DisCoPy as a general, programmatic diagramming tool

DisCoPy is a Python toolkit for building diagrams using principles from the mathematics of *category theory*.

Comforting words: You do not need to understand category theory to use DisCoPy in the basic way that I am about to describe.

The purpose of this document is to demonstrate how it can used and explain why it might be of interest to technical authors specifically.

For detailed background and more sophisticated examples refer to the official docs.

Types and boxes

Many systems, processes and structures can be represented by combinations of arrows and boxes.

In DisCoPy, things or objects are called `Types` and these can be inputs and outputs for `Boxes`.

A box can be thought of as an event, process or function.

Before making any diagrams we need import these classes.

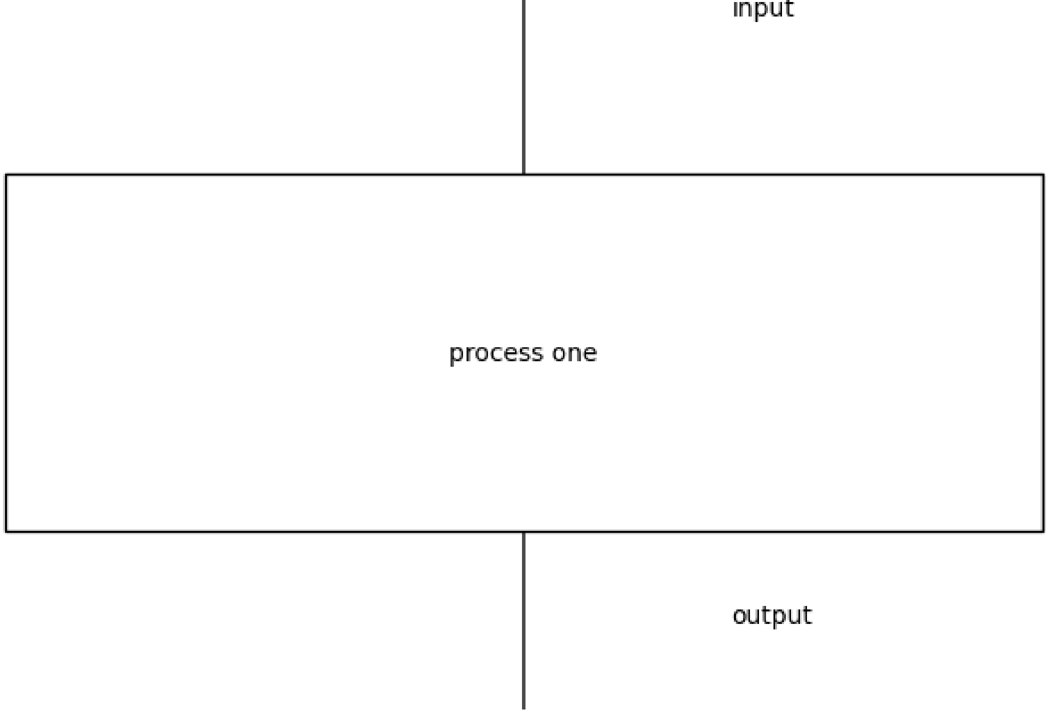
In [ ]:
```
from discopy.symmetric import Ty, Box
```

Input and output

A simple process consists of a single step that takes an input and returns an output.

Here the `input` and `output` are instantiated as types using `Ty()`. A process called "processOne" is defined with `Box()` that takes the input and returns the output. This is stored in a variable that is passed to `draw()` to render the diagram.

In [ ]:
```
input, output = Ty('input'), Ty('output')

processOne = Box('process one', input, output)

processOne.draw()
```



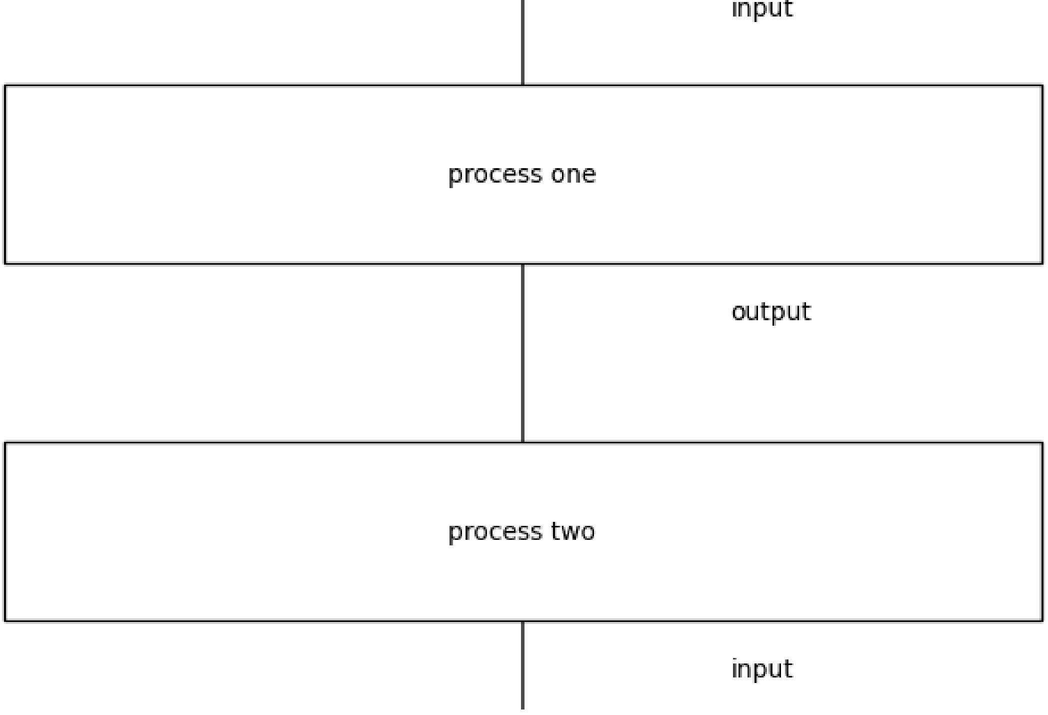
Composing Diagrams

A key aspect of DisCoPy is that a given diagram can be thought of as a module. It can be composed together with others to create more complex diagrams, as long as the composition is logical.

If a second process `processTwo` is defined it can be connected to the first in series using the `>>` operator. This new process can be stored in a new variable `combinedProcess` and drawn.

The resultant process takes the output from the first step and transforms it back to the input: not so useful but good enough for now!

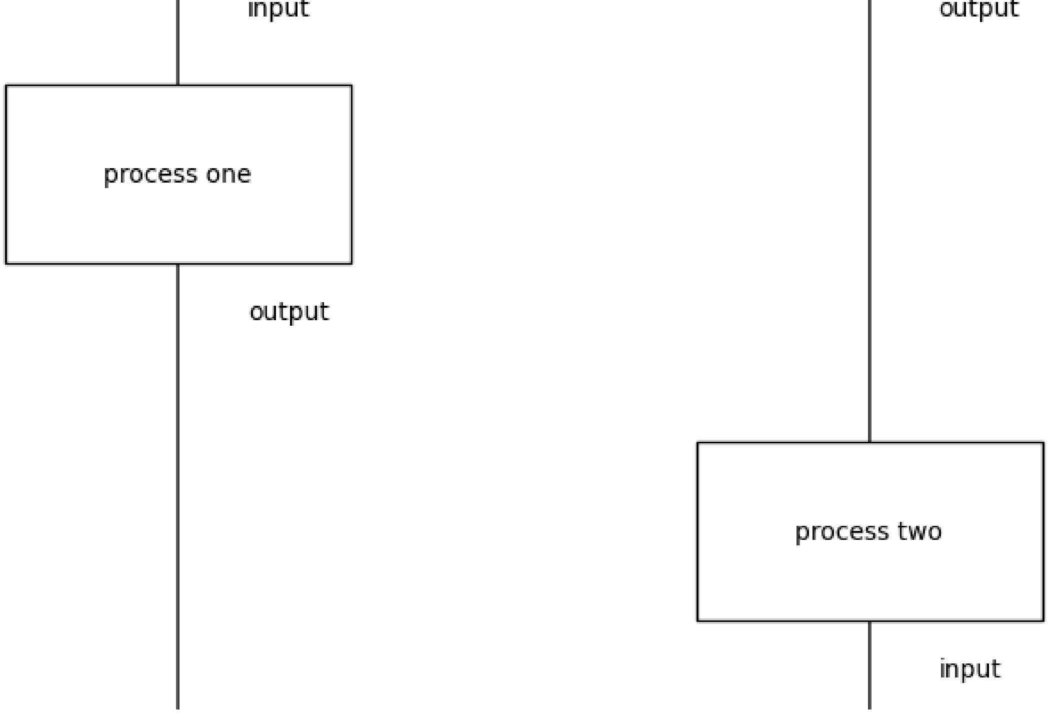
In [ ]:
```
processTwo = Box('process two', output, input)

combinedProcess = processOne >> processTwo

combinedProcess.draw()
```



Another operator — `@` — enables us to represent types and boxes in parallel.

Two separate processes, $processOne : input \to output$ and $processTwo : output \to input$, can be represented as parallel processes using the code below:

```
In [ ]: parallelProcesses = processOne @ processTwo

        parallelProcesses.draw()
```



Math alert: If you want to (you don't have to) it is possible to describe any of these diagrams as equations. For example:

1. $processOne : input \to output$
2. $processTwo : output \to input$
3. $processOne \cdot processTwo : input \to input$
4. $processOne \otimes processTwo : input \otimes output \to output \otimes input$

Proof and Logic

After creating these diagrams it is possible to leverage some of Python's built-in functionality.

For example, we can do an `assert()` check to confirm that a diagram is what we think it is:

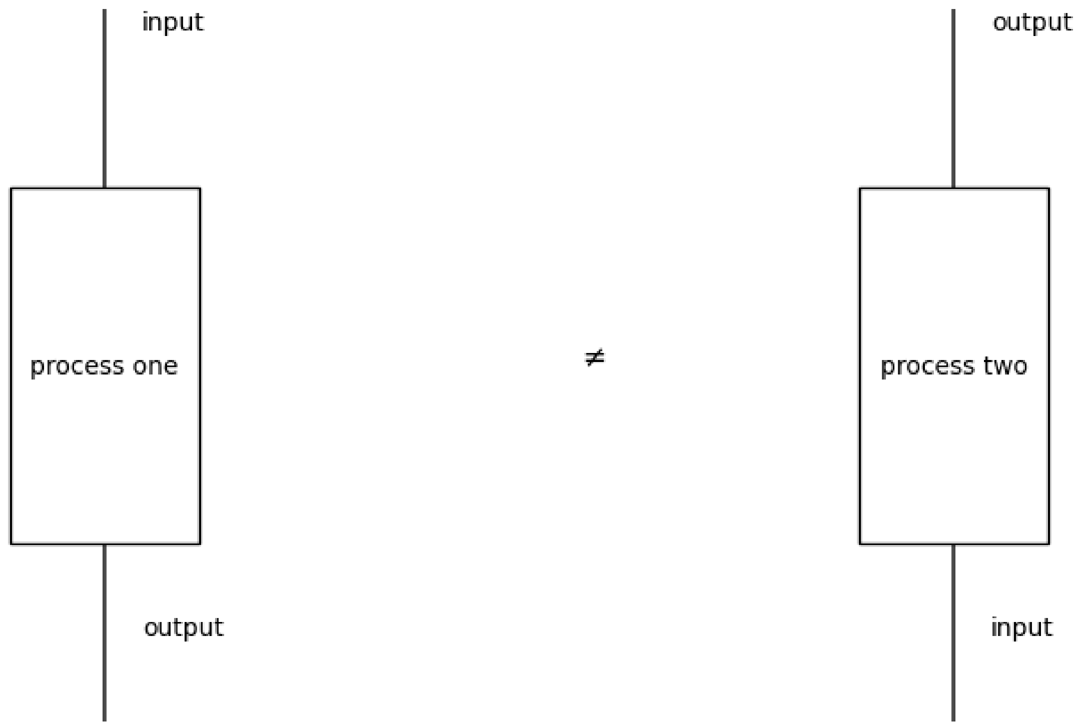
```
In [ ]: assert(parallelProcesses == processOne @ processTwo) # no error
```

```
In [ ]: assert(parallelProcesses == processOne >> processTwo) # error
```

```
AssertionError                            Traceback (most recent call las
t)
Cell In[284], line 1
----> 1 assert(parallelProcesses == processOne >> processTwo) # error

AssertionError:
```

To represent a logical (or mathematical) relation between diagrams we can import and use `Equation()` .

In [ ]:
```python
from discopy.drawing import Equation

Equation(processOne, processTwo, symbol='$\\neq$').draw()
```



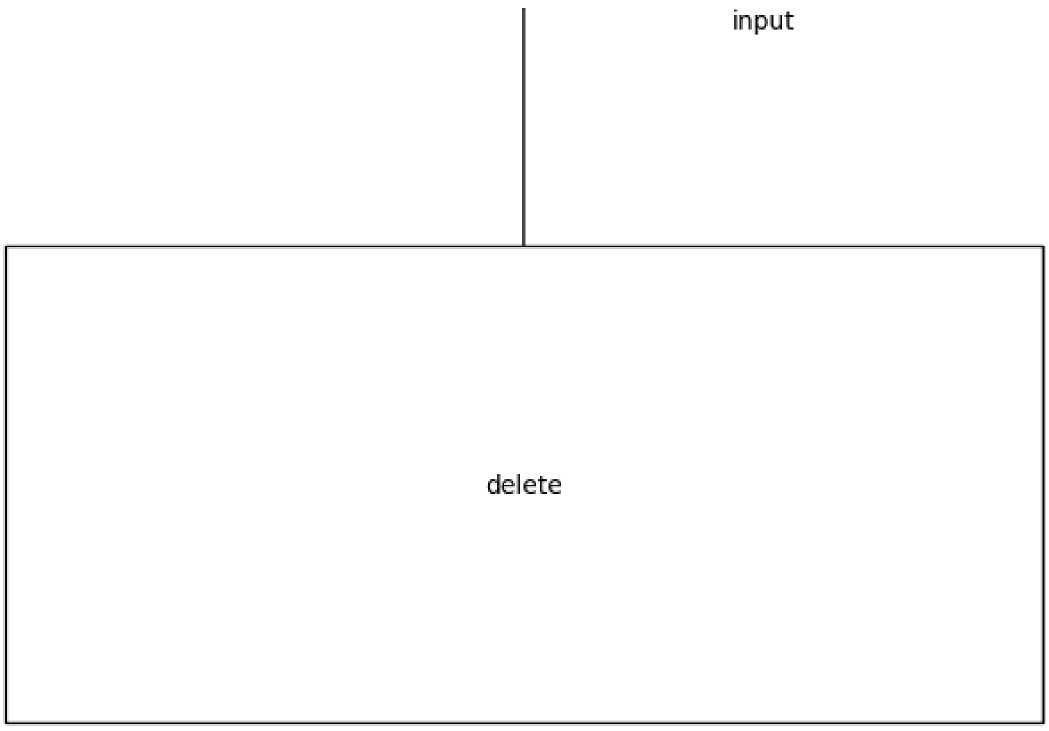
Abstract Processes

A process might be recognised as a *kind* of some more abstract process. For example, a *sink* is something that takes an input and returns no output. The concept of *delete* can be conceived as a kind of sink. Similarly, there is the general concept to `duplicate` an object, the computational concept to `copy` a file and the physical concept to `split` some material.

Below we define boxes for `duplicate` and `sink` using lambda functions then use them to define analogous processes with different names.

thought of as representing nothing or null.

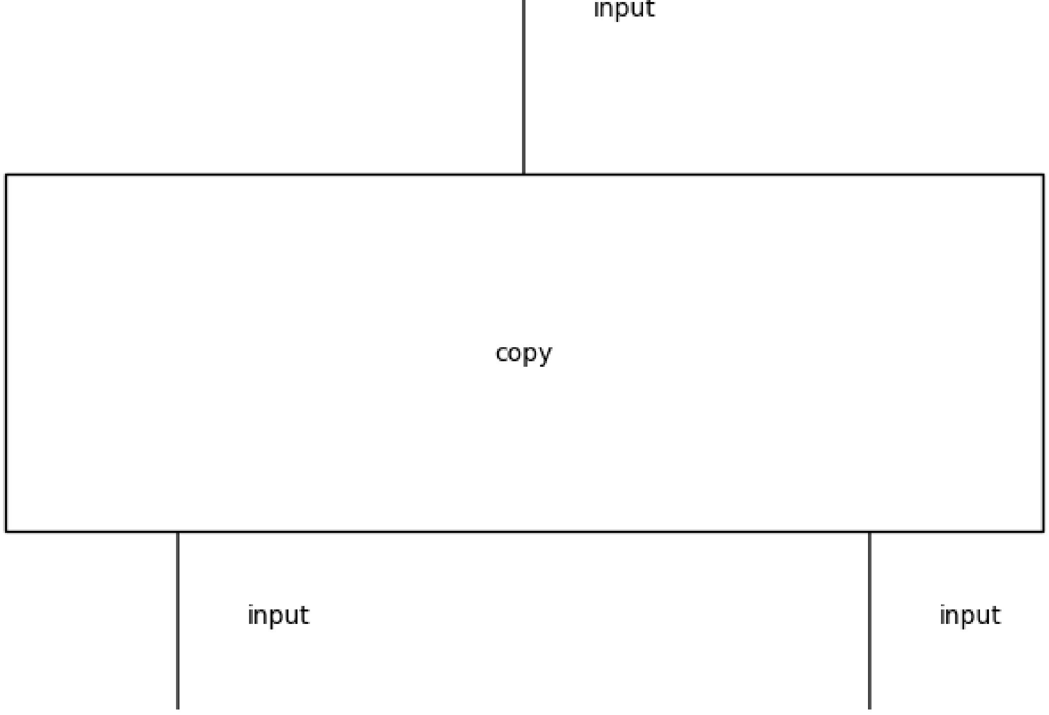
```
In [ ]: sink = lambda name, thing: Box(name, thing, Ty())

        delete = sink('delete', input)

        delete.draw()
```



For `duplicate` the strategy is the same.

The box takes a thing as input and returns two versions of the same thing. Note: previously the `@` operator was used on boxes themselves but here it is used on the types.

```
In [ ]: duplicate = lambda name, thing: Box(name, thing, thing @ thing)

        copy = duplicate('copy', input)
        split = duplicate('split', input)

        copy.draw()
```

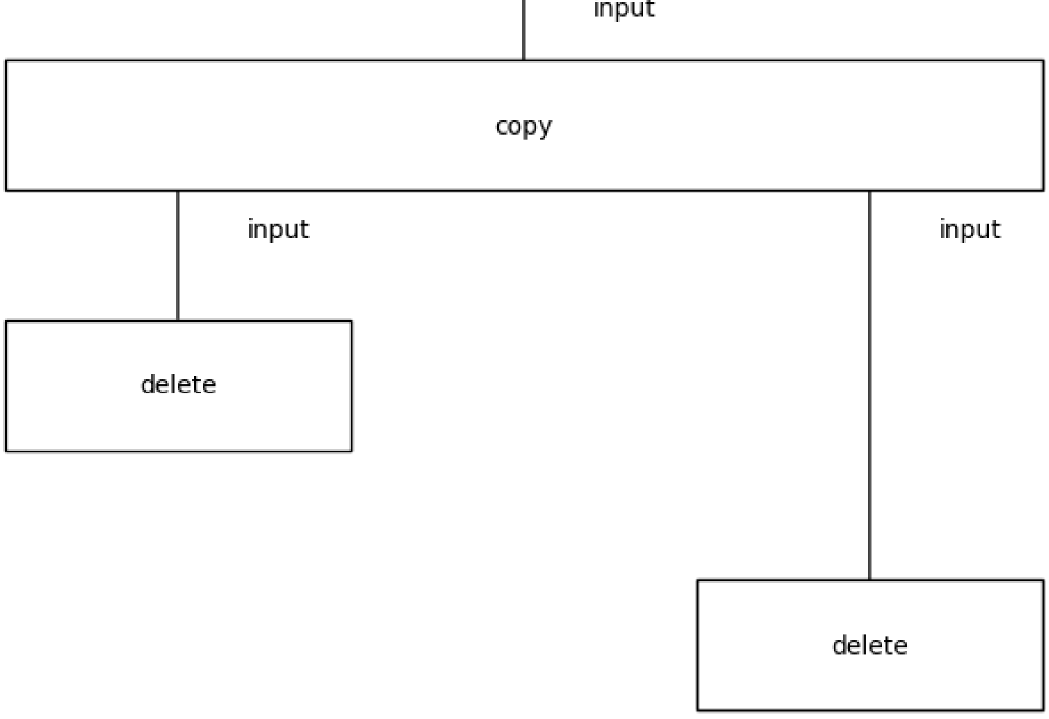


How do we represent the useless operation of copying something then deleting both copies?

```
cp myNewPoem.txt myPoemCopyForFriends.txt # optimism
    rm myNewPoem.txt myPoemCopyForFriends.txt # fear
```

We compose what we have designed already.

```
In [ ]: (copy >> delete @ delete).draw()
```



This opens the possibility of crafting a well-defined and reusable set of objects and processes for representing our domain of interest.

Granularity and Analogy

I mentioned earlier that DisCoPy is based on category theory but then skillfully avoided the subject. Category theory is all about abstraction. It has interested computer scientists for this very reason. Programmers are practitioners of abstraction and some category theorists have suggested that category theory could be a very powerful tool for representing complex software and reasoning about its design.

In some ways category theory is simple:

> Everything is either a box or an arrow

However, it gets pretty mind-bending.

One of the first topics that confused people is the `functor`. This is often explained intuitively as a way to "zoom in" or "zoom out". Interestingly, it is also suggested as a way to find **analogies** between systems. For simplicity, I want to focus on the functor-as-zooming idea.

To make this obvious I'm going to do a little obfuscation and give `Functor` the alias `Zoom`. I don't want your mind to wander on the meaning of `Functor` — I want you just think about it as a "zoom-controller" for now.

example.

In [ ]:
```
from discopy.symmetric import Functor as Zoom
from discopy.symmetric import Id
```

Now I'm going to define two systems:

- High-level
- Low-level

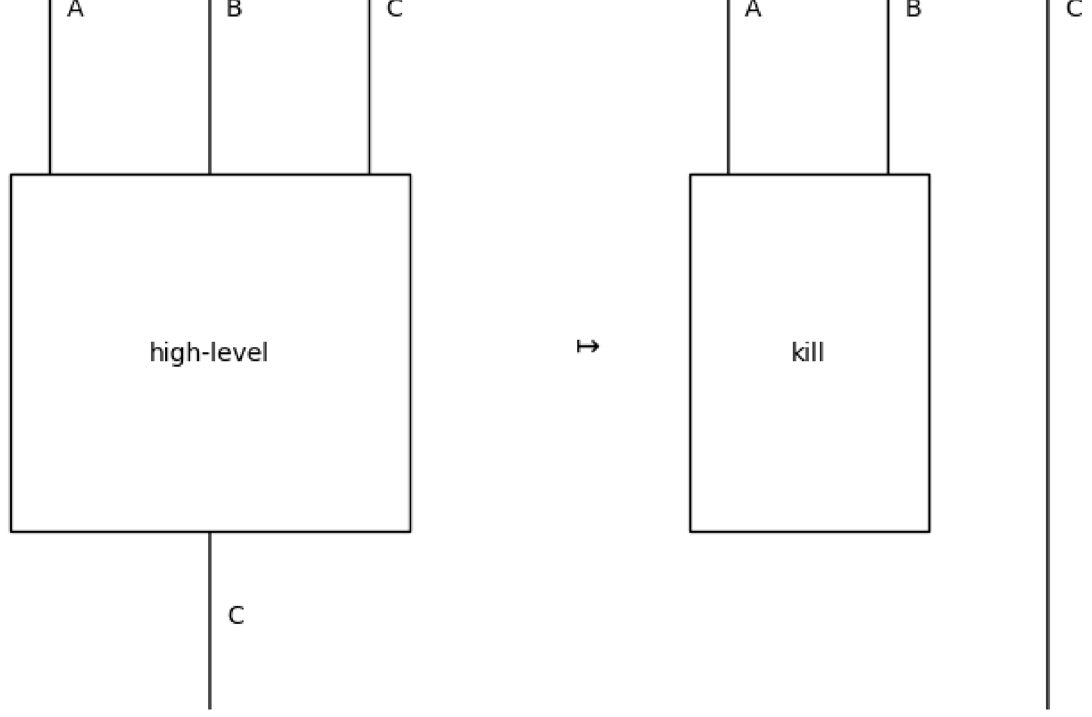
Then we are going to use `Zoom` to move between the higher and lower levels.

Note on the C4 model: If you are familiar with the C4 framework for defining architectural diagrams this idea might seem familiar. I show one case of zooming-in here, but multiple zooms could be defined to establish a hierarchy of diagrams. If you know C4 think of the following example as going from a **system** to a **container** representation. Remember, however, that there is nothing stopping use creating **component** and code levels if we want. At some point if we wanted another level in this hierarchy it would be up to us to define it.

In [ ]:
```
# define the things to be processed
inA, inB, inC = Ty('A'), Ty('B'), Ty('C')

# define the high level process
high = Box('high-level', inA @ inB @ inC, inC) # three go in, one goes ou

# define the low level implementation
killAB = sink('kill', inA @ inB) # sub-process: two get killed and don't
leaveC = Id(inC) # sub-process: one passes right through

# parrallelize the implementation
low =  killAB @ leaveC

# define the mapping from one level to the next
# ob is for changing the names of types when we zoom (we leave them the s
# ar is for changing the boxes (we are changing from a high-level to a lo
zoomToLowerLevel = Zoom(
    ob=lambda x: x,
    ar={high: low}
)
```

On the left we will see the high-level view and on the right the low-level view. In one sense they are equal (the same transformation is achieved) but one provides more context.

In [ ]:
```
Equation(high, zoomToLowerLevel(high), symbol='$\\mapsto$').draw()
```



We have defined two diagrams, each representing a different level of abstraction.

We have also defined a mapping between the two levels.

What is a functor?

We can define systems using arrows and boxes. These systems can be concrete or conceptual:

- Concrete: chemical reactions, software interfaces, manufacturing processes
- Conceptual: mathematical functions, power relations, analogies, paradigms

We can also draw arrows **between different systems**. Think about these examples:

- An arrow that defines the translation between a sentence in English and a sentence in French
- An arrow that defines the analogy between a recipe and an algorithm, or between a recipe and a how-to
- An arrow that defines the relationship between a software interface and its implementation
- An arrow that defines the transition between a popular description and a technical description
- An arrow that defines the analogy between documentation and coding workflows (docs-as-code)

teach us something about our mental model.

Conclusion

The good

- Diagrams can be designed and rendered in Python, a familiar language that can be used in automated build tasks
- No 3rd party tooling or infrastructure is needed other than a Python interpreter
- Can accomodate a particular model (e.g., C4) and domain (e.g., software) while not being restricted to them
- Concepts are based on the rigid yet massively generalisable framework of category theory, which will outlive us and our software
- Text, code and diagrams can be combined in notebooks (`.ipynb`) that have native support on GitHub and Sphinx

The bad

- Requires some willingess to use Python and experience the friction of coding a diagram
- DisCoPy is not particularly targeted at product documentation, so documentation-specific features may not be developed
- Options for styling diagrams are potentially quiet limited

Resources

- DisCoPy docs: the QNLP Tutorial in particular has some wonderful cooking examples.
- CatLab docs: there is a similar implementation written for the Julia language that has some additional features.
- Mascarpone blog: a lovely blog that talks through (another) cooking example using category theory
- An Invitation to Category Theory: book by category theorist David Spivak that is the best introduction I found on the subject and includes some interesting examples ranging from chemistry to software
- widip: an interactive environment built using DisCoPy that generates diagrams from `.yaml` files