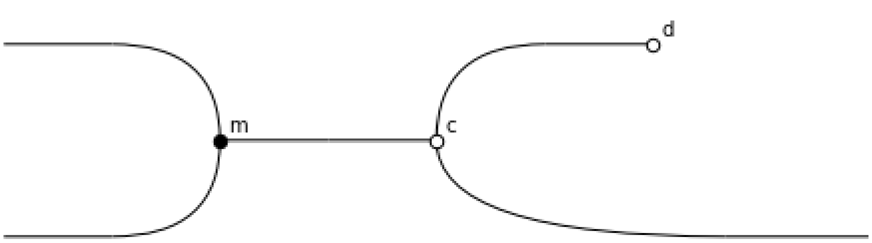
1. *(archaic)* to overturn, throw down
2. a library for **rew**riting, **al**gebra, and **t**opology, developed in Tallinn (aka **Reval**)



`rewalt` is a toolkit for **higher-dimensional diagram rewriting**, with applications in

- **higher** and **monoidal category theory**,
- **homotopical algebra**,
- **combinatorial topology**,

and more. Thanks to its visualisation features, it can also be used as a structure-aware **string diagram** editor, supporting TikZ output so the string diagrams can be directly embedded in your LaTeX files.



It implements diagrammatic sets which, by the "higher-dimensional rewriting" paradigm, double as a model of

- *higher-dimensional rewrite systems*, and of

- *directed cell complexes*.

This model is "topologically sound": a diagrammatic set built in `rewalt` presents a finite CW complex, and a diagram constructed in the diagrammatic set presents a valid homotopy in this CW complex.

A diagrammatic set can be seen as a generalisation of a *simplicial set* or of a *cubical set* with many more "cell shapes". As a result, `rewalt` also contains a *full implementation* of finitely presented **simplicial sets** and **cubical sets with connections**.

CHAPTER

ONE

INSTALLATION

`rewalt` is available for Python 3.7 and higher. You can install it with the command

```
pip install rewalt
```

If you want the bleeding edge, you can check out the GitHub repository.

CHAPTER TWO

GETTING STARTED

To get started, we recommend you check the Notebooks, which contain a number of worked examples from category theory, algebra, and homotopy theory.

CHAPTER

THREE

FURTHER READING

For a first introduction to the ideas of higher-dimensional rewriting, diagrammatic sets, and "topological soundness", you may want to watch these presentations at the CIRM meeting on Higher Structures and at the GETCO 2022 conference.

A nice overview of the general landscape of higher-dimensional rewriting is Yves Guiraud's mémoire d'habilitation.

So far there are two papers on the theory of diagrammatic sets: the first one containing the foundations, the second one containing some developments applied to categorical universal algebra.

A description and complexity analysis of some of the data structures and algorithms behind `rewalt` will be published in the proceedings of ACT 2022.

CHAPTER

FOUR

LICENSE

`rewalt` is distributed under the BSD 3-clause license.

CHAPTER

FIVE

CONTRIBUTING

Currently, the only active developer of `rewalt` is Amar Hadzihasanovic.

Contributions are welcome. Please reach out either by sending me an email, or by opening an issue.

5.1 The theory of monoids

In this notebook, we will construct a presentation of the *theory of monoids* or *associative algebras* in `rewalt`. Depending on your favourite gadget, you may see this as the data presenting a monoidal category (PRO) or an operad.

5.1.1 Adding the sorts and operations

Let's first import `rewalt` and create an empty diagrammatic set — an object of class `DiagSet` — that we will call `Mon`.

[1]:
```
import rewalt

Mon = rewalt.DiagSet()
```

You know how a monoidal category can be seen as a one-object bicategory (its delooping)? This is how we do it in `rewalt` too: the sorts of a monoidal theory are *1-cells* going to and from a single 0-cell.

So first of all, we add a single 0-dimensional generator to our diagrammatic set.

[2]:
```
pt = Mon.add('pt')
```

This adds a 0-dimensional generator to `Mon`, assigns it the name `'pt'` and returns the `Diagram` object that "picks" that generator only; we assign this diagram to the variable `pt`.

Next, we add a single 1-dimensional generator, corresponding to the single sort of our theory.

[3]:
```
a = Mon.add('a', pt, pt)
```

The two extra arguments that we gave to `add` specify the *input*, or *source boundary* of the new generator, and the *output*, or *target boundary* of the new generator, respectively. In this case they are both equal to the unique "point".

By the way, if you fail to assign the output of `add` to a variable, you can always retrieve it later by giving the generator's name to `Mon`'s indexer.

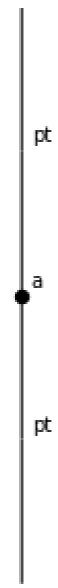
[4]:
```
assert a == Mon['a']
```

There is not much that we can do with 0-cells… but with 1-cells, we can create larger diagrams by *pasting*.

The `paste` method pastes together diagrams along the k-dimensional output boundary of one and the k-dimensional input boundary of the other, when these match each other.

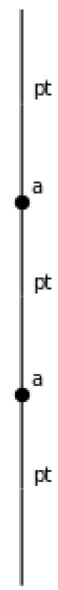
For a 1-cell, the only non-trivial boundary is the 0-dimensional one; pasting along it corresponds to "concatenation of paths". We can concatenate `a` to itself as many times as we want. Let's also visualise the result as a "1-dimensional string diagram".

[5]:
```
a.draw()
```



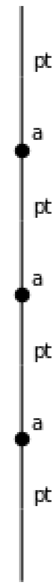
nbsphinx-code-borderwhite

[6]:
```
a.paste(a).draw()
```



nbsphinx-code-borderwhite

[7]:
```
a.paste(a).paste(a).draw()
```



nbsphinx-code-borderwhite

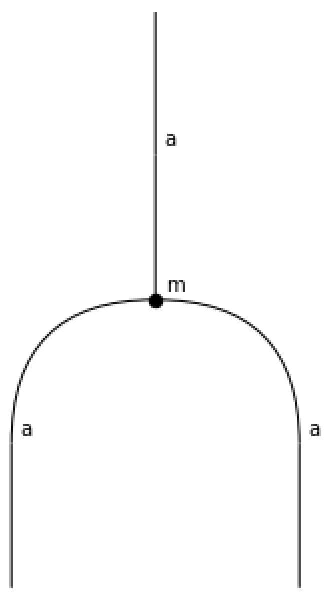
And so on. Note that `paste` can also take an integer argument specifying the dimension of the boundary along which to paste; it defaults to the *minimum of the two diagrams' dimensions, minus 1*. In this case the minimum of 1 and 1 is 1, which minus 1 equals 0, and that's the boundary we want.

Now that we have the sorts, let's add the operations. The *monoid multiplication* takes two inputs and returns one output. This corresponds to a 2-dimensional generator, whose input is `a.paste(a)`, and output `a`.

[8]:
```
m = Mon.add('m', a.paste(a), a)
```

And let's picture this as a string diagram.

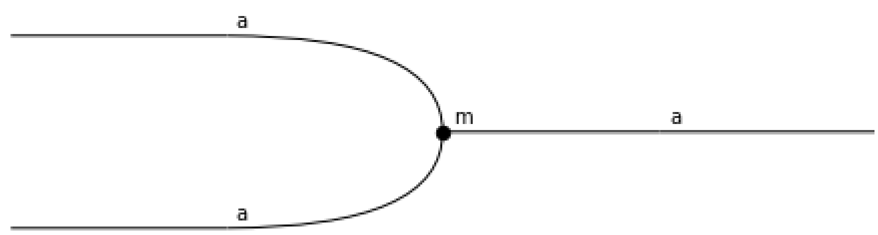
[9]:
```
m.draw()
```



nbsphinx-code-borderwhite

(As you can see, string diagrams by default go from bottom to top. If you prefer left-to-right, or top-to-bottom, or right-to-left orientation, you can pass it as an argument to `draw`; or to change the default setting, reassign `rewalt.strdiags.DEFAULT['orientation']`.)

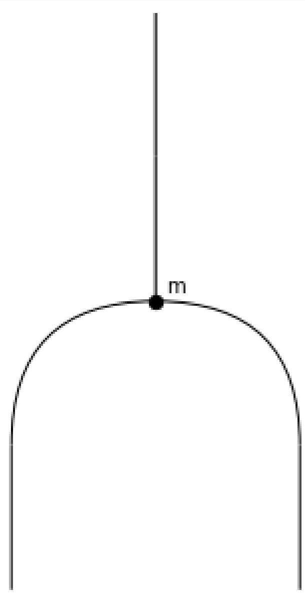
[10]:
```
m.draw(orientation='lr')
```



nbsphinx-code-borderwhite

Since we have a single sort, it is a little pointless to label the wires. Same for labelling the unique point. Let's switch labels off for these generators.

[11]:
```
Mon.update('a', draw_label=False)
Mon.update('pt', draw_label=False)
m.draw()
```



nbsphinx-code-borderwhite

Next, we want to add the monoid unit, which is a "nullary" operation. Here things get a little more subtle.

Cells in `rewalt` are not allowed to have "strictly lower-dimensional" inputs or outputs: if we try to add a 2-dimensional generator whose input is a 0-dimensional diagram, we will get an error.

[12]:
```
try:
```

(continues on next page)

(continued from previous page)

```
    u = Mon.add('u', pt, a)
except ValueError:
    print('Nope')
```

```
Nope
```

Instead, we have to use "weak units", in the form of *degenerate diagrams*. (This may seem like a hassle in dimension 2, where "everything can be strictified", but pays off in higher dimensions.)

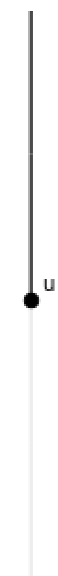
A simple constructor for degenerate diagrams is the `unit` method, which creates a "unit diagram", one dimension higher.

[13]:
```
assert pt.dim == 0
assert not pt.isdegenerate

assert pt.unit().dim == 1
assert pt.unit().isdegenerate
```

So to add the monoid unit, we make `pt.unit()` its input.

In string diagrams, degenerate cells are represented as translucent wires (when wires), or as "node-less nodes" (when nodes).

[14]:
```
u = Mon.add('u', pt.unit(), a)
u.draw()
```

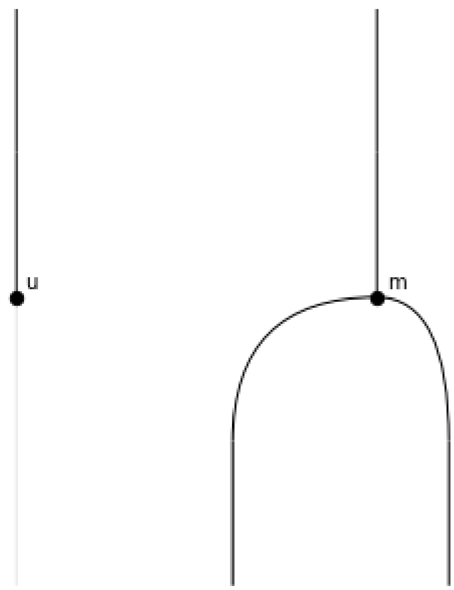


nbsphinx-code-borderwhite

5.1.2 Adding "oriented equations"

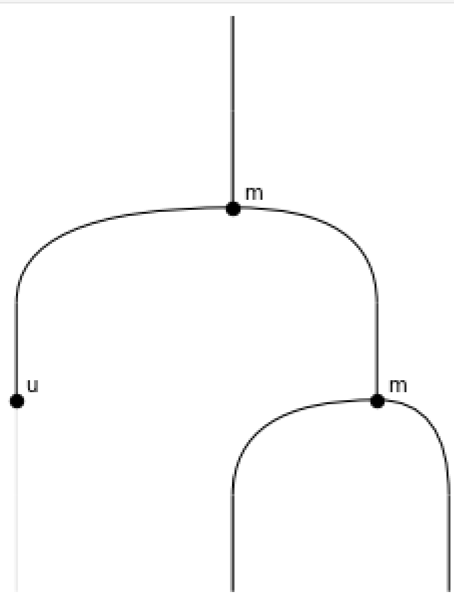
Now we can compose diagrams with `paste` in two directions, along the 0-boundary ("horizontally") or the 1-boundary ("vertically")...

[15]:
```
u.paste(m, 0).draw()  # "horizontal" pasting
```



nbsphinx-code-borderwhite

[16]:
```
u.paste(m, 0).paste(m).draw()  # ...and now "vertical" pasting
```

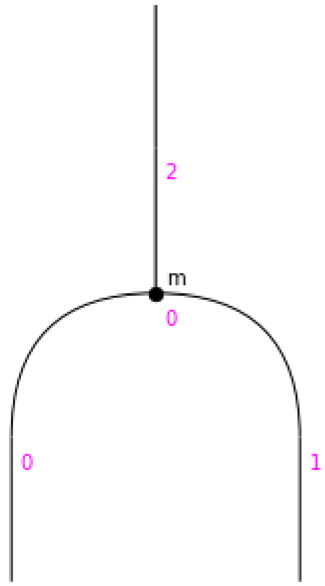


nbsphinx-code-borderwhite

A useful alternative to `paste` (especially in an "operadic" setting) are the methods `to_inputs` and `to_outputs`, which allow us to paste a diagram only to *some* inputs and outputs of another diagram.

To use these in practice, one must know that every node and wire in a string diagram have a unique *position*. We can use the keyword arguments `positions` (both nodes and wires), `nodepositions`, and `wirepositions` to enable positions in string diagram output.

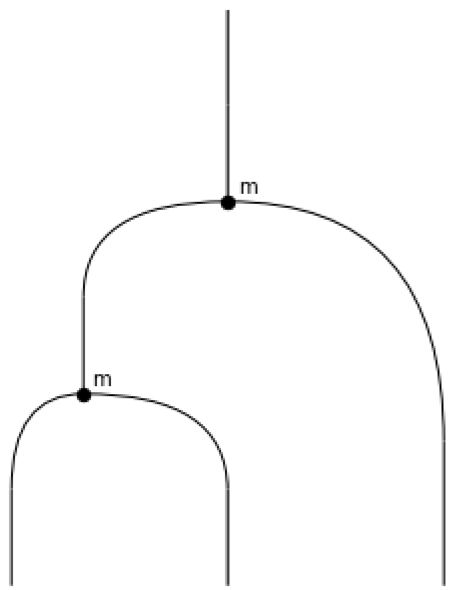
[17]:
```
m.draw(positions=True)
```



nbsphinx-code-borderwhite

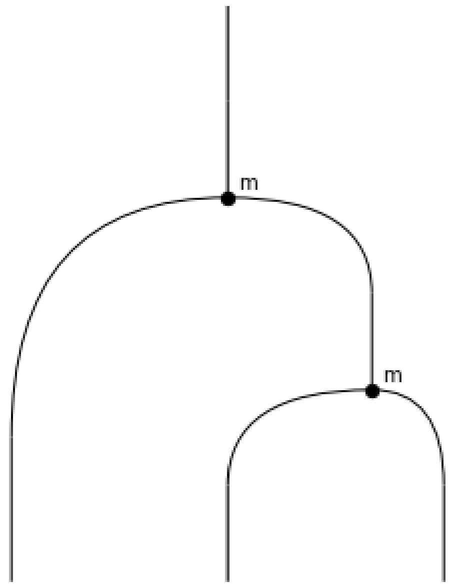
Now, we can paste another multiplication either to the input in position 0, or the input in position 1.

[18]:
```
m.to_inputs(0, m).draw()
```



nbsphinx-code-borderwhite

[19]:
```
m.to_inputs(1, m).draw()
```

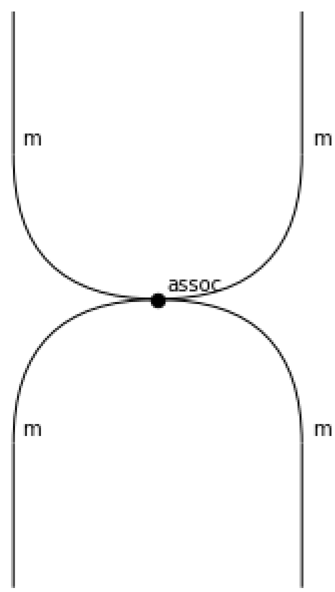


nbsphinx-code-borderwhite

These two diagrams happen to be the two sides of the *associativity* equation, so let's add this equation to our presentation!

Or rather, we add an *oriented* associativity equation, or associativity *rewrite*, or "associator", as a *3-dimensional generator*. All the cells in diagrammatic sets have a direction.

[20]:
```
assoc = Mon.add('assoc', m.to_inputs(0, m), m.to_inputs(1, m))
assoc.draw()
```



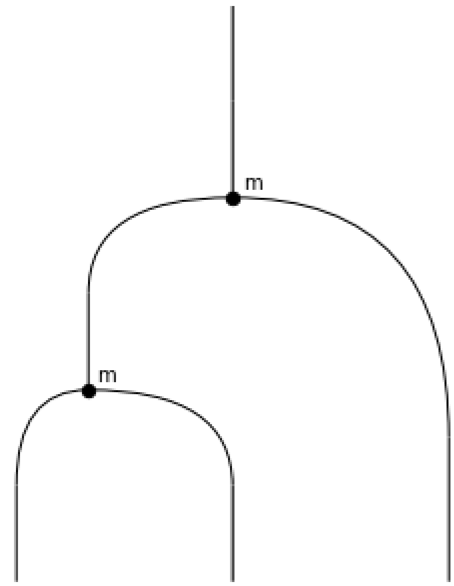
nbsphinx-code-borderwhite

You can see that, when we draw a 3-dimensional diagram, we obtain a "2-dimensional slice" string diagram, where nodes correspond to 3-cells and wires to 2-cells. (In general, for an n-dimensional diagram, nodes are n-dimensional cells and wires are (n-1)-dimensional cells).

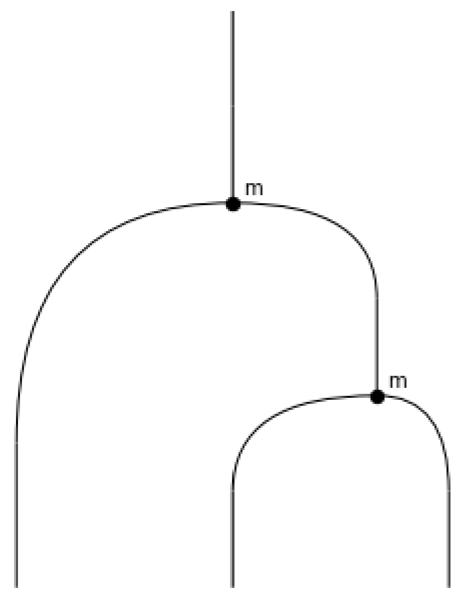
Here, `assoc` is a 3-dimensional cell that has two `m` 2-cells in its input, and two `m` 2-cells in its output.

To see the two "sides" of the rewrite, we can either use the `draw_boundaries` method, or first call `input`/`output` and only then `draw`.

[21]:
```
assoc.draw_boundaries()
```



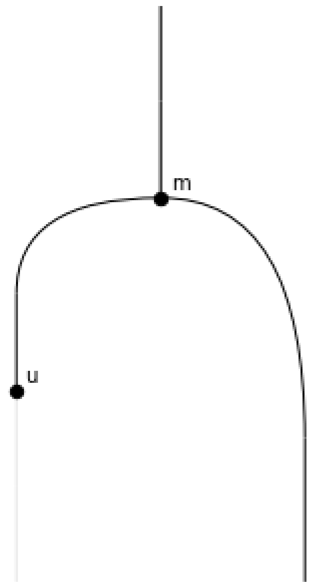
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

Next, let's add left unitality and right unitality equations/rewrites. The left-hand side of the left unitality equation is this.

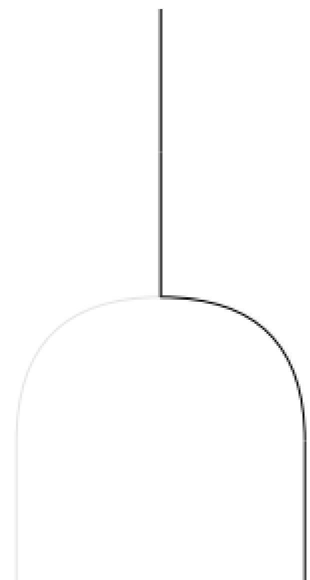
[22]:
```
m.to_inputs(0, u).draw()
```



nbsphinx-code-borderwhite

This diagram is supposed to be equal to "the identity operation" on our sort (which would be the unit on `a`)... but not quite, because it contains a weak unit in the input; instead we want to equate to another degenerate cell called the *left unitor* on `a`. We build it like this.

[23]:
```
a.lunitor('-').draw()
```

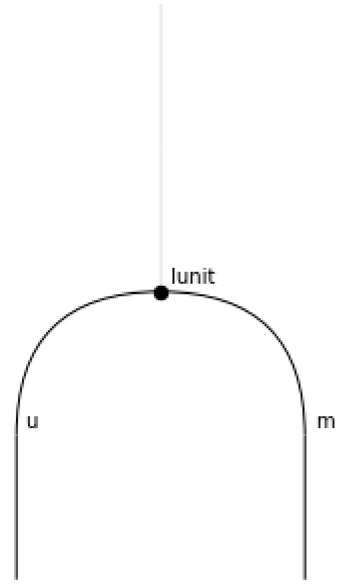


nbsphinx-code-borderwhite

The argument `'-'` specifies that the unit should appear in the input, and not the output.

Now we can add the "left unitality" generator.

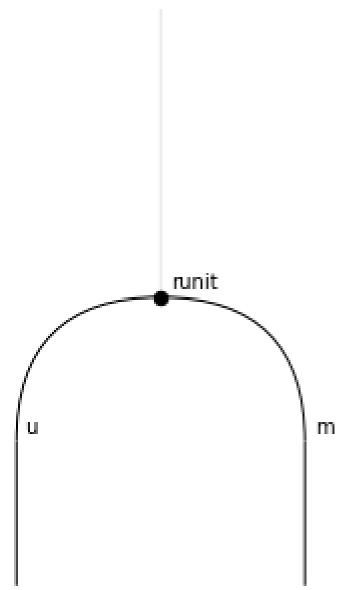
[24]:
```
lunit = Mon.add('lunit', m.to_inputs(0, u), a.lunitor('-'))
lunit.draw()
```



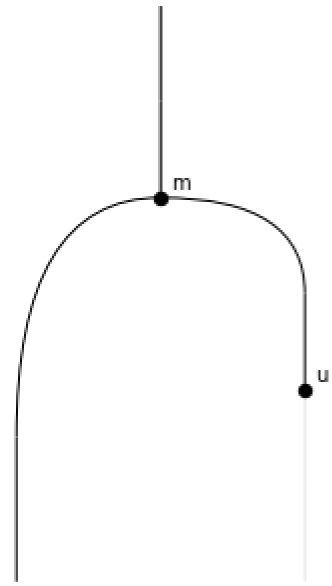
nbsphinx-code-borderwhite

We proceed similarly for the “right unitality” generator.

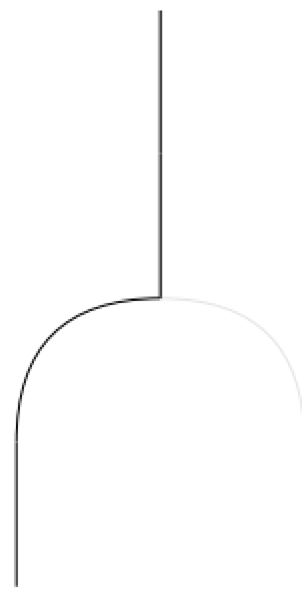
[25]:
```
runit = Mon.add('runit', m.to_inputs(1, u), a.runitor('-'))
runit.draw()
runit.draw_boundaries()
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

5.1.3 Making the equations go both ways

That’s it, we now have a presentation of the theory of monoids!

Except our “equations” are really directed *rewrites*. What if we want to use them in both directions? Luckily, we have methods for “weakly inverting” a generator. Let’s try it on `assoc`.

[26]:
```
Mon.invert('assoc')
```

[26]:
```
(<rewalt.diagrams.Diagram at 0x7f72f7faf100>,
 <rewalt.diagrams.Diagram at 0x7f72f7faeb60>,
 <rewalt.diagrams.Diagram at 0x7f72f843f250>)
```

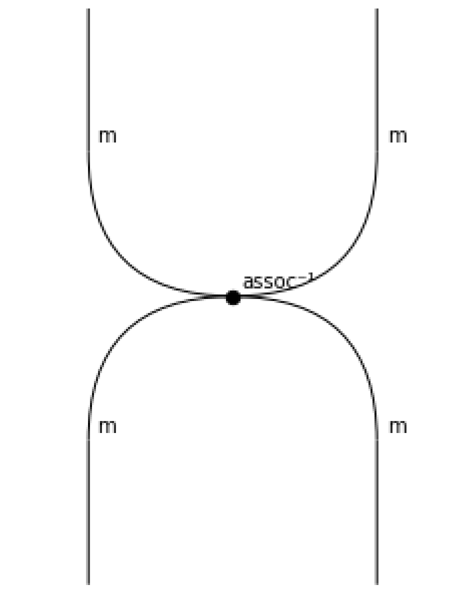
This returned 3 diagrams, which corresponds to the fact that 3 new generators were added. Let’s see what happened. We can see a list of the generators, ordered by dimension, with the `DiagSet` method `by_dim`.

```
[27]: Mon.by_dim
```

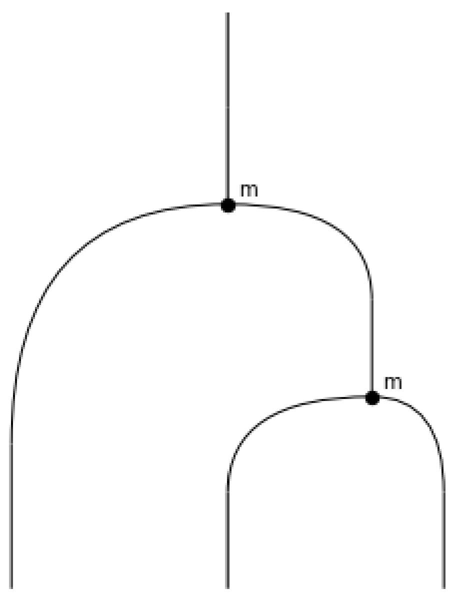
```
[27]: {0: {'pt'},
       1: {'a'},
       2: {'m', 'u'},
       3: {'assoc', 'assoc¹', 'lunit', 'runit'},
       4: {'inv(assoc, assoc¹)', 'inv(assoc¹, assoc)'}}
```

So, first of all, there's a new 3-dimensional generator, `assoc`^{1}.

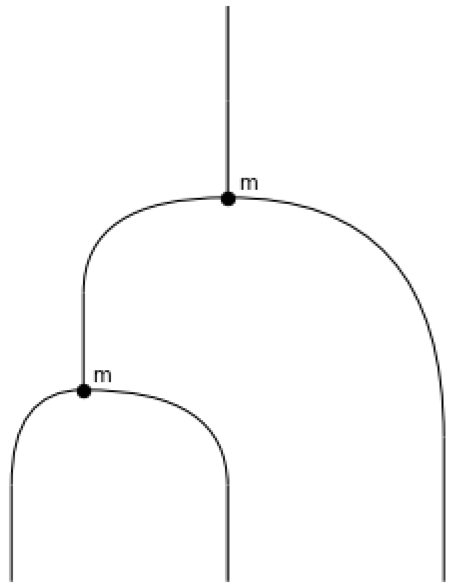
```
[28]: Mon['assoc¹'].draw()
      Mon['assoc¹'].draw_boundaries()
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



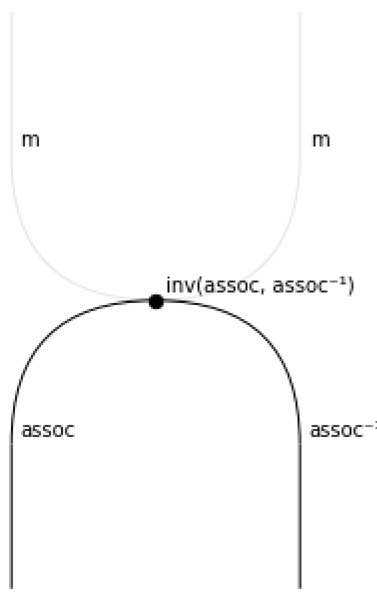
nbsphinx-code-borderwhite

This is the "weak inverse" of `assoc`: a generator with the same boundaries as `assoc`, but going in the reverse direction. If a generator has a weak inverse, we can get it with the `inverse` attribute.

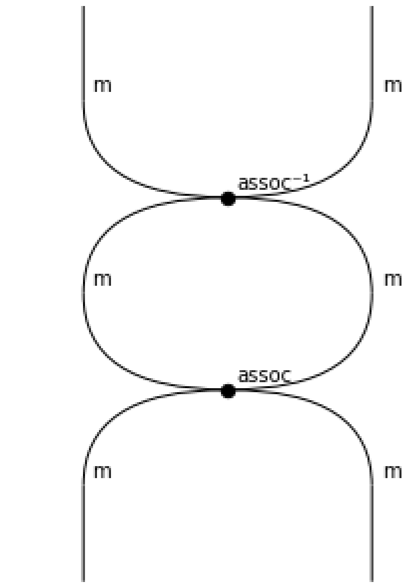
[29]:
```
assert assoc.inverse == Mon['assoc¹']
```

Then, we have two new *4-dimensional* generators, `inv(assoc, assoc`^{1}`)` and `inv(assoc`^{1}`, assoc)`.

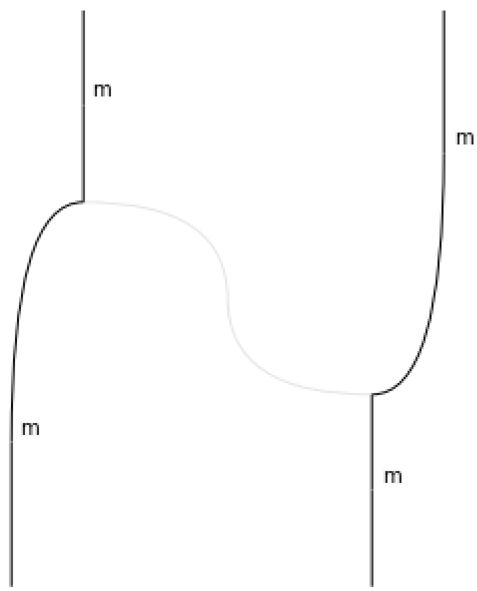
[30]:
```
Mon['inv(assoc, assoc¹)'].draw()
Mon['inv(assoc, assoc¹)'].draw_boundaries()
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

This generator "exhibits" the fact that `assoc`^{1} is a right inverse (right in diagrammatic order; left in composition order) for `assoc`: it goes from the pasting of `assoc` and `assoc`^{1}, to a weak unit on the input of `assoc`.

We call this a *right invertor* for `assoc`, and can get it with the `rinvertor` attribute.

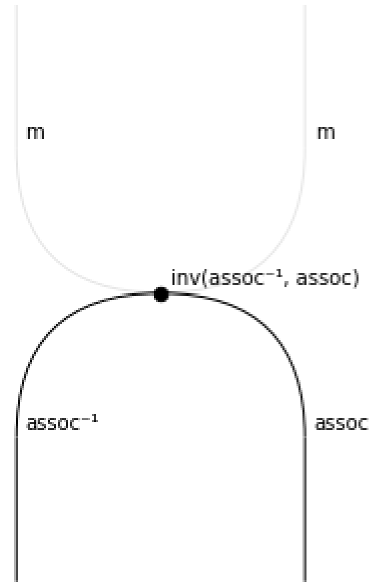
Similarly, `inv(assoc, assoc`^{1}`)` exhibits the fact that `assoc`^{1} is a left inverse for `assoc`. We call this a *left invertor* for `assoc`, and can retrieve it with the `linvertor` attribute.

Note that the left invertor for `assoc` is the right invertor for `assoc`^{1}, and vice versa!

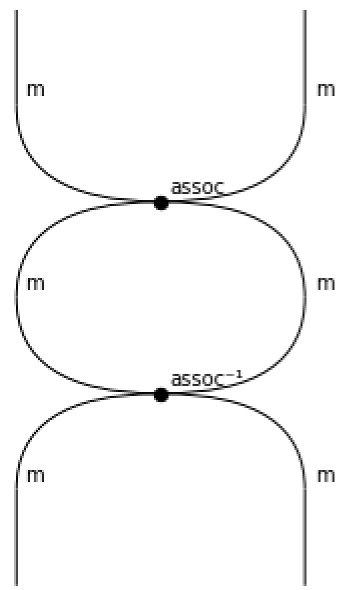
[31]:
```
assert assoc.rinvertor == Mon['inv(assoc, assoc¹)']
assert assoc.linvertor == assoc.inverse.rinvertor
```

In the theory of diagrammatic sets, these two "witnesses" should, themselves, be weakly invertible cells; since this would require an infinite number of generators, we leave it to the user to invert them when/if needed.

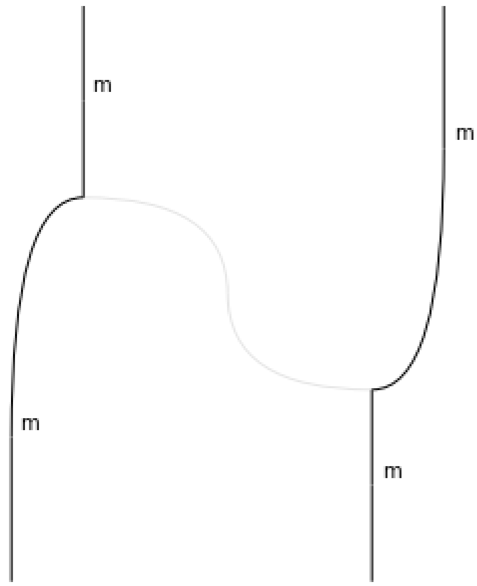
[32]:
```
Mon['inv(assoc¹, assoc)'].draw()
Mon['inv(assoc¹, assoc)'].draw_boundaries()
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

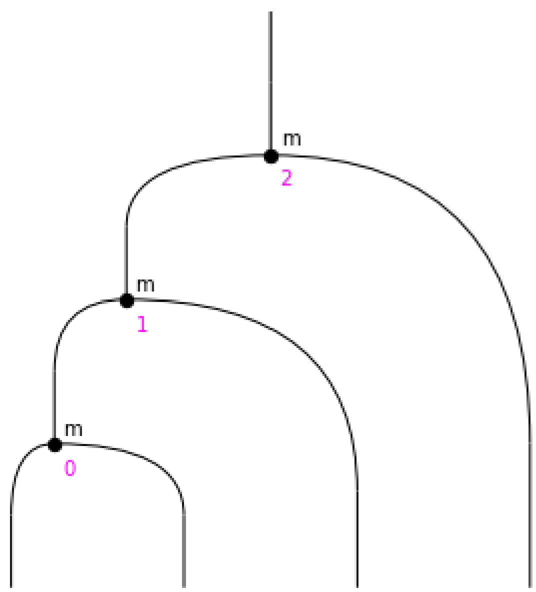


nbsphinx-code-borderwhite

5.1.4 Computing with diagrammatic rewrites

Let's start using our presentation to make some diagrammatic computations. First, we create a 2-dimensional diagram.

[33]:
```
start = m.to_inputs(0, m).to_inputs(0, m)
start.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

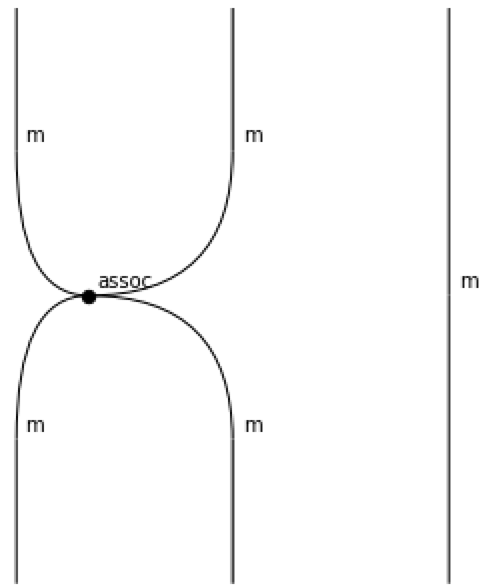
In traditional algebraic notation, this would correspond to the term $m(m(m(x, y), z), w)$.

We see that we can apply an associativity rewrite/equation in two places, corresponding to the nodes in positions (0, 1) and to the nodes in positions (1, 2).

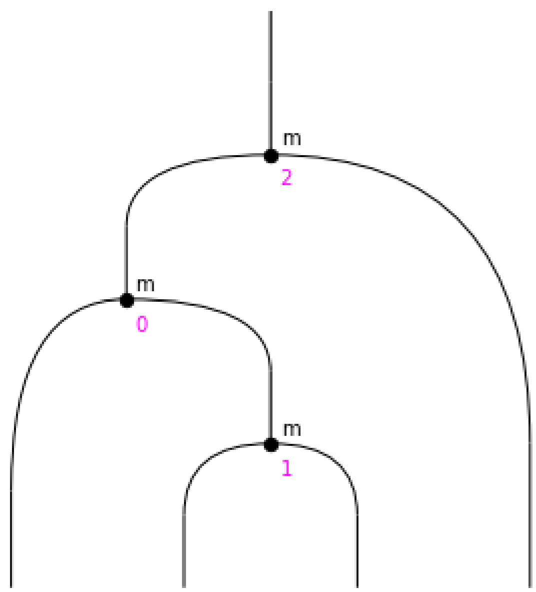
We can "apply rewrites" with the `rewrite` method. The result of `rewrite` is *not* going to be the "rewritten" 2-dimensional diagram. Instead, it will be a *3-dimensional* diagram whose input is the original diagram, and output is the rewritten diagram: an "embodiment" of the rewrite operation.

(The `rewrite` method is, in fact, a special instance of `to_outputs`; once you understand the principles of higher-dimensional rewriting, you should be able to see why).

[34]:
```
rew1 = start.rewrite([0,1], assoc)
rew1.draw()
rew1.output.draw(nodepositions=True)
```



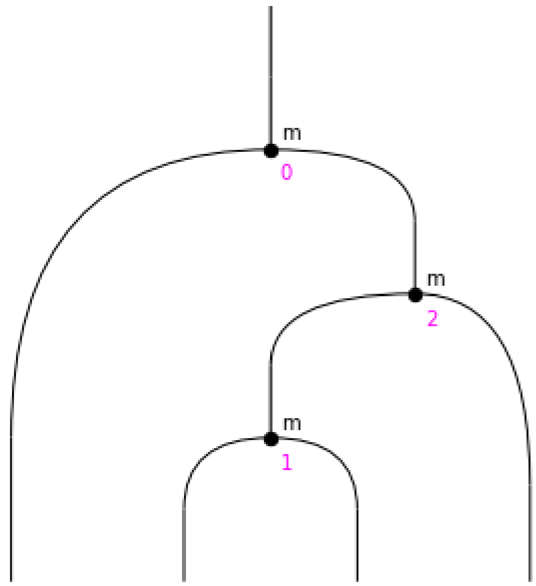
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

In the rewritten diagram, we can only apply `assoc` to the nodes (0, 2).

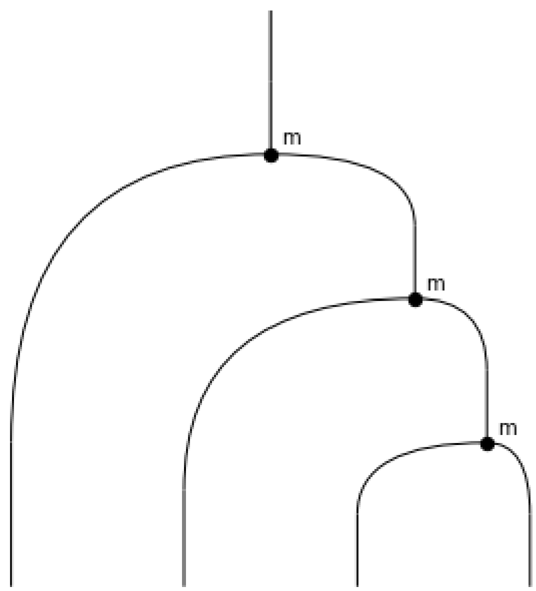
[35]:
```
rew2 = rew1.output.rewrite([0, 2], assoc)
rew2.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

Now, we can apply `assoc` to the nodes (1, 2).

```
[36]: rew3 = rew2.output.rewrite([1, 2], assoc)
      rew3.output.draw()
```

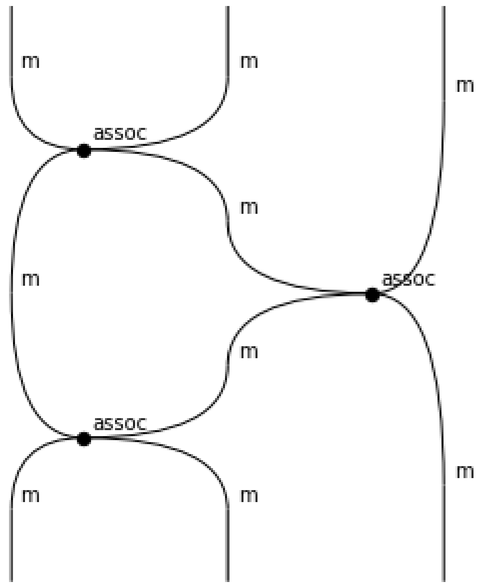


nbsphinx-code-borderwhite

We cannot apply `assoc` anywhere else. (Of course we could start applying `assoc`[1]).

Let's put together our sequence of rewrites.

```
[37]: seq1 = rewalt.Diagram.with_layers(rew1, rew2, rew3)
      seq1.draw()
```



nbsphinx-code-borderwhite

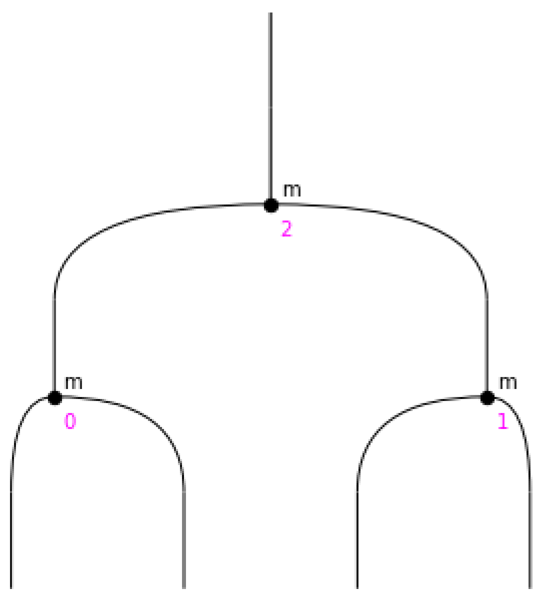
(We could have equally defined `seq1` as `rew1.paste(rew2).paste(rew3)`).

We can use the method `rewrite_steps` to get all our rewrite steps… and we can even produce a little gif animation with all the steps. (We'll make it loop backwards as well so it doesn't end too soon.)

[38]:
```
rewalt.strdiags.to_gif(*seq1.rewrite_steps, loop=True, path='monoids_1.gif')
```

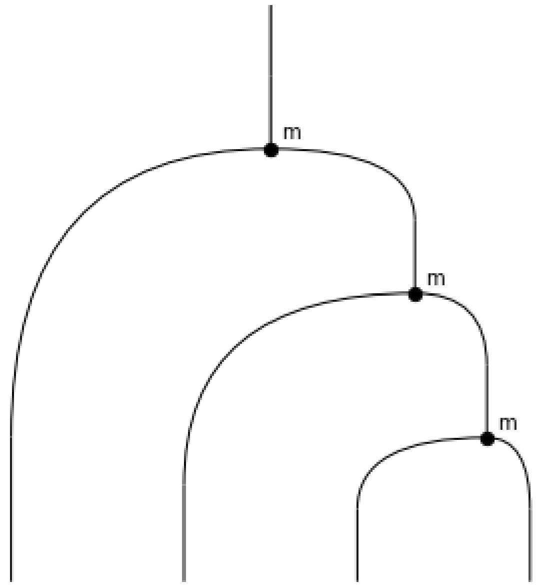
Let's go back to the `start` and pick a different rewrite, the one on nodes (1, 2).

[39]:
```
rew4 = start.rewrite([1, 2], assoc)
rew4.output.draw(nodepositions=True)
```



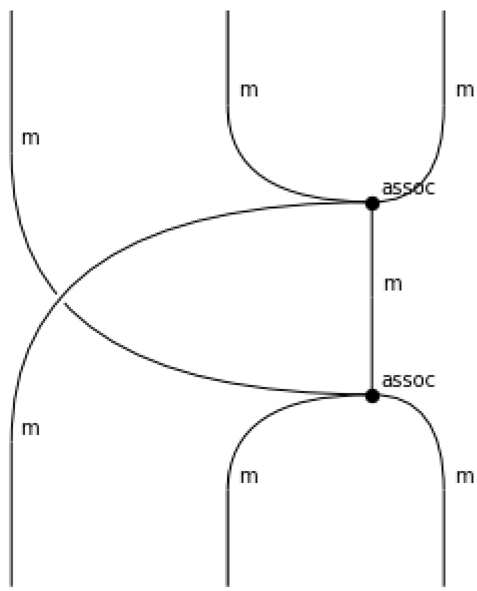
nbsphinx-code-borderwhite

[40]:
```
rew5 = rew4.output.rewrite([0, 2], assoc)
rew5.output.draw()
```



nbsphinx-code-borderwhite

[41]:
```
seq2 = rew4.paste(rew5)
seq2.draw()
rewalt.strdiags.to_gif(*seq2.rewrite_steps, loop=True, path='monoids_2.gif')
```



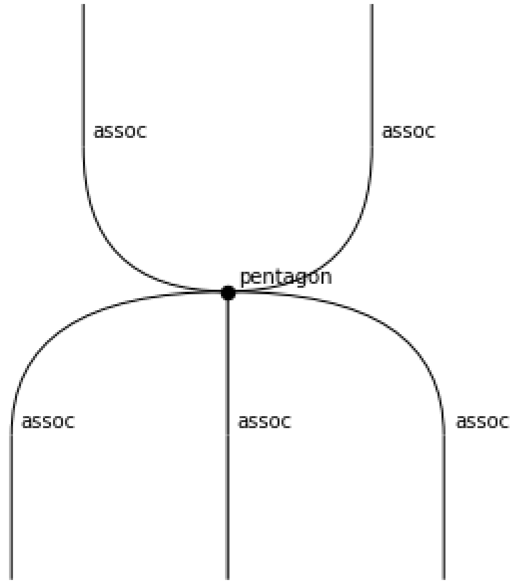
nbsphinx-code-borderwhite

You can see that `seq1` and `seq2` are two different sequences of rewrites with the same starting and ending point.

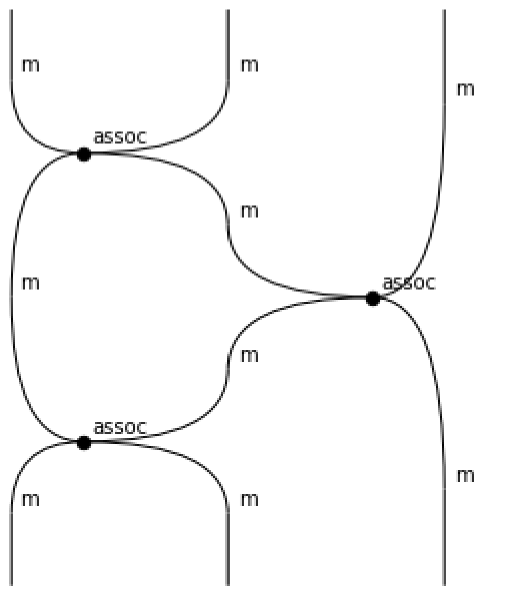
If you are familiar with the characterisation of monoidal categories as *pseudomonoids* in the monoidal 2-category of categories with cartesian product, you may recognise the two sides of Mac Lane's *pentagon equation*!

Indeed, we can add a 4-dimensional generator between the two, embodying Mac Lane's pentagon.

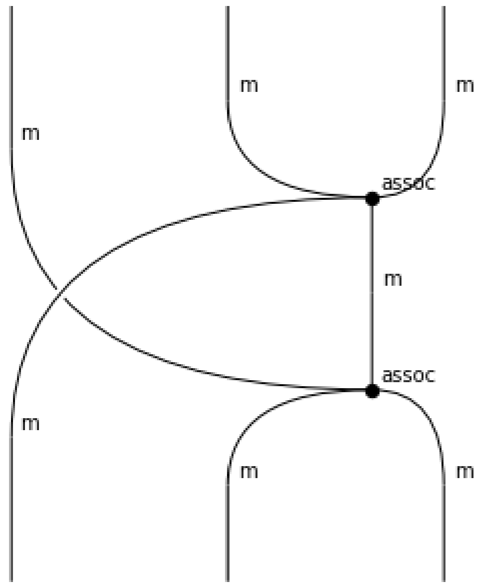
[42]:
```
pentagon = Mon.add('pentagon', seq1, seq2)
pentagon.draw()
pentagon.draw_boundaries()
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

We could go on and add generators corresponding to Mac Lane's *triangle*… but this was supposed to be about the theory of *monoids*, not of *lax* or *pseudo*monoids, so let's stop here instead.

5.2 Generating string diagrams

For any higher-dimensional diagram that we can create in `rewalt`, we can output a *string diagram* representation both as an image (with the Matplotlib backend), or as TikZ code that we can include in our LaTeX files.

Thus, one of the intended applications of `rewalt` is also to be a structure-aware, type-aware string diagram generator: we can build our string diagrams the way we build the morphisms/homotopies/operations/rewrites that they represent, and let `rewalt` do the typesetting for us.

In this notebook, we will work out one example, and explore the customisation options that we have.

Note that the placement and general style of nodes and wires is not currently customisable (except for the choice of orientation). However, `rewalt` is open source software and everyone is welcome to modify the algorithm to suit their aesthetic preferences.

5.2.1 A presentation of adjunctions

As an example, we will construct a presentation of the "theory of adjunctions", or "walking adjunction", whose models in a bicategory are adjunctions internal to that bicategory. (This has "dualities in monoidal categories" as a special case.) The triangle/zigzag/snake equations of adjunctions are some of the most well-known and recognisable in string diagrams.

The theory of adjunctions has two 0-cells and two 1-cells between them, going in opposite directions.

[1]:
```
import rewalt

Adj = rewalt.DiagSet()
x = Adj.add('x')
y = Adj.add('y')
```

(continues on next page)

(continued from previous page)

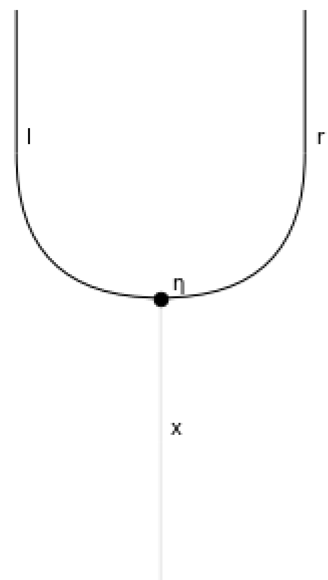
```
l = Adj.add('l', x, y)
r = Adj.add('r', y, x)
```

Then, we need to add two 2-cells, the *unit* and *counit* of the adjunction.

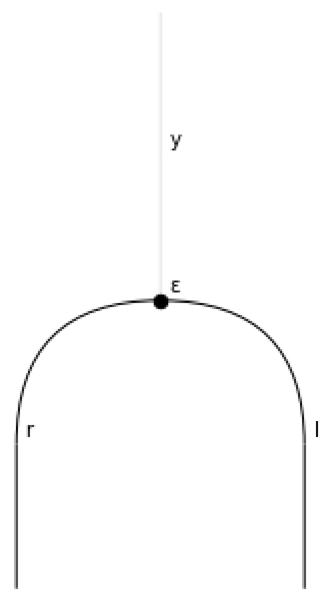
[2]:
```
eta = Adj.add('', x.unit(), l.paste(r))  # unit
eps = Adj.add('', r.paste(l), y.unit())  # counit
```

This is how `rewalt` draws the unit and counit by default.

[3]:
```
eta.draw()
eps.draw()
```



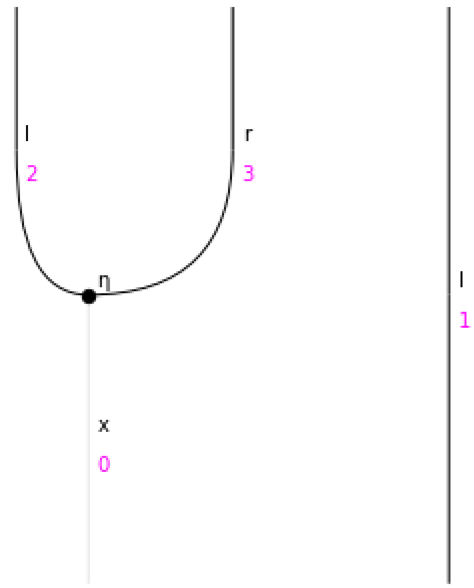
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

We can use the picture as a visual aid to see how to paste the unit and counit together to get the left-hand side of the triangle equations. For example, if we add an `l` to the *right* of `eta`…

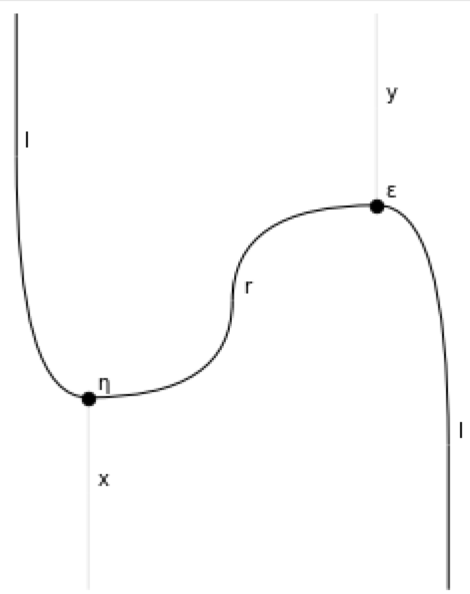
[4]:
```
eta.paste(l).draw(wirepositions=True)
```



nbsphinx-code-borderwhite

… we can plug an `eps` to the wires in positions (3, 1).

[5]:
```
lhs1 = eta.paste(l).to_outputs([3, 1], eps)
lhs1.draw()
```

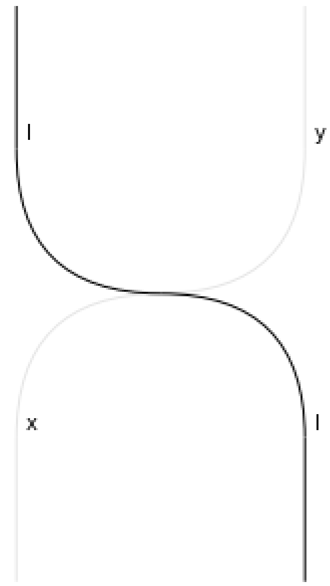


nbsphinx-code-borderwhite

This needs to be equated to "the identity on `l`", except we have weak units on `x` in the input and on `y` in the output.

We can in fact obtain the degenerate 2-cell with the right type as one of the *cubical degeneracies* on `l`.

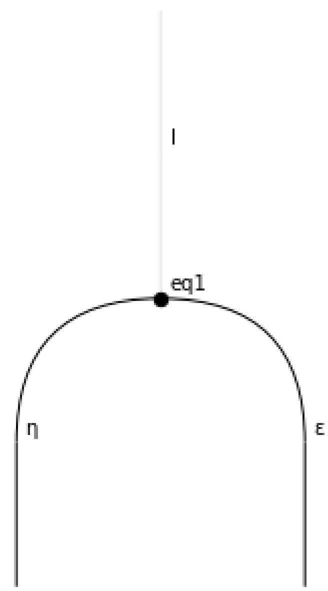
[6]:
```
rhs1 = l.cube_degeneracy(1)
rhs1.draw()
```



nbsphinx-code-borderwhite

We can now add our first "oriented equation".

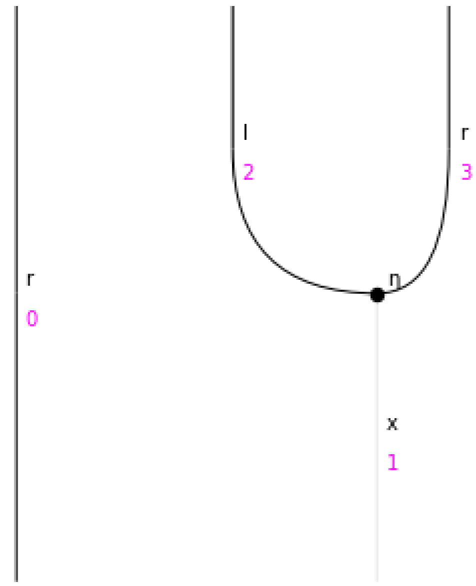
[7]:
```
eq1 = Adj.add('eq1', lhs1, rhs1)
eq1.draw()
```



nbsphinx-code-borderwhite

For the second one, we can proceed symmetrically. We add an `r` to the *left* of `eta`…

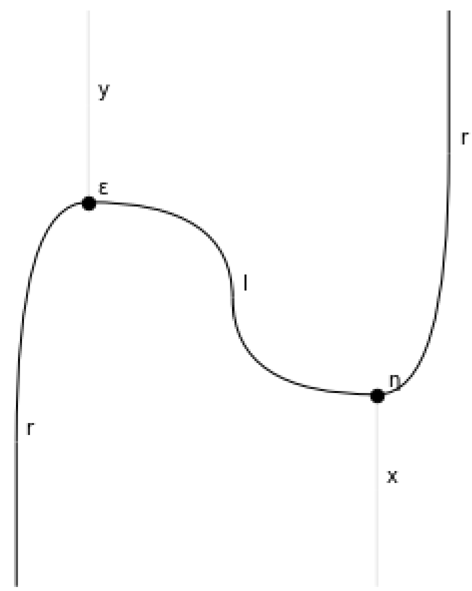
[8]:
```
r.paste(eta).draw(wirepositions=True)
```



nbsphinx-code-borderwhite

… and we plug an eps to the wires in positions (0, 2) to get the left-hand side of the second equation.

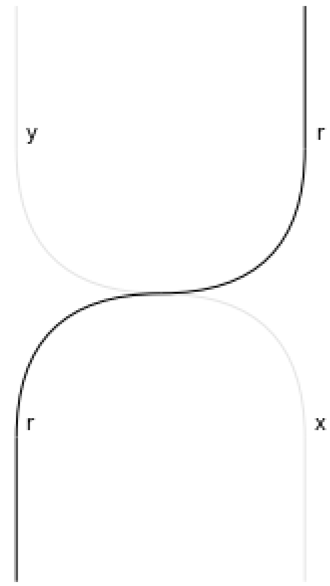
[9]:
```
lhs2 = r.paste(eta).to_outputs([0, 2], eps)
lhs2.draw()
```



nbsphinx-code-borderwhite

To get the right-hand-side, we use a different cubical degeneracy on r.

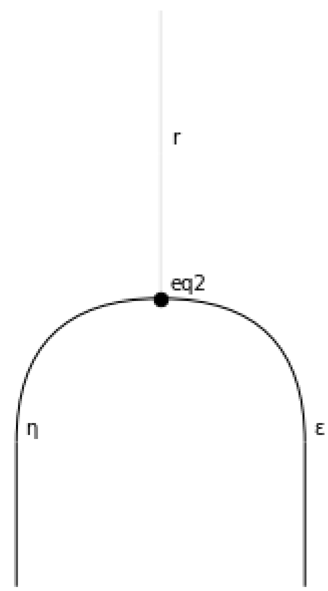
[10]:
```
rhs2 = r.cube_degeneracy(0)
rhs2.draw()
```



nbsphinx-code-borderwhite

And finally, we add the second triangle equation.

[11]:
```
eq2 = Adj.add('eq2', lhs2, rhs2)
eq2.draw()
```



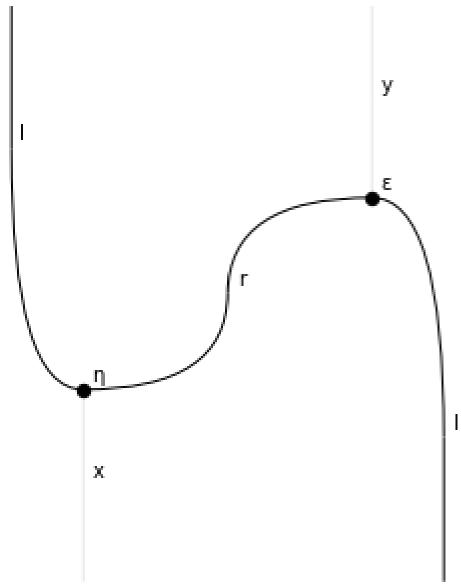
nbsphinx-code-borderwhite

That's it, we have a presentation. (We could also invert `eq1` and `eq2` but that's besides the point of this exercise).

5.2.2 Customising string diagrams

Let's return to the first triangle equation. The default string diagram representation of its left-hand side is this.

[12]:
```
eq1.input.draw()
```



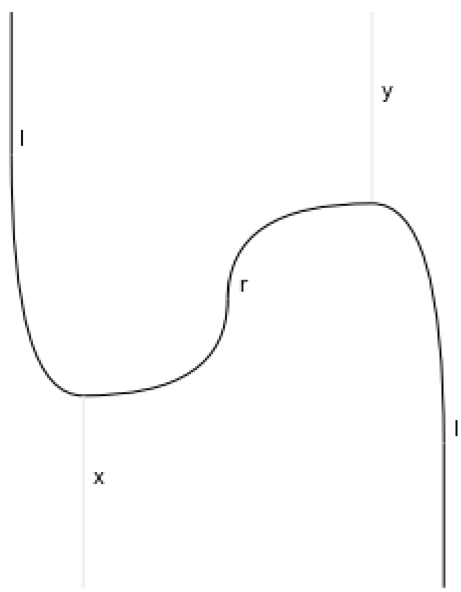
nbsphinx-code-borderwhite

Let's make it a bit nicer.

First of all, it is quite common to draw units and counits as "bent wires" (aka "cups and caps"), without a node, so that the triangle equations look like topological trivialities.

We can do this by disabling node drawing for these generators of `Adj`.

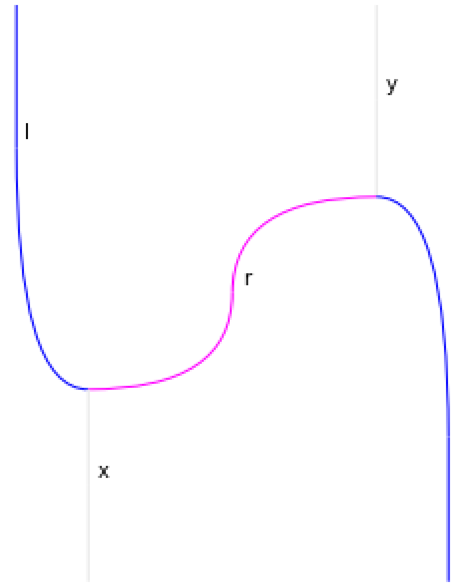
[13]:
```
Adj.update('', draw_node=False)
Adj.update('', draw_node=False)
eq1.input.draw()
```



nbsphinx-code-borderwhite

Then, since we have only two 1-cells, why not also colour-code them?

[14]:
```
Adj.update('l', color='blue')
Adj.update('r', color='magenta')
eq1.input.draw()
```

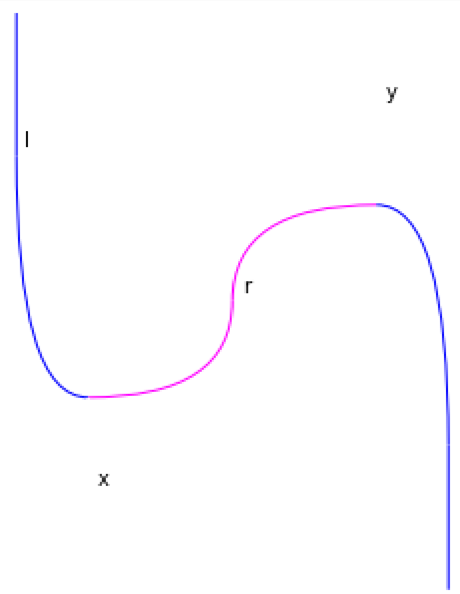


nbsphinx-code-borderwhite

When we are working in `rewalt`, it is good to see the weak units, because we need to take them into account to know that everything typechecks.

However, we may want to “hide them away” if, for example, our diagrams are to be interpreted in a strict 2-category. We can do this by changing the alpha factor for degenerate wires to 0.

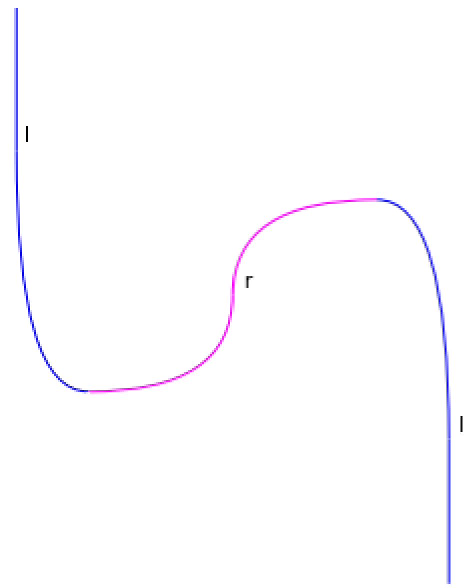
[15]:
```
eq1.input.draw(degenalpha=0)
```



nbsphinx-code-borderwhite

Note that this still shows the weak unit labels, which is actually helpful in this setting because it reminds us of the type of `l` and `r`. If we wanted to get rid of them, we could deactivate labels for these generators.

[16]:
```
Adj.update('x', draw_label=False)
Adj.update('y', draw_label=False)
eq1.input.draw(degenalpha=0)
```

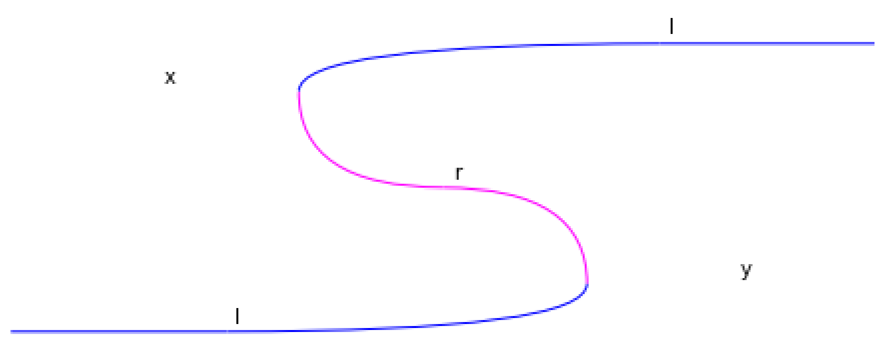


nbsphinx-code-borderwhite

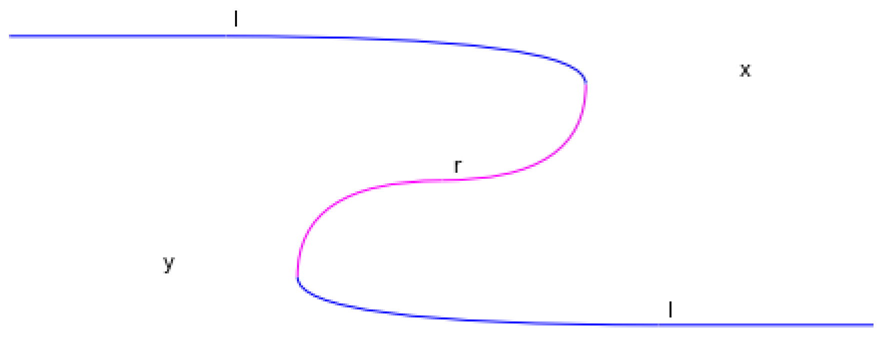
[17]:
```
Adj.update('x', draw_label=True)
Adj.update('y', draw_label=True)
```

There are different factions on what the "correct" orientation of string diagrams is. In `rewalt`, the default is bottom-to-top, but it can be changed.

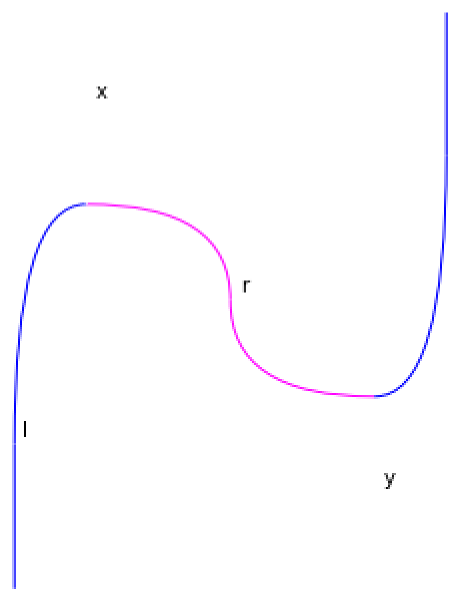
[18]:
```
eq1.input.draw(degenalpha=0, orientation='lr')
eq1.input.draw(degenalpha=0, orientation='rl')
eq1.input.draw(degenalpha=0, orientation='tb')
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



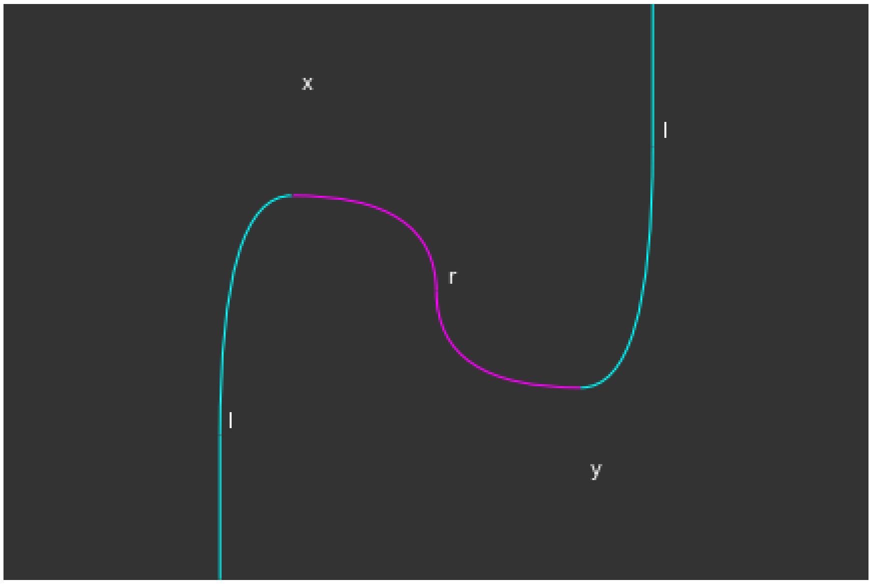
nbsphinx-code-borderwhite

We can change the default settings by reassigning the values of `rewalt.strdiags.DEFAULT`. Let's say we want all our string diagrams to be top-to-bottom with no degenerate wires.

[19]:
```
rewalt.strdiags.DEFAULT['orientation'] = 'tb'
rewalt.strdiags.DEFAULT['degenalpha'] = 0
```

Now, how about a dark theme?

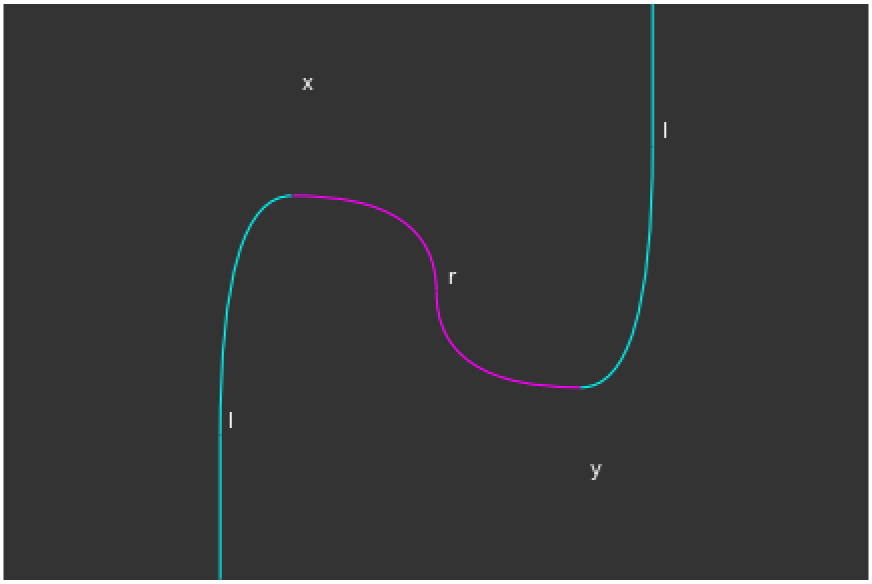
[20]:
```
Adj.update('l', color='cyan')
eq1.input.draw(bgcolor='0.2', fgcolor='white')
```



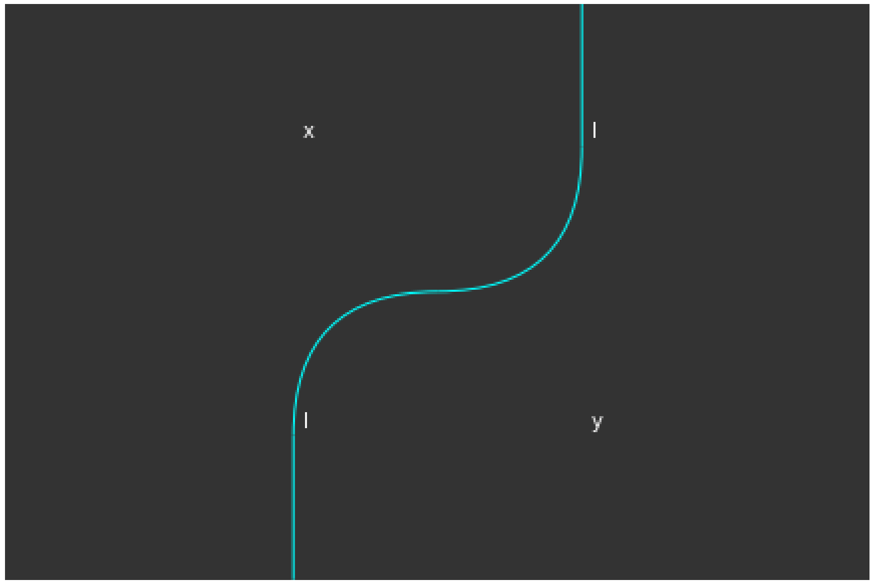
nbsphinx-code-borderwhite

Let's see what the sides of our two triangle equations look like now.

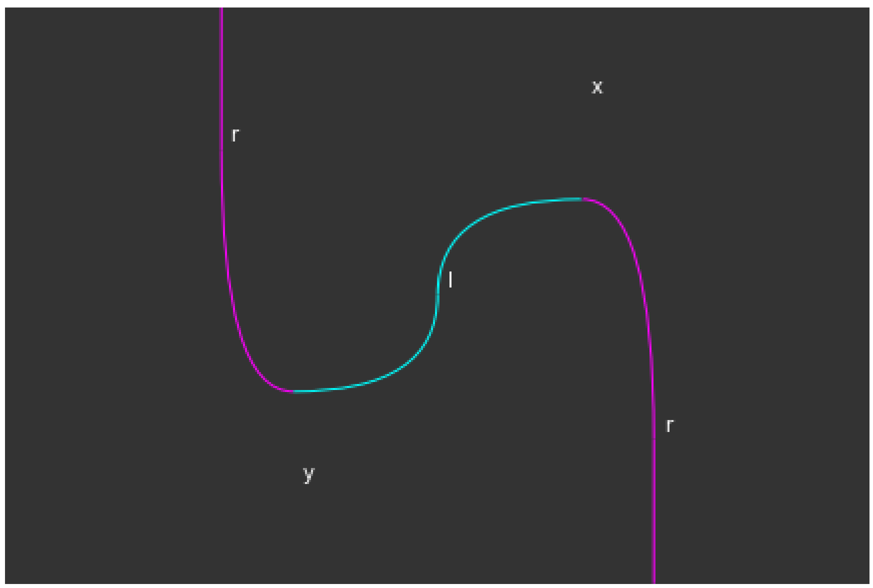
[21]:
```
eq1.draw_boundaries(bgcolor='0.2', fgcolor='white')
eq2.draw_boundaries(bgcolor='0.2', fgcolor='white')
```



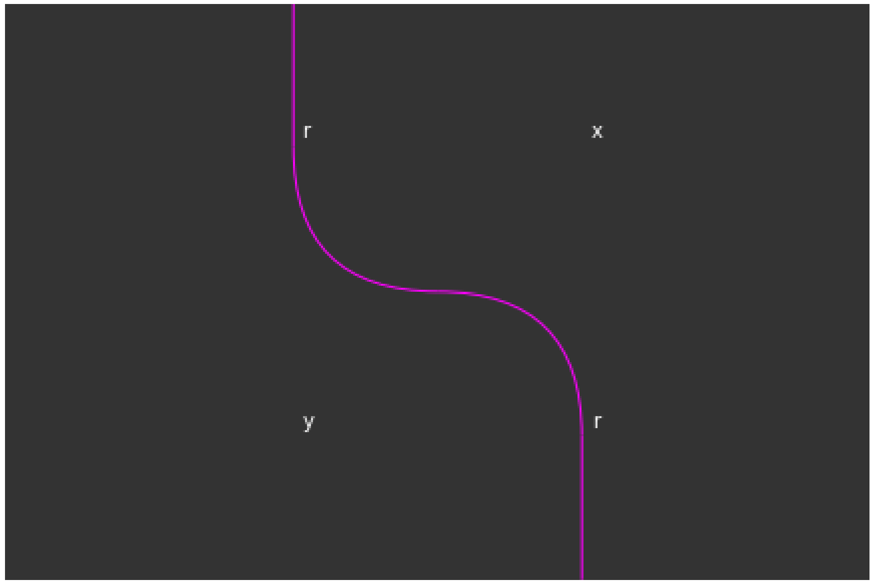
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

nbsphinx-code-borderwhite 

If we are happy with the look, we can output TikZ code. Note that both labels and colour settings are passed to the TikZ output as they are, so we should change the background colour setting to something that LaTeX can recognise.

TikZ output uses coordinates in $[0,1] \times [0,1]$. Since this is quite small, the output is scaled 3x by default; this can be changed with the `scale`, `xscale`, and `yscale` keyword arguments.

Also, by default, all wires are drawn with a contour, which is useful in higher dimensions when wires can overlap. Since we are in 2d and this doesn't happen, we can avoid drawing contours by setting the `depth` keyword argument to `False`.

[22]:
```
eq1.input.draw(
    bgcolor='darkgray', fgcolor='white', depth=False,
    tikz=True, xscale=8, yscale=6, path='stringdiagrams_1.tex')
```

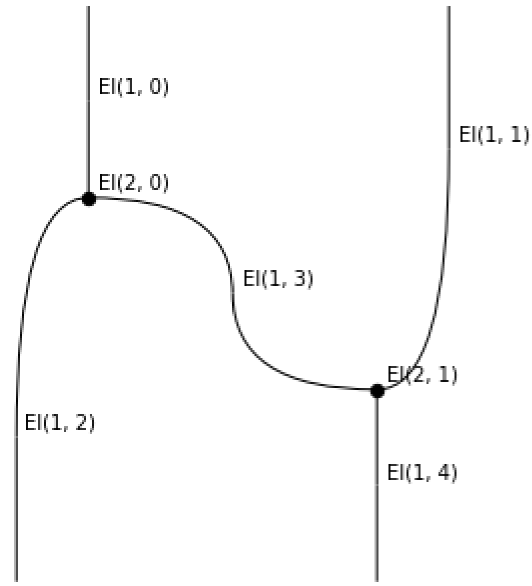
Here's the generated TikZ code and the output PDF compiled with pdflatex.

5.2.3 Fun with higher-dimensional shapes

We can have string diagram representations not only of "diagrams in a `DiagSet`", but also of *shapes* and *maps of shapes* of diagrams.

For example, this is the shape of the diagram we have been using as example.

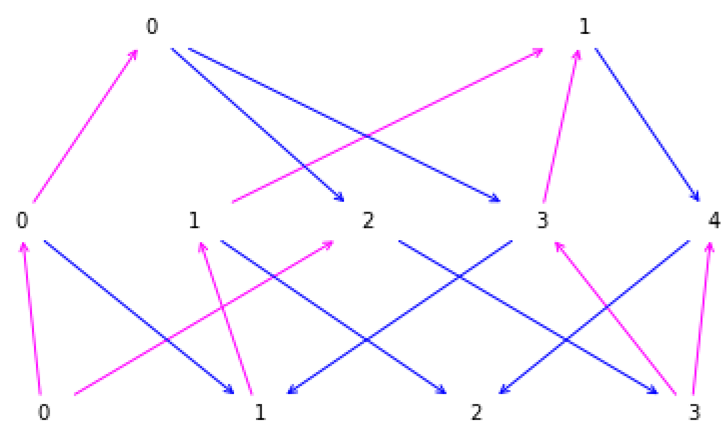
[23]:
```
eq1.input.shape.draw()
```



nbsphinx-code-borderwhite

Every wire and node corresponds to a unique face of the diagram shape, specified by its dimension (2 for nodes, 1 for wires) and position. We can match them to elements of the *oriented face poset* of the diagram shape.

[24]:
```
eq1.input.shape.hasse(labels=False)
```

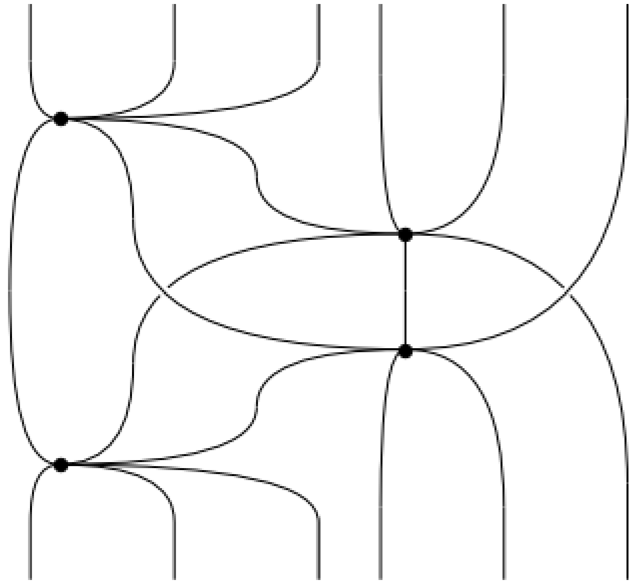


nbsphinx-code-borderwhite

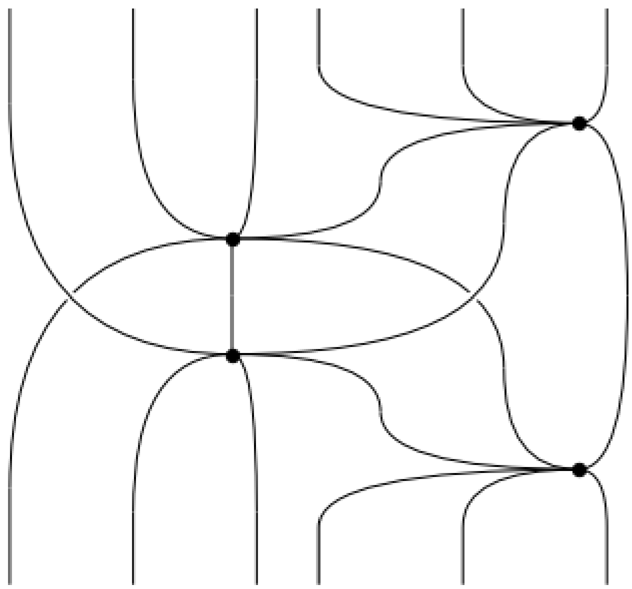
A quick way to get some interesting higher-dimensional diagrams, and see some of the things that happen with string diagram representations in higher dimensions, is to use some of the constructors for special higher-dimensional shapes, such as simplices and cubes.

For example, these are the string diagrams for the 3-dimensional boundaries of the oriented 4-cube.

[25]:
```
tesseract = rewalt.Shape.cube(4)
tesseract.draw_boundaries(labels=False)
```



nbsphinx-code-borderwhite



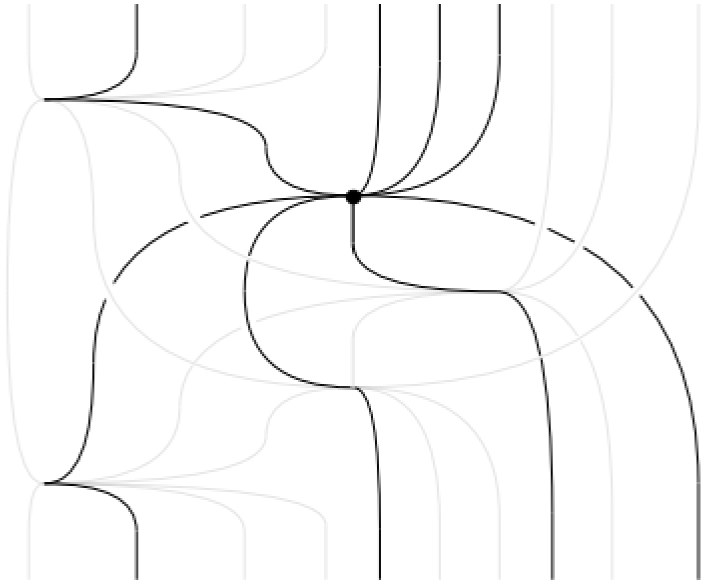
nbsphinx-code-borderwhite

You can see that wires can cross each other in 3-dimensional diagrams.

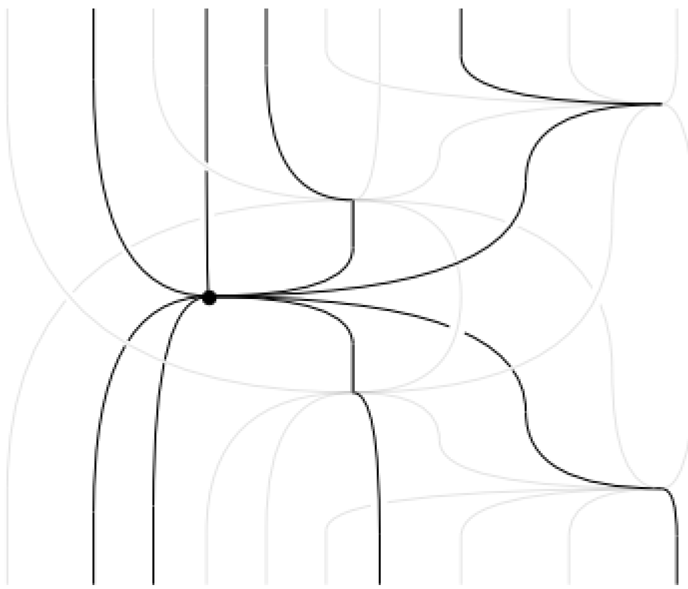
For something even more complicated, let's look at a *cubical connection* map on the 4-cube, which is a surjective map from the 5-cube.

(Since this will contain many degenerate cells, we will reinstate weak units in string diagrams.)

[26]:
```
connection = tesseract.cube_connection(1, '-')
connection.draw_boundaries(labels=False, degenalpha=0.1)
```



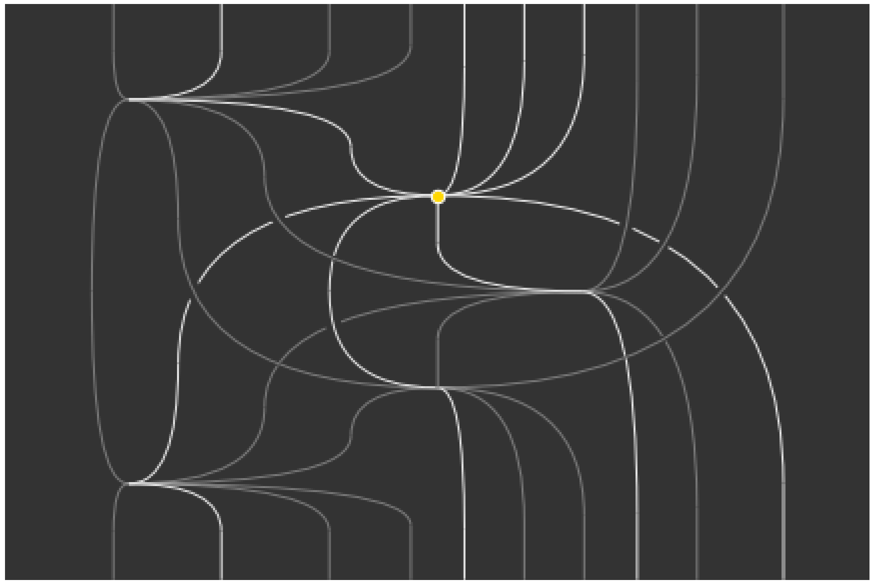
nbsphinx-code-borderwhite



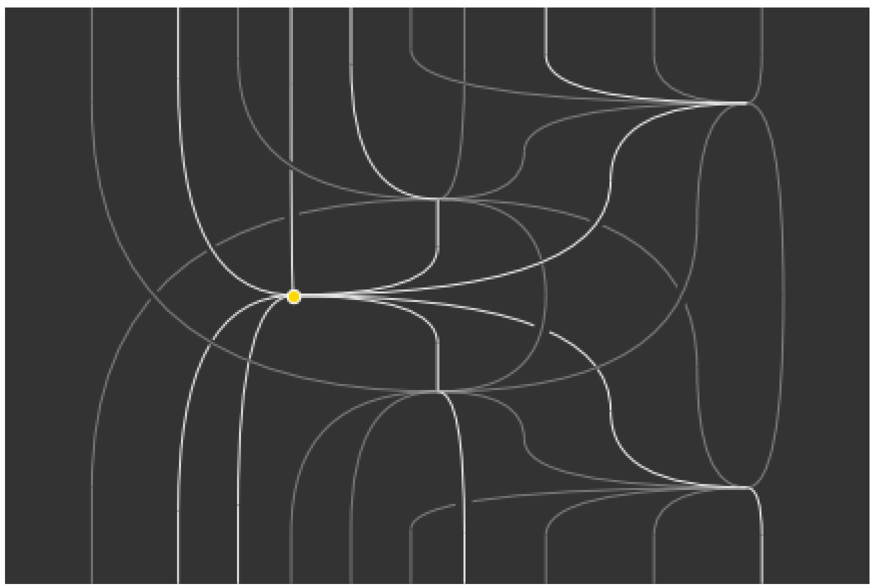
nbsphinx-code-borderwhite

And let's play a little bit with colours.

[27]:
```
connection.draw_boundaries(
    labels=False, bgcolor='0.2', fgcolor='0.9', degenalpha=0.4,
    nodecolor='gold', nodestroke='white')
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

Looks nice, no?

5.3 Exploring simplices and cubes

Diagrammatic sets — the structure implemented by `rewalt`'s `DiagSet` class — support a wide variety of "shapes of diagrams", while remaining "topologically sound". This makes them a convenient tool for diagrammatic reasoning in higher category, higher algebra, and homotopy theory.

Among these shapes are some subclasses that are widely used on their own: in particular, the *simplices* and the *cubes*. Indeed, both simplicial sets and cubical sets with connections are special instances of diagrammatic sets (their categories are full subcategories of the category of diagrammatic sets).

Reflecting this, `rewalt` contains a full implementation of (finitely presented) simplicial sets and of (finitely presented) cubical sets with connections. These are nothing more than diagrammatic sets whose generators all have simplicial and cubical shapes! The `Diagram` objects that have simplicial or cubical shapes come with special methods for constructing simplicial and cubical faces, degeneracies, and connections.

Since all our shapes have a "globular" orientation (half a boundary is "input", half a boundary is "output"), our simplices are in fact Street's oriented simplices. Similarly our cubes are "oriented" as in cubical -categories.

Understanding higher-dimensional oriented simplices and cubes can be difficult. In this notebook, we will try to use `rewalt` and its visualisation methods to get a grip on some low-but-not-too-low-dimensional examples.

5.3.1 Oriented simplices

Oriented simplices of any dimension are built with the `Shapes.simplex` constructor. Let's start with the lowest possible dimension: -1.

[1]:
```
import rewalt

empty = rewalt.Shape.simplex(-1)
```

This is just the empty diagram shape.

[2]:
```
len(empty)
```

[2]:
```
0
```

The 0-dimensional simplex is a point.

[3]:
```
point = rewalt.Shape.simplex(0)
point.draw()
```

nbsphinx-code-borderwhite

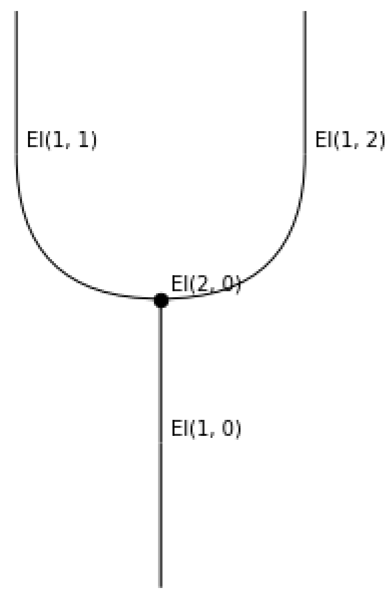
The 1-dimensional oriented simplex is an arrow.

[4]:
```
arrow = rewalt.Shape.simplex(1)
arrow.draw()
```

nbsphinx-code-borderwhite

Things get a little more interesting in dimension 2. The oriented 2-simplex is a triangle with two output sides and one input side. In string diagrams, it is, for example, the shape of a *comonoid comultiplication*.

[5]:
```
triangle = rewalt.Shape.simplex(2)
triangle.draw()
```

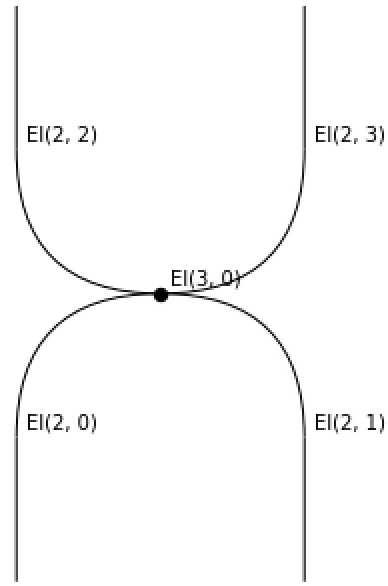


nbsphinx-code-borderwhite

Let's go one dimension higher. The oriented 3-simplex is a tetrahedron with two output faces and two input faces, each of them shaped as an oriented 2-simplex.

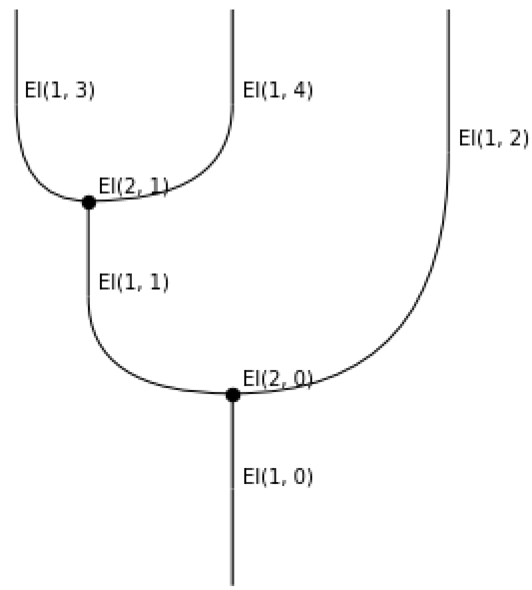
Let's draw both its top-dimensional "slice" string diagram, and its input and output boundaries.

[6]:
```
tetrahedron = rewalt.Shape.simplex(3)
tetrahedron.draw()
```

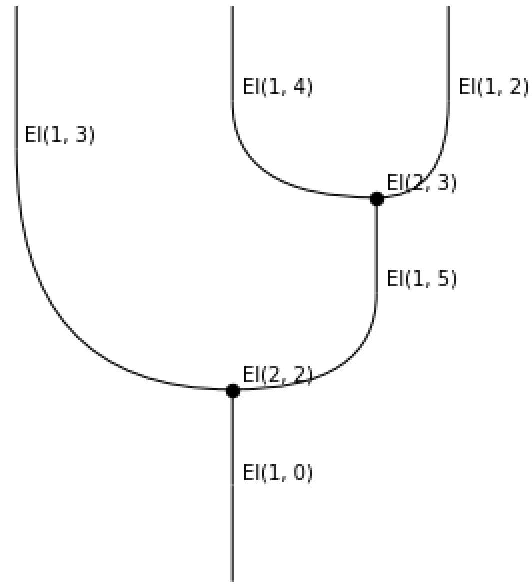


nbsphinx-code-borderwhite

[7]:
```
tetrahedron.draw_boundaries()
```



nbsphinx-code-borderwhite

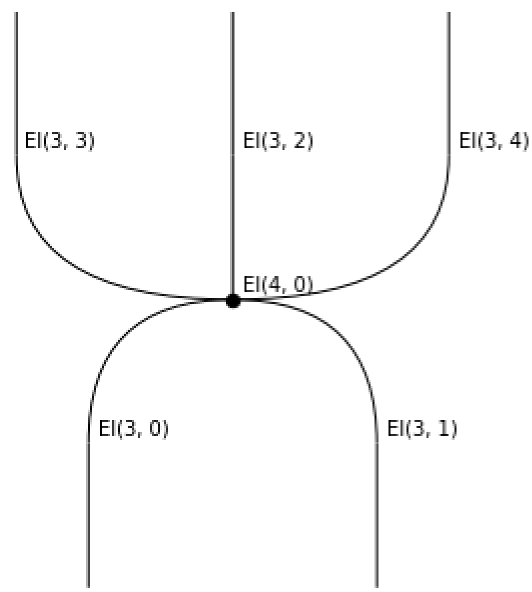


nbsphinx-code-borderwhite

If we stick to the interpretation of the oriented 2-simplex as “the shape of a comultiplication”, then the oriented 3-simplex is “the shape of a (co)associativity equation”, or “the shape of a coassociator”!

What happens if we go to dimension 4?

[8]:
```
pentachoron = rewalt.Shape.simplex(4)
pentachoron.draw()
```

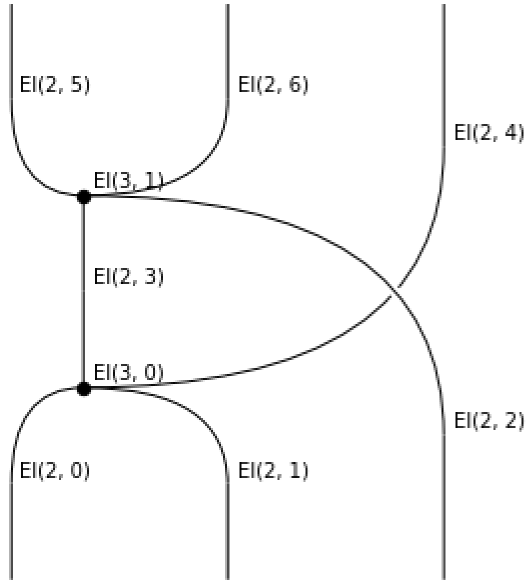


nbsphinx-code-borderwhite

This is a *pentachoron*, also known as the 5-cell, with three output tetrahedral faces and two input tetrahedral faces.

Let’s see what its boundaries look like, starting from the input.

[9]:
```
penta_input = pentachoron.input
penta_input.draw()
```



nbsphinx-code-borderwhite

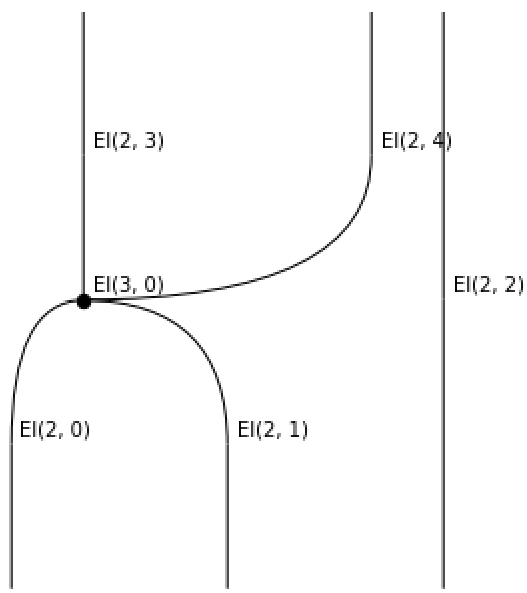
This is a slice of a 3-dimensional diagram with two 3-dimensional cells.

This is still hard to visualise directly in three dimensions; instead, we are going to try to visualise it as a *sequence of rewrites on 2-dimensional diagrams*.

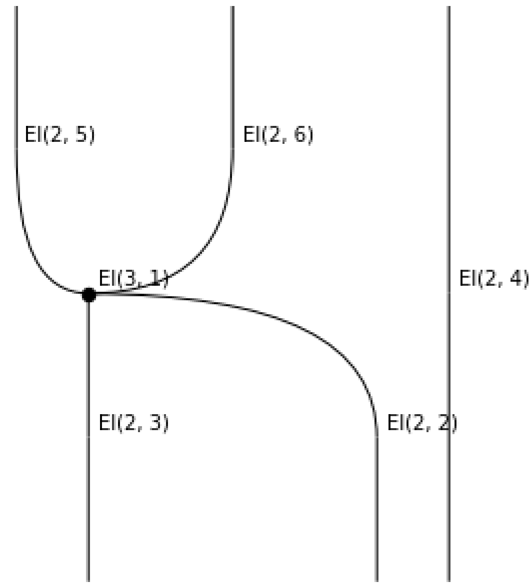
For that purpose, we use the `generate_layering` method, which creates a "layering" of a diagram into a sequence of rewrites, one for each one of its top-dimensional cells. Then, we can

- get a list of the layers with the `layers` attribute, or
- get a list of the corresponding "rewrite steps" with the `rewrite_steps` attribute.

[10]:
```
penta_input.generate_layering()
rewalt.strdiags.draw(*penta_input.layers)
```

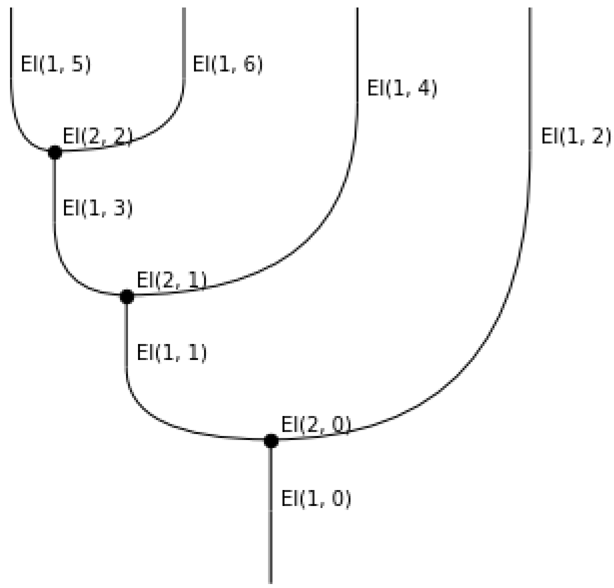


nbsphinx-code-borderwhite

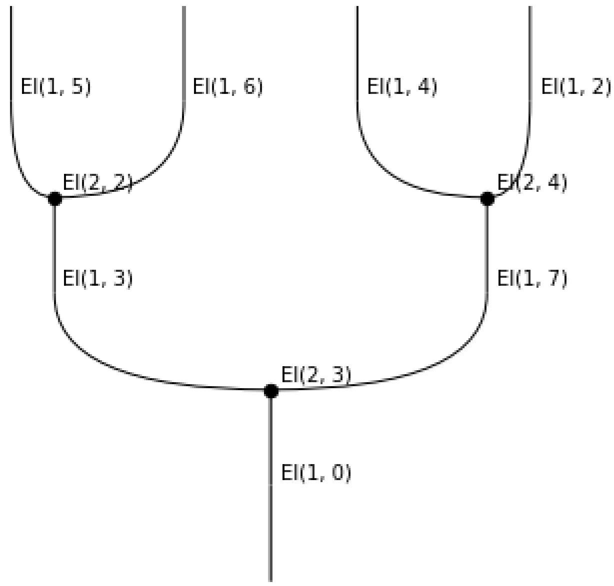


nbsphinx-code-borderwhite

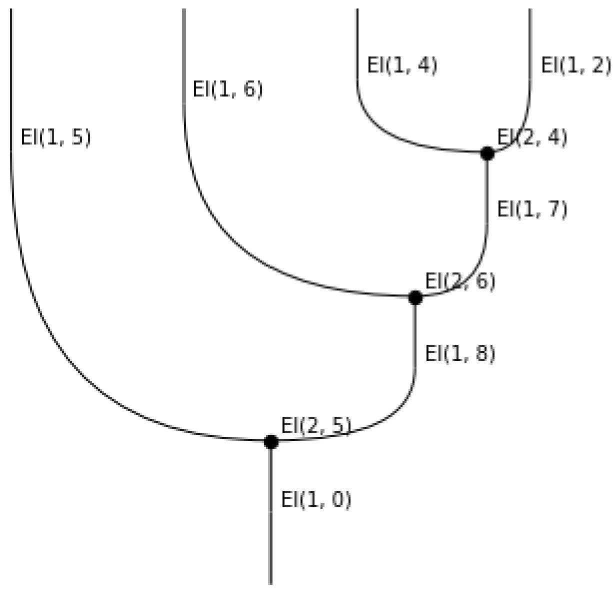
```
[11]: rewalt.strdiags.draw(*penta_input.rewrite_steps)
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

So, we can see that

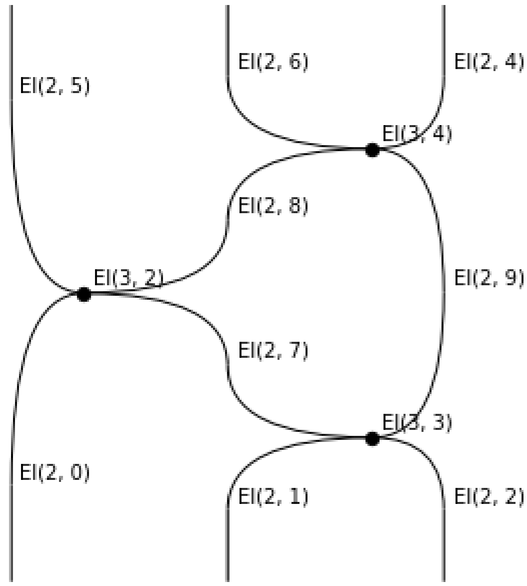
- first the 3-dimensional face `El(3, 0)` "rewrites" the triangles `El(2, 0)` and `El(2, 1)` into the triangles `El(2, 3)` and `El(2, 4)`,
- then the 3-dimensional face `El(3, 1)` "rewrites" the triangles `El(2, 2)` and `El(2, 3)` into the triangles `El(2, 5)` and `El(2, 6)`.

We can also create a gif "movie" of the rewrite steps (and make it loop backwards so it doesn't stop too soon).

[12]:
```
rewalt.strdiags.to_gif(
    *penta_input.rewrite_steps,
    loop=True, path='simplicescubes_1.gif')
```

Now, let's look at the output boundary of the oriented 4-simplex.

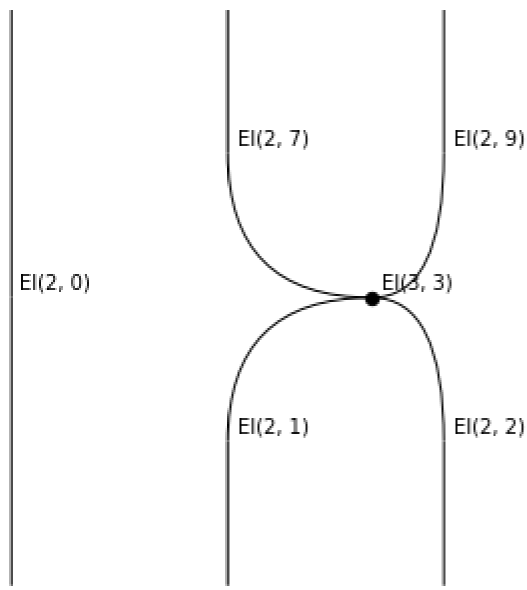
```
[13]: penta_output = pentachoron.output
      penta_output.draw()
```



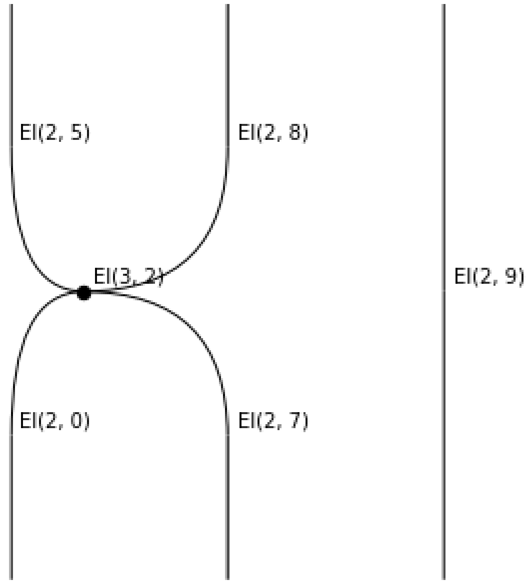
nbsphinx-code-borderwhite

This is the slice of a 3-dimensional diagram with three 3-dimensional cells. Let's proceed as with the input.

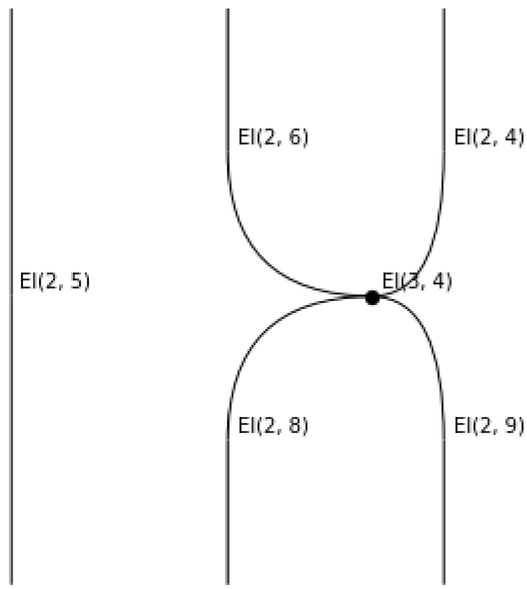
```
[14]: penta_output.generate_layering()
      rewalt.strdiags.draw(*penta_output.layers)
```



nbsphinx-code-borderwhite

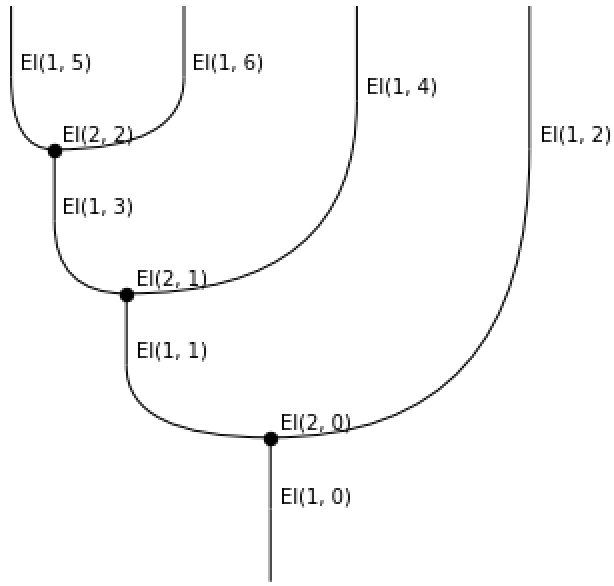


nbsphinx-code-borderwhite

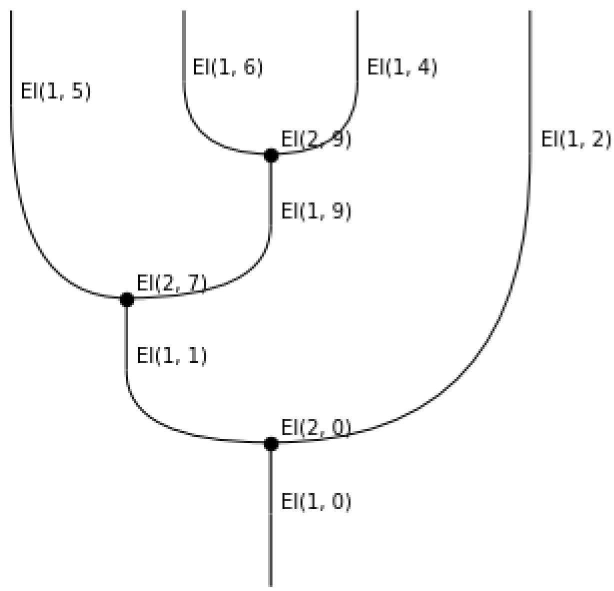


nbsphinx-code-borderwhite

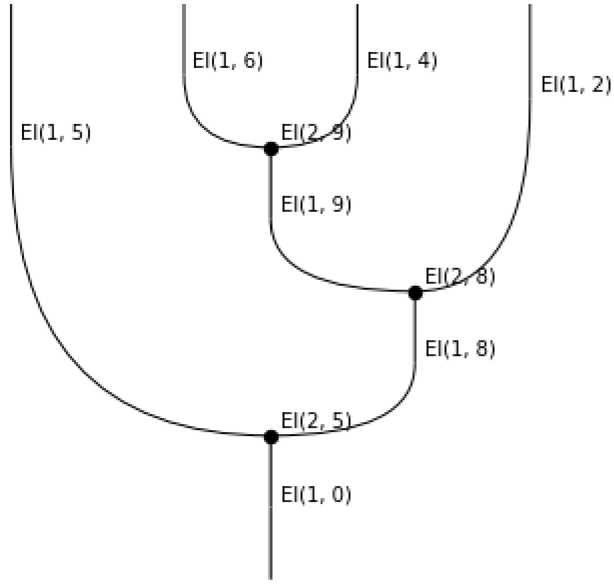
[15]:
```
rewalt.strdiags.draw(*penta_output.rewrite_steps)
```



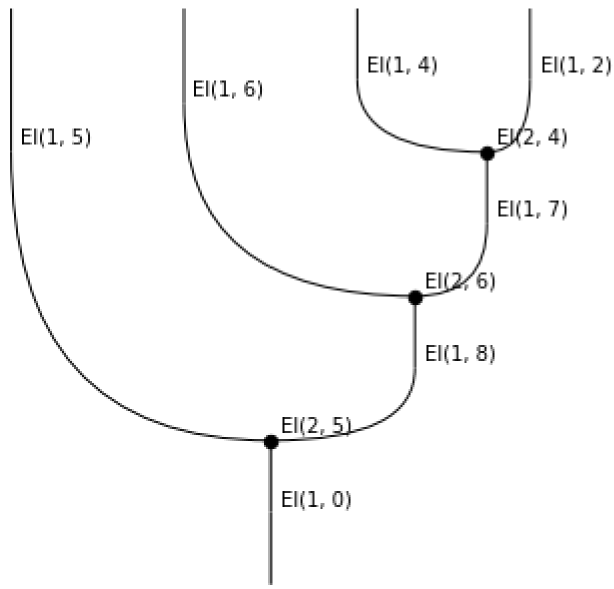
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

Let's also make a movie of these.

[16]:
```
rewalt.strdiags.to_gif(
    *penta_output.rewrite_steps, loop=True,
    path='simplicescubes_2.gif')
```

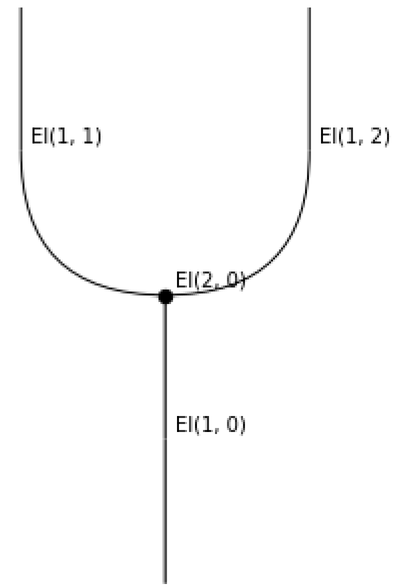
The two sides of the oriented 4-simplex are, in fact, the two sides of an equation dual to *Mac Lane's pentagon*. This was featured at the end of this other notebook.

5.3.2 Maps of simplices

So far we have only looked at the oriented simplices "in isolation". Let's see how we can use `rewalt` to understand their face and degeneracy maps.

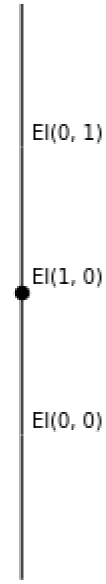
Faces are quite simple; let's look at the example of the 2-simplex. This has 3 faces.

[17]:
```
triangle.draw()
for n in range(3):
    triangle.simplex_face(n).draw()
```

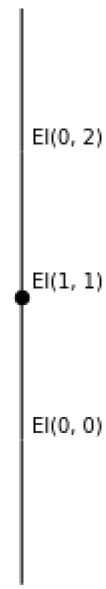


nbsphinx-code-borderwhite

nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

By comparing labels, we can see that

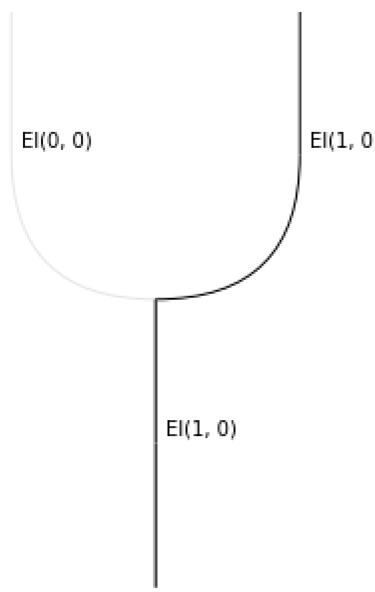
- the 0th face of the 2-simplex is the *rightmost* output,
- the 1st face of the 2-simplex is the only input, and
- the 2nd face of the 2-simplex is the *leftmost* output.

In general, the faces of an oriented simplex alternate between inputs and outputs, always starting with an output at index 0.

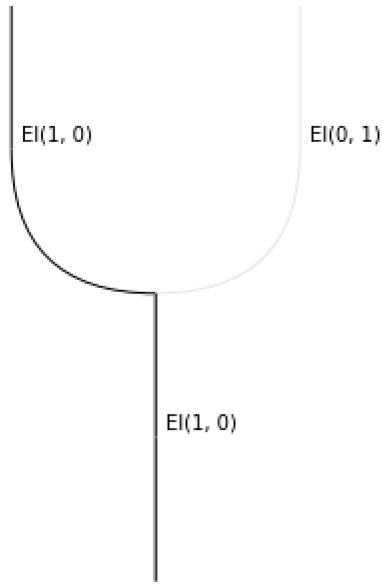
Let's look at degeneracies; these are somewhat more interesting. There are two degeneracies on the 1-simplex.

[18]:
```
arrow.draw()
for n in range(2):
    arrow.simplex_degeneracy(n).draw()
```

nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



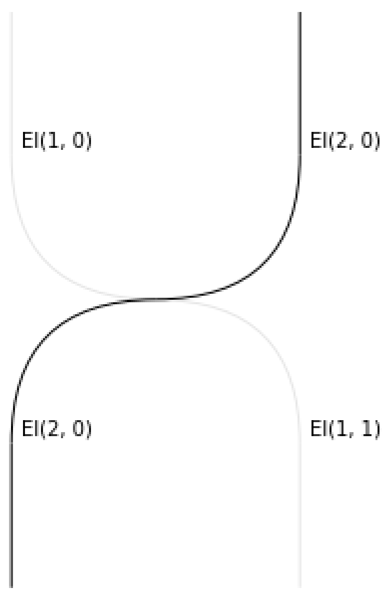
nbsphinx-code-borderwhite

The two diagrams represent two surjective ("collapsing") maps from the 2-simplex to the 1-simplex. The string diagrams tell us that

- the 0th degeneracy sends the 2-cell, its input, and the *rightmost* output of the 2-simplex onto the 1-cell of the 1-simplex, and collapses the *leftmost* output onto its input 0-cell;
- the 1st degeneracy sends the 2-cell, its input, and the *leftmost* output of the 2-simplex onto the 1-cell of the 1-simplex, and collapses the *rightmost* output onto its output 0-cell.

Now, let's take a look at one degeneracy of the 2-simplex.

[19]:
```
triangle.simplex_degeneracy(0).draw()
```

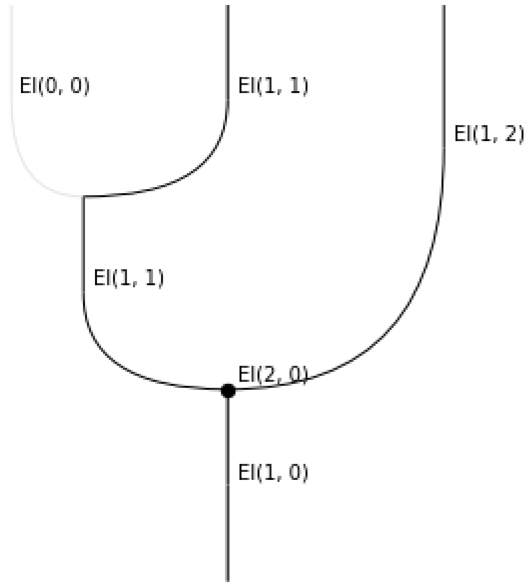


nbsphinx-code-borderwhite

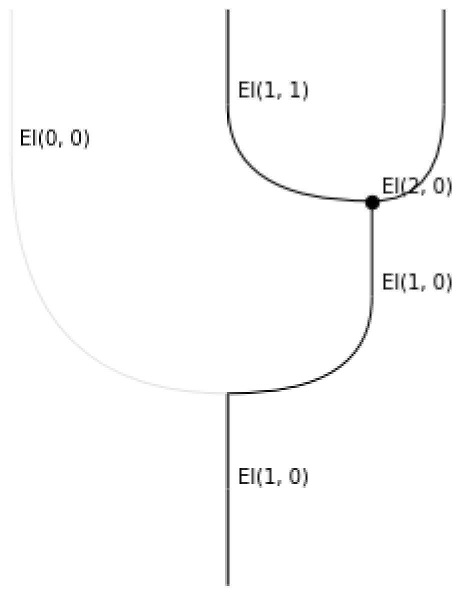
This represents a collapsing map from the 3-simplex onto the 2-simplex; the string diagram tells us which input and which output of the 3-simplex are collapsed, and which are sent to the 2-cell of the 2-simplex.

Let's obtain some more information by looking at the boundaries.

[20]:
```
triangle.simplex_degeneracy(0).draw_boundaries()
```



nbsphinx-code-borderwhite

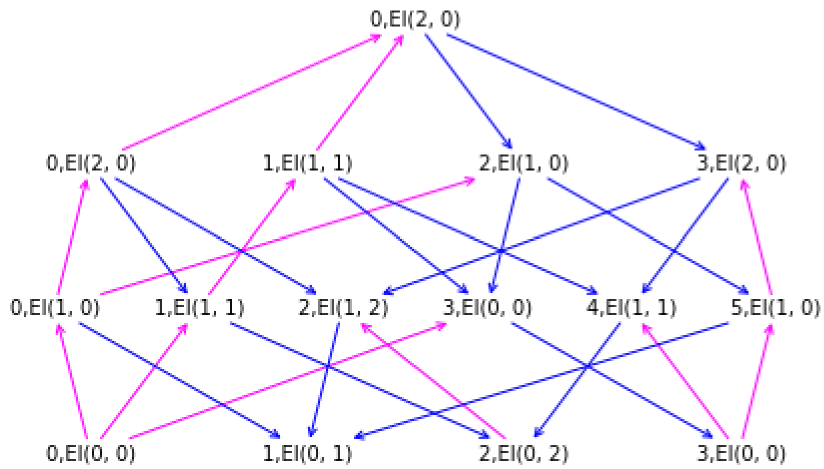


nbsphinx-code-borderwhite

This tells us exactly how the two collapsed 2-dimensional faces of the 3-simplex are collapsed: we can tell that, in both cases, it is the leftmost output that is collapsed, hence the *0-th* degeneracy of the 1-simplex is used.

By the way, if we want a precise (but not very intuitive) description of a map, we can use the Hasse diagram visualisation:

[21]:
```
triangle.simplex_degeneracy(0).hasse()
```



nbsphinx-code-borderwhite

This shows us the "oriented face poset" of the source of the map — here, the 3-simplex — with each element labelled with its image through the map. For example, the third element of the third row from the bottom is labelled with `El(1, 0)`; this means that the map sends `El(2, 2)` to `El(1, 0)` (we are counting from 0).

5.3.3 Constructing a simplicial set

Let's briefly look at how we can use `rewalt` to construct a simplicial set. As a simple example, we will construct the 3-dimensional *real projective space* $\mathbb{R}P^3$, with its cell structure made up of a single cell in each dimension.

The first step is to create an empty diagrammatic set.

[22]:
```
RP3 = rewalt.DiagSet()
```

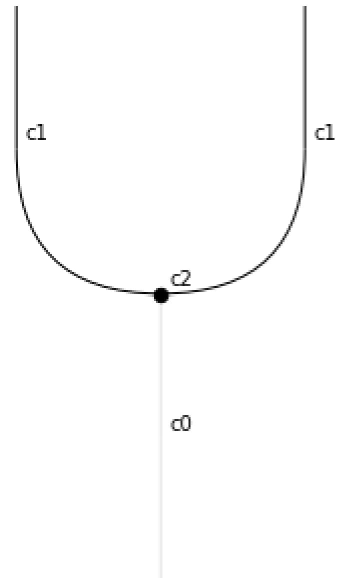
To ensure that this is really a simplicial set, we only add generators with the `add_simplex` method, taking, as arguments, the simplicial faces of the new generator in the same order as given by `simplex_face`.

(In dimension 0 and 1, there's no substantial difference between `add` and `add_simplex`).

[23]:
```
c0 = RP3.add_simplex('c0')
c1 = RP3.add_simplex('c1', c0, c0)
```

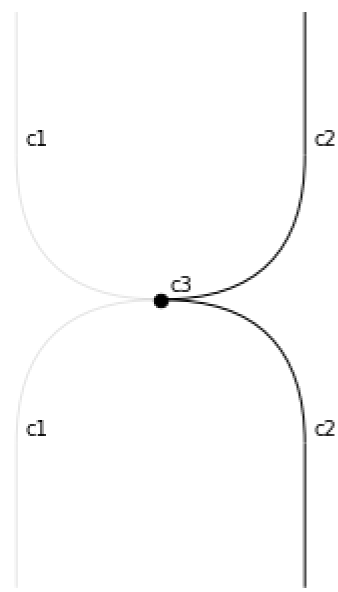
We construct degenerate simplices over the generators with the `simplex_degeneracy` method.

[24]:
```
c2 = RP3.add_simplex('c2', c1, c0.simplex_degeneracy(0), c1)
c2.draw()
```



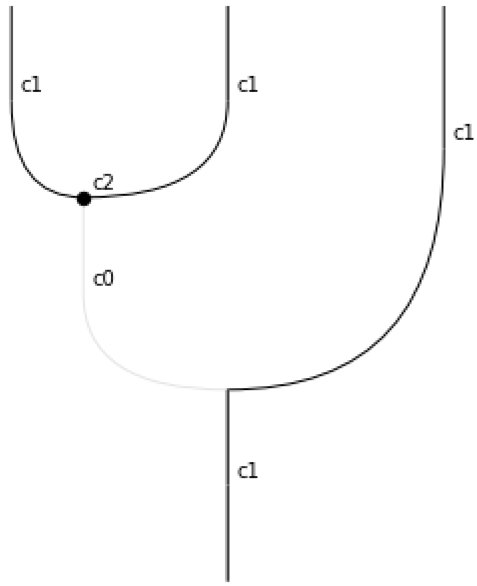
nbsphinx-code-borderwhite

[25]:
```
c3 = RP3.add_simplex(
    'c3',
    c2, c1.simplex_degeneracy(0), c1.simplex_degeneracy(1), c2)
c3.draw()
```

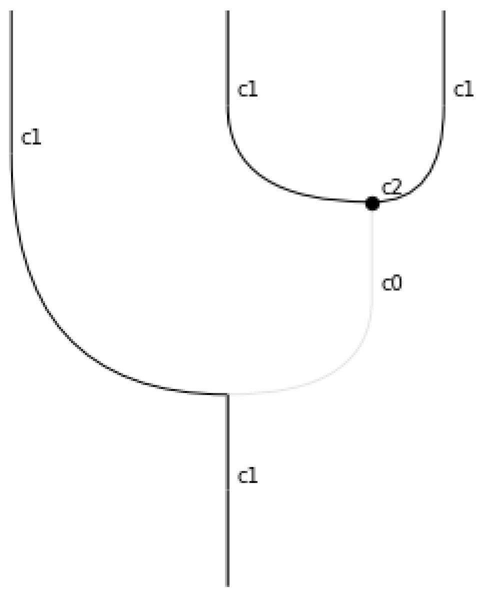


nbsphinx-code-borderwhite

[26]:
```
c3.draw_boundaries()
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

There we go; `RP3` is now a simplicial model of the 3-dimensional real projective space. We can check that this is "really" a simplicial set:

[27]:
```
RP3.issimplicial
```

[27]:
```
True
```

In future releases, we plan to add features that will allow us to automatically compute some topological invariants of cell complexes constructed as `DiagSet` objects.

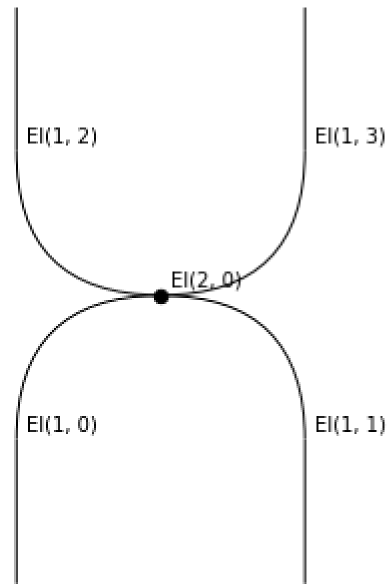
5.3.4 Oriented cubes

Let's move on from simplices to cubes; these can be obtained with the `Shape.cube` constructor. Unlike in simplices, there is no (-1)-cube. The 0-cube and the 1-cube are, in fact, the same as the 0-simplex and the 1-simplex.

[28]:
```
assert point == rewalt.Shape.cube(0)
assert arrow == rewalt.Shape.cube(1)
```

So the first interesting case is the oriented 2-cube: this is a square with two output faces and two input faces.

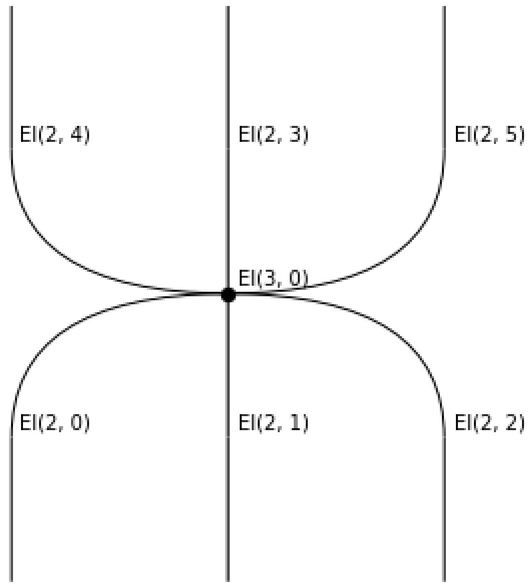
[29]:
```
square = rewalt.Shape.cube(2)
square.draw()
```



nbsphinx-code-borderwhite

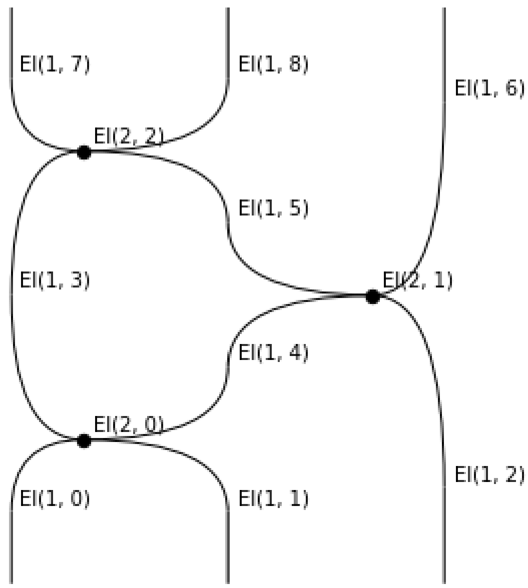
Next, the oriented 3-cube has three output faces and three input faces. (In fact, the oriented n-cube always has n inputs and n outputs.)

[30]:
```
cube = rewalt.Shape.cube(3)
cube.draw()
```

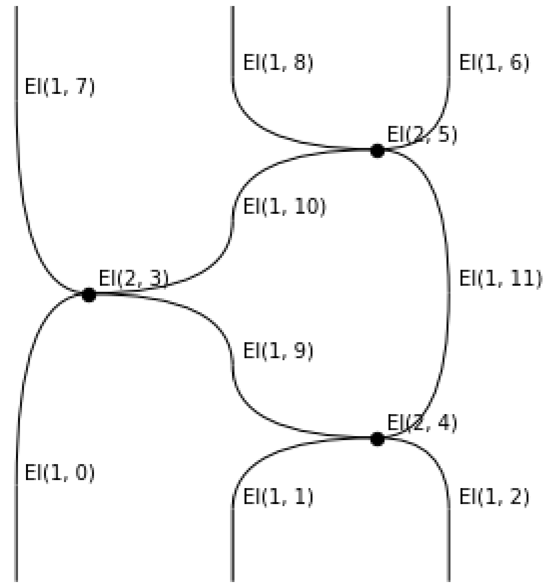


nbsphinx-code-borderwhite

[31]:
```
cube.draw_boundaries()
```



nbsphinx-code-borderwhite

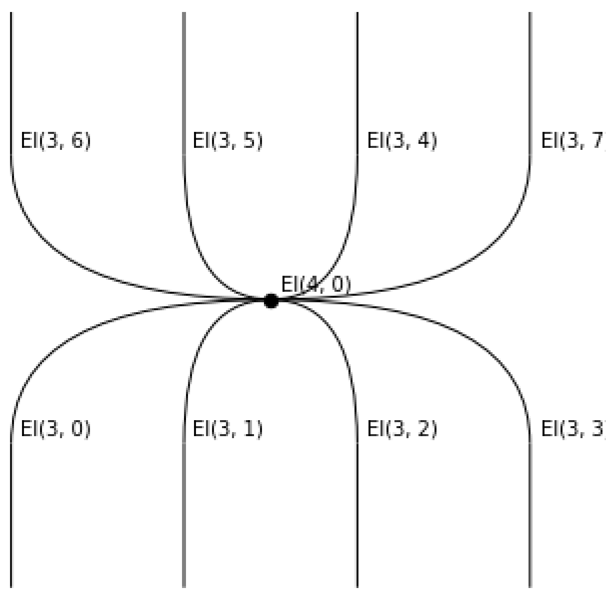


nbsphinx-code-borderwhite

You may see the 2-dimensional boundaries of the oriented 3-cube, in string diagrams, as the shapes of the two sides of the Yang-Baxter equation, or the two sides of the third Reidemeister move.

Let's move on to the 4-dimensional cube.

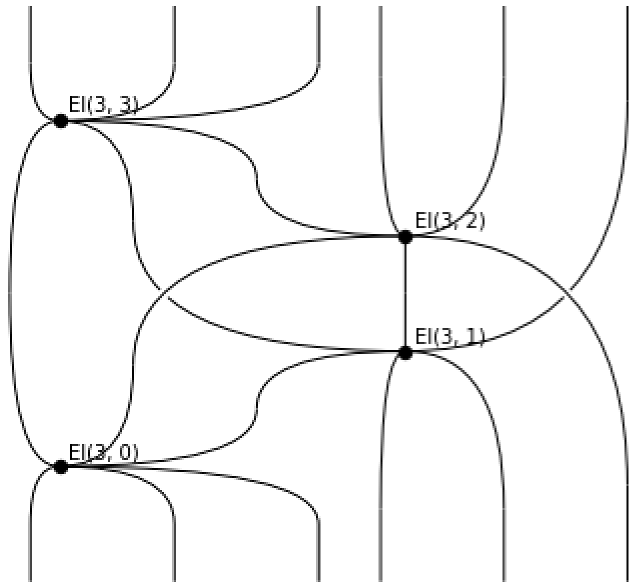
[32]:
```
tesseract = rewalt.Shape.cube(4)
tesseract.draw()
```



nbsphinx-code-borderwhite

As expected, it has four input faces and four output faces. Let's proceed as we did with the 4-simplex to understand what is happening.

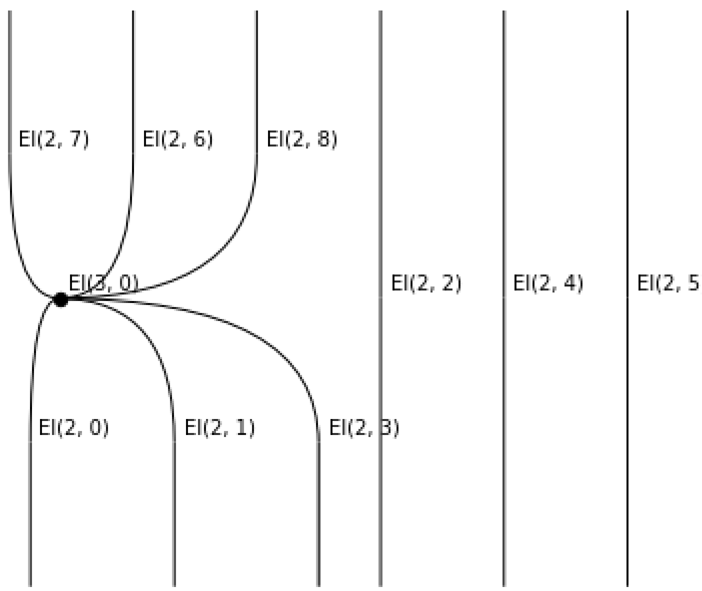
[33]:
```
tess_input = tesseract.input
tess_input.draw(wirelabels=False)
```



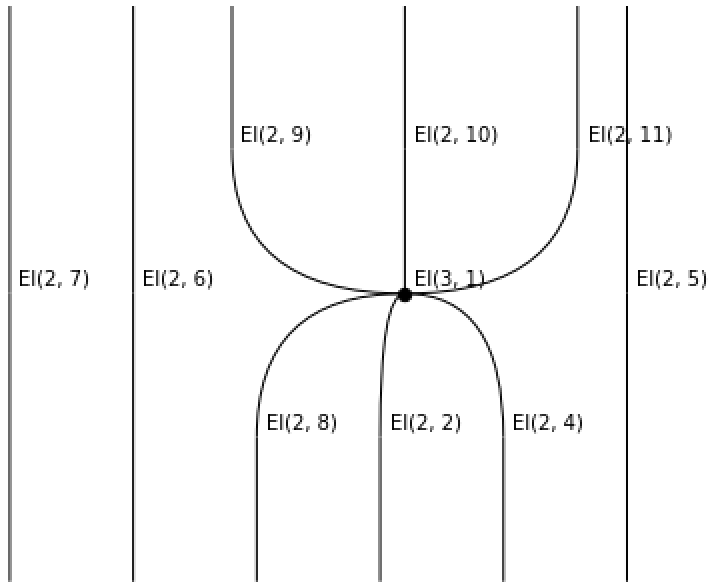
nbsphinx-code-borderwhite

(We have deactivated wire labels to make the image less crowded.)

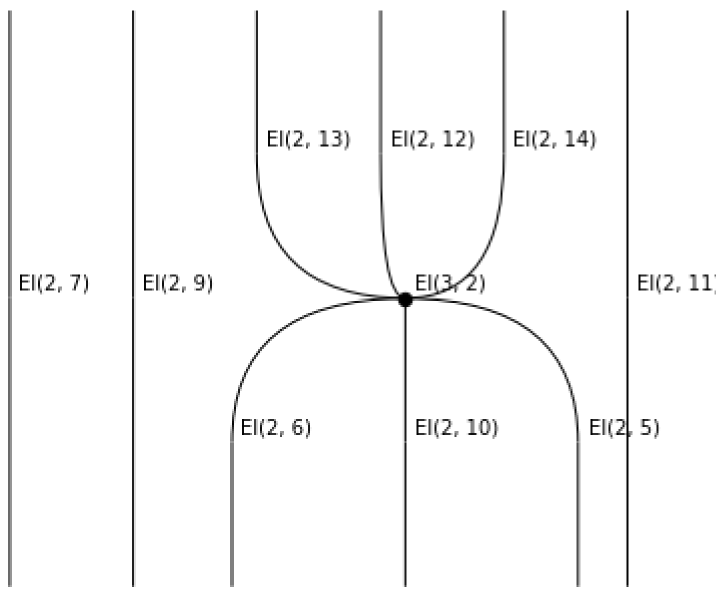
[34]:
```
tess_input.generate_layering()
rewalt.strdiags.draw(*tess_input.layers)
```



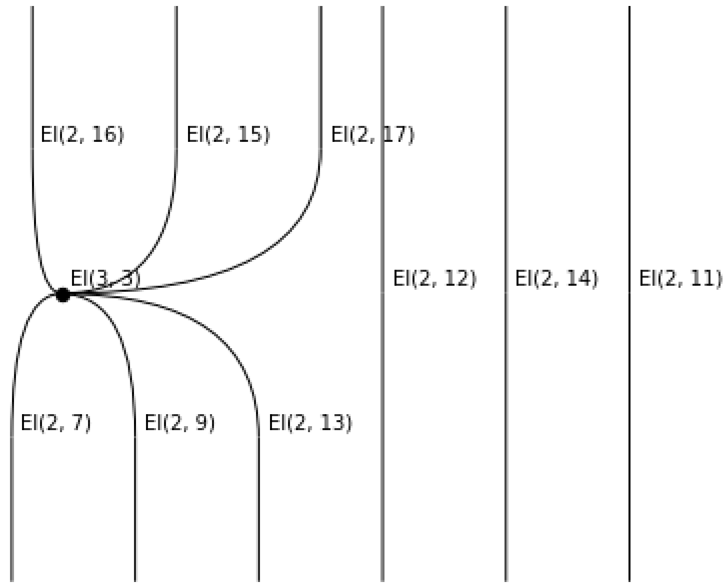
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

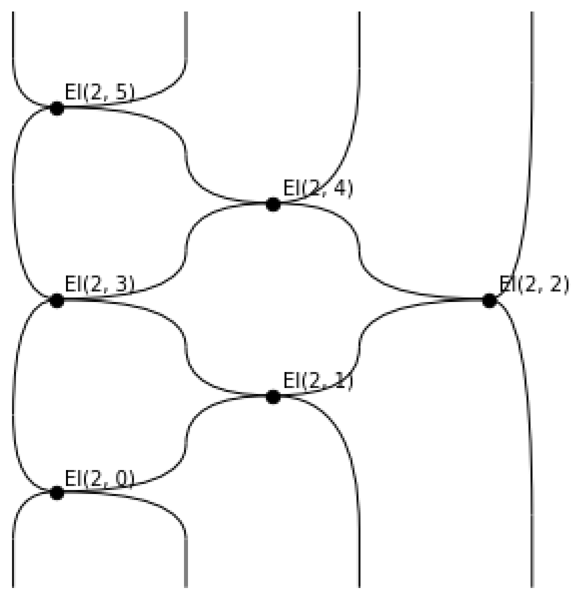


nbsphinx-code-borderwhite

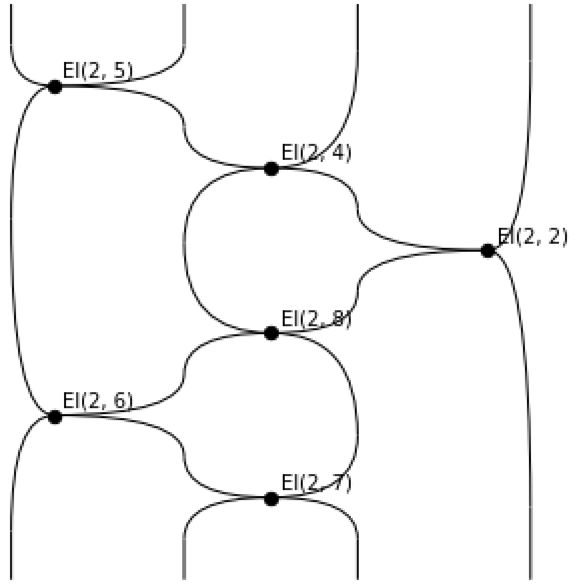


nbsphinx-code-borderwhite

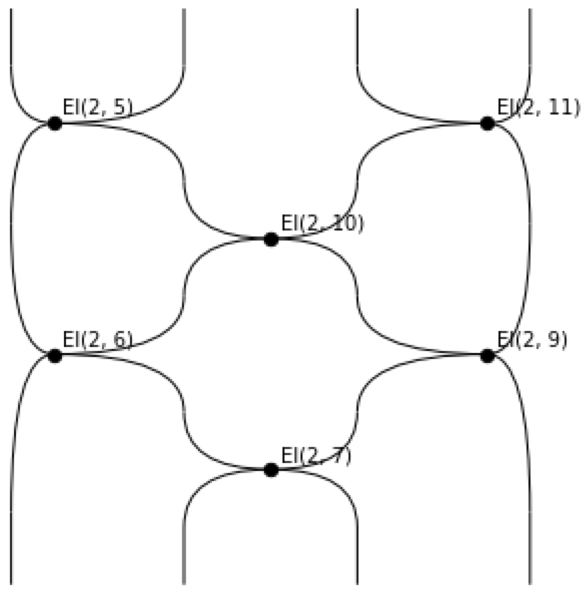
[35]:
```
rewalt.strdiags.draw(*tess_input.rewrite_steps, wirelabels=False)
```



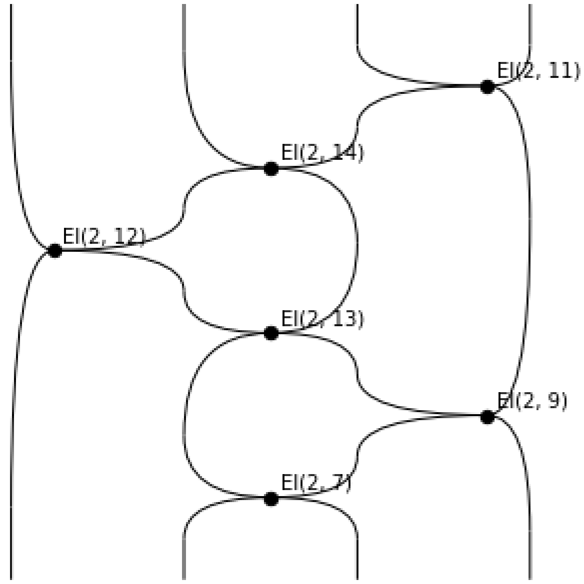
nbsphinx-code-borderwhite



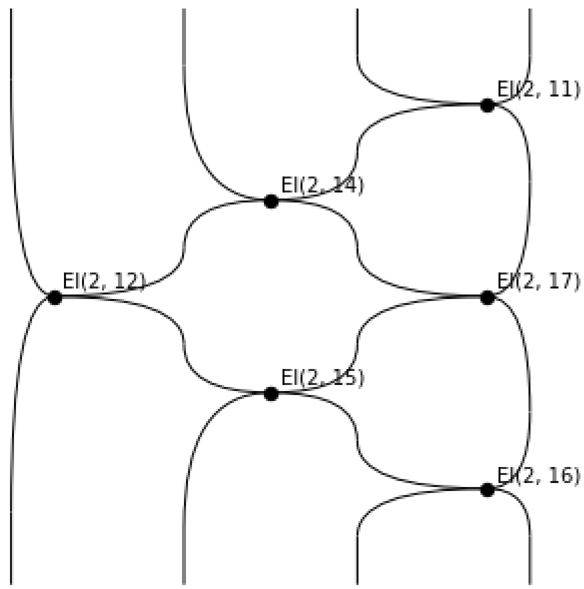
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



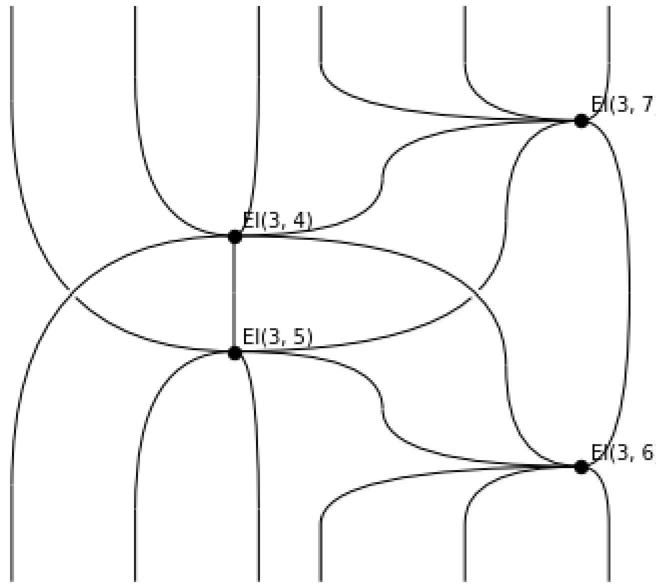
nbsphinx-code-borderwhite

Now we turn the sequence of rewrite steps into a gif.

[36]:
```
rewalt.strdiags.to_gif(
    *tess_input.rewrite_steps, loop=True,
    wirelabels=False,
    path='simplicescubes_3.gif')
```

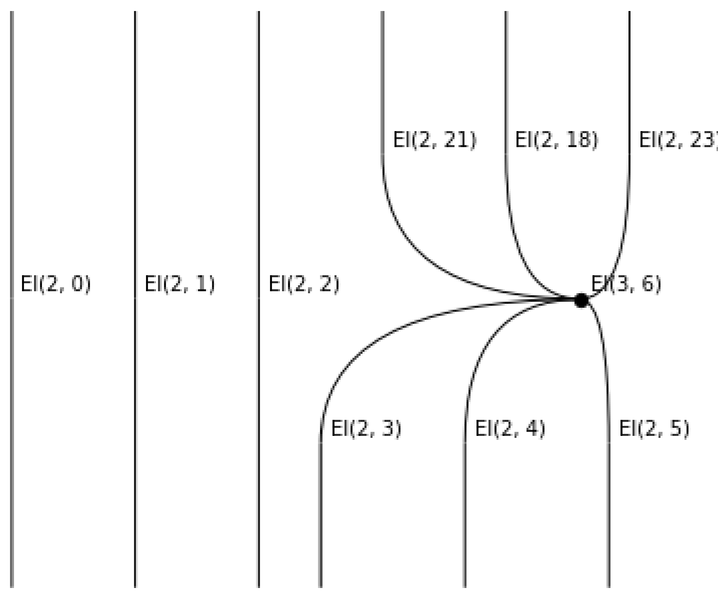
Next we focus on the output of the 4-cube.

[37]:
```
tess_output = tesseract.output
tess_output.draw(wirelabels=False)
```

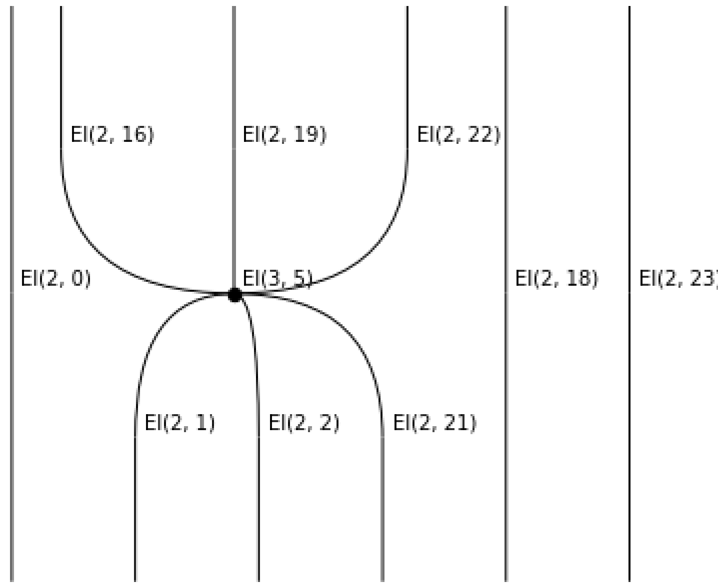


nbsphinx-code-borderwhite

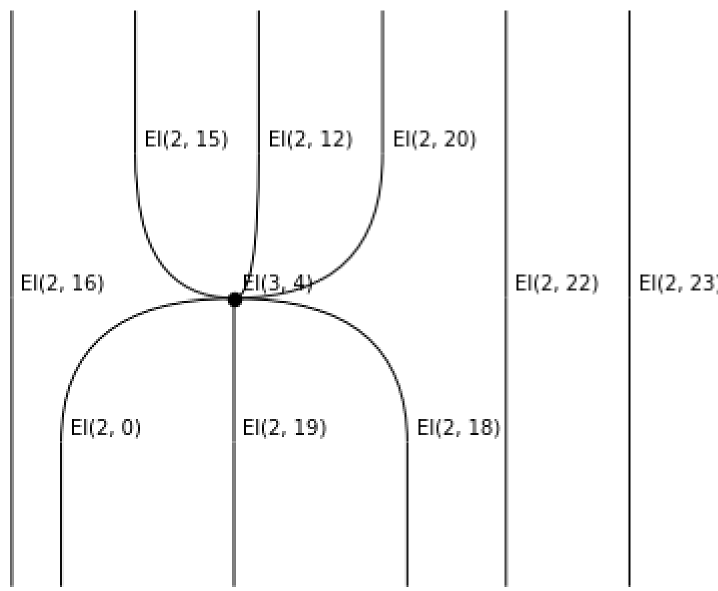
[38]:
```
tess_output.generate_layering()
rewalt.strdiags.draw(*tess_output.layers)
```



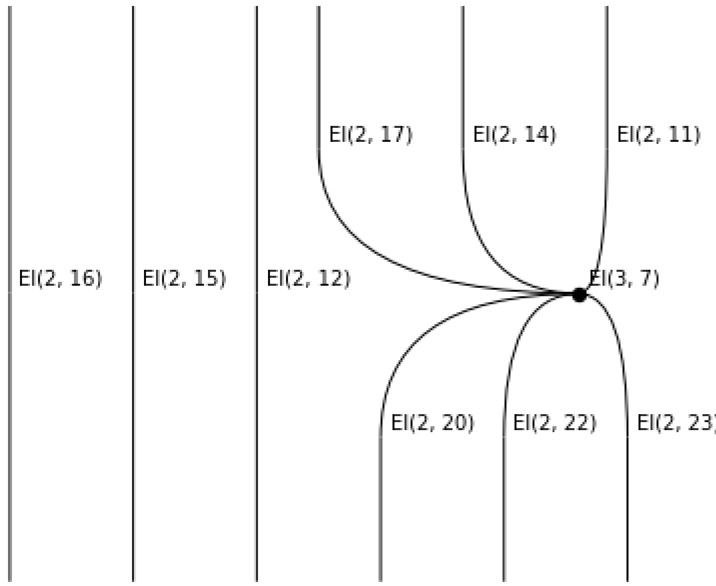
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

[39]:
```
rewalt.strdiags.to_gif(
    *tess_output.rewrite_steps, loop=True,
    wirelabels=False,
    path='simplicescubes_4.gif')
```

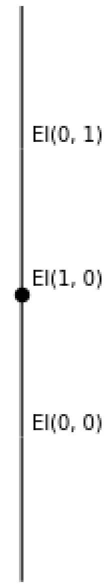
In the two rewrite sequences corresponding to the input and output boundary of the 4-cube, you may recognise the shapes of the two sides of the Zamolodchikov tetrahedron equation.

(Why "tetrahedron equation" if its shape is a 4-cube? Not sure!)

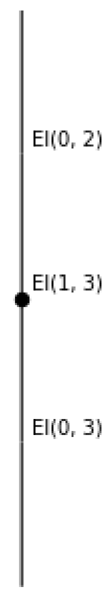
5.3.5 Maps of cubes

In contrast to simplices, faces of cubes are specified by two arguments: thinking of the n-cube as $[0, 1]^n$, one argument is an integer ranging from 0 to (n-1), specifying which coordinate to fix, and the other is a bit (for us, a *sign*: `'-'` or `'+'`) specifying whether to set the coordinate to 0 or to 1.

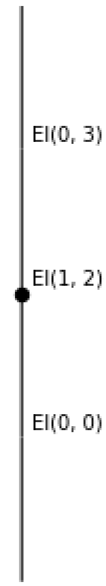
[40]:
```
for n in range(2):
    for sign in ('-', '+'):
        square.cube_face(n, sign).draw()
```



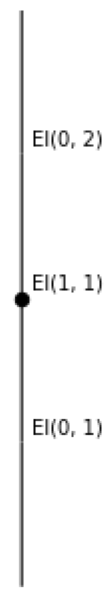
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite



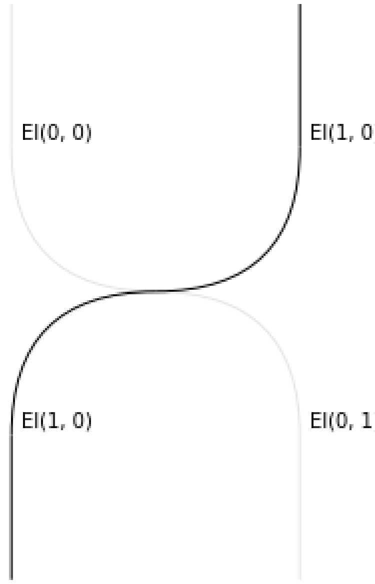
nbsphinx-code-borderwhite

Cubes also have two different kinds of “collapse” maps:

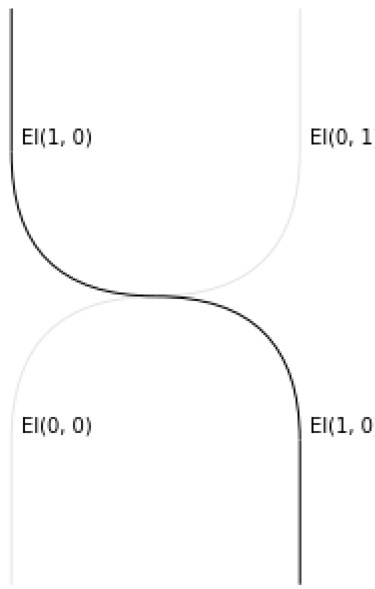
- *degeneracies*, which collapse the cube along a single coordinate (specified by an integer argument), and
- *connections*, which “fold” the cube along a pair of consecutive coordinates (specified by an integer argument), in two different ways (specified by a “sign” argument).

In `rewalt`, we can get a string-diagrammatic picture of these collapse maps.

[41]:
```
for n in range(2):
    arrow.cube_degeneracy(n).draw()
```

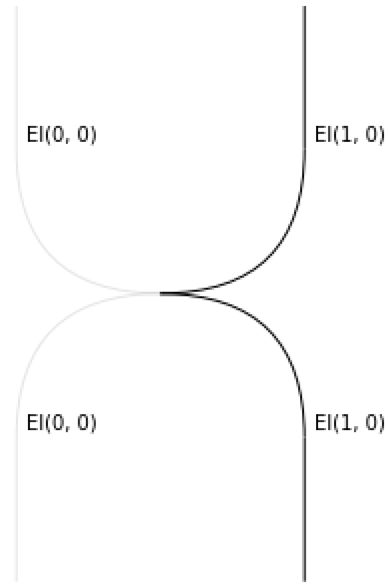


nbsphinx-code-borderwhite

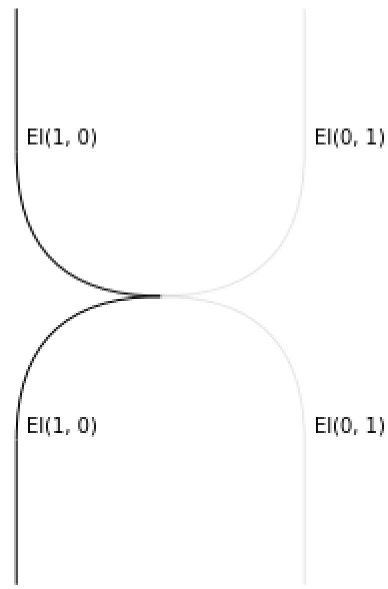


nbsphinx-code-borderwhite

[42]:
```
for sign in ('-', '+'):
    arrow.cube_connection(0, sign).draw()
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

As we saw in another notebook, being familiar with these degeneracies, which are neither "units" or "unitors", can be handy when constructing presentations of monoidal or higher algebraic theories.

5.3.6 Constructing a cubical set

Constructing a cubical set with connections is just like constructing a simplicial set, except we use the `add_cube` method instead of the `add_simplex` method when adding generators.

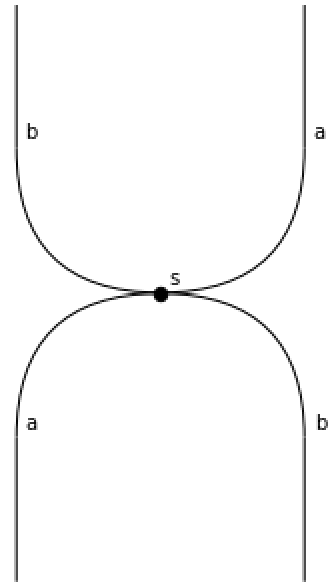
Let's construct a simple cubical model of the torus, with one 0-cell, two 1-cells, and one 2-cell.

[43]:
```
T = rewalt.DiagSet()
pt = T.add_cube('pt')
a = T.add_cube('a', pt, pt)
b = T.add_cube('b', pt, pt)
```

(continues on next page)

(continued from previous page)

```
s = T.add_cube('s', a, a, b, b)
s.draw()
```



nbsphinx-code-borderwhite

That's all! `T` is a torus.

We can check that the diagrammatic set we constructed is, indeed, a cubical set:

[44]:
```
T.iscubical
```

[44]:
```
True
```

Notice that if we look at this diagrammatic set as *string rewrite system* instead, it is a presentation of the free commutative monoid on the 2 generators `a` and `b`. Of course, the free *abelian group* on two generators is the first homology group of the torus.

5.3.7 Mixing them together

One of the reasons why simplices and cubes are "nice" families of shapes is that both are generated by the iteration of a binary operation, which defines a monoidal structure on their respective shape categories:

- simplices are iterated **joins** of points;
- cubes are iterated **products** of intervals.

In fact, both joins and products have "oriented" counterparts, and *all* shapes of `rewalt` are closed under both of these operations:

- the *join* of shapes, accessed either with the `join` method, or with the shift operators >> and <<, and
- the *Gray product* of shapes, accessed either with the `gray` method, or with the multiplication operator `*`.

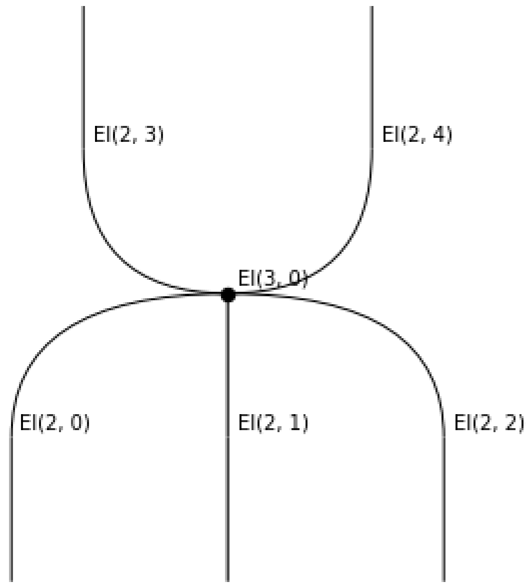
Indeed, this is how `rewalt` constructs oriented simplices and oriented cubes.

[45]:
```
assert arrow == point >> point
assert triangle == arrow >> point
assert square == arrow * arrow
assert cube == arrow * square
```

Joins are useful, for instance, for constructing *cones*, while products are useful for constructing *cylinders*. So the first operation is natural in a simplicial context, but not in a cubical context; while the second operation is natural in a cubical context but not in a simplicial context.

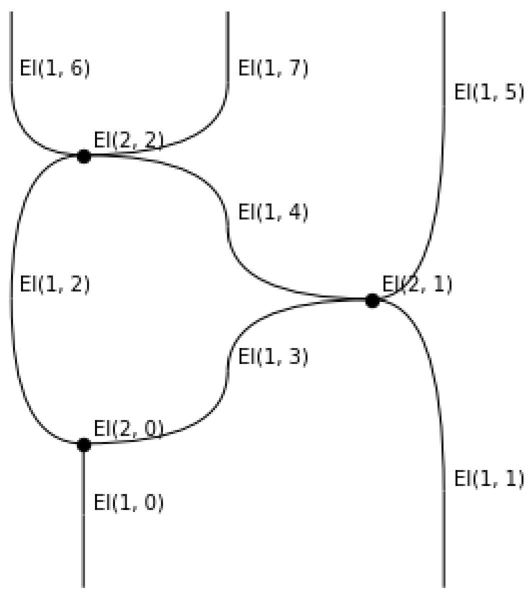
One nice thing about diagrammatic sets is that we do not need to choose! We can build a cylinder on a simplex…

[46]:
```
cylinder = arrow * triangle
cylinder.draw()
```

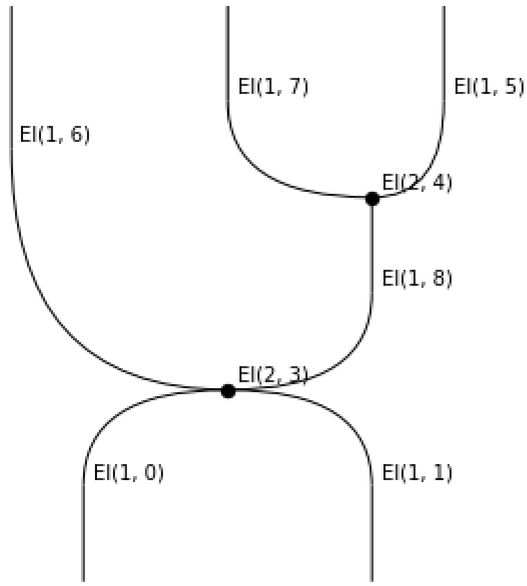


nbsphinx-code-borderwhite

[47]:
```
cylinder.draw_boundaries()
```



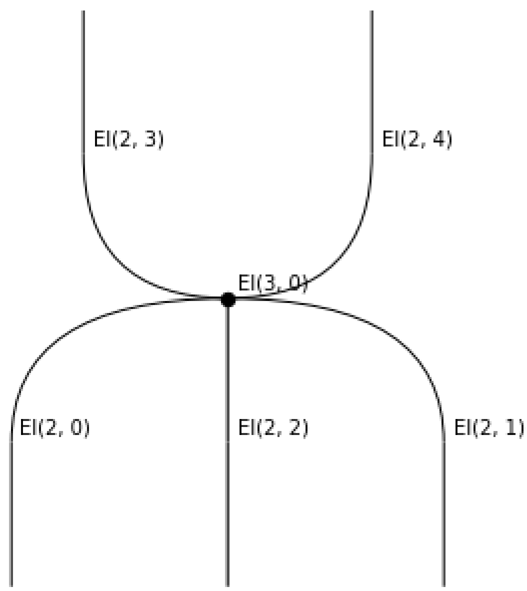
nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

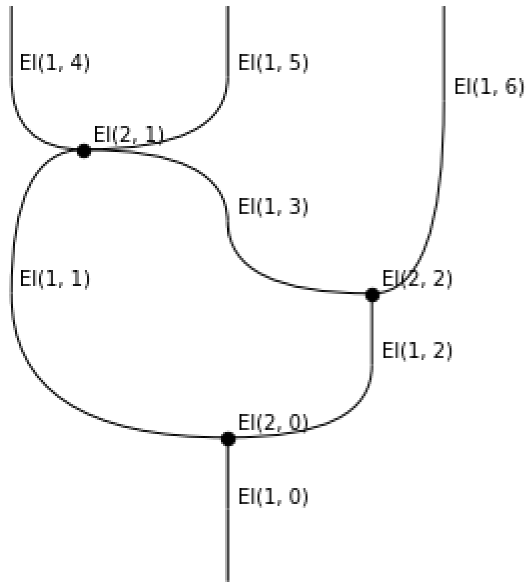
… and we can build a cone on a cube.

[48]:
```
cone = square >> point
cone.draw()
```

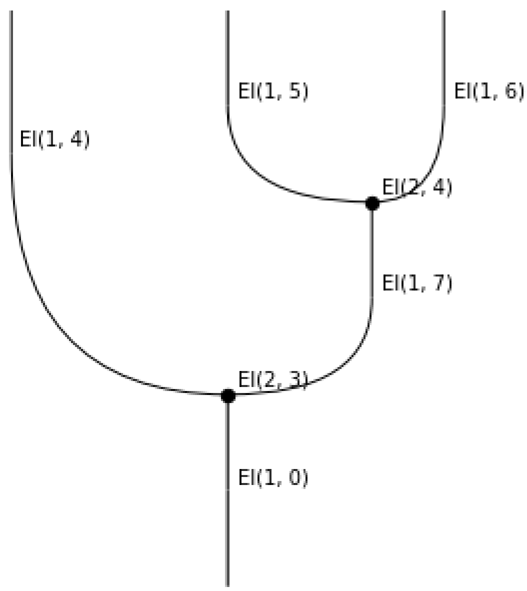


nbsphinx-code-borderwhite

[49]:
```
cone.draw_boundaries()
```



nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

5.4 The Eckmann–Hilton argument

A nice theoretical feature of `rewalt` is “topological soundness”: a diagrammatic set can be *geometrically realised* as a CW complex with one cell for each of its generators, and every diagram that we construct in the diagrammatic set corresponds to a valid homotopy in its realisation.

One of the first non-trivial homotopies that one encounters in algebraic topology are the “braiding” homotopies between two 2-cells, exhibiting the fact that π_2 of a space is always an *abelian* group. The construction of these homotopies is known as *Eckmann–Hilton argument*, and is also the basis of the identification of braided monoidal categories with “doubly degenerate” tricategories.

In this notebook, we will implement the Eckmann–Hilton argument in `rewalt`, by constructing both homotopies in a diagrammatic set with a single 0-dimensional generator and two 2-dimensional generators. Thanks to topological soundness, you can also see this as a *formal proof* of the usual homotopical Eckmann–Hilton.

First of all, let's create a diagrammatic set, and add all the generators. We will colour-code the two 2-cells, one in blue and one in magenta.

[1]:
```
import rewalt

EH = rewalt.DiagSet()
pt = EH.add('pt', draw_label=False)
a = EH.add('a', pt.unit(), pt.unit(), color='blue')
b = EH.add('b', pt.unit(), pt.unit(), color='magenta')
```

5.4.1 First braiding

The "braiding homotopies" will be made of degenerate cells, starting from the pasting "b after a", and ending in the pasting "a after b".

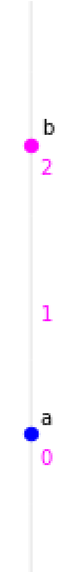
Our construction of these homotopies will be, essentially, an implementation of the "train tracks" proof by André Joyal and Joachim Kock. Let's start from the beginning.

[2]:
```
start = a.paste(b)
start.draw(nodepositions=True)
```

nbsphinx-code-borderwhite

Let's introduce some weak units between a and b; one would be sufficient, but we'll do two for reasons of symmetry.

[3]:
```
rew1 = start.rewrite(0, a.runitor('+'))
rew1.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

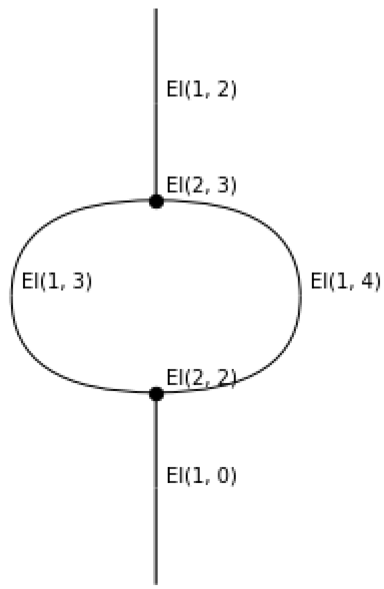
[4]:
```
rew2 = rew1.output.rewrite(2, b.lunitor('+'))
rew2.output.draw(nodepositions=True)
```

nbsphinx-code-borderwhite

Now, we want to "split" the units in positions (1, 2) into two "train tracks". This can be done with a "fully degenerate" cell over `pt`, of the appropriate shape:

[5]:
```
globe = rewalt.Shape.globe(2)
triangle = rewalt.Shape.simplex(2)

track_split_shape = globe.paste(globe).atom(triangle.paste(triangle.dual()))
track_split_shape.draw_boundaries()
```

nbsphinx-code-borderwhite



nbsphinx-code-borderwhite

You can see that `track_split_shape` is a 3-dimensional shape with input and output of the shape we desire, going from "single track" (pasting of two 2-globes) to "double track" (pasting of a 2-simplex with its dual).

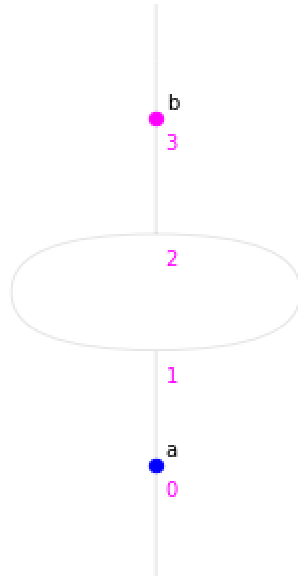
To get a "fully degenerate" cell over `pt` of shape `track_split_shape`, we use the `degeneracy` method.

[6]:
```
track_split = pt.degeneracy(track_split_shape)
track_split.draw_boundaries()
```

nbsphinx-code-borderwhite

nbsphinx-code-borderwhite

[7]:
```
rew3 = rew2.output.rewrite([1, 2], track_split)
rew3.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

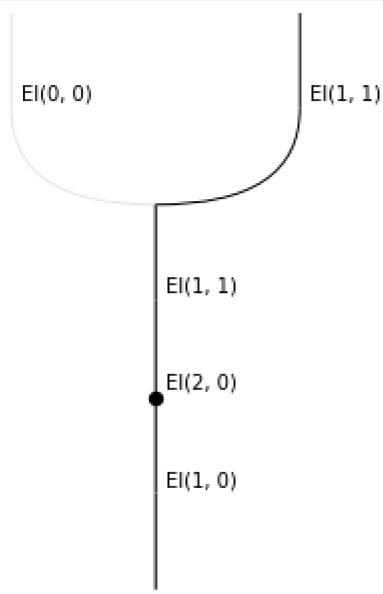
Now, our goal is to “move a to the right track, and move b to the left track”. This can be done with appropriate degenerate cells over a and b.

These degenerate cells are neither units or unitors. However, just like units and unitors, they can be obtained from pullbacks of a and b over particular collapse maps from a “partially collapsed cylinder” on their shape, as provided by the inflate method of the Shape class.

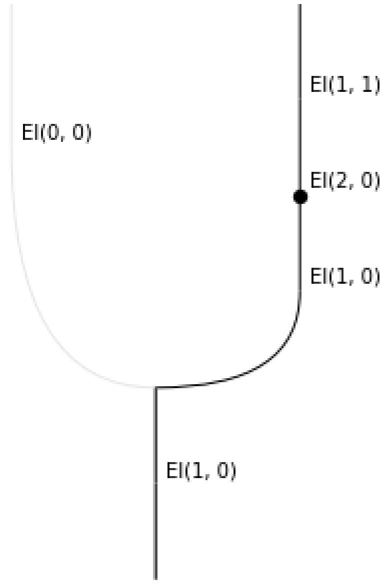
(I do not expect that this is particulary intuitive; you should try fiddling with inflate to get an idea of the collapses you can get.)

This, for example, is the map we can use to move a from the bottom to the right track.

[8]:
```
switch_br_map = globe.inflate(globe.all().boundary('+', 0))
switch_br_map.draw_boundaries()
```



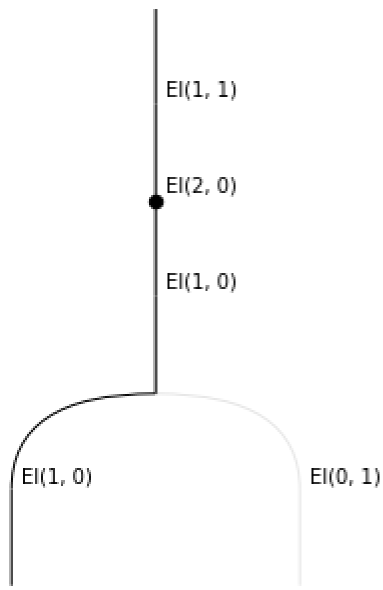
nbsphinx-code-borderwhite



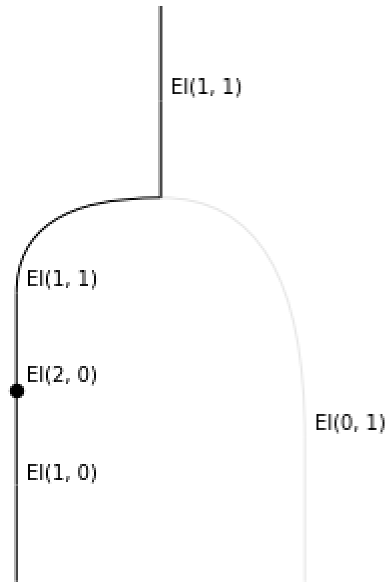
nbsphinx-code-borderwhite

Every other "switch" map we will get as a *dual* of this one. For example, the "top-to-left" that we need for `b` is the dual in dimensions 1 and 2 ("horizontal and vertical flip").

[9]:
```
switch_tl_map = switch_br_map.dual(1, 2)
switch_tl_map.draw_boundaries()
```

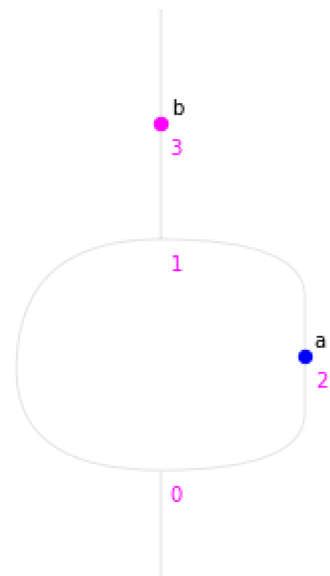


nbsphinx-code-borderwhite



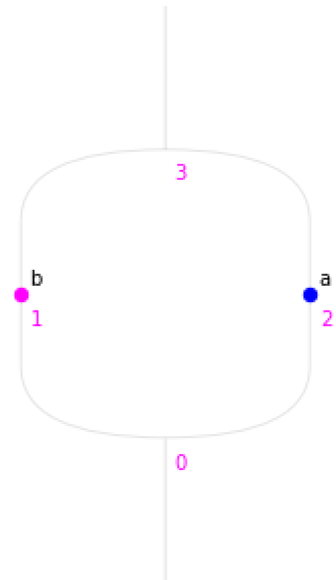
nbsphinx-code-borderwhite

[10]:
```
a_switch_br = a.pullback(switch_br_map)

rew4 = rew3.output.rewrite([0, 1], a_switch_br)
rew4.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

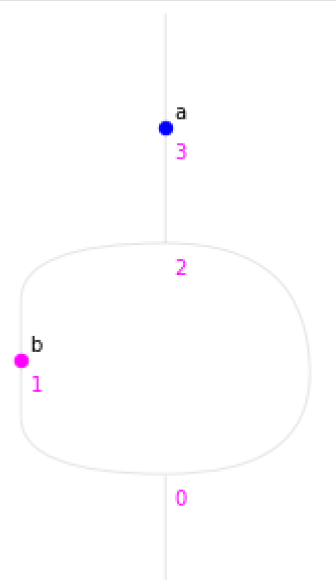
[11]:
```
b_switch_tl = b.pullback(switch_tl_map)

rew5 = rew4.output.rewrite([1, 3], b_switch_tl)
rew5.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

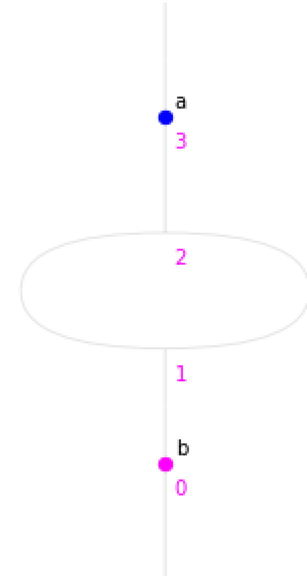
Now we will move a to the top, then b to the bottom. For that, we use pullbacks along other duals of our original "switch" map.

[12]:
```
switch_rt_map = switch_br_map.dual(2, 3)
a_switch_rt = a.pullback(switch_rt_map)

rew6 = rew5.output.rewrite([2, 3], a_switch_rt)
rew6.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

[13]:
```
switch_lb_map = switch_br_map.dual(1, 3)
b_switch_lb = b.pullback(switch_lb_map)

rew7 = rew6.output.rewrite([0, 1], b_switch_lb)
rew7.output.draw(nodepositions=True)
```

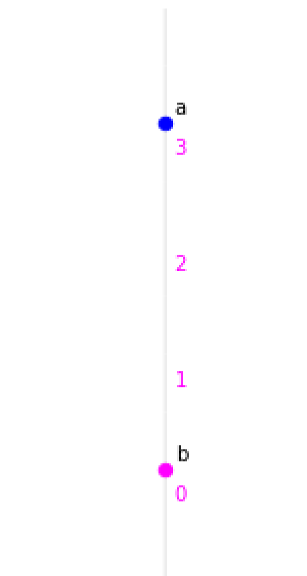


nbsphinx-code-borderwhite

The relative positions of a and b have been exchanged! Now we only need to get rid of the "train tracks" and other units between them.

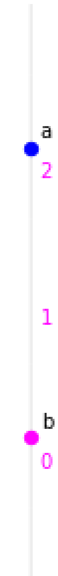
We used degenerate cells to introduce them, and degenerate cells are always "weakly invertible", so we can just use their "weak inverses", obtained with the `inverse` method.

[14]:
```
rew8 = rew7.output.rewrite([1, 2], track_split.inverse)
rew8.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

[15]:
```
rew9 = rew8.output.rewrite([0, 1], b.runitor('-'))
rew9.output.draw(nodepositions=True)
```



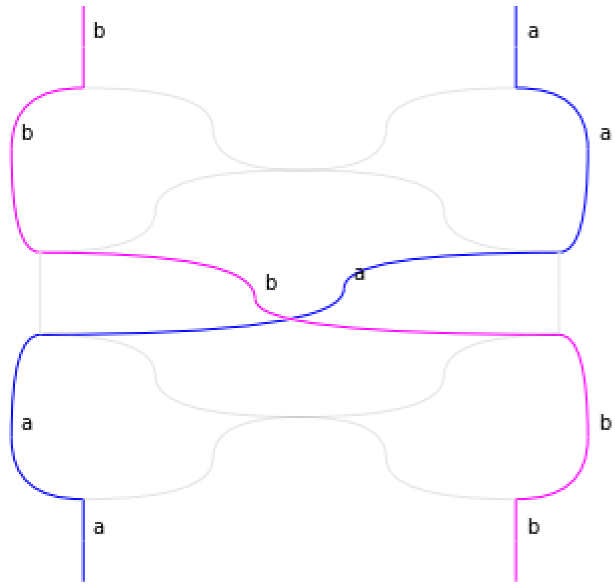
nbsphinx-code-borderwhite

[16]:
```
rew10 = rew9.output.rewrite([1, 2], a.lunitor('-'))
rew10.output.draw(nodepositions=True)
```

nbsphinx-code-borderwhite

We are done! Let's put all our rewrites together, and see what our proof looks like as a slice of a 3-dimensional diagram.

[17]:
```
eh1 = rewalt.Diagram.with_layers(
    rew1, rew2, rew3, rew4, rew5, rew6, rew7, rew8, rew9, rew10)
eh1.draw()
```



nbsphinx-code-borderwhite

See? It's a braiding where the b strand is passing over the a strand.

We can also assemble all our rewrites into a gif animation. We will also make it loop backwards.

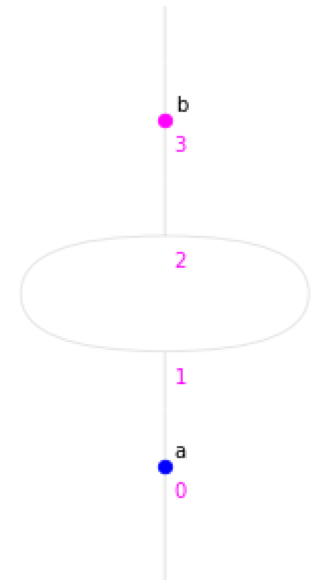
[18]:
```
rewalt.strdiags.to_gif(
    *eh1.rewrite_steps, degenalpha=0.2,
    loop=True, path='eckmannhilton_1.gif')
```

5.4.2 Second braiding

In our proof, we made the choice of moving a onto the right track, and b onto the left track; but we might as well have made a different choice. This would have led to a non-equivalent homotopy, the *dual* braiding.

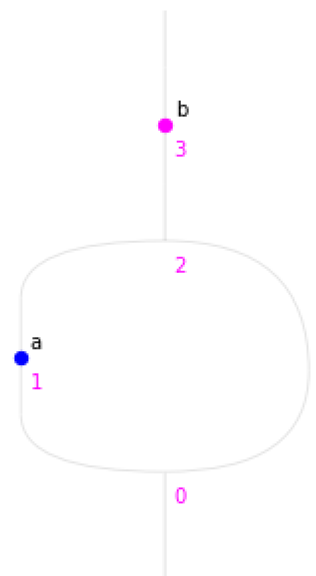
Let's go back to the step where we had the choice, and make a different one. This corresponds to "horizontally flipping" all the maps we used the first time.

[19]:
```
rew3.output.draw(nodepositions=True)
```



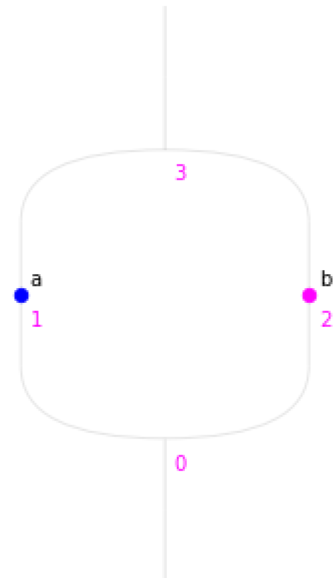
nbsphinx-code-borderwhite

[20]:
```
switch_bl_map = switch_br_map.dual(1)
a_switch_bl = a.pullback(switch_bl_map)

rew4d = rew3.output.rewrite([0, 1], a_switch_bl)
rew4d.output.draw(nodepositions=True)
```



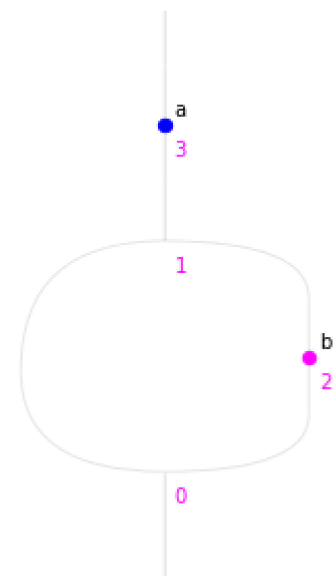
nbsphinx-code-borderwhite

[21]:
```
switch_tr_map = switch_tl_map.dual(1)
b_switch_tr = b.pullback(switch_tr_map)

rew5d = rew4d.output.rewrite([2, 3], b_switch_tr)
rew5d.output.draw(nodepositions=True)
```



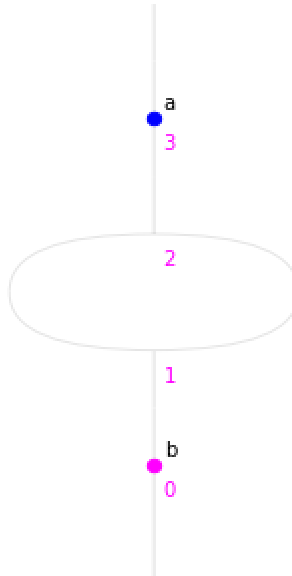
nbsphinx-code-borderwhite

[22]:
```
switch_lt_map = switch_rt_map.dual(1)
a_switch_lt = a.pullback(switch_lt_map)

rew6d = rew5d.output.rewrite([1, 3], a_switch_lt)
rew6d.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

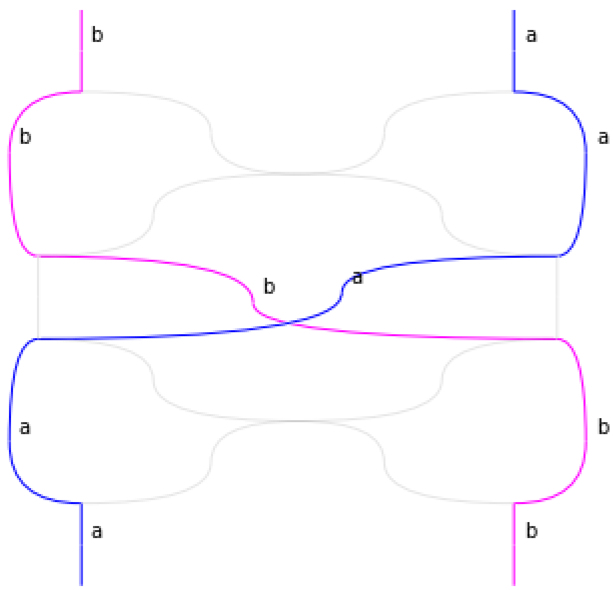
[23]:
```
switch_rb_map = switch_lb_map.dual(1)
b_switch_rb = b.pullback(switch_rb_map)

rew7d = rew6d.output.rewrite([0, 2], b_switch_rb)
rew7d.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

That's it; the last few steps are the same as the first time. Let's put the whole sequence together.

[24]:
```
eh2 = rewalt.Diagram.with_layers(
    rew1, rew2, rew3, rew4d, rew5d, rew6d, rew7d, rew8, rew9, rew10)
eh2.draw()
```



nbsphinx-code-borderwhite

See? Now it is the blue (a) strand that crosses over the magenta (b) strand.

And let's make another animation.

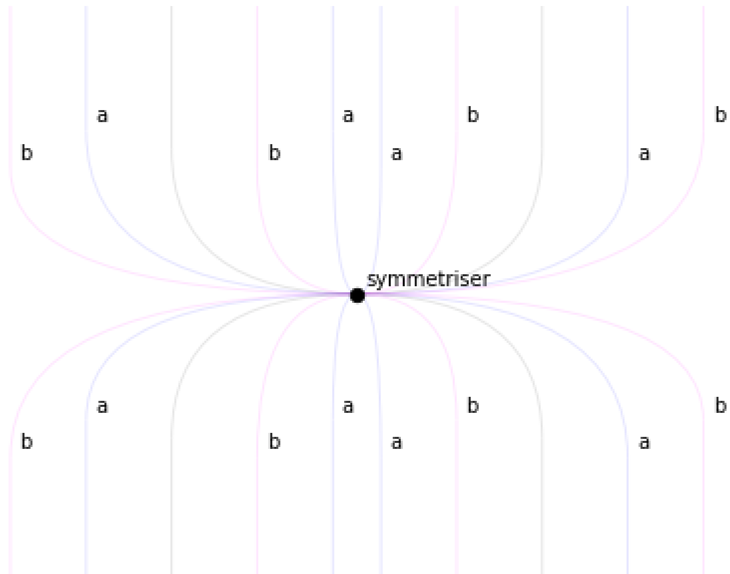
[25]:
```
rewalt.strdiags.to_gif(
    *eh2.rewrite_steps, degenalpha=0.2,
    loop=True, path='eckmannhilton_2.gif')
```

The diagrams eh1 and eh2 have the same input and output; they could, in principle, be the input and output of another cell.

By topological soundness, however, we know that there *isn't* a diagram between eh1 and eh2: the geometric realisation of EH is a bouquet of two 2-spheres, and in this space there isn't a homotopy between the two "braidings".

You are welcome to add one by hand, if you really want.

[26]:
```
symmetriser = EH.add('symmetriser', eh1, eh2)
symmetriser.draw()
```



nbsphinx-code-borderwhite

5.5 Presenting a category

The "higher-dimensional rewrite systems" that we construct in rewalt are interpretable in higher-dimensional categories, but they are, in general, different from higher-dimensional categories, in that they have no notion of *composition* of diagrams; that is, there's no way, in general, to "turn a diagram with many n-cells into a single n-cell".

Nevertheless, rewalt contains an implementation of a model of higher categories, in the form of diagrammatic sets with weak composites. This allows us to "declare" a cell to be the composite of a diagram; the composition is exhibited by a higher-dimensional *compositor* cell.

In this notebook, we will use the dedicated methods to construct a presentation of a simple finite category, consisting of a commuting square of four morphisms.

5.5.1 Adding all objects and morphisms

We start by creating an empty DiagSet, and adding all the objects and morphisms of our category. We have four objects (0-generators).

[1]:
```
import rewalt

C = rewalt.DiagSet()
```

(continues on next page)

(continued from previous page)

```
x0 = C.add('x0')
x1 = C.add('x1')
x2 = C.add('x2')
x3 = C.add('x3')
```

Then we add the four morphisms (1-generators) that form the boundary of our commuting square.

[2]:
```
f0 = C.add('f0', x0, x1)
f1 = C.add('f1', x1, x3)
g0 = C.add('g0', x0, x2)
g1 = C.add('g1', x2, x3)
```

Now we have two parallel diagrams of two 1-cells: `f0.paste(f1)` and `g0.paste(g1)`. We add the "diagonal" morphism that will be the composite of both diagrams.

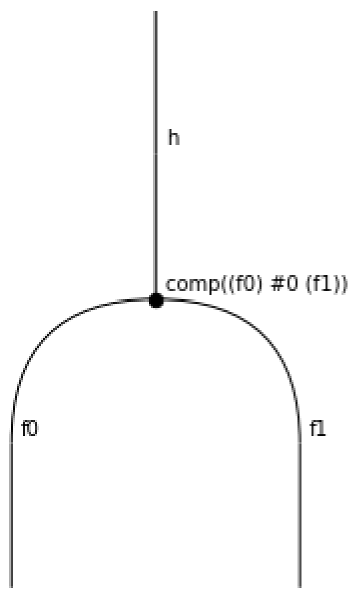
[3]:
```
h = C.add('h', x0, x3)
```

That's it; now we move on to the compositors.

5.5.2 Adding compositors

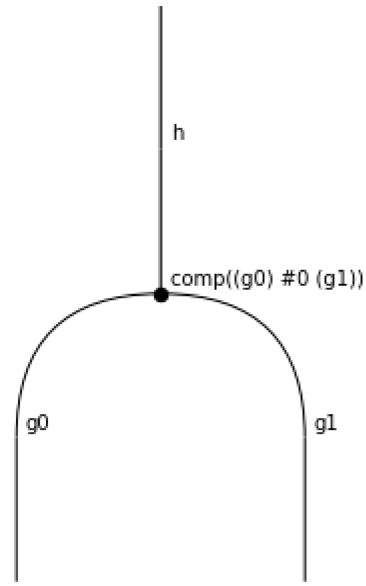
We declare a generator to be the "weak composite" of a diagram with the `make_composite` method. This will add a "compositor" 2-cell, and return it as a `Diagram` object.

[4]:
```
c_f = C.make_composite('h', f0.paste(f1))
c_f.draw()
```



nbsphinx-code-borderwhite

[5]:
```
c_g = C.make_composite('h', g0.paste(g1))
c_g.draw()
```



nbsphinx-code-borderwhite

We can check that a diagram has a composite with the `hascomposite` attribute; if a diagram has a composite, we can retrieve it with the `composite` attribute.

[6]:
```
f0.paste(f1).hascomposite
```

[6]:
```
True
```

[7]:
```
f0.paste(f1).composite == h
```

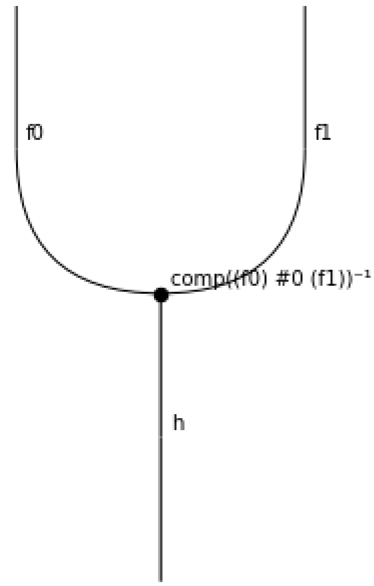
[7]:
```
True
```

A compositor allows us to rewrite a diagram into a cell. Now, according to the theory, to exhibit a genuine weak composite, the compositor would need to be *weakly invertible*.

As we saw in another notebook, since weak invertibility requires an infinite "tower" of cells, the approach of `rewalt` is to "invert only when needed". That also applies to compositors, which are created in "one direction only", and must be explicitly inverted if needed.

(Another reason to not invert by default is that one may want to use `DiagSet` objects to implement different kinds of higher structures, such as representable multicategories or "lax" versions thereof, where it is important that compositors only go "one way".)

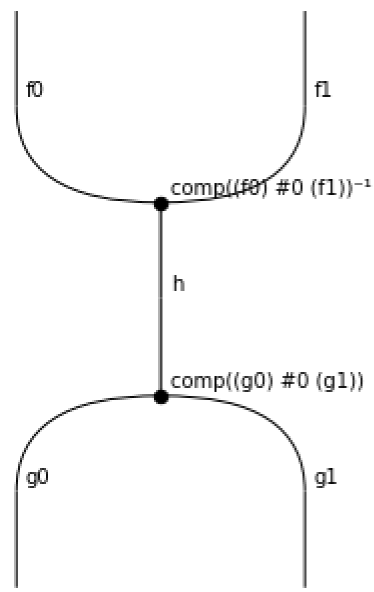
[8]:
```
c_f_inv, c_f_rinvertor, c_f_linvertor = C.invert(c_f)
c_f_inv.draw()
```



nbsphinx-code-borderwhite

Now that we have an inverse compositor, we can “rewrite” `g0.paste(g1)` into `f0.paste(f1)` via their shared composite.

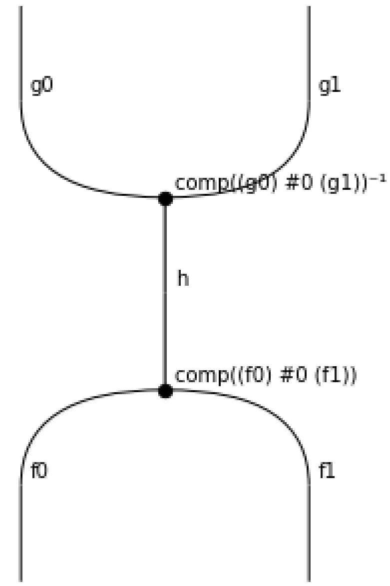
[9]:
```
g_to_f = c_g.paste(c_f_inv)
g_to_f.draw()
```



nbsphinx-code-borderwhite

To go the other way around, we need to invert the compositor for `g0.paste(g1)`.

[10]:
```
c_g_inv, c_g_rinvertor, c_g_linvertor = C.invert(c_g)

f_to_g = c_f.paste(c_g_inv)
f_to_g.draw()
```

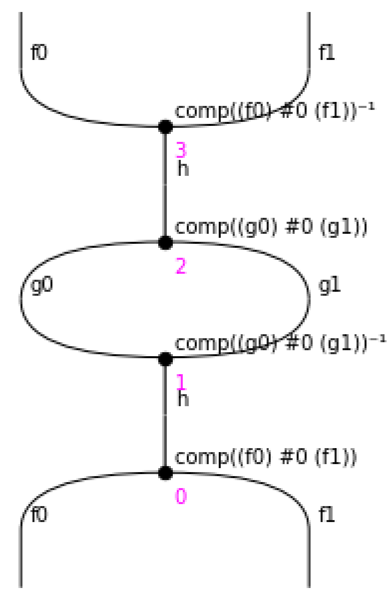


nbsphinx-code-borderwhite

This pair of diagrams "embodies" the commuting square with sides `f0`, `f1`, `g0`, `g1`.

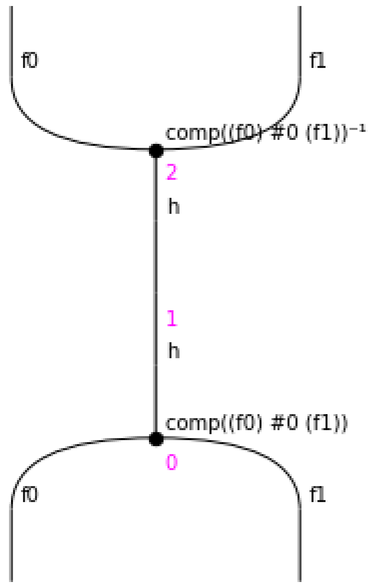
We can use the "invertors" to show that the two diagrams are each other's weak inverse.

[11]:
```
f_to_g.paste(g_to_f).draw(nodepositions=True)
```



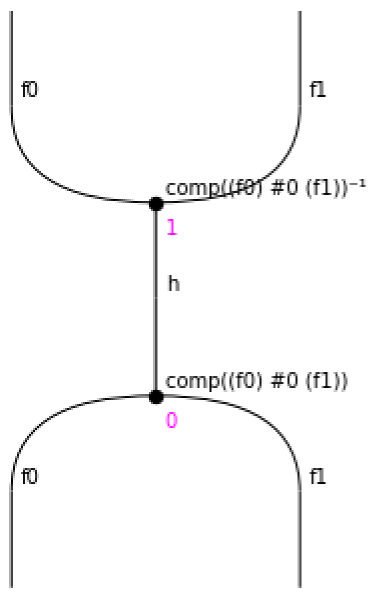
nbsphinx-code-borderwhite

[12]:
```
rew1 = f_to_g.paste(g_to_f).rewrite([1, 2], c_g_linvertor)
rew1.output.draw(nodepositions=True)
```



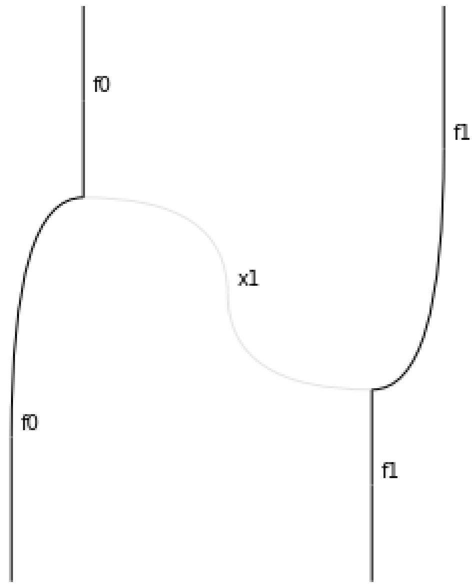
nbsphinx-code-borderwhite

[13]:
```
rew2 = rew1.output.rewrite([0, 1], c_f.runitor('-'))
rew2.output.draw(nodepositions=True)
```



nbsphinx-code-borderwhite

[14]:
```
rew3 = rew2.output.rewrite([0, 1], c_f_rinvertor)
rew3.output.draw()
```



nbsphinx-code-borderwhite

5.5.3 Composites involving units

Now in `C` all 1-dimensional diagrams have composites, so we can see `C` as a category.

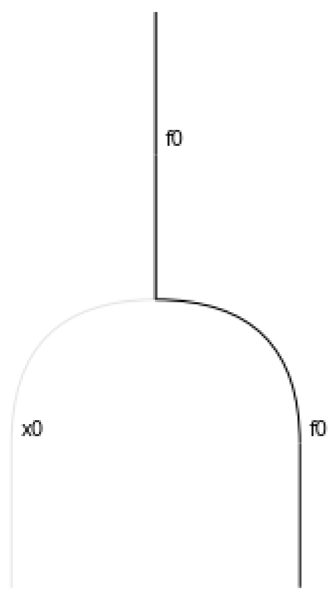
Except, in fact, not all 1-dimensional diagrams have composites that `C` *knows of*!

[15]:
```
x0.unit().paste(f0).hascomposite
```

[15]:
```
False
```

Nevertheless, we can certainly turn this diagram into a single cell, using the left unitor for `f0`.

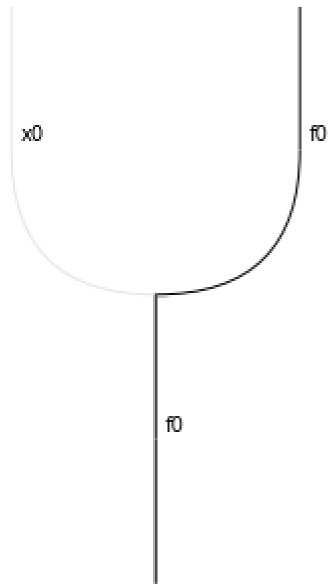
[16]:
```
f0.lunitor('-').draw()
```



nbsphinx-code-borderwhite

This is even already "weakly invertible", as all degenerate cells are.

```
[17]: f0.lunitor('-').inverse.draw()
```



nbsphinx-code-borderwhite

So why does `rewalt` not consider unitors to be compositors?

There is a good reason: `rewalt` does *not* make a distinction between presentations of categories, bicategories, or n-categories for any other n. And there are certainly non-strict bicategories in which the composite of a 1-cell with a unit is *not* equal to the 1-cell.

So if we want `C` to know that `f0` is, indeed, the composite of `x0.unit()` and `f0`, we need to make it explicit.

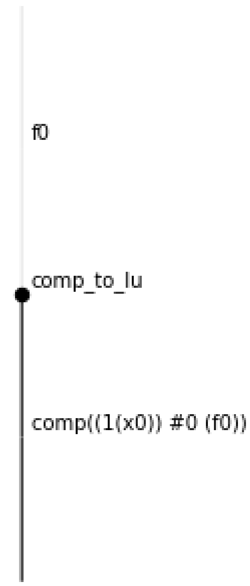
```
[18]: c_x0_f0 = C.make_composite('f0', x0.unit().paste(f0))
```

This will add a compositor which is *not* the same as the left unitor on `f0`.

(The reason you cannot declare an existing degenerate cell to be a compositor is that `rewalt` wants compositors to be generators, so it can remember which compositors a `DiagSet` contains just by their list of names).

So if we want to "equate" the compositor to the unitor, we have to do it "weakly", by adding a 3-cell between them.

```
[19]: comp_to_lu = C.add('comp_to_lu', c_x0_f0, f0.lunitor('-'))
      comp_to_lu.draw()
```



nbsphinx-code-borderwhite

5.6 diagrams

Implements diagrammatic sets and diagrams.

rewalt.diagrams.DiagSet()	Class for diagrammatic sets, a model of higher-dimensional rewrite systems and/or directed cell complexes.
rewalt.diagrams.Diagram(ambient)	Class for diagrams, that is, mappings from a shape to an "ambient" diagrammatic set.
rewalt.diagrams.SimplexDiagram(ambient)	Subclass of `Diagram` for diagrams whose shape is an oriented simplex.
rewalt.diagrams.CubeDiagram(ambient)	Subclass of `Diagram` for diagrams whose shape is an oriented cube.
rewalt.diagrams.PointDiagram(ambient)	Subclass of `Diagram` for diagrams whose shape is a point.

5.6.1 diagrams.DiagSet

class `rewalt.diagrams.DiagSet`

Bases: `object`

Class for diagrammatic sets, a model of higher-dimensional rewrite systems and/or directed cell complexes.

A diagrammatic set is constructed by creating an empty object, then adding named *generators* of different dimensions. The addition of a generator models the gluing of an atomic `shapes.Shape` object along its boundary.

This operation produces a *diagram*, that is, a map from a shape to the diagrammatic set, modelled as a `Diagram` object. From these "basic" diagrams, we can construct "derived" diagrams either by pasting, or by pulling back along shape maps (this is used to produce "unit" or "degenerate" diagrams).

To add a 0-dimensional generator (a point), we just give it a name. In the main constructor `add()`, the gluing of an `n`-dimensional generator is specified by a pair of round, `(n-1)`-dimensional `Diagram` objects, describing the gluing maps for the input and output boundaries of a shape.

Simplicial sets, cubical sets with connections, and reflexive globular sets are all special cases of diagrammatic sets, where the generators have simplicial, cubical, or globular shapes. There are special constructors `add_simplex()` and `add_cube()` for adding simplicial and cubical generators by listing all their faces.

The generators of a diagrammatic set are, by default, "directed" and not invertible. The class supports a model of weak or pseudo- invertibility, where two generators being each other's "weak inverse" is witnessed by a pair of higher-dimensional generators (*invertors*). This is produced by the methods `invert()` (creates an inverse) and `make_inverses()` (makes an existing generator the inverse).

Diagrammatic sets do not have an intrinsic notion of composition of diagrams, so they are not by themselves a model of higher categories. However, the class supports a model of higher categories in which one generator being the composite of a diagram is witnessed by a higher-dimensional generator (a *compositor*). This is produced by the methods `compose()` (creates a composite) and `make_composite()` (makes an existing generator the composite).

Notes

There is an alternative constructor `yoneda()` which turns a `shapes.Shape` object into a diagrammatic set with one generator for every face of the shape.

Methods

`add`(name[, input, output])	Adds a generator and returns the diagram that maps the new generator into the diagrammatic set.
`add_cube`(name, *faces, **kwargs)	Variant of `add()` for cube-shaped generators.
`add_simplex`(name, *faces, **kwargs)	Variant of `add()` for simplex-shaped generators.
`compose`(diagram[, name, compositorname])	Given a round diagram, adds a weak composite for it, together with a compositor witnessing the composition, and returns them as diagrams.
`copy`()	Returns a copy of the object.
`invert`(generatorname[, inversename, ...])	Adds a weak inverse for a generator, together with left and right invertors that witness the inversion, and returns them as diagrams.
`make_composite`(generatorname, diagram[, ...])	Given a generator and a round diagram, it makes the first the weak composite of the second by adding a compositor, and returns the compositor as a diagram.
`make_inverses`(generatorname1, generatorname2)	Makes two generators each other's weak inverse by adding invertors, and returns the invertors.
`remove`(generatorname)	Removes a generator, together with all other generators that depend on it.
`update`(generatorname, **kwargs)	Updates the optional arguments of a generator.
`yoneda`(shape)	Alternative constructor creating a diagrammatic set from a `shapes.Shape`.

Attributes

by_dim	Returns the set of generators indexed by dimension.
compositors	Returns a dictionary of diagrams that have a non-trivial composite, indexed by their compositor's name.
dim	Returns the maximal dimension of a generator.
generators	Returns the object's internal representation of the set of generators and related data.
iscubical	Returns whether the diagrammatic sets is cubical, that is, all its generators are cube-shaped.
issimplicial	Returns whether the diagrammatic sets is simplicial, that is, all its generators are simplex-shaped.

property generators

Returns the object's internal representation of the set of generators and related data.

This is a dictionary whose keys are the generators' names. For each generator, the object stores another dictionary, which contains at least

- the generator's shape (`shape`, `shapes.Shape`),
- the mapping of the shape (`mapping`, `list[list[hashable]]`),
- the generator's set of "faces", that is, other generators appearing as codimension-1 faces of the generator (`faces`, `set[hashable]`),
- the generator's set of "cofaces", that is, other generators that have the generator as a face (`cofaces`, `set[hashable]`).

If the generator has been inverted, it will also contain

- its inverse's name (`inverse`, `hashable`),
- the left invertor's name (`linvertor`, `hashable`),
- the right invertor's name (`rinvertor`, `hashable`).

If the generator happens to be a compositor, it will also contain the name of the composite it is exhibiting (`composite`, `hashable`).

This also stores any additional keyword arguments passed when adding the generator.

Returns
generators – The generators data.

Return type
`dict[dict]`

property by_dim

Returns the set of generators indexed by dimension.

Returns
by_dim – The set of generators indexed by dimension.

Return type
`dict[hashable]`

property compositors

Returns a dictionary of diagrams that have a non-trivial composite, indexed by their compositor's name.

More precisely, rather than *Diagram* objects, the dictionary stores the `shape` and `mapping` data that allows to reconstruct them.

Returns
compositors – The dictionary of composed diagrams.

Return type
`dict[dict]`

property dim

Returns the maximal dimension of a generator.

Returns
dim – The maximal dimension of a generator, or `-1` if empty.

Return type
`int`

property issimplicial

Returns whether the diagrammatic sets is simplicial, that is, all its generators are simplex-shaped.

Returns
issimplicial – `True` if and only if the shape of every generator is a `shapes.Simplex` object.

Return type
`bool`

property iscubical

Returns whether the diagrammatic sets is cubical, that is, all its generators are cube-shaped.

Returns
iscubical – `True` if and only if the shape of every generator is a `shapes.Cube` object.

Return type
`bool`

add(*name*, *input=None*, *output=None*, ***kwargs*)

Adds a generator and returns the diagram that maps the new generator into the diagrammatic set.

The gluing of the generator is specified by a pair of round diagrams with identical boundaries, corresponding to the input and output diagrams of the new generator. If none are given, adds a point (0-dimensional generator).

Parameters

- **name** (`hashable`) – Name to assign to the new generator.
- **input** (*Diagram*, optional) – The input diagram of the new generator (default is `None`)
- **output** (*Diagram*, optional) – The output diagram of the new generator (default is `None`)

Keyword Arguments

- **color** (*multiple types*) – Fill color when pictured as a node in string diagrams. If `stroke` is not specified, this is also the color when pictured as a wire.
- **stroke** (*multiple types*) – Stroke color when pictured as a node, and color when pictured as a wire.
- **draw_node** (`bool`) – If `False`, no node is drawn when picturing the generator in string diagrams.

- **draw_label** (`bool`) – If `False`, no label is drawn when picturing the generator in string diagrams.

Returns
generator – The diagram picking the new generator.

Return type
Diagram

Raises
ValueError – If the name is already in use, or the input and output diagrams do not have round and matching boundaries.

add_simplex(*name*, **faces*, ***kwargs*)

Variant of *add()* for simplex-shaped generators.

The gluing of the generator is specified by a number of *SimplexDiagram* objects, corresponding to the faces of the new generator as listed by *SimplexDiagram.simplex_face*.

Parameters

- **name** (`hashable`) – Name to assign to the new generator.
- ***faces** (*SimplexDiagram*) – The simplicial faces of the new generator.

Keyword Arguments
****kwargs** – Same as *add()*.

Returns
generator – The diagram picking the new generator.

Return type
SimplexDiagram

Raises
ValueError – If the name is already in use, or the faces do not have matching boundaries.

add_cube(*name*, **faces*, ***kwargs*)

Variant of *add()* for cube-shaped generators.

The gluing of the generator is specified by a number of *CubeDiagram* objects, corresponding to the faces of the new generator as listed by *CubeDiagram.cube_face*, in the order `(0, '-')`, `(0, '+')`, `(1, '-')`, `(1, '+')`, etc.

Parameters

- **name** (`hashable`) – Name to assign to the new generator.
- ***faces** (*CubeDiagram*) – The cubical faces of the new generator.

Keyword Arguments
****kwargs** – Same as *add()*.

Returns
generator – The diagram picking the new generator.

Return type
CubeDiagram

Raises
ValueError – If the name is already in use, or the faces do not have matching boundaries.

`invert`(*generatorname*, *inversename=None*, *rinvertorname=None*, *linvertorname=None*, ***kwargs*)

Adds a weak inverse for a generator, together with left and right invertors that witness the inversion, and returns them as diagrams.

Both the inverse and the invertors can be given custom names. If the generator to be inverted is named `'a'`, the default names are

- `'a¹'` for the inverse,
- `'inv(a, a¹)'` for the right invertor,
- `'inv(a¹, a)'` for the left invertor.

In the theory of diagrammatic sets, weak invertibility would correspond to the situation where the invertors themselves are weakly invertible, coinductively. In the implementation, we take an "invert when necessary" approach, where invertors are not invertible by default, and should be inverted when needed.

Notes

The right invertor for the generator is the left invertor for its inverse, and the left invertor for the generator is the right invertor for its inverse.

Parameters

- **generatorname** (`hashable`) – Name of the generator to invert.
- **inversename** (`hashable`, optional) – Name assigned to the inverse.
- **rinvertorname** (`hashable`, optional) – Name assigned to the right invertor.
- **linvertorname** (`hashable`, optional) – Name assigned to the left invertor.

Keyword Arguments

****kwargs** – Passed to *add()* when adding the inverse.

Returns

- **inverse** (*Diagram*) – The diagram picking the inverse.
- **rinvertor** (*Diagram*) – The diagram picking the right invertor.
- **linvertor** (*Diagram*) – The diagram picking the left invertor.

Raises

ValueError – If the generator is already inverted, or 0-dimensional.

`make_inverses`(*generatorname1*, *generatorname2*, *rinvertorname=None*, *linvertorname=None*)

Makes two generators each other's weak inverse by adding invertors, and returns the invertors.

In what follows, "right/left" invertors are relative to the first generator. Both invertors can be given custom names. If the generators are named `'a'`, `'b'`, the default names for the invertors are

- `'inv(a, b)'` for the right invertor,
- `'inv(b, a)'` for the left invertor.

In the theory of diagrammatic sets, weak invertibility would correspond to the situation where the invertors themselves are weakly invertible, coinductively. In the implementation, we take an "invert when necessary" approach, where invertors are not invertible by default, and should be inverted when needed.

Parameters

- **generatorname1** (`hashable`) – Name of the first generator.
- **generatorname2** (`hashable`, optional) – Name of the second generator.

- **rinvertorname** (hashable, optional) – Name assigned to the right invertor.
- **linvertorname** (hashable, optional) – Name assigned to the left invertor.

Returns

- **rinvertor** (*Diagram*) – The diagram picking the right invertor.
- **linvertor** (*Diagram*) – The diagram picking the left invertor.

Raises

ValueError – If the generators are already inverted, or 0-dimensional, or do not have compatible boundaries.

compose(*diagram*, *name=None*, *compositorname=None*, ***kwargs*)

Given a round diagram, adds a weak composite for it, together with a compositor witnessing the composition, and returns them as diagrams.

Both the composite and the compositor can be given custom names. If the diagram to be composed is named `'a'`, the default names are

- `'a'` for the composite,
- `'comp(a)'` for the compositor.

In the theory of diagrammatic sets, a weak composite is witnessed by a weakly invertible compositor. In the implementation, we take an "invert when necessary" approach, where compositors are not invertible by default, and should be inverted when needed.

Notes

A cell (a diagram whose shape is an atom) is treated as already having itself as a composite, witnessed by a unit cell; this method can only be used on non-atomic diagrams.

Parameters

- **diagram** (*Diagram*) – The diagram to compose.
- **name** (hashable, optional) – Name of the weak composite.
- **compositorname** (hashable, optional) – Name of the compositor.

Keyword Arguments

****kwargs** – Passed to *add()* when adding the composite.

Returns

- **composite** (*Diagram*) – The diagram picking the composite.
- **compositor** (*Diagram*) – The diagram picking the compositor.

Raises

ValueError – If the diagram is not round, or already has a composite.

make_composite(*generatorname*, *diagram*, *compositorname=None*)

Given a generator and a round diagram, it makes the first the weak composite of the second by adding a compositor, and returns the compositor as a diagram.

The compositor can be given a custom name. If the diagram to be composed is named `'a'`, the default name is `'comp(a)'`.

In the theory of diagrammatic sets, a weak composite is witnessed by a weakly invertible compositor. In the implementation, we take an "invert when necessary" approach, where compositors are not invertible by default, and should be inverted when needed.

Notes

A cell (a diagram whose shape is an atom) is treated as already having itself as a composite, witnessed by a unit cell; this method can only be used on non-atomic diagrams.

Parameters

- **generatorname** (`hashable`) – Name of the generator that should be its composite.
- **diagram** (*Diagram*) – The diagram to compose.
- **compositorname** (`hashable`, optional) – Name of the compositor.

Returns
compositor – The diagram picking the compositor.

Return type
Diagram

Raises
`ValueError` – If the diagram is not round, or already has a composite, or the diagram and the generator do not have matching boundaries.

`remove`(*generatorname*)

Removes a generator, together with all other generators that depend on it.

Parameters
generatorname (`hashable`) – Name of the generator to remove.

`update`(*generatorname*, ***kwargs*)

Updates the optional arguments of a generator.

Parameters
generatorname (`hashable`) – Name of the generator to update.

Keyword Arguments
****kwargs** – Any arguments to update.

Raises
`AttributeError` – If the optional argument uses a private keyword.

`copy`()

Returns a copy of the object.

Returns
copy – A copy of the object.

Return type
DiagSet

`static yoneda`(*shape*)

Alternative constructor creating a diagrammatic set from a `shapes.Shape`.

Mathematically, diagrammatic sets are certain sheaves on the category of shapes and maps of shapes; this constructor implements the Yoneda embedding of a shape. This has an n-dimensional generator for each n-dimensional element of the shape.

Parameters
shape (`shapes.Shape`) – A shape.

Returns
yoneda – The Yoneda-embedded shape.

Return type
DiagSet

5.6.2 diagrams.Diagram

class rewalt.diagrams.**Diagram**(*ambient*)

Bases: `object`

Class for diagrams, that is, mappings from a shape to an "ambient" diagrammatic set.

To create a diagram, we start from *generators* of a diagrammatic set, returned by the `DiagSet.add()` method or requested with indexer operators.

Then we produce other diagrams in two main ways:

- pulling back a diagram along a map of shapes (`pullback()`), or
- pasting together two diagrams along their boundaries (`paste()`, `to_inputs()`, `to_outputs()`).

In practice, the direct use of `pullback()`, which requires an explicit shape map, can be avoided in common cases by using `unit()`, `lunitor()`, `runitor()`, or the specialised `SimplexDiagram.simplex_degeneracy`, `CubeDiagram.cube_degeneracy`, and `CubeDiagram.cube_connection` methods.

Notes

Initialising a `Diagram` directly creates an empty diagram in a given diagrammatic set.

Parameters
ambient (*DiagSet*) – The ambient diagrammatic set.

Methods

`boundary(sign[, dim])`	Returns the boundary of a given orientation and dimension.
`draw(**params)`	Bound version of `strdiags.draw()`.
`draw_boundaries(**params)`	Bound version of `strdiags.draw_boundaries()`.
`generate_layering()`	Assigns a layering to the diagram, iterating through all the layerings, and returns it.
`hasse(**params)`	Bound version of `hasse.draw()`.
`lunitor([sign, positions])`	Returns a left unitor on the diagram: a degenerate diagram one dimension higher, with one boundary equal to the diagram, and the other equal to the diagram with units pasted to some of its inputs.
`paste(other[, dim])`	Given two diagrams and `k` such that the output `k`-boundary of the first is equal to the input `k`-boundary of the second, returns their pasting along the matching boundaries.
`pullback(shapemap[, name])`	Returns the pullback of the diagram along a shape map.
`rename(name)`	Renames the diagram.
`rewrite(positions, diagram)`	Returns the diagram representing the application of a higher-dimensional rewrite to a subdiagram, specified by the positions of its top-dimensional elements.
`runitor([sign, positions])`	Returns a right unitor on the diagram: a degenerate diagram one dimension higher, with one boundary equal to the diagram, and the other equal to the diagram with units pasted to some of its outputs.
`to_inputs(positions, other[, dim])`	Returns the pasting of another diagram along a round subshape of the input `k`-boundary, specified by the positions of its `k`-dimensional elements.
`to_outputs(positions, other[, dim])`	Returns the pasting of another diagram along a round subshape of the output `k`-boundary, specified by the positions of its `k`-dimensional elements.
`unit()`	Returns the unit on the diagram: a degenerate diagram one dimension higher, with input and output equal to the diagram.
`with_layers(fst, *layers)`	Given a non-zero number of diagrams that can be pasted sequentially in the top dimension, returns their pasting.
`yoneda(shapemap[, name])`	Alternative constructor creating a diagram from a `shapes.ShapeMap`.

Attributes

ambient	Returns the ambient diagrammatic set.
composite	Returns the composite of the diagram, if it exists.
compositor	Returns the compositor of the diagram, if it exists.
dim	Shorthand for `shape.dim`.
hascomposite	Returns whether the diagram has a composite.
input	Alias for `boundary('-')`.
inverse	Returns the inverse of an invertible cell.
iscell	Shorthand for `shape.isatom` (a *cell* is a diagram whose shape is an atom).
isdegenerate	Returns whether the diagram is *degenerate*, that is, its image has dimension strictly lower than the dimension of its shape.
isinvertiblecell	Returns whether the diagram is an invertible cell.
isround	Shorthand for `shape.isround`.
layers	Returns the layering of the diagram corresponding to the current layering of the shape.
linvertor	Returns the left invertor for an invertible cell.
mapping	Returns the data specifying the mapping of shape elements to generators.
name	Returns the name of the diagram.
output	Alias for `boundary('+')`.
rewrite_steps	Returns the sequence of rewrite steps associated to the current layering of the diagram.
rinvertor	Returns the right invertor for an invertible cell.
shape	Returns the shape of the diagram.

property name

Returns the name of the diagram.

Returns
name – The name of the diagram.

Return type
`hashable`

property shape

Returns the shape of the diagram.

Returns
shape – The shape of the diagram.

Return type
`shapes.Shape`

property ambient

Returns the ambient diagrammatic set.

Returns
ambient – The ambient diagrammatic set.

Return type
DiagSet

property mapping

Returns the data specifying the mapping of shape elements to generators.

The mapping is specified as a list of lists, similar to `ogposets.OgMap`, in the following way: `mapping[n][k] == s` if the diagram sends `El(n, k)` to the generator named `s`.

Returns
mapping – The data specifying the diagram's assignment.

Return type
`list[list[hashable]]`

property layers

Returns the layering of the diagram corresponding to the current layering of the shape.

Returns
layers – The current layering.

Return type
`list[Diagram]`

property rewrite_steps

Returns the sequence of rewrite steps associated to the current layering of the diagram.

The `0`-th rewrite step is the input boundary of the diagram. For `n > 0`, the `n`-th rewrite step is the output boundary of the `(n-1)`-th layer.

Returns
rewrite_steps – The current sequence of rewrite steps.

Return type
`list[Diagram]`

property dim

Shorthand for `shape.dim`.

property isdegenerate

Returns whether the diagram is *degenerate*, that is, its image has dimension strictly lower than the dimension of its shape.

Returns
isdegenerate – `True` if and only if the diagram is degenerate.

Return type
`bool`

property isround

Shorthand for `shape.isround`.

property iscell

Shorthand for `shape.isatom` (a *cell* is a diagram whose shape is an atom).

property isinvertiblecell

Returns whether the diagram is an invertible cell.

A cell is invertible if either

- it is degenerate, or
- its image is an invertible generator.

Returns
isinvertiblecell – `True` if and only if the diagram is an invertible cell.

Return type
`bool`

property hascomposite

Returns whether the diagram has a composite.

Returns
hascomposite – `True` if and only if the diagram has a composite.

Return type
`bool`

rename(*name*)

Renames the diagram.

Parameters
name (`hashable`) – The new name for the diagram.

paste(*other*, *dim=None*, ***params*)

Given two diagrams and `k` such that the output `k`-boundary of the first is equal to the input `k`-boundary of the second, returns their pasting along the matching boundaries.

Parameters

- **fst** (*Diagram*) – The first diagram.
- **snd** (*Diagram*) – The second diagram.
- **dim** (`int`, optional) – The dimension of the boundary along which they will be pasted (default is `min(fst.dim, snd.dim) - 1`).

Keyword Arguments
cospan (`bool`) – Whether to also return the cospan of inclusions of the two diagrams' shapes into the pasting (default is `False`).

Returns

- **paste** (*Diagram*) – The pasted diagram.
- **paste_cospan** (`ogposets.OgMapPair`, optional) – The cospan of inclusions of the two diagrams' shapes into their pasting.

Raises
ValueError – If the boundaries do not match.

to_outputs(*positions*, *other*, *dim=None*, ***params*)

Returns the pasting of another diagram along a round subshape of the output `k`-boundary, specified by the positions of its `k`-dimensional elements.

Parameters

- **positions** (`list[int] | int`) – The positions of the outputs along which to paste. If given an integer `n`, interprets it as the list `[n]`.
- **other** (*Diagram*) – The other diagram to paste.
- **dim** (`int`, optional) – The dimension of the boundary along which to paste (default is `self.dim - 1`)

Keyword Arguments
`cospan` (`bool`) – Whether to return the cospan of inclusions of the two diagrams' shapes into the pasting (default is `False`).

Returns

- **to_outputs** (`Shape`) – The pasted diagram.
- **paste_cospan** (`ogposets.OgMapPair`, optional) – The cospan of inclusions of the two diagrams' shapes into their pasting.

Raises
`ValueError` – If the boundaries do not match, or the pasting does not produce a well-formed shape.

`to_inputs`(*positions*, *other*, *dim=None*, ***params*)

Returns the pasting of another diagram along a round subshape of the input `k`-boundary, specified by the positions of its `k`-dimensional elements.

Parameters

- `positions` (`list[int] | int`) – The positions of the inputs along which to paste. If given an integer `n`, interprets it as the list `[n]`.
- `other` (*`Diagram`*) – The other diagram to paste.
- `dim` (`int`, optional) – The dimension of the boundary along which to paste (default is `self.dim - 1`)

Keyword Arguments
`cospan` (`bool`) – Whether to return the cospan of inclusions of the two diagrams' shapes into the pasting (default is `False`).

Returns

- **to_inputs** (`Shape`) – The pasted diagram.
- **paste_cospan** (`ogposets.OgMapPair`, optional) – The cospan of inclusions of the two diagrams' shapes into their pasting.

Raises
`ValueError` – If the boundaries do not match, or the pasting does not produce a well-formed shape.

`rewrite`(*positions*, *diagram*)

Returns the diagram representing the application of a higher-dimensional rewrite to a subdiagram, specified by the positions of its top-dimensional elements.

This is in fact an alias for *`to_outputs()`* in the top dimension, reflecting the intuitions of higher-dimensional rewriting in this situation.

Parameters

- `positions` (`list[int] | int`) – The positions of the top-dimensional elements to rewrite. If given an integer `n`, interprets it as the list `[n]`.
- `diagram` (*`Diagram`*) – The diagram representing the rewrite to apply.

Returns
rewrite – The diagram representing the application of the rewrite to the given positions.

Return type
`Shape`

`pullback(shapemap, name=None)`

Returns the pullback of the diagram along a shape map.

Parameters

- **shapemap** (`shapes.ShapeMap`) – The map along which to pull back.
- **name** (`hashable`, optional) – The name to give to the new diagram.

Returns
pullback – The pulled back diagram.

Return type
Diagram

Raises
`ValueError` – If the target of the map is not equal to the diagram shape.

`boundary(sign, dim=None)`

Returns the boundary of a given orientation and dimension.

This is, by definition, the pullback of a diagram along the inclusion map `self.shape.boundary(sign, dim)`.

Parameters

- **sign** (`str`) – Orientation: `'-'` for input, `'+'` for output.
- **dim** (`int`, optional) – Dimension of the boundary (default is `self.dim - 1`).

Returns
boundary – The requested boundary.

Return type
Diagram

property input

Alias for `boundary('-')`.

property output

Alias for `boundary('+')`.

`unit()`

Returns the unit on the diagram: a degenerate diagram one dimension higher, with input and output equal to the diagram.

This is, by definition, the pullback of the diagram along `self.shape.inflate()`.

Returns
unit – The unit diagram.

Return type
Diagram

`lunitor(sign='-', positions=None)`

Returns a left unitor on the diagram: a degenerate diagram one dimension higher, with one boundary equal to the diagram, and the other equal to the diagram with units pasted to some of its inputs.

Parameters

- **sign** (`str`, optional) – The boundary on which the units are: `'-'` (default) for input, `'+'` for output.

- **positions** (list[int] | int) – The positions of the inputs to which a unit is attached (default is *all* of the inputs). If given an integer n, interprets it as the list [n].

Returns
lunitor – The left unitor diagram.

Return type
Diagram

Raises
ValueError – If the positions do not correspond to inputs.

runitor(*sign='-'*, *positions=None*)

Returns a right unitor on the diagram: a degenerate diagram one dimension higher, with one boundary equal to the diagram, and the other equal to the diagram with units pasted to some of its outputs.

Parameters

- **sign** (str, optional) – The boundary on which the units are: '-' (default) for input, '+' for output.
- **positions** (list[int] | int) – The positions of the outputs to which a unit is attached (default is *all* of the outputs). If given an integer n, interprets it as the list [n].

Returns
runitor – The right unitor diagram.

Return type
Diagram

Raises
ValueError – If the positions do not correspond to outputs.

property inverse

Returns the inverse of an invertible cell.

Returns
inverse – The inverse cell.

Return type
Diagram

Raises
ValueError – If the diagram is not an invertible cell.

property rinvertor

Returns the right invertor for an invertible cell.

Returns
rinvertor – The right invertor.

Return type
Diagram

Raises
ValueError – If the diagram is not an invertible cell.

property linvertor

Returns the left invertor for an invertible cell.

Returns
linvertor – The left invertor.

Return type
Diagram

Raises
`ValueError` – If the diagram is not an invertible cell.

property composite

Returns the composite of the diagram, if it exists.

Returns
composite – The composite.

Return type
Diagram

Raises
`ValueError` – If the diagram does not have a composite.

property compositor

Returns the compositor of the diagram, if it exists.

Returns
compositor – The compositor.

Return type
Diagram

Raises
`ValueError` – If the diagram does not have a composite.

generate_layering()

Assigns a layering to the diagram, iterating through all the layerings, and returns it.

Returns
layers – The generated layering.

Return type
`list[Diagram]`

hasse(***params*)

Bound version of `hasse.draw()`.

Calling `x.hasse(**params)` is equivalent to calling `hasse.draw(x, **params)`.

draw(***params*)

Bound version of `strdiags.draw()`.

Calling `x.draw(**params)` is equivalent to calling `strdiags.draw(x, **params)`.

draw_boundaries(***params*)

Bound version of `strdiags.draw_boundaries()`.

Calling `x.draw_boundaries(**params)` is equivalent to calling `strdiags.draw_boundaries(x, **params)`.

static yoneda(*shapemap*, *name=None*)

Alternative constructor creating a diagram from a `shapes.ShapeMap`.

Mathematically, diagrammatic sets are certain sheaves on the category of shapes and maps of shapes; this constructor implements the Yoneda embedding of a map of shapes.

Parameters

- **shapemap** (shapes.Shape) – A map of shapes.
- **name** (hashable, optional) – The name of the generated diagram.

Returns
yoneda – The Yoneda-embedded map.

Return type
Diagram

static with_layers(*fst*, **layers*)

Given a non-zero number of diagrams that can be pasted sequentially in the top dimension, returns their pasting.

Parameters

- **fst** (*Diagram*) – The first diagram.
- ***layers** (*Diagram*) – Any number of additional diagrams.

Returns
with_layers – The pasting of all the diagrams in the top dimension.

Return type
Diagram

Raises
ValueError – If the diagrams are not pastable.

5.6.3 diagrams.SimplexDiagram

class rewalt.diagrams.**SimplexDiagram**(*ambient*)

Bases: *Diagram*

Subclass of *Diagram* for diagrams whose shape is an oriented simplex.

The methods of this class provide an implementation of the structural maps of a simplicial set.

Methods

simplex_degeneracy(k)	Returns one of the degeneracies of the simplex.
simplex_face(k)	Returns one of the faces of the simplex.

simplex_face(*k*)

Returns one of the faces of the simplex.

This is, by definition, the pullback of the diagram along `self.shape.simplex_face(k)`.

Parameters
k (int) – The index of the face, ranging from `0` to `self.dim`.

Returns
simplex_face – The face.

Return type
Diagram

Raises
ValueError – If the index is out of range.

simplex_degeneracy(*k*)

Returns one of the degeneracies of the simplex.

This is, by definition, the pullback of the diagram along `self.shape.simplex_degeneracy(k)`.

Parameters
k (`int`) – The index of the degeneracy, ranging from `0` to `self.dim`.

Returns
simplex_degeneracy – The degeneracy.

Return type
Diagram

Raises
ValueError – If the index is out of range.

5.6.4 diagrams.CubeDiagram

class rewalt.diagrams.**CubeDiagram**(*ambient*)

Bases: *Diagram*

Subclass of *Diagram* for diagrams whose shape is an oriented cube.

The methods of this class provide an implementation of the structural maps of a cubical set with connections.

Methods

cube_connection(k, sign)	Returns one of the connections of the cube.
cube_degeneracy(k)	Returns one of the degeneracies of the cube.
cube_face(k, sign)	Returns one of the faces of the cube.

cube_face(*k*, *sign*)

Returns one of the faces of the cube.

This is, by definition, the pullback of the diagram along `self.shape.cube_face(k, sign)`.

Parameters

- **k** (`int`) – Index of the face, ranging from `0` to `self.dim - 1`.
- **sign** (`str`) – Side: `'-'` or `'+'`.

Returns
cube_face – The face.

Return type
Diagram

Raises
ValueError – If the index is out of range.

cube_degeneracy(*k*)

Returns one of the degeneracies of the cube.

This is, by definition, the pullback of the diagram along `self.shape.cube_degeneracy(k)`.

Parameters
k (`int`) – The index of the degeneracy, ranging from `0` to `self.dim`.

Returns
cube_degeneracy – The degeneracy.

Return type
Diagram

Raises
`ValueError` – If the index is out of range.

`cube_connection`(*k*, *sign*)

Returns one of the connections of the cube.

This is, by definition, the pullback of the diagram along `self.shape.cube_connection(k, sign)`.

Parameters

- **k** (`int`) – Index of the connection, ranging from `0` to `self.dim - 1`.
- **sign** (`str`) – Side: `'-'` or `'+'`.

Returns
cube_face – The connection.

Return type
Diagram

Raises
`ValueError` – If the index is out of range.

5.6.5 diagrams.PointDiagram

`class rewalt.diagrams.PointDiagram`(*ambient*)

Bases: *SimplexDiagram*, *CubeDiagram*

Subclass of *Diagram* for diagrams whose shape is a point.

Methods

degeneracy(shape)	Given a shape, returns the unique degenerate diagram of that shape over the point.

`degeneracy`(*shape*)

Given a shape, returns the unique degenerate diagram of that shape over the point.

This is, by definition, the pullback of the point diagram along `self.shape.terminal()`.

Parameters
shape (`shapes.Shape`) – The shape of the degenerate diagram.

Returns
degeneracy – The degenerate diagram.

Return type
Diagram

5.7 shapes

Implements shapes of cells and diagrams.

rewalt.shapes.Shape()	Inductive subclass of `ogposets.OgPoset` for shapes of cells and diagrams.
rewalt.shapes.ShapeMap(ogmap, **params)	An overlay of `ogposets.OgMap` for total maps between *Shape* objects.
rewalt.shapes.Simplex()	Subclass of *Shape* for oriented simplices.
rewalt.shapes.Cube()	Subclass of *Shape* for oriented cubes.

5.7.1 shapes.Shape

class `rewalt.shapes.`**`Shape`**

Bases: *OgPoset*

Inductive subclass of `ogposets.OgPoset` for shapes of cells and diagrams.

Properly formed objects of the class are unique encodings of the *regular molecules* from the theory of diagrammatic sets (plus the empty shape, which is not considered a regular molecule).

To create shapes, we start from basic constructors such as *empty()*, *point()*, or one of the named shape constructors, such as *globe()*, *simplex()*, *cube()*.

Then we generate new shapes by gluing basic shapes together with *paste()*, *to_inputs()*, *to_outputs()*, or by producing new higher-dimensional shapes with operations such as *atom()*, *gray()*, *join()*.

When possible, the constructors place the shapes in appropriate subclasses of separate interest, which include the *globes*, the *oriented simplices*, the *oriented cubes*, and the *positive opetopes*. This is to enable the specification of special methods for subclasses of shapes.

The following diagram summarises the hierarchy of subclasses of shapes:

```
 Simplex  Cube  OpetopeTree  Theta
 |    |\  |\     |      |      |
 |    | \ | \ Opetope  GlobeString
 |    |  \|  \   |    /
 |    |   \   \  Globe
 |    |   |\   \/    |
Empty |   | \  /\    |
      |   |  \/  \   |
      |   |  /\   \  |
      |   | /  \   \ |
      |   |/    \   \|
      Point      Arrow
```

Currently only the *Cube* and *Simplex* classes have special methods implemented.

Methods

all_layerings()	Returns an iterator on all *layerings* of a shape of dimension n into shapes with a single n-dimensional element, pasted along their (n-1)-dimensional boundary.
arrow()	Constructs the arrow, the unique 1-dimensional atomic shape.
atom(fst, snd, **params)	Given two shapes with identical round boundaries, returns a new atomic shape whose input boundary is the first one and output boundary the second one.
atom_inclusion(element)	Returns the inclusion of the closure of an element, which is an atomic shape, in the shape.
boundary([sign, dim])	Returns the inclusion of the boundary of a given orientation and dimension into the shape.
cube([dim])	Constructs the oriented cube of a given dimension.
draw(**params)	Bound version of `strdiags.draw()`.
draw_boundaries(**params)	Bound version of `strdiags.draw_boundaries()`.
dual(shape, *dims, **params)	Returns the shape with orientations reversed in given dimensions.
empty()	Constructs the initial, empty shape.
generate_layering()	Assigns a layering to the shape, iterating through all the layerings, and returns it.
globe([dim])	Constructs the globe of a given dimension.
gray(*shapes)	Returns the Gray product of any number of shapes.
id()	Returns the identity map on the shape.
inflate([collapsed])	Given a closed subset of the boundary of the shape, forms a cylinder on the shape, with the sides incident to the closed subset collapsed, and returns its projection map onto the original shape.
initial()	Returns the unique map from the initial, empty shape.
join(*shapes)	Returns the join of any number of shapes.
merge()	Returns the unique atomic shape with the same boundary, if the shape is round.
paste(fst, snd[, dim])	Given two shapes and k such that the output k-boundary of the first is equal to the input k-boundary of the second, returns their pasting along the matching boundaries.
paste_along(fst, snd, **params)	Given a span of shape maps, where one is the inclusion of the input (resp output) k-boundary of a shape, and the other the inclusion of a round subshape of the output (resp input) k-boundary of another shape, returns the pasting (pushout) of the two shapes along the span.
point()	Constructs the terminal shape, consisting of a single point.
simplex([dim])	Constructs the oriented simplex of a given dimension.
suspend(shape[, n])	Returns the n-fold suspension of a shape.
terminal()	Returns the unique map to the point, the terminal shape.
theta(*thetas)	Inductive constructor for the objects of the Theta category, sometimes known as Batanin cells.
to_inputs(positions, other[, dim])	Returns the pasting of another shape along a round subshape of the input k-boundary, specified by the positions of its k-dimensional elements.
to_outputs(positions, other[, dim])	Returns the pasting of another shape along a round subshape of the output k-boundary, specified by the positions of its k-dimensional elements.

Attributes

`isatom`	Returns whether the shape is an atom (has a greatest element).
`isround`	Shorthand for `all().isround`.
`layers`	Returns the current layering of the shape.
`rewrite_steps`	Returns the sequence of rewrite steps associated to the current layering of the shape.

property isatom

Returns whether the shape is an atom (has a greatest element).

Returns
isatom – `True` if and only if the shape has a greatest element.

Return type
`bool`

Examples

```
>>> arrow = Shape.arrow()
>>> assert arrow.isatom
>>> assert not arrow.paste(arrow).isatom
```

property isround

Shorthand for `all().isround`.

property layers

Returns the current layering of the shape.

Returns
layers – The current layering.

Return type
`list[ShapeMap]`

Examples

```
>>> arrow = Shape.arrow()
>>> globe = Shape.globe(2)
>>> cospan = globe.paste(arrow).paste(
...         arrow.paste(globe), cospan=True)
>>> shape = cospan.target
>>> assert shape.layers == [cospan.fst, cospan.snd]
```

property rewrite_steps

Returns the sequence of rewrite steps associated to the current layering of the shape.

The `0`-th rewrite step is the input boundary of the shape. For `n > 0`, the `n`-th rewrite step is the output boundary of the `(n-1)`-th layer.

Returns
rewrite_steps – The current sequence of rewrite steps.

Return type
`list[ShapeMap]`

Examples

```
>>> arrow = Shape.arrow()
>>> globe = Shape.globe(2)
>>> cospan = globe.paste(arrow).paste(
...         arrow.paste(globe), cospan=True)
>>> shape = cospan.target
>>> assert shape.rewrite_steps == [
...         cospan.fst.input,
...         cospan.fst.output,
...         cospan.snd.output]
```

static atom(*fst*, *snd*, ***params*)

Given two shapes with identical round boundaries, returns a new atomic shape whose input boundary is the first one and output boundary the second one.

Parameters

- **fst** (*Shape*) – The input boundary shape.
- **snd** (*Shape*) – The output boundary shape.

Keyword Arguments
cospan (`bool`) – Whether to return the cospan of inclusions of the input and output boundaries (default is `False`).

Returns
atom – The new atomic shape (optionally with the cospan of inclusions of its boundaries).

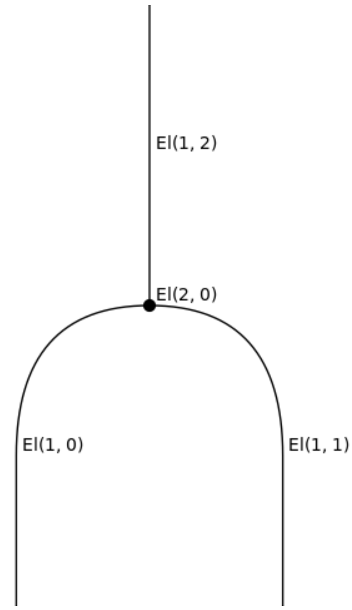
Return type
Shape | `ogposets.OgMapPair`

Raises
ValueError – If the boundaries do not match, or are not round.

Examples

We create a 2-dimensional cell shape with two input 1-cells and one output 2-cell.

```
>>> arrow = Shape.arrow()
>>> binary = arrow.paste(arrow).atom(arrow)
>>> binary.draw(path='docs/_static/img/Shape_atom.png')
```



static paste(*fst*, *snd*, *dim=None*, ***params*)

Given two shapes and `k` such that the output `k`-boundary of the first is equal to the input `k`-boundary of the second, returns their pasting along the matching boundaries.

Parameters

- **fst** (*Shape*) – The first shape.
- **snd** (*Shape*) – The second shape.
- **dim** (`int`, optional) – The dimension of the boundary along which they will be pasted (default is `min(fst.dim, snd.dim) - 1`).

Keyword Arguments

cospan (`bool`) – Whether to return the cospan of inclusions of the two shapes into the pasting (default is `False`).

Returns

paste – The pasted shape (optionally with the cospan of inclusions of its components).

Return type

Shape | `ogposets.OgMapPair`

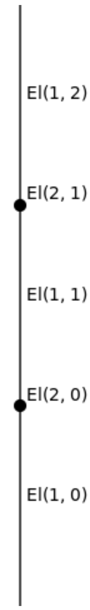
Raises

ValueError – If the boundaries do not match.

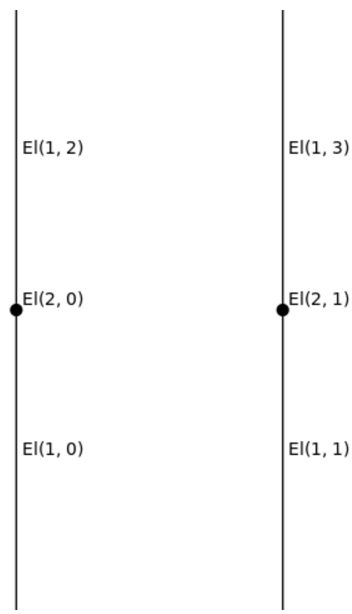
Examples

We can paste two 2-dimensional globes either "vertically" along their 1-dimensional boundary or "horizontally" along their 0-dimensional boundary.

```
>>> globe = Shape.globe(2)
>>> vert = globe.paste(globe)
>>> horiz = globe.paste(globe, 0)
>>> vert.draw(path='docs/_static/img/Shape_paste_vert.png')
```

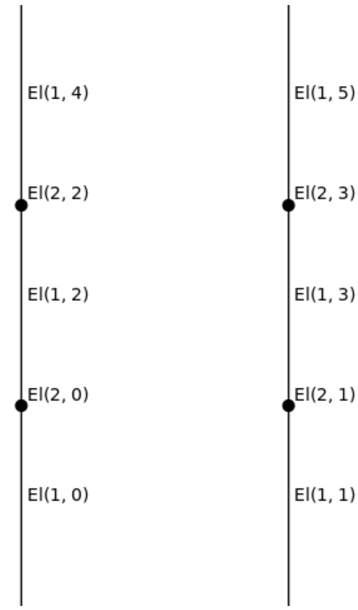



```
>>> horiz.draw(path='docs/_static/img/Shape_paste_horiz.png')
```



We can also check that the interchange equation holds.

```
>>> assert vert.paste(vert, 0) == horiz.paste(horiz)
>>> horiz.paste(horiz).draw(
...     path='docs/_static/img/Shape_paste_interchange.png')
```



static paste_along(*fst*, *snd*, ***params*)

Given a span of shape maps, where one is the inclusion of the input (resp output) `k`-boundary of a shape, and the other the inclusion of a round subshape of the output (resp input) `k`-boundary of another shape, returns the pasting (pushout) of the two shapes along the span.

In practice, it is convenient to use `to_inputs()` and `to_outputs()` instead, where the data of the span is specified by `k` and the positions of the `k`-dimensional elements in the round subshape along which the pasting occurs.

Parameters

- **fst** (*ShapeMap*) – The first inclusion.
- **snd** (*ShapeMap*) – The second inclusion.

Keyword Arguments

- **wfcheck** (`bool`) – Check if the span gives rise to a well-formed pasting (default is `True`).
- **cospan** (`bool`) – Whether to return the cospan of inclusions of the two shapes into the pasting (default is `False`).

Returns

paste_along – The pasted shape (optionally with the cospan of inclusions of its components).

Return type

Shape | `ogposets.OgMapPair`

Raises

ValueError – If the pair of maps is not an injective span.

to_outputs(*positions*, *other*, *dim=None*, ***params*)

Returns the pasting of another shape along a round subshape of the output `k`-boundary, specified by the positions of its `k`-dimensional elements.

Parameters

- **positions** (`list[int] | int`) – The positions of the outputs along which to paste. If given an integer `n`, interprets it as the list `[n]`.
- **other** (*Shape*) – The other shape to paste.

- **dim** (int, optional) – The dimension of the boundary along which to paste (default is `self.dim - 1`)

Keyword Arguments
cospan (bool) – Whether to return the cospan of inclusions of the two shapes into the pasting (default is `False`).

Returns
to_outputs – The pasted shape (optionally with the cospan of inclusions of its components).

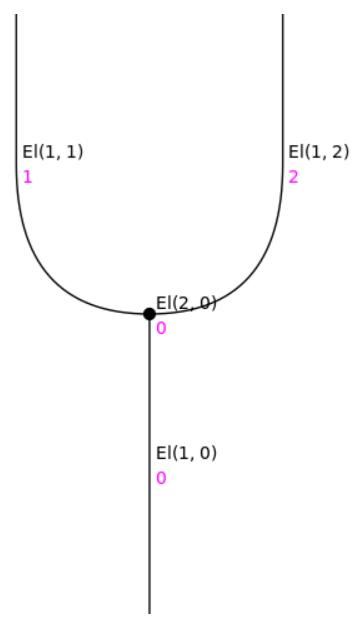
Return type
Shape | `ogposets.OgMapPair`

Raises
ValueError – If the boundaries do not match, or the pasting does not produce a well-formed shape.

Examples

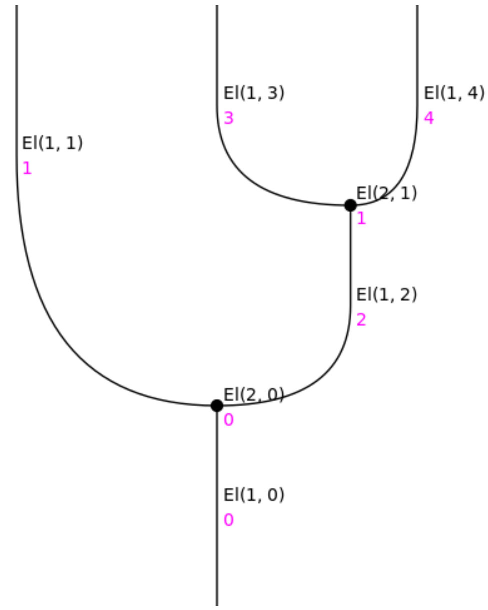
We create a 2-simplex and visualise it as a string diagram with the `positions` parameter enabled.

```
>>> simplex = Shape.simplex(2)
>>> simplex.draw(
...     positions=True, path='docs/_static/img/Shape_to_outputs1.png')
```



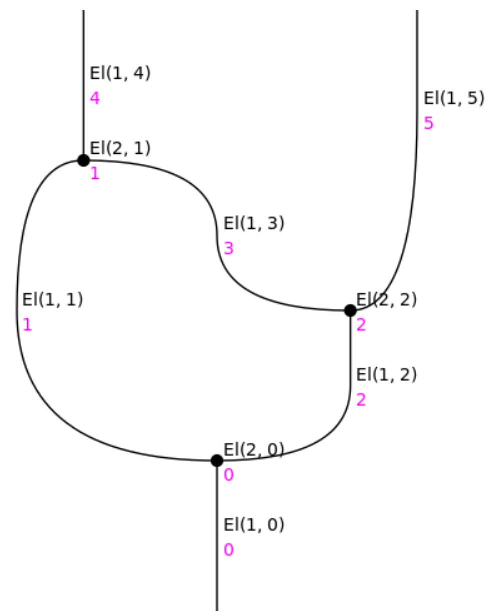
We paste another 2-simplex to the output in position 2.

```
>>> paste1 = simplex.to_outputs(2, simplex)
>>> paste1.draw(
...     positions=True, path='docs/_static/img/Shape_to_outputs2.png')
```



Finally, we paste the *dual* of a 2-simplex to the outputs in positions `2, 3`.

```
>>> paste2 = paste1.to_outputs([1, 3], simplex.dual())
>>> paste2.draw(
...     positions=True, path='docs/_static/img/Shape_to_outputs3.png')
```



to_inputs(*positions*, *other*, *dim=None*, ***params*)

Returns the pasting of another shape along a round subshape of the input k-boundary, specified by the positions of its k-dimensional elements.

Parameters

- **positions** (list[int] | int) – The positions of the inputs along which to paste. If given an integer n, interprets it as the list [n].
- **other** (*Shape*) – The other shape to paste.

- **dim** (int, optional) – The dimension of the boundary along which to paste (default is self.dim - 1)

Keyword Arguments
cospan (bool) – Whether to return the cospan of inclusions of the two shapes into the pasting (default is False).

Returns
to_inputs – The pasted shape (optionally with the cospan of inclusions of its components).

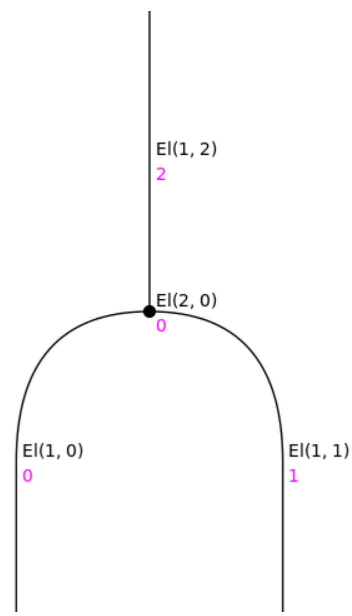
Return type
Shape | ogposets.OgMapPair

Raises
ValueError – If the boundaries do not match, or the pasting does not produce a well-formed shape.

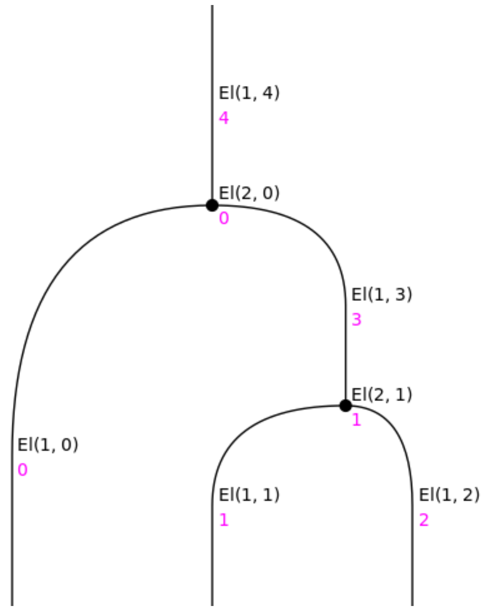
Examples

We work dually to the example for *to_outputs()*.

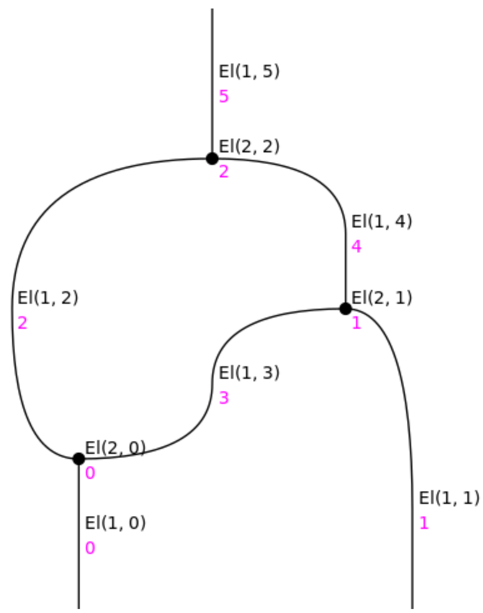
```
>>> binary = Shape.simplex(2).dual()
>>> binary.draw(
...     positions=True, path='docs/_static/img/Shape_to_inputs1.png')
```




```
>>> paste1 = binary.to_inputs(1, binary)
>>> paste1.draw(
...     positions=True, path='docs/_static/img/Shape_to_inputs2.png')
```




```
>>> paste2 = paste1.to_inputs([0, 1], binary.dual())
>>> paste2.draw(
...     positions=True, path='docs/_static/img/Shape_to_inputs3.png')
```



static suspend(*shape*, *n=1*)

Returns the n-fold suspension of a shape.

This static method can be also used as a bound method after an object is initialised, that is, `shape.suspend(n)` is equivalent to `suspend(shape, n)`.

Parameters

- **shape** (*Shape*) – The object to suspend.
- **n** (`int`, optional) – The number of iterations of the suspension (default is 1).

Returns
suspension – The suspended shape.

Return type
Shape

Examples

The suspension of the point is the arrow, and the suspension of an arrow is the 2-globe.

```
>>> assert Shape.point().suspend() == Shape.arrow()
>>> assert Shape.arrow().suspend() == Shape.globe(2)
```

In general, the suspension of the n-globe is the (n+1)-globe.

static gray(**shapes*)

Returns the Gray product of any number of shapes.

This method can be called with the math operator `*`, that is, `fst * snd` is equivalent to `gray(fst, snd)`.

This static method can also be used as a bound method after an object is initialised, that is, `fst.gray(*shapes)` is equivalent to `gray(fst, *shapes)`.

Parameters
***shapes** (*Shape*) – Any number of shapes.

Returns
gray – The Gray product of the arguments.

Return type
Shape

Example

The point is a unit for the Gray product.

```
>>> point = Shape.point()
>>> arrow = Shape.arrow()
>>> assert point*arrow == arrow*point == arrow
```

The Gray product of two arrows is the oriented square (2-cube).

```
>>> arrow = Shape.arrow()
>>> assert arrow*arrow == Shape.cube(2)
```

In general, the Gray product of the n-cube with the k-cube is the (n+k)-cube.

static join(**shapes*)

Returns the join of any number of shapes.

This method can be called with the shift operators >> and <<, that is, `fst >> snd` is equivalent to `join(fst, snd)` and `fst << snd` is equivalent to `join(snd, fst)`.

This static method can also be used as a bound method after an object is initialised, that is, `fst.join(*shapes)` is equivalent to `join(fst, *shapes)`.

Parameters
***shapes** (*Shape*) – Any number of shapes.

Returns
join – The join of the arguments.

Return type
Shape

Examples

The empty shape is a unit for the join.

```
>>> empty = Shape.empty()
>>> point = Shape.point()
>>> assert empty >> point == point >> empty == point
```

The join of two points is the arrow, and the join of an arrow and a point is the 2-simplex.

```
>>> arrow = Shape.arrow()
>>> assert point >> point == Shape.arrow()
>>> assert arrow >> point == Shape.simplex(2)
```

In general, the join of an n-simplex with a k-simplex is the (n+k+1)-simplex.

`static dual`(*shape*, **dims*, ***params*)

Returns the shape with orientations reversed in given dimensions.

The dual in all dimensions can also be called with the bit negation operator ~, that is, `~shape` is equivalent to `shape.dual()`.

This static method can be also used as a bound method after an object is initialised, that is, `shape.dual(*dims)` is equivalent to `dual(shape, *dims)`.

Parameters

- **shape** (*Shape*) – A shape.
- ***dims** (`int`) – Any number of dimensions; if none, defaults to *all* dimensions.

Returns
dual – The shape, dualised in the given dimensions.

Return type
Shape

Examples

```
>>> arrow = Shape.arrow()
>>> simplex = Shape.simplex(2)
>>> binary = arrow.paste(arrow).atom(arrow)
>>> assert binary == simplex.dual()
```

```
>>> assoc_l = binary.to_inputs(0, binary)
>>> assoc_r = binary.to_inputs(1, binary)
>>> assert assoc_r == assoc_l.dual(1)
```

`merge()`

Returns the unique atomic shape with the same boundary, if the shape is round.

Returns
merge – The unique atomic shape with the same boundary.

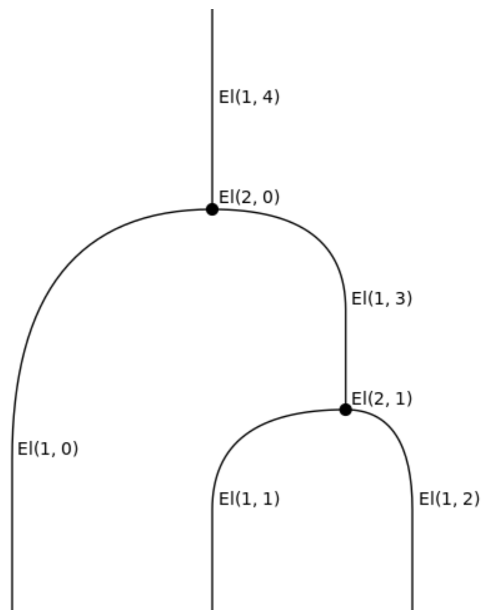
Return type
Shape

Raises
`ValueError` – If the shape is not round.

Examples

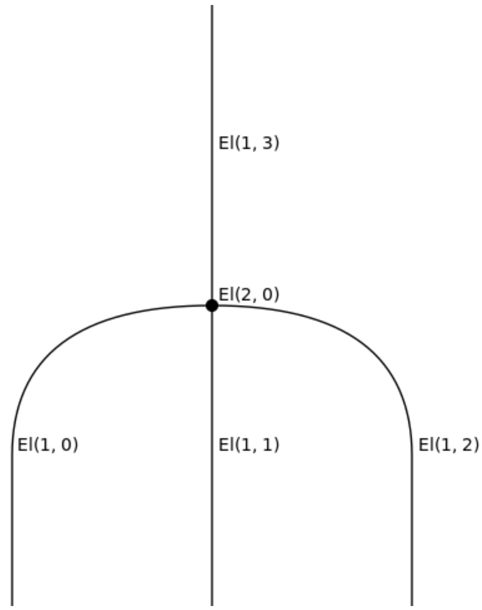
We create a 2-dimensional shape with two input 1-cells and one output 1-cell, and paste it to itself along one of the inputs.

```
>>> arrow = Shape.arrow()
>>> binary = arrow.paste(arrow).atom(arrow)
>>> to_merge = binary.to_inputs(1, binary)
>>> to_merge.draw(path='docs/_static/img/Shape_merge1.png')
```



The “merged” shape is the 2-dimensional atom with three input 2-cells and one output 1-cell.

```
>>> merged = to_merge.merge()
>>> merged.draw(path='docs/_static/img/Shape_merge2.png')
```



static empty()

Constructs the initial, empty shape.

Returns
empty – The empty shape.

Return type
`Empty`

static point()

Constructs the terminal shape, consisting of a single point.

Returns
point – The point.

Return type
`Point`

static arrow()

Constructs the arrow, the unique 1-dimensional atomic shape.

Returns
arrow – The arrow.

Return type
`Arrow`

static simplex(*dim=-1*)

Constructs the oriented simplex of a given dimension.

Parameters
dim (`int`) – The dimension of the simplex (default is `-1`).

Returns
simplex – The simplex of the requested dimension.

Return type
`Simplex`

static cube(*dim=0*)

Constructs the oriented cube of a given dimension.

Parameters
dim (int) – The dimension of the cube (default is `0`).

Returns
cube – The cube of the requested dimension.

Return type
Cube

static globe(*dim=0*)

Constructs the globe of a given dimension.

Parameters
dim (int) – The dimension of the globe (default is `0`).

Returns
globe – The globe of the requested dimension.

Return type
Globe

static theta(**thetas*)

Inductive constructor for the objects of the Theta category, sometimes known as Batanin cells.

Batanin cells are in 1-to-1 correspondence with finite plane trees. The constructor is based on this correspondence, using the well-known inductive definition of plane trees: given any number `k` of Batanin cells, it returns the Batanin cell encoded by a root with `k` children, to which the `k` plane trees encoding the arguments are attached.

Parameters
thetas (Theta) – Any number of Batanin cells.

Returns
theta – The resulting Batanin cell.

Return type
Theta

Examples

Every globe is a Batanin cell, encoded by the linear tree of length equal to its dimension.

```
>>> assert Shape.theta() == Shape.globe(0)
>>> assert Shape.theta(Shape.theta()) == Shape.globe(1)
>>> assert Shape.theta(Shape.theta(Shape.theta())) == Shape.globe(2)
```

The tree with one root with n children corresponds to a string of n arrows.

```
>>> point = Shape.theta()
>>> arrow = Shape.arrow()
>>> assert Shape.theta(point, point) == arrow.paste(arrow)
```

id()

Returns the identity map on the shape.

Returns
id – The identity map on the object.

Return type
ShapeMap

`boundary`(*sign=None*, *dim=None*)

Returns the inclusion of the boundary of a given orientation and dimension into the shape.

Note that input and output boundaries of shapes are shapes, so they are returned as shape maps; however, the entire (input + output) boundary of a shape is not a shape, so it is returned simply as a map of oriented graded posets.

Parameters

- **sign** (`str`, optional) – Orientation: `'-'` for input, `'+'` for output, `None` (default) for both.
- **dim** (`int`, optional) – Dimension of the boundary (default is `self.dim - 1`).

Returns
boundary – The inclusion of the requested boundary into the object.

Return type
ShapeMap | `OgMap`

Examples

```
>>> point = Shape.point()
>>> arrow = Shape.arrow()
>>> binary = arrow.paste(arrow).atom(arrow)
>>> assert binary.boundary('-').source == arrow.paste(arrow)
>>> assert binary.boundary('+').source == arrow
>>> assert binary.boundary('-', 0).source == point
>>> assert binary.boundary('-').target == binary
```

`atom_inclusion`(*element*)

Returns the inclusion of the closure of an element, which is an atomic shape, in the shape.

Parameters
element (`El`) – An element of the shape.

Returns
atom_inclusion – The inclusion of the closure of the element.

Return type
ShapeMap

Examples

```
>>> arrow = Shape.arrow()
>>> globe = Shape.globe(2)
>>> whisker_l = arrow.paste(globe)
>>> assert whisker_l.atom_inclusion(El(2, 0)).source == globe
```

`initial`()

Returns the unique map from the initial, empty shape.

Returns
initial – The unique map from the empty shape.

Return type
ShapeMap

Examples

```
>>> point = Shape.point()
>>> empty = Shape.empty()
>>> assert point.initial() == empty.terminal()
>>> assert empty.initial() == empty.id()
```

terminal()

Returns the unique map to the point, the terminal shape.

Returns
terminal – The unique map to the point.

Return type
ShapeMap

Examples

```
>>> point = Shape.point()
>>> assert point.terminal() == point.id()
```

inflate(*collapsed=None*)

Given a closed subset of the boundary of the shape, forms a cylinder on the shape, with the sides incident to the closed subset collapsed, and returns its projection map onto the original shape.

This is mainly used in constructing units and unitors on diagrams; see `diagrams.Diagram.unit()`, `diagrams.Diagram.lunitor()`, `diagrams.Diagram.runitor()`.

Parameters
collapsed (`Closed`, optional) – A closed subset of the boundary of the shape (default is the entire boundary).

Returns
inflate – The projection map of the "partially collapsed cylinder" onto the shape.

Return type
`Closed`

Raises
ValueError – If *collapsed* is not a subset of the boundary.

all_layerings()

Returns an iterator on all *layerings* of a shape of dimension `n` into shapes with a single `n`-dimensional element, pasted along their `(n-1)`-dimensional boundary.

Returns
all_layerings – The iterator on all layerings of the shape.

Return type
`Iterable`

generate_layering()

Assigns a layering to the shape, iterating through all the layerings, and returns it.

Returns
layers – The generated layering.

Return type
list[ShapeMap]

Examples

```
>>> arrow = Shape.arrow()
>>> globe = Shape.globe(2)
>>> chain = globe.paste(globe, 0)
>>> chain.generate_layering()
>>> assert chain.layers[0].source == arrow.paste(globe)
>>> assert chain.layers[1].source == globe.paste(arrow)
>>> chain.generate_layering()
>>> assert chain.layers[0].source == globe.paste(arrow)
>>> assert chain.layers[1].source == arrow.paste(globe)
```

draw(***params*)

Bound version of `strdiags.draw()`.

Calling `x.draw(**params)` is equivalent to calling `strdiags.draw(x, **params)`.

draw_boundaries(***params*)

Bound version of `strdiags.draw_boundaries()`.

Calling `x.draw_boundaries(**params)` is equivalent to calling `strdiags.draw_boundaries(x, **params)`.

5.7.2 shapes.ShapeMap

class rewalt.shapes.**ShapeMap**(*ogmap*, ***params*)

Bases: *OgMap*

An overlay of `ogposets.OgMap` for total maps between *Shape* objects.

It is used to extend constructions of shapes functorially to their maps, in a way that is compatible with the unique representation of shapes by their underlying `ogposets.OgPoset` objects.

The most common *ShapeMap* objects are created by methods of *Shape* such as *Shape.boundary()* and *Shape.inflate()*, or of its subclasses, such as *Simplex.simplex_degeneracy()* or *Cube.cube_connection()*.

Nevertheless, occasionally we may need to define a map explicitly, in which case we first define an object `f` of class `ogposets.OgMap`, then upgrade it to a *ShapeMap* with the constructor `ShapeMap(f)`.

Parameters
ogmap (`ogposets.OgMap`) – A total map between shapes.

Keyword Arguments
wfcheck (`bool`) – Check whether the given map is a total map between shapes (default is `True`).

Methods

draw(**params)	Bound version of `strdiags.draw()`.
draw_boundaries(**params)	Bound version of `strdiags.draw_boundaries()`.
dual(*dims)	Functorial extension of `OgPoset.dual()` to maps of oriented graded posets.
generate_layering()	Shorthand for `source.generate_layering()`.
gray(*maps)	Functorial extension of `OgPoset.gray()` to maps of oriented graded posets.
join(*maps)	Functorial extension of `OgPoset.join()` to maps of oriented graded posets.
then(other, *others)	Returns the composite with other maps or pairs of maps of oriented graded posets, when defined.

Attributes

layers	Returns the current layering of the map's source, composed with the map.
rewrite_steps	Returns the sequence of rewrite steps associated to the current layering of the map's source, composed with the map.

then(*other*, **others*)

Returns the composite with other maps or pairs of maps of oriented graded posets, when defined.

If given an `OgMapPair` as argument, it returns the pair of composites of the map with each map in the pair.

Parameters

- **other** (`OgMap` | `OgMapPair`) – The first map or pair of maps to follow.
- ***others** (`OgMap` | `OgMapPair`, optional) – Any number of other maps or pair of maps to follow.

Returns

composite – The composite with all the other arguments.

Return type

`OgMap` | `OgMapPair`

Notes

If all the maps have type `shapes.ShapeMap`, their composite has the same type.

property layers

Returns the current layering of the map's source, composed with the map.

Returns

layers – The source's current layering, composed with the map.

Return type

`list[ShapeMap]`

property rewrite_steps

Returns the sequence of rewrite steps associated to the current layering of the map's source, composed with the map.

Returns
rewrite_steps – The source's current sequence of rewrite steps, composed with the map.

Return type
`list[ShapeMap]`

static gray(**maps*)

Functorial extension of `OgPoset.gray()` to maps of oriented graded posets.

This method can be called with the math operator `*`, that is, `fst * snd` is equivalent to `gray(fst, snd)`.

This static method can also be used as a bound method, that is, `fst.gray(*maps)` is equivalent to `gray(fst, *maps)`.

Parameters
***maps** (`OgMap`) – Any number of maps of oriented graded posets.

Returns
gray – The Gray product of the arguments.

Return type
`OgMap`

Notes

If all the arguments have type `shapes.ShapeMap`, so does their Gray product.

static join(**maps*)

Functorial extension of `OgPoset.join()` to maps of oriented graded posets.

This method can be called with the shift operators >> and <<, that is, `fst >> snd` is equivalent to `join(fst, snd)` and `fst << snd` is equivalent to `join(snd, fst)`.

This static method can also be used as a bound method, that is, `fst.join(*maps)` is equivalent to `join(fst, *maps)`.

Parameters
***maps** (`OgMap`) – Any number of maps of oriented graded posets.

Returns
join – The join of the arguments.

Return type
`OgMap`

Notes

If all the arguments have type `shapes.ShapeMap`, so does their join.

dual(**dims*)

Functorial extension of `OgPoset.dual()` to maps of oriented graded posets.

The dual in all dimensions can also be called with the negation operator ~, that is, `~ogmap` is equivalent to `ogmap.dual()`.

This static method can be also used as a bound method, that is, `self.dual(*dims)` is equivalent to `dual(self, *dims)`.

Parameters

- **ogmap** (`OgMap`) – A map of oriented graded posets.
- ***dims** (`int`) – Any number of dimensions; if none, defaults to *all* dimensions.

Returns

dual – The map dualised in the given dimensions.

Return type

`OgMap`

Notes

If the map is a *ShapeMap*, so is its dual.

generate_layering()

Shorthand for `source.generate_layering()`.

draw(***params*)

Bound version of `strdiags.draw()`.

Calling `f.draw(**params)` is equivalent to calling `strdiags.draw(f, **params)`.

draw_boundaries(***params*)

Bound version of `strdiags.draw_boundaries()`.

Calling `f.draw_boundaries(**params)` is equivalent to calling `strdiags.draw_boundaries(f, **params)`.

5.7.3 shapes.Simplex

class rewalt.shapes.**Simplex**

Bases: *Shape*

Subclass of *Shape* for oriented simplices.

The methods of this class provide a full implementation of the category of simplices, which is generated by the face and degeneracy maps between simplices one dimension apart.

Use *Shape.simplex()* to construct.

Examples

We create a 1-simplex (arrow), a 2-simplex (triangle), and a 3-simplex (tetrahedron).

```
>>> arrow = Shape.simplex(1)
>>> triangle = Shape.simplex(2)
>>> tetra = Shape.simplex(3)
```

We can then check some of the simplicial relations between degeneracy and face maps.

```
>>> map1 = triangle.simplex_degeneracy(2).then(
...     arrow.simplex_degeneracy(1))
>>> map2 = triangle.simplex_degeneracy(1).then(
...     arrow.simplex_degeneracy(1))
>>> assert map1 == map2
```

```
>>> map3 = tetra.simplex_face(2).then(
...     triangle.simplex_degeneracy(2))
>>> assert map3 == triangle.id()
```

```
>>> map4 = tetra.simplex_face(0).then(
...     triangle.simplex_degeneracy(2))
>>> map5 = arrow.simplex_degeneracy(1).then(
...     triangle.simplex_face(0))
>>> assert map4 == map5
```

Methods

simplex_degeneracy(k)	Returns one of the collapse (degeneracy) maps of the simplex one dimension higher.
simplex_face(k)	Returns one of the face inclusion maps of the simplex.

simplex_face(*k*)

Returns one of the face inclusion maps of the simplex.

Parameters
k (`int`) – The index of the face map, ranging from `0` to `self.dim`.

Returns
simplex_face – The face map.

Return type
ShapeMap

Raises
ValueError – If the index is out of range.

simplex_degeneracy(*k*)

Returns one of the collapse (degeneracy) maps of the simplex one dimension higher.

Parameters
k (`int`) – The index of the degeneracy map, ranging from `0` to `self.dim`.

Returns
simplex_degeneracy – The degeneracy map.

Return type
ShapeMap

Raises
ValueError – If the index is out of range.

5.7.4 shapes.Cube

class rewalt.shapes.**Cube**

Bases: *Shape*

Subclass of *Shape* for oriented cubes.

The methods of this class provide a full implementation of the category of cubes with connections, which is generated by the face, degeneracy, and connection maps between cubes one dimension apart.

Use *Shape.cube()* to construct.

Examples

We create a 1-cube (arrow), 2-cube (square), and 3-cube (cube).

```
>>> arrow = Shape.cube(1)
>>> square = Shape.cube(2)
>>> cube = Shape.cube(3)
```

We can then check some of the relations between cubical face, connection, and degeneracy maps.

```
>>> map1 = square.cube_degeneracy(2).then(
...     arrow.cube_degeneracy(1))
>>> map2 = square.cube_degeneracy(1).then(
...     arrow.cube_degeneracy(1))
>>> assert map1 == map2
```

```
>>> map3 = square.cube_face(0, '+').then(
...     cube.cube_face(2, '-'))
>>> map4 = square.cube_face(1, '-').then(
...     cube.cube_face(0, '+'))
>>> assert map3 == map4
```

```
>>> map5 = square.cube_connection(1, '-').then(
...     arrow.cube_connection(0, '-'))
>>> map6 = square.cube_connection(0, '-').then(
...     arrow.cube_connection(0, '-'))
>>> assert map5 == map6
```

Methods

cube_connection(k, sign)	Returns one of the "connection" collapse maps of the cube one dimension higher.
cube_degeneracy(k)	Returns one of the "degeneracy" collapse maps of the cube one dimension higher.
cube_face(k, sign)	Returns one of the face inclusion maps of the cube.

cube_face(*k*, *sign*)

Returns one of the face inclusion maps of the cube.

Parameters

- **k** (int) – Index of the face map, ranging from `0` to `self.dim - 1`.
- **sign** (str) – Side: `'-'` or `'+'`.

Returns
cube_face – The face map.

Return type
ShapeMap

Raises
ValueError – If the index is out of range.

`cube_degeneracy`(*k*)

Returns one of the "degeneracy" collapse maps of the cube one dimension higher.

Parameters
k (int) – The index of the degeneracy map, ranging from `0` to `self.dim`.

Returns
cube_degeneracy – The degeneracy map.

Return type
ShapeMap

Raises
ValueError – If the index is out of range.

`cube_connection`(*k*, *sign*)

Returns one of the "connection" collapse maps of the cube one dimension higher.

Parameters

- **k** (int) – Index of the connection map, ranging from `0` to `self.dim - 1`.
- **sign** (str) – Side: `'-'` or `'+'`.

Returns
cube_face – The connection map.

Return type
ShapeMap

Raises
ValueError – If the index is out of range.

5.8 ogposets

Implements oriented graded posets, their elements, subsets, and maps.

rewalt.ogposets.OgPoset(face_data, ...)	Class for oriented graded posets, that is, finite graded posets with an orientation, defined as a `{'-', '+'}`-labelling of the edges of their Hasse diagram.
rewalt.ogposets.OgMap(source, target[, mapping])	Class for (partial) maps of oriented graded posets, compatible with boundaries.
rewalt.ogposets.El(dim, pos)	Class for elements of an oriented graded poset.
rewalt.ogposets.GrSet(*elements)	Class for sets of elements of an oriented graded poset, graded by their dimension.
rewalt.ogposets.GrSubset(support, ambient, ...)	Class for graded subsets, that is, pairs of a *GrSet* and an "ambient" *OgPoset*, where the first is seen as a subset of the second.
rewalt.ogposets.Closed(support, ambient, ...)	Subclass of *GrSubset* for (downwards) closed subsets.
rewalt.ogposets.OgMapPair(fst, snd)	Class for pairs of maps of oriented graded posets.

5.8.1 ogposets.OgPoset

class `rewalt.ogposets.`**`OgPoset`**(*face_data*, *coface_data*, ***params*)

Bases: `object`

Class for oriented graded posets, that is, finite graded posets with an orientation, defined as a `{'-', '+'}`-labelling of the edges of their Hasse diagram.

In this implementation, the elements of a given dimension (grade) are linearly ordered, so that each element is identified by its dimension and the position in the linear order, encoded as an object of class *El*.

If `El(n, k)` covers `El(n-1, j)` with orientation `o`, we say that `El(n-1, j)` is an *input face* of `El(n, k)` if `o == '-'` and an *output face* of `El(n, k)` if `o == '+'`.

Defining an *OgPoset* directly is not recommended; use constructors of `shapes.Shape` instead.

Parameters

- **face_data** (`list[list[dict[set[int]]]]`) – Data encoding the oriented graded poset as follows: `j in face_data[n][k][o]` if and only if `El(n, k)` covers `El(n-1, j)` with orientation `o`, where `o == '-'` or `o == '+'`.
- **coface_data** (`list[list[dict[set[int]]]]`) – Data encoding the oriented graded poset as follows: `j in coface_data[n][k][o]` if and only if `El(n+1, j)` covers `El(n, k)` with orientation `o`, where `o == '-'` or `o == '+'`.

Keyword Arguments

- **wfcheck** (`bool`) – Check that the data is well-formed (default is `True`)
- **matchcheck** (`bool`) – Check that *face_data* and *coface_data* match (default is `True`)

Notes

Each of *face_data*, *coface_data* determines the other uniquely. There is an alternative constructor `from_face_data()` that computes *coface_data* from *face_data*.

Examples

Let us construct explicitly the "oriented face poset" of an arrow, or directed edge.

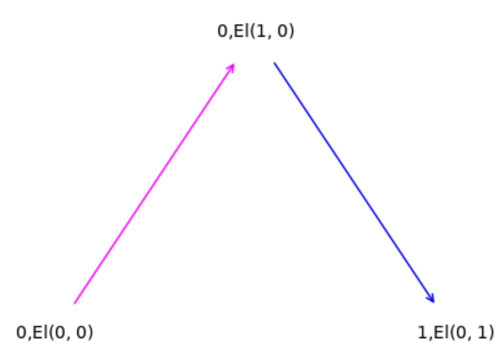
```
>>> face_data = [
...     [
...         {'-': set(), '+': set()},
...         {'-': set(), '+': set()},
...     ], [
...         {'-': {0}, '+': {1}}
...     ]]
>>> coface_data = [
...     [
...         {'-': {0}, '+': set()},
...         {'-': set(), '+': {0}},
...     ], [
...         {'-': set(), '+': set()}
...     ]]
>>> arrow = OgPoset(face_data, coface_data)
```

This has two 0-dimensional elements and one 1-dimensional element.

```
>>> arrow.size
[2, 1]
```

We can visualise its Hasse diagram, with orientation conveyed by colour (magenta for input, blue for output) and direction of arrows.

```
>>> arrow.hasse(path='docs/_static/img/OgPoset_arrow.png')
```



We can ask for the faces and cofaces of a specific element.

```
>>> arrow.faces(El(1, 0), '-')
GrSet(El(0, 0))
>>> arrow.cofaces(El(0, 1))
GrSet(El(1, 0))
```

We can construct other oriented graded posets using various operations, such as suspensions, Gray products, joins, or duals.

```
>>> print(arrow.suspend())
OgPoset with [2, 2, 1] elements
>>> print(arrow * arrow)
OgPoset with [4, 4, 1] elements
>>> print(arrow >> arrow)
OgPoset with [4, 6, 4, 1] elements
>>> print(arrow.dual())
OgPoset with [2, 1] elements
```

Methods

all()	Returns the closed subset of all elements.
bot()	Returns the object augmented with a bottom element, covered with orientation `'+'`.
boundary([sign, dim])	Returns the inclusion of the boundary of a given orientation and dimension into the object.
co()	Returns the *dual()* in all *even* dimensions.
cofaces(element[, sign])	Returns the cofaces of an element as a graded set.
coproduct(fst, snd)	Returns the coproduct cospan of two oriented graded posets.
disjoint_union(fst, snd)	Returns the disjoint union of two oriented graded posets, that is, the target of their coproduct cospan.
dual(ogp, *dims)	Returns an oriented graded poset with orientations reversed in given dimensions.
empty()	Returns the initial oriented graded poset, with no elements.
faces(element[, sign])	Returns the faces of an element as a graded set.
from_face_data(face_data, **params)	Alternative constructor computing *coface_data* from *face_data*.
gray(*ogps)	Returns the Gray product of any number of oriented graded posets.
hasse(**params)	Bound version of `hasse.draw()`.
id()	Returns the identity map on the object.
image(ogmap)	Returns the image of the object through a map.
join(*ogps)	Returns the join of any number of oriented graded posets.
maximal()	Returns the subset of maximal elements, that is, those that are not covered by any elements.
none()	Returns the empty closed subset.
op()	Returns the *dual()* in all *odd* dimensions.
point()	Returns the terminal oriented graded poset, with a single element.
suspend(ogp[, n])	Returns the n-fold suspension of an oriented graded poset.
underset(*elements)	Returns the closure of a set of elements in the object.

Attributes

as_chain	Returns a "chain complex" representation of the face data.
coface_data	Returns the coface data as given to the object constructor.
dim	Returns the dimension of the object, that is, the maximum of the dimensions of its elements.
face_data	Returns the face data as given to the object constructor.
input	Alias for `boundary('-')`.
output	Alias for `boundary('+')`.
size	Returns the number of elements in each dimension as a list.

property face_data

Returns the face data as given to the object constructor.

An *OgPoset* is meant to be immutable; create a new object if you need to modify the face data.

Returns
face_data – The face data as given to the object constructor.

Return type
`list[list[dict[set[int]]]]`

property coface_data

Returns the coface data as given to the object constructor.

An *OgPoset* is meant to be immutable; create a new object if you need to modify the coface data.

Returns
coface_data – The coface data as given to the object constructor.

Return type
`list[list[dict[set[int]]]]`

property size

Returns the number of elements in each dimension as a list.

Returns
size – The `k` th entry is the number of `k` -dimensional elements.

Return type
`list[int]`

property dim

Returns the dimension of the object, that is, the maximum of the dimensions of its elements.

Returns
dim – The dimension of the object.

Return type
`int`

property as_chain

Returns a "chain complex" representation of the face data.

Returns
chain – Encodes the face data as follows: `chain[n][i][j] == 1` if `El(n, i)` is an output face of `El(n+1, j)`, `-1` if it is an input face, `0` otherwise.

Return type
`list[numpy.array]`

all()

Returns the closed subset of all elements.

Returns
all – The closed subset of all elements of the object.

Return type
Closed

none()

Returns the empty closed subset.

Returns
none – The closed subset with no elements.

Return type
Closed

`underset`(**elements*)
Returns the closure of a set of elements in the object.

Parameters
elements (*El*) – Any number of elements.

Returns
underset – The downwards closure of the given elements.

Return type
Closed

`maximal`()
Returns the subset of maximal elements, that is, those that are not covered by any elements.

Returns
maximal – The subset of maximal elements.

Return type
GrSubset

`faces`(*element*, *sign=None*)
Returns the faces of an element as a graded set.

Parameters

- **element** (*El*) – An element of the object.
- **sign** (`str`, optional) – Orientation: `'-'` for input, `'+'` for output, `None` (default) for both.

Returns
faces – The set of faces of the given element.

Return type
GrSet

`cofaces`(*element*, *sign=None*)
Returns the cofaces of an element as a graded set.

Parameters

- **element** (*El*) – An element of the object.
- **sign** (`str`, optional) – Orientation: `'-'` for input, `'+'` for output, `None` (default) for both.

Returns
cofaces – The set of cofaces of the given element.

Return type
GrSet

`id`()
Returns the identity map on the object.

Returns
id – The identity map on the object.

Return type
`OgMap`

`image`(*ogmap*)

Returns the image of the object through a map.

Parameters
ogmap (`OgMap`) – A map from the object to another `OgPoset`.

Returns
image – The image of the object through the given map.

Return type
`Closed`

`boundary`(*sign=None*, *dim=None*)

Returns the inclusion of the boundary of a given orientation and dimension into the object.

Parameters

- **sign** (`str`, optional) – Orientation: `'-'` for input, `'+'` for output, `None` (default) for both.
- **dim** (`int`, optional) – Dimension of the boundary (default is `self.dim - 1`).

Returns
boundary – The inclusion of the requested boundary into the object.

Return type
`OgMap`

`property input`

Alias for `boundary('-')`.

`property output`

Alias for `boundary('+')`.

`classmethod from_face_data`(*face_data*, ***params*)

Alternative constructor computing *coface_data* from *face_data*.

Parameters
face_data (`list[list[dict[set[int]]]]`) – As in the main constructor.

Keyword Arguments
wfcheck (`bool`) – Check that the data is well-formed (default is `True`).

`static empty()`

Returns the initial oriented graded poset, with no elements.

Returns
empty – The empty oriented graded poset.

Return type
`OgPoset`

`static point()`

Returns the terminal oriented graded poset, with a single element.

Returns
point – The oriented graded poset with a single element.

Return type
`OgPoset`

static `coproduct`(*fst*, *snd*)

Returns the coproduct cospan of two oriented graded posets.

Parameters

- `fst` (*OgPoset*) – The first factor of the coproduct.
- `snd` (*OgPoset*) – The second factor of the coproduct.

Returns

coproduct – The coproduct cospan.

Return type

OgMapPair

static `disjoint_union`(*fst*, *snd*)

Returns the disjoint union of two oriented graded posets, that is, the target of their coproduct cospan.

This method can be called with the math operator `+`, that is, `fst + snd` is equivalent to `disjoint_union(fst, snd)`.

Parameters

- `fst` (*OgPoset*) – The first factor of the disjoint union.
- `snd` (*OgPoset*) – The second factor of the disjoint union.

Returns

disjoint_union – The disjoint union of the two.

Return type

OgPoset

static `suspend`(*ogp*, *n=1*)

Returns the n-fold suspension of an oriented graded poset.

This static method can be also used as a bound method after an object is initialised, that is, `ogp.suspend(n)` is equivalent to `suspend(ogp, n)`.

Parameters

- `ogp` (*OgPoset*) – The object to suspend.
- `n` (`int`, optional) – The number of iterations of the suspension (default is 1).

Returns

suspension – The suspended object.

Return type

OgPoset

static `gray`(**ogps*)

Returns the Gray product of any number of oriented graded posets.

This method can be called with the math operator `*`, that is, `fst * snd` is equivalent to `gray(fst, snd)`.

This static method can also be used as a bound method after an object is initialised, that is, `fst.gray(*ogps)` is equivalent to `gray(fst, *ogps)`.

Parameters

`*ogps` (*OgPoset*) – Any number of oriented graded posets.

Returns

gray – The Gray product of the arguments.

Return type
OgPoset

`bot()`

Returns the object augmented with a bottom element, covered with orientation `'+'`.

Returns
bot – The object augmented with a bottom element.

Return type
OgPoset

`static join(*ogps)`

Returns the join of any number of oriented graded posets.

This method can be called with the shift operators >> and <<, that is, `fst >> snd` is equivalent to `join(fst, snd)` and `fst << snd` is equivalent to `join(snd, fst)`.

This static method can also be used as a bound method after an object is initialised, that is, `fst.join(*ogps)` is equivalent to `join(fst, *ogps)`.

Parameters
***ogps** (*OgPoset*) – Any number of oriented graded posets.

Returns
join – The join of the arguments.

Return type
OgPoset

`static dual(ogp, *dims)`

Returns an oriented graded poset with orientations reversed in given dimensions.

The dual in all dimensions can also be called with the bit negation operator ~, that is, `~ogp` is equivalent to `ogp.dual()`.

This static method can be also used as a bound method after an object is initialised, that is, `ogp.dual(*dims)` is equivalent to `dual(ogp, *dims)`.

Parameters

- **ogp** (*OgPoset*) – An oriented graded poset.
- ***dims** (`int`) – Any number of dimensions; if none, defaults to *all* dimensions.

Returns
dual – The oriented graded poset, dualised in the given dimensions.

Return type
OgPoset

`op()`

Returns the `dual()` in all *odd* dimensions.

`co()`

Returns the `dual()` in all *even* dimensions.

`hasse(**params)`

Bound version of `hasse.draw()`.

Calling `x.hasse(**params)` is equivalent to calling `hasse.draw(x, **params)`.

5.8.2 ogposets.OgMap

class rewalt.ogposets.**OgMap**(*source*, *target*, *mapping=None*, ***params*)

Bases: `object`

Class for (partial) maps of oriented graded posets, compatible with boundaries.

To define a map on one element, it must have been defined on all elements below it. The assignment can be made all at once, or element by element. Once the map has been defined on an element, the assignment cannot be modified.

Parameters

- **source** (*OgPoset*) – The source (domain) of the map.
- **target** (*OgPoset*) – The target (codomain) of the map.
- **mapping** (`list[list[El]]`, optional) – Data specifying the partial map as follows: `mapping[n][k] == El(m, j)` if the map sends `El(n, k)` to `El(m, j)`, and `None` if the map is undefined on `El(n, k)` (default is the nowhere defined map).

Keyword Arguments

wfcheck (`bool`) – Check whether the data defines a well-formed map compatible with all boundaries (default is `True`).

Notes

Objects of the class are callable on objects of type *El* (returning the image of an element) and of type *GrSubset* and *GrSet* (returning the image of a subset of their source).

Examples

Let us create two simple oriented graded posets, the "point" and the "arrow".

```
>>> point = OgPoset.point()
>>> arrow = point >> point
```

We define the map that collapses the arrow onto the point. First we create a nowhere defined map.

```
>>> collapse = OgMap(arrow, point)
>>> assert not collapse.istotal
```

We declare the assignment first on the 0-dimensional elements, then on the single 1-dimensional element. Trying to do otherwise results in a `ValueError`.

```
>>> collapse[El(0, 0)] = El(0, 0)
>>> collapse[El(0, 1)] = El(0, 0)
>>> collapse[El(1, 0)] = El(0, 0)
```

We can check various properties of the map.

```
>>> assert collapse.istotal
>>> assert collapse.issurjective
>>> assert not collapse.isinjective
```

Alternatively, we could have defined the map all at once, as follows.

```
>>> mapping = [[El(0, 0), El(0, 0)], [El(0, 0)]]
>>> assert collapse == OgMap(arrow, point, mapping)
```

Methods

`bot()`	Functorial extension of `OgPoset.bot()` to maps.
`boundary([sign, dim])`	Returns the map restricted to a specified boundary of its source.
`co()`	Returns the dual in all *even* dimensions.
`dual(ogmap, *dims)`	Functorial extension of `OgPoset.dual()` to maps of oriented graded posets.
`gray(*maps)`	Functorial extension of `OgPoset.gray()` to maps of oriented graded posets.
`hasse(**params)`	Bound version of `hasse.draw()`.
`image()`	Returns the image of the map.
`inv()`	Returns the inverse of the map if it is an isomorphism.
`isdefined(element)`	Returns whether the map is defined on a given element.
`join(*maps)`	Functorial extension of `OgPoset.join()` to maps of oriented graded posets.
`op()`	Returns the dual in all *odd* dimensions.
`then(other, *others)`	Returns the composite with other maps or pairs of maps of oriented graded posets, when defined.

Attributes

`input`	Alias for `boundary('-')`.
`isinjective`	Returns whether the map is injective.
`isiso`	Returns whether the map is an isomorphism, that is, total, injective, and surjective.
`issurjective`	Returns whether the map is surjective.
`istotal`	Returns whether the map is total.
`mapping`	Returns the data specifying the map's assignments.
`output`	Alias for `boundary('+')`.
`source`	Returns the source (domain) of the map.
`target`	Returns the target (codomain) of the map.

property source

Returns the source (domain) of the map.

Returns
source – The source of the map.

Return type
`OgPoset`

property target

Returns the target (codomain) of the map.

Returns
target – The target of the map.

Return type
OgPoset

property mapping

Returns the data specifying the map's assignments.

Returns
mapping – The mapping data.

Return type
`list[list[El]]`

property istotal

Returns whether the map is total.

Returns
istotal – `True` if and only if the map is total.

Return type
`bool`

property isinjective

Returns whether the map is injective.

Returns
isinjective – `True` if and only if the map is injective.

Return type
`bool`

property issurjective

Returns whether the map is surjective.

Returns
issurjective – `True` if and only if the map is surjective.

Return type
`bool`

property isiso

Returns whether the map is an isomorphism, that is, total, injective, and surjective.

Returns
isiso – `True` if and only if the map is an isomorphism.

Return type
`bool`

isdefined(*element*)

Returns whether the map is defined on a given element.

Parameters
element (*El*) – The element to check.

Returns
isdefined – `True` if and only if the map is defined on the element.

Return type
`bool`

then(*other*, **others*)

Returns the composite with other maps or pairs of maps of oriented graded posets, when defined.

If given an `OgMapPair` as argument, it returns the pair of composites of the map with each map in the pair.

Parameters

- **other** (`OgMap` | `OgMapPair`) – The first map or pair of maps to follow.
- ***others** (`OgMap` | `OgMapPair`, optional) – Any number of other maps or pair of maps to follow.

Returns

composite – The composite with all the other arguments.

Return type

`OgMap` | `OgMapPair`

Notes

If all the maps have type `shapes.ShapeMap`, their composite has the same type.

inv()

Returns the inverse of the map if it is an isomorphism.

Returns

inv – The inverse of the map, if defined.

Return type

`OgMap`

Raises

ValueError – If the map is not an isomorphism.

image()

Returns the image of the map.

Returns

image – The image of the source through the map.

Return type

`Closed`

boundary(*sign=None*, *dim=None*)

Returns the map restricted to a specified boundary of its source.

Parameters

- **sign** (`str`, optional) – Orientation: `'-'` for input, `'+'` for output, `None` (default) for both.
- **dim** (`int`, optional) – Dimension of the boundary (default is `self.dim - 1`).

Returns

boundary – The map restricted to the requested boundary.

Return type

`OgMap`

property input

Alias for `boundary('-')`.

property output

Alias for `boundary('+')`.

bot()

Functorial extension of *OgPoset.bot()* to maps.

Returns
bot – The map extended to a map from `source.bot` to `target.bot`.

Return type
OgMap

static gray(**maps*)

Functorial extension of *OgPoset.gray()* to maps of oriented graded posets.

This method can be called with the math operator `*`, that is, `fst * snd` is equivalent to `gray(fst, snd)`.

This static method can also be used as a bound method, that is, `fst.gray(*maps)` is equivalent to `gray(fst, *maps)`.

Parameters
***maps** (*OgMap*) – Any number of maps of oriented graded posets.

Returns
gray – The Gray product of the arguments.

Return type
OgMap

Notes

If all the arguments have type `shapes.ShapeMap`, so does their Gray product.

static join(**maps*)

Functorial extension of *OgPoset.join()* to maps of oriented graded posets.

This method can be called with the shift operators >> and <<, that is, `fst >> snd` is equivalent to `join(fst, snd)` and `fst << snd` is equivalent to `join(snd, fst)`.

This static method can also be used as a bound method, that is, `fst.join(*maps)` is equivalent to `join(fst, *maps)`.

Parameters
***maps** (*OgMap*) – Any number of maps of oriented graded posets.

Returns
join – The join of the arguments.

Return type
OgMap

Notes

If all the arguments have type `shapes.ShapeMap`, so does their join.

static dual(*ogmap*, **dims*)

Functorial extension of *OgPoset.dual()* to maps of oriented graded posets.

The dual in all dimensions can also be called with the negation operator ~, that is, `~ogmap` is equivalent to `ogmap.dual()`.

This static method can be also used as a bound method, that is, `self.dual(*dims)` is equivalent to `dual(self, *dims)`.

Parameters

- **ogmap** (*OgMap*) – A map of oriented graded posets.
- ***dims** (`int`) – Any number of dimensions; if none, defaults to *all* dimensions.

Returns
dual – The map dualised in the given dimensions.

Return type
OgMap

Notes

If the map is a `ShapeMap`, so is its dual.

op()

Returns the dual in all *odd* dimensions.

co()

Returns the dual in all *even* dimensions.

hasse(***params*)

Bound version of `hasse.draw()`.

Calling `f.hasse(**params)` is equivalent to calling `hasse.draw(f, **params)`.

5.8.3 ogposets.El

class rewalt.ogposets.**El**(*dim*, *pos*)

Bases: `tuple`

Class for elements of an oriented graded poset.

An element is encoded as a pair of non-negative integers: the dimension of the element, and its position in a linear order of elements of the given dimension.

Parameters

- **dim** (`int`) – The dimension of the element.
- **pos** (`int`) – The position of the element.

Examples

```
>>> x = El(2, 3)
>>> x.dim
2
>>> x.pos
3
```

Methods

shifted(k)	Returns the element of the same dimension, with position shifted by a given integer.

Attributes

dim	Returns the dimension of the element.
pos	Returns the position of the element.

property dim

Returns the dimension of the element.

Returns
dim – The dimension of the element.

Return type
`int`

property pos

Returns the position of the element.

Returns
pos – The position of the element

Return type
`int`

shifted(*k*)

Returns the element of the same dimension, with position shifted by a given integer.

Parameters
k (`int`) – The shift in position.

Returns
shifted – The shifted element.

Return type
El

5.8.4 ogposets.GrSet

class rewalt.ogposets.**GrSet**(**elements*)

Bases: object

Class for sets of elements of an oriented graded poset, graded by their dimension.

Objects of the class behave as sets; several methods of the set class are supported. However the data is stored in a way that allows fast access to elements of a given dimension.

Parameters
elements (*El*) – Any number of elements.

Examples

We create an instance by listing elements; repetitions do not count.

```
>>> test = GrSet(El(0, 2), El(0, 2), El(0, 3), El(2, 0), El(3, 1))
>>> test
GrSet(El(0, 2), El(0, 3), El(2, 0), El(3, 1))
>>> len(test)
4
```

We can access the subsets of elements of given dimensions with indexer operators. These support slice syntax.

```
>>> test[0]
GrSet(El(0, 2), El(0, 3))
>>> test[0:3]
GrSet(El(0, 2), El(0, 3), El(2, 0))
```

The iterator for graded sets goes through the elements in increasing dimension and, for each dimension, in increasing position.

```
>>> for x in test:
...     print(x)
...
El(0, 2)
El(0, 3)
El(2, 0)
El(3, 1)
```

We can add and remove elements.

```
>>> test.remove(El(0, 3))
>>> test
GrSet(El(0, 2), El(2, 0), El(3, 1))
>>> test.add(El(1, 1))
>>> test
GrSet(El(0, 2), El(1, 1), El(2, 0), El(3, 1))
```

Set methods such as union, difference, and intersection are available with the same syntax.

Methods

`add`(element)	Adds a single element.
`copy`()	Returns a copy of the graded set.
`difference`(other)	Returns the difference of the graded set with another graded set.
`intersection`(*others)	Returns the intersection of the graded set with other graded sets.
`isdisjoint`(other)	Returns whether the graded set is disjoint from another.
`issubset`(other)	Returns whether the graded set is a subset of another.
`remove`(element)	Removes a single element.
`union`(*others)	Returns the union of the graded set with other graded sets.

Attributes

`as_list`	Returns the list of elements in increasing dimension, and, dimensionwise, in increasing position.
`as_set`	Returns a Python set containing the same elements.
`dim`	Returns the maximal dimension in which the graded set is not empty, or -1 if it is empty.
`grades`	Returns the list of dimensions in which the graded set is not empty.

property grades

Returns the list of dimensions in which the graded set is not empty.

Returns
grades – The list of dimensions in which the graded set is not empty.

Return type
`list[int]`

property dim

Returns the maximal dimension in which the graded set is not empty, or -1 if it is empty.

Returns
dim – The maximal dimension in which the graded set is not empty.

Return type
`int`

property as_set

Returns a Python set containing the same elements.

Returns
as_set – A Python set containing the same elements.

Return type
`set[El]`

property as_list

Returns the list of elements in increasing dimension, and, dimensionwise, in increasing position.

Returns
as_list – A list containing the same elements.

Return type
`list[El]`

`add`(*element*)
Adds a single element.

Parameters
`element` (*El*) – The element to add.

`remove`(*element*)
Removes a single element.

Parameters
`element` (*El*) – The element to remove.

`union`(**others*)
Returns the union of the graded set with other graded sets.

Parameters
`*others` (*GrSet*) – Any number of graded sets.

Returns
union – The union of the graded set with all the given others.

Return type
GrSet

`intersection`(**others*)
Returns the intersection of the graded set with other graded sets.

Parameters
`*others` (*GrSet*) – Any number of graded sets.

Returns
intersection – The intersection of the graded set with all the given others.

Return type
GrSet

`difference`(*other*)
Returns the difference of the graded set with another graded set.

Parameters
`other` (*GrSet*) – Another graded set.

Returns
difference – The difference between the two graded sets.

Return type
GrSet

`issubset`(*other*)
Returns whether the graded set is a subset of another.

Parameters
`other` (*GrSet*) – Another graded set.

Returns
issubset – `True` if and only *self* is a subset of *other*.

Return type
`bool`

`isdisjoint`(*other*)

Returns whether the graded set is disjoint from another.

Parameters
other (*GrSet*) – Another graded set.

Returns
isdisjoint – `True` if and only *self* and *other* are disjoint.

Return type
`bool`

`copy`()

Returns a copy of the graded set.

Returns
copy – A copy of the graded set.

Return type
GrSet

5.8.5 ogposets.GrSubset

class `rewalt.ogposets.GrSubset`(*support*, *ambient*, ***params*)

Bases: `object`

Class for graded subsets, that is, pairs of a *GrSet* and an "ambient" *OgPoset*, where the first is seen as a subset of the second.

While objects of the class *GrSet* are mutable, once they are tied to an *OgPoset* they should be treated as immutable.

Parameters

- **support** (*GrSet*) – The underlying graded set.
- **ambient** (*OgPoset*) – The ambient oriented graded poset.

Keyword Arguments
wfcheck (`bool`) – Check whether the support is a well-formed subset of the ambient, that is, it has no elements out of range (default is `True`).

Notes

Two graded subsets are equal if and only if they have the same elements, *and* they are subsets of the same *OgPoset*.

Examples

We create an oriented graded poset and a pair of graded sets.

```
>>> point = OgPoset.point()
>>> triangle = point >> point >> point
>>> set1 = GrSet(El(1, 1), El(0, 1))
>>> set2 = GrSet(El(0, 3))
```

We can attach `set1` to `triangle` as a subset.

```
>>> subset = GrSubset(set1, triangle)
>>> assert subset.support == set1
```

Trying to do the same with `set2` returns a `ValueError` because `El(0, 3)` is out of range.

We can compute the downwards closure of `set1` in `triangle`.

```
>>> subset.closure().support
GrSet(El(0, 0), El(0, 1), El(0, 2), El(1, 1))
```

All the set-theoretic operations apply to graded subsets as long as they have the same ambient *OgPoset*.

Methods

closure()	Returns the downwards closure of the graded subset.
difference(other)	Returns the difference with another graded subset of the same oriented graded poset.
image(ogmap)	Returns the image of the graded subset through a map of oriented graded posets.
intersection(*others)	Returns the intersection with other graded subsets of the same oriented graded poset.
isdisjoint(other)	Returns whether the object is disjoint from another graded subset of the same oriented graded poset.
issubset(other)	Returns whether the object is a subset of another subset of the same oriented graded poset.
union(*others)	Returns the union with other graded subsets of the same oriented graded poset.

Attributes

ambient	Returns the ambient oriented graded poset.
dim	Shorthand for `support.dim`.
isclosed	Returns whether the subset is (downwards) closed.
support	Returns the underlying graded set (the "support" of the subset).

property support

Returns the underlying graded set (the "support" of the subset).

Returns

support – The underlying graded set.

Return type
GrSet

property ambient

Returns the ambient oriented graded poset.

Returns
ambient – The ambient oriented graded poset.

Return type
OgPoset

property dim

Shorthand for `support.dim`.

property isclosed

Returns whether the subset is (downwards) closed.

Returns
isclosed – `True` if and only if the subset is downwards closed.

Return type
`bool`

union(**others*)

Returns the union with other graded subsets of the same oriented graded poset.

Parameters
***others** (*GrSubset*) – Any number of graded subsets of the same oriented graded poset.

Returns
union – The union of the graded subset with all the given others.

Return type
GrSubset

Notes

If all the arguments have type *Closed*, the union also has type *Closed*.

intersection(**others*)

Returns the intersection with other graded subsets of the same oriented graded poset.

Parameters
***others** (*GrSubset*) – Any number of graded subsets of the same oriented graded poset.

Returns
intersection – The intersection of the graded subset with all the given others.

Return type
GrSubset

Notes

If all the arguments have type *Closed*, the intersection also has type *Closed*.

difference(*other*)

Returns the difference with another graded subset of the same oriented graded poset.

Parameters
other (*GrSubset*) – Another graded subset of the same oriented graded poset.

Returns
difference – The difference between the two graded subsets.

Return type
GrSubset

issubset(*other*)

Returns whether the object is a subset of another subset of the same oriented graded poset.

Parameters
other (*GrSubset*) – Another graded subset of the same oriented graded poset.

Returns
issubset – `True` if and only *self* is a subset of *other*.

Return type
`bool`

isdisjoint(*other*)

Returns whether the object is disjoint from another graded subset of the same oriented graded poset.

Parameters
other (*GrSubset*) – Another graded subset of the same oriented graded poset.

Returns
issubset – `True` if and only *self* and *other* are disjoint.

Return type
`bool`

closure()

Returns the downwards closure of the graded subset.

Returns
closure – The downwards closure of the subset.

Return type
Closed

image(*ogmap*)

Returns the image of the graded subset through a map of oriented graded posets.

Parameters
ogmap (*OgMap*) – A map from the ambient to another *OgPoset*.

Returns
image – The image of the subset through the given map.

Return type
GrSubset

Notes

If the object has type *Closed*, its image has also type *Closed*.

5.8.6 ogposets.Closed

class rewalt.ogposets.**Closed**(*support*, *ambient*, ***params*)

Bases: *GrSubset*

Subclass of *GrSubset* for (downwards) closed subsets.

Implements a number of methods that do not make sense for non-closed subsets, in particular those computing input and output boundaries in each dimension.

Parameters

- **support** (*GrSet*) – The underlying graded set.
- **ambient** (*OgPoset*) – The ambient oriented graded poset.

Keyword Arguments

wfcheck (bool) – Check whether the support is a well-formed, closed subset of the ambient (default is True).

Notes

There is an alternative constructor *subset()* which takes a *GrSubset*, and "upgrades" it to a *Closed* if it is downwards closed.

Examples

After creating an oriented graded poset, we can obtain the closed subset of *all* its elements with *OgPoset.all()*.

```
>>> point = OgPoset.point()
>>> triangle = point >> point >> point
>>> all = triangle.all()
```

We can compute its input and output boundary…

```
>>> all_in = all.input
>>> all_out = all.output
```

And since all happens to be a *molecule*, we can check the "globular" relations.

```
>>> assert all_in.input == all_out.input
>>> assert all_in.output == all_out.output
```

Methods

boundary([sign, dim])	Returns the boundary of a given orientation and dimension.
boundary_max([sign, dim])	Returns the subset of maximal elements of the boundary of a given orientation and dimension.
maximal()	Returns the subset of maximal elements, that is, those that are not covered by any other element in the closed subset.
subset(grsubset, **params)	Alternative constructor that promotes a *GrSubset* to a *Closed*.

Attributes

as_map	Returns an injective map representing the inclusion of the closed subset in the ambient.
input	Alias for `boundary('-')`.
ispure	Returns whether the maximal elements of the closed subset all have the same dimension.
isround	Returns whether the closed subset is round ("has spherical boundary").
output	Alias for `boundary('+')`.

property as_map

Returns an injective map representing the inclusion of the closed subset in the ambient.

Returns
as_map – A map of oriented graded posets representing the inclusion of the closed subset.

Return type
OgMap

property ispure

Returns whether the maximal elements of the closed subset all have the same dimension.

Returns
ispure – `True` if and only if the subset is pure.

Return type
`bool`

property isround

Returns whether the closed subset is round ("has spherical boundary").

This means that, for all `k` smaller than the dimension of the subset, the intersection of its input `k`-boundary and of its output `k`-boundary is equal to its `(k-1)`- boundary.

Returns
isround – `True` if and only if the subset is round.

Return type
`bool`

maximal()

Returns the subset of maximal elements, that is, those that are not covered by any other element in the closed subset.

Returns
maximal – The subset of maximal elements.

Return type
GrSubset

boundary_max(*sign=None*, *dim=None*)

Returns the subset of maximal elements of the boundary of a given orientation and dimension.

Parameters

- **sign** (`str`, optional) – Orientation: `'-'` for input, `'+'` for output, `None` (default) for both.
- **dim** (`int`, optional) – Dimension of the boundary (default is `self.dim - 1`).

Returns
boundary_max – The maximal elements of the requested boundary.

Return type
GrSubset

boundary(*sign=None*, *dim=None*)

Returns the boundary of a given orientation and dimension.

Parameters

- **sign** (`str`, optional) – Orientation: `'-'` for input, `'+'` for output, `None` (default) for both.
- **dim** (`int`, optional) – Dimension of the boundary (default is `self.dim - 1`).

Returns
boundary – The requested boundary subset.

Return type
Closed

property input

Alias for `boundary('-')`.

property output

Alias for `boundary('+')`.

static subset(*grsubset*, ***params*)

Alternative constructor that promotes a *GrSubset* to a *Closed*.

Parameters
grsubset (*GrSubset*) – The subset to promote.

Keyword Arguments
wfcheck (`bool`) – Check whether the subset is downwards closed (default is `True`).

5.8.7 ogposets.OgMapPair

class `rewalt.ogposets.OgMapPair`(*fst*, *snd*)

Bases: `tuple`

Class for pairs of maps of oriented graded posets.

This is used as the argument and/or return type of pushouts and coequalisers, which play a prominent role in the theory.

Parameters

- **fst** (*OgMap*) – The first map in the pair.
- **snd** (*OgMap*) – The second map in the pair.

Methods

coequaliser(**params)	Returns the coequaliser of a parallel pair of total maps, if it exists.
pushout(**params)	Returns the pushout of a span of total maps, if it exists.
then(other, *others)	Returns the composite with other maps or pairs of maps of oriented graded posets, when defined.

Attributes

fst	Returns the first map in the pair.
iscospan	Returns whether the pair is a cospan (has a common target).
isinjective	Returns whether both maps are injective.
isparallel	Returns whether the pair is parallel (both a span and a cospan).
isspan	Returns whether the pair is a span (has a common source).
issurjective	Returns whether both maps are surjective.
istotal	Returns whether both maps are total.
snd	Returns the second map in the pair.
source	Returns the pair of sources of the pair of maps, or, if a span, their common source.
target	Returns the pair of targets of the pair of maps, or, if a cospan, their common target.

property fst

Returns the first map in the pair.

Returns
fst – The first map in the pair.

Return type
OgMap

property snd

Returns the second map in the pair.

Returns
snd – The second map in the pair.

Return type
OgMap

property source

Returns the pair of sources of the pair of maps, or, if a span, their common source.

Returns
source – The source or sources of the pair.

Return type
OgMap | `tuple[OgMap]`

property target

Returns the pair of targets of the pair of maps, or, if a cospan, their common target.

Returns
target – The target or targets of the pair.

Return type
OgMap | `tuple[OgMap]`

property isspan

Returns whether the pair is a span (has a common source).

Returns
isspan – `True` if and only if the pair is a span.

Return type
`bool`

property iscospan

Returns whether the pair is a cospan (has a common target).

Returns
iscospan – `True` if and only if the pair is a cospan.

Return type
`bool`

property isparallel

Returns whether the pair is parallel (both a span and a cospan).

Returns
isparallel – `True` if and only if the pair is parallel.

Return type
`bool`

property istotal

Returns whether both maps are total.

Returns
istotal – `True` if and only if both maps are total.

Return type
`bool`

property isinjective

Returns whether both maps are injective.

Returns
isinjective – `True` if and only if both maps are injective.

Return type
`bool`

property issurjective

Returns whether both maps are surjective.

Returns
issurjective – `True` if and only if both maps are surjective.

Return type
`bool`

then(*other*, **others*)

Returns the composite with other maps or pairs of maps of oriented graded posets, when defined.

If given two pairs, it composes the first map with the first map, and the second map with the second map. If given a pair and a map, it composes both maps in the pair with the map.

Parameters

- **other** (*OgMap* | *OgMapPair*) – The first map or pair of maps to follow.
- **others** (*OgMap* | *OgMapPair*, optional) – Any number of other maps or pair of maps to follow.

Returns
composite – The composite with all the other arguments.

Return type
OgMapPair

coequaliser(***params*)

Returns the coequaliser of a parallel pair of total maps, if it exists.

Keyword Arguments
wfcheck (`bool`) – Check whether the coequaliser is well-defined.

Returns
coequaliser – The coequaliser of the pair of maps.

Return type
OgMap

Raises
ValueError – If the pair is not total and parallel.

pushout(***params*)

Returns the pushout of a span of total maps, if it exists.

Pushouts do not always exist in the category of oriented graded posets and maps; however, pushouts of injective (total) maps do always exist.

Keyword Arguments
wfcheck (`bool`) – Check whether the pushout is well-defined.

Returns
pushout – The pushout cospan of the pair of maps.

Return type
OgMapPair

Raises
`ValueError` – If the pair is not total and a span.

5.9 strdiags

Implements string diagram visualisations.

rewalt.strdiags.StrDiag(diagram)	Class for string diagram visualisations of diagrams and shapes.
rewalt.strdiags.draw(*diagrams, **params)	Given any number of diagrams, generates their string diagrams and draws them.
rewalt.strdiags.draw_boundaries(diagram[, dim])	Given a diagram, generates the string diagram of its input and output boundaries of a given dimension, and draws them.
rewalt.strdiags.to_gif(diagram, *diagrams, ...)	Given a non-zero number of diagrams, generates their string diagrams and outputs a GIF animation of the sequence of their visualisations.

5.9.1 strdiags.StrDiag

class rewalt.strdiags.**StrDiag**(*diagram*)

Bases: `object`

Class for string diagram visualisations of diagrams and shapes.

A string diagram depicts a top-dimensional "slice" of a diagram. The top-dimensional cells are represented as *nodes*, and the codimension-1 cells are represented as *wires*. The inputs of a top-dimensional cell are incoming wires of the associated node, and the outputs are outgoing wires.

The input->node->output order determines an acyclic flow between nodes and wires, which is represented in a string diagram by placing them at different "heights".

There are two other "flows" that we take into account:

- from codimension-2 inputs, to top-dimensional or codimension-1 cell, to codimension-2 outputs (only in dimension > 1);
- from codimension-3 inputs, to codimension-1 cells, to codimension-3 outputs (only in dimension > 2).

These are not in general acyclic; however, we obtain an acyclic flow by removing all directed loops. If there is a flow of the first kind between nodes and wires, we place them at different "widths".

If there is a flow of the second kind between wires, we place them at different "depths"; this is only seen when wires cross each other, in which case the one of lower depth is depicted as passing over the one of higher depth.

Internally, these data are encoded as a triple of NetworkX directed graphs, sharing the same vertices, partitioned into "node vertices" and "wire vertices". These graphs encode the "main (height) flow", the "width flow" and the "depth flow" between nodes and wires.

The class then contains a method *place_vertices()* that places the vertices on a [0, 1]x[0, 1] canvas, taking into account the height and width relations and resolving clashes.

Finally, it contains a method *draw()* that outputs a visualisation of the string diagram. The visualisation has customisable colours, orientation, and labels, and works with any `drawing.DrawBackend`; currently available are

- a Matplotlib backend, and
- a TikZ backend.

Parameters
diagram (`diagrams.Diagram | shapes.Shape | shapes.ShapeMap`) – A diagram or a shape or a shape map.

Notes

The "main flow" graph is essentially the *open graph* encoding of the string diagram in the sense of Dixon & Kissinger.

Methods

draw(**params)	Outputs a visualisation of the string diagram, using a backend.
place_vertices()	Places node and wire vertices on the unit square canvas, and returns their coordinates.

Attributes

depthgraph	Returns the "depth" flow graph between wire vertices.
graph	Returns the main flow graph between node and wire vertices.
nodes	Returns the nodes of the string diagram, together with all the stored associated information.
widthgraph	Returns the "width" flow graph between node and wire vertices.
wires	Returns the wires of the string diagram, together with all the stored associated information.

property graph

Returns the main flow graph between node and wire vertices.

Returns
graph – The main flow graph.

Return type
`networkx.DiGraph`

property widthgraph

Returns the "width" flow graph between node and wire vertices.

Returns
widthgraph – The width flow graph.

Return type
`networkx.DiGraph`

`property depthgraph`

Returns the "depth" flow graph between wire vertices.

Returns
depthgraph – The depth flow graph.

Return type
`networkx.DiGraph`

`property nodes`

Returns the nodes of the string diagram, together with all the stored associated information.

This is a dictionary whose keys are the elements of the diagram's shape corresponding to nodes. For each node, the object stores another dictionary, which contains

- the node's label (`label`),
- the node's fill colour (`color`) and stroke colour (`stroke`),
- booleans specifying whether to draw the node and/or its label (`draw_node`, `draw_label`), and
- a boolean specifying whether the node represents a degenerate cell (`isdegenerate`).

Returns
nodes – The nodes of the string diagram.

Return type
`dict[dict]`

`property wires`

Returns the wires of the string diagram, together with all the stored associated information.

This is a dictionary whose keys are the elements of the diagram's shape corresponding to wires. For each node, the object stores another dictionary, which contains

- the wire's label (`label`),
- the wire's colour (`color`),
- a boolean specifying whether to draw the wire's label (`draw_label`), and
- a boolean specifying whether the wire represents a degenerate cell (`isdegenerate`).

Returns
wires – The nodes of the string diagram.

Return type
`dict[dict]`

`place_vertices()`

Places node and wire vertices on the unit square canvas, and returns their coordinates.

The node and wire vertices are first placed on different heights and widths, proportional to the ratio between the longest path to the vertex and the longest path from the vertex in the main flow graph and the width flow graph.

In dimension > 2, this may result in clashes, where some vertices are given the same coordinates. In this case, these are resolved by "splitting" the clashing vertices, placing them at equally spaced angles of a circle centred on the clash coordinates, with an appropriately small radius that does not result in further clashes.

The coordinates are returned as a dictionary whose keys are the elements corresponding to nodes and wires.

Returns
coordinates – The coordinates assigned to wire and node vertices.

Return type
`dict[tuple[float]]`

draw(***params*)

Outputs a visualisation of the string diagram, using a backend.

Currently supported are a Matplotlib backend and a TikZ backend; in both cases it is possible to show the output (as a pop-up window for Matplotlib, or as code for TikZ) or save to file.

Various customisation options are available, including different orientations and colours.

Keyword Arguments

- **tikz** (`bool`) – Whether to output TikZ code (default is `False`).
- **show** (`bool`) – Whether to show the output (default is `True`).
- **path** (`str`) – Path where to save the output (default is `None`).
- **orientation** (`str`) – Orientation of the string diagram: one of `'bt'` (bottom-to-top), `'lr'` (left-to-right), `'tb'` (top-to-bottom), `'rl'` (right-to-left) (default is `'bt'`).
- **depth** (`bool`) – Whether to take into account the depth flow graph when drawing wires (default is `True`).
- **bgcolor** (*multiple types*) – The background colour (default is `'white'`).
- **fgcolor** (*multiple types*) – The foreground colour, given by default to nodes, wires, and labels (default is `'black'`).
- **infocolor** (*multiple types*) – The colour of additional information displayed in the diagram, such as positions (default is `'magenta'`).
- **wirecolor** (*multiple types*) – The default wire colour (default is same as *fgcolor*).
- **nodecolor** (*multiple types*) – The default node fill colour (default is same as *fgcolor*).
- **nodestroke** (*multiple types*) – The default node stroke colour (default is same as *nodecolor*).
- **degenalpha** (`float`) – The alpha factor of wires corresponding to degenerate cells (default is `0.1`).
- **labels** (`bool`) – Whether to display node and wire labels (default is `True`).
- **nodelabels** (`bool`) – Whether to display node labels (default is same as *labels*).
- **wirelabels** (`bool`) – Whether to display wire labels (default is same as *labels*).
- **labeloffset** (`tuple[float]`) – Point offset of labels relative to vertices (default is `(4, 4)`).
- **positions** (`bool`) – Whether to display node and wire positions (default is `False`).
- **nodepositions** (`bool`) – Whether to display node positions (default is same as *positions*).
- **wirepositions** (`bool`) – Whether to display wire positions (default is same as *positions*).
- **positionoffset** (`tuple[float]`) – Point offset of positions relative to vertices (default is `(4, -16)` for Matplotlib, `(4, -6)` for TikZ).
- **scale** (`float`) – (TikZ only) Scale factor to apply to output (default is `3`).

- **xscale** (float) – (TikZ only) Scale factor to apply to x axis in output (default is same as *scale*)
- **yscale** (float) – (TikZ only) Scale factor to apply to y axis in output (default is same as *scale*)

5.9.2 strdiags.draw

class rewalt.strdiags.**draw**(**diagrams*, ***params*)

Bases:

Given any number of diagrams, generates their string diagrams and draws them.

This is the same as generating the string diagram for each diagram, and calling `StrDiag.draw()` with the given parameters on each one of them.

Parameters

***diagrams** (diagrams.Diagram | shapes.Shape | shapes.ShapeMap) – Any number of diagrams or shapes or shape maps.

Keyword Arguments

****params** – Passed to `StrDiag.draw()`.

5.9.3 strdiags.draw_boundaries

class rewalt.strdiags.**draw_boundaries**(*diagram*, *dim=None*, ***params*)

Bases:

Given a diagram, generates the string diagram of its input and output boundaries of a given dimension, and draws them.

Parameters

- **diagram** (diagrams.Diagram | shapes.Shape | shapes.ShapeMap) – A diagram or a shape or a shape map.
- **dim** (int, optional) – Dimension of the boundary (default is `diagram.dim - 1`).

Keyword Arguments

***params** – Passed to `StrDiag.draw()`.

5.9.4 strdiags.to_gif

class rewalt.strdiags.**to_gif**(*diagram*, **diagrams*, ***params*)

Bases:

Given a non-zero number of diagrams, generates their string diagrams and outputs a GIF animation of the sequence of their visualisations.

Parameters

- **diagram** (diagrams.Diagram | shapes.Shape | shapes.ShapeMap) – A diagram or a shape or a shape map.
- ***diagrams** (diagrams.Diagram | shapes.Shape | shapes.ShapeMap) – Any number of diagrams or shapes or shape maps.

Keyword Arguments

- **timestep** (int) – The time step for the animation (default is 1000).
- **loop** (bool) – Whether to loop around the animation (default is False).
- ****params** – Passed to *StrDiag.draw()*.

5.10 hasse

Implements oriented Hasse diagram visualisation.

rewalt.hasse.Hasse(ogp)	Class for "oriented Hasse diagrams" of oriented graded posets.
rewalt.hasse.draw(*ogps, **params)	Given any number of oriented graded posets, or maps, or diagrams, generates their Hasse diagrams and draws them.

5.10.1 hasse.Hasse

class rewalt.hasse.**Hasse**(*ogp*)

Bases: object

Class for “oriented Hasse diagrams” of oriented graded posets.

The oriented Hasse diagram is stored as a NetworkX directed graph whose nodes are the elements of the oriented graded poset.

The orientation information is encoded by having edges corresponding to input faces point *from* the face, and edges corresponding to output faces point *towards* the face. To recover the underlying poset’s Hasse diagram, it suffices to reverse the edges that point from an element of higher dimension.

Objects of the class can also store labels for nodes of the Hasse diagram, for example the images of the corresponding elements through a map or a diagram.

The class also has a method *draw()* that outputs a visualisation of the Hasse diagram. This works with any drawing.DrawBackend; currently available are

- a Matplotlib backend, and
- a TikZ backend.

Parameters

ogp (ogposets.OgPoset | ogposets.OgMap | diagrams.Diagram) – The oriented graded poset, or a map of oriented graded posets, or a diagram.

Notes

If given a map of oriented graded posets (or shapes), produces the Hasse diagram of its source, with nodes labelled with the images of elements through the map.

If given a diagram, produces the Hasse diagram of its shape, with nodes labelled with the images of elements through the diagram.

Methods

draw(**params)	Outputs a visualisation of the Hasse diagram, using a backend.
place_nodes()	Places the nodes of the Hasse diagram on the unit square canvas, and returns their coordinates.

Attributes

diagram	Returns the oriented Hasse diagram as a NetworkX graph.
labels	Returns the labels of nodes of the Hasse diagram, in the same format as `ogposets.OgMap.mapping()`.
nodes	Returns the set of nodes of the Hasse diagram, that is, the graded set of elements of the oriented graded poset it encodes.

property nodes

Returns the set of nodes of the Hasse diagram, that is, the graded set of elements of the oriented graded poset it encodes.

Returns
nodes – The set of nodes of the Hasse diagram.

Return type
`ogposets.GrSet`

property diagram

Returns the oriented Hasse diagram as a NetworkX graph.

Returns
diagram – The oriented Hasse diagram.

Return type
`networkx.DiGraph`

property labels

Returns the labels of nodes of the Hasse diagram, in the same format as `ogposets.OgMap.mapping()`.

Returns
labels – The labels of the Hasse diagram.

Return type
`list[list]`

place_nodes()

Places the nodes of the Hasse diagram on the unit square canvas, and returns their coordinates.

The nodes are placed on different heights according to the dimension of the element their correspond to. Elements of the same dimension are then placed at different widths in order of position.

The coordinates are returned as a dictionary whose keys are the elements corresponding to nodes of the diagram.

Returns
coordinates – The coordinates assigned to nodes.

Return type
`dict[tuple[float]]`

draw(***params*)

Outputs a visualisation of the Hasse diagram, using a backend.

Currently supported are a Matplotlib backend and a TikZ backend; in both cases it is possible to show the output (as a pop-up window for Matplotlib, or as code for TikZ) or save to file.

Various customisation options are available, including different orientations and colours.

Keyword Arguments

- **tikz** (`bool`) – Whether to output TikZ code (default is `False`).
- **show** (`bool`) – Whether to show the output (default is `True`).
- **path** (`str`) – Path where to save the output (default is `None`).
- **orientation** (`str`) – Orientation of the Hasse diagram: one of `'bt'` (bottom-to-top), `'lr'` (left-to-right), `'tb'` (top-to-bottom), `'rl'` (right-to-left) (default is `'bt'`).
- **bgcolor** (*multiple types*) – The background colour (default is `'white'`).
- **fgcolor** (*multiple types*) – The foreground colour, given by default to nodes and labels (default is `'black'`).
- **labels** (`bool`) – Whether to display node labels (default is `True`).
- **inputcolor** (*multiple types*) – The colour of edges corresponding to input faces (default is `'magenta'`).
- **outputcolor** (*multiple types*) – The colour of edges corresponding to output faces (default is `'blue'`).
- **xscale** (`float`) – (TikZ only) Scale factor to apply to x axis in output (default is based on the dimension and maximal number of elements in one dimension).
- **yscale** (`float`) – (TikZ only) Scale factor to apply to y axis in output (default is based on the dimension and maximal number of elements in one dimension).

5.10.2 hasse.draw

class rewalt.hasse.**draw**(**ogps*, ***params*)

Bases:

Given any number of oriented graded posets, or maps, or diagrams, generates their Hasse diagrams and draws them.

This is the same as generating the Hasse diagram for each argument, and calling *Hasse.draw()* with the given parameters on each one of them.

Parameters
***ogps** (`ogposets.OgPoset | ogposets.OgMap | diagrams.Diagram`) – Any number of oriented graded posets or maps or diagrams.

Keyword Arguments
****params** – Passed to *Hasse.draw()*.

5.11 drawing

Drawing backends.

rewalt.drawing.DrawBackend(**params)	Abstract drawing backend for placing nodes, wires, arrows, and labels on a canvas.
rewalt.drawing.MatBackend(**params)	Drawing backend outputting Matplotlib figures.
rewalt.drawing.TikZBackend(**params)	Drawing backend outputting TikZ code that can be embedded in a LaTeX document.

5.11.1 drawing.DrawBackend

class rewalt.drawing.**DrawBackend**(*****params***)

Bases: `ABC`

Abstract drawing backend for placing nodes, wires, arrows, and labels on a canvas.

The purpose of this class is simply to describe the signature of methods that subclasses have to implement.

Keyword Arguments

- **bgcolor** (*multiple types*) – The background colour (default is `'white'`).
- **fgcolor** (*multiple types*) – The foreground colour (default is `'black'`).
- **orientation** (`str`) – Orientation: one of `'bt'` (bottom-to-top), `'lr'` (left-to-right), `'tb'` (top-to-bottom), `'rl'` (right-to-left) (default is `'bt'`).

Notes

All coordinates should be passed to the backend *as if* the orientation was bottom-to-top; the backend will then make rotations and adjustments according to the chosen orientation.

Methods

draw_arrow(xy0, xy1, **params)	Draws an arrow on the canvas.
draw_label(label, xy, offset, **params)	Draws a label next to a location on the canvas.
draw_node(xy, **params)	Draws a node on the canvas.
draw_wire(wire_xy, node_xy, **params)	Draws a wire from a wire vertex to a node vertex on the canvas.
output(**params)	Output the picture.
rotate(xy)	Returns coordinates rotated according to the orientation of the picture.

draw_wire(*wire_xy*, *node_xy*, *****params***)

Draws a wire from a wire vertex to a node vertex on the canvas.

Parameters

- **wire_xy** (`tuple[float]`) – The coordinates of the wire vertex.
- **node_xy** (`tuple[float]`) – The coordinates of the node vertex.

Keyword Arguments

- **color** (*multiple types*) – The colour of the wire (default is `self.fgcolor`).
- **alpha** (`float`) – Alpha factor of the wire (default is 1).
- **depth** (`bool`) – Whether to draw the wire with a contour, to simulate "crossing over" objects that are already on the canvas (default is `True`).

draw_label(*label*, *xy*, *offset*, ***params*)

Draws a label next to a location on the canvas.

Parameters

- **label** (`str`) – The label.
- **xy** (`tuple[float]`) – The coordinates of the object to be labelled.
- **offset** (`tuple[float]`) – Point offset of the label relative to the object.

Keyword Arguments

color (*multiple types*) – The colour of the label (default is `self.fgcolor`).

draw_node(*xy*, ***params*)

Draws a node on the canvas.

Parameters

xy (`tuple[float]`) – The coordinates of the node.

Keyword Arguments

- **color** (*multiple types*) – Fill colour of the node (default is `self.fgcolor`).
- **stroke** (*multiple types*) – Stroke colour of the node (default is same as *color*).

draw_arrow(*xy0*, *xy1*, ***params*)

Draws an arrow on the canvas.

Parameters

- **xy0** (`tuple[float]`) – The coordinates of the starting point.
- **xy1** (`tuple[float]`) – The coordinates of the ending point.

Keyword Arguments

- **color** (*multiple types*) – Colour of the arrow (default is `self.fgcolor`).
- **shorten** (`float`) – Factor by which to scale the length (default is 1).

output(***params*)

Output the picture.

Keyword Arguments

- **show** (`bool`) – Whether to show the output (default is `True`).
- **path** (`str`) – Path where to save the output (default is `None`).
- **scale** (`float`) – (TikZ only) Scale factor to apply to output (default is 3).
- **xscale** (`float`) – (TikZ only) Scale factor to apply to x axis in output (default is same as *scale*)
- **yscale** (`float`) – (TikZ only) Scale factor to apply to y axis in output (default is same as *scale*)

rotate(*xy*)

Returns coordinates rotated according to the orientation of the picture.

Parameters
xy (`tuple[float]`) – The coordinates to rotate.

Returns
rotate – The rotated coordinates.

Return type
`tuple[float]`

5.11.2 drawing.MatBackend

class `rewalt.drawing.`**MatBackend**(***params*)

Bases: *DrawBackend*

Drawing backend outputting Matplotlib figures.

Methods

draw_arrow(xy0, xy1, **params)	Draws an arrow on the canvas.
draw_label(label, xy, offset, **params)	Draws a label next to a location on the canvas.
draw_node(xy, **params)	Draws a node on the canvas.
draw_wire(wire_xy, node_xy, **params)	Draws a wire from a wire vertex to a node vertex on the canvas.
output(**params)	Output the picture.

draw_wire(*wire_xy*, *node_xy*, ***params*)

Draws a wire from a wire vertex to a node vertex on the canvas.

Parameters

- **wire_xy** (`tuple[float]`) – The coordinates of the wire vertex.
- **node_xy** (`tuple[float]`) – The coordinates of the node vertex.

Keyword Arguments

- **color** (*multiple types*) – The colour of the wire (default is `self.fgcolor`).
- **alpha** (`float`) – Alpha factor of the wire (default is 1).
- **depth** (`bool`) – Whether to draw the wire with a contour, to simulate "crossing over" objects that are already on the canvas (default is `True`).

draw_label(*label*, *xy*, *offset*, ***params*)

Draws a label next to a location on the canvas.

Parameters

- **label** (`str`) – The label.
- **xy** (`tuple[float]`) – The coordinates of the object to be labelled.
- **offset** (`tuple[float]`) – Point offset of the label relative to the object.

Keyword Arguments
color (*multiple types*) – The colour of the label (default is `self.fgcolor`).

draw_node(*xy*, ***params*)

Draws a node on the canvas.

Parameters

xy (`tuple[float]`) – The coordinates of the node.

Keyword Arguments

- **color** (*multiple types*) – Fill colour of the node (default is `self.fgcolor`).
- **stroke** (*multiple types*) – Stroke colour of the node (default is same as *color*).

draw_arrow(*xy0*, *xy1*, ***params*)

Draws an arrow on the canvas.

Parameters

- **xy0** (`tuple[float]`) – The coordinates of the starting point.
- **xy1** (`tuple[float]`) – The coordinates of the ending point.

Keyword Arguments

- **color** (*multiple types*) – Colour of the arrow (default is `self.fgcolor`).
- **shorten** (`float`) – Factor by which to scale the length (default is 1).

output(***params*)

Output the picture.

Keyword Arguments

- **show** (`bool`) – Whether to show the output (default is `True`).
- **path** (`str`) – Path where to save the output (default is `None`).
- **scale** (`float`) – (TikZ only) Scale factor to apply to output (default is 3).
- **xscale** (`float`) – (TikZ only) Scale factor to apply to x axis in output (default is same as *scale*)
- **yscale** (`float`) – (TikZ only) Scale factor to apply to y axis in output (default is same as *scale*)

5.11.3 drawing.TikZBackend

class rewalt.drawing.**TikZBackend**(***params*)

Bases: *DrawBackend*

Drawing backend outputting TikZ code that can be embedded in a LaTeX document.

Methods

draw_arrow(xy0, xy1, **params)	Draws an arrow on the canvas.
draw_label(label, xy, offset, **params)	Draws a label next to a location on the canvas.
draw_node(xy, **params)	Draws a node on the canvas.
draw_wire(wire_xy, node_xy, **params)	Draws a wire from a wire vertex to a node vertex on the canvas.
output(**params)	Output the picture.

draw_wire(*wire_xy*, *node_xy*, ***params*)

Draws a wire from a wire vertex to a node vertex on the canvas.

Parameters

- **wire_xy** (tuple[float]) – The coordinates of the wire vertex.
- **node_xy** (tuple[float]) – The coordinates of the node vertex.

Keyword Arguments

- **color** (*multiple types*) – The colour of the wire (default is `self.fgcolor`).
- **alpha** (float) – Alpha factor of the wire (default is 1).
- **depth** (bool) – Whether to draw the wire with a contour, to simulate "crossing over" objects that are already on the canvas (default is `True`).

draw_label(*label*, *xy*, *offset*, ***params*)

Draws a label next to a location on the canvas.

Parameters

- **label** (str) – The label.
- **xy** (tuple[float]) – The coordinates of the object to be labelled.
- **offset** (tuple[float]) – Point offset of the label relative to the object.

Keyword Arguments

color (*multiple types*) – The colour of the label (default is `self.fgcolor`).

draw_node(*xy*, ***params*)

Draws a node on the canvas.

Parameters

xy (tuple[float]) – The coordinates of the node.

Keyword Arguments

- **color** (*multiple types*) – Fill colour of the node (default is `self.fgcolor`).
- **stroke** (*multiple types*) – Stroke colour of the node (default is same as *color*).

draw_arrow(*xy0*, *xy1*, ***params*)

Draws an arrow on the canvas.

Parameters

- **xy0** (tuple[float]) – The coordinates of the starting point.
- **xy1** (tuple[float]) – The coordinates of the ending point.

Keyword Arguments

- **color** (*multiple types*) – Colour of the arrow (default is `self.fgcolor`).
- **shorten** (float) – Factor by which to scale the length (default is 1).

output(***params*)

Output the picture.

Keyword Arguments

- **show** (bool) – Whether to show the output (default is `True`).
- **path** (str) – Path where to save the output (default is `None`).

- **scale** (float) – (TikZ only) Scale factor to apply to output (default is 3).
- **xscale** (float) – (TikZ only) Scale factor to apply to x axis in output (default is same as *scale*)
- **yscale** (float) – (TikZ only) Scale factor to apply to y axis in output (default is same as *scale*)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

Y