1 Introduction

Tape diagrams are a diagrammatic language for rig categories with finite biproducts. These can be intuitively thought as *string diagrams within string diagrams*. These have been used to provide an axiomatization of the positive fragment of Tarski's Calculus of Relations, and can be extended with notions such as traces to provide a *diagrammatic algebra for program logics*. In this document, we present a tool that allows the description, manipulation and automated depiction of tape diagrams.

2 The tool

The tool consists of a *description language*, which allows the user to define and manipulate terms, a *type-checking module*, which checks whether the given terms are well-typed, a *drawing module*, which produces a LaTeX (TikZ) picture of the diagrams, and a *rewriting module*, which does not exist yet. The tool is written in OCaml, using the Menhir library for generating the parser of the language.

2.1 The Language

A program in the language we provide consists of a series of declarations and commands. The former are used to define sorts, terms of a sesquistrict rig signature and tape diagrams, whereas the latter can be used to act on them, e.g. to typecheck, draw, or otherwise manipulate the terms and diagrams. The following is the BNF representation of a program p:

$$
\begin{aligned}
p &::= com \mid d \mid p\,.\,p \\
d &::= \texttt{let}\ v : type = e \mid \texttt{let}\ v : \texttt{sort} \\
com &::= \texttt{check}\ e \mid \texttt{draw}\ e\ \text{to}\ path \mid \text{Add other commands here} \\
e &::= v \mid SSR \mid TD \\
type &::= \texttt{tape} \mid \texttt{term}
\end{aligned}
$$

Where v is a variable name, SSR is a term of a sesquistrict rig signature and TD is a tape diagram. In the language:

- The $\otimes$ and $\oplus$ symbols are notated as `*` and `+`.
- The symmetries $\sigma^{\otimes}_{\bullet,\bullet}$ and $\sigma^{\oplus}_{\bullet,\bullet}$ are notated as `s*(•,•)` and `s+(•,•)`.
- The left distributor $\delta^{l}_{\bullet,\bullet,\bullet}$ is notated as `dl(•,•,•)`.
- The named generators are written as:

$$\texttt{gen}(\texttt{name}, \texttt{arity}, \texttt{coarity})$$

- A circuit enclosed in a tape, i.e. $\overline{c}$, is notated as `[c]`

As an example, the following is a valid program:

Listing 1: Example program

```
let A : sort.   let B : sort.   let C : sort.
let t : tape = id(A + B) ; s+(A, B).

check t.                          // true
check id(A) ; id(B).              // false
draw t to "./figure1.txt".
draw s*(A + B, C) to "./figure2.txt"
```

```
Typecheck result:        true ✅
Typecheck result:        false ❌
Drawing saved at path:   './figure1.txt'
Drawing saved at path:   './figure2.txt'
```

Figure 1: Result of the program in Listing 1.

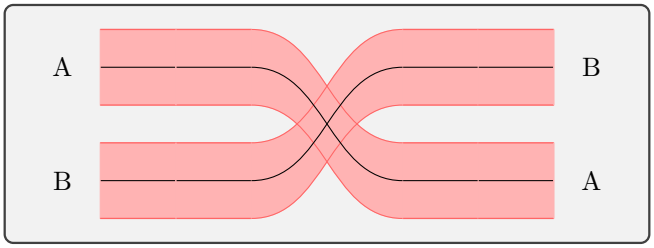


Figure 2: Image encoded in figure1.txt, written by the program in Listing 1.

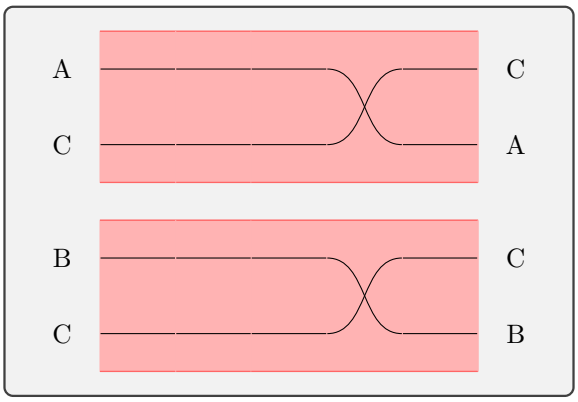


Figure 3: Image encoded in figure2.txt, written by the program in Listing 1.

2.2 Drawing Tape Diagrams

2.2.1 Introduction to the approach

The approach we used to draw the diagrams was to essentially *"compile"* the terms down to a string of LaTeX macros. As an example, this is a very simple macro that can be used to depict an identity circuit:

Listing 2: LaTeX macro for rendering identities.

```
% fresh posx posy
\newcommand{\id}[3]{
  \node [nodestyle] (ida#1) at (#2,#3) {};
  \node [nodestyle] (idb#1) at (#2 + 1,#3) {};
  \draw (ida#1) -- (idb#1);
}
```

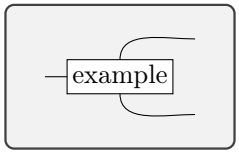
When the tool is asked to render the circuit id_S, it will simply generate the code:

```
\id {0} {0} {1}
```

Where "1" is a fresh name, uniquely generated every time the identity-drawing routine is called. The result will be the following:

This simple idea can be extended to much more complex diagrams; an example of this is the macro for rendering generators (Listing 3), which can be used to generate arbitrary generators:

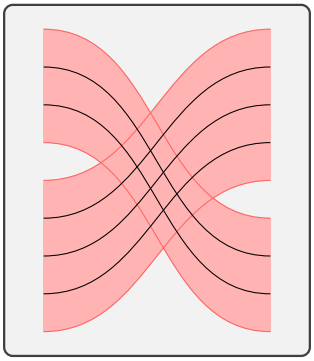
```
% fresh posx posy arity-1 coarity-1 name otimesdist
\gen {1} {0.0} {0.0} {1} {2} {example} {0.5}
```



A further example of a complex, parametric generator is the macro for drawing arbitrary tape-swaps. A generator of the form $\sigma^{\oplus}_{\bullet,\bullet}$ is compiled to the macro:

```
% posx posy n1 n2 oplusdist otimesdist tapepadding width
\swaptape 0 0 2 3 0.5 0.5 0.5 2
```

Which renders the following:



These basic *building blocks* can then be combined and connected by the renderer to produce even very complex tape diagrams. In the following, we provide an overview of the whole drawing process.

2.2.2 The anatomy of a tape diagram

In order to discuss how the basic building blocks can be composed, we first need to discuss what their fundamental properties are. Among these are their *position*, *height*, *length*, and their *interfaces*. Since the first three are self-explanatory, we shall focus on the fourth one. The left and right interfaces of a tape are an extension of their arity and coarity, which include information regarding the positions of the "*circuit pins*" and the bounds of the tapes. These are defined as follows:

```
type circuit_draw_interface =
  | EmptyCircuit
  | CircuitTens of circuit_draw_interface * circuit_draw_interface
  | CircuitPin of float * float

type tape_draw_interface =
  | EmptyTape of (float * float) * (float * float)
  | TapeTens of tape_draw_interface * tape_draw_interface
  | TapeInterface of (float * float) * (float * float) * circuit_draw_interface
```

In particular, a circuit interface is essentially a list of pairs of floats, whereas a tape interface is a list of circuit interfaces together with the positions of the bounds of the tape. Note that, unlike an empty circuit, an empty tape still carries information regarding its bounds (which will not enclose a circuit).

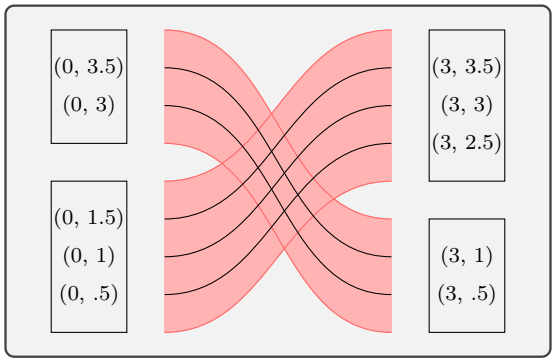


Figure 4: A generator, together with its interfaces.

In addition to these generator-specific properties, there are also some **gloabal properties** that need to be specified. Among these are the distance between two "lanes" of a circuit, denoted as $d_{\otimes}$, the distance between two vertically stacked tapes, denoted as $d_{\oplus}$, and the distance the edge of a tape and the first circuit, which we will call *padding distance* or d_p.

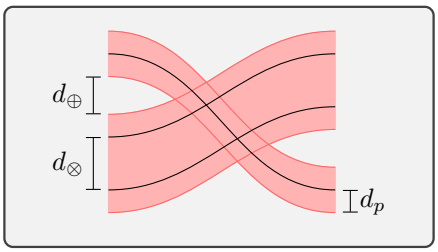


Figure 5: The global properties $d_{\otimes}$, $d_{\oplus}$ and d_p.

We are now ready to discuss how these properties can be used to combine the generators and obtain arbitrary tape diagrams.

2.2.3 Combining circuit generators

There are two ways to combine the circuit generators (identity, symmetry and named generators): tensor composition and sequential composition.

Tensor Composition The vertical composition is fairly straight-forward. We simply need to vertically stack the two diagrams, with a gap of length $d_{\otimes}$ between them:

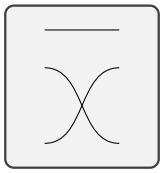


Figure 6: Simple example of vertical composition of circtuits.

There are some cases in which this cannot be done in the straight-forward manner: those in which the two diagrams have different lengths. In those cases we need to perform an *interface alignment*, in which the right interface of the shorter circuit is extended in such a way to be aligned with the right interface of the longer circuit:

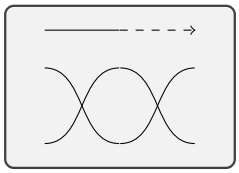


Figure 7: Vertical composition of circuits of different length.

Sequential Composition Sequential composition of circuits is simpler: we just need to horizontally stack the two circuits, making sure to place them in such a way that the interfaces match. By design of the constructors, this amounts to aligning the centers of the two circuits.

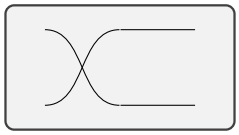


Figure 8: Sequential composition of circuits.

2.2.4 Combining tape generators

There are three ways to combine the generators to obtain non-elementary diagrams. These are tensor composition, sequential composition and the embedding of a circuit within a tape.

Tensor composition Analogously to the case of circuits, we simply need to vertically stack the two tape diagrams, with a gap of length $d_{\oplus}$ between them.

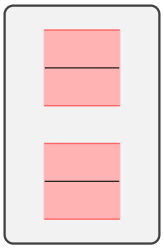


Figure 9: Simple example of vertical composition of tapes.

Once again, when two diagrams of different lengths are summed, we need to perform an *interface alignment*, in which the right interface of the shorter tape is extended in such a way to be aligned with the right interface of the longer tape.

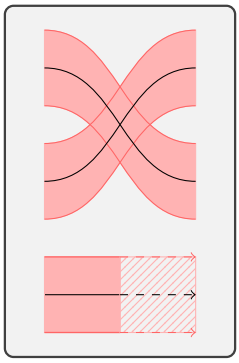


Figure 10: Vertical composition of tapes with interface alignment.

Sequential composition In a similar vein, when sequentially aligning two tape diagrams, we want to stack the two pictures horizontally. It might happen that two diagrams we want to horizontally compose have different heights; in these cases, we must add an *adapter* between their interfaces.

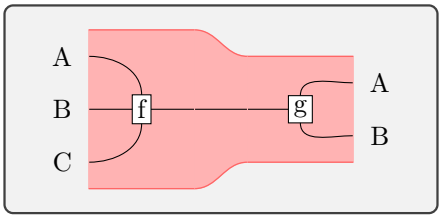


Figure 11: Sequential composition of tapes with adapter.

Embedding of a circuit within a tape Embedding a circuit c within a tape is also fairly straightforward, as we simply need to build a tape of height $h_c + 2d_p$, where h_c is the height of the circuit, around the circuit itself. Note that, by definition of the `TapeInterface` constructor, the interfaces of this new tape will be of the form:

$$\texttt{TapeInterface}(\texttt{tape\_bot\_l}, \texttt{tape\_top\_l}, \texttt{left\_interface}(\texttt{c}))$$

$$\texttt{TapeInterface}(\texttt{tape\_bot\_r}, \texttt{tape\_top\_r}, \texttt{right\_interface}(\texttt{c}))$$

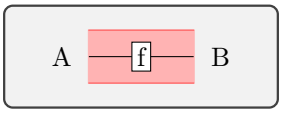


Figure 12: Generator embedded within a tape.

2.2.5 Examples of complex circuits

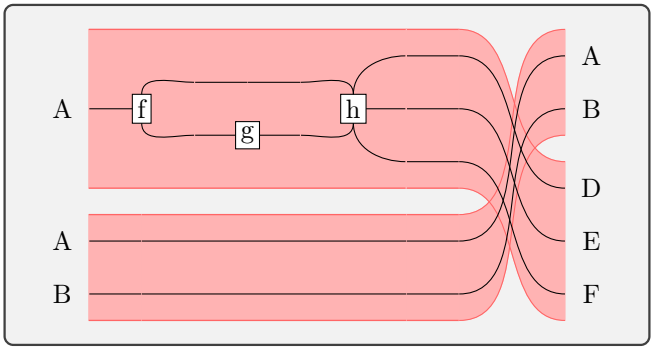


Figure 13: $(\overline{\texttt{gen(f, A, AB)}}; (\texttt{id(A)} \otimes \texttt{gen(g, B, C)}); \overline{\texttt{gen(h, AC, DEF)}} \oplus \texttt{id(A} \otimes \texttt{B)}); \sigma^{\oplus}_{\texttt{DEF,AB}}$

A LaTeX Macros

Listing 3: LaTeX macro for rendering generators.

```
% fresh posx posy arity-1 coarity-1 name otimesdist
\newcommand{\gen}[7]{
  \pgfmathsetmacro\arminone{#4};
  \pgfmathsetmacro\coarminone{#5};
  \pgfmathsetmacro\otimesdist{#7};
  \pgfmathsetmacro\arity{#4 + 1};
  \pgfmathsetmacro\coarity{#5 + 1};

  % compute height of the generator
  \pgfmathparse{
    (\arity>\coarity)
      ? \arminone * \otimesdist
      : \coarminone * \otimesdist
  }
  \let\height\pgfmathresult

  % Draw generator body
  \node [boxstyle] (a) at (#2 + 1,#3 + \height / 2) {#6};

  \pgfmathparse{
    (\arity - 1) / 2 * \otimesdist
  }
  \let\arshift\pgfmathresult

  \pgfmathparse{
    (\coarity - 1) / 2 * \otimesdist
  }
  \let\coarshift\pgfmathresult

   % Draw left interface of the generator
   \foreach \i in {0,...,\arminone} {
    \node [nodestyle] (#1in\i) at (#2, #3 + \i * \otimesdist + \height / 2 -
        \arshift) {};
    \ifnum\arminone=0
      \def\angle{180}
    \else
      \pgfmathsetmacro{\angle}{(180 / \arminone) * -\i - 90}
    \fi
    \draw[in=0, out=\angle] (a) to (#1in\i);
  }

  % Draw right interface of the generator
  \foreach \i in {0,...,\coarminone} {
    \node [nodestyle] (#1out\i) at (#2 + 2, #3 + \i * \otimesdist + \height / 2 -
        \coarshift) {};
    \ifnum\coarminone=0
      \def\angle{0}
    \else
      \pgfmathsetmacro{\angle}{(180 / \coarminone) * \i - 90}
    \fi

    \draw[in=180, out=\angle] (a) to (#1out\i);
  }
}
```