Rewriting for Monoidal Closed Categories

Alvarez-Picallo, Mario; Ghica, Dan; Sprunger, David; Zanasi, Fabio

DOI:
10.4230/LIPIcs.FSCD.2022.29

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (Harvard):
Alvarez-Picallo, M, Ghica, D, Sprunger, D & Zanasi, F 2022, Rewriting for Monoidal Closed Categories. in AP Felty (ed.), *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).*, 29, Leibniz international proceedings in informatics, vol. 228, Schloss Dagstuhl, 7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, Haifa, Israel, 2/08/22.
https://doi.org/10.4230/LIPIcs.FSCD.2022.29

Rewriting for Monoidal Closed Categories

Mario Alvarez-Picallo
Programming Languages Laboratory, Huawei Research Centre, UK

Dan Ghica
Department of Computer Science, University of Birmingham, UK
Programming Languages Laboratory, Huawei Research Centre, Reading, UK

David Sprunger
Department of Computer Science, University of Birmingham, UK

Fabio Zanasi
Department of Computer Science, University College London, UK

Abstract

This paper develops a formal string diagram language for monoidal closed categories. Previous work has shown that string diagrams for freely generated symmetric monoidal categories can be viewed as hypergraphs with interfaces, and the axioms of these categories can be realized by rewriting systems. This work proposes hierarchical hypergraphs as a suitable formalization of string diagrams for monoidal closed categories. We then show double pushout rewriting captures the axioms of these closed categories.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting

Keywords and phrases string diagrams, rewriting, hierarchical hypergraph, monoidal closed category

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.29

Funding This work was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/V001612/1 and EP/V002376/1].

1 Introduction

Symmetric monoidal categories are algebraic settings for abstract functions (called *morphisms* or *arrows*) between different sources and targets (called *objects*), which support both sequential composition and parallel composition. These categories have two widespread notations: terms and string diagrams. Term notation is generally regarded as the *de facto* standard; string diagrams were once thought of as a "private device" useful for quick calculation, but which "should be provided with a firm theoretical foundation" in order to be credibly used in public [23].

Since (and including) Joyal and Street's landmark paper in 1991, much scholarly work has gone into the formalization of string diagrams. This has alternately supported and been motivated by a proliferation of applications in compositional system modelling, for example in the representation of Petri nets [28], (analog) electrical circuits [4], digital circuits [17], quantum processes [14], differentiable programs [36], and signal flow graphs [8].

Separately, monoidal closed categories, particularly cartesian closed categories, have been used in theoretical computer science as models for the simply typed lambda calculus and functional programming languages. String diagrams in these contexts give an alternative method for specifying, implementing, and reasoning about complex program transformations, such as automatic differentiation [2]. There is unfilled need for a diagrammatic language to reason in closed categories for these applications.

We propose *hierarchical string diagrams* as a language to represent morphisms in these closed categories, which extend ordinary monoidal string diagrams with a bubble operation to represent curried terms. Following [7, 5, 6], which formalizes string diagrams as hypergraphs

and gives a formal notion of hypergraph rewriting, we formalize these hierarchical string diagrams with *hierarchical hypergraphs*. Finally, we present a double-pushout rewriting system which is sound and complete for the axioms of monoidal closed categories, and also extensible to any other equational system.

Outline. In Section 2, we describe monoidal (closed) categories and their string diagrams. We then formalize string diagrams for monoidal categories in Section 3 as certain hypergraphs. We introduce hierarchical hypergraphs and formalize their rewriting in Section 4. Finally, we establish connections to term rewriting. Further comparisons with related work can be found in Section 6.

2 Monoidal closed categories and their string diagrams

In this section, we recall (strict symmetric) monoidal categories and their basic string diagrams. Then we recall monoidal closed categories and reintroduce a proposed graphical syntax for their morphisms, *hierarchical string diagrams*.

We assume familiarity with basic category theory. In this paper, the collection of objects of a category $\mathcal{C}$ is denoted $|\mathcal{C}|$ and the set of morphisms from A to B is $\mathcal{C}(A, B)$. The sequential composition of $f : A \to B$ and $g : B \to C$ is $f; g : A \to C$.[1] The identity morphism on an object A is denoted 1_A. The set of lists with entries in the set X is denoted X^* and similarly the application of a function $f : X \to Y$ to a list from X^* is denoted f^*.

2.1 Monoidal categories

▶ **Definition 1.** *A category $\mathcal{C}$ is* monoidal *means it is equipped with both a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ which is associative (up to a natural isomorphism) and an object I which is both a left and right unit for $\otimes$ (up to a natural isomorphism).*

The category is called strict monoidal *when these natural isomorphisms are identities.*

A strict monoidal category is called symmetric *if there is a natural isomorphism $\sigma_{A,B} : A \otimes B \to B \otimes A$ satisfying three properties: (1) $1_A = \sigma_{A,I}$, (2) $\sigma_{A,B}; \sigma_{B,A} = 1_{A\otimes B}$ (3) $(1_A \otimes \sigma_{B,C}); (\sigma_{A,C} \otimes 1_B) = \sigma_{A\otimes B,C}$.*

To reduce the use of grouping symbols, we adopt the convention that $\otimes$ binds tighter than ; meaning, e.g., $f \otimes g; h$ is $(f \otimes g); h$, not $f \otimes (g; h)$. We will refer to $f \otimes g$ as the *parallel composition (of f and g)*, and we call $f; g$ the *sequential composition (of f and g)*.

For simplicity, we restrict our attention to strict monoidal categories, noting that every monoidal category is monoidally equivalent to a strict monoidal category [26]. In every strict monoidal category, the equations in the first two rows of Figure 1 hold; the last row holds for strict symmetric monoidal categories.

$$
\begin{array}{lll}
(f; g); h = f; (g; h) & 1_A; f = f = f; 1_B & (f; g) \otimes (h; k) = (f \otimes h); (g \otimes k) \\
(f \otimes g) \otimes h = f \otimes (g \otimes h) & 1_I \otimes f = f = f \otimes 1_I & 1_{A\otimes B} = \sigma_{A,B}; \sigma_{B,A} \\
f \otimes g; \sigma_{C,D} = \sigma_{A,B}; g \otimes f & 1_{A\otimes B} = 1_A \otimes 1_B & \sigma_{A,B\otimes C} = \sigma_{A,B} \otimes 1_C; 1_B \otimes \sigma_{A,C}
\end{array}
$$

Figure 1 Laws of strict (symmetric) monoidal categories.

[1] In this paper, we use the relational order of composition, as opposed to the more common notation $g \circ f$.

String diagrams are a graphical syntax for morphisms in strict monoidal categories. In this syntax, morphisms are represented by labelled boxes with labelled wires on either side: the morphism $f : A \to B$ is denoted 

. A sequential composition is formed by vertically stacking the component string diagrams and joining corresponding wires, as in 

. Parallel composition places the diagrams side-by-side as in 

.

Some advantages of string diagram notation become clear when considering the equations of Figure 1. With the common conventions that the identity morphism on an object can be denoted by a plain wire 

and the identity morphism on the tensor unit I can be denoted by a blank space, the string diagrams representing either side of many of these equations appear the same up to some topological deformations.

A challenge in the use of string diagrams is the *wire matching problem*, where the operation of "stack and join corresponding wires" may not be well-defined. For example, if we represent $k : X \to A \otimes C$ by 

, we cannot match wires to compose with 

. A common way to avoid this problem is to restrict attention to freely generated monoidal categories where objects have unique representations. This ensures that every string diagram has a canonical arrangement of input and output wires, and therefore ensures that string diagrams for sequentially composable morphisms have matchable inputs and outputs.

Strict symmetric monoidal categories (SMCs) stipulate the existence of a family of morphisms: $\sigma_{A,B}$. Following common convention, instead of using a box labelled $\sigma_{A,B}$ as a string diagram for this morphism, we use a pair of crossing wires:  .

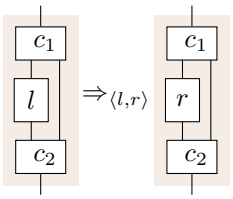


Figure 2 A rewriting step.

Similar to [7, 5, 6], we consider *string diagram rewriting*. In a symmetric monoidal category, a *string diagram rewriting rule* is a pair of parallel morphisms $\langle l, r\rangle$. A morphism d *rewrites directly* to another morphism e under such a rule (denoted $d \Rightarrow_{\langle l,r\rangle} e$) if there are morphisms c_1, c_2 and an object k such that $d = c_1; 1_k \otimes l; c_2$ and $e = c_1; 1_k \otimes r; c_2$. A *string diagram rewriting system* is a collection of rewriting rules; a morphism d *rewrites* to a parallel morphism e in a rewriting system if there is a sequence of direct rewrites in the system starting with d and ending in e.

2.2 Term rewriting for morphisms in symmetric monoidal categories

Commonly, morphisms in a category are described with terms. (For example, the equations of Figure 1.) Reasoning with these terms is performed by rewriting them with known equations. We recall a formal treatment of this next.

A *monoidal signature* $\Sigma = (\Sigma_O, \Sigma_M, t)$ consists of a set of types Σ_O, a set of constant symbols Σ_M and an assignment $t : \Sigma_M \to \Sigma_O^* \times \Sigma_O^*$ giving a list of input and output types to each constant symbol; abusing notation we denote this assignment by $\xi :_t A_1 \otimes \cdots \otimes A_n \to B_1 \otimes \cdots \otimes B_m$ where $\xi \in \Sigma_M$ and $A_i, B_j \in \Sigma_O$. We will often drop the subscript t. Again abusing notation, we denote the empty list of types by I.

Morphism terms τ in this signature are generated by the following BNF grammar: $\tau ::= \xi \mid f \mid 1_I \mid 1_A \mid \sigma_{A,B} \mid \tau_1; \tau_2 \mid \tau_1 \otimes \tau_2$ where $\xi \in \Sigma_M$ ranges over constants in the signature, f ranges over a set of variables, and $A, B \in \Sigma_O$ range over types in the signature. Terms are given lists of input and output types from Σ_O; given a context Γ (i.e. a typing of the variables denoted $f :_\Gamma X \to Y$), a term is said to be *well-typed* as usual. A term is a *ground term* if it contains no variables. An *equation* is a pair of terms of the same type in the same context. Substitution of a term (β) for a variable (f) in another term (α) is defined as usual and denoted $\alpha[\beta/f]$.

The free strict symmetric monoidal category S_Σ on a signature Σ has as objects lists of types from the signature and as morphisms equivalence classes of well-typed ground terms in the signature. The equivalence classes are the congruence classes generated by the equations of Figure 1, except the last two equations in the last column. Note the set of terms does not include identities or symmetries at product types ($1_{A\otimes B}$ or $\sigma_{A,B\otimes C}$): these are defined as composites of other generators via the last two equations.

A *term rewrite rule* is a pair of morphism terms $\langle \lambda, \rho \rangle$. A *substitution instance* of a rewrite rule $\langle \lambda, \rho \rangle$ is a pair of morphism terms $\langle \zeta, \xi \rangle$ such that applying the same substitutions to λ and ρ yields ζ and ξ, respectively. A *term rewrite system* R is a set of rewrite rules. A *context* is a term with a single occurrence of a designated variable $\bullet$ called its *hole*. We say a term α *rewrites directly* to the term β in the system R if there is a context C and a substitution instance of a rewrite rule $\langle \lambda, \rho \rangle \in R$ such that $C[\zeta/\bullet] = \alpha$ and $C[\xi/\bullet] = \beta$. In this case, we write $\alpha \to_R \beta$. A term α *rewrites* to the term β in a system R if there is a sequence of direct rewrites starting from α and ending in β.

If α and β are morphism terms that rewrite to one another via the bidirectional rewrite system formed from the equations of Figure 1), we say they are *equivalent modulo the laws of SMCs* and write $\alpha =_{SMC} \beta$.

Though many mathematicians default to reasoning with terms, string diagram rewriting possesses several powerful advantages. Emulating a string diagram rewrite step with terms may require several intermediate steps invoking laws of SMCs to find the appropriate representative of the term's equivalence class. This redex search is made drastically easier in string diagrams since morphisms are already quotiented by SMC laws.

However, string diagrams have disadvantages as well. Morphisms from a category are not a natural candidate for automated manipulation, whereas terms have a clear inductive structure. Additionally, due to the lack of variables, universally quantified equations (such as $f; !_B = !_A$) are emulated in string diagram rewriting system with a collection of rewrite rules, possibly even a rule schema, rather than with a single rewrite rule.

Implementing string diagrams with hypergraphs [7, 5, 6] ameliorates many of these computational disadvantages while retaining the automatic enforcement of SMC laws. We show how these hypergraphs can be defined inductively on morphism terms, while satisfying the SMC laws and thus serving as a functorial semantics for string diagrams. Following [7, 5, 6], we then use *double-pushout (DPO) graph rewriting* to rewrite hypergraphs in analogy with term rewriting and string diagram rewriting.

2.3 Monoidal closed categories

▶ **Definition 2.** *A* monoidal (right) closed category *is a monoidal category $\mathcal{C}$ satisfying for every pair of objects B, C there is an object $B \Rightarrow C$ and a morphism $ev_{B,C} : (B \Rightarrow C) \otimes B \to C$, and for every triple of objects A, B, C there is an operation $\Lambda_{A,B,C} : \mathcal{C}(A \otimes B, C) \to \mathcal{C}(A, B \Rightarrow C)$ satisfying three equations for all $f : A \otimes B \to C$ and $g : Z \to A$:*

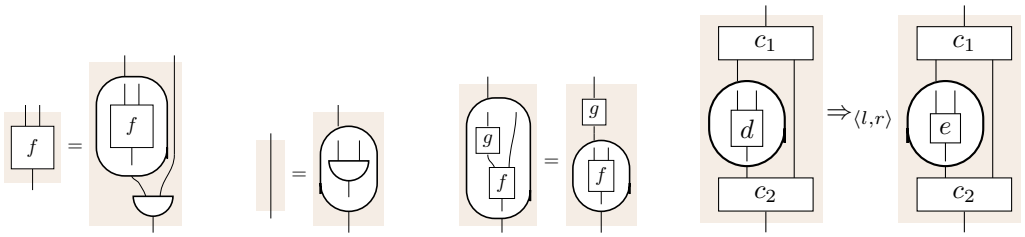
1. $f = \Lambda_{A,B,C}(f) \otimes 1_B ; ev_{B,C}$
2. $1_{B \Rightarrow C} = \Lambda_{B \Rightarrow C, B, C}(ev_{B,C})$
3. $\Lambda_{Z,B,C}(g \otimes 1_B ; f) = g ; \Lambda_{A,B,C}(f)$

A more common equivalent definition is that the right tensor functor $- \otimes B : \mathcal{C} \to \mathcal{C}$ has a right adjoint $B \Rightarrow -$. The currying operation Λ represents one direction of the homset presentation of this adjunction. The other direction is $h \mapsto h \otimes 1_B ; ev_{B,C}$.

If tensoring on the left has a right adjoint, the category is left closed. A symmetric monoidal category is right closed if and only if it left closed; we simply call it *closed*.

This presentation of monoidal closed categories induces a manageable set of changes to both the term calculus and the string diagram calculus. Though it is possible to present adjunctions via string diagrams, for example with functorial boxes [27], here we only need a representation for the currying operation Λ and some extra objects and morphisms.

Morphism terms for monoidal categories can be extended to terms for monoidal *closed* categories by adding a type-forming operation ($\Rightarrow$), a new collection of constant terms (ev), and a new term-forming operation (Λ). The BNF generating *closed morphism terms* is then $\rho ::= \xi \mid f \mid 1_I \mid 1_A \mid \sigma_{A,B} \mid ev_{A,B} \mid \rho_1 ; \rho_2 \mid \rho_1 \otimes \rho_2 \mid \Lambda(\rho)$. The free symmetric closed monoidal category can be defined again as equivalence classes of well-typed ground closed morphism terms; term rewriting is again similar to the symmetric monoidal case.



(a) β law. **(b)** (Small) η law. **(c)** Slide rule. **(d)** A hierarchical rewriting step.

Figure 3 Diagrammatic equations for monoidal closed categories and hierarchical rewriting.

For string diagrams, the new objects mean wires have some new labels available. For convenience, we introduce the new morphism box shape , as syntactic sugar for a box labelled ev. We diagrammatically represent Λ with a bubble operation. That is, if $f : A \otimes B \to C$, we write for $\Lambda(f)$. We call these *hierarchical string diagrams* since string diagrams may appear within wires of other string diagrams. The three equations of Definition 2 are interpreted diagrammatically as in Figure 3a-3c.

Rewriting of these hierarchical string diagrams is similarly an extension of rewriting string diagrams. A hierarchical rewriting rule is a pair of hierarchical string diagrams $\langle l, r \rangle$. A hierarchical string diagram directly rewrites to another if it can be put in the form of Figure 2 or the form of Figure 3d where d is a hierarchical string diagram that directly rewrites to e with the same rule.

3 String diagrams as hypergraphs

In this section, we formalize string diagrams as *monogamous directed acyclic hypergraphs* (Definition 7). We further define a rewriting scheme for these hypergraphs based on double pushout rewriting. In the next section, we will extend both of these notions to treat hierarchical string diagrams as hierarchical hypergraphs with a similar rewriting scheme.

3.1 Hypergraphs with interfaces

▶ **Definition 3.** *A* (directed) hypergraph (with labels from $\Sigma = (\Sigma_V, \Sigma_E)$) *is a tuple* $(V, E, s, t, \ell_V, \ell_E)$ *consisting of finite sets of* vertices V *and* edges E, source *and* target functions $s, t : E \to V^*$ *and* vertex *and* edge labelling functions $\ell_V : V \to \Sigma_V$ *and* $\ell_E : E \to \Sigma_E$.

When considering multiple hypergraphs, we distinguish their components by subscripting, so $V_{\mathcal{G}}$ is the vertices of $\mathcal{G}$, $s_{\mathcal{H}}$ is the source function for $\mathcal{H}$, etc. Note that our hypergraphs' edges come with *lists* (rather than sets) of source and target vertices. Despite this, we write $v \in s(e)$ when the vertex v occurs in the list of source vertices of the edge e.

We will depict hypergraphs graphically as in Figure 4a. In this hypergraph, there are three edges (white boxes labelled $+$, $\times$ and δ), and seven vertices (black dots). The black arrows indicate the source/target relationship: a vertex is a source of an edge if there is an arrow from the dot to the box and a target if the arrow goes from the box to the dot. The leftmost arrow at the top of the box corresponds to the first source vertex in the list, and similarly the leftmost bottom arrow is the first target.

(a) Hypergraph without interfaces. **(b)** Hypergraphs with interfaces.

Figure 4 Example hypergraphs.

We borrow terminology from graph theory for hypergraphs. We say a hypergraph is *discrete* if the edge set is empty. A *directed path* in a hypergraph is a finite list of alternating vertices and edges $(v_0, e_0, v_1, \ldots, v_k)$ with the property that $v_i \in s(e_i)$ and $v_{i+1} \in t(e_i)$ for all $0 \leq i < k$. The *length* of the path is the number of edges in the path. A hypergraph is *directed acyclic* if there are no postitive-length directed paths from a vertex to itself.

A *sub-hypergraph* $\mathcal{G}$ of the hypergraph $\mathcal{H}$ is a subset of $\mathcal{H}$'s vertices and edges such that the restrictions of s and t to $\mathcal{G}$ make it a hypergraph. A sub-hypergraph $\mathcal{G}$ is *convex (in* $\mathcal{H}$*)* if all directed paths in $\mathcal{H}$ between vertices in $\mathcal{G}$ are directed paths in $\mathcal{G}$.

The *in-degree* of a vertex v is the number of pairs $(e, i) \in E \times \mathbb{N}$ such that v is entry i in the list $t(e)$. Similarly, the *out-degree* is the number of occurrences of v in source lists.

▶ **Definition 4.** *Suppose* $\mathcal{F}$ *and* $\mathcal{G}$ *are two hypergraphs with labels from* Σ. *A* hypergraph morphism *from* $\mathcal{F}$ *to* $\mathcal{G}$ *is a pair of functions* $\phi_V : V_{\mathcal{F}} \to V_{\mathcal{G}}$ *and* $\phi_E : E_{\mathcal{F}} \to E_{\mathcal{G}}$ *compatible with source and target labelling in the sense that (1)* $s_{\mathcal{F}}; \phi_V^* = \phi_E; s_{\mathcal{G}}$*, (2)* $t_{\mathcal{F}}; \phi_V^* = \phi_E; t_{\mathcal{G}}$ *(3)* $\ell_{V,\mathcal{F}} = \phi_V; \ell_{V,\mathcal{G}}$*, and (4)* $\ell_{E,\mathcal{F}} = \phi_E; \ell_{E,\mathcal{G}}$

Recall $\phi_V^* : V_{\mathcal{F}}^* \to V_{\mathcal{G}}^*$ is the elementwise application of ϕ_V to the argument list.

Hypergraphs with a fixed set of labels and the morphisms between them form a category which we denote by Hyp_Σ.

▶ **Definition 5.** *A* hypergraph with (ordered) interfaces *is a cospan* $n \xrightarrow{f} \mathcal{F} \xleftarrow{g} m$ *in* Hyp_Σ *where the* n *and* m *are discrete hypergraphs with a specified total ordering on their vertices, and* f *and* g *are monos.*

We often refer to the hypergraphs n and m in the cospan as the *interfaces*, and the hypergraph $\mathcal{F}$ as just the hypergraph. Graphically, we distinguish interfaces with a blue background, as in Figure 4b. Since the interfaces are finite sets with a total order, we may also think of their vertex set as a list. (This happens, for example, in Definition 6.)

3.2 Monogamous directed acyclic hypergraphs

Next we will consider hypergraphs with labels drawn from a signature. When these hypergraphs have certain extra properties, we can think of them as syntax for representing morphisms from the free SMC on that signature. First, some preliminary notions.

▶ **Definition 6.** *Suppose* $n \to \mathcal{F} \leftarrow m$ *is a hypergraph with interfaces whose vertex labels are types in a signature* Σ*. The* input object *of the hypergraph is* $\widetilde{n} = \ell_{V,n}^*(V_n)$*, and the* output object *is* $\widetilde{m} = \ell_{V,m}^*(V_m)$*. For each edge* $e \in E_{\mathcal{F}}$*, the edge's* source object *is* $\widetilde{s(e)} = \ell_{V,\mathcal{F}}^*(s(e))$ *and its* target object *is* $\widetilde{t(e)} = \ell_{V,\mathcal{F}}^*(t(e))$*.*

We say these are source and target *objects* since these lists of types are objects in the free SMC on Σ. As examples, the input object of the hypergraph of Figure 5 is $A \otimes B \otimes A \otimes A$, the output object is $C \otimes A$, the source object of the edge labelled f is $A \otimes B$, and the target object of the edge labelled f is C.

▶ **Definition 7.** *Suppose* $\Sigma = (\Sigma_O, \Sigma_M, t)$ *is a signature and let* (Σ_A, Γ) *be a set of variables with a context. An* mda-hypergraph (in Σ) *is a hypergraph with ordered interfaces* $n \xrightarrow{f} \mathcal{F} \xleftarrow{g} m$ *with labels from* $(\Sigma_O, \Sigma_M + \Sigma_A)$ *satisfying*

1. *directed acyclicity,*
2. *the in-degree and out-degree of every vertex of* $\mathcal{F}$ *is at most 1,*
3. *vertices of* $\mathcal{F}$ *with in-degree 0 are precisely the image of* f*,*
4. *vertices of* $\mathcal{F}$ *with out-degree 0 are precisely the image of* g*,*
5. *for all* $e \in E$ *with* $\ell_{E,\mathcal{F}}(e) \in \Sigma_M$*, we have* $\ell_{E,\mathcal{F}}(e) :_t \widetilde{s(e)} \to \widetilde{t(e)}$*, and*
6. *for all* $e \in E$ *with* $\ell_{E,\mathcal{F}}(e) \in \Sigma_A$*, we have* $f :_\Gamma \widetilde{s(e)} \to \widetilde{t(e)}$*.*

In this case, we say $\mathcal{F}$ *is an mda-hypergraph* from the object $\widetilde{n}$ to the object $\widetilde{m}$*, and we say* $\widetilde{n}$ *and* $\widetilde{m}$ *are the* source and target objects *of* $\mathcal{F}$*, respectively. Two mda-hypergraphs with the same source and target objects are* parallel.

Cospans satisfying (2)–(4) are called *monogamous* [5, Definition 9]; "mda-hypergraph" is an abbreviation for "monogamous directed acyclic hypergraph". Conditions (5) and (6) are well-typing conditions. Condition (5) says if an edge is labelled by a constant, the input and output objects of that edge match the typing required by the signature. Condition (6) enforces the context's typing for the inputs and outputs of variable-labelled edges.

Two mda-hypergraphs in Σ may have different variables or different contexts for those variables. We say that two mda-hypergraphs are *compatible* if their contexts give the same type to the variables they have in common.

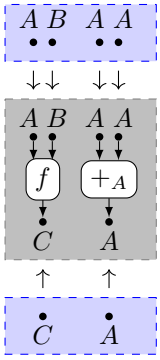


Figure 5 Example mda-hypergraph.

The hypergraph of Figure 4a fails to be an mda-hypergraph on at least three counts. It does not have interfaces, the vertex labelled $A \otimes C$ has in-degree 2, and there is a directed cycle. The hypergraphs with interfaces of Figure 4b also are not mda-hypergraphs. The left fails a monogamy condition: there is a vertex with out-degree 0 that is not in the image of the output interface. The right fails a well-typedness condition since the morphism $\sigma_{A,B}$ has codomain $B \otimes A$ but the target vertex labels of the edge labelled $\sigma_{A,B}$ is $A \otimes B$.

The hypergraph depicted in Figure 5 is an mda-hypergraph. Further positive examples of mda-hypergraphs can be found in Figure 6.

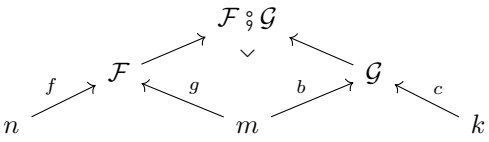
Since we intend to use mda-hypergraphs to represent morphisms, it is not surprising that these hypergraphs can be composed in various ways. Parallel composition is easiest.

▶ **Definition 8.** *If $n \xrightarrow{f} \mathcal{F} \xleftarrow{g} m$ and $h \xrightarrow{b} \mathcal{G} \xleftarrow{c} k$ are compatible mda-hypergraphs in Σ, their* parallel composition *$\mathcal{F} \boxtimes \mathcal{G}$ is the mda-hypergraph $n+h \xrightarrow{f+b} \mathcal{F}+\mathcal{G} \xleftarrow{g+c} m+k$. The ordering on $n+h$ has all the elements of n before all the elements of h and the given orderings within n and h. Similarly for $m+k$.*

▶ **Lemma 9.** *Suppose $\mathcal{F}$ and $\mathcal{G}$ are mda-hypergraphs in Σ. If the source and target objects of $\mathcal{F}$ are X and Y and the source and targets of $\mathcal{G}$ are Z and W, then the source and target of $\mathcal{F} \boxtimes \mathcal{G}$ are $X \otimes Z$ and $Y \otimes W$.*

Mda-hypergraphs can also be composed in sequence using pushouts.

▶ **Definition 10.** *If $n \xrightarrow{f} \mathcal{F} \xleftarrow{g} m$ and $m \xrightarrow{b} \mathcal{G} \xleftarrow{c} k$ are mda-hypergraphs in $\mathcal{C}$, their* sequential composition *$\mathcal{F} \fatsemi \mathcal{G}$ is an mda-hypergraph formed by the pushout*



In more detail, this pushout is the quotient of disjoint union of the hypergraphs $\mathcal{F}$ and $\mathcal{G}$ where for each $v \in V_m$, we identify $g(v)$ in the copy of $\mathcal{F}$ with $b(v)$ in the copy of $\mathcal{G}$. The monogamy conditions ensure that after the output vertices of $\mathcal{F}$ are identified with the input vertices of $\mathcal{G}$, the resulting hypergraph again has the monogamy property.

▶ **Lemma 11.** *Suppose $\mathcal{F}$ and $\mathcal{G}$ are mda-hypergraphs in a signature Σ. If the source and target objects of $\mathcal{F}$ are X and Y and the source and targets of $\mathcal{G}$ are Y and Z, then the source and target of $\mathcal{F} \fatsemi \mathcal{G}$ are X and Z.*

When considering non-freely generated monoidal categories, the labels of hypergraphs may involve tensor products of objects. In such cases, hypergraphs with a common object may not have composable interfaces. For this, an adapter hypergraph can be used.

▶ **Definition 12.** *Suppose n and m are two discrete ordered hypergraphs with the property that $\widetilde{n} = \widetilde{m}$. The n,m-*adapter *is the mda-hypergraph with a single edge e labelled 1_A and vertices $n + m$ with $s(e) = n$, $t(e) = m$ with vertex labels matching the corresponding vertices in n and m.*

This solves the wire-matching problem in non-freely generated monoidal categories, but we continue to assume our categories are generated from a signature for clarity.

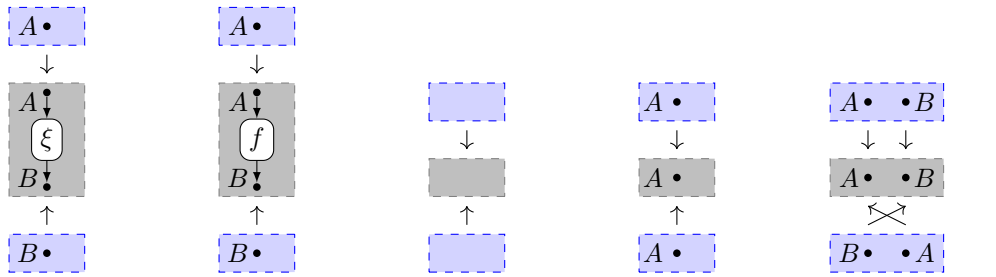
3.3 Mda-hypergraphs of morphism terms

Now we can define an mda-hypergraph interpretation for every morphism term.

▶ **Definition 13.** *Suppose τ is a morphism term in the signature (Σ_O, Σ_M) with context (Σ_A, Γ). The* mda-hypergraph interpretation of τ *is denoted $\widetilde{\tau}$ and is defined by induction.*

- *$\widetilde{\xi}$ where $\xi :_t A \to B$ is a constant is as in Figure 6a,*
- *$\widetilde{f}$ where $f :_\Gamma A \to B$ is a variable is as in Figure 6b,*
- *$\widetilde{1_I}$, $\widetilde{1_A}$, and $\widetilde{\sigma_{A,B}}$ are as in Figure 6c-6e,*
- *$\widetilde{\tau_1 \otimes \tau_2} = \widetilde{\tau_1} \boxtimes \widetilde{\tau_2}$, and $\widetilde{\tau_1 ; \tau_2} = \widetilde{\tau_1} \fatsemi \widetilde{\tau_2}$.*

If $\mathcal{F}$ is an mda-hypergraph of a ground term (i.e. without variables), then we call it a ground mda-hypergraph.



(a) Morphism $\widetilde{\xi}$. **(b)** Variable $\widetilde{f}$. **(c)** Identity $\widetilde{1_I}$. **(d)** Identity $\widetilde{1_A}$. **(e)** Symmetry $\widetilde{\sigma_{A,B}}$.

Figure 6 Interpretation of morphism terms as mda-hypergraphs.

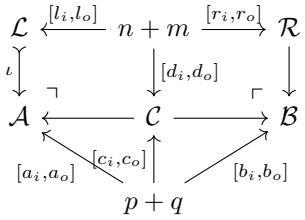
▶ **Lemma 14.** *If $\alpha =_{SMC} \beta$, then $\widetilde{\alpha} = \widetilde{\beta}$. Consequently, there is a strict symmetric monoidal functor $[\![-]\!]$ from S_Σ to Σ-labelled mda-hypergraphs.*

3.4 Double pushout rewriting for mda-hypergraphs

String diagrams and morphism terms both support rewriting systems to facilitate equational reasoning. Hypergraphs have a similar rewriting system based on *double-pushout rewriting*. We recall this for the case of mda-hypergraphs next.

▶ **Definition 15.** *An* mda rewrite rule *is a parallel pair of mda-hypergraphs $\mathcal{L}$ and $\mathcal{R}$ with interfaces n and m. An* mda rewrite system *is a set of rewrite rules. We say that the mda-hypergraph $\mathcal{A}$ with interfaces p and q* rewrites directly *to the parallel hypergraph $\mathcal{B}$ if*

- *there is a mono* $\iota : \mathcal{L} \to \mathcal{A}$ *whose image is a convex subgraph of* $\mathcal{A}$,
- *there is an mda-hypergraph* $p + m \xrightarrow{[c_i,d_o]} \mathcal{C} \xleftarrow{[c_o,d_i]} q + n$, *and*
- *the diagram below commutes and the two marked squares are pushouts.*



As an example, consider an (instance of an) equation for a final map: $f;\omega_B = \omega_A$. An mda-rewrite turning $\widetilde{f;\omega_B}$ into $\widetilde{\omega_A}$ in $\delta_A; \widetilde{g \otimes (f;\omega_B)}$ is shown in Figure 7.

The mda-hypergraph $\mathcal{C}$ is $\mathcal{A}$ with the image of $\mathcal{L}$ deleted (except $\mathcal{L}$'s interface vertices). Inputs to $\mathcal{L}$ are then outputs of $\mathcal{C}$ and similarly outputs of $\mathcal{L}$ are inputs to $\mathcal{C}$. The explains the peculiar mixing of inputs and outputs in the cospan $p + m \xrightarrow{[c_i,d_o]} \mathcal{C} \xleftarrow{[c_o,d_i]} q + n$. In [5, Definition 23], $\mathcal{C}$ is called a *boundary complement*, a strengthening of the notion of *pushout complement*.

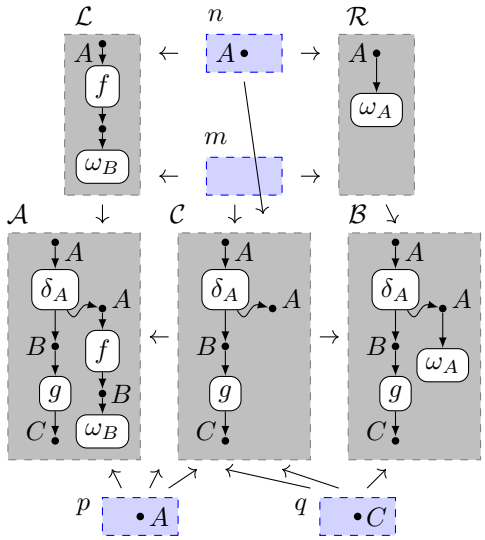


Figure 7 Example rewrite step.

The connection between string diagram rewriting and (convex) double-pushout rewriting of ground mda-hypergraphs is thoroughly examined in [7, 5, 6]. As long as the convex embedding $\iota : \mathcal{L} \to \mathcal{A}$ exists, this DPO rewriting step can be completed uniquely (up to isomorphism).

Connecting term rewriting to double-pushout rewriting is similiarly possible, and requires only a treatment of substitution. If f is a variable in the Σ-term α, then $\widetilde{f}$ is an edge in $\widetilde{\alpha}$. Since this is a convex subgraph of $\widetilde{\alpha}$, we can use DPO rewriting to replace an edge labelled f with a parallel hypergraph $\widetilde{\beta}$. The substitution $\widetilde{\alpha[\beta/f]}$ is then formed by subtituting all edges labelled f in $\widetilde{\alpha}$ with $\widetilde{\beta}$ in this manner. This allows the substitutions required both to find substitution instances of rewrite rules and to create contexts in rewrite steps in DPO systems.

4 Hierarchical string diagrams

In this section, we embark on a similar project for hierarchical string diagrams for monoidal closed categories. We devise a suitable combinatorial structure, called *hypernets*, and show that two closed morphism terms are interpreted as the same hypernet whenever they are equivalent modulo the laws of symmetric monoidal categories (Proposition 24). This allows us to conclude that rewriting can be "implemented" as double-pushout rewriting of hypernets (Proposition 26).

Hierarchical hypergraphs have been used before, see e.g. [10, 15, 31]. Our approach is broadly similar, but with enough subtle differences that it is necessary to give our own definitions. For a more detailed comparison, see section 6.2.

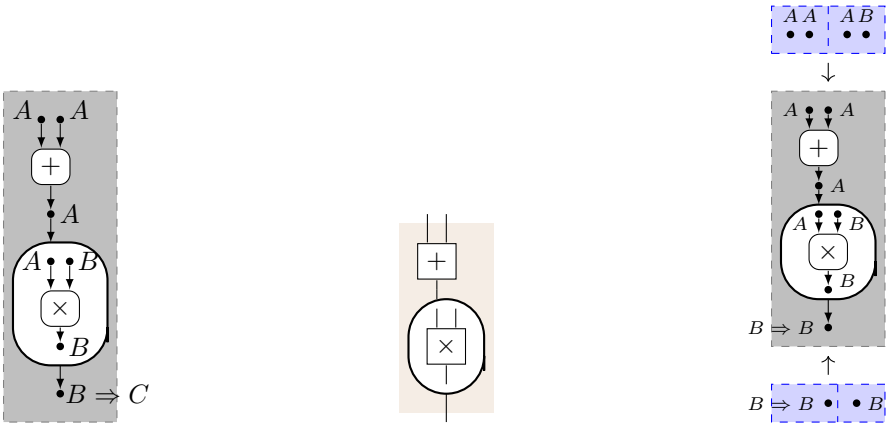
4.1 Hierarchical hypergraphs and hypernets

A hierarchical hypergraph is a hypergraph with an extra parent relationship determining the hierarchical structure.

▶ **Definition 16.** *A hierarchical hypergraph is a tuple* $(V, E, s, t, \ell_V, \ell_E, p_V, p_E)$ *where* V, E, s, t, *and* ℓ_V *are as in a hypergraph, the edge labelling is modified to include an extra value* $\ell_E : E \to \Sigma_E + 1$, *and parent functions* $p_V : V \to E + 1$ *and* $p_E : E \to E + 1$.

The parent functions satisfy some conditions. First, 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. Second, the parent relation must be acyclic. More precisely, we assume for all* $e \in E$ *there is some* $k \geq 1$ *such that* $(p_{E,\perp})^k(e) = \perp$ *where* $\perp$ *is the element of 1 and* $p_{E,\perp} : E + 1 \to E + 1$ *is the extension of* p_E *adding* $p_{E,\perp}(\perp) = \perp$.

When the parent of a vertex or edge is the element $\perp$ from the right summand, we say it is an *outermost vertex* or *outermost edge*. If the label of an edge is $\perp$, we say (with some abuse) that it is *unlabelled*. When considering multiple hierarchical hypergraphs, we use subscripts to disambiguate these data.



(a) Hierarchical hypergraph. **(b)** Hierarchical string diagram. **(c)** HHG with interfaces.

Figure 8 Example hierarchical structures.

In every hierarchical hypergraph $\mathcal{F}$, associated to every edge $\hat{e}$ is a subgraph, namely the subgraph of edges e (and vertices v) satisfying $p^k_{E,\perp}(e) = \hat{e}$ (and $(p_{E,\perp})^j(p_V(v)) = \hat{e}$) for some $k \geq 1$ (and $j \geq 0$). We denote this subgraph $\mathcal{F}_{\hat{e}}$ and call it "the inner hypergraph of $\hat{e}$".

When depicting a hierarchical hypergraph, we indicate the inner hypergraph of an edge by nesting the inner subgraph within its edge, like abstraction in hierarchical string diagrams. If a subgraph $\mathcal{G}$ of a hierarchical hypergraph $\mathcal{F}$ has the property that $\mathcal{F}_{\hat{e}} \subseteq \mathcal{G}$ for all $\hat{e} \in E_{\mathcal{G}}$, we call $\mathcal{G}$ *down-closed*.

An example hierarchical hypergraph is shown in Figure 8a. This hierarchical hypergraph has 3 edges, labelled $+$, $\times$, and one unlabelled. The unlabelled edge is the parent of the edge labelled $\times$ (and its sources and targets), and so the $\times$ edge is depicted inside. This notation echoes the bubble operation in hierarchical string diagrams. The corresponding hierarchical string diagram for this hypergraph is depicted in Figure 8b.

▶ **Definition 17.** *A* morphism of hierarchical hypergraphs $\phi : \mathcal{F} \to \mathcal{G}$ *is a pair of functions* $\phi_V : V_{\mathcal{F}} \to V_{\mathcal{G}}$ *and* $\phi_E : E_{\mathcal{F}} \to E_{\mathcal{G}}$*, which is a morphism of the underlying hypergraphs and respects the hierarchial structure in the following sense:*

1. $(p_{V,\mathcal{F}}; \phi_E)(v) = (\phi_V; p_{V,\mathcal{G}})(v)$ *if* $p_{V,\mathcal{F}}(v) \in E_{\mathcal{F}}$
2. $(p_{E,\mathcal{F}}; \phi_E)(e) = (\phi_E; p_{E,\mathcal{G}})(e)$ *if* $p_{E,\mathcal{F}}(e) \in E_{\mathcal{F}}$

Note that we do not require that outermost vertices and edges are sent to outermost vertices and edges. If ϕ_V and ϕ_E additionally satisfy the property that $p_{V,\mathcal{G}}(\phi_V(v)) = \bot$ and $p_{E,\mathcal{G}}(\phi_E(e)) = \bot$ whenever $p_{V,\mathcal{F}}(v) = \bot$ and $p_{E,\mathcal{G}}(e) = \bot$, then the morphism is *strict*.

Hierarchical hypergraphs and the morphisms between them form a category. A hierarchical hypergraph *with interfaces* is a cospan in this category, $n \xrightarrow{f} \mathcal{H} \xleftarrow{g} m$ with n and m discrete and ordered. The *input interface of the edge* $e \in E_{\mathcal{H}}$ is the subgraph of vertices v of n such that $(f; p_{V,\mathcal{H}})(v) = e$. Similarly, the *output interface of the edge* e is the subgraph of m satsifying $(g; p_{V,\mathcal{H}})(v) = e$. The *outermost input and output interfaces* are the subgraphs of the interfaces whose image has parent $\bot$.

When labels from a hierarchical hypergraph are drawn from a signature, we define the input and output **objects** for each of these interfaces as in Definition 6: the list of labels of the vertices in the interface in the same order as the vertices.

An example hierarchical hypergraph with interfaces is shown in Figure 8c. Note that the input interface has the B-labelled source vertex of the $\times$ edge in its image, and the output interface similarly includes the B-labelled target vertex. These are *internal interfaces* for the hierarchical hypergraph, since they are not outermost vertices. We separate the outermost interface and internal interfaces in our graphical depiction with a vertical line.

▶ **Definition 18.** *Suppose* $n \xrightarrow{f} \mathcal{H} \xleftarrow{g} m$ *is a* Σ*-labelled hierarchical hypergraph with interfaces. Let* $e \in E_{\mathcal{H}}$ *be an edge with* $\ell_E(e) = \bot$*, let* P *be the input object for the input interface of* e*, and let* C *be the output object for the output interface of* e*. We say* e *is a* well-typed abstraction *if there is an object* B *such that* $\widetilde{s(e)} \otimes B = P$ *and* $\widetilde{t(e)} = B \Rightarrow C$*.*

Note also the difference between the "input/output object of an edge", which considers the edge's sources and targets, and the "input/output object for the interface of an edge", which considers the dangling vertices whose parent is that edge.

As examples, the unlabelled edge in Figure 8a is not a well-typed abstraction, assuming A, B and C are distinct generators. Its input, output, input interface, and output interface objects are A, $B \Rightarrow C$, $A \otimes B$, and B, respectively. There is no object X such that $A \otimes X = A \otimes B$ and $B \Rightarrow C = X \Rightarrow B$, so it is not a well-typed abstraction. On the other hand, the unlabelled edge in Figure 8c is a well-typed abstraction: its output object has been changed to $B \Rightarrow B$, so clearly $X = B$ is an object satisfying the necessary conditions.

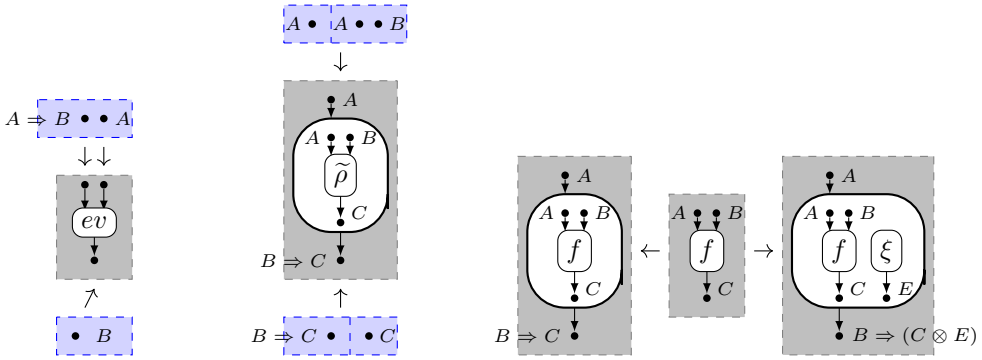
Hierarchical hypergraphs satisfying this well-typedness condition are our formal model for hierarchical string diagrams. We call members of this restricted class *hypernets*.

▶ **Definition 19.** *A* hypernet *is a hierarchical hypergraph $n \to \mathcal{H} \leftarrow m$ with interfaces such that (1) it is an mda-hypergraph when the hierarchical structure is forgotten, (2) if $\ell_E(e) \neq \bot$, then $\mathcal{H}_e$ is the empty hypergraph, and (3) if $\ell_E(e) = \bot$, then e is a well-typed abstraction.*

If a hierarchical hypergraph only satisfies properties (1) and (2) only, we call it a weak hypernet.

Hypernets can be composed in parallel just like mda-hypergraphs with interfaces. They can be composed in sequence anytime their *outermost* interfaces match again using a pushout.

▶ **Definition 20.** *Suppose ρ is a closed morphism term in a signature Σ. The* hypernet interpretation of *ρ is denoted $\widetilde{\rho}$, and again defined by induction on ρ. The cases shared in common with morphism terms are as in Definition 13. The two new cases are ev and $\Lambda(\rho)$, which are defined as in Figure 9a and 9b.*



(a) Evaluation $\widetilde{ev_{A,B}}$. **(b)** Abstraction $\widetilde{\Lambda(\rho)}$. **(c)** Failure of pushouts.

Figure 9 Hypernet interpretations and pushouts.

4.2 Pushout rewriting of hypernets

We next consider pushouts in order to support double pushout rewriting. When allowing all hierarchical hypergraph morphisms, the category of hierarchical hypergraphs does not have all pushouts or even pushouts along monos. This is due to ambiguities in the parents of outermost vertices and edges. Two non-strict morphisms can embed a graph into two unmergeable parts of different graphs. As an example, the cospan of Figure 9c does not have a pushout, even if we allow non-well-typed abstractions.

Consequently, much of the advanced categorical structure of hypergraphs which have historically proven useful, such as the fact they form an adhesive category, cannot be used when studying hierarchical hypergraphs or hypernets. However, the pushouts we need to exist still exist.

As a running example, we will consider a hypernet rewriting rule corresponding to the slide rule of Figure 3c. The hypernets corresponding to each of these terms are shown in Figure 10. The most obvious difference between this span and a rewriting span for mda-hypergraphs is that the interfaces for the left and right legs do not exactly match! In fact, we only need the outermost interfaces to match: the interior interfaces are only important for enforcing the well-typedness of abstractions.

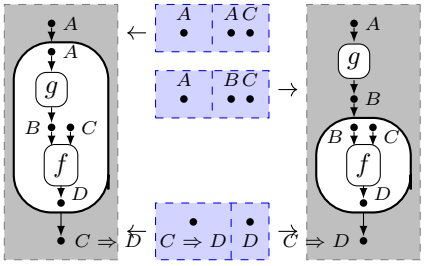
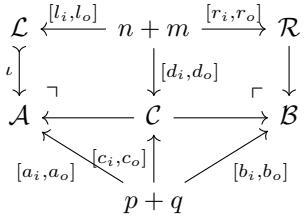


Figure 10 A hypernet span for the slide rule.

We have generally been using terms for hypergraphs also for hierarchical hypergraphs. However, we must refine two important defintions for rewriting hypernets. We say that two hierarchical hypergraphs are *parallel* if they have the same *outermost* input and output interfaces. We say a subgraph is *convex* (in a larger hierarchical hypergraph) if it is convex (as a hypergraph) **and** down-closed.

▶ **Definition 21.** *A* hypernet rewrite rule *is a parallel pair of hypernets* $\mathcal{L}$ *and* $\mathcal{R}$ *with outermost interfaces* n *and* m*. We say that the hypernet* $\mathcal{A}$ *with interfaces* $\mathbf{p}$ *and* $\mathbf{q}$ *and outermost interfaces* p *and* q rewrites directly *to the parallel hypernet* $\mathcal{B}$ *if*

- *there is a mono* $\iota : \mathcal{L} \to \mathcal{A}$ *whose image is a convex subgraph of* $\mathcal{A}$*,*
- *there is a weak hypernet* $\mathbf{p} + m \xrightarrow{[\mathbf{c_i}, d_o]} \mathcal{C} \xleftarrow{[\mathbf{c_o}, d_i]} \mathbf{q} + n$*, and*
- *the diagram below commutes and the two marked squares are pushouts.*



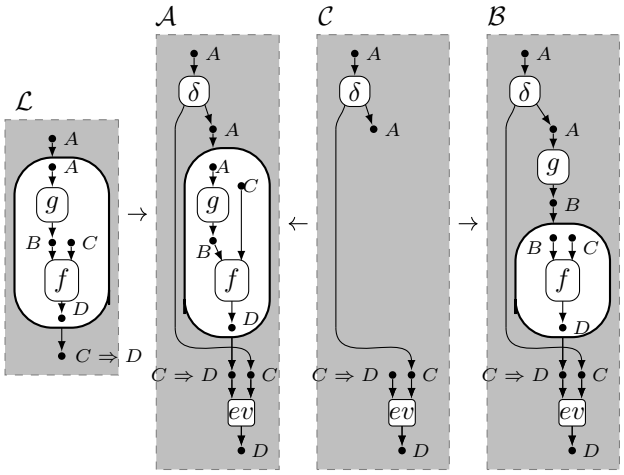


Figure 11 A rewriting span.

The primary difference between hypernet rewriting and mda-hypergraph rewriting (Definition 15) is that the diagram for hypergraph rewriting only places conditions on the outermost interfaces, rather than the entire interface.

Next we give an example application of the hypernet rewrite rule of Figure 10. The leftmost morphism in Figure 11 is a convex embedding of the leftmost hypernet in the rewrite rule, $\mathcal{L}$, into a larger hypernet, $\mathcal{A}$. The next hypernet, $\mathcal{C}$, is the pushout complement of $\mathcal{L}$ in $\mathcal{A}$. Note that the outermost interface of $\mathcal{C}$ consists of the outermost interface of $\mathcal{A}$ together with the outermost interface of $\mathcal{L}$. This occurs exactly when the embedding morphism of the left side of the rule is strict (sends outermost to outermost). If this matching were not strict (sending $\mathcal{L}$ to an interior hypergraph), the outermost interface of $\mathcal{L}$ would still be part of the interface of $\mathcal{C}$, but would be an internal interface. Note also that internal interfaces in $\mathcal{L}$ are not part of the interface of $\mathcal{C}$, since all of $\mathcal{L}$ (except its interface vertices) are deleted in the pushout complement. Finally, $\mathcal{B}$ is the pushout of $\mathcal{C}$ with $\mathcal{R}$ along their common interfaces.

▶ **Proposition 22.** *Suppose $\langle \mathcal{L}, \mathcal{R} \rangle$ is a hypernet rewriting rule, $\mathcal{A}$ is a hypernet, and $\iota : \mathcal{L} \to \mathcal{A}$ is a mono with a convex image in $\mathcal{A}$. Then the boundary complement $\mathcal{C}$ of Definition 21 exists, and further the pushout $\mathcal{B}$ is a hypernet.*

Proof (Idea). The boundary complement is constructed as in hypergraphs. The unusual part is that the boundary complement can fail to be a hypernet. This happens precisely when the embedding ι is not strict. In this case, new interfaces are created inside an edge; that edge fails to be a well-typed abstraction so $\mathcal{C}$ will only be a weak hypernet. Fortunately, takng the pushout with a hypernet $\mathcal{R}$ with the same outermost interface as $\mathcal{L}$ restores the well-typedness of the abstraction. ◀

5 Soundness and completeness

To describe the connection between term rewriting and hypernet rewriting, we first note that the hypernets corresponding to the subterms of a term are always convex subgraphs of the hypernet for the full term. The converse is not generally true: $\widetilde{f \otimes h}$ is a convex subgraph of $\widetilde{f \otimes g \otimes h}$, but $f \otimes h$ is not a subterm of $f \otimes g \otimes h$, regardless of how the latter is constructed. Note, however, that there is a term equivalent to $f \otimes g \otimes h$ under SMC equations, namely $1 \otimes \sigma; f \otimes h \otimes g; 1 \otimes \sigma$, for which $f \otimes h$ does appear. This motivates our next definition and result.

▶ **Definition 23.** *A closed morphism term ρ is a* possible subterm *of another closed morphism term α if there is a closed morphism term $\beta =_{SMC} \alpha$ such that ρ is a subterm of β.*

▶ **Proposition 24.** *For every closed morphism term ρ, possible subterms of ρ and (isomorphism classes of) convex sub-hypernets of $\widetilde{\rho}$ are in bijective correspondence.*

In particular, this implies that every hypernet has exactly one SMC equivalence class of terms representing it.

The next critical notion is that of *substitution*. In terms, substitution replaces all instances of a variable in a term with another term. Hypernet rewriting, however, may not be able to mimic term substitution as a *single* rewrite if the subgraph of edges labelled with this variable do not form a convex subgraph. However, it can always be accomplished in several steps.

▶ **Lemma 25.** *Suppose α and β are closed morphism terms and f is a variable. There is a finite sequence of hypergraphs $\mathcal{H}_0, \ldots, \mathcal{H}_n$ such that $\widetilde{\alpha} = \mathcal{H}_0$, $\widetilde{\alpha[\beta/f]} = \mathcal{H}_n$ and $\mathcal{H}_i$ directly rewrites to $\mathcal{H}_{i+1}$ under the rule $\langle \widetilde{f}, \widetilde{\beta} \rangle$.*

In such a case, we say $\mathcal{H}_n$ is a *substitution instance* of $\mathcal{H}_0$. Finally, we can formalize the correspondence between term rewriting and hypernet rewriting.

▶ **Proposition 26.** *Suppose α, β, λ, and ρ are closed morphism terms such that $\alpha \to_{\langle\lambda,\rho\rangle} \beta$. There are hypernets $\mathcal{L}$ and $\mathcal{R}$, substitution instances of $\widetilde{\lambda}$ and $\widetilde{\rho}$ respectively, such that $\widetilde{\alpha} \Rightarrow_{\langle\mathcal{L},\mathcal{R}\rangle} \widetilde{\beta}$.*

▶ **Proposition 27.** *Suppose α, β, λ, and ρ are closed morphism terms such that $\widetilde{\alpha} \Rightarrow_{\langle\widetilde{\lambda},\widetilde{\rho}\rangle} \widetilde{\beta}$. There are closed morphism terms γ, δ such that $\alpha =_{SMC} \gamma$ and $\beta =_{SMC} \delta$ and $\gamma \to_{\langle\lambda,\rho\rangle} \delta$.*

These propositions, taken together, show an equivalence of expressiveness: every step in one rewriting system can be accomplished in the other (though potentially in many steps). Term rewriting has the advantage that substitution instances can be formed in a single operation, where imitating this in hypernets requires each instance of the variable be replaced individually. However, hypernet rewriting has the advantage that it is able to replace *possible subterms* from a context, where term rewriting may need to do SMC rewriting steps first to realize the possible subterm as an actual subterm before replacing.

6 Related work

6.1 String diagrams

Monoidal closed categories have been thoroughly studied in the context of logic and type theory, because of the well-known correspondence of their internal language with (linear) simply typed λ-calculus and linear logic [35, 19].

To the best of our knowledge, we provide the first fully specified string diagrammatic language for closed categories. Our approach shares similarities with the formalisms of sharing graphs for describing λ-calculus computations [25]. The main difference is that string diagrams, albeit graphical in appearance, can be manipulated as a syntax, whereas sharing graphs are usually studied as combinatorial objects. Unlike syntax, reasoning about graphs algebraically requires a higher degree of technical sophistication [20]. Finally, sharing graphs are typically used to study low-level computational models for functional languages, in particular quantitative models [29], whereas our approach is more focussed on equational reasoning and rewriting, and does not have the ambition of investigating the resources employed during computation.

Monoidal closed categories also extend to $\star$-autonomous categories. These are relevant to the study of multiplicative linear logic and have been extensively studied in terms of proof nets. Our graphical calculus is essentially different from proof nets. The grammar of morphisms does not stem from a sequent calculus, and we capture the intended semantics via equations rather than a correctness criterion. But the connection might be made precise relying on the existing translations between proof nets and string diagrams [22, 34]. Finally, a different style of hierarchical string diagrams appear in the literature to represent universal properties graphically such as Kan extensions [21] and free monads [32].

The only other proposal for a string-diagram language for monoidal closed categories which we are aware of is that of [3]. To keep the language of types as simple as possible and as *strict* as possible they propose an intriguing graphical innovation, a so-called *clasp* operator on stems. The exponential type is represented using the clasp, and much like in our own language, a *bubble* is used to represent currying.

Although not presented explicitly as a string diagram language, the treatment of closures in [33] is related in methodology to our work, although the setting of *partially-traced partially-closed premonoidal categories* is significantly different to ours.

6.2 Hierarchical hypergraphs and rewriting

The notion of hierarchical hypergraph used in this paper is inspired by and a formalisation of the graphs used in [18].

Although there is no consensus on a standard definition of hierarchical graphs, the various approaches to these structures [10, 15, 31] give slight variations on the idea of graphs containing other graphs and notions of morphisms between them. Some of the differences are minor – ours are directed, others [15] are not – but other differences are fundamental. Sometimes edges are permitted to connect vertices with different parents, as in [31, 10], sometimes this is prohibited (as it is here). Some approaches consider only strict morphisms, but others relax the notion of morphism. Due to the subtle but technically significant differences between our requirements and the properties of previous works, it was not possible to reuse previous work wholesale, and we found it necessary to introduce our own variation.

The formal correspondence between monoidal closed categories and hierarchical hypergraphs lies in a tradition of analogous results relating string diagram rewriting and double-pushout hypergraph rewriting, see [7, 5, 6]. To the best of our knowledge, such correspondence has not been spelled out in the way presented in our work, although the idea of linking the exponential structure of closed categories with the hierarchy structure of hierarchical hypergraphs may be found in [13]. Although it does not uses string diagrams or other categorical tools, the algebraic specification language for hierarchical graphs studied in [9] is aiming towards similar goals. Representing intermediate stages of the compiler as graphs is a long-established practice in compiler design and engineering. Graphs are an *efficient* syntactic representation which are recognised as a better target for optimisation and analysis than raw text. In its simplest incarnation the graph representation of terms is just an *abstract syntax tree*, but more sophisticated representations were increasingly used [12], sometimes leading to specific and novel optimisation techniques [30].

The use of graph-like representation outside of compiler engineering has a lot of untapped potential, as advocated by some [16]. This is not entirely new, for example interaction nets are a graph-like semantics of higher-order computation [24], albeit highly informal.

We would also like to point out recent work on $\mathcal{M},\mathcal{N}$-adhesivity that can be used with hierarchical graphs [11]. Though our category of hierarchical hypergraphs is not adhesive, it is possible that choosing a different family of morphisms could make it $\mathcal{M},\mathcal{N}$-adhesive. For example, we believe this category with strict morphisms only is adhesive.

Finally, another related line of work which we found inspirational is the use of graph-like languages inspired by proof nets to bridge the gap between syntax and abstract machines, in order to provide a quantitative analysis of reduction strategies for the lambda calculus [1].

7 Conclusion and further work

In this paper, we have presented a hypergraph-based formalism for representing string diagrams of monoidal closed categories. Our approach is based on interpreting morphism terms as hypernets and comparing their rewriting systems. This makes it easy to express universally quantified equations while retaining a simple redex search. An alternative approach, closer in spirit to [7, 5, 6], would be to interpret the morphisms of the (closed) monoidal category instead. This would make the interpretation functorial and has further computational complexity benefits; we plan to develop this connection between hypernet and hierarchical string diagram rewriting in further work.

References

1 Beniamino Accattoli. Proof nets and the call-by-value λ-calculus. *Theor. Comput. Sci.*, 606:2–24, 2015. doi:10.1016/j.tcs.2015.08.006.

2 Mario Alvarez-Picallo, Dan R Ghica, David Sprunger, and Fabio Zanasi. Functorial string diagrams for reverse-mode automatic differentiation. *arXiv preprint arXiv:2107.13433*, 2021.

3 John Baez and Mike Stay. Physics, topology, logic and computation: a rosetta stone. In *New structures for physics*, pages 95–172. Springer, 2010. doi:10.1007/978-3-642-12821-9_2.

4 Guillaume Boisseau and Paweł Sobociński. String diagrammatic electrical circuit theory. *arXiv preprint arXiv:2106.07763*, 2021.

5 Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory II: rewriting with symmetric monoidal structure. *CoRR*, abs/2104.14686, 2021.

6 Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory III: confluence with and without Frobenius. *Mathematical Structures in Computer Science*, to appear, 2021.

7 Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory I: rewriting with frobenius structure. *J. ACM*, 69(2):14:1–14:58, 2022.

8 Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. Full abstraction for signal flow graphs. *ACM SIGPLAN Notices*, 50(1):515–526, 2015.

9 Roberto Bruni, Fabio Gadducci, and Alberto Lluch-Lafuente. An algebra of hierarchical graphs. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Trustworthly Global Computing - 5th International Symposium, TGC 2010, Munich, Germany, February 24-26, 2010, Revised Selected Papers*, volume 6084 of *Lecture Notes in Computer Science*, pages 205–221. Springer, 2010. doi:10.1007/978-3-642-15640-3_14.

10 Roberto Bruni, Fabio Gadducci, and Alberto Lluch-Lafuente. An algebra of hierarchical graphs and its application to structural encoding. *Sci. Ann. Comput. Sci.*, 20:53–96, 2010. URL: http://www.info.uaic.ro/bin/Annals/Article?v=XX&a=2.

11 Davide Castelnovo, Fabio Gadducci, and Marino Miculan. A new criterion for $\mathcal{M},\mathcal{N}$-adhesivity, with an application to hierarchical graphs, 2022. doi:10.48550/ARXIV.2201.00233.

12 Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In Michael D. Ernst, editor, *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, pages 35–49. ACM, 1995. doi:10.1145/202529.202534.

13 Matteo Coccia, Fabio Gadducci, and Ugo Montanari. Gs.lambda theories: A syntax for higher-order graphs. In Richard Blute and Peter Selinger, editors, *Category Theory and Computer Science, CTCS 2002, Ottawa, Canada, August 15-17, 2002*, volume 69 of *Electronic Notes in Theoretical Computer Science*, pages 83–100. Elsevier, 2002. doi:10.1016/S1571-0661(04)80560-7.

14 Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 298–310, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

15 Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *J. Comput. Syst. Sci.*, 64(2):249–283, 2002. doi:10.1006/jcss.2001.1790.

16 Dan R. Ghica. Operational semantics with hierarchical abstract syntax graphs. In Patrick Bahr, editor, *Proceedings 11th International Workshop on Computing with Terms and Graphs, TERMGRAPH@FSCD 2020, Online, 5th July 2020*, volume 334 of *EPTCS*, pages 1–10, 2020. doi:10.4204/EPTCS.334.1.

17 Dan R. Ghica, Achim Jung, and Aliaume Lopez. Diagrammatic semantics for digital circuits. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden*, volume 82 of *LIPIcs*, pages

24:1–24:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CSL.2017.24.

18 Dan R. Ghica, Koko Muroya, and Todd Waugh Ambridge. Local reasoning for robust observational equivalence. *CoRR*, abs/1907.01257, 2019. arXiv:1907.01257.

19 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

20 Stefano Guerrini. A general theory of sharing graphs. *Theor. Comput. Sci.*, 227(1-2):99–151, 1999. doi:10.1016/S0304-3975(99)00050-X.

21 Ralf Hinze. Kan extensions for program optimisation or: Art and dan explain an old trick. In Jeremy Gibbons and Pablo Nogueira, editors, *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings*, volume 7342 of *Lecture Notes in Computer Science*, pages 324–362. Springer, 2012. doi:10.1007/978-3-642-31113-0_16.

22 Dominic Hughes. Simple free star-autonomous categories and full coherence. *CoRR*, 2005. URL: https://arxiv.org/abs/math/0506521.

23 André Joyal and Ross Street. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112, July 1991. doi:10.1016/0001-8708(91)90003-P.

24 Yves Lafont. Interaction nets. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 95–108. ACM Press, 1990. doi:10.1145/96709.96718.

25 John Lamping. An algorithm for optimal lambda calculus reduction. In Frances E. Allen, editor, *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California, USA, January 1990*, pages 16–30. ACM Press, 1990. doi:10.1145/96709.96711.

26 Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

27 Paul-André Melliès. Functorial boxes in string diagrams. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2006. doi:10.1007/11874683_1.

28 José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, 1990. doi:10.1016/0890-5401(90)90013-8.

29 Koko Muroya and Dan R. Ghica. The dynamic geometry of interaction machine: A token-guided graph rewriter. *Log. Methods Comput. Sci.*, 15(4), 2019. doi:10.23638/LMCS-15(4:7)2019.

30 Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 31–44. ACM, 2020. doi:10.1145/3385412.3386012.

31 Wojciech Palacz. Algebraic hierarchical graph transformation. *J. Comput. Syst. Sci.*, 68(3):497–520, 2004. doi:10.1016/S0022-0000(03)00064-3.

32 Maciej Piróg and Nicolas Wu. String diagrams for free monads (functional pearl). In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 490–501. ACM, 2016. doi:10.1145/2951913.2951947.

33 Ralf Schweimeier and Alan Jeffrey. A categorical and graphical treatment of closure conversion. In Stephen D. Brookes, Achim Jung, Michael W. Mislove, and Andre Scedrov, editors, *Fifteenth Conference on Mathematical Foundations of Progamming Semantics, MFPS 1999, Tulane University, New Orleans, LA, USA, April 28 - May 1, 1999*, volume 20 of *Electronic Notes in Theoretical Computer Science*, pages 481–511. Elsevier, 1999. doi:10.1016/S1571-0661(04)80090-2.

34 Michael Shulman. $*$-autonomous envelopes and 2-conservativity of duals. *CoRR*, 2020. arXiv:2004.08487.

35 Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

36 David Sprunger and Shin-ya Katsumata. Differentiable causal computations via delayed trace. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE, 2019.