typst-ribbony

Manual

A library for creating ribbony diagrams in Typst, such as Sankey and Chord diagrams.

Content

1 Usage

1.1 Installation

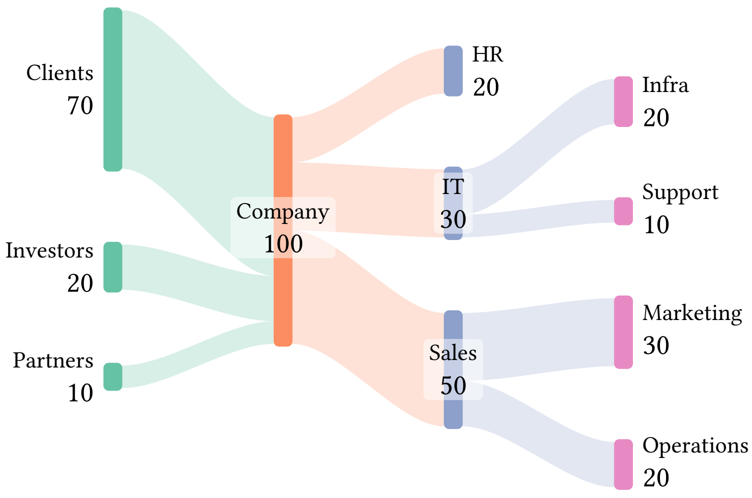
```
#import "@preview/ribbony:0.1.0": *
```

1.2 Quick start

1.2.1 Sankey Diagram

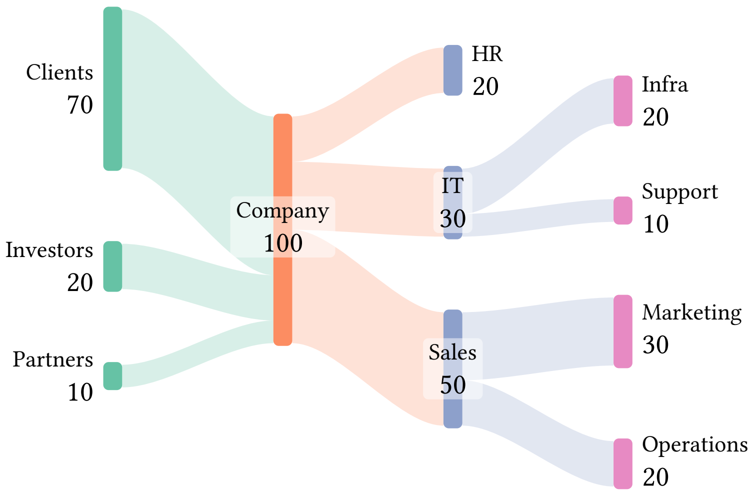
To draw a sankey diagram, uses the `sankey-diagram` function, pass in a dictionary:

```
#sankey-diagram((
  "Clients": ("Company": 70),
  "Investors": ("Company": 20),
  "Partners": ("Company": 10),
  "Company": ("HR": 20, "IT": 30, "Sales":
50),
  "IT": ("Infra": 20, "Support": 10),
  "Sales": ("Marketing": 30, "Operations": 20)
))
```



Alternatively, you can use adjacency list format:

```
#sankey-diagram(
  (
    ("Clients", "Company", 70),
    ("Investors", "Company", 20),
    ("Partners", "Company", 10),
    ("Company", "HR", 20),
    ("Company", "IT", 30),
    ("Company", "Sales", 50),
    ("IT", "Infra", 20),
    ("IT", "Support", 10),
    ("Sales", "Marketing", 30),
    ("Sales", "Operations", 20)
  )
)
```



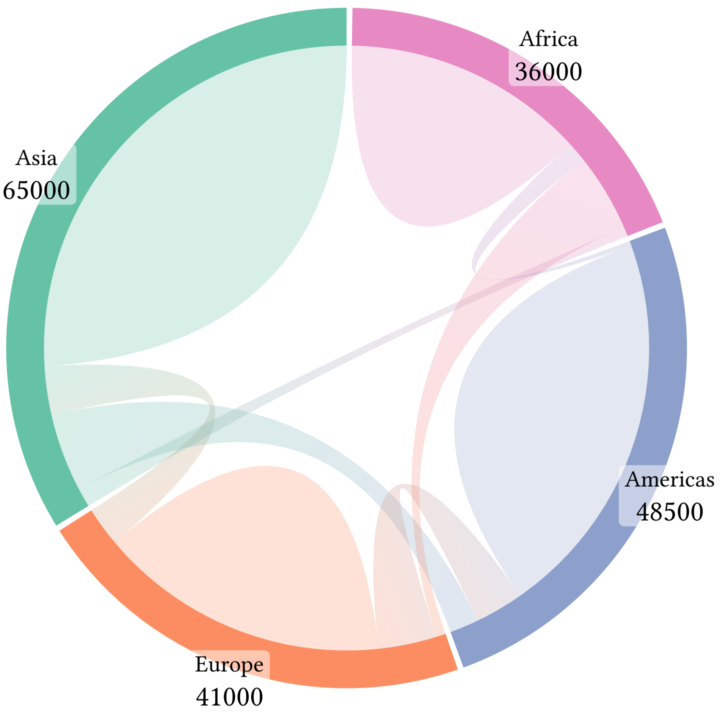
1.2.2 Chord Diagram

To draw a chord diagram, uses the `chord-diagram` function:

```
#chord-diagram((
    "Asia": ("Asia": 50000, "Europe": 5000, "Americas": 8000, "Africa": 2000),
    "Europe": ("Asia": 4000, "Europe": 30000, "Americas": 6000, "Africa": 1000),
    "Americas": ("Asia": 3000, "Europe": 5000, "Americas": 40000, "Africa": 500),
    "Africa": ("Asia": 1000, "Europe": 8000, "Americas": 2000, "Africa": 25000),
))
```

We can use adjacency matrix format to make it simpler:

```
#chord-diagram((
  matrix: (
    (50000, 5000, 8000, 2000),
    (4000, 30000, 6000, 1000),
    (3000, 5000, 40000, 500),
    (1000, 8000, 2000, 25000)
  ),
  ids: ("Asia", "Europe", "Americas",
"Africa")
))
```



1.2.3 Customization

Visit drawing functions section for details.

2 Drawing Functions

The library offers `ribbony-diagram` function as the base to draw any ribbon diagrams.

`sankey-diagram`, `chord-diagram` and other specific functions are just a wrapper of `ribbony-diagram` with preset parameters. So it suffices to get to know `ribbony-diagram` here, as other functions share the same parameters except for different default values.

2.1 `ribbony-diagram`

The core of `ribbony` library.

```
ribbony-diagram(
    data data ,
    aliases: dict ,
    categories: dict ,
    layout: layout ,
    tinter: tinter ,
    ribbon-stylizer: ribbon-stylizer ,
    draw-label: label-drawer ,
    debug: false number ,
) -> content
```

`data` `data` required
The data for the ribbon diagram.

`aliases` `dict`
default: `(:)`
A dictionary for mapping original node ids to display names. Keys are ids in the data, and values are the display names. If not provided, original ids are used.

`categories` `dict`
default: `(:)`
A dictionary for specifying categories for nodes. Keys are categories names, and values are arrays of node ids belonging to each category. Category will be attached to the node properties, can be used by tinter and other customization functions.

`layout` `layout`
default: `layout.auto-linear()`
The layout (including layouter and drawer) to arrange and draw the ribbony.

`tinter` `tinter`
default: `tinter.default-tinter()`
The tinter function to assign colors to nodes.

`ribbon-stylizer` `ribbon-stylizer`
default: `ribbon-stylizer.default()`
The stylizer function to define the appearance of the ribbony, such as color, gradient, stroke, etc.

`draw-label` `label-drawer`
default: `none`
A function to draw labels for each node.

`debug` `false` `number`
default: `false`
Can pass a number (1 to 3) to inspect the data after each major processing step.

1: after pre-processing; 2: after layouter; 3: after tinter.

2.2 `sankey-diagram`

A wrapper of `ribbony-diagram` for drawing sankey diagrams.

Takes the same parameters as `ribbony-diagram`, with different default values for some parameters.

2.3 `chord-diagram`

A wrapper of `ribbony-diagram` for drawing chord diagrams.

Takes the same parameters as `ribbony-diagram`, with different default values for some parameters.

3 Data

`data` is the data source for ribbon diagrams. It supports various forms.

```
data = Adjacency List
     | Adjacency Dictionary
     | Adjacency Matrix
```

Different formats has their own advantages and disadvantages, however, some of them lack certain functoinalities. `Adjacency List` is the most complete format that supports all functionalities, while `Adjacency List` and `Adjacency Matrix` are more concise but lack support of some features (for example, only `Adjacency List` supports edge attributes, and only `Adjacency Dictionary` and `Adjacency List` support multiple edges between two nodes).

Here lists the supported formats.

Definition:

3.1 Supported Formats

3.1.1 Adjacency List

An array of edges, where each edge is represented as a tuple of `(from-id, to-id, size, edge-attrs?)`.

`edge-attrs` is an optional dictionary of edge attributes, which can include a `style` dictionary/function for the edge style attributes (will be merged into the style of ribbon during rendering), and other custom attributes if you want to use them in custom functions. All attributes in `edge-attrs` will be added to the edge properties. The details of `style` attribute is explained in edge style override section.

Definition:

```
Adjacency List = array<
  array (
    from-id: string,
    to-id: string,
    size: number,
    ?edge-attrs: dict ("style"?: dict, ..other-attrs)
  )
>
```

Example:

```
(
  ("Clients", "Company", 70),
  ("Investors", "Company", 5),
  ("Investors", "Company", 15),
  ("Partners", "Company", 10),
  ("Company", "HR", 20),
  ("Company", "IT", 30),
  ("Company", "Sales", 50),
  ("IT", "Infra", 20),
  ("IT", "Support", 10),
  ("Sales", "Marketing", 30),
  ("Sales", "Operations", 20)
)
```

An example with edge attributes:

```
(
  ("A", "B", 10, ("style": ("fill": red, "stroke": black + 1pt))),
  ("A", "C", 20),
  ("C", "D", 15, ("id": "example"))
)
```

3.1.2 Adjacency Dictionary

A dictionary where key are node ids, and values are dictionaries of outgoing edges of which keys are target node ids and values are edge sizes or array of edge sizes.

Definition:

```
Adjacency Dictionary = dict<
  [id: string]: dict<
    [to-id: string]: (size: number | array<size: number>)
  >
>
```

Example:

```
(
  "Clients": ("Company": 70),
  "Investors": ("Company": 20),
  "Partners": ("Company": 10),
  "Company": ("HR": 20, "IT": 30, "Sales": 50),
  "IT": ("Infra": 20, "Support": 10),
  "Sales": ("Marketing": 30, "Operations": 20)
)
```

If there are multiple edges between two nodes, we can use an array instead of a single number:

```
(
  "A": ("B": (5, 10), "C": 15), // A has two edges to B with sizes 5 and 10 respectively
)
```

3.1.3 Adjacency Matrix

A dictionary with two keys: `matrix` and `ids`. `matrix` is a square 2D array representing the adjacency matrix, and `ids` is an array of node ids corresponding to the rows and columns of the matrix.

Definition:

```
Adjacency Matrix = dict<
  "matrix": array<array<number>>,
  "ids": array<string>
>
```

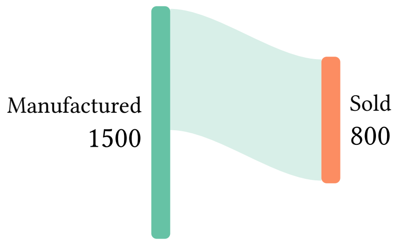
Example:

```
(
  matrix: (
    (50000, 5000, 8000, 2000),
    (4000, 30000, 6000, 1000),
    (3000, 5000, 40000, 500),
    (1000, 8000, 2000, 25000)
  ),
  ids: ("Asia", "Europe", "Americas", "Africa")
)
```

In this example, the `6000` at row 2 column 3 indicates an edge from `Europe` to `Americas` with size `6000`.

3.2 #void and Node Size Override

In the sankey layout, the size of each node is calculated by
max(sum of incoming edge sizes, sum of outgoing edge sizes). In some cases, you may want to customize the size of certain nodes, i.e. to discard some incoming or outgoing portions, for example:



To achieve this, you can use the `#void` node.

`#void` is a special name that can be used in the data to achieve this. It acts as a “hidden node” that absorbs all incoming or outgoing edges without being displayed in the diagram. The edges connected to `#void` will still contribute to the size calculation of the connected nodes, but `#void` itself will not be rendered.

The source code of the above example:

```
#sankey-diagram(
  (
    ("Manufactured", "Sold", 800),
    ("Manufactured", "#void", 700),
  )
)
```

Or you can use `#void` as a source node too, it will give the same result:

```
#sankey-diagram(
  (
    ("Manufactured", "Sold", 800),
    ("#void", "Manufactured", 1500),
  )
)
```

3.3 Behind the scenes

Internally, all data formats will be converted to a new format for easier processing. The document doesn’t list this for now. If you want to know more about this, pass `debug: 1` to `ribbony-diagram` function to see the structure of pre-processed data.

4 Layout

`layout` is a tuple of `(layouter, drawer)`, which defines how to layout and draw the ribbon diagram.

Layout is the main part of the diagram drawing process. The type of diagram is mainly determined by the layout passed to `ribbony-diagram`.

Definition:

```
layout = (layouter, drawer)
layouter = (nodes: processed-data) -> layouted-nodes
drawer = (nodes: layouted-nodes, ribbon-stylizer, draw-label) -> content
```

4.1 Built-in Layouts

Ribbony library provides a few built-in layouts for common ribbon diagrams. They take some parameters to customize the layouting and drawing process, then return the `layout` needed for `ribbony-diagram`.

4.1.1 Auto Linear Layout (Sankey): `layout.auto-linear`

A layout for drawing linear ribbon diagrams (Sankey diagrams).

It automatically arranges nodes in layers based on their topological order, and layouts the the nodes using a linear layouting algorithm.

```
layout.auto-linear(
    layer-gap: number ,
    node-gap: number ,
    node-width: number ,
    base-node-height: number ,
    min-node-height: number ,
    centerize-layer: bool ,
    vertical: bool ,
    layers: dict ,
    radius: length ,
    curve-factor: number ,
) -> layout
```

`layer-gap` `number`
default: `2`
The gap between layers.

`node-gap` `number`
default: `1`
The gap between nodes in the same layer.

`node-width` `number`
default: `0.25`
The width (thickness) of each node.

`base-node-height` `number`
default: `3`
The base height of each node.

`min-node-height` `number`
default: `0.1`
The minimum height of each node.

`centerize-layer` `bool`
default: `false`
Whether to centerize nodes in each layer respectively.

`vertical` `bool`
default: `false`
If true, layout the diagram vertically, otherwise horizontally.

`layers` `dict`
default: `(:)`
A dictionary to override the layer for specific nodes. Keys are layer indices (in string form because typst does not

support integer dict keys), and values are arrays of node ids to be assigned to that layer. A single node id also works for value.

Example:

```
(
  "0": ("A", "B", "C"), // Assign nodes A, B, C to layer 0
  "1": "D", // Assign node D to layer 1
  "2": ("E", "F") // Assign nodes E, F to layer 2
)
```

`radius` `length`
default: `2pt`
The corner radius of nodes.

`curve-factor` `number`
default: `0.3`
The curve factor for the ribbony. Larger values result in more curved ribbony.

4.1.2 Circular Layout (Chord): `layout.circular`

A layout for drawing circular ribbon diagrams (Chord diagrams).

Circular layout does not assign each node a layer. It puts every node on a circle and put ribbony between them.

It has directed and undirected mode. In directed mode, each edge is represented by a ribbon flowing from source node to target node. In undirected mode, multi-edges connecting to the same node pairs are merged, then forward and backward flows are merged into a single ribbon with two different sizes on each side.

```
layout.circular(
    radius: number ,
    node-width: number ,
    node-gap: angle ,
    angle-offset: angle ,
    directed: bool ,
) -> layout
```

`radius` `number`
default: `4`
The radius of the circle.

`node-width` `number`
default: `0.5`
The width (thickness) of each node.

`node-gap` `angle`
default: `1deg`
The gap between nodes in angle.

`angle-offset` `angle`
default: `0deg`
The offset angle for the first node.

`directed` `bool`
default: `false`
Whether the diagram is directed or undirected.

5 Tinter

`tinter` is a function that assigns colors to nodes based on their properties. It takes the data after layouting and returns the new data with colors assigned to each node.

Definition:

```
tinter = (data: layouted-data) -> colored-data
```

Ribbony library provides some built-in tinters, and you can also create your own custom tinter functions as well.

5.1 Built-in Tinters

5.1.1 Default Tinter: `tinter.default-tinter`

Chooses `tinter.layer-tinter` if layers exist, otherwise `tinter.node-tinter`.

5.1.2 Layer Tinter: `tinter.layer-tinter`

Give the nodes in the same layer the same color from the palette.

Definition:

```
tinter.layer-tinter(
    palette: palette ,
) -> tinter
```

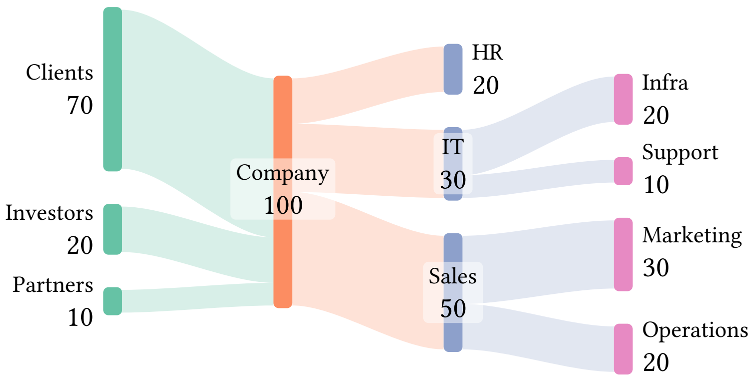
`palette` `palette`

default: `palette.default`

The palette to use for coloring.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  tinter: tinter.layer-tinter()
)
```



5.1.3 Node Tinter: `tinter.node-tinter`

Give each node a different color in palette.

```
tinter.node-tinter(
    palette: palette ,
) -> tinter
```

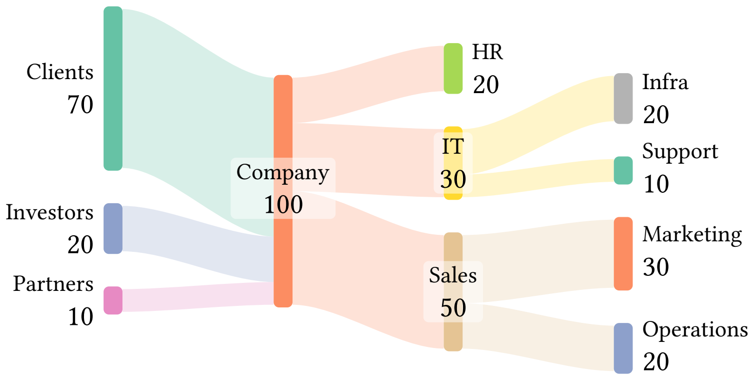
`palette` `palette`

default: `palette.default`

The palette to use for coloring.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  tinter: tinter.node-tinter()
)
```



5.1.4 Categorical Tinter: `tinter.categorical-tinter`

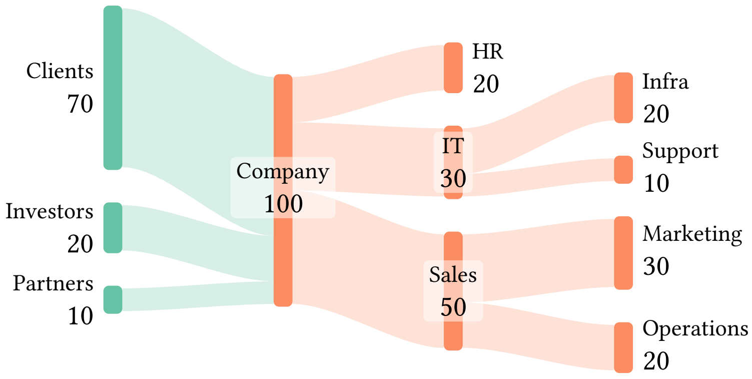
Give the nodes in the same category the same color from the palette. Give all uncategorized nodes a separate color.

```
tinter.categorical-tinter(
    palette: palette ,
) -> tinter
```

`palette` `palette`
default: `palette.default`
The palette to use for coloring.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  categories: (
    "External": ("Clients", "Investors",
"Partners"),
    "Internal": ("Company", "HR", "IT",
"Sales", "Infra", "Support", "Marketing",
"Operations")
  ),
  tinter: tinter.categorical-tinter()
)
```



5.1.5 Dict Tinter: `tinter.dict-tinter`

Tint each node based on a provided color map dictionary you specified.

```
tinter.dict-tinter(
    dict color-map ,
    override-on: tinter ,
) -> tinter
```

`color-map` `dict` required
A dictionary mapping node ids to colors.

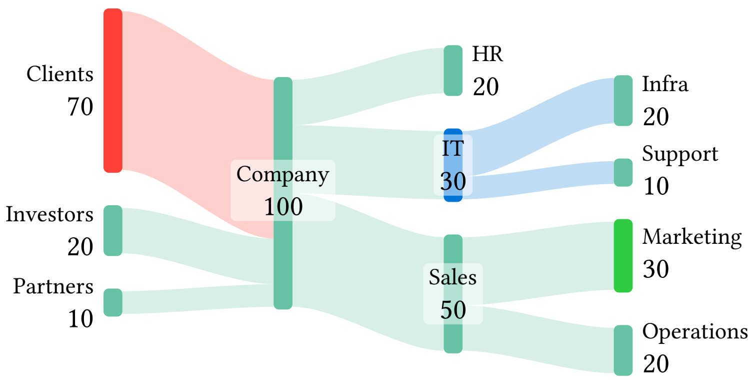
`override-on` `tinter`

default: `none`
An optional tinter function. If passed, it will be called first to assign colors to nodes, then the colors will be overridden by the `color-map` if defined in it.

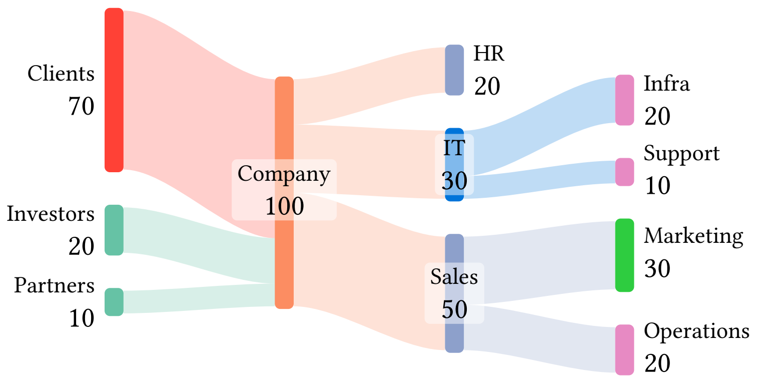
It will assign a default color if a node's color is unspecified.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  tinter: tinter.dict-tinter(
    (
      "Clients": red,
      "IT": rgb("#0074d9"),
      "Marketing": green,
    )
  )
)
```



```
#sankey-diagram(
  (
    // Data
  ),
  tinter: tinter.dict-tinter(
    (
      "Clients": red,
      "IT": rgb("#0074d9"),
      "Marketing": green,
    ),
    override-on: tinter.layer-tinter(
  )
)
```



5.1.6 Constant Tinter: `tinter.constant-tinter`

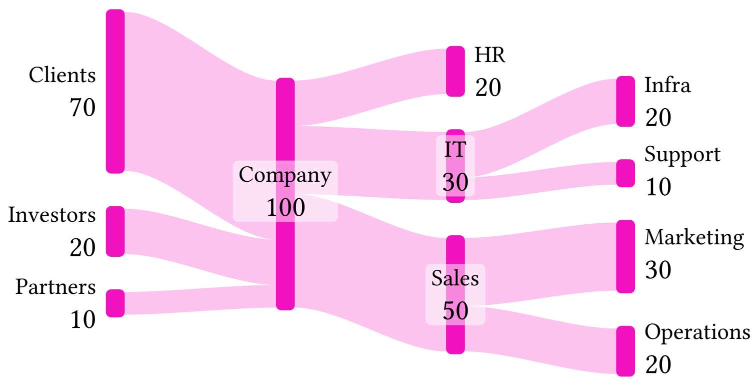
Assign the same color to all nodes.

```
tinter.constant-tinter(
    color: color gradient tilling ,
) -> tinter
```

`color` `color` `gradient` `tilling`
default: `pelette.default.at(0)`
The color to assign to all nodes.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  tinter: tinter.constant-tinter(color:
fuchsia)
),
```



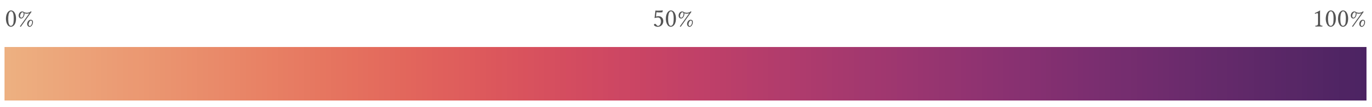
5.2 Make a Custom Tinter

You can create your own custom tinter functions by defining a function that takes the layouted data as input and returns the colored data. Layouted data is a dictionary that maps node ids to their properties, including position, size, and any other attributes added during the layouting process. To see the structure of layouted data, you can pass `debug: 2` to `ribbony-diagram` function to inspect the data after layouting.

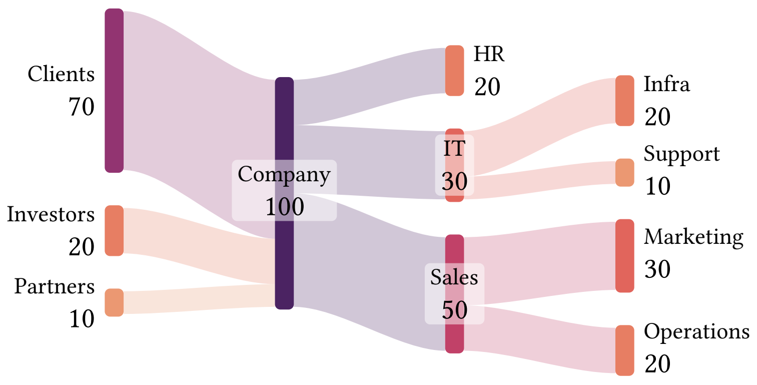
Tinter function should iterate over each node in the layouted data, and set their `color` field to the desired color.

Example:

A tinter that assigns colors based on the size of each node. It uses a gradient to to map node sizes to colors:



```
#sankey-diagram(
  (
    // Data
  ),
  tinter: (nodes) => {
    let max-size = calc.max(..nodes.keys().map(id => nodes.at(id).size))
    for (node-id, properties) in nodes {
      let color = gradient.linear(..color.map.flare).sample(properties.size / max-size * 100%)
      nodes.at(node-id).insert("color", color)
    }
    return nodes
  }
),
```



Here is what one of the nodes (`Sales`) looks like in the passed data:

```
(
    // ...
    Sales: (
        edges: (
            (from: "Sales", to: "Marketing", size: 30),
            (from: "Sales", to: "Operations", size: 20),
        ),
        from-edges: ((from: "Company", to: "Sales", size: 50),),
        id: "Sales",
        name: "Sales",
        number-id: 6,
        in-size: 50,
        out-size: 50,
        layer: 2,
        size: 50,
        width: 0.25,
        height: 1.5,
        x: 4.625,
        y: 1.05,
    ),
    // ...
)
```

5.3 Palette (`tinter.palette`)

A palette is an array of `color`, used by tinters to assign colors to nodes. Ribbony library provides some built-in palettes in `palette` module.

Definition:

```
palette = array<color>
```

Example:

```
(aqua, teal, purple, red, orange, yellow, green) // array of typst built-in colors
```

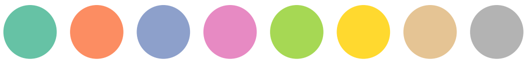
```
(rgb("#555555"),  rgb("#777777"), rgb("#999999")) // shade of gray
```

5.3.1 Built-in Palettes

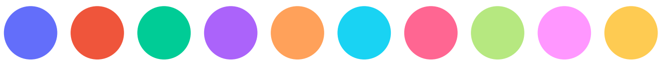
5.3.1.1 Default Palette: `tinter.palette.default`

The default palette is `color-brewer`.

5.3.1.2 Color Brewer Palette: `tinter.palette.color-brewer`



5.3.1.3 Plotly Palette: `tinter.palette.plotly`



5.3.1.4 Plotly Pastel Palette: `tinter.palette.plotly-pastel`

5.3.1.5 Catppuccin Palette: `tinter.palette.catppuccin`

6 Ribbon Stylizer

Ribbon stylizers give styles to ribbony based on such as color, gradient, stroke, etc.

`ribbon-stylizer` is a function that takes 5 positional parameters: `edge`, `from-color`, `to-color`, `from-node`, `to-node`, and potentially some extra parameters (such as `angle` for gradient direction) that drawer may pass in for extra information. It returns a dictionary of style attributes for the ribbon representing the edge, keys including `fill`, `stroke`, etc.

For the details of `edge`, please read internal types: edge section.

Definition:

```
ribbon-stylizer = (edge: edge, from-color, to-color, from-node, to-node, ..extra-params) ->
ribbon-style-dict
```

The library provides some built-in ribbon stylizers.

6.1 Built-in Ribbon Stylizers

6.1.1 Match From Node Color: `ribbon-stylizer.match-from`

Match the ribbon color to the same as the `from-node` color.

```
ribbon-stylizer.match-from(
    transparency: ratio ,
    stroke-width: length ,
    stroke-color: auto color gradient tilling ,
) -> ribbon-stylizer
```

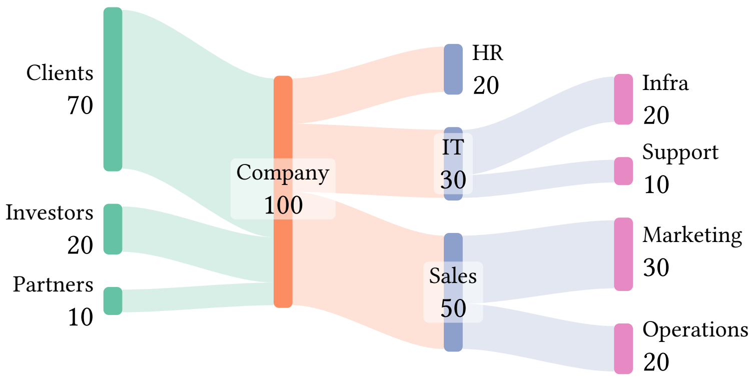
`transparency` `ratio`
default: `75%`
The transparency of the ribbon fill color.

`stroke-width` `length`
default: `0pt`
The width of the ribbon border.

`stroke-color` `auto` `color` `gradient` `tilling`
default: `auto`
The color of the ribbon border. If set to `auto`, it will use the same color as the ribbon.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  ribbon-stylizer: ribbon-stylizer.match-
from()
)
```



6.1.2 Match To Node Color: `ribbon-stylizer.match-to`

Match the ribbon color to the same as the `to-node` color.

```
ribbon-stylizer.match-to(
    transparency: ratio ,
    stroke-width: length ,
    stroke-color: auto  color  gradient  tilling ,
) -> ribbon-stylizer
```

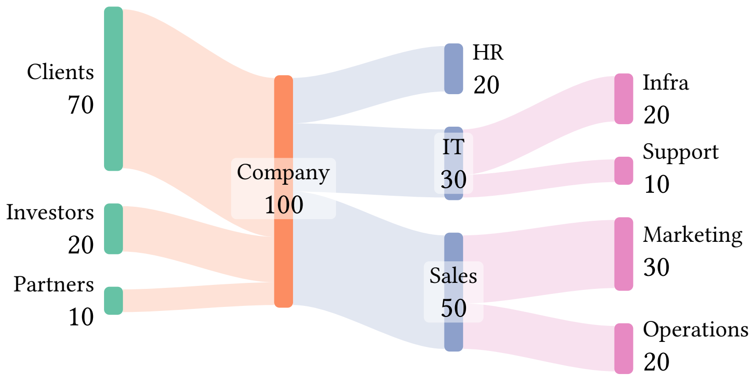
`transparency` `ratio`
default: `75%`
The transparency of the ribbon fill color.

`stroke-width` `length`
default: `0pt`
The width of the ribbon border.

`stroke-color` `auto` `color` `gradient` `tilling`
default: `auto`
The color of the ribbon border. If set to `auto`, it will use the same color as the ribbon.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  ribbon-stylizer: ribbon-stylizer.match-to()
)
```



6.1.3 Gradient from-to: `ribbon-stylizer.gradient-from-to`

Fill the ribbon with a gradient from `from-node` color to `to-node` color.

```
ribbon-stylizer.gradient-from-to(
    transparency: ratio ,
    stroke-width: length ,
    stroke-color: auto  color  gradient  tilling ,
) -> ribbon-stylizer
```

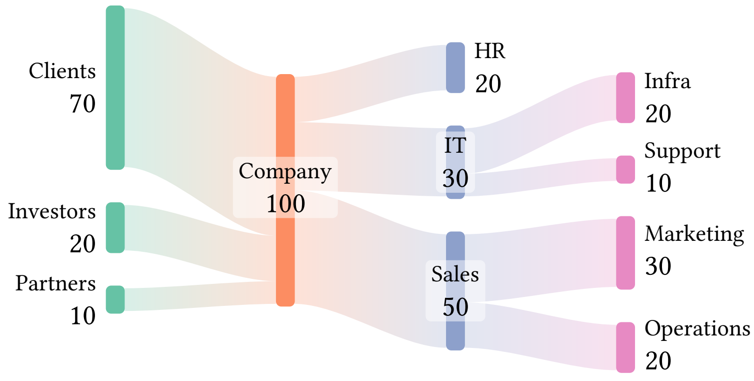
`transparency` `ratio`
default: `75%`
The transparency of the ribbon fill color.

`stroke-width` `length`
default: `0pt`
The width of the ribbon border.

`stroke-color` `auto` `color` `gradient` `tilling`
default: `auto`
The color of the ribbon border. If set to `auto`, it will use the same color as the ribbon.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  ribbon-stylizer: ribbon-stylizer.gradient-
from-to()
)
```



6.1.4 solid-color: `ribbon-stylizer.solid-color`

Fill the ribbon with a specific solid color. `ribbon-stylizer.solid-color(`

```
     color  gradient  tilling color ,
    stroke-width: length ,
    stroke-color: auto  color  gradient  tilling ,
) -> ribbon-stylizer
```

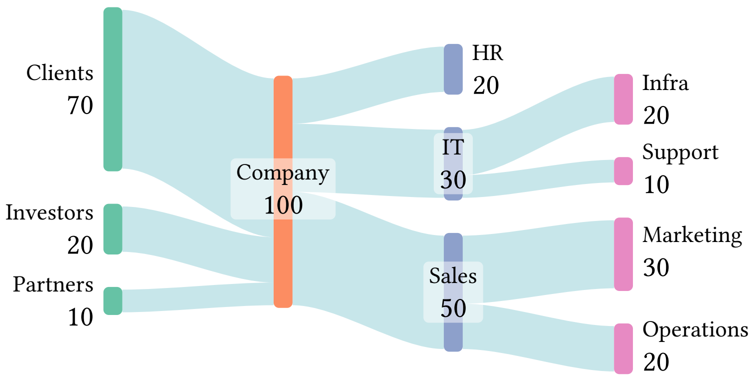
`color` `color` `gradient` `tilling` required
The color to fill the ribbon.

`stroke-width` `length`
default: `0pt`
The width of the ribbon border.

`stroke-color` `auto` `color` `gradient` `tilling`
default: `auto`
The color of the ribbon border. If set to `auto`, it will use the same color as the ribbon.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  ribbon-stylizer: ribbon-stylizer.solid-color(rgb("#239dad").transparentize(75%))
)
```



6.1.5 Match Greater Node Color: `ribbon-stylizer.match-greater`

Match the ribbon color to the color of the node with greater size between `from-node` and `to-node`. `ribbon-stylizer.match-greater(`

```
    transparency: ratio ,
    stroke-width: length ,
    stroke-color: auto  color  gradient  tilling ,
) -> ribbon-stylizer
```

`transparency` `ratio`

default: `75%`

The transparency of the ribbon fill color.

`stroke-width` `length`

default: `0pt`

The width of the ribbon border.

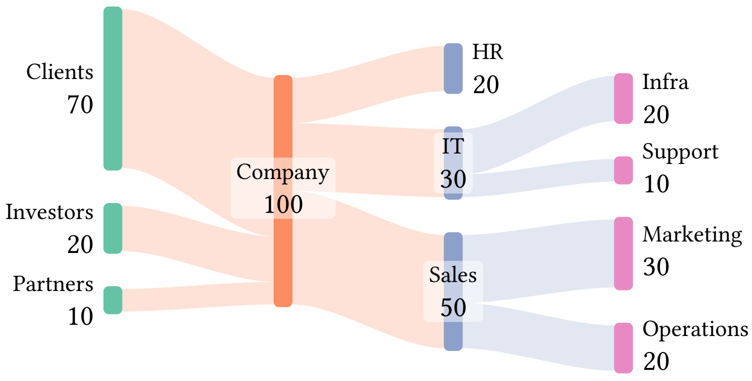
`stroke-color` `auto` `color` `gradient` `tilling`

default: `auto`

The color of the ribbon border. If set to `auto`, it will use the same color as the ribbon.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  ribbon-stylizer: ribbon-stylizer.match-greater()
)
```



6.1.6 Match Lesser Node Color: `ribbon-stylizer.match-lesser`

Match the ribbon color to the color of the node with lesser size between `from-node` and `to-node`. `ribbon-stylizer.match-lesser(`

```
    transparency: ratio ,
    stroke-width: length ,
    stroke-color: auto  color  gradient  tilling ,
) -> ribbon-stylizer
```

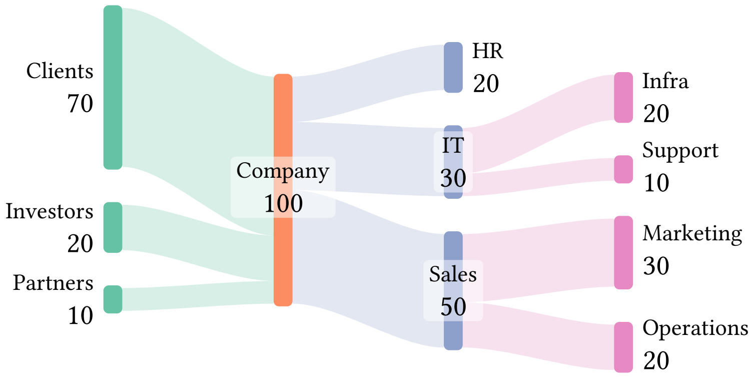
`transparency` `ratio`
default: `75%`
The transparency of the ribbon fill color.

`stroke-width` `length`
default: `0pt`
The width of the ribbon border.

`stroke-color` `auto` `color` `gradient` `tilling`
default: `auto`
The color of the ribbon border. If set to `auto`, it will use the same color as the ribbon.

Example:

```
#sankey-diagram(
  (
    // Data
  ),
  ribbon-stylizer: ribbon-stylizer.match-
lesser()
)
```



6.1.7 Match Greater Direction Node Color: `ribbon-stylizer.match-greater-direction`

This stylizer only applies to undirected diagrams (such as undirected circular layout). It matches the ribbon color to the color of the node with greater size in the direction of flow of the current edge. `ribbon-stylizer.match-greater-direction(`

```
    transparency: ratio ,
    stroke-width: length ,
    stroke-color: auto color gradient tilling ,
) -> ribbon-stylizer
```

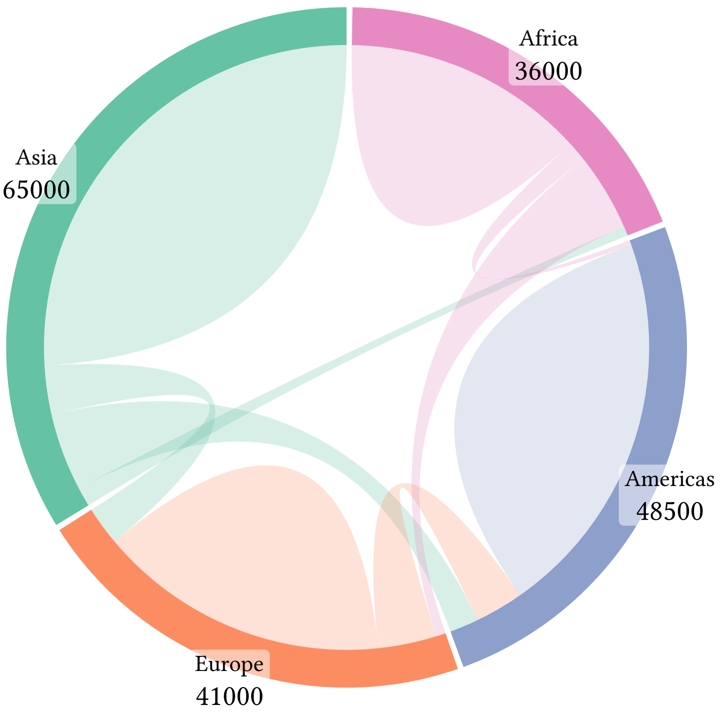
`transparency` `ratio`
default: `75%`
The transparency of the ribbon fill color.

`stroke-width` `length`
default: `0pt`
The width of the ribbon border.

`stroke-color` `auto` `color` `gradient` `tilling`
default: `auto`
The color of the ribbon border. If set to `auto`, it will use the same color as the ribbon.

Example:

```
#chord-diagram(
  (
    // Data
  ),
  ribbon-stylizer: ribbon-stylizer.match-
greater-direction()
)
```



6.1.8 Match Lesser Direction Node Color: `ribbon-stylizer.match-lesser-direction`

This stylizer only applies to undirected diagrams (such as undirected circular layout). It matches the ribbon color to the color of the node with lesser size in the direction of flow of the current edge.

```
ribbon-stylizer.match-lesser-direction(
    transparency: ratio ,
    stroke-width: length ,
    stroke-color: auto color gradient tilling ,
) -> ribbon-stylizer
```

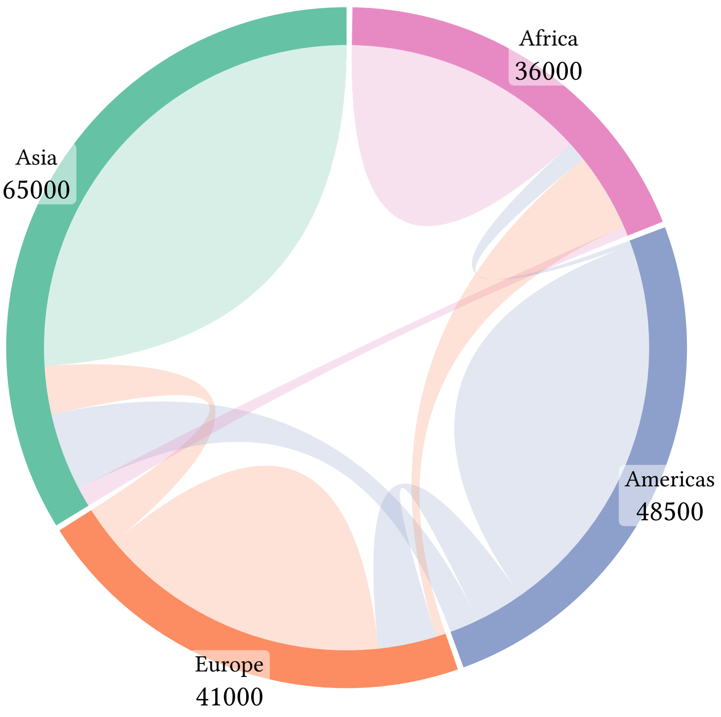
`transparency` `ratio`
default: `75%`
The transparency of the ribbon fill color.

`stroke-width` `length`
default: `0pt`
The width of the ribbon border.

`stroke-color` `auto` `color` `gradient` `tilling`
default: `auto`
The color of the ribbon border. If set to `auto`, it will use the same color as the ribbon.

Example:

```
#chord-diagram(
  (
    // Data
  ),
  ribbon-stylizer: ribbon-stylizer.match-
lesser-direction()
)
```



6.2 Make a Custom Ribbon Stylizer

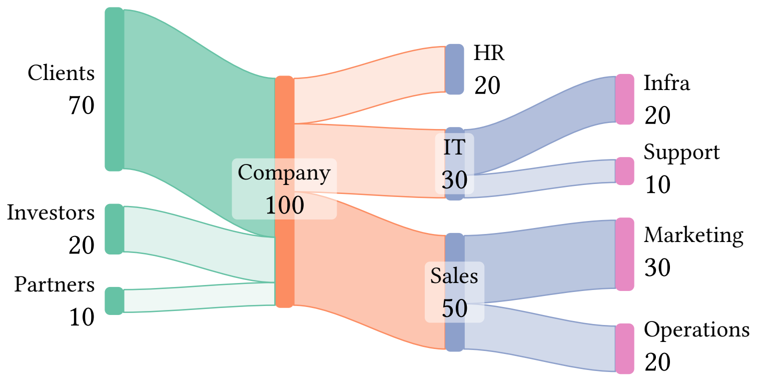
You can create your own custom ribbon stylizer functions by defining a function that takes the required parameters and returns a dictionary of style attributes for the ribbon.

Recall that the parameters are:
- `edge`: The edge object representing the ribbon, which includes fields `from`, `to`, and `size`. For the details, please read internal types: edge section.
- `from-color`: The color of the `from-node`.
- `to-color`: The color of the `to-node`.
- `from-node`: The node object representing the from node.
- `to-node`: The node object representing the to node.
- ..extra-params: Additional parameters that drawer may pass in for extra information.

Make sure to always include `..` at the end of the parameter list to accept the extra potentially unwanted parameters.

Example: A ribbon stylizer that fills the ribbon with the same color as the `from-node`, but with a transparency that matches the size of the edge relative to the the larger one of from-node's total outgoing size and to-node's total incoming size:

```
#sankey-diagram(
  (
    // Data
  ),
  ribbon-stylizer: (edge, from-color, to-color, from-node, to-node, ..) => {
    let percentage = edge.size / calc.max(from-node.out-size, to-node.in-size)
    let transparency = 100% * (1 - percentage)
    return (
      "fill": from-color.transparentize(transparency),
      "stroke": 0.5pt + from-color,
    )
  }
),
```



6.3 Edge style override

The adjacency list definition of edges supports an optional `edge-attrs` dictionary, which can:

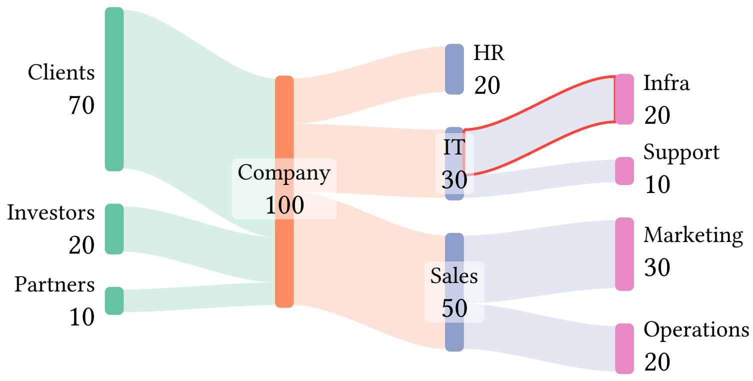
- include a `style` attribute, which can be
  - a dictionary for overriding the style of the specific edge/ribbon, after the ribbon stylizer is applied. The keys and values in the `style` dictionary will be merged into the style of the ribbon.
  - a function that returns a dictionary for dynamic style override based on the edge and node properties. The function takes 3 positional parameters: `edge`, `from-node`, `to-node`, and returns a dictionary of style attributes to override the ribbon style.
- include other custom attributes if you want them to be included in the edge properties, so you can use them in custom ribbon stylizers. Such attributes will be accessible in the `edge` parameter passed to the ribbon stylizer or edge style override function.

Example:

We want to set a specific ribbon to have a red border of 1pt width. We can define the edge like this:

```
("IT", "Infra", 20, (styles: ("stroke": red + 1pt))),
```

```
#sankey-diagram(
  (
    ("Clients", "Company", 70),
    ("Investors", "Company", 20),
    ("Partners", "Company", 10),
    ("Company", "HR", 20),
    ("Company", "IT", 30),
    ("Company", "Sales", 50),
    ("IT", "Infra", 20, (styles: ("stroke":
red + 1pt))),
    ("IT", "Support", 10),
    ("Sales", "Marketing", 30),
    ("Sales", "Operations", 20)
  )
),
```



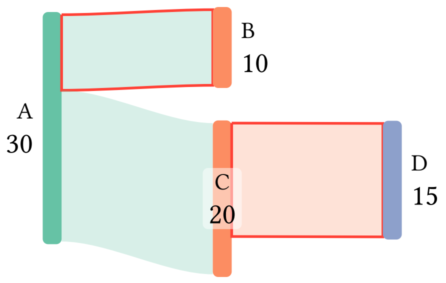
Example:

We want to highlight a series of edges we specified by adding a border to them. We can define the edges like this:

```
("A", "B", 10, (highlighted: true)),
("A", "C", 20),
("C", "D", 15, (highlighted: true)),
```

Then we can create a custom ribbon stylizer that adds a border to the highlighted edges:

```
#sankey-diagram(
  (
    ("A", "B", 10, (highlighted: true)),
    ("A", "C", 20),
    ("C", "D", 15, (highlighted: true)),
  ),
  ribbon-stylizer: (edge, from-color, to-color, from-node, to-node, ..) => {
    let stroke = if (edge.at("highlighted", default: false)) {
      red + 1pt
    } else {
      none
    }
    return (
      fill: from-color.transparentize(75%),
      "stroke": stroke,
    )
  }
)
```



7 Label Drawer

`label-drawer` is a function that takes a `node-name`, `properties` and other extra parameters about the layout itself, such as `layer-gap` for linear layout, and returns the Cetz content to draw the label for the node.

Definition:

```
label-drawer = (node-name: string, properties: dict, ..extra-params) -> CetZ-content
```

`node-name` is NOT the id of the node, but the name corresponding to the node in CetZ, which can be use for relative positioning.

7.1 Built-in Label Drawers

7.1.1 Default Linear Label Drawer: `label-drawer.default-linear-label-drawer`

The default label drawer for linear layout. It is the labels you've seen in the previous sankey examples.

```
label-drawer.default-linear-label-drawer(
    snap: "io-auto" auto align ,
    offset: auto CetZ-coord ,
    width-limit: auto false number length ,
    styles: dict ,
    draw-content: function ,
    formatter: function ,
) -> label-drawer
```

`snap` `"io-auto"` `auto` `align`
default: `"io-auto"`
Where to snap the label to the node. For horizontal layout, accepts `left`, `center`, `right`. For vertical layout, accepts `top`, `center`, `bottom`.
If set to `auto`, it will choose `right` for horizontal layout and `bottom` for vertical layout.
If set to `"io-auto"`, it will choose `right`/`bottom` for nodes with only outgoing edges, `left`/`top` for nodes with only incoming edges, and `center` for nodes with both incoming and outgoing edges.

`offset` `auto` `CetZ-coord`
default: `auto`
The offset of the label from the snap position. If set to `auto`, it will offset the label a small distance away from the node.

`width-limit` `auto` `false` `number` `length`
default: `auto`
The maximum width of the label. If set to `auto`, it will be set to the layer gap minus a small padding. If set to `false`, there will be no width limit.

`styles` `dict`
default: `(inset: 0.2em, fill: white.transparentize(50%), radius: 2pt)`
A dictionary of styles to apply to the label background.

`draw-content` `function`
default:

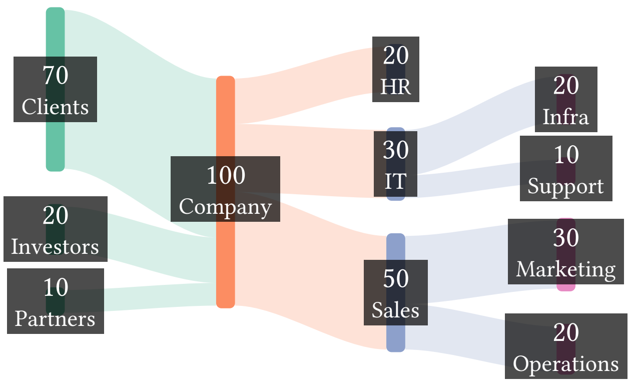
```
(properties, formatter: (val) => val) => [
  #set par(leading: 0.5em)
  #text(properties.name, size: 0.8em) \
  #text(str(formatter(properties.size)), size: 1em)
]
```

A function that takes the node properties and returns the content to draw inside the label.

`formatter` `function`
default: `(val) => val`
A function to format the size value when displaying it in the label. Takes a number and returns a primitive that can be converted to string.

Example: Display translucent black labels without radius, centered at the nodes, with 3pt inset and showing size before name:

```
#sankey-diagram(
  (
    // Data
  ),
  draw-label: label.default-linear-label-
drawer(
    snap: center,
    styles: (
      fill: black.transparentize(30%),
      radius: 0pt,
      inset: 3pt
    ),
    draw-content: (properties, ..) => [
      #set par(leading: 0.5em)
      #set text(fill: white)
      #text(str(properties.size), size: 1em) \
      #text(properties.name, size: 0.8em)
    ]
  )
)
```



7.2 Default Circular Label Drawer: `label-drawer.default-circular-label-drawer`

The default label drawer for circular layout. It places labels on the circumference of the circle.

```
label-drawer.default-circular-label-drawer(
    offset: number ,
    styles: dict ,
    draw-content: function ,
    formatter: function ,
) -> label-drawer
```

`offset` `number`
default: `0.2`
The offset of the label away from the center of the circle.

`styles` `dict`
default: `(inset: 0.2em, fill: white.transparentize(50%), radius: 2pt)`
A dictionary of styles to apply to the label background.

`draw-content` `function`
default:

```
(properties, formatter: (val) => val) => [
  #set par(leading: 0.5em)
  #text(properties.name, size: 0.8em) \
  #text(str(formatter(properties.size)), size: 1em)
]
```

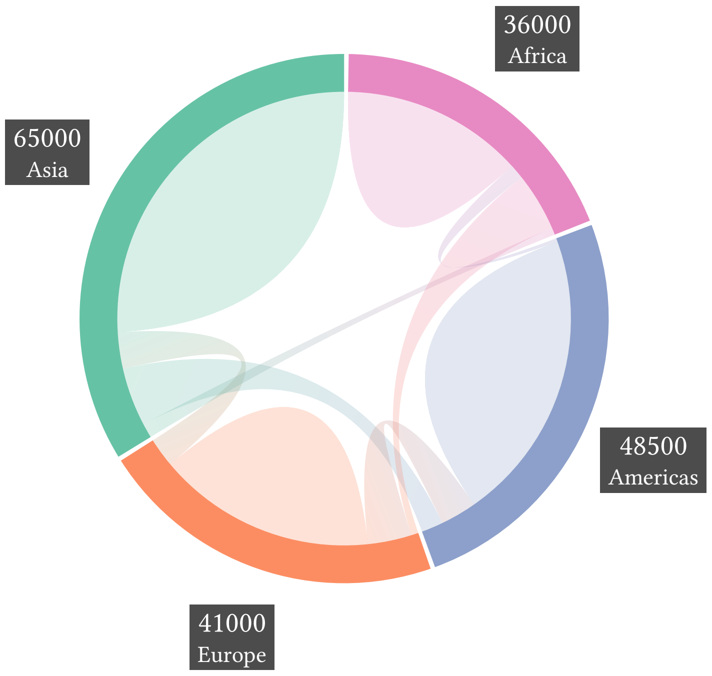
A function that takes the node properties and returns the content to draw inside the label.

`formatter` `function`
default: `(val) => val`

A function to format the size value when displaying it in the label. Takes a number and returns a primitive that can be converted to string.

Example: Display translucent black labels without radius, 1cm away from the center, with 3pt inset and showing size before name:

```
#chord-diagram(
  (
    // Data
  ),
  draw-label: label.default-circular-label-
drawer(
    offset: 1cm,
    styles: (
      fill: black.transparentize(30%),
      radius: 0pt,
      inset: 3pt
    ),
    draw-content: (properties, ..) => [
      #set par(leading: 0.5em)
      #set text(fill: white)
      #text(str(properties.size), size: 1em) \
      #text(properties.name, size: 0.8em)
    ]
  )
)
```



7.3 Make a Custom Label Drawer

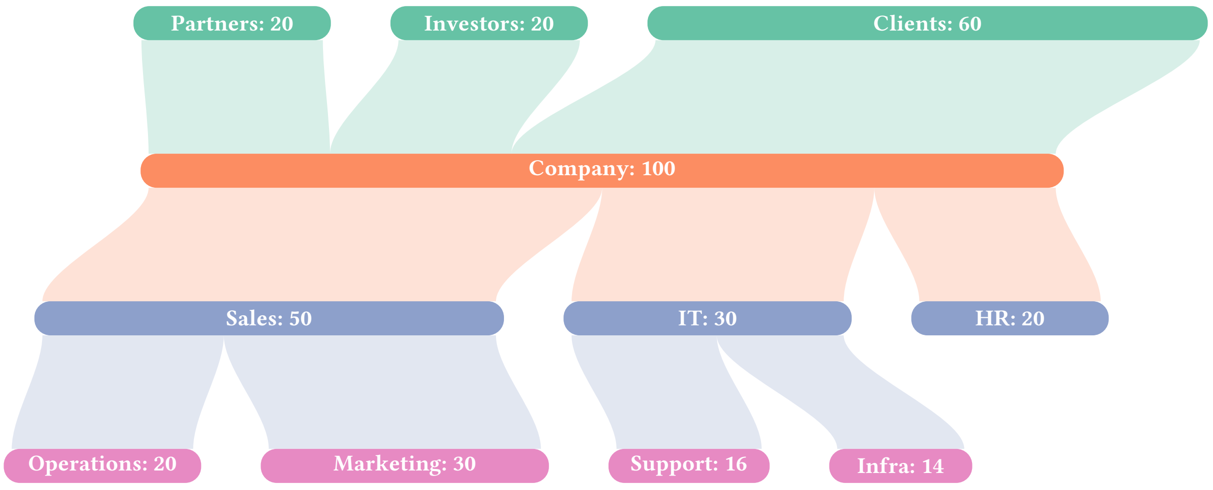
You can create your own custom label drawer functions by defining a function that takes the properties of node as parameter and returns the CetZ content to draw the label.

Example:

We want to show text inside each nodes in the vertical linear layout, with the name on the left.

```
#sankey-diagram(
  (
    "Clients": ("Company": 60),
    "Investors": ("Company": 20),
    "Partners": ("Company": 20),
    "Company": ("HR": 20, "IT": 30, "Sales": 50),
    "IT": ("Infra": 14, "Support": 16),
    "Sales": ("Marketing": 30, "Operations": 20)
  ),
  layout: layout.auto-linear(
    vertical: true,
    node-gap: 1,
    base-node-height: 12,
    node-width: 0.45,
    radius: 6pt,
    layer-gap: 1.5,
  ),
  draw-label: (node-name, properties, ..) => {
    import "@preview/cetz:0.4.2"
    import cetz.draw: *
    content(
      (node-name + ".center"),
      text(properties.name + ": " + str(properties.size), size: 0.3cm, fill: white, weight:
"bold")
    )
  }

)
```

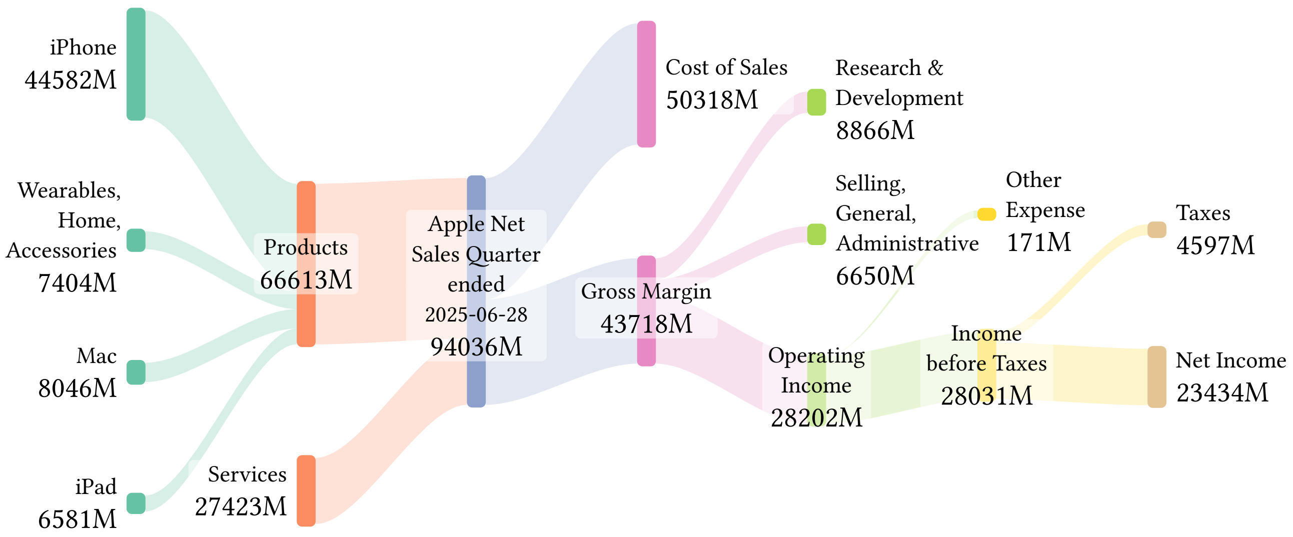


8 Showcase

8.1 Apple Financial Report

Data source: https://sankeymatic.com/data/apple/#diagram-notes-2025-q3, https://www.apple.com/newsroom/pdfs/fy2025-q3/FY25_Q3_Consolidated_Financial_Statements.pdf (ISC)

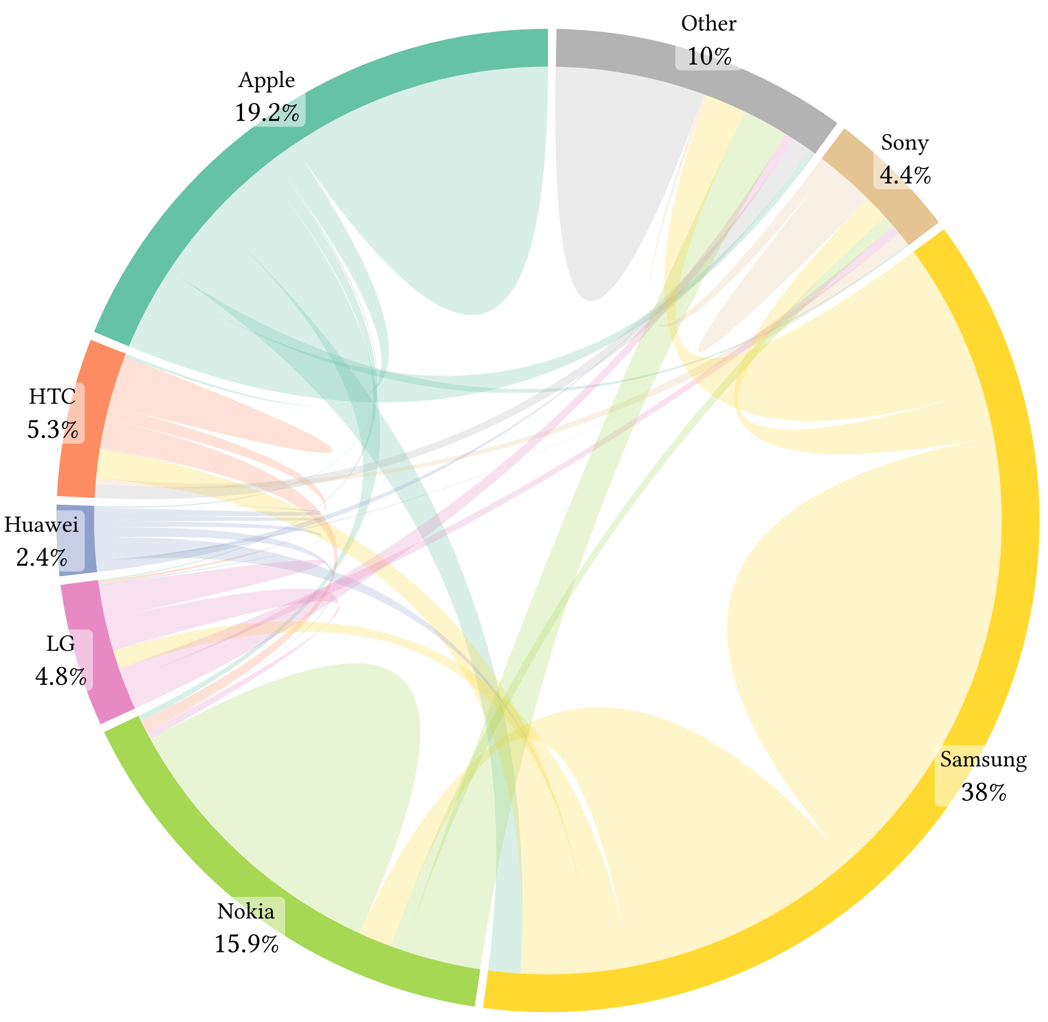
```
#sankey-diagram(
  (
    ("iPhone", "Products", 44582),
    ("Wearables, Home, Accessories", "Products", 7404),
    ("Mac", "Products", 8046),
    ("iPad", "Products", 6581),
    ("Products", "Apple Net Sales Quarter ended 2025-06-28", 800),
    ("Products", "Apple Net Sales Quarter ended 2025-06-28", 42820),
    ("Services", "Apple Net Sales Quarter ended 2025-06-28", 6698),
    ("Products", "Apple Net Sales Quarter ended 2025-06-28", 22993),
    ("Services", "Apple Net Sales Quarter ended 2025-06-28", 20725),
    ("Apple Net Sales Quarter ended 2025-06-28", "Cost of Sales", 800),
    ("Apple Net Sales Quarter ended 2025-06-28", "Cost of Sales", 42820),
    ("Apple Net Sales Quarter ended 2025-06-28", "Cost of Sales", 6698),
    ("Apple Net Sales Quarter ended 2025-06-28", "Gross Margin", 22993),
    ("Apple Net Sales Quarter ended 2025-06-28", "Gross Margin", 20725),
    ("Gross Margin", "Research & Development", 8866),
    ("Gross Margin", "Selling, General, Administrative", 6650),
    ("Gross Margin", "Operating Income", 28202),
    ("Operating Income", "Other Expense", 171),
    ("Operating Income", "Income before Taxes", 28031),
    ("Income before Taxes", "Taxes", 4597),
    ("Income before Taxes", "Net Income", 23434)
  ),
  layout: layout.auto-linear(node-gap: 1.5),
  draw-label: label.default-linear-label-drawer(formatter: (val) => str(val) + "M"),
)
```



8.2 Phone Brands Loyalty Flow

Data source: https://gist.github.com/nbremer/94db779237655907b907

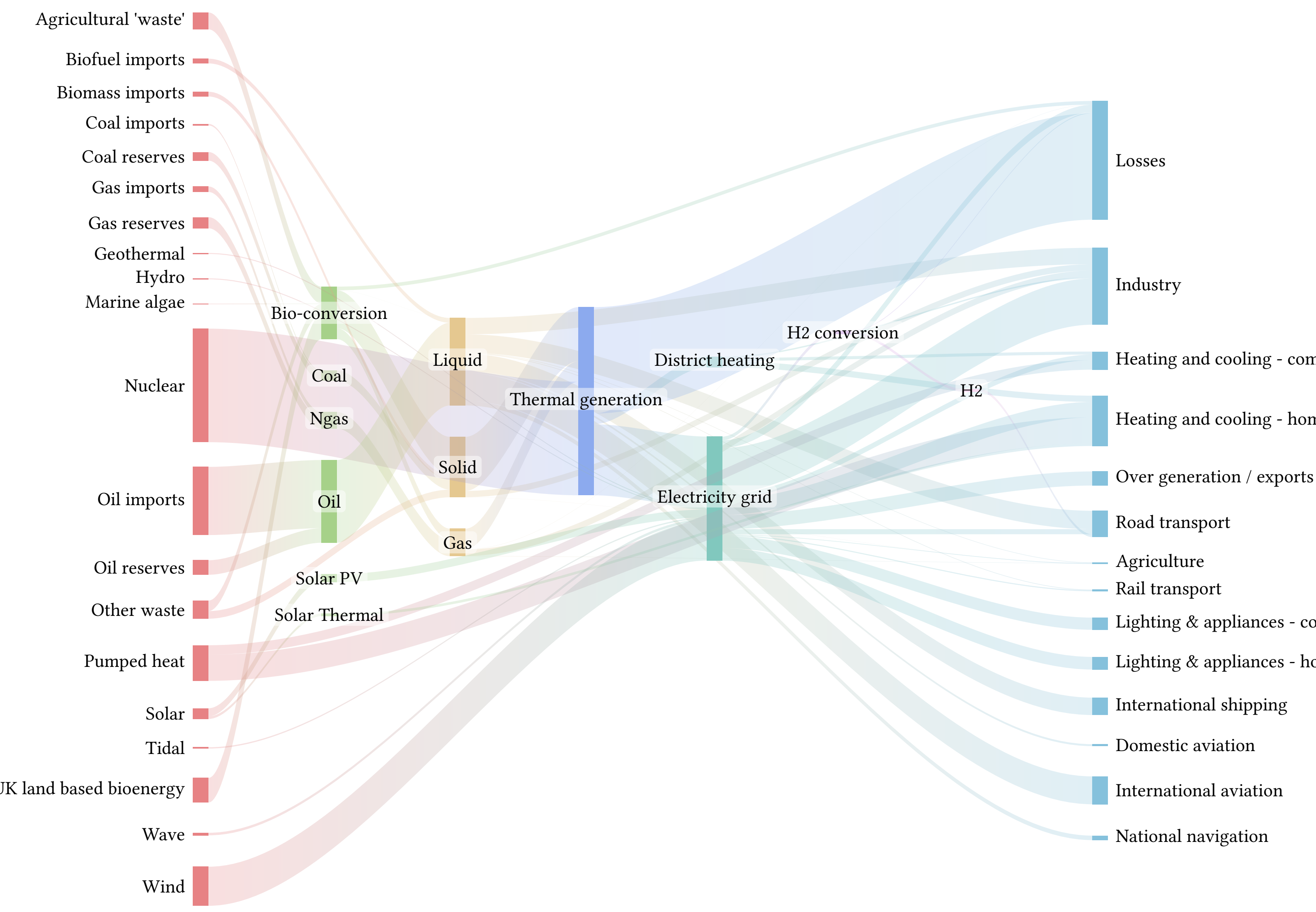
```
#chord-diagram(
  (
    matrix: (
      (9.6899, 0.8859, 0.0554, 0.443, 2.5471, 2.4363, 0.5537, 2.5471),
      (0.1107, 1.8272, 0, 0.4983, 1.1074, 1.052, 0.2215, 0.4983),
      (0.0554, 0.2769, 0.2215, 0.2215, 0.3876, 0.8306, 0.0554, 0.3322),
      (0.0554, 0.1107, 0.0554, 1.2182, 1.1628, 0.6645, 0.4983, 1.052),
      (0.2215, 0.443, 0, 0.2769, 10.4097, 1.2182, 0.4983, 2.8239),
      (1.1628, 2.6024, 0, 1.3843, 8.7486, 16.8328, 1.7165, 5.5925),
      (0.0554, 0.4983, 0, 0.3322, 0.443, 0.8859, 1.7719, 0.443),
      (0.2215, 0.7198, 0, 0.3322, 1.6611, 1.495, 0.1107, 5.4264)
    ),
    ids: ("Apple", "HTC", "Huawei", "LG", "Nokia", "Samsung", "Sony", "Other")
  ),
  draw-label: label.default-circular-label-drawer(formatter: (val) => str(calc.round(val,
digits:1)) + "%"),
  ribbon-stylizer: ribbon-stylizer.match-greater-direction(),
  layout: layout.circular(radius: 6),
)
```



8.3 Energy Flow

Data source: https://observablehq.com/@d3/sankey-component, https://www.gov.uk/guidance/2050-pathways-analysis (ISC, Open Government Licence v3.0)

```
#sankey-diagram(
  (
    ("Agricultural 'waste'", "Bio-conversion", 124.729),
    ("Bio-conversion", "Liquid", 0.597),
    ("Bio-conversion", "Losses", 26.862),
    ("Bio-conversion", "Solid", 280.322),
    ("Bio-conversion", "Gas", 81.144),
    ("Biofuel imports", "Liquid", 35),
    ("Biomass imports", "Solid", 35),
    ("Coal imports", "Coal", 11.606),
    ("Coal reserves", "Coal", 63.965),
    ("Coal", "Solid", 75.571),
    ("District heating", "Industry", 10.639),
    ("District heating", "Heating and cooling - commercial", 22.505),
    ("District heating", "Heating and cooling - homes", 46.184),
    ("Electricity grid", "Over generation / exports", 104.453),
    ("Electricity grid", "Heating and cooling - homes", 113.726),
    ("Electricity grid", "H2 conversion", 27.14),
    ("Electricity grid", "Industry", 342.165),
    ("Electricity grid", "Road transport", 37.797),
    ("Electricity grid", "Agriculture", 4.412),
    ("Electricity grid", "Heating and cooling - commercial", 40.858),
    ("Electricity grid", "Losses", 56.691),
    ("Electricity grid", "Rail transport", 7.863),
    ("Electricity grid", "Lighting & appliances - commercial", 90.008),
    ("Electricity grid", "Lighting & appliances - homes", 93.494),
    ("Gas imports", "Ngas", 40.719),
    ("Gas reserves", "Ngas", 82.233),
    ("Gas", "Heating and cooling - commercial", 0.129),
    ("Gas", "Losses", 1.401),
    ("Gas", "Thermal generation", 151.891),
    ("Gas", "Agriculture", 2.096),
    ("Gas", "Industry", 48.58),
    ("Geothermal", "Electricity grid", 7.013),
    ("H2 conversion", "H2", 20.897),
    ("H2 conversion", "Losses", 6.242),
    ("H2", "Road transport", 20.897),
    ("Hydro", "Electricity grid", 6.995),
    ("Liquid", "Industry", 121.066),
    ("Liquid", "International shipping", 128.69),
    ("Liquid", "Road transport", 135.835),
    ("Liquid", "Domestic aviation", 14.458),
    ("Liquid", "International aviation", 206.267),
    ("Liquid", "Agriculture", 3.64),
    ("Liquid", "National navigation", 33.218),
    ("Liquid", "Rail transport", 4.413),
    ("Marine algae", "Bio-conversion", 4.375),
    ("Ngas", "Gas", 122.952),
    ("Nuclear", "Thermal generation", 839.978),
    ("Oil imports", "Oil", 504.287),
    ("Oil reserves", "Oil", 107.703),
    ("Oil", "Liquid", 611.99),
    ("Other waste", "Solid", 56.587),
    ("Other waste", "Bio-conversion", 77.81),
    ("Pumped heat", "Heating and cooling - homes", 193.026),
    ("Pumped heat", "Heating and cooling - commercial", 70.672),
    ("Solar PV", "Electricity grid", 59.901),
    ("Solar Thermal", "Heating and cooling - homes", 19.263),
    ("Solar", "Solar Thermal", 19.263),
    ("Solar", "Solar PV", 59.901),
    ("Solid", "Agriculture", 0.882),
    ("Solid", "Thermal generation", 400.12),
    ("Solid", "Industry", 46.477),
    ("Thermal generation", "Electricity grid", 525.531),
    ("Thermal generation", "Losses", 787.129),
    ("Thermal generation", "District heating", 79.329),
    ("Tidal", "Electricity grid", 9.452),
    ("UK land based bioenergy", "Bio-conversion", 182.01),
    ("Wave", "Electricity grid", 19.013),
    ("Wind", "Electricity grid", 289.366)
  ),
  layout: layout.auto-linear(
    node-gap: 0.5,
    layer-gap: 1.8,
    min-node-height: 0,
    radius: 0,
    layers: (
      "0": ("Nuclear", "Gas imports", "Gas reserves", "Agricultural 'waste'", "UK land based
bioenergy", "Marine algae", "Geothermal", "Other waste", "Solar", "Hydro", "Tidal", "Biomass
imports", "Wave", "Coal imports", "Wind", "Coal reserves", "Pumped heat", "Oil imports", "Oil
reserves", "Biofuel imports"),
      "1": ("Ngas", "Bio-conversion", "Solar PV", "Solar Thermal", "Coal", "Oil"),
      "2": ("Gas", "Liquid", "Solid"),
      "3": "Thermal generation",
      "4": ("District heating", "Electricity grid"),
      "5": "H2 conversion",
      "6": "H2",
      "7": ("Losses", "Over generation / exports", "Heating and cooling - homes", "Heating and
cooling - commercial", "Lighting & appliances - homes", "Lighting & appliances - commercial",
"Industry", "Road transport", "Rail transport", "International aviation", "Domestic aviation",
"International shipping", "National navigation", "Agriculture")
    )
  ),
  tinter: tinter.layer-tinter(palette: tinter.palette.catppuccin),
  ribbon-stylizer: ribbon-stylizer.gradient-from-to(),
  draw-label: label.default-linear-label-drawer(
    width-limit: false,
    draw-content: (properties, ..) => [
      #text(properties.name, size: 0.3cm) \
    ]
  ),
)
```



9 Internal Types

Here are the definitions and examples of some internal types used in the `typst-ribbony` library.

9.1 Edge

Internally, an edge is represented as a dictionary with the following keys:

- `from`: The id of the source node.
- `to`: The id of the target node.
- `size`: The size of the edge. If it is an undirected edge, it will be the a array of two numbers, representing the size in both directions.
- `..other-attrs`: Any other custom attributes defined in the adjacency list's `edge-attrs` dictionary will also be included here. `style` is a special attribute that will be used for edge style override in drawing process.

Definition:

```
edge = dict<
  "from": string,
  "to": string,
  "size": (number | array<number>),
  ..other-attrs: dict
>
```

Example:

A typical edge:

```
(
  from: "A", to: "B", size: 10,
)
```

An edge with custom style and attributes:

```
(
  from: "A",
  to: "B",
  size: 10,
  style: ("stroke": red + 1pt),
  highlighted: true,
)
```

An undirected edge (e.g. used in circular layout with `directed: false`):

```
(
  from: "A",
  to: "B",
  size: (5, 15), // 5 from A to B, 15 from B to A
)
```

10 Changelog

v0.1.1

- Added `"io-auto"` option for `snap` parameter in `label-drawer.default-linear-label-drawer` as the new default, which automatically chooses the best snap position based on the node's incoming and outgoing edges.
- Added the `#void` node as a "hidden node" that can be used to override/customize the sizes of nodes.

v0.1.0

- First version.