

# Rewriting for Traced Monoidal Closed Categories

Alessandro Di Giorgio<sup>1</sup>[0000–0002–6428–6461], Dan R.  
Ghica<sup>2,3</sup>[0000–0002–4003–8893], and Fabio Zanasi<sup>4</sup>[0000–0001–6457–1345]

<sup>1</sup> Tallinn University of Technology, Estonia  
`alessd@taltech.ee`

<sup>2</sup> University of Birmingham, United Kingdom

<sup>3</sup> Huawei Central Software Institute, United Kingdom  
`d.r.ghica@bham.ac.uk`

<sup>4</sup> University College London, United Kingdom  
`f.zanasi@ucl.ac.uk`

**Abstract.** Traced monoidal closed categories are a model for higher-order functional computation. We develop a formal language of string diagrams for these categories, and a faithful interpretation in terms of certain hypergraphs. We then use the interpretation to show that string diagram rewriting can be implemented as double-pushout rewriting in a sound and complete way. Finally, we showcase our approach on the  $\lambda$ -calculus with explicit recursion.

**Keywords:** String diagrams · Hypergraphs · Rewriting

## 1 Introduction

String diagrams are a graphical formal language for reasoning in symmetric monoidal categories, see eg. [30, 31] for a survey. In recent years, they have been increasingly adopted as a framework to reason about a variety of systems, including quantum processes [11], dynamical systems [9, 4], digital circuits [17], relational databases [8], automata [29] and neural networks [13], amongst others. Compared to traditional syntax, string diagrams highlight resource-exchange between components, allowing to reason visually about subtle interactions such as quantum entanglement, concurrency, probabilistic dependency, and variable sharing. At the same time, string diagrams are a fully formal syntax, which may be reasoned about with the same tools as programming languages.

In a nutshell, string diagrams can be thought of as a circuit language, represented with boxes and wires, whose building blocks may be composed in sequence and in parallel to form more complex diagrams. In this setting, the laws of symmetric monoidal categories, rendered in Figure 1, correspond to geometric manipulations, e.g. pulling of wires or sliding boxes.

Because of their compositional nature, string diagrams allow for a natural notion of rewriting consisting in replacing a sub-diagram with another. However, differently than traditional term rewriting, such replacement may not always be immediately apparent. For example, consider the rule on the left and the first string diagram on the right.

$$\begin{array}{c} \boxed{c} \\ \boxed{d} \end{array} = \boxed{e} \qquad \begin{array}{c} \boxed{f} \\ \boxed{c} \end{array} \begin{array}{c} \text{---} \\ \text{---} \end{array} \begin{array}{c} \boxed{d} \end{array} = \begin{array}{c} \boxed{f} \\ \boxed{c} \end{array} \begin{array}{c} \boxed{c} \\ \boxed{d} \end{array} = \begin{array}{c} \boxed{f} \\ \boxed{e} \end{array}$$

Before applying the substitution we need to manipulate the diagram, using the laws of symmetric monoidal categories, in order to get an exact match. Therefore, before reaching a match we might need to potentially inspect all those string diagrams equivalent to a given one. This makes string diagram rewriting computationally impractical. A recent line of works [5–7] proposes a solution to this problem by interpreting string diagrams as certain hypergraphs and their rewriting as double-pushout rewriting with interfaces (DPOI) [6], an extension of the classical double pushout rewriting (DPO) [12, 14]. The advantage is that an equivalence class of string diagrams corresponds to exactly one hypergraph, making matching computationally more straightforward.

By now, this approach has been adapted to various kinds of categories and their string diagrams. Particularly relevant for our work are the hierarchical hypergraphs for monoidal closed categories [2] and the partial monogamous hypergraphs for traced monoidal categories [18].

The main contribution of our work is to develop a framework for string diagram rewriting in a more articulated setting: *traced monoidal closed categories*. This extends [2] with the handling of a trace structure, and [18] with the handling of a closed structure. The study of such extension is motivated by a computational understanding of the traced monoidal closed structure. First, traced monoidal categories, and notably traced cartesian categories, have been thoroughly studied as a model for recursion and iteration, starting with the work of Hasegawa [20]. Similarly, monoidal closed categories, and in particular cartesian closed categories, correspond to models of the simply typed  $\lambda$ -calculus [24]. Therefore, a combination of these structures provides a semantic framework to study computational processes featuring both recursion and higher-order functions. In particular, traced cartesian closed categories constitute models for the  $\lambda$ -calculus with explicit recursion [21]. The latter is essentially a version of the  $\lambda$ -calculus that features a **letrec** construct, typical of functional languages.

Our contribution will develop as follows. First, we will define a suitable notion of string diagram, accounting for both traces and the closed structure. Then we will introduce traced hierarchical hypergraphs, and show that they faithfully interpret string diagrams. Finally, we will be able to characterise string diagram rewriting in terms of double-pushout rewriting of these hypergraphs— with some subtleties due to the hierarchical nature of these structures. We will conclude by interpreting the  $\lambda$ -calculus with explicit recursion in our framework. This also allows us to discuss non-confluence of the calculus through the lenses of string diagram rewriting.

*Synopsis:* In §2 we introduce the string diagrams for traced monoidal closed categories. In §3 we recall the definition of hierarchical hypergraphs and their categorical structure. §4 contains our first contribution: we characterise exactly those hypergraphs that correspond to traced hierarchical string diagrams. As a second contribution, in §5 we develop a suitable notion of DPOI rewriting which adequately interprets rewriting for traced monoidal closed categories. Finally,

in §6 we consider a case study: the string diagrammatic interpretation of the  $\lambda$ -calculus with explicit recursion.

## 2 String Diagrams for Traced Monoidal Closed Categories

In this section we introduce string diagrams for traced monoidal closed categories. These can be thought of as a combination of the usual graphical language for traced categories [22, 31] and the hierarchical string diagrams for monoidal closed categories introduced in [18]. We first recall the notion of monoidal closed signature, adapted from [16].

**Definition 1.** *Given a set  $\mathcal{S}$  of generating objects, the elements of  $\text{obj}(\mathcal{S})$  are inductively generated as follows, where  $I$  is a designated object,  $A \in \mathcal{S}$ , and  $\otimes$  and  $\multimap$  are type forming operations:*

$$X, Y ::= I \mid A \mid X \otimes Y \mid X \multimap Y$$

A monoidal closed signature is a pair  $(\mathcal{S}, \Sigma)$  where  $\Sigma$  is a set of generating morphisms  $s: X \rightarrow Y$  with  $X, Y \in \text{obj}(\mathcal{S})$ .

A traced monoidal closed (tmc) category may be freely constructed starting from a monoidal closed signature, in the expected way. The morphisms of such category will be string diagrams accounting both for the trace and for the monoidal closed structure, which we now introduce.

**Definition 2.** *The traced monoidal closed  $(\mathcal{S}, \Sigma)$ -terms are generated by the following grammar, where  $s: X \rightarrow Y$  is an element of  $\Sigma$ , and we use the typical boxes-and-wires notation of string diagrams [30] to represent terms pictorially:*

$$c, d ::= [\ ] \mid x \multimap x \mid x \text{---} [s] \text{---} y \mid \begin{array}{c} x \\ \diagup \\ y \end{array} \begin{array}{c} y \\ \diagdown \\ x \end{array} \mid x \text{---} [c] \text{---} [d] \text{---} z \mid \begin{array}{c} x \\ \text{---} [c] \text{---} y \\ z \\ \text{---} [d] \text{---} w \end{array} \mid \begin{array}{c} x \multimap y \\ \text{---} [c] \text{---} y \end{array} \mid \begin{array}{c} x \multimap y \\ \text{---} [c] \text{---} z \end{array} \mid \begin{array}{c} x \multimap y \\ \text{---} [c] \text{---} y \end{array} \quad (1)$$

**Definition 3.** *Given a monoidal closed signature  $(\mathcal{S}, \Sigma)$ , the free traced monoidal closed category  $\mathbf{T}_\Sigma$  on  $(\mathcal{S}, \Sigma)$  is defined by letting objects be elements of  $\text{obj}(\mathcal{S})$  and morphisms the tmc  $(\mathcal{S}, \Sigma)$ -terms quotiented by the axioms in Figures 1, 2, 3.*

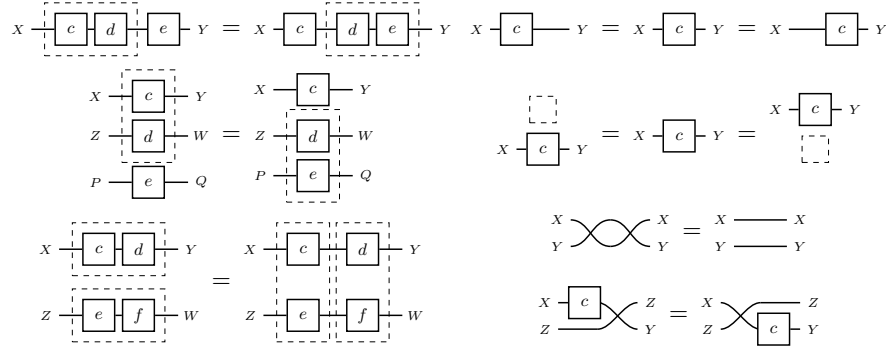
We write  $\mathbf{C}_\Sigma$  for the monoidal closed wide subcategory of  $\mathbf{T}_\Sigma$  with morphisms the  $(\mathcal{S}, \Sigma)$ -terms generated by (1) without the last production and quotiented only by the axioms in Figures 1 and 3.

*Remark 1.* We are tacitly assuming that wires labelled solely with  $X \otimes Y$  are always drawn as two separate wires stacked on top of each other. Thus, for example

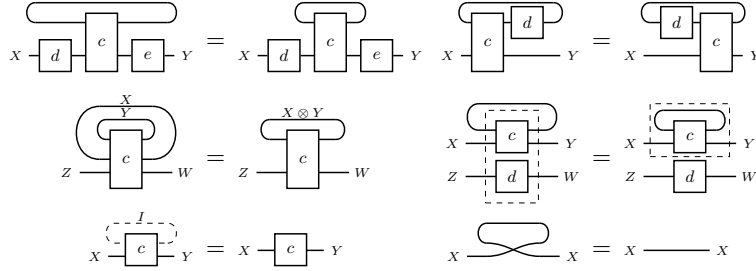
$$x \otimes y \multimap x \otimes y \text{ is } \begin{array}{c} x \\ \text{---} \\ y \end{array} \begin{array}{c} x \\ \text{---} \\ y \end{array}, \text{ and } \begin{array}{c} x \multimap (y \otimes z) \\ \text{---} \\ x \end{array} \begin{array}{c} y \otimes z \\ \text{---} \\ y \otimes z \end{array} \text{ is } \begin{array}{c} x \multimap (y \otimes z) \\ \text{---} \\ x \end{array} \begin{array}{c} y \\ \text{---} \\ z \end{array}.$$

In other words,  $\mathbf{T}_\Sigma$ , as well as the other categories presented in this paper, is a coloured PROP [19] whose colours are either generating objects  $A \in \mathcal{S}$  or elements of  $\text{obj}(\mathcal{S})$  of the form  $X \multimap Y$ .

The reader may recognize the usual syntax of string diagrams for symmetric monoidal categories on the first line of (1). The second line introduces the additional structure needed for the categories of interest. In particular, the semicircle renders the morphism  $ev_{X,Y}$ , and the bubble around a diagram  $c$  is the operator  $\Lambda_{X,Y,Z}(c)$ , both of which are part of the definition of a monoidal closed category (see e.g. [16, Definition 3.5]). Similarly, a loop over  $c$  corresponds to the trace operator  $Tr_{X,Y}^Z(c)$  of traced monoidal categories (see e.g. [22]). We refer to the arrows of  $\mathbf{T}_\Sigma$  as *traced hierarchical string diagrams*. The fact that such syntax and equations suffice to make  $\mathbf{T}_\Sigma$  and  $\mathbf{C}_\Sigma$  monoidal closed is detailed in [16]. Intuitively,  $\mathbf{C}_\Sigma$  is the category of hierarchical string diagrams without loops.

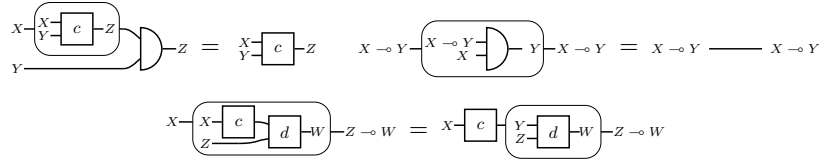


**Fig. 1.** Axioms of symmetric monoidal categories.



**Fig. 2.** Axioms of traced symmetric monoidal categories.

We now introduce the notion of string diagram rewriting that we intend to study. In the framework of [5], which we extend here, one is usually interested in ‘domain-specific’ rules, which describe some transformation of the computational processes represented by the string diagrams. Within this perspective, equations



**Fig. 3.** Axioms of closed monoidal categories.

in Figures 1-2 are considered ‘structural’ for the purpose of rewriting — they yield alternative representation of the same system, rather than transformations to be accounted for as actual rewrite rules. Therefore, the notion of rewriting we seek works ‘modulo traced monoidal structure’. We use TSMC to collectively refer to the axioms of traced symmetric monoidal categories, in Figures 1-2.

In order to soundly perform rewriting, we need to consider the categories of hierarchical string diagrams as TSMCs. Formally, we freely generate the categories  $\mathbb{C}_\Sigma$  and  $\mathbb{T}_\Sigma$  as done in Definition 3, but without quotienting by the axioms in Figure 3. These axioms will be taken instead as rewrite rules.

**Definition 4.** Let  $f, g: X \rightarrow Y$  and  $l, r: Z \rightarrow W$  be pairs of traced hierarchical string diagrams in  $\mathbb{T}_\Sigma$ . We say that  $f$  rewrites into  $g$  according to the rewrite rule  $\langle l, r \rangle$ , denoted as  $f \Rightarrow_{\langle l, r \rangle} g$ , if

$$x \text{---} [f] \text{---} y \stackrel{\text{TSMC}}{=} x \text{---} [c] \text{---} [l] \text{---} [d] \text{---} y \quad \text{and} \quad x \text{---} [g] \text{---} y \stackrel{\text{TSMC}}{=} x \text{---} [c] \text{---} [r] \text{---} [d] \text{---} y \quad (2)$$

or

$$x \text{---} [f] \text{---} y \stackrel{\text{TSMC}}{=} x \text{---} [c] \text{---} [a] \text{---} [d] \text{---} y \quad \text{and} \quad x \text{---} [g] \text{---} y \stackrel{\text{TSMC}}{=} x \text{---} [c] \text{---} [b] \text{---} [d] \text{---} y \quad (3)$$




where  $a$  and  $b$  are traced hierarchical string diagrams, such that  $a \Rightarrow_{\langle l, r \rangle} b$ .

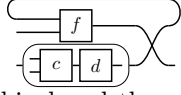
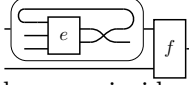
The above definition deserves some further remark. First, observe that, as anticipated, rewriting happens only modulo the axioms of TSMCs. Second, the definition is recursive: this is due to the fact that the string diagrams considered are hierarchical, and thus rewriting can happen at any level of the hierarchy.

For example, given  $R = \langle \begin{array}{c} [d] \\ [c] \end{array}, [e] \rangle$ , we claim that  $\begin{array}{c} [c] \\ [d] \end{array} \Rightarrow_R \begin{array}{c} [e] \end{array}$ .

At first, it might not be obvious why there is a match for  $R$  in  $\begin{array}{c} [c] \\ [d] \end{array}$ . The match appears clearly once we massage  $\begin{array}{c} [c] \\ [d] \end{array}$  using the axioms of TSMCs.

$$\begin{array}{c} [c] \\ [d] \end{array} = \begin{array}{c} [c] \\ [d] \end{array} \text{---} [c] \text{---} [d] = \begin{array}{c} [c] \\ [d] \end{array} \text{---} [c] \text{---} [d] = \begin{array}{c} [d] \\ [c] \end{array} \quad (4)$$

Now that the diagram is in the form of (2), we can directly substitute the occurrence of  with  and obtain .

Similarly, we can show that   $\Rightarrow_R$  , except that now the diagram is hierarchical and the rewrite happens inside a bubble. Therefore, this is the case of (3): we must bring the outer layer into the usual shape, and then proceed recursively for the diagram inside the bubble using (2).

Definition 4 can be easily extended to the case of multiple rewrite rules. In particular, given a set  $R$  of rewrite rules, we say that  $f \Rightarrow_R g$  if there is some  $\langle l, r \rangle \in R$  such that  $f \Rightarrow_{\langle l, r \rangle} g$ . As anticipated, to capture the monoidal closed axioms we need to consider them as rewrite rules. Thus, we build the set  $MC$  consisting of pairs  $\langle l, r \rangle$ , for each equation  $l = r$  in Figure 3.

Furthermore, if one is interested in *cartesian* closed categories it is necessary to add extra generators and equations to  $\mathbf{T}_\Sigma$ . Specifically, for each object  $X \in \text{obj}(\mathcal{S})$ , we add  $x \multimap^X : X \rightarrow X \otimes X$  and  $x \bullet : X \rightarrow I$  to  $\Sigma$  and quotient the morphisms by the axioms of natural cocommutative comonoids in Figure 4. We call this category  $\mathbf{T}_\Sigma^C$  and observe that, by Fox's theorem [15], it is a cartesian category, i.e. the monoidal product coincides with the categorical product. Again, these additional equations are not structural<sup>5</sup> and, for the purpose of rewriting in  $\mathbf{T}_\Sigma$ , they are considered as rewrite rules. We call  $\mathbf{T}_\Sigma^C$  the category augmented with the generators for cocommutative comonoids and we add the equations in Figure 4 to the set of rewrite rules  $MC$ , denoting the resulting set as  $CC$ .

$$\begin{array}{ccc}
 x \multimap^X = x \multimap^X & x \bullet^X = x \bullet^X & x \bullet^X = x \bullet^X \\
 x \multimap^Y = x \multimap^Y & x \bullet^Y = x \bullet^Y & x \bullet^Y = x \bullet^Y
 \end{array}$$

**Fig. 4.** Axioms of natural cocommutative comonoids.

### 3 Interpreting String Diagrams: Hierarchical Hypergraphs

String diagram rewriting as in Definition 4 is unpractical from a computational viewpoint: to find a matching we need to explore an equivalence class of objects — the string diagrams that are equivalent modulo the axioms of TSMCs to the given one. In the same style as [5], we introduce a notion of hypergraph with the

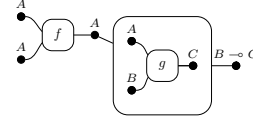
<sup>5</sup> More precisely, the axioms that are not structural are the naturality equations in the second row of Figure 4. Those on the first row can be absorbed by a sensible variation [25] of the hypergraph interpretation introduced in Section 3.

purpose of implementing string diagram rewriting as hypergraph rewriting. The improvement is given by the fact that the string diagrams equivalent modulo the axioms of TSMCs are going to be represented by the same hypergraph, thus making matching computationally more straightforward. The notion we seek needs to accommodate the hierarchical structure. It is the same used in [2], with the difference that we do not require acyclicity, to reflect the presence of traces.

**Definition 5.** Let  $(\mathcal{S}, \Sigma)$  be a monoidal closed signature. A  $(\mathcal{S}, \Sigma)$ -labelled hierarchical hypergraph  $\mathcal{H}$  is a tuple  $(V, E, s, t, l_V, l_E, p_V, p_E)$  consisting of:

- finite sets of vertices  $V$  and edges  $E$ ;
- source and target functions  $s, t: E \rightarrow V^*$ , mapping an edge to a finite list of vertices, consistently with the type of the generators in  $\Sigma$ ;
- a vertex labelling function  $l_V: V \rightarrow \text{obj}(\mathcal{S})$ ;
- an edge labelling function  $l_E: E \rightarrow \Sigma \cup \{\perp\}$ , assigning to each edge a symbol in  $\Sigma$  or no symbol, i.e.  $\perp$ ;
- parent functions  $p_V: V \rightarrow E \cup \{\perp\}$  and  $p_E: E \rightarrow E \cup \{\perp\}$ , assigning to each vertex and edge a parent edge or no parent, i.e.  $\perp$ ;
- the parent functions must satisfy some conditions: (1) an edge and any of its source and target vertices must have the same parent:  $p_V(v) = p_E(e) = p_V(v')$  for all  $v \in s(e)$  and  $v' \in t(e)$ , respectively; (2) the parent relation must be acyclic, in the sense that for all  $e \in E$  there is some  $k \geq 1$  such that  $p_{E, \perp}^k(e) = \perp$  where  $p_{E, \perp}: E \cup \{\perp\} \rightarrow E \cup \{\perp\}$  extends  $p_E$  with  $p_{E, \perp}(\perp) = \perp$ .

*Example 1.* Let  $(\mathcal{S}, \Sigma)$  be the monoidal closed signature where  $\mathcal{S} = \{A, B, C\}$  and  $\Sigma = \{f: A \otimes A \rightarrow A, g: A \otimes B \rightarrow C\}$ . The one on the right is a depiction of a  $(\mathcal{S}, \Sigma)$ -labelled hierarchical hypergraphs whose vertices are represented by the labelled black dots and the edges are the boxes.



Notice that there are three edges: those labelled with  $f$  and  $g$ , and one containing a sub-hypergraph, thus formally labelled with  $\perp$ .

In essence, a hierarchical hypergraph is a hypergraph with layers. The latter are determined by the parent function. The *outermost* layer 0 is formed by those vertices and edges that have no parent, i.e. their parent is  $\perp$ . Vertices and edges in layer  $n + 1$  are those with parent edge sitting at layer  $n$ .

A hypergraph is *discrete* when its set of edges is empty. We use letters  $v, w, \dots$  to denote discrete hypergraphs, with  $v, w$  standing for words over  $\text{obj}(\mathcal{S})$ .

It is important to also consider a notion of morphism between these structures. Note that a morphism may exist between hierarchical hypergraphs with a different number of layers: the outermost layer of the source hypergraph may be ‘anchored’ to any layer of the target one, as long as the hierarchy is respected.

**Definition 6.** Given two  $(\mathcal{S}, \Sigma)$ -labelled hierarchical hypergraphs  $\mathcal{H} = (V, E, s, t, l_V, l_E, p_V, p_E)$  and  $\mathcal{G} = (V', E', s', t', l'_V, l'_E, p'_V, p'_E)$ . A morphism of hierarchical hypergraphs  $\phi: \mathcal{H} \rightarrow \mathcal{G}$  is a pair of functions  $\phi_V: V \rightarrow V'$  and  $\phi_E: E \rightarrow E'$ , respecting the labelling of vertices and edges, the typing of the edges and the hierarchical structure, in the sense that:

1.  $l'_V(\phi_V(v)) = l_V(v)$  and  $l'_E(\phi_E(e)) = l_E(e)$ ;
2.  $s'(\phi_E(e)) = \hat{\phi}_V(s(e))$  and  $t'(\phi_E(e)) = \hat{\phi}_V(t(e))$ , where  $\hat{\phi}_V: V^* \rightarrow (V')^*$  is the obvious lifting of  $\phi_V: V \rightarrow V'$ ;
3.  $\phi_V(p_V(v)) = p'_V(\phi_V(v))$  if  $p_V(v) \neq \perp$ ;
4.  $\phi_E(p_E(e)) = p'_E(\phi_E(e))$  if  $p_E(e) \neq \perp$ .

A morphism  $\phi: \mathcal{H} \rightarrow \mathcal{G}$  is a monomorphism if  $\phi_V$  and  $\phi_E$  are both injective functions.  $(\mathcal{S}, \Sigma)$ -labelled hierarchical hypergraphs and morphisms between them form a category  $\mathbf{HHyp}_\Sigma$ , which is in fact symmetric monoidal.

In order to interpret string diagrams, we need to regard hierarchical hypergraphs as morphisms of a monoidal category, and thus define a notion of sequential and parallel composition for them. As in the case of ‘simple’ hypergraphs [5], this is achieved by adding structures identifying a left and a right interface of the hierarchical hypergraph. Formally, this amounts to consider *discrete cospans*, i.e. cospans of the kind  $v \rightarrow \mathcal{H} \leftarrow w$ , where  $v$  and  $w$  are discrete hypergraphs. The idea is that such a hypergraph will interpret a string diagram of type  $V \rightarrow W$ , where  $V$  and  $W$  correspond to some words  $v$  and  $w$  over  $\text{obj}(\mathcal{S})$ .

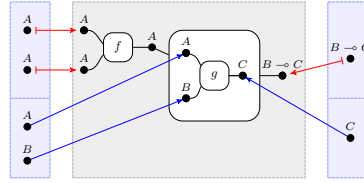
Given two discrete cospans of hypergraphs  $v \xrightarrow{f} \mathcal{H} \xleftarrow{g} u$  and  $w \xrightarrow{h} \mathcal{G} \xleftarrow{i} z$ , their parallel composition  $v \xrightarrow{f} \mathcal{H} \xleftarrow{g} u \otimes w \xrightarrow{h} \mathcal{G} \xleftarrow{i} z$  is  $v + w \xrightarrow{f+h} \mathcal{H} + \mathcal{G} \xleftarrow{g+i} u + z$  where  $+$  is the coproduct in  $\mathbf{HHyp}_\Sigma$ .

Similarly, given  $v \xrightarrow{f} \mathcal{H} \xleftarrow{g} u$  and  $u \xrightarrow{h} \mathcal{G} \xleftarrow{i} w$ , their sequential composition  $v \xrightarrow{f} \mathcal{H} \xleftarrow{g} u ; u \xrightarrow{h} \mathcal{G} \xleftarrow{i} w$  is obtained via the pushout above.

**Definition 7.** A hierarchical hypergraph with interfaces is a cospan in  $\mathbf{HHyp}_\Sigma$ , written  $v \rightarrow \mathcal{H} \leftarrow w$  with  $v$  and  $w$  discrete. These can be composed in sequence via pushout, and in parallel via the coproduct in  $\mathbf{HHyp}_\Sigma$ , and, in fact, they form a SMC that we denote as  $\mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$ .

Our graphical depiction of hierarchical hypergraphs with interfaces is as in Figure 5. We depict the interfaces in a blue background and the rest of the structure in a grey background. When the hypergraphs has more than one layer, we distinguish between an *outer* interface and *inner* interfaces. For instance, consider the one on the right.

The horizontal dotted line between interfaces separates the outer interface with the inner one. This is also made explicit by using a red/blue colouring for the interface maps. Multiple sets of inner interfaces, e.g. when the hypergraphs have more than two layers, are separated by multiple dotted lines.



**Fig. 5.** A hierarchical hypergraph with interfaces.



We now recall the hypergraph interpretation of hierarchical string diagrams, from [2], and we extend it to the case of traces.

**Definition 8.** Let  $[\cdot] : \mathbb{C}_\Sigma \rightarrow \mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$  be the symmetric monoidal functor defined as follows:

$$\begin{aligned}
 [\boxed{\phantom{x}}] &:= \text{blue box} \rightarrow \text{grey box} \leftarrow \text{blue box} & [x \multimap x] &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} \\
 \left[ \begin{array}{c} x \\ \text{---} \\ y \end{array} \times \begin{array}{c} y \\ \text{---} \\ x \end{array} \right] &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} & [x \multimap s \multimap y] &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} \\
 \left[ \begin{array}{c} x \multimap y \\ \text{---} \\ x \end{array} \right] &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} & [x \multimap c \multimap y] &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} \\
 \left[ \begin{array}{c} x \\ \text{---} \\ y \end{array} \right] \multimap z &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} & [x \multimap c \multimap y] \multimap z &:= [x \multimap c \multimap y] ; [y \multimap z] \\
 \left[ \begin{array}{c} x \\ \text{---} \\ y \end{array} \right] \multimap c \multimap z &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} & [x \multimap c \multimap y] \otimes [z \multimap d \multimap w] &:= [x \multimap c \multimap y] \otimes [z \multimap d \multimap w]
 \end{aligned}$$

**Proposition 1 ([2]).**  $[\cdot] : \mathbb{C}_\Sigma \rightarrow \mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$  is a faithful functor of SMCs.

The category of hypergraphs has a richer structure than  $\mathbb{C}_\Sigma$ . In particular, it is traced, in a way that is compatible with the functor  $[\cdot]$ . This yields an extension of the above interpretation to traced hierarchical string diagrams.

**Lemma 1.** There is a faithful functor of TSMCs  $[\![\cdot]\!] : \mathbb{T}_\Sigma \rightarrow \mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$ .

*Proof.* The category  $\mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$  is a hypergraph category, that is every object is equipped with a Frobenius bimonoid [5], given by the following hypergraphs:

$$\begin{aligned}
 \blacktriangleleft_X &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} & !_X &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} \\
 \blacktriangleright_X &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box} & i_X &:= \text{blue box} \xrightarrow{\bullet} \text{grey box} \xleftarrow{\bullet} \text{blue box}
 \end{aligned} \tag{5}$$

The category  $\mathbb{C}_\Sigma$  can be augmented with a Frobenius structure, by taking the coproduct  $\mathbb{C}_\Sigma + \mathbf{Frob}$ , where  $\mathbf{Frob}$  is the free traced hypergraph category on the signature consisting of:

$$\begin{aligned}
 x \multimap \begin{array}{c} x \\ \text{---} \\ x \end{array} &: X \rightarrow X \otimes X & x \multimap \bullet &: X \rightarrow I \\
 \begin{array}{c} x \\ \text{---} \\ x \end{array} \multimap x &: X \otimes X \rightarrow X & \bullet \multimap x &: I \rightarrow X
 \end{aligned} \tag{6}$$

Both  $\mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$  and  $\mathbb{C}_\Sigma + \mathbf{Frob}$  are TSMCs, with the trace being the canonical one arising from the Frobenius structure. For example, in  $\mathbb{C}_\Sigma + \mathbf{Frob}$ ,

the trace of  $\begin{array}{c} z \\ \text{---} \\ x \end{array} \multimap c \multimap \begin{array}{c} z \\ \text{---} \\ y \end{array}$  is  $\begin{array}{c} \bullet \\ \text{---} \\ x \end{array} \multimap \begin{array}{c} z \\ \text{---} \\ c \end{array} \multimap \begin{array}{c} z \\ \text{---} \\ y \end{array} \bullet$ .

Now observe that  $\mathbb{T}_\Sigma$  is a subcategory of  $\mathbb{C}_\Sigma + \mathbf{Frob}$  and that there is a faithful functor  $F : \mathbf{Frob} \rightarrow \mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$ , mapping the structure in (6) to the one in (5). The faithful functor  $[\![\cdot]\!] : \mathbb{T}_\Sigma \rightarrow \mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$  is thus obtained as the composite  $\mathbb{T}_\Sigma \xrightarrow{\iota} \mathbb{C}_\Sigma + \mathbf{Frob} \xrightarrow{[\cdot] + F} \mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$ .

It is convenient to visualise the action of  $\llbracket \cdot \rrbracket$  on string diagrams. If  $\llbracket \begin{array}{c} Z \\ \bullet \\ X \end{array} \text{---} c \text{---} \begin{array}{c} Z \\ \bullet \\ Y \end{array} \rrbracket =$

$$\begin{array}{c} Z \\ \bullet \\ X \end{array} \rightarrow \begin{array}{c} Z \\ \bullet \\ X \end{array} \text{---} c \text{---} \begin{array}{c} Z \\ \bullet \\ Y \end{array} \leftarrow \begin{array}{c} Z \\ \bullet \\ Y \end{array}, \text{ then } \llbracket \begin{array}{c} Z \\ \bullet \\ X \end{array} \text{---} c \text{---} \begin{array}{c} Z \\ \bullet \\ Y \end{array} \rrbracket = \begin{array}{c} X \\ \bullet \\ X \end{array} \rightarrow \begin{array}{c} Z \\ \bullet \\ X \end{array} \text{---} c \text{---} \begin{array}{c} Z \\ \bullet \\ Y \end{array} \leftarrow \begin{array}{c} Y \\ \bullet \\ Y \end{array}.$$

## 4 The Image of the Interpretation: Hypernets

As noticed in previous work, starting with [6], double-pushout rewriting of hypergraphs representing string diagrams does not mirror exactly string diagram rewriting: a more restricted notion of DPOI rewriting is required to achieve soundness. The same issue needs to be tackled in our context. For this purpose, it is important to first understand which class of hierarchical hypergraphs are in the image of the functor  $\llbracket \cdot \rrbracket$  we just introduced. The resulting notion will be the one of *hypernet*, generalising the acyclic hypernets defined in [2]. This generalisation is made possible by moving from *monogamous acyclic* to *partial monogamous* cospans of hypergraphs.

**Definition 9 (Acyclicity).** For a hypergraph  $\mathcal{H}$  and hyperedges  $e, e'$ , a path from  $e$  to  $e'$  is a finite sequence of hyperedges  $e_1, \dots, e_n$  such that  $e_1 = e$ ,  $e_n = e'$  and for  $i < n$  there exists a  $u$  that is in the target of  $e_i$  and in the source of  $e_{i+1}$ .

A hypergraph is acyclic if there exists no path from a vertex to itself.

Acyclicity extends to hypergraphs with interfaces  $v \rightarrow \mathcal{H} \leftarrow w$ , by simply requiring  $\mathcal{H}$  to be acyclic. However, cospan composition does not preserve acyclicity [6]. This issue is avoided by considering monogamous hypergraphs.

**Definition 10 (Degree of a vertex).** The in-degree of a vertex  $u$  in a hypergraph  $\mathcal{H}$  is the number of pairs  $(e, i)$  where  $e$  is a hyperedge with  $u$  as its  $i$ -th target. Similarly, the out-degree of  $u$  is the number of pairs  $(e, j)$  where  $e$  is a hyperedge with  $u$  as its  $j$ -th source.

**Definition 11 (Monogamy).** Let  $v \xrightarrow{f} \mathcal{H} \xleftarrow{g} w$  be a hypergraph with interfaces,  $\text{in}(\mathcal{H})$  the image of  $f$  and  $\text{out}(\mathcal{H})$  the image of  $g$ . We say that the cospan is monogamous if  $f$  and  $g$  are mono and for all vertices  $u$  of  $\mathcal{H}$ :

- the in-degree of  $u$  is 0 if  $u \in \text{in}(\mathcal{H})$ , and 1 otherwise;
- the out-degree of  $u$  is 0 if  $u \in \text{out}(\mathcal{H})$ , and 1 otherwise.


*Example 2.* The cospan in Figure 5 is monogamous since the in- and out-degree of each vertex is at most 1 and, moreover, the in- (resp. out-) degree of the vertices connected to a left (resp. right) interface is 0. In contrast, the two cospans below are *not* monogamous. For the cospan on the left, the vertex labelled  $A$  has in-degree 0 but out-degree 2, as it is connected to an interface on the left but it appears twice as the source of  $f$ . For the one on the right, the vertex labelled  $C$  has both in- and out-degree 0, but it is not connected to any interface.



Monogamous acyclic cospans of hierarchical hypergraphs have been shown to be a suitable combinatorial characterization of hierarchical string diagrams. However, for the traced hierarchical setting, it is necessary to drop the acyclicity condition and require a weaker notion of monogamy.

**Definition 12 (Partial monogamy).** A hypergraph with interfaces  $v \xrightarrow{f} \mathcal{H} \xleftarrow{g} w$ , is partial monogamous if  $f$  and  $g$  are mono and for all vertices  $u$  of  $\mathcal{H}$ :

- the in and out-degrees of  $u$  are both 0 if  $u \in \text{in}(\mathcal{H})$  and  $u \in \text{out}(\mathcal{H})$ ;
- the in-degree of  $u$  is 0 and the out-degree of  $u$  is 1 if  $u \in \text{in}(\mathcal{H})$ ;
- the in-degree of  $u$  is 1 and the out-degree of  $u$  is 0 if  $u \in \text{out}(\mathcal{H})$ ;
- the in and out-degrees of  $u$  are both 1 or 0.

An example of partial monogamous cospan is the hypergraph with interfaces in Example 2 on the right. The intuition is that vertices like the one labelled  $C$  in the example represent traced identities, i.e. closed diagrams like .

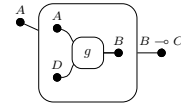
**Definition 13.** A subgraph  $\mathcal{G}$  of a hierarchical hypergraph  $\mathcal{F}$  is down-closed if for all edges  $e$  in  $\mathcal{G}$ ,  $\mathcal{F}_e \subseteq \mathcal{G}$ . Where  $\mathcal{F}_e$  consists of vertices  $\hat{v}$  and edges  $\hat{e}$ , such that  $p_{E,\perp}^k(\hat{e}) = e$  and  $p_{E,\perp}^j(p_V(\hat{v})) = e$ , for some  $k \geq 1$  and  $j \geq 0$ . In other words,  $\mathcal{F}_e$  is the inner hypergraph sitting inside of  $e$ .

**Definition 14.** A hierarchical hypergraph with interfaces  $v \rightarrow \mathcal{H} \leftarrow w$  is a hypernet if

1. it is partial monogamous when the hierarchical structure is forgotten,
2. if  $l_E(e) \neq \perp$  then  $e$  has no children (it is not parent of any vertex nor hyperedge),
3. if  $l_E(e) = \perp$ , then  $e$  is a well-typed abstraction, that is: if the input and output interfaces of  $\mathcal{H}_e$  are labelled with  $X$  and  $Y$ , respectively, then there exists  $Z$  such that  $s(e) \otimes Z = X$  and  $t(e) = Z \multimap Y$ .

A hierarchical hypergraph that only satisfies the first two conditions is called a weak hypernet. We denote as  $\mathbf{Hynet}_\Sigma$  the subcategory of  $\mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$  consisting of hypernets and morphisms between them.

The well-typed abstraction condition in Definition 14 ensures that the interfaces of the hierarchical hyperedges are typed accordingly to the outer interfaces of the hypergraph sitting inside of it, and thus mimicking the abstraction operator of monoidal closed categories. For example, the hierarchical hyperedge in Figure 5 respects this condition, whereas the one on the right does not, since the vertex labelled  $B \multimap C$  should instead be  $D \multimap B$ .



**Definition 15.** An acyclic hypernet is an hypernet whose underlying hypergraph structure is monogamous acyclic when the hierarchical structure is forgotten. We denote as  $\mathbf{AHynet}_\Sigma$  the subcategory of  $\mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$  consisting of acyclic hypernets and morphisms between them.

**Proposition 2.**  $\mathbf{Hynet}_\Sigma$  is a traced symmetric monoidal category.

*Proof (sketch).* Unlike  $\mathbf{Csp}_D(\mathbf{HHyp}_\Sigma)$ ,  $\mathbf{Hynet}_\Sigma$  is not a hypergraph category. In fact, the hypergraphs in (5) do not satisfy the partial monogamy requirement, and thus they are not hypernets. However, the Frobenius structure is only used to construct the canonical trace, which always yields a partial monogamous cospan.

The remaining of this section aims at showing the correspondence between traced hierarchical string digrams and hypernets.

**Proposition 3.** Any hypernet  $v \rightarrow \mathcal{A} \leftarrow w$  can be factored as

$$Tr_{v,w}^i(i + v \rightarrow \mathcal{C}_1 \leftarrow u + v' ; (u \rightarrow u \leftarrow u \otimes v' \rightarrow \mathcal{G} \leftarrow w') ; u + w' \rightarrow \mathcal{C}_2 \leftarrow i + w)$$

where  $i + v \rightarrow \mathcal{C}_1 \leftarrow u + v'$  and  $u + w' \rightarrow \mathcal{C}_2 \leftarrow i + w$  are acyclic hypernets and  $\mathcal{G}$  is an outermost-layer subnet of  $\mathcal{A}$ .

*Proof.* Since  $\mathbf{Hynet}_\Sigma$  is a TSMC,  $\mathcal{A}$  can be factored as  $Tr_{v,w}^i(i + v \rightarrow \mathcal{B} \leftarrow i + w)$ , where  $\mathcal{B}$  is a hypernet whose outermost layer is trace-free. At this point the outermost layer of  $\mathcal{B}$  is an acyclic hypernet that can be factored as  $i + v \rightarrow \mathcal{C}_1 \leftarrow u + v' ; (u \rightarrow u \leftarrow u \otimes v' \rightarrow \mathcal{G} \leftarrow w') ; u + w' \rightarrow \mathcal{C}_2 \leftarrow i + w$  [3, Lemma 4.11].

The following result establishes the correspondence between hypernets and traced hierarchical string diagrams.

**Lemma 2.** Every hierarchical hypergraph with interfaces is in the image of  $\llbracket \cdot \rrbracket$  if and only if it is a hypernet.

*Proof.* For the left to right direction it suffices to check, by induction on the syntax of  $\mathbb{T}_\Sigma$ , that the requirements of Definition 14 are met by  $\llbracket c \rrbracket$ .

For the other direction observe that, by Proposition 3, the factorisation of a hypernet directly yields a traced hierarchical string diagram in the same shape.

## 5 Rewriting for Traced Monoidal Closed Categories

In the previous section we established a correspondence between traced hierarchical string diagrams and hypernets. In this section we provide a notion of rewriting for hypernets as a suitable version of DPOI rewriting. Finally, we show a correspondence between rewriting of string diagrams (Definition 4) and its hypernet counterpart (Definition 16).

**Definition 16 (Hypernet Rewriting).** Let  $\mathcal{L}, \mathcal{R}$  be parallel hypernets, i.e. with same outermost interfaces  $v', w'$ . We say that the hypernet  $\mathcal{F}$  with interfaces  $\mathbf{v}, \mathbf{w}$  and outermost interfaces  $v, w$ , rewrites into the parallel hypernet  $\mathcal{G}$  according to the rewrite rule  $\langle \mathcal{L}, \mathcal{R} \rangle$ , denoted as  $\mathcal{F} \Rightarrow_{\langle \mathcal{L}, \mathcal{R} \rangle} \mathcal{G}$ , if there is

- $$\begin{array}{ccccc}
\mathcal{L} & \xleftarrow{[l_i, l_o]} & v' + w' & \xrightarrow{[r_i, r_o]} & \mathcal{R} \\
m \downarrow & & \downarrow [d_i, d_o] & & \downarrow \\
\mathcal{F} & \xleftarrow{\quad} & \mathcal{C} & \xrightarrow{\quad} & \mathcal{G} \\
& \nwarrow [f_i, f_o] & \uparrow [c_i, c_o] & \nearrow [g_i, g_o] & \\
& & v + w & & 
\end{array}$$

For example, if  $m$  was mono we could not match  $\bullet \boxed{c} \bullet$  in  $\boxed{c}$ . While requiring convexity does not permit rewrites as in (4).

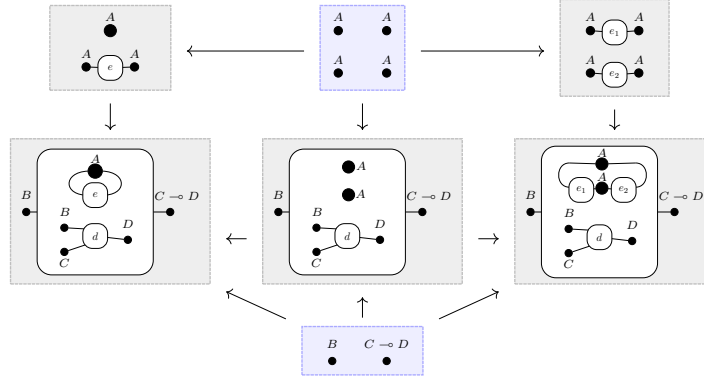
**Proposition 4.** *Let  $(\mathcal{L}, \mathcal{R})$  be a hypernet rewrite rule,  $\mathcal{F}$  a hypernet and  $m: \mathcal{L} \rightarrow \mathcal{F}$  a morphism of hypernets. Then the object  $\mathcal{C}$ , called boundary complement, of Definition 16 exists and  $\mathcal{G}$  is a hypernet.*

Now, the pushout with  $\mathcal{R}$  restores the well-typedness of the abstraction, since  $\mathcal{L}$  and  $\mathcal{R}$  have the same outermost interfaces, and produces the hypernet  $\mathcal{G}$  obtained by gluing  $\mathcal{R}$  in  $\mathcal{C}$  in place of the hole left by  $\mathcal{L}$ .

$$v' \rightarrow \mathcal{L} \leftarrow w' = \begin{array}{|c|} \hline A \\ \hline \bullet \\ \hline A \\ \hline \bullet \\ \hline \end{array} \rightarrow \begin{array}{|c|} \hline A \\ \hline \bullet \\ \hline A \\ \hline \bullet \\ \hline \end{array} \leftarrow \begin{array}{|c|} \hline A \\ \hline \bullet \\ \hline A \\ \hline \bullet \\ \hline \end{array}$$

$$v' \rightarrow \mathcal{R} \leftarrow w' = \begin{array}{|c|} \hline A \\ \hline \bullet \\ \hline A \\ \hline \bullet \\ \hline \end{array} \rightarrow \begin{array}{|c|} \hline A \quad A \\ \hline \bullet \quad e_1 \quad \bullet \\ \hline A \quad A \\ \hline \bullet \quad e_2 \quad \bullet \\ \hline A \quad A \\ \hline \end{array} \leftarrow \begin{array}{|c|} \hline A \\ \hline \bullet \\ \hline A \\ \hline \bullet \\ \hline \end{array}$$

Now  $\mathcal{F} \Rightarrow_{\langle \mathcal{L}, \mathcal{R} \rangle} \mathcal{G}$  by the valid DPOI diagram below, where we name the  $A$ -labelled vertices to clarify how they are mapped by the morphisms.



Notice that the boundary complement is only a weak hypernet, since the hypergraph in the middle does not respect the well-typed abstraction condition, i.e. the outermost interfaces are not typed according to the hypergraph sitting inside the hierarchical hyperedge.

The hypernet rewriting above corresponds to a rewriting of string diagrams. Let  $l, r, f$  and  $g$  be the following diagrams in  $\mathbb{T}_\Sigma$

$$l = \begin{array}{c} A \text{---} A \\ A \text{---} [e] \text{---} A \end{array} \quad r = \begin{array}{c} A \text{---} [e_1] \text{---} A \\ A \text{---} [e_2] \text{---} A \end{array} \quad f = \begin{array}{c} A \\ B \text{---} [e] \text{---} D \\ C \text{---} [d] \text{---} D \end{array} \quad g = \begin{array}{c} A \\ B \text{---} [e_1] \text{---} D \\ C \text{---} [d] \text{---} D \end{array} \quad C \multimap D.$$

Then one can show that  $f \Rightarrow_{\langle l, r \rangle} g$ . The full derivation is similar to the one in (4), except that it happens inside the hierarchical structure.

The correspondence hinted above between hypernet rewriting and string diagrams rewriting is formalised by the following completeness theorem.

**Theorem 1.** *Let  $f, g: X \rightarrow Y$  and  $l, r: Z \rightarrow W$  be pairs of traced hierarchical string diagrams in  $\mathbb{T}_\Sigma$ . Then,  $f \Rightarrow_{\langle l, r \rangle} g$  if and only if  $\llbracket f \rrbracket \Rightarrow_{\langle \llbracket l \rrbracket, \llbracket r \rrbracket \rangle} \llbracket g \rrbracket$ .*

*Proof (sketch).* For one direction assume  $f \Rightarrow_{\langle l, r \rangle} g$ , then  $f$  and  $g$  can be put in the form of Definition 4. Then we compute cospans  $\llbracket l \rrbracket, \llbracket r \rrbracket, \llbracket f \rrbracket, \llbracket g \rrbracket$ . The weak hypernet  $\mathcal{C}$  is computed starting from  $f$ . In particular, we erase the occurrence of  $l$  in  $f$  (observe that this may happen at any level of the hierarchy, if we are in the case of Definition 4.(2)) and we rearrange the wires in the expected way. Call such diagram  $C$ , then  $\mathcal{C} := \llbracket C \rrbracket$ . Composition of cospans is by pushout and thus, by aptly combining this data, the DPOI diagram in Definition 16 is recovered.

As a corollary, we obtain the characterisation of rewriting that accounts also for the non-structural rules of monoidal closed categories.

**Corollary 1.** *Let  $f, g: X \rightarrow Y$  and  $l, r: Z \rightarrow W$  be pairs of traced hierarchical string diagrams in  $\mathbb{T}_\Sigma$ . Then,  $f \Rightarrow_{\text{MC}} g$  if and only if  $\llbracket f \rrbracket \Rightarrow_{\{\langle \llbracket l \rrbracket, \llbracket r \rrbracket \rangle \mid (l, r) \in \text{MC}\}} \llbracket g \rrbracket$ .*

Note that equivalence in traced symmetric monoidal closed categories is not captured on the nose. However, observe that the rules in MC have an operational meaning, reminiscent of the  $\eta$  and  $\beta$  rules of the  $\lambda$ -calculus. This justifies making their application tangible in diagrammatic reasoning, rather than being absorbed by the graph-theoretic interpretation.

*Remark 2.* As briefly mentioned in the introduction, string diagram rewriting is computationally infeasible because each rewrite step requires inspecting equivalence classes of diagrams to find a redex. Such a computational overload is avoided in hypernet rewriting, since hypergraphs internalise the axioms corresponding to the geometrical manipulations of the diagrams in Figures 1 and 2.

Let us give some more detail on how a match in DPOI rewriting may be effectively computed. It mainly involves two steps: (a) solving a subgraph isomorphism problem and (b) enumerating boundary complements. For (a) there exist efficient solutions to the problem, notably Ullmann’s algorithm [32]. For (b), the situation is more articulated. In acyclic hypernet rewriting, the matching is required to be convex and the cospans are monogamous, making the boundary complements unique (up to iso). The presence of the trace in the hypernets studied in this paper forces the matching to be not necessarily convex, and the cospans under consideration to be partial monogamous rather than monogamous. In this setting the boundary complements are not necessarily unique, but can be enumerated. The number of boundary complements in hypergraph rewriting is tied to the notion of *adhesive* category [23]. Briefly, if the category of cospans under consideration is adhesive and certain additional conditions on rewriting are met, then the boundary complements are ensured to be unique. While  $\mathbf{Hynet}_\Sigma$  is not adhesive, it might be related to the notion of  $\mathcal{M}, \mathcal{N}$ -adhesivity studied in [10]. This could, in principle, allow us to define a bound on the number of boundary complements present in our setting. We leave a more detailed investigation of the implementation details to a follow up of this work.

## 6 $\lambda$ -calculus with Explicit Recursion

In the  $\lambda$ -calculus, recursion is usually represented by means of a fixed-point combinator. Although this method is theoretically sound, it is often impractical and does not reflect actual implementations. A more convenient way to represent recursive functions is provided by an explicit recursion operator, commonly found in functional languages as the `letrec` construct.

A version of the  $\lambda$ -calculus featuring explicit recursion, that we denote as  $\lambda_{\text{letrec}}$ , has been studied in [21]. In particular, its categorical interpretation extends the well-known correspondence between the simply typed  $\lambda$ -calculus and cartesian closed categories to  $\lambda_{\text{letrec}}$  and traced cartesian closed categories.

In this section we develop a string diagrammatic interpretation of  $\lambda$ -terms featuring explicit recursion, crucially exploiting the graphical language introduced in this paper. To do so, we first recall the syntax of  $\lambda_{\text{letrec}}$ .

Fixed a set of variables  $X$  and a set of basic types  $\mathcal{S}$ , we build types as  $T, T' := A \mid T \rightarrow T'$ , where  $A \in \mathcal{S}$ . Terms are generated according to the following grammar, where  $x \in X$ :

$$u, v := x \mid \lambda x. u \mid v u \mid \mathbf{letrec} \ x = u \ \mathbf{in} \ v \quad (7)$$

Among terms generated by (7), we consider only those that can be typed according to type judgements of the form  $\Gamma \vdash u : T$ , where  $\Gamma = x_1 : T_1, \dots, x_n : T_n$  is a variable type assignment. Then  $\Gamma \vdash u : T$  is interpreted as  $u$  has type  $T$  if each  $x_i$  appearing in  $u$  has type  $T_i$ . Type judgements are defined according to the following rules:

$$\Gamma, x : T \vdash x : T \quad \frac{\Gamma, x : T \vdash u : T'}{\Gamma \vdash \lambda x. u : T \rightarrow T'} \quad \frac{\Gamma \vdash u : T \quad \Gamma \vdash v : T \rightarrow T'}{\Gamma \vdash v u : T'} \quad \frac{\Gamma, x : T \vdash u : T \quad \Gamma, x : T \vdash v : T'}{\Gamma \vdash \mathbf{letrec} \ x = u \ \mathbf{in} \ v : T'}$$

Now, recall the traced cartesian closed category  $\mathbf{T}_\Sigma^C$  and the traced symmetric monoidal category  $\mathbb{T}_\Sigma^C$ . As we want to perform rewriting on  $\lambda$ -terms, we interpret them as traced hierachial string diagrams in the latter category. Thus, we define an encoding map  $\epsilon : \lambda_{\mathbf{letrec}} \rightarrow \mathbb{T}_\Sigma^C$  as follows.

Types are interpreted as objects of the category: in particular for every  $A \in \mathcal{S}$ ,  $\epsilon(A) = A$ ,  $\epsilon(T \rightarrow T') = \epsilon(T) \multimap \epsilon(T')$  and  $\epsilon(\Gamma) = \epsilon(x_1 : T_1, \dots, x_n : T_n) = \epsilon(T_1) \otimes \dots \otimes \epsilon(T_n)$ . Typed terms are then interpreted inductively as follows:

$$\begin{aligned} \epsilon(\Gamma, x : T \vdash x : T) &= \begin{array}{c} \epsilon(\Gamma) \text{---} \bullet \\ \epsilon(T) \text{---} \epsilon(T) \end{array} \\ \epsilon(\Gamma \vdash \lambda x. u : T \rightarrow T') &= \begin{array}{c} \epsilon(\Gamma) \text{---} \boxed{\epsilon(\Gamma, x : T \vdash u : T')} \text{---} \epsilon(T') \\ \epsilon(T) \text{---} \epsilon(T) \end{array} \\ \epsilon(\Gamma \vdash v u : T') &= \begin{array}{c} \epsilon(\Gamma) \text{---} \boxed{\epsilon(\Gamma \vdash v : T \rightarrow T')} \\ \epsilon(\Gamma) \text{---} \boxed{\epsilon(\Gamma \vdash u : T)} \end{array} \text{---} \epsilon(T') \\ \epsilon(\Gamma \vdash \mathbf{letrec} \ x = u \ \mathbf{in} \ v : T') &= \begin{array}{c} \epsilon(\Gamma) \text{---} \boxed{\epsilon(x : T, \Gamma \vdash u : T)} \\ \epsilon(\Gamma) \text{---} \boxed{\epsilon(u : T, \Gamma \vdash v : T')} \end{array} \text{---} \epsilon(T') \end{aligned}$$

Notice from the diagrammatic encoding the various structures at play: the comonoids are necessary to deal with variable sharing; the hierachical structure is essential to express the  $\lambda$ -abstraction; and finally, the trace allows to model the explicit recursion operator.

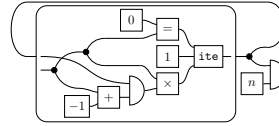
By adding more generators and equations we can model other convenient features of a programming language such as arithmetic and logical operations or the **if-then-else** construct.

For example, the recursive program computing the factorial, below on the left, is represented as the diagram on the right, where **ite** corresponds to a function that selects a branch (second or third argument) according to the value of the guard (first argument).

```

letrec f =  $\lambda x.$  if   x = 0
                then 1
                else x  $\times$  f(x - 1)
in f(n)

```





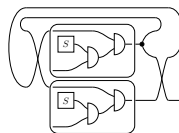
A direct consequence of the encoding of  $\lambda_{\text{letrec}}$  is that we can reason diagrammatically to prove properties of functional programs. Even more, the results presented in §§3, 4 and 5, yield an interpretation of  $\lambda_{\text{letrec}}$ -terms as hierarchical hypergraphs whose rewriting can be studied in terms of hypernet rewriting.

### 6.1 On confluence

A known result by Ariola and Klopp [1] shows that the  $\lambda$ -calculus with explicit recursion is not confluent. In this section we discuss this result under the lens of string diagrams rewriting, leveraging the algebraic nature of our axiomatisation.

Consider the following mutual-recursive program of  $\lambda_{\text{letrec}}$  below on the left, together with its string diagrammatic interpretation on the right.

$$\begin{aligned} \text{letrec } f &= \lambda x. g(Sx) , \\ g &= \lambda y. f(Sx) \\ \text{in } f \end{aligned}$$



Using the axioms of traced symmetric monoidal categories, one can easily show that the diagram above is equivalent to the one in the middle below. Moreover their combinatorial representations coincide to the same hypernet.

(8)

Note that the diagram in the middle can be rewritten into two non-joinable diagrams. The rewrite sequence labeled (a) applies the axioms of monoidal closed categories to perform a  $\beta$ -reduction, while the sequence labeled (b) first duplicates the bubble using the copier and then performs (a).

As noted in [1], non-confluence lies in the fact that the two diagrams are “out-of-sync”. Specifically, any applicable rewrite will result in one diagram containing an odd number of  $S$ ’s, while the other will have an even number.

Finally, note that in (8), we are working with string diagrams that include the additional generators and axioms from Figure 4. Therefore (8) is not, in general, a counterexample to confluence of the rewriting system for hypernets.

## 7 Discussion and Future Work

Traced monoidal closed categories provide an expressive setting for reasoning about computational processes with both recursion and higher-order features. In this paper we provide a comprehensive account of the diagrammatic language for these categories. We represent their morphisms as a graphical syntax of hierarchical string diagrams with a trace forming operation. We then give a combinatorial interpretation of these structures, in terms of partial monogamous cospans of hierarchical hypergraphs. Importantly, string diagrams that are equivalent modulo the laws of traced symmetric monoidal categories are represented

as the same cospan. This allows us to interpret string diagram rewriting as a certain variant of DPO hypergraph rewriting. Correctness of this characterisation is particularly subtle, as we need to handle both partial monogamy and the hierarchical structure of hypergraphs in the rewriting. As typical in this kind of result (see eg. [6]), the key payoff is that a rewrite step modulo traced symmetric monoidal structure corresponds to a rewrite step ‘on-the-nose’ in the hypergraph interpretation. As a proof of concept, we show a diagrammatic representation of  $\lambda_{\text{letrec}}$  and we discuss confluence of the resulting rewriting system.

As future work, we plan to investigate various properties of traced monoidal closed categories in our diagrammatic calculus. In particular, the relationship between the Int construction and the categories discussed in this paper. We are also interested in using our framework for program transformation, in the same spirit as string diagrams with additional structures are employed in [3] for automatic differentiation and in [16] for closure conversion. Finally, we intend to explore the link between our notion of rewriting and the graph rewriting techniques in [27, 28, 26] for compiler optimisation.

*Acknowledgements.* The authors acknowledge support from EPSRC EP/V002376/1. F. Zanasi is also supported by the ARIA Safeguarded AI TA1.1 programme. We thank the anonymous reviewers for their detailed comments and feedback.

*Disclosure of Interests.* The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Cyclic lambda graph rewriting. In: Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science. pp. 416–425. IEEE (1994)
2. Alvarez-Picallo, M., Ghica, D., Sprunger, D., Zanasi, F.: Rewriting for monoidal closed categories. In: 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2022)
3. Alvarez-Picallo, M., Ghica, D.R., Sprunger, D., Zanasi, F.: Functorial string diagrams for reverse-mode automatic differentiation. arXiv preprint arXiv:2107.13433 (2021)
4. Baez, J., Erbele, J.: Categories in control. Theory and Applications of Categories **30**, 836–881 (2015), <http://www.tac.mta.ca/tac/volumes/30/24/30-24abs.html>
5. Bonchi, F., Gadducci, F., Kissinger, A., Sobocinski, P., Zanasi, F.: String Diagram Rewrite Theory I: Rewriting with Frobenius Structure. J. ACM **69**(2), 14:1–14:58 (2022). <https://doi.org/10.1145/3502719>, <https://doi.org/10.1145/3502719>
6. Bonchi, F., Gadducci, F., Kissinger, A., Sobocinski, P., Zanasi, F.: String diagram rewrite theory II: Rewriting with symmetric monoidal structure. Mathematical Structures in Computer Science **32**(4), 511–541 (2022)
7. Bonchi, F., Gadducci, F., Kissinger, A., Sobociński, P., Zanasi, F.: String diagram rewrite theory III: Confluence with and without Frobenius. Mathematical Structures in Computer Science **32**(7), 829–869 (2022)

8. Bonchi, F., Seeber, J., Sobocinski, P.: Graphical Conjunctive Queries. In: Ghica, D., Jung, A. (eds.) 27th EACSL Annual Conference on Computer Science Logic (CSL 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 119, pp. 13:1–13:23. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018). <https://doi.org/10.4230/LIPIcs.CSL.2018.13>, <http://drops.dagstuhl.de/opus/volltexte/2018/9680>
9. Bonchi, F., Sobocinski, P., Zanasi, F.: Full Abstraction for Signal Flow Graphs. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 515–526. POPL '15, Association for Computing Machinery, New York, NY, USA (Jan 2015). <https://doi.org/10.1145/2676726.2676993>
10. Castelnovo, D., Gadducci, F., Miculan, M.: A simple criterion for m,n-adhesivity. *Theor. Comput. Sci.* **982**, 114280 (2024). <https://doi.org/10.1016/J.TCS.2023.114280>, <https://doi.org/10.1016/j.tcs.2023.114280>
11. Coecke, B., Kissinger, A.: Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning. Cambridge University Press (2017). <https://doi.org/10.1017/9781316219317>
12. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation - part I: basic concepts and double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, pp. 163–246. World Scientific (1997)
13. Cruttwell, G.S., Gavranović, B., Ghani, N., Wilson, P., Zanasi, F.: Categorical foundations of gradient-based learning. In: European Symposium on Programming. pp. 1–28. Springer International Publishing Cham (2022)
14. Ehrig, H., Kreowski, H.J.: Parallelism of manipulations in information structures. In: MFCS 1976. pp. 284–293 (1976)
15. Fox, T.: Coalgebras and cartesian categories. *Communications in Algebra* **4**(7), 665–667 (1976). <https://doi.org/10.1080/00927877608822127>
16. Ghica, D., Zanasi, F.: String diagrams for  $\lambda$ -calculi and functional computation (2023)
17. Ghica, D.R., Jung, A.: Categorical semantics of digital circuits. In: 2016 Formal Methods in Computer-Aided Design (FMCAD). pp. 41–48 (2016). <https://doi.org/10.1109/FMCAD.2016.7886659>
18. Ghica, D.R., Kaye, G.: Rewriting modulo traced comonoid structure. In: 8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023). Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2023)
19. Hackney, P., Robertson, M.: On the category of props. *Applied Categorical Structures* **23**, 543–573 (2015)
20. Hasegawa, M.: Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi. In: International Conference on Typed Lambda Calculi and Applications. pp. 196–213. Springer (1997)
21. Hasegawa, M.: Models of Sharing Graphs: A Categorical Semantics of let and letrec. Springer Science & Business Media (2012)
22. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. *Math Procs Cambridge Philosophical Society* **119**(3), 447–468 (4 1996)
23. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. *Theoretical Informatics and Applications* **39**(3), 511–546 (2005)
24. Lambek, J.: Cartesian closed categories and typed  $\lambda$ -calculi. In: LITP Spring School on Theoretical Computer Science, pp. 136–175. Springer (1985)

25. Milosavljevic, A., Piedeleu, R., Zanasi, F.: Rewriting for symmetric monoidal categories with commutative (co)monoid structure. *Logical Methods in Computer Science* **Volume 21, Issue 1**, 12 (Jan 2025). [https://doi.org/10.46298/lmcs-21\(1:12\)2025](https://doi.org/10.46298/lmcs-21(1:12)2025), <https://lmcs.episciences.org/14937>
26. Muroya, K.: Hypernet semantics of programming languages. Ph.D. thesis, University of Birmingham, UK (2020)
27. Muroya, K., Ghica, D.R.: The dynamic geometry of interaction machine: a call-by-need graph rewriter. arXiv preprint arXiv:1703.10027 (2017)
28. Muroya, K., Ghica, D.R.: The dynamic geometry of interaction machine: A token-guided graph rewriter. *Logical Methods in Computer Science* **15** (2019)
29. Piedeleu, R., Zanasi, F.: A finite axiomatisation of finite-state automata using string diagrams. *Logical Methods in Computer Science* **Volume 19, Issue 1**, 13 (Feb 2023). [https://doi.org/10.46298/lmcs-19\(1:13\)2023](https://doi.org/10.46298/lmcs-19(1:13)2023), <https://lmcs.episciences.org/10400>
30. Piedeleu, R., Zanasi, F.: An introduction to string diagrams for computer scientists. *CoRR abs/2305.08768* (2023)
31. Selinger, P.: A Survey of Graphical Languages for Monoidal Categories. In: Coecke, B. (ed.) *New Structures for Physics*, *Lecture Notes in Physics*, vol. 813, pp. 289–355. Springer, Berlin, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-12821-9\\_4](https://doi.org/10.1007/978-3-642-12821-9_4)
32. Ullmann, J.R.: An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* **23**(1), 31–42 (1976)