0.1 Drawing & reading

The previous section defined diagrams as a data structure based on lists of layers, in this section we define *pictures of diagrams*. Concretely, such a picture will be encoded in a computer memory as a bitmap, i.e. a matrix of colour values. Abstractly, we will define these pictures in terms of topological subsets of the Cartesian plane. We first recall the topological definition from Joyal's and Street's unpublished manuscript *Planar diagrams and tensor algebra* [JS88] and then discuss the isomorphism between the two definitions. In one direction, the isomorphism sends a `Diagram` object to its drawing. In the other direction, it reads the picture of a diagram and translates it into a `Diagram` object, i.e. its domain, codomain and list of layers.

0.1.1 Labeled generic progressive plane graphs

A *topological graph*, also called 1-dimensional cell complex, is a tuple (G, G_0, G_1) of a Hausdorff space G and a pair of a closed subset $G_0 \subseteq G$ and a set of open subsets $G_1 \subseteq P(G)$ called *nodes* and *wires* respectively, such that:

- G_0 is discrete and $G - G_0 = \bigcup G_1$,
- each wire $e \in G_1$ is homeomorphic to an open interval and its boundary is contained in the nodes $\partial e \subseteq G_0$.

From a topological graph G, one can construct an undirected graph in the usual sense by forgetting the space G, taking G_0 as nodes and edges $(x, y) \in G_0 \times G_0$ for each $e \in G_1$ with $\partial e = \{x, y\}$. A topological graph is finite (planar) if its undirected graph is finite (planar, i.e. there is some embedding in the plane).

A *plane graph* between two real numbers $a < b$ is a finite, planar topological graph G with an embedding in $\mathbb{R} \times [a, b]$. We define the domain $\texttt{dom}(G) = G_0 \cap \mathbb{R} \times \{a\}$, the codomain $\texttt{cod}(G) = G_0 \cap \mathbb{R} \times \{b\}$ as lists of nodes ordered by horizontal coordinates and the set $\texttt{boxes}(G) = G_0 \cap \mathbb{R} \times (a, b)$. We require that:

- $G \cap \mathbb{R} \times \{a\} = \texttt{dom}(G)$ and $G \cap \mathbb{R} \times \{b\} = \texttt{cod}(G)$, i.e. the graph touches the horizontal boundaries only at domain and codomain nodes,
- every domain and codomain node $x \in G \cap \mathbb{R} \times \{a, b\}$ is in the boundary of exactly one wire $e \in G_1$, i.e. wires can only meet at box nodes.

A plane graph is *generic* when the projection on the vertical axis $p_1 : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is injective on $G_0 - \mathbb{R} \times \{a, b\}$, i.e. no two box nodes are at the same height. From a generic plane graph, we can get a list $\texttt{boxes}(G) \in G_0^\star$ ordered by height. A plane graph is *progressive* (also called *recumbent* by Joyal and Street) when p_1 is injective on each wire $e \in G_1$, i.e. wires go from top to bottom and do not bend backwards.

From a progressive plane graph G, one can construct a directed graph by forgetting the space G, taking G_0 as nodes and edges $(x, y) \in G_0 \times G_0$ for each $e \in G_1$ with $\partial e = \{x, y\}$ and $p_1(x) < p_1(y)$. We can also define the domain and the codomain of each box node $\texttt{dom}, \texttt{cod} : \texttt{boxes}(G) \to G_1^\star$ with $\texttt{dom}(x) = \{e \in G_1 \mid \partial e = \{x, y\}, p_1(x) < p_1(y)\}$ the wires coming in from the top and $\texttt{cod}(x) = \{e \in G_1 \mid \partial e = \{x, y\}, p_1(x) > p_1(y)\}$ the wires going out to the bottom, these sets are linearly ordered as follows. Take some $\epsilon > 0$ such that the horizontal line at height $p_1(x) - \epsilon$ crosses each of the wires in the domain. Then list $\texttt{dom}(x) \in G_1^\star$ in order of horizontal coordinates of their intersection points, i.e. $e < e'$ if $p_0(y) < p_0(y')$ for the projection $p_0 : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and $y^{(\prime)} = e^{(\prime)} \cap \{p_1(x) - \epsilon\} \times \mathbb{R}$. Symmetrically we define the list of codomain nodes $\texttt{cod}(x) \in G_1^\star$ with a horizontal line at $p_1 + \epsilon$.

A *labeling* of progressive plane graph G by a monoidal signature Σ is a pair of functions from wires to objects $\lambda : G_1 \to \Sigma_0$ and from boxes to boxes $\lambda : \texttt{boxes}(G) \to \Sigma_1$ which commutes with the domain and codomain. From an lgpp (*labeled generic progressive plane*) graph, one can construct a `Diagram`.

Listing 0.1.1. Translation from labeled generic progressive plane graphs to `Diagram`.

```
def read( $G$, $\lambda : G_1 \to$ Ty, $\lambda : \texttt{boxes}(G) \to$ Box ) -> Diagram:
    dom = [ $\lambda(e)$ for $x \in \texttt{dom}(G)$ for $e \in G_1$ if $x \in \partial e$ ]
    boxes = [ $\lambda(x)$ for $x \in \texttt{boxes}(G)$ ]
    offsets = [len( $G_1 \cap \{p_0(x)\} \times \mathbb{R}$ ) for $x \in \texttt{boxes}(G)$ ]
    return decode(dom, zip(boxes, offsets))
```

0.1.2 From diagrams to graphs...

In the other direction, there are many possible ways to draw a given `Diagram` as a lgpp graph, i.e. to embed its graph into the plane. Vicary and Delpeuch [DV18] give a linear-time algorithm to compute such an embedding with the following disadvantage: the drawing of a tensor $f \otimes g$ does not necessarily look like the horizontal juxtaposition of the drawings for f and g. For example, if we tensor an identity with a scalar, the wire representing the identity will wiggle around the node representing the scalar. DisCoPy uses a quadratic-time drawing algorithm with the following design decision: we make every wire a straight line and as vertical as possible. We first initialise the lgpp graph of the identity with a constant spacing between each wire, then for each layer we update the embedding so that there is enough space for the output wires of the box before we add it to the graph.

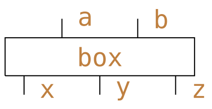
Listing 0.1.2. Outline of `Diagram.draw` from `Diagram` to `PlaneGraph`.

```
Embedding = dict[Node, tuple[float, float]]
PlaneGraph = tuple[Graph, Embedding]

def draw(self: Diagram) -> PlaneGraph:
    graph = diagram2graph(self)
    def make_space(scan: list[Node], box: Box, offset: int) -> float:
        """ Update the graph to make space and return the left of the box. """
    box_nodes = [Node('box', box, -1, j) for j, box in enumerate(self.boxes)]
    dom_nodes = scan = [Node('dom', x, i, -1) for i, x in enumerate(self.dom)]
    position = {node: (i, -1) for i, node in enumerate(dom_nodes)}
    for j, (left, box, _) in enumerate(self.layers):
        box_node, left_of_box = Node('box', box, -1, j), make_space(scan, box, offset)
        position[box_node] = (left_of_box + max(len(box.dom), len(box.cod)) / 2, j)
        for kind, epsilon in (('dom', -.1), ('cod', .1)):
            for i, x in enumerate(getattr(box, kind)):
                position[Node(kind, x, i, j)] = (left_of_box + i, j + epsilon)
        box_cod_nodes = [Node('cod', x, i, j) for i, x in enumerate(box.cod)]
        scan = scan[:len(left)] + box_cod_nodes + scan[len(left @ box.dom):]
    for i, x in enumerate(self.cod):
        position[Node('cod', x, i, len(self))] = (position[scan[i]][0], len(self))
    return graph, position

Diagram.draw = draw
```

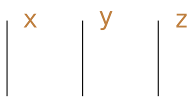
Note that when we draw the plane graph for a diagram, we do not usually draw the box nodes as points. Instead, we draw them as boxes, i.e. a box node $x \in \texttt{boxes}(G)$ is depicted as the rectangle with corners $(l, p_1(x) \pm \epsilon)$ and $(r, p_1(x) \pm \epsilon)$ for $l, r \in \mathbb{R}$ the left- and right-most coordinate of its domain and codomain nodes. In this way, we do not need to draw the in- and out-going wires of the box node: they are hidden by the rectangle. Exceptions include *spider boxes* where we draw the box node (the head) and its outgoing wires (the legs of the spider) as well as *swap, cup and cap boxes* where we do not draw the box node at all, only its outgoing wires which are drawn as Bézier curves to look like swaps, cups and caps respectively. These special boxes will be discussed, and drawn, in section 0.2.

Example 0.1.3. Drawing of a box, an identity, a layer, a composition and a tensor.

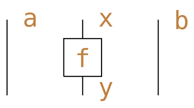
```
a, b, c, x, y, z, w = map(Ty, "abcxyzw")
Box('box', a @ b, x @ y @ z).draw()
```



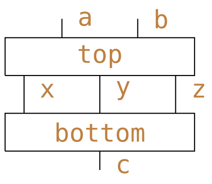
```
Diagram.id(x @ y @ z).draw()
```



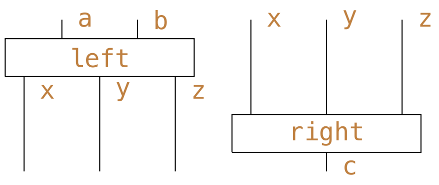
```
layer = a @ Box('f', x, y) @ b
layer.draw()
```



```
top, bottom = Box('top', a @ b, x @ y @ z), Box('bottom', x @ y @ z, c)
(top >> bottom).draw()
```

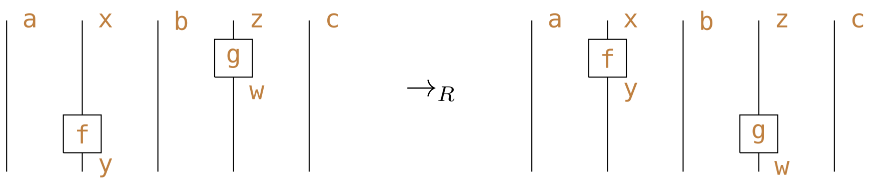


```
left, right = Box('left', a @ b, x @ y @ z), Box('right', x @ y @ z, c)
(left @ right).draw()
```



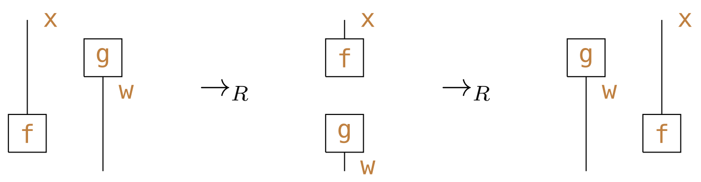
Example 0.1.4. Drawing of the interchanger in the general case.

```
f, g = Box('f', x, y), Box('g', z, w)
(a @ f @ b @ g @ c).interchange(0).draw(); (a @ f @ b @ g @ c).draw()
```



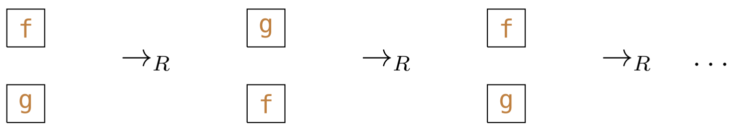
Example 0.1.5. Drawing of the interchangers for an effect then a state.

```
f, g = Box('f', x, Ty()), Box('g', Ty(), w)
(f >> g).interchange(0).draw()
(f >> g).draw()
(f >> g).interchange(0, left=True).draw()
```



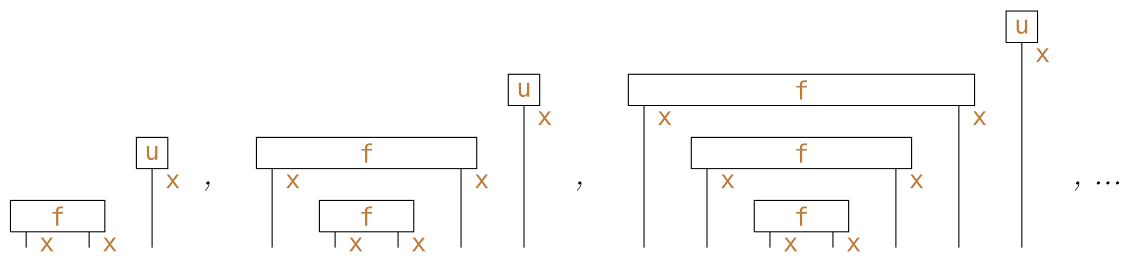
Example 0.1.6. Drawing of the Eckmann-Hilton argument.

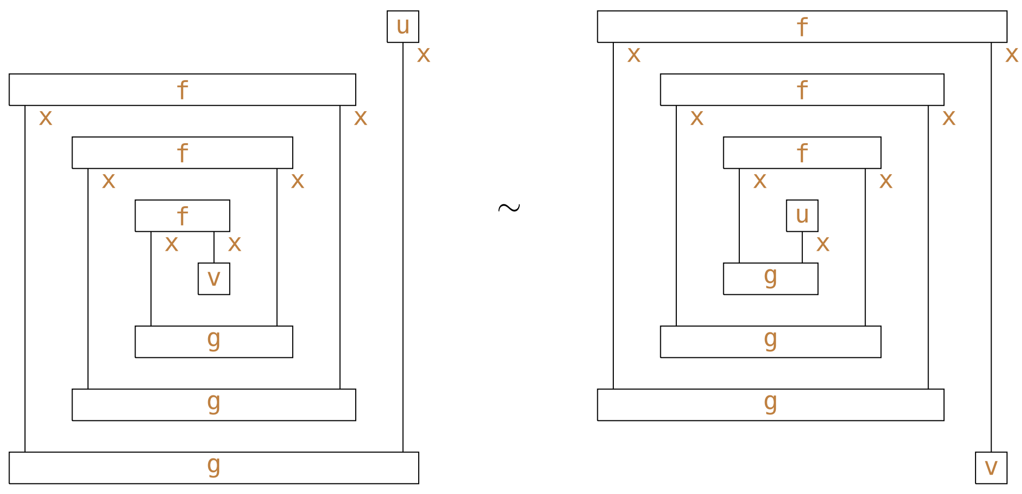
```
f, g = Box('f', Ty(), Ty()), Box('g', Ty(), Ty())
(f @ g).draw()
(f @ g).interchange(0).draw()
(f @ g).interchange(0).interchange(0).draw()
```



Example 0.1.7. *The following spiral diagram is the cubic worst-case for interchanger normal form. It is also the quadratic worst-case for drawing, at each layer of the first half we need to update the position of every preceding layer in order to make space for the output wires.*

```
x = Ty('x')
f, g = Box('f', Ty(), x @ x), Box('g', x @ x, Ty())
u, v = Box('u', Ty(), x), Box('v', x, Ty())

def spiral(length: int) -> Diagram:
    diagram, n = u, length // 2 - 1
    for i in range(n): diagram >>= Id(x ** i) @ f @ Id(x ** (i + 1))
    diagram >>= Id(x ** n) @ v @ Id(x ** n)
    for i in range(n): diagram >>= Id(x ** (n - i - 1)) @ g @ Id(x ** (n - i - 1))
    return diagram

diagram = spiral(8)
for i in [1, 2, 3]: diagram[:i + 1].draw()
diagram.draw(); diagram.normal_form().draw()
```





Next, we define the inverse translation `graph2diagram`.

Listing 0.1.8. Translation from `PlaneGraph` to `Diagram`.

```
def graph2diagram(graph: Graph, position: Embedding) -> Diagram:
    dom = Ty(*[node.label for node in graph.nodes if node.kind == 'dom' and node.j == -1])
    boxes = [node.label for node in graph.nodes if node.kind == 'box']
    scan, offsets = [Node('dom', x, i, -1) for i, x in enumerate(dom)], []
    for j, box in enumerate(boxes):
        left_of_box = position[Node('dom', box.dom[0], 0, j)][0]\
            if box.dom else position[Node('box', box, -1, j)][0]
        offset = len([node for node in scan if position[node][0] < left_of_box])
        box_cod_nodes = [Node('cod', x, i, j) for i, x in enumerate(box.cod)]
        scan = scan[:offset] + box_cod_nodes + scan[offset + len(box.dom):]
        offsets.append(offset)
    return decode(dom, zip(boxes, offsets))
```

Theorem 0.1.9. `graph2diagram(self.draw()) == self` *for all* `self: Diagram`.

Proof. By induction on `n = len(self.layers)`. If `not len(self.layers)` we get that `dom == self.dom` and `boxes == offsets == []`. If the theorem holds for `self`, it holds for `self >> Layer(left, box, right)`. Indeed, we have:

- `dom == self.dom and boxes == self.boxes + [box]`

- `(x, Node('cod', self.cod[i], i, n)) in graph for i, x in enumerate(scan)`

Moreover, the horizontal coordinates of the nodes in `scan` are strictly increasing, thus we get the desired `offsets == self.offsets + [len(left)]`. □

0.1.3 ...and back!

From a labeled generic progressive plane graph, we get a unique diagram *up to deformation*. A deformation $h : G \to G'$ between two labeled plane graphs G, G' is a continuous map $h : G \times [0,1] \to \mathbb{R} \times \mathbb{R}$ such that:

- $h(G,t)$ is a plane graph for all $t \in [0,1]$, $h(G,0) = G$ and $h(G,1) = G'$,
- $x \in \texttt{boxes}(G)$ implies $h(x,t) \in \texttt{boxes}(h(G,t))$ for all $t \in [0,1]$,
- $h(G,t) \fatsemi \lambda = \lambda$ for all $t \in [0,1]$, i.e. the labels are preserved throughout.

A deformation is progressive (generic) when $h(G,t)$ is progressive (generic) for all $t \in [0,1]$. We write $G \sim G'$ when there exists some deformation $h : G \to G'$, this defines an equivalence relation.

Theorem 0.1.10. *For all lgpp graphs G,* `Diagram.draw(graph2diagram(` G `))` $\sim G$ *up to generic progressive deformation.*

Proof. By induction on the length of $\texttt{boxes}(G)$. If there are no boxes, G is the graph of the identity and we can deform it so that each wire is vertical with constant spacing. If there is one box, G is the graph of a layer and we can cut it in three vertical slices with the box node and its outgoing wires in the middle. We can apply the case of the identity to the left and right slices, for the middle slice we make the wires straight with a constant spacing between the domain and codomain. Because G is generic, we can cut a graph with $n > 2$ boxes in two horizontal slices between the last and the one-before-last box, then apply the case for layers and the induction hypothesis. To glue the two slices back together while keeping the wires straight, we need to make space for the wires going out of the box.

This deformation is indeed progressive, i.e. we never bend wires we only make them straight. It is also generic, i.e. we never move a box node past another. □

Theorem 0.1.11. *There is a progressive deformation $h : G \to G'$ between two lgpp graphs iff* `graph2diagram(` G `) == graph2diagram(` G' `)` *up to interchanger.*

Proof. By induction on the number n of *coincidences*, the times at which the deformation h fails to be generic, i.e. two or more boxes are at the same height. WLOG (i.e. up to continuous deformation of deformations) this happens at a discrete number of time steps $t_1, \dots, t_n \in [0, 1]$. Again WLOG at each time step there is at most two boxes at the same height, e.g. if there are two boxes moving below a third at the same time, we deform the deformation so that they move one after the other. The list of boxes and offsets is preserved under generic deformation, thus if $n = 0$ then `graph2diagram(` G `) == graph2diagram(` G' `)` on the nose. If $n = 1$, take `i: int` the index of the box for which the coincidence happens and `left: bool` whether it is a left or right interchanger, then `graph2diagram(` G `).interchange(i, left) == graph2diagram(` G' `)`. Given a deformation with $n+1$ coincidences, we can cut it in two time slices with 1 and n coincidences respectively then apply the cases for $n = 1$ and the induction hypothesis.

For the converse, a proof of `graph2diagram(` G `) == graph2diagram(` G' `)`, i.e. a sequence of n interchangers, translates into a deformation with n coincidences. DisCoPy can output these proofs as videos using `Diagram.normalize` to iterate through the rewriting steps and `Diagram.to_gif` to produce a `.gif` file. □

0.1.4 A natural isomorphism

We have established an isomorphism between the class of lgpp graphs (up to progressive deformation) and the class of `Diagram` objects (up to interchanger). It remains to show that this actually forms an isomorphism of monoidal categories. That is for every monoidal signature Σ, there is a monoidal category $G(\Sigma)$ with objects $\Sigma_0^\star$ and arrows the equivalence classes of lgpp graphs with labels in Σ. The domain and codomain of an arrow is given by the labels of the domain and codomain of the graph. The identity $\texttt{id}(x_1 \dots x_n)$ is the graph with wires $(i, a) \to (i, b)$ for $i \leq n$ and $a, b \in \mathbb{R}$ the horizontal boundaries. The tensor of two graphs G and G' is given by horizontal juxtaposition, i.e. take $w = \max(p_0(G)) + 1$ the right-most point of G plus a margin and set $G \otimes G' = G \cup \{(p_0(x) + w, p_1(x)) \mid x \in G'\}$. The composition $G \fatsemi G'$ is given by vertical juxtaposition and connecting the codomain nodes of G to the domain nodes of G'. That is, $G \fatsemi G' = s^+(G) \cup s^-(G') \cup E$ for $s^\pm(x) = \left(p_0(x), \frac{p_1(x) \pm (b-a)}{2}\right)$ and wires $s^+(\texttt{cod}(G)_i) \to s^-(\texttt{dom}(G')_i) \in E$ for each $i \leq \texttt{len}(\texttt{cod}(G)) = \texttt{len}(\texttt{dom}(G'))$.

The deformations for the unitality axioms are straightforward: there is a deformation $G \fatsemi \texttt{id}(\texttt{cod}(G)) \sim G \sim \texttt{id}(\texttt{dom}(G)) \fatsemi G$ which contracts the wires of the identity graph, the unit of the tensor is the empty diagram so we have an equality

$G \otimes \texttt{id}(1) = G = \texttt{id}(1) \otimes G$. The deformations for the associativity axioms are better described by the hand-drawn diagrams of Joyal and Street in figure 1.

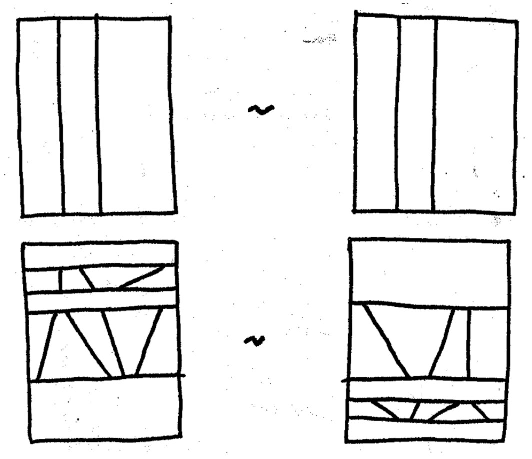


Figure 1: Deformations for the associativity of tensor and composition.

The interchange law holds on the nose, i.e. $(G \otimes G') ⨟ (H \otimes H') = (G ⨟ H) \otimes (G' ⨟ H')$, as witnessed by figure 2, the hand-drawn diagram which is the result of both sides.

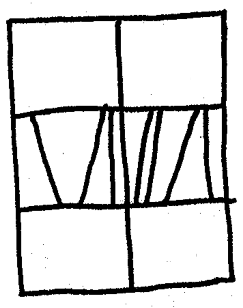


Figure 2: The graph of the interchange law.

Thus, we have defined a monoidal category $G(\Sigma)$. Given a morphism of monoidal signatures $f : \Sigma \to \Gamma$, there is a functor $G(f) : G(\Sigma) \to G(\Gamma)$ which sends a graph to itself relabeled with $f ⨟ \lambda$, its image on arrows is given in listing 0.1.12. Hence, we have defined a functor $G : \mathbf{Monsig} \to \mathbf{MonCat}$ which we claim is naturally isomorphic to the free functor $F : \mathbf{Monsig} \to \mathbf{MonCat}$ defined in the previous section.

Listing 0.1.12. Implementation of the functor $G : \mathbf{Monsig} \to \mathbf{MonCat}$.

```
SigMorph = tuple[dict[Ob, Ob], dict[Box, Box]]

def G(f: SigMorph) -> Callable[[Graph], Graph]:
    def G_of_f(graph: Graph) -> Graph:
        relabel = lambda node: Node('box', f[1][node.label], node.i, node.j)\
            if node.kind == 'box'\
            else Node(node.kind, f[0][node.label], node.i, node.j)
        return Graph(map(relabel, graph.edges))
    return G_of_f
```

Theorem 0.1.13. *There is a natural isomorphism $F \simeq G$.*

Proof. From theorems 0.1.10 and 0.1.11, we have an isomorphism between `Diagram` and `PlaneGraph` given by `d2g = Diagram.draw` and `g2d = graph2diagram`, up to deformation and interchanger respectively. Now define the image of F on arrows `F = lambda f: Functor(ob=f[0], ar=f[1])`. Given a morphism of monoidal signatures `f: SigMorph` we have the following two naturality squares.

$$\begin{tikzcd}
\texttt{Diagram} \arrow[r, "\texttt{d2g}"] \arrow[d, "\texttt{F(f)}"'] & \texttt{PlaneGraph} \arrow[d, "\texttt{G(f)}"] \\
\texttt{Diagram} \arrow[r, "\texttt{d2g}"] & \texttt{PlaneGraph}
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
\texttt{PlaneGraph} \arrow[r, "\texttt{g2d}"] \arrow[d, "\texttt{G(f)}"'] & \texttt{Diagram} \arrow[d, "\texttt{F(f)}"] \\
\texttt{PlaneGraph} \arrow[r, "\texttt{g2d}"] & \texttt{Diagram}
\end{tikzcd}$$

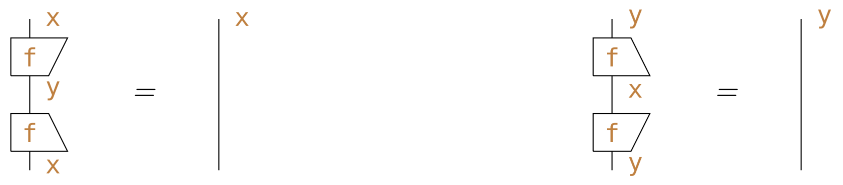
□

0.1.5 Daggers, sums and bubbles

As in the previous sections, we now discuss the drawing of daggers, sums and bubbles. When we draw a diagram in the free †-monoidal category, we add some asymmetry to the drawing of each box so that it looks like the vertical reflection of its dagger.

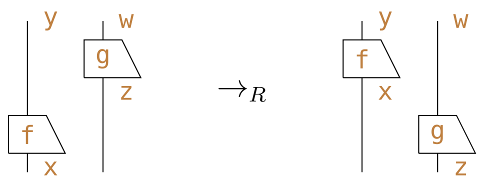
Example 0.1.14. Drawing of the axiom for unitaries.

```
f, g = Box('f', x, y), Box('g', z, w)
(f >> f[::-1]).draw(); Diagram.id(x).draw(); (f[::-1] >> f).draw(); Diagram.id(y).draw()
```



Example 0.1.15. Drawing of the axiom for †-monoidal categories.

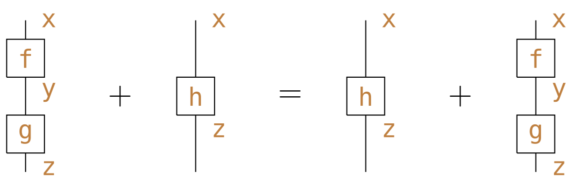
```
f, g = Box('f', x, y), Box('g', z, w)
(f @ g)[::-1].draw(); (f[::-1] @ g[::-1]).draw()
assert (f @ g)[::-1].normal_form() == f[::-1] @ g[::-1]
```



When we draw a sum, we just draw each term with an addition symbol in between. More generally, `drawing.equation` allows to draw any list of diagrams and `drawing.Equation` allows to draw equations within equations.

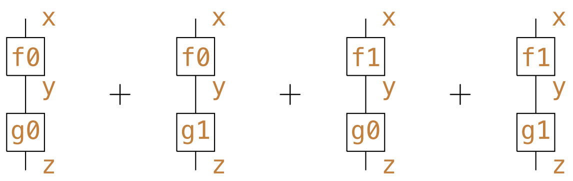
Example 0.1.16. Drawing of a commutativity equation.

```
from discopy import drawing

f, g, h = Box('f', x, y), Box('g', y, z), Box('h', x, z)
drawing.equation(drawing.Equation(f >> g, h, symbol='$+$'),
                 drawing.Equation(h, f >> g, symbol='$+$'))
```

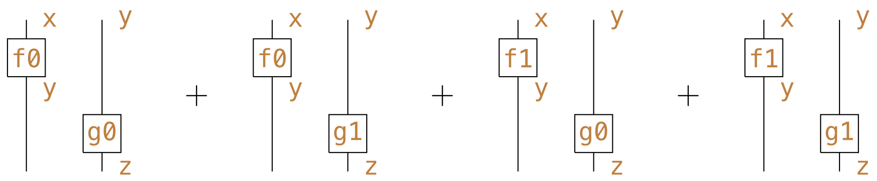


Example 0.1.17. Drawing a composition and tensor of sums.

```
f0, g0 = Box("f0", x, y), Box("g0", y, z)
f1, g1 = Box("f1", x, y), Box("g1", y, z)
((f0 + f1) >> (g0 + g1)).draw()
```



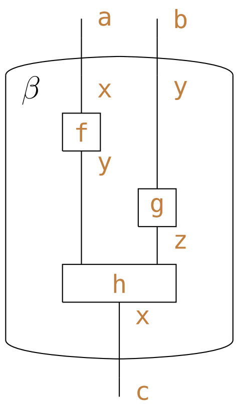
```
((f0 + f1) @ (g0 + g1)).draw()
```

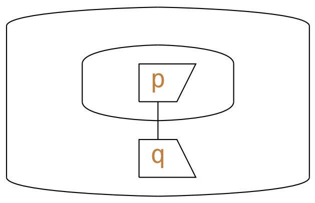


The case of drawing bubbles is more interesting. One solution would be to draw the bubble as a rectangle like any other box, then draw the content of the bubble inside the rectangle. However, this would require some clever scaling so that the boxes of the diagram inside the bubble have the same size as the boxes outside, i.e. we would need to add more complexity to our drawing algorithm. The solution implemented in DisCoPy is to apply a faithful functor `downgrade` : $F(\Sigma^\beta) \to F(\Sigma \cup \texttt{open}^\beta \cup \texttt{close}^\beta)$ from the free monoidal category with bubbles $F(\Sigma^\beta)$ to the free monoidal category generated by the following signature. Take the objects $\texttt{open}_0^\beta = \texttt{close}_0^\beta = \Sigma_0 + \{\bullet\}$ and boxes for opening $\texttt{open}^\beta(x) : \beta_{\texttt{dom}}(x) \to \bullet \otimes x \otimes \bullet$ and closing $\texttt{close}^\beta(x) : \bullet \otimes x \otimes \bullet \to \beta_{\texttt{cod}}(x)$ bubbles for each type $x \in \Sigma_0^\star$. Now define $\texttt{downgrade(f.bubble())} = \texttt{open}^\beta(\texttt{f.dom}) \fatsemi (\bullet \otimes \texttt{f} \otimes \bullet) \fatsemi \texttt{close}^\beta(\texttt{f.cod})$ for any diagram `f` inside a bubble. That is, we draw a bubble as its opening, its inside with identity wires on both sides then its closing. The $\bullet$-labeled wires are drawn with Bézier curves so that the bubble looks a bit closer to a circle than a rectangle. In the case of bubbles that are length-preserving on objects, we also want to override the drawing of its opening and closing boxes so that the wires go straight through the bubble rather than meeting at the box node.

Example 0.1.18. Drawing of a bubbled diagram and a first-order logic formula.

```
f, g, h = Box('f', x, y), Box('g', y, z), Box('h', y @ z, x)
(f @ g >> h).bubble(dom=a @ b, cod=c, name="$\\beta$").draw()

men, mortal = Predicate('p'), Predicate('q')
(men.cut() >> mortal.dagger()).cut().draw()
```





0.1.6 Automatic diagram recognition

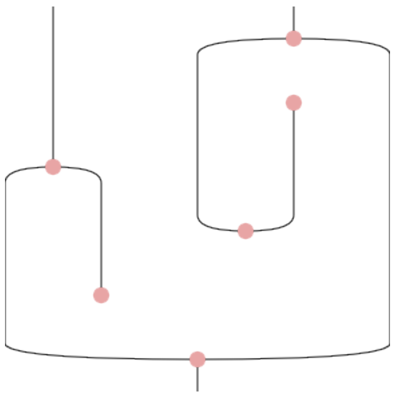
We conclude this section with an application of theorem 0.1.13 to *automatic diagram recognition*: turning pictures of diagrams into diagrams. In listing 0.1.1, we described an abstract reading algorithm which took lgpp graphs as input and returned diagrams. We make it a concrete algorithm by taking *bitmaps* as input: grids of Boolean pixels describing a black-and-white picture. The algorithm `read` listed below takes as input a pair of bitmaps for the box nodes and the wires of the plane graph, it returns a `Diagram`. It is more general than the `graph2diagram` algorithm of listing 0.1.1 where we assumed that the embedding of the graph looked like the output of `Diagram.draw`. i.e. that edges are straight vertical lines. Indeed, our reading algorithm will accept *any bitmaps* as input and always return a valid diagram, however bended the edges are. If the bitmaps indeed represent a progressive generic plane graph G, then we get `read(` G `).draw()` $\sim G$ up to progressive generic deformation. If not, the output will still be a diagram but its drawing may not look anything like the input.

Listing 0.1.19. Implementation of the reading algorithm of listing 0.1.1.

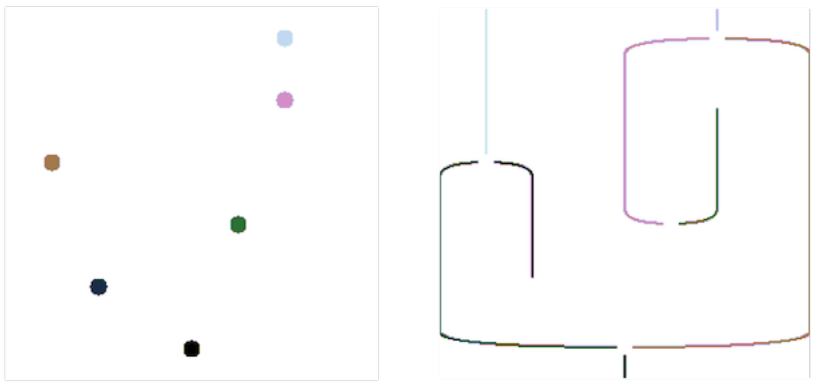
```python
from numpy import array, argmin
from skimage.measure import regionprops, label

connected_components = lambda img: regionprops(label(img))

def read(box_pixels: array, wire_pixels: array) -> Diagram:
    assert wire_pixels.shape == box_pixels.shape
    length, width = box_pixels.shape
    box_nodes, wires = map(connected_components, (box_pixels, wire_pixels))
    critical_heights = array(
        [0] + [int(node.centroid[0]) for node in box_nodes] + [length])
    source, target = [], []
    for wire, region in enumerate(wires):
        top, bottom = (minmax(i for i, _ in region.coords) for minmax in (min, max))
        source.append(argmin(abs(critical_heights - top)))
        target.append(argmin(abs(critical_heights - bottom)))
    print(list(zip(source, target)))
    scan = [wire for wire, node in enumerate(source) if node == 0]
    dom, boxes_and_offsets = Ty('x') ** len(scan), []
    for depth, box_node in enumerate(box_nodes):
        i, j = map(int, box_node.centroid)
        input_wires = [wire for wire in scan if target[wire] == depth + 1]
        output_wires = [wire for wire, node in enumerate(source) if node == depth + 1]
        dom, cod = Ty('x') ** len(input_wires), Ty('x') ** len(output_wires)
        box = Box('box_{}'.format(depth), dom, cod)
        left_of_box = [wire for wire in scan if wire not in input_wires
                       and dict(wires[wire].coords).get(i, width) < j]
        offset = max(len(left_of_box), 0)
        boxes_and_offsets.append((box, offset))
        scan = scan[:offset] + output_wires + scan[offset + len(input_wires):]
    return decode(dom, boxes_and_offsets)
```

We use the `array` *data structure of* `numpy` *[vCV11] for bitmaps. Connected components are computed with* `scikit-image` *[Wal+14], we make use of their default ordering by lexicographic order of their top-left pixel.*

Example 0.1.20. *Suppose we start from the following picture, where the red pixels are boxes and the black pixels are wires:*



We compute the connected components of red and black bitmaps and count $\{1, \dots, 6\}$ *boxes and 8 wires:*



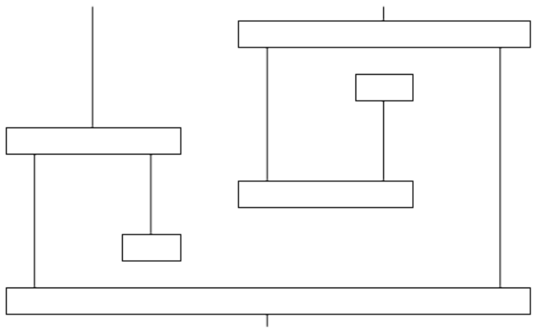
$$0 \to 3, \quad 0 \to 1, \quad 1 \to 4, \quad 1 \to 6, \quad 2 \to 4, \quad 3 \to 6, \quad 3 \to 5 \quad \textit{and} \quad 6 \to 7$$

for 0 *and* 7 *the domain and codomain of the whole diagram respectively.*

From this, we can reconstruct the whole diagram by scanning top to bottom.

```
diagram = x @ box_0\
    >> x @ x @ box_1 @ x\
    >> box_2 @ x @ x @ x\
    >> x @ x @ box_3 @ x\
    >> x @ box_4 @ x\
>> box_5
```

```
diagram.draw(draw_box_labels=False, draw_type_labels=False)
```



Example 0.1.21. *Suppose we start from the following picture.*
BROKEN

applications to automatic analysis of document layout [Bor+19]

0.2 Adding extra structure

0.2.1 Rigidity: wire bending

0.2.2 Symmetry: wire swapping

0.2.3 Cartesian closed categories

0.2.4 Hypergraph categories

0.3 A premonoidal approach

0.3.1 Abstract premonoidal categories

0.3.2 Concrete premonoidal categories

0.3.3 Free premonoidal categories

0.3.4 The state construction

0.4 Related & future work

0.4.1 Graph-based data structures

0.4.2 Higher-dimensional diagrams

1 Quantum natural language processing

2
Diagrammatic differentiation

References

[Bor+19] Emanuela Boros, Alexis Toumi, Erwan Rouchet, Bastien Abadie, Dominique Stutzmann, and Christopher Kermorvant. “Automatic Page Classification in a Large Collection of Manuscripts Based on the International Image Interoperability Framework”. In: *International Conference on Document Analysis and Recognition.* 2019. DOI: 10.1109/ICDAR.2019.00126.

[DV18] Antonin Delpeuch and Jamie Vicary. “Normalization for Planar String Diagrams and a Quadratic Equivalence Algorithm”. In: *arXiv:1804.07832 [cs]* (Apr. 2018). arXiv: 1804.07832 [cs].

[JS88] André Joyal and Ross Street. “Planar Diagrams and Tensor Algebra”. In: *Unpublished manuscript, available from Ross Street’s website* (1988).

[vCV11] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. “The NumPy Array: A Structure for Efficient Numerical Computation”. In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37.

[Wal+14] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. “Scikit-Image: Image Processing in Python”. In: *PeerJ* 2 (June 2014), e453. DOI: 10.7717/peerj.453.