The Cost of Compositionality
A High-Performance Implementation of String Diagram Composition

Paul Wilson
University of Southampton
paul@statusfailed.com

Fabio Zanasi
University College London
f.zanasi@ucl.ac.uk

String diagrams are an increasingly popular algebraic language for the analysis of graphical models of computations across different research fields. Whereas string diagrams have been thoroughly studied as semantic structures, much less attention has been given to their algorithmic properties, and efficient implementations of diagrammatic reasoning are almost an unexplored subject.

This work intends to be a contribution in such a direction. We introduce a data structure representing string diagrams in terms of adjacency matrices. This encoding has the key advantage of providing simple and efficient algorithms for composition and tensor product of diagrams. We demonstrate its effectiveness by showing that the complexity of the two operations is linear in the size of string diagrams. Also, as our approach is based on basic linear algebraic operations, we can take advantage of heavily optimised implementations, which we use to measure performances of string diagrammatic operations via several benchmarks.

1 Introduction

String diagrams are a ubiquitous graphical notation for depicting morphisms of a monoidal category, and have been used in a variety of settings— see e.g. [3, 5, 6, 16] and [13] for an overview.

In order to work with string diagrams on a computer, we require a representation of them which we can manipulate. Several such representations have been explored in the literature—see for example the wiring diagrams of Catlab.jl [12], and the hypergraphs of [2] as used in CARTOGRAPHER [15].

However, to support 'industrial scale' uses of string diagrams where modelisations are very large, there is a pressing need to ensure that operations for combining these structures are *efficient*. In this work, we define a string diagram representation inspired by the parallel programming literature (specifically [8, 14]). Our data structure of choice is based on sparse adjacency matrices representing hypergraphs, and thus we call it HAR — hypergraph adjacency representation. We shall show how to encode string diagrams into HARs, using their characterisation as hypergraphs (from [2]) as intermediate steps. The following picture summarises the various steps of the encoding:

String Diagram ⟶ Hypergraph with Interfaces ⟶ Bipartite Graph with Interfaces ⟶ Har

Abstract ⇢ Concrete

The main point of this implementation is that the encoding allows for simple algorithms for composition and tensor product. Composition is especially simplified, being completely expressible in terms of permutation and tensor product operations on matrices (see Definition 4.1 below).

Furthermore, the algorithms we describe are completely in terms of linear-algebraic operations on matrices. Since highly optimised implementations of such operations are widely available, this makes our approach straightforward to implement while providing good performance. Additionally, since implementations of linear algebra routines are also available for specialised parallel hardware such as GPUs, our algorithms require little additional effort to support such settings.

K. Kishida (Ed.): Fourth International Conference
on Applied Category Theory (ACT 2021).
EPTCS 372, 2022, pp. 262–275, doi:10.4204/EPTCS.372.19

Importantly, we also show that the operations of composition and tensor product for our representation have linear complexity, a fact which we support with empirical validation on synthetic benchmarks.

We summarise our main contributions as follows:

- An isomorphic representation of string diagrams in terms of adjacency matrices of certain graphs
- Algorithms for tensoring and composition of string diagrams (via their representation)
- Computational complexity bounds for tensor and composition algorithms
- An empirical analysis of the performance of our approach

The structure of the paper is as follows. In Section 2 we recall the directed hypergraphs of [2] and discuss a bipartite encoding of *undirected* hypergraphs from the parallel processing literature [8]. In Section 3 we formally describe our proposed encoding of hypergraphs in terms of adjacency matrices. We then proceed in Section 4 to describe operations such as composition and tensor product for our encoding, before showing that these form a symmetric monoidal category, which is isomorphic to the original category of string diagrams, in Section 5. Finally, in Section 6 we discuss the complexity of the operations described in Section 4, and show empirical performance results on some synthetic benchmarks in Section 7.

2 Background

2.1 Hypergraphs with Interfaces

Following [2], we will regard string diagrams combinatorially as a certain class of hypergraphs, which we now recall. Throughout this section we fix a *monoidal signature* Σ, that is, a set of operations $o\colon n \to m$, where n is the *arity* and m the *coarity* of o.

Hypergraphs are a generalisation of directed graphs where edges (ordered pairs of vertices) are replaced by *hyper*edges (ordered lists of vertices). As shown in [2], hypergraphs serve as a characterisation of string diagrams over Σ when equipped with the following features: (i) a labeling of hyperedges with Σ-operations; (ii) the identification of a *left* and a *right* interface of the hypergraph; (iii) the restriction to hypergraphs with interfaces that are *monogamous*. We recall the relevant definitions below.

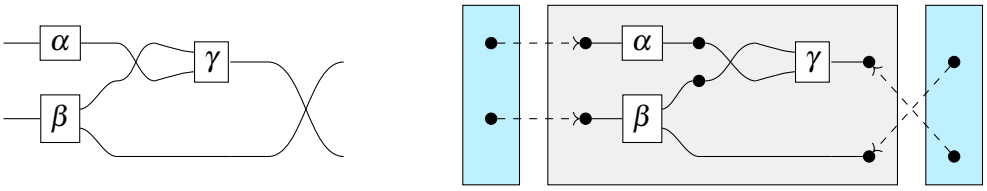
Definition 2.1. *A* ***Σ-labeled (directed) hypergraph*** *H is a triple (V,E,L), where V is a set of* ***nodes****, $E \subseteq \mathsf{List}(V) \times \mathsf{List}(V)$ is a set of* ***hyperedges****, and $L\colon E \to \Sigma$ is a* ***labeling*** *function, where the arity and coarity of $L(e)$ must agree with the length of lists $e \fatsemi \pi_0$ and $e \fatsemi \pi_1$ respectively, for each $e \in E$. A node v is a* ***source*** *of $e \in E$ if it appears in the list $e \fatsemi \pi_0$, and a* ***target*** *if it appears in $e \fatsemi \pi_1$. Σ-labeled hypergraphs with the evident structure-preserving morphisms form a category Hyp_Σ.*

A ***Σ-labeled hypergraph with interfaces*** *is a cospan $n \xrightarrow{f} G \xleftarrow{g} m$ in Hyp_Σ, where n and m are the discrete hypergraphs constiting of n and m nodes respectively. We call $f[n]$ the* ***left interface*** *of G and $g[m]$ the* ***right interface*** *of G. We write $\mathsf{Csp}(\mathsf{Hyp}_\Sigma)_I$ for the PROP[1] whose morphisms $n \to m$ are the hypergraph with interfaces $n \xrightarrow{f} G \xleftarrow{g} m$. Composition is defined by pushout.[2] The notation is due to $\mathsf{Csp}(\mathsf{Hyp}_\Sigma)_I$ being a subcategory of the category of* cospans *in Hyp_Σ.*

[1] PROPs [9] are symmetric monoidal categories with objects the natural numbers. They are widely adopted as a way to express algebraic theories of string diagrams categorically.

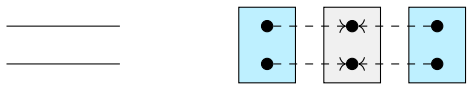
[2] In order for composition to be uniquely defined, strictly speaking morphisms of $\mathsf{Csp}(\mathsf{Hyp}_\Sigma)_I$ should be equivalence classes of hypergraphs with interfaces, where $n \to G \leftarrow m$ and $n \to G' \leftarrow m$ are equivalent when there is an isomorphism $G \to G'$ commuting with the cospan legs. For the sake of simplicity, we shall use representatives of such equivalence classes when working with morphisms of $\mathsf{Csp}(\mathsf{Hyp}_\Sigma)_I$. This does not have any consequence for the theory developed in the rest of the paper.

Σ-labeled hypergraphs with interfaces serve as a faithful interpretation for the PROP Free_Σ whose morphisms are freely generated by the signature Σ [2]. For example, the string diagram $c\colon 2 \to 2$ in Free_Σ as on the left below is interpreted as the hypergraph with interfaces $2 \xrightarrow{f} G \xleftarrow{g} 2$ on the right.



(1)

Note Σ-operations $\alpha\colon 1 \to 1$, $\beta\colon 1 \to 2$ and $\gamma\colon 2 \to 1$ appearing in c are mapped to hyperedges with the appropriate number of source and target nodes, and the 'dangling wires' of c are expressed by the left and right interfaces $2 \xrightarrow{f} G$ and $2 \xrightarrow{g} G$, depicted as dashed arrows. Note also that diagrams consisting of only wires are represented by hypergraphs with no hyperedges. For example, the string diagram and hypergraph representation for id_2 are depicted as follows:



(2)

Although this interpretation is faithful, it is not full — there are hypergraphs not representing any string diagram [2]. One may achieve a full interpretation by restricting to *monogamous acyclic* hypergraphs with interfaces.

Before recalling the definition of monogamous, we need to record a few preliminaries. The *in-degree* (respectively, *out-degree*) of a node v in a hypergraph G is the number of hyperedges having v as target (source). Write $\mathsf{in}(G)$ (*inputs*) for the set of nodes with in-degree 0 and $\mathsf{out}(G)$ (*outputs*) for the set of nodes with out-degree 0.

Definition 2.2. *A hypergraph G is **monogamous acyclic** (ma-hypergraph) if it contains no cycle (acyclicity) and every node has at most in- and out-degree* 1 *(monogamy).*

*A hypergraph with interfaces $n \xrightarrow{f} G \xleftarrow{g} m$ is **monogamous acyclic** when G is an ma-hypergraph, f is a monomorphism and its image is $\mathsf{in}(G)$, g is a monomorphism and its image is $\mathsf{out}(G)$. Ma-hypergraphs with interfaces form a sub-PROP $\mathsf{Csp}(\mathsf{Hyp}_\Sigma)_{MI}$ of $\mathsf{Csp}(\mathsf{Hyp}_\Sigma)_I$.*

Ma-hypergraphs with interfaces are in 1-to-1 correspondence with string diagrams over the same signature, yielding the isomorphism $\mathsf{Free}_\Sigma \cong \mathsf{Csp}(\mathsf{Hyp}_\Sigma)_{MI}$ [2].

2.2 Parallel Hypergraph Processing

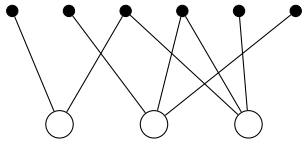
Hypergraphs have many choices of implementation as a data structure. For example, one might choose to model hyperedges directly as pairs of lists. However, the code for such representations must be written from scratch, and can typically be complicated and error-prone. Instead, we would like to take advantage of existing, high performance code for representing graphs, and apply it to hypergraphs. To this end, we take inspiration from the parallel programming literature.

The authors of [8] describe a distributed processing system for *undirected* hypergraphs.

Definition 2.3. *An **undirected hypergraph** U is a pair (V,E) where V is a set of **nodes**, and $E \subseteq \mathscr{P}(V)\setminus\emptyset$ is a set of **hyperedges**.*

Analogously, this definition is a generalisation of the notion of *undirected graphs*: where an edge is an *unordered* pair of vertices, a *hyper*edge is a *set* of vertices.

In order to achieve high performance, the authors define an encoding of their undirected hypergraphs as labeled bipartite graphs. Concretely, vertices are labeled either $\bullet$ or $\circ$, with $\bullet$-vertices playing the role of hypernodes, and $\circ$-vertices playing the role of hyperedges. For example, the bipartite graph below depicts such an encoding, where an edge $\bullet \to \circ$ indicates that the source hypernode appears in the hyperedge set.



(3)

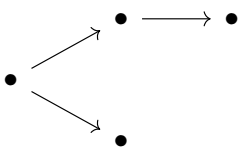
However, since this encoding is specific to the undirected hypergraphs of [8], we must adapt it to suit our purposes.

2.3 PROPs of Matrices

We denote the PROP of matrices over a semiring S by Mat_S, where the tensor product $f \otimes g$ is the **direct sum**, i.e.: $\begin{vmatrix} f & \mathbf{0} \\ \mathbf{0} & g \end{vmatrix}$. In this paper, we will only consider matrices over the semirings of booleans $\mathbb{B}$ and natural numbers $\mathbb{N}$. We denote the $m \times n$ zero matrix as $\mathbf{0}_{n,m}$, dropping the subscripts where unambiguous. We refer to the set of $m \times n$ matrices as $\mathsf{Mat}_S(n,m)$, and note that we always write composition in diagrammatic order: that is, for composition $n \xrightarrow{f} m \xrightarrow{g} l$ we always write $f \fatsemi g$.

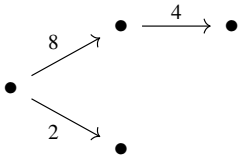
2.4 Adjacency Matrices

The adjacency matrix representation of a graph is central to our representation of hypergraphs, and so we recall it now. The adjacency matrix of a K-node directed graph is a matrix $\mathsf{Mat}_{\mathbb{B}}(K,K)$ where the i^{th} column denotes the *outgoing* edges of the i^{th} node. For example, consider the graph and its adjacency matrix below:



$$\begin{vmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix} \tag{4}$$

Note that in this particular representation, there can be exactly one edge between two nodes of the graph. We can also introduce labeled edges by varying the semiring of Mat: for example, by considering matrices $\mathsf{Mat}_{\mathbb{N}}(K,K)$ we can consider edges to have labels in the set $\{1,2,\ldots\}$, with 0 denoting no edge. Consider for example the same graph as (4) but with labeled edges:

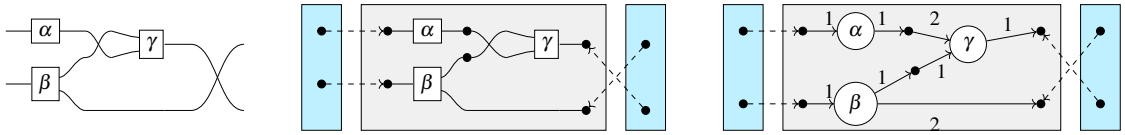


$$\begin{vmatrix} 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \end{vmatrix} \tag{5}$$

3 Hypergraph Adjacency Representation

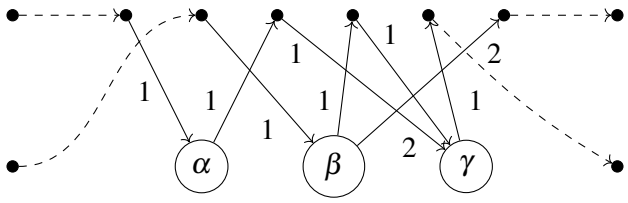
In this section we provide the main technical definition of the paper: the notion of Hypergraph Adjacency Representation (HAR). We begin by providing a roadmap to the formal definition. In a nutshell, the main hurdle is to adapt the approach to undirected hypergraphs in [8] (reported in Section 2.2) to (directed) hypergraphs with interfaces. This will provide us a means of representing hypergraphs with interfaces as bipartite graphs, and thus as the corresponding adjacency matrices. As string diagrams can be identified as a certain class of hypergraphs with interfaces, this methodology will yield an implementation of string diagrams as adjacency matrices.

Before delving in the formal definition, the approach is best illustrated via an example. Recall the string diagram in (1) (below left) with its interpretation as a hypergraph with interfaces (below center). Its bipartite graph encoding is displayed below right.



(6)

Note this is similar to the bipartite graph encoding shown in (3), as made evident when we rearrange the bipartite graph of (6) as follows:



(7)

The differences are (i) the presence of interfaces (needed because we are interested in *composing* these structures), (ii) the labeling of $\circ$-vertices with Σ-operations, and (iii) the labeling of edges with natural numbers. The latter information indicates the position, in the original hypergraph with interfaces, of a node in the source/target lists of a hyperedge. For instance, the edge labeled with 2 indicates that, in the original hypergraph, the target node of hyperedge α is in the second position in the source list of hyperedge γ.

The next step is translating this bipartite graph into an adjacency matrix (along the lines of Section 2.4), together with information on what are the interfaces of the graph. This leads us to the data structure called HAR: a 4-tuple (M,L,R,N), with M serving double-duty as the adjacency matrix and edge-label data, N a vector of node labels, and L and R permutation matrices reordering M so that left boundary nodes are the first and right boundary nodes the last, respectively. Returning to our example hypergraph in (6), we represent it with the following data (in which we write N twice for clarity):

$$M = \begin{vmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \end{vmatrix} \qquad N = \begin{vmatrix} \bullet \\ \bullet \\ \alpha \\ \beta \\ \bullet \\ \bullet \\ \gamma \\ \bullet \\ \bullet \end{vmatrix} \qquad L = \mathsf{id}_9 \qquad R = \begin{vmatrix} \mathsf{id}_7 & \mathbf{0} \\ \mathbf{0} & \sigma_{1,1} \end{vmatrix}$$

$$N = \begin{vmatrix} \bullet & \bullet & \alpha & \beta & \bullet & \bullet & \gamma & \bullet & \bullet \end{vmatrix}$$

We can read the columns of M as the outgoing edges for a particular node. See for example the column for β, which has two outgoing edges labeled 1 and 2, both of which connect to nodes labeled $\bullet$. Note that L is the identity matrix: this means that the left interface nodes appear first, and moreover they appear in the same order as in the interface. Hence, the first 2 rows contain only zeros because the left interface nodes have no incoming edges. On the other hand R is the block matrix $\begin{vmatrix} \mathsf{id} & \mathbf{0} \\ \mathbf{0} & \sigma \end{vmatrix}$ and so while the final two nodes are the right interface nodes, their order in the interface is swapped.

3.1 Main Definition

We can now give our main definition—for background on matrix notation see Section 2.3.

Definition 3.1 (Hypergraph Adjacency Representation)**.** *Fix a monoidal signature Σ. A hypergraph adjacency representation of type $n \to m$ is written $\mathsf{Har}_{n,m}$ and consists of the following data:*

- ***Size*** $K \in \mathbb{N}$
- ***Labeled Adjacency Matrix*** $M \in \mathsf{Mat}_{\mathbb{N}}(K,K)$
- ***Left Permutation*** $L \in \mathsf{Mat}_{\mathbb{B}}(K,K)$
- ***Right Permutation*** $R \in \mathsf{Mat}_{\mathbb{B}}(K,K)$
- ***Node Labels*** $N \in (\{\bullet\} + (\{\circ\} \times \Sigma))^K$

satisfying the following conditions:

- *The graph represented by M is acyclic.*
- *The matrix $L^T \fatsemi M \fatsemi L$ is ordered such that the **first** m nodes are the **left interface nodes***
- *The matrix $R^T \fatsemi M \fatsemi R$ is ordered such that the **last** n nodes are the **right interface nodes**.*
- *If a node labeled $\bullet$ is not an interface node, then it has exactly one incoming and outgoing edge.*
- *For each vertex v labeled $(\circ, g)$ with g having arity/coarity m,n,*
  - *v has incoming edges $e_1 \dots e_m$ with labels $1 \dots m$ respectively.*
  - *v has outgoing edges $e_1 \dots e_n$ with labels $1 \dots n$ respectively.*

3.2 Permutation Equivalence and Boundary Orderings

In Section 4 we will see that composition of Hars is only associative up to isomorphism. Therefore, in order to form a category of Hars, we will quotient by the following equivalence relation,[3] which equates Hars having isomorphic graphs.

Definition 3.2 (Permutation Equivalence)**.** *$f,g : \mathsf{Har}_{n,m}$ are **equivalent up to permutation** P, denoted $f \overset{P}{\sim} g$, when P is a permutation matrix such that the following conditions hold:*

$$g_M = P^T \fatsemi f_M \fatsemi P \qquad g_L = P^T \fatsemi f_L \qquad g_R = P^T \fatsemi f_R \qquad g_N = f_N \fatsemi P$$

Remark 3.3. *Note that this definition ensures that if $f \overset{P}{\sim} g$ then g_M is a graph isomorphic to f_M and also that the interfaces of f and g are the same.*

Proposition 3.4 (Permutation Equivalence is an Equivalence Relation)**.** *Fix some $f,g \in \mathsf{Har}_{n,m}$. Then there exists an equivalence relation denoted $\sim$ such that $f \sim g$ iff there exists some permutation matrix $P \in \mathsf{Mat}_{\mathbb{B}}(K,K)$ such that $f \overset{P}{\sim} g$ (cf. Definition 3.2)*

Proof. Clearly $\sim$ is reflexive because $f \overset{\mathsf{id}}{\sim} f$. Further, it is symmetric because if $f \overset{P}{\sim} g$, then $g \overset{P^T}{\sim} f$. Finally, transitivity follows from matrix composition: If $f \overset{P}{\sim} g$ and $g \overset{Q}{\sim} h$, then $f \overset{P \fatsemi Q}{\sim} h$. □

This definition means that each $f : \mathsf{Har}_{n,m}$ can be put into an equivalent left (resp. right) boundary order by permuting by f_L (resp. f_R). We will make heavy use of these particular permutations in defining composition and tensor product, so we define them explicitly.

Definition 3.5. *The **left boundary order** of $f \in \mathsf{Har}_{n,m}$ is denoted $L(f)$ and has the following data:*

$$L(f)_M = f_L^T \fatsemi f_M \fatsemi f_L \qquad L(f)_L = \mathsf{id}_{f_K} \qquad L(f)_R = f_L^T \fatsemi f_R \qquad L(f)_N = f_N \fatsemi f_L$$

Definition 3.6. *The **right boundary order** of $f \in \mathsf{Har}_{n,m}$ is denoted $R(f)$ and has the following data:*

$$R(f)_M = f_R^T \fatsemi f_M \fatsemi f_R \qquad R(f)_L = f_R^T \fatsemi f_L \qquad R(f)_R = \mathsf{id}_{f_K} \qquad R(f)_N = f_N \fatsemi f_R$$

Remark 3.7. *Note that by definition $L(f) \overset{f_L}{\sim} f$ and vice-versa, $R(f) \overset{f_R}{\sim} f$.*

4 Operations on Hars

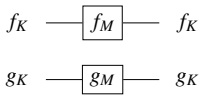
The main motivation for introducing Hars is providing an efficient implementation for composing string diagrams. To this aim, in this section we define the operations for constructing and combining Hars. These developments will also allow us to prove that Hars form a category, in the next section.

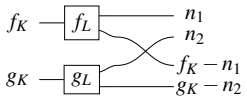
Definition 4.1.

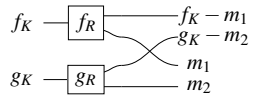
- *The identity* Har *of type $n \to n$ is* $(\mathbf{0}_{n,n}, \mathsf{id}_n, \mathsf{id}_n, \mathbf{0}_{n,1})$.
- *The symmetry $\sigma_{n,m} : n \otimes m \to m \otimes n$ is $(\mathbf{0}_{n+m,n+m}, \mathsf{id}_{n+m}, P, \mathbf{0}_{n+m,1})$ with P the block matrix* $\begin{vmatrix} \mathbf{0} & \mathsf{id}_m \\ \mathsf{id}_n & \mathbf{0} \end{vmatrix}$.

[3] We could take the alternative perspective that Har forms a weak 2-category with permutation matrices as 2-cells, but we will take the equivalence relation perspective to simplify our presentation.

- *Given an operation $g\colon n \to m \in \Sigma$, the 'singleton'*[4] Har *is given by $(M, \mathsf{id}_K, \mathsf{id}_K, N)$, with size $K = n+m+1$, node labels $N = (\mathbf{0}_{1,n}, g, \mathbf{0}_{1,m})$ and M the block matrix $\begin{vmatrix} 0 & 0 & 0 \\ S & 0 & 0 \\ 0 & T & 0 \end{vmatrix}$, where $S \in \mathsf{Mat}_{\mathbb{N}}(1,n)$ is the row vector $(1,2,\ldots,n)$ and $T \in \mathsf{Mat}_{\mathbb{N}}(m,1)$ the column vector $(1,2,\ldots,m)$.*
- *Let $f : n_1 \to m_1$ and $g : n_2 \to m_2$ be* Har*s. The tensor product $f \otimes g$ is component-wise as follows.*

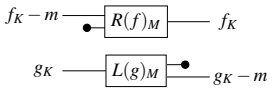
$(f \otimes g)_M =$ 

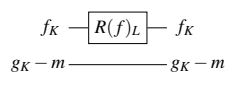
$(f \otimes g)_L =$ 

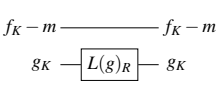
$(f \otimes g)_R =$ 

with $(f \otimes g)_N$ given by appending f_N and g_N, i.e., the block vector $(f_N \quad g_N)$.

- *Let $f : n \to m$ and $g : m \to l$ be* Har*s. Composition is defined component-wise as follows:*

$(f \fatsemi g)_M =$ 

$(f \fatsemi g)_L =$ 

$(f \fatsemi g)_R =$ 

with $(f \fatsemi g)_N$ given by appending $R(f)_N$ and $L(g)_N(b:)$, where $x(b:)$ denotes all but the first b elements of the array x. Alternatively, one may regard $(f \fatsemi g)_N$ as a diagonal matrix, and define composition as for $(f \fatsemi g)_M$.

Remark 4.2. *Note that in the definition of $(f \fatsemi g)_M$, the morphisms*  *and*  *represent the projection $\pi_2 : A \times B \to B$ and embedding $\iota_1 : A \to A \times B$ morphisms, respectively. One may think of the composition $M \fatsemi \pi_1$ as selecting the last columns of M and $\iota_0 \fatsemi M$ as selecting the first rows.*

5 Adequacy of the Har-Implementation

In this section we show how the interpretation of string diagrams as Hars can be described as a *full and faithful* functor between PROPs, meaning that our implementation is actually a 1-to-1 correspondence. As a preliminary step, we need to show how Hars form a category.

Proposition 5.1. *There is a PROP Har_Σ whose morphisms $n \to m$ are equivalence classes of values $\mathsf{Har}_{n,m}$ under the equivalence relation $\sim$, and identity, symmetries, composition and tensor product are as defined in Section 4.*

Proof. We give a graphical proof that composition is assocative up to permutation in the full version of our paper [17, Appendix A.6].

It is straightforward to check that $f \fatsemi \mathsf{id} = f$, and similarly one can check that $\sigma \fatsemi \sigma \overset{\sigma}{\sim} \mathsf{id}$. Finally, one can see that the tensor product is associative essentially because the direct sum is. □

Definition 5.2. *Let $[\![\cdot]\!] : \mathsf{Free}_\Sigma \to \mathsf{Har}_\Sigma$ be the identity-on-objects symmetric monoidal functor freely obtained by the mapping of operations $g \in \Sigma$ to singleton* Har*s, as defined in Section 4.*

Proposition 5.3. *$[\![\cdot]\!] : \mathsf{Free}_\Sigma \to \mathsf{Har}_\Sigma$ is an isomorphism of PROPs.*

[4] The name 'singleton' refers to the fact that such a Har contains a single generator. We choose this name based on its common usage in Haskell libraries for a function creating a datastructure (e.g. a set) with a single element.

We now give a sketch of our proof, leaving the full details to the full version of our paper [17, Appendix B].

Proof. Thanks to Proposition [17, B.6], it suffices to show that:

- There is a symmetric monoidal functor $\langle\cdot\rangle\colon \mathsf{Har}_\Sigma \to \mathsf{Free}_\Sigma$,
- Har_Σ is generated by the singleton Hars corresponding to the operations $g \in \Sigma$, and
- $\langle[\![g]\!]\rangle = g$ for $g \in \Sigma$.

Essentially, the idea is to show that Har_Σ is just a 'relabeling of generators' of Free_Σ. □

6 Complexity

We now give the time complexity of the composition and tensor product operations defined in Section 4. We give empirical results to validate our claims in Section 7.

Naively, since our algorithm is expressed in terms of matrix multiplication, it should have a time complexity of at best $O(n^{2.3728596})$ (at time of writing [1]). However, we can do significantly better by exploiting the high degree of *sparsity* of the matrices of a Har.

Concretely, observe that for a finite monoidal signature Σ and $f \in \mathsf{Har}(n,m)$, one can guarantee that the number of non-zero elements in f_M is $O(f_K)$:

Proposition 6.1 (Bounded sparsity)**.** *Fix a finite monoidal signature Σ and let f be a* Har*. Now Let m be the largest arity of any generator $g \in \Sigma$ and n the largest* coarity*. Then the rows of f_M have at most m non-zero elements, the columns at most n non-zero elements, and f_M has $O(f_K)$ non-zero elements.*

Proof. By Definition 3.1, each vertex v in the graph represented by f_M must have exactly m incoming and n outgoing edges. These edges correspond to the non-zero rows and columns of f_M, respectively, and so the non-zero elements of each row (resp. column) is at most m (resp. n). □

Now, it happens that the time complexity of the 'naive' sparse matrix multiplication algorithm [7] is essentially linear in the number of *non-trivial multiplications* required—that is, those scalar multiplications where neither multiplicand is zero. From this fact and the property of bounded sparsity, it follows that both composition and tensor product of Hars are linear-time operations. To make this clear, we introduce the following proposition:

Proposition 6.2 (Permutation of Har has linear complexity)**.** *Choose some $f \in \mathsf{Har}_\Sigma$ and a permutation matrix $P \in \mathsf{Mat}(f_K, f_K)$. Then $P \fatsemi f_M$ and $f_M \fatsemi P$ can be computed in linear time.*

Proof. For matrices $A, B \in \mathsf{Har}(k,k)$, the complexity of Gustavson's sparse matrix multiplication routine [7] is $O(2k + \mathsf{nnz}(A) + m)$. Here $\mathsf{nnz}(A)$ is the number of non-zero entries of A and m is the number of non-trivial multiplications required.

By the bounded sparsity property (Proposition 6.1), one can see that computing a row of the matrix $f_M \fatsemi P$ requires only a *constant* number of non-trivial multiplications, and further $\mathsf{nnz}(f_M)$ is $O(f_K)$. Thus, computing $f_M \fatsemi P$ is $O(f_K)$.

Alternatively, one may also see that linear complexity is possible using Gustavson's `HALFPERM` algorithm [7], which can compute $P \fatsemi f_M \fatsemi Q^T$ in $O(\mathsf{nnz}(f_M))$ operations. Since $\mathsf{nnz}(f_M)$ is $O(f_K)$, this operation has linear complexity. □

Using Proposition 6.2 we can now show that composition and tensor product have linear time complexity.

Proposition 6.3 (Tensor Product of Hars $f \otimes g$ is $O(f_K + g_K)$)**.** *Given $f \in \mathsf{Har}(n_1, m_1)$ and $g \in \mathsf{Har}(n_2, m_2)$, computation of $f \otimes g$ is $O(f_K + g_K)$.*

Proof. It is clear from definition 3.1 that each component of $f \otimes g$ is computed either as a direct sum or a multiplication of permutation matrices of size $f_K + g_K$. Since each of these operations is $O(f_K + g_K)$, it is clear that the whole operation is as well. □

Proposition 6.4 (Composition of Hars $f \fatsemi g$ is $O(f_K + g_K)$)**.** *Given $f \in \mathsf{Har}(n, m)$ and $g \in \mathsf{Har}(m, l)$, computation of $f \fatsemi g$ is $O(f_K + g_K)$.*

Proof. The proof is similar to that of Proposition 6.3, except that we must include the cost of the operations $R(f)$ and $L(g)$. These operations are linear by proposition 6.2, and so the composition $f \fatsemi g$ must also be $O(f_K + g_K)$. □

7 Empirical Results

We now give an empirical evaluation of our complexity claims on several synthetic benchmarks. We compare our own implementation to the wiring diagrams of `Catlab.jl` [11, 12].[5] Both our implementation and benchmarking code are available on GitHub at https://github.com/statusfailed/cartographer-har.

In the following benchmarks we use a fixed monoidal signature based on the finite presentation of boolean circuits described in [10]. We choose boolean circuits since they are a real-world application of string diagrams in which string diagrams would typically be very large. For example, a string-diagrammatic representation of a CPU would need (at least) hundreds of thousands of generators. In particular, our benchmarks use the following generators:

$$\Sigma = \{\mathsf{COPY} : 1 \to 2 \qquad \mathsf{XOR} : 2 \to 1 \qquad \mathsf{AND} : 2 \to 1 \qquad \mathsf{NOT} : 1 \to 1\}$$

Experiment Details Each benchmark has the same structure: for $k \in \{1 \dots 20\}$ we construct two string diagrams consisting of 2^{k-1} generators, and then measure tensor product or composition of those diagrams. We repeat each measurement 10 times for each k and plot the mean with minimum and maximum error bars. Further, if a result takes longer than 60 seconds to compute, it is omitted. More details of our experimental setup can be found in the full version of our paper [17, Appendix C].

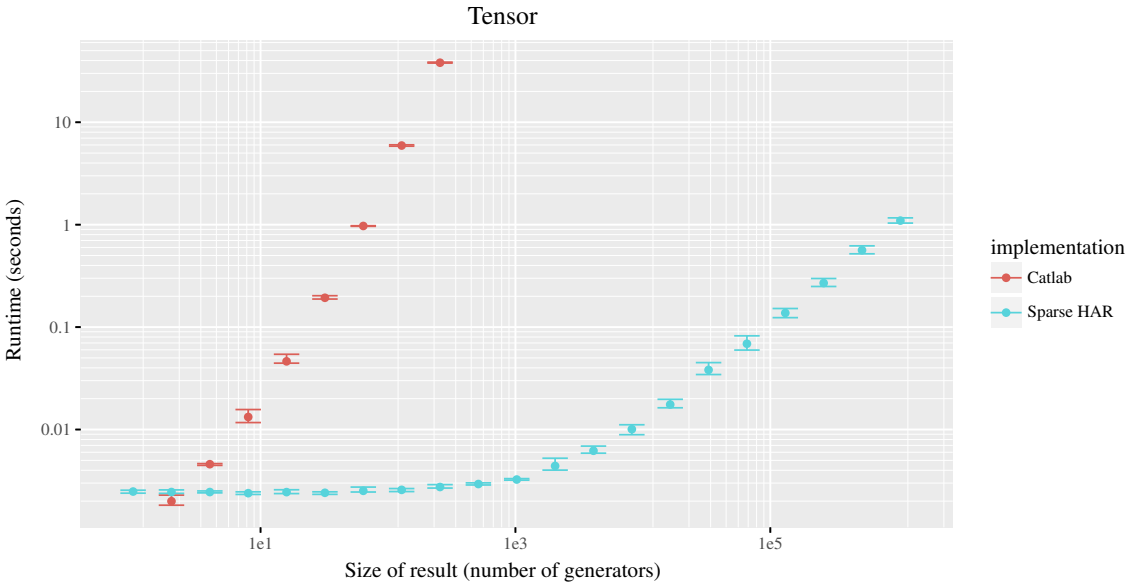
Note carefully that the performance chart for each benchmark uses a log scale on both axes, since for each k we construct a string diagram of size 2^k.[6]

7.1 Benchmark #1: Repeated Tensor

We first measure the performance of the tensor product of large representations. Concretely, let f be the k-fold tensor product of AND, i.e., $f = \mathsf{AND} \otimes \overset{k}{\dots} \otimes \mathsf{AND}$. We measure the performance of computing $f \otimes f$.

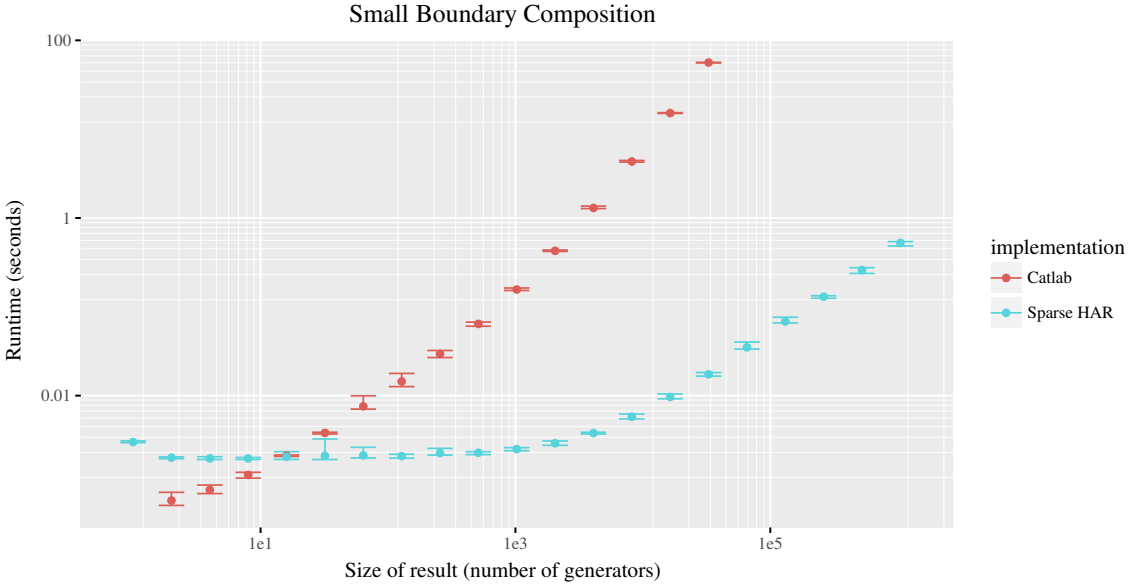
[5] Note however that Catlab's wiring diagrams provide a strictly more general setting than ours. We discuss possible generalisations of our approach to address this in Section 8.

[6] In these experiments, by "size" we mean specifically the number of generators in the diagram.



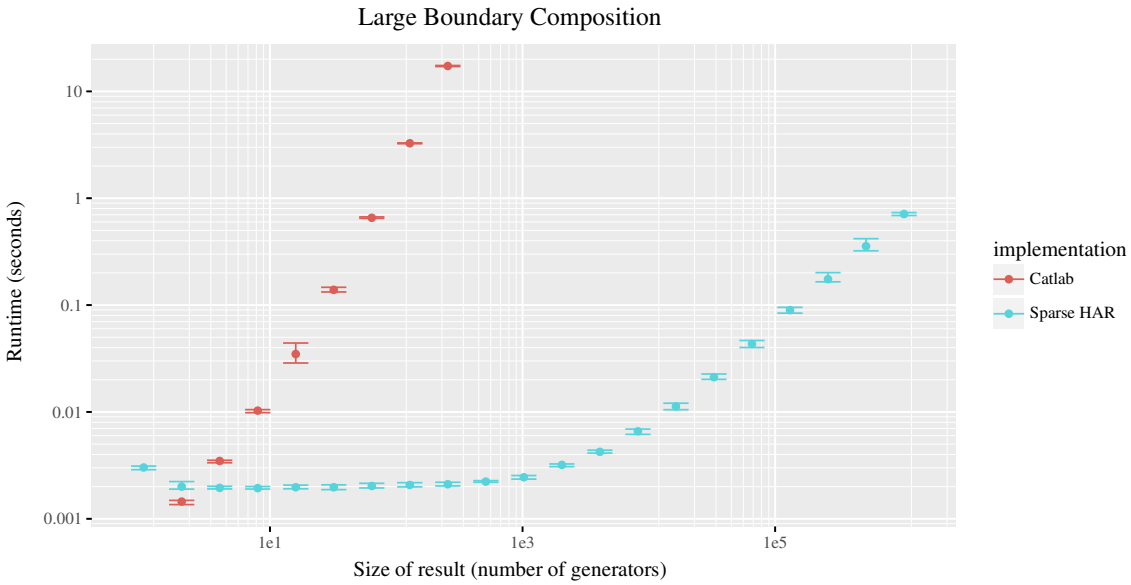
7.2 Benchmark #2: Small-Boundary Composition

We measure the performance of composition $n \xrightarrow{f} m \xrightarrow{g} l$ along a small shared boundary, i.e., where $m \ll f_K + g_K$. Concretely, let f be the k-fold composition of NOT, so that $f = \mathsf{NOT} \fatsemi \overset{k}{\cdots} \fatsemi \mathsf{NOT}$. We measure the performance of computing $f \fatsemi f$.



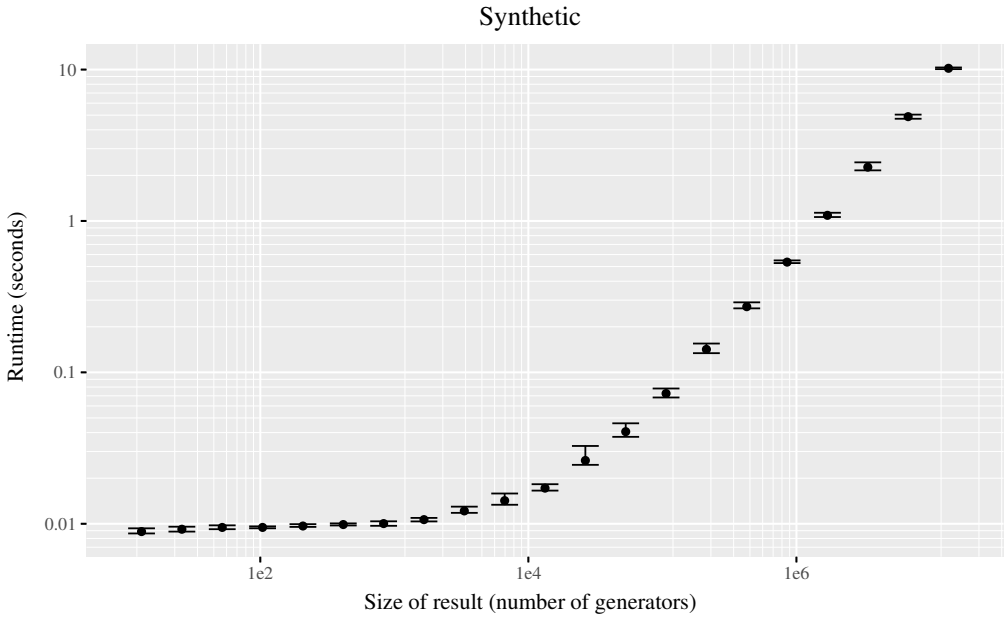
7.3 Benchmark #3: Large-Boundary Composition

We measure the performance of composition $n \xrightarrow{f} m \xrightarrow{g} l$ along a *large* shared boundary, i.e. where $m \approx \min(f_K, g_K)$. In particular, let f be the k-fold tensor product of NOT. Then we measure $f \fatsemi f$.



7.4 Benchmark #4: Synthetic Benchmark

We give a final benchmark as a validity check to ensure our implementation still performs well on realistic-looking representations. Specifically, we measure the performance of composing two 2^{k-1}-bit adder circuits to form a 2^k-bit adder.



8 Discussion & Future Work

We consider our work a step towards a set of high-performance algorithms for manipulating string diagrams, but naturally a number of avenues for improvement remain.

Most obviously, it remains to explore algorithms for matching and rewriting, which are necessary to support applications like a string-diagrammatic proof assistant. Perhaps less obviously, we would also like to study algorithms for *evaluating* circuit diagrams: this can be useful for e.g., simulating a boolean circuit or writing an interpreter for a programming language whose syntax is based on SMCs.

There are also several optimizations that could be made to our current algorithm. Firstly, we represent permutations as matrices, but a more efficient approach could be to use dense vectors of indices. However, this would require the implementor to have access to a function like the `HALFPERM` algorithm of [7].

Finally, several generalisations may be possible. Most useful would be to generalise to arbitrary symmetric monoidal syntax rather than just PROPs. Secondly, by modifying our representation slightly, we could account for arbitrary hypergraphs with interfaces — although we also believe this would affect the complexity bounds.

Acknowledgements Fabio Zanasi acknowledges support from EPSRC EP/V002376/1.

References

[1] Josh Alman & Virginia Vassilevska Williams (2020): *A Refined Laser Method and Faster Matrix Multiplication*.

[2] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński & Fabio Zanasi (2016): *Rewriting modulo symmetric monoidal structure*. *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. Available at `http://dx.doi.org/10.1145/2933575.2935316`.

[3] Bob Coecke & Aleks Kissinger (2017): *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, doi:10.1017/9781316219317.

[4] Lucas Dixon & Aleks Kissinger (2010): *Open Graphs and Monoidal Theories*.

[5] Fabrizio Genovese & Jelle Herold (2021): *A Categorical Semantics for Hierarchical Petri Nets*.

[6] Fabrizio Romano Genovese, Fosco Loregian & Daniele Palombi (2021): *Nets with Mana: A Framework for Chemical Reaction Modelling*.

[7] Fred G. Gustavson (1978): *Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition*. *ACM Trans. Math. Softw.* 4(3), p. 250–269. Available at `https://doi.org/10.1145/355791.355796`.

[8] Benjamin Heintz, Rankyung Hong, Shivangi Singh, Gaurav Khandelwal, Corey Tesdahl & Abhishek Chandra (2019): *MESH: A Flexible Distributed Hypergraph Processing System*.

[9] Stephen Lack (2004): *Composing PROPs*. *Theory and Applications of Categories* 13(9), pp. 147–163. Available at `http://www.tac.mta.ca/tac/volumes/13/9/13-09abs.html`.

[10] Yves Lafont (2003): *Towards an algebraic theory of Boolean circuits*. *Journal of Pure and Applied Algebra* 184(2-3), pp. 257–310, doi:10.1016/S0022-4049(03)00069-0.

[11] Evan Patterson, Andrew Baas, James Fairbanks, Micah Halter, Sophie Libkind & Owen Lynch (2020): *Catlab.jl: A framework for applied category theory*. doi:10.17605/OSF.IO/HMNFE. Available at `https://osf.io/hmnfe/`. Publisher: OSF.

[12] Evan Patterson, David I. Spivak & Dmitry Vagner (2020): *Wiring diagrams as normal forms for computing in symmetric monoidal categories*. In: *Proceedings of the 2020 Applied Category Theory Conference*. Available at `http://dx.doi.org/10.4204/EPTCS.333.4`.

[13] Peter Selinger (2010): *A survey of graphical languages for monoidal categories*. *arXiv:0908.3347 [math]* 813, pp. 289–355, doi:10.1007/978-3-642-12821-9-4.

[14] Julian Shun (2020): *Practical Parallel Hypergraph Algorithms*. In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, Association for Computing Machinery, New York, NY, USA, p. 232–249. Available at `https://doi.org/10.1145/3332466.3374527`.

[15] Pawel Sobocinski, Paul W. Wilson & Fabio Zanasi (2019): *CARTOGRAPHER: A Tool for String Diagrammatic Reasoning (Tool Paper)*. In Markus Roggenbach & Ana Sokolova, editors: *8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 139, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 20:1–20:7, doi:10.4230/LIPIcs.CALCO.2019.20. Available at `http://drops.dagstuhl.de/opus/volltexte/2019/11448`.

[16] David I. Spivak (2020): *Poly: An abundant categorical setting for mode-dependent dynamics*.

[17] Paul Wilson & Fabio Zanasi (2021): *The Cost of Compositionality: A High-Performance Implementation of String Diagram Composition*. Available at `https://arxiv.org/abs/2105.09257`.

[18] Raphael Yuster & Uri Zwick (2005): *Fast Sparse Matrix Multiplication*. *ACM Trans. Algorithms* 1(1), p. 2–13. Available at `https://doi.org/10.1145/1077464.1077466`.