University of Southampton

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science
Cyber Physical Systems Group

Category-Theoretic Datastructures and Algorithms for Learning Polynomial Circuits

by

Paul William Wilson

MEng

ORCiD: 0000-0003-3575-135X

A thesis for the degree of
Doctor of Philosophy

October 2023

University of Southampton

Abstract

Faculty of Engineering and Physical Sciences
School of Electronics and Computer Science

Doctor of Philosophy

Category-Theoretic Datastructures and Algorithms for Learning Polynomial Circuits

by Paul William Wilson

The purpose of this thesis is to provide practical, high performance tools for working with string diagrams for the specific application of machine learning. The thesis consists of two main lines of research towards this aim.

In Part I, we define a family of categories of differentiable circuits suitable for machine learning. This construction is made in a modular way: we first give an alternative axiomatisation of Reverse Derivative categories which is then used to prove a functional completeness result showing that these circuits are sufficiently expressive. We then show how 'gradient-like' learning can be understood in terms of morphisms of these categories, and discuss how to generalise gradient-based methods as applied to neural networks to new settings by varying the 'underlying arithmetic' of models.

Part II of the thesis is concerned with how to represent and manipulate large string diagrams, specifically with an eye to those defined in Part I. We develop datastructures and define efficient algorithms for tensor and composition of string diagrams in terms of simple linear-algebraic operations. We show that complexity of operations is linear in the size of the resulting diagram, and validate our claims with empirical evidence that our approach can handle diagrams constructed from millions of generators. Finally, we give a graphical calculus allowing terms of *non-strict* monoidal categories to be represented by our datastructure, which in turn yields novel proofs of Mac Lane's strictness and coherence theorems.

Contents

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;

7. Parts of this work have been published as: Paul Wilson and Fabio Zanasi. Categories of differentiable polynomial circuits for machine learning, 2022. URL `https://arxiv.org/abs/2203.06430` Paul Wilson and Fabio Zanasi. An axiomatic approach to differentiation of polynomial circuits. *Journal of Logical and Algebraic Methods in Programming*, 135:100892, 2023. ISSN 2352-2208. doi: https://doi.org/10.1016/j.jlamp.2023.100892. URL `https://www.sciencedirect.com/science/article/pii/S2352220823000469` Paul Wilson and Fabio Zanasi. Reverse derivative ascent: A categorical approach to learning boolean circuits. *Electronic Proceedings in Theoretical Computer Science*, 333:247–260, feb 2021. doi: 10.4204/eptcs.333.17. URL `https://doi.org/10.4204%2Feptcs.333.17` G. S. H. Cruttwell, Bruno Gavranović, Neil Ghani, Paul Wilson, and Fabio Zanasi. Categorical foundations of gradient-based learning, 2021. URL `https://arxiv.org/abs/2103.01931` Paul Wilson and Fabio Zanasi. The cost of compositionality: A high-performance implementation of string diagram composition, 2021. URL `https://arxiv.org/abs/2105.09257` Paul Wilson, Dan Ghica, and Fabio Zanasi. String diagrams for non-strict monoidal categories, 2022. URL `https://arxiv.org/abs/2201.11738`

Signed:.. Date:..................

Acknowledgements

I am indebted to Fabio Zanasi not only for his timely advice (which always seemed to arrive precisely when I needed it most), but also for the trust he placed in me to work independently. This balance has been everything I had hoped for in a research degree, and I look forward to our continued future collaboration. I am also grateful to Paweł Sobociński for taking a chance on a student with an atypical academic background, and for providing a great deal of helpful guidance in my early studies.

I also wish to thank mentors, friends, and colleagues from a past age: John Greenall, for his ongoing support and friendship during both my professional and academic career; Danijela Horak, whose puzzles inspired me to return to Mathematics; Simon Worgan and Henk Muller, who were supportive long past any official responsibility, and yet never hesitated to help.

Thanks go last but not least to my family. First to my wife Liu, who has enriched my life more than I can say. To Dad, who helped a spark of interest in computers become a flame, and Mom, who made sure they weren't the *only* thing I was interested in. Finally to the rest of my family, who never stopped asking when I'd get a real job.

To Liu

Without whom I could not have begun,
and would surely not have finished.

Chapter 1

Introduction

This thesis consists of two main lines of work: (1) a string-diagrammatic account of gradient-based machine learning, and (2) datastructures and algorithms for representing large diagrams. Together, the aim is to motivate the adoption of string diagrams for use-cases involving diagrams of 'industrial scale', such as circuits.

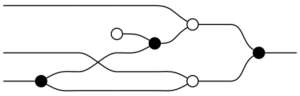
We begin by motivating these lines of research in more detail.

1.1 Motivation

The language of Symmetric Monoidal Categories (SMCs) is capable of describing many classes of open system. Examples include combinational circuits [76], electrical circuits [55, 57] signal flow graphs [21] petri nets [8], and more.

An important benefit to framing an application domain as a symmetric monoidal category is that one can exploit the intuitive but completely formal graphical language of *string diagrams*. Selinger [92] gives a survey of string diagrammatic syntax, and early formal treatments are given by Joyal and Street [70] and Lafont [77]. A first example of a string diagram which we will return to in Chapter 4 is shown in Example 1.1.

Example 1.1.



This particular diagram represents the polynomial $\langle x_0, x_1, x_2 \rangle \mapsto (1 + x_2) \cdot x_0 + x_2 \cdot x_1$, but can equivalently be thought of as a 'circuit' whose wires carry values in some semiring S.

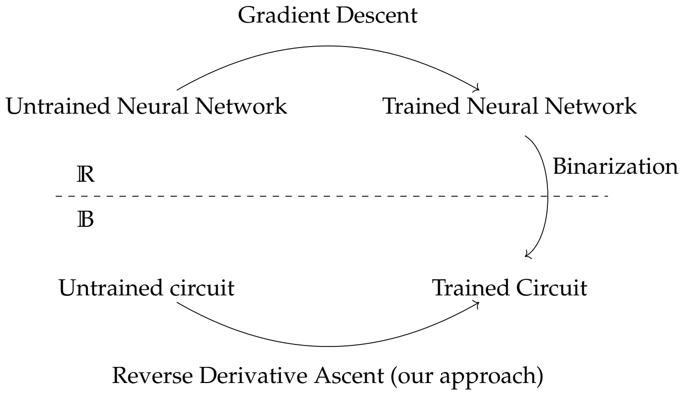
1.1.1 String Diagrams and Gradient-Based Learning

One domain where the use of such diagrammatic representations is less explored is that of gradient-based learning for neural networks. Despite this, recent mainstream machine learning literature often includes informal string-diagram-like notation to communicate the structure of models (their 'architecture') to the reader, for example as in [100, Figure 1] and [72, Figure 3]. Thus, by proposing *string diagrams* as such a notation, we immediately gain the benefits of a pedagogically useful syntax which–by nature of being completely formal–is also *unambiguous*. Additionally, using string diagrams means we can leverage the theoretical results of category theory, for instance rewriting diagrams *modulo symmetric monoidal structure* as described in [17]. Lastly, but most importantly for this thesis, framing model architectures in category theoretic terms allows us to see new avenues to generalise.

The driving application for this thesis is to generalise the 'underlying arithmetic' of neural networks. More concretely, consider again Example 1.1 and fix the semiring $S = \mathbb{R}$. Now we may think of this circuit as an extremely simple neural network architecture; since it represents a polynomial it is certainly differentiable, and thus we may optimise some of its inputs with gradient descent. A naturally arising question is then: "can gradient descent be used for circuits whose underlying semiring is not $\mathbb{R}$?" We argue that the answer is yes, and that synthetic categorical approaches to differentiation allow us to do so.

Generalising the underlying arithmetic of neural networks is not merely an abstract exercise: it has real applications. Modern neural network models can be extremely large, requiring power-hungry GPU hardware for both training and inference [33, 88]. This makes it challenging to run large models in low-power and embedded settings. One approach to mitigating this increased need for computational resources is an approach called *binarisation* [34, 101]. In this case, a neural network is first trained with real-valued weights before quantising them to 'binarised' (boolean) values. In doing so, the 'forward' (predictive) pass of the model can be made simpler and thus more efficient to execute. For example in [33], the authors are able to replace floating-point multiply-accumulate operations by accumulations alone. In the more extreme example of [101], the authors are able to extract a boolean circuit from the binarised model which is then run on FPGA hardware.

However, binarisation schemes still typically require the use of $\mathbb{R}$-valued gradients in the training phase. This means that the performance benefits of binarisation do not typically extend to model training, since one must still make use of expensive floating-point operations. Now, if the ultimate goal is to extract a boolean circuit, the natural question to ask is whether the parameters of such a circuit can simply be trained directly. We depict these contrasting approaches informally as follows.



(1.1)

Here, the 'gradient descent' arrow represents standard approaches to training neural networks, while 'binarization' represents the extraction of a boolean circuit from a trained network. The bottom arrow labeled 'Reverse Derivative Ascent' represents our proposed approach, in which a model is trained purely within the $\mathbb{B}$-valued regime. More generally we might consider any semiring S, in which case we arrive at the first research question of this thesis: can gradient-based learning be directly applied to any circuit over a semiring S, without use of $\mathbb{R}$-valued gradients? In answering this question, we hope to gain the benefit of smaller, more efficient models in *both* the training and prediction phases.

1.1.2 Datastructures for Representing Large Diagrams

In order for these ideas to progress beyond mere theory, we must develop tools for their practical application. Specifically, this means developing datastructures and algorithms to represent models (circuits) over some arbitrary semiring S. However, representing just these circuits is not the only goal of such datastructures.

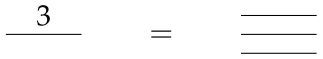
Aside from the informal use of string-diagram-like diagrams in machine learning literature, use of graphical syntax also occurs in industry. Such examples can be found for example in the 'node-based editor' of Nuke [52] and the 'visual scripting' language of Unity3D [99]. These examples suggest that datastructures for string diagrams may also be generally useful as the basis for such applications. For that reason, we propose that a datastructure representing the *free symmetric monoidal category on some signature* to have general applicability.

With the goal of a general purpose datastructure in mind, we set out two main desiderata for their use. Firstly, our datastructures must be *formal objects*: they should themselves described in terms of categories. Secondly, they must be *scalable*: our algorithms must be data-parallel by construction, and admit efficient implementation on both sequential and parallel (e.g., GPU) hardware. This is critical when diagrams become very large, as for instance in the case of circuits.

However, before we can consider such datastructures to be truly 'general purpose', there remains a theoretical issue to address. Namely, that of *strictness*.

1.1.3 String Diagrams and (Non-)Strictness

One limitation on the use of string diagrams for applications is the underlying assumption of *strictness*. While we will give more rigorous exposition in the main thesis content, one consequence of this requirement is that when modelling a string diagram with a datastructure, each distinct wire must be explicitly represented. Consider for example the identity diagram on three inputs, below left.



With the assumption of strictness, a datastructure representation of this diagram must explicitly model a 'bundle' of n wires with n distinct objects in the datastructure. This is cumbersome for both pedagogical and practical purposes: it is hard to read, and working with such a diagram is analogous to writing a program in which one must explicitly name every element of an array. More preferable would be the option to work with 'bundles' of wires directly. In terms of our datastructure, this means distinguishing between n distinct wires, and a single wire labeled n. More formally, we can think of this as relaxing the equality above to merely an isomorphism.

In addition to this problem, some useful monoidal categories are not strict to begin with. This means that without additional theoretical developments, we may not consider string diagrams for such categories to be formal objects at all.

The latter issue is resolved in principle by Mac Lane's strictness theorem for monoidal categories [80]. Mac Lane famously shows that every monoidal category is monoidally equivalent to a strict one, thus justifying the use of string diagrams. However, the strictness theorem is stated in terms of the *coherence* theorem, making it unclear how to mechanically translate between the non-strict and 'strictified' settings. Moreover, the theorems as stated do not give an explicit graphical calculus for manipulating non-strict terms in the strict setting.

In order to make these results more useful for computer implementation, we give novel proofs of Mac Lane's results. Concretely, we will define a graphical calculus for non-strict monoidal categories, and then use it to prove the strictness theorem. As a bonus, we will use our proof of the strictness theorem to provide a novel proof of the *coherence* theorem. This approach will yield a explicit, computer-friendly specification of functors mapping between non-strict categories and their strict equivalents.

In reformulating the strictness and coherence results, we hope to make our datastructures for string diagrams more generally useful. One application is to allow for a wider variety of machine learning model class to be treated diagrammatically. An example of such a model class is the (reverse) differentiable programming language described by Cruttwell et al. [37].

1.1.4 Summary

To summarise, this thesis consists of two major lines of work:

1. A string-diagrammatic account of gradient based learning which generalises beyond the semiring of real numbers.
2. High-performance, general-purpose datastructures and algorithms for representing large string diagrams, and a graphical calculus extending their applicability to *non-strict* categories.

1.2 Content of the Thesis

We now briefly summarise the structure and content of the thesis, which is split into two parts. We begin with a brief sketch of each of these two parts, then describe chapter structure, and finally give a more in-depth description of the contributions of the thesis.

In Part I, we give a categorical formulation of gradient-based machine learning using *reverse derivative categories.* In particular, we define categories of differentiable 'polynomial circuits' in Chapter 3, and then give algorithms for learning their parameters in Chapter 4.

Part II of the thesis is concerned with the datastructures and algorithms used to represent morphisms of monoidal categories. We begin in Chapter 5 by describing datastructures and algorithms for representing morphisms of **PROP**s, which are able to represent the circuits of Chapter 3. We then give a graphical calculus allowing for the representation of morphisms of non-strict categories in Chapter 6.

Each chapter is structured as follows. We begin with an overview and synopsis of the chapter contents. We then include *preliminary* content for each chapter, consisting of background material specific to the chapter. Note that each 'preliminaries' section does not contain original contributions of the author. The remaining sections of each chapter consist of the original contributions of the author, unless explicitly noted.

We now give a more in-depth summary of the contributions of Parts I and II of the thesis.

1.2.1 Part I

In Chapter 3, we describe a family of reverse-differentiable categories $\mathbf{PolyCirc}_S$ parametrised by a semiring S. We build this definition in a modular way, starting from a re-axiomatisation of the theory of Reverse Derivative Categories [29] in terms of presentations by generators and equations. This reformulation allows us to prove a theorem showing how to extend categories having reverse derivative structure to include new generating morphisms while retaining reverse derivative structure. To complete the chapter, we show how for any given semiring S, one can extend the category $\mathbf{PolyCirc}_S$ to obtain the property of 'functional completeness'. Essentially, this means that for any desired function of type $S^m \to S^n$, there exists a circuit with that interpretation. This result is key for use in machine learning, because it ensures the 'model class' of circuits is sufficiently expressive.

Chapter 4 uses reverse derivative structure to define concrete learning algorithms. We begin by defining 'Reverse Derivative Ascent', an algorithm for learning the parameters of models expressed as morphisms of $\mathbf{PolyCirc}_S$ for a ring S. In contrast to existing techniques, this allows for models to be trained using only values in S, without requiring the use of floating-point approximations. Consequently, in the case of 'boolean circuits' where $S = \mathbb{Z}_2$, it is possible to learn parameters directly as depicted in (1.1). We then show how the Reverse Derivative Ascent algorithm generalises (using joint work with Cruttwell et al. [36]) before finally giving empirical results of our algorithms on some benchmark datasets.

1.2.2 Part II

While Part I of the thesis is concerned with particular categories and morphisms, Part II is concerned with how such morphisms can be efficiently *represented*. In Chapter 5 we describe a general-purpose datastructure for representing morphisms of **PROP**s. Our primary motivation is to represent the polynomial circuits of Part I, but we also intend to provide a datastructure for general use. For this reason, we also describe algorithms for tensor product and composition of morphisms, and ensure their complexity is linear in the size of the resulting diagram. We conclude the chapter with empirical results of our algorithms' performance, and discuss how to extend the representation to the case of categories with more than one generating object.

The final contributions of the thesis are in Chapter 6, where we are concerned with how to represent morphisms of *non-strict* monoidal categories. In doing so, we develop a graphical calculus for non-strict monoidal categories. Ultimately, this amounts to a novel proof of Mac Lane's strictness theorem, but with key differences. Primarily, our proof makes no reference to the *coherence* theorem, and so we are also able to give a novel graphical proof of the coherence theorem without introducing circularity.

1.3 Synopsis and Original Contributions

We now give a synopsis of the thesis content. Each chapter summary is accompanied by a table of references to the main contributions, as well as a list of published works upon which each chapter is based. Chapters 3 - 6 are structured to be largely self-contained, beginning with an overview, chapter synopsis, and prerequisites, with remaining sections constituting original contributions unless explicitly stated.

Chapter 2 recalls the graphical language of string diagrams for symmetric monoidal categories, and some well-known theorems of use in the thesis. This chapter consists only of background material, and does not contain original contributions of the author.

Chapter 3 begins with a re-axiomatisation of reverse derivatives (Definition 3.13) and a proof of its equivalence to the original formulation (Theorem 3.14). We give an 'Extension Theorem' showing how presentations of RDCs can be extended with additional generators in Theorem 3.15. These results are used to define a category $\mathbf{PolyCirc}_S$ of 'differentiable polynomial circuits' (Definition 3.22), which have reverse derivative structure (Proposition 3.24). We define the property of 'functional completeness' in Definition 3.29, and show that the inclusion of a 'comparator' function is sufficient to gain this property in Theorem 3.30. Finally, we define $\mathbf{PolyCirc}_S^{=}$ (Definition 3.31) as an extension of $\mathbf{PolyCirc}_S$ with a comparator operation. We use the extension theorem to show that $\mathbf{PolyCirc}_S^{=}$ has reverse derivative structure in Theorem 3.32, and prove it is functionally complete in Corollary 3.33. The chapter concludes with a discussion of the impact of the choice of semiring S on model construction, as well as a number of examples of neural network layers defined as morphisms in $\mathbf{PolyCirc}_S$.

TABLE 1.1: Main Contributions of Chapter 3

Contribution	Reference
Graphical formulation of reverse derivatives	Definition 3.13
Proof of equivalence to original theorem	Theorem 3.14
Extension Theorem	Theorem 3.15
Definition of $\mathbf{PolyCirc}_S$	Definition 3.22
$\mathbf{PolyCirc}_S$ is an RDC	Proposition 3.24
Sufficiency of comparators for functional completeness	Theorem 3.30
Definition of $\mathbf{PolyCirc}_S^{=}$	Definition 3.31
$\mathbf{PolyCirc}_S^{=}$ is an RDC	Theorem 3.32
Functional completeness for $\mathbf{PolyCirc}_S^{=}$	Corollary 3.33

Chapter 3 is based on the following papers.
• Wilson and Zanasi [105] – *Categories of Differentiable Polynomial Circuits for Machine Learning*
• Wilson and Zanasi [104] – *Reverse Derivative Ascent: A Categorical Approach to Learning Boolean Circuits*

Chapter 4 discusses the practicalities of gradient-based learning with polynomial circuits. Our contributions begin with the definition of a learning algorithm called 'Reverse Derivative Ascent' (Definition 4.6) for morphisms of $\mathbf{PolyCirc}_S$. We give empirical evidence of its effectiveness by applying models to chosen benchmark datasets in Section 4.6, with experimental results in Table 4.2. In addition, we show experimentally that our formulation of learning is consistent with mainstream approaches to training neural networks (Table 4.1). Section 4.7 concludes the chapter with a discussion of subtleties arising from the implementation of our experiments as *interpretations* of morphisms of $\mathbf{PolyCirc}_S$ as functions. We formalise this as a statement about 'compositions of interpretations' of morphisms of $\mathbf{PolyCirc}_S$ in Proposition 4.20.

TABLE 1.2: Main Contributions of Chapter 4

Contribution	Reference
Reverse Derivative Ascent algorithm	Definition 4.6
Experimental Results: (Reverse Derivative Ascent)	Table 4.2
Experimental Results: (Neural networks)	Table 4.1
Implementation of Reverse Derivative Ascent	Implementation D.1
Implementation of RDA experiments	Implementation D.2
Implementation of Numeric Optics & Experiments	Implementation D.3
Composition of Interpretations	Proposition 4.20

Chapter 4 is based on the following papers:
- Wilson and Zanasi [104] – *Reverse Derivative Ascent: A Categorical Approach to Learning Boolean Circuits*
- Cruttwell et al. [36] – *Categorical Foundations for Gradient-Based Learning*

Note carefully that Section 4.5 is the result of joint work with Cruttwell et al. [36], and is not the author's sole contribution.

Chapter 5 introduces the Hypergraph Adjacency Representation (**Har**), a datastructure for representing morphisms of **PROP**s. The **Har** representation is specified in terms of adjacency matrices in Definition 5.14. We then give operations for tensor product (Definition 5.26) and composition (Definition 5.28), before showing that both have time complexity linear in the size of the resulting diagram (Propositions 5.42 and 5.43, respectively). Empirical performance results of our algorithms compared to those of an existing library for string diagrams in Section 5.8. Section 5.9 concludes the chapter with a discussion of how the **Har** datastructure may be extended to model morphisms of the free symmetric monoidal category with multiple generating objects.

TABLE 1.3: Main Contributions of Chapter 5

Contribution	Reference
Har datastructure	Definition 5.14
Tensor product of **Har**s	Definition 5.26
Composition of **Har**s	Definition 5.28
Complexity of **Har** tensor is linear	Proposition 5.42
Complexity of **Har** composition is linear	Proposition 5.43
Implementation of **Har** datastructure and operations	Implementation D.4

Chapter 5 is based on the following papers:

• Wilson and Zanasi [103] – *The Cost of Compositionality: A High Performance Implementation of String Diagram Composition*

Chapter 6 introduces a graphical language for non-strict monoidal categories. Given an arbitrary monoidal category $\mathscr{C}$, we define its 'strictification' $\overline{\mathscr{C}}$ (Definition 6.5) as a presentation by generators and equations. We then construct functors $\mathcal{S} : \mathscr{C} \to \overline{\mathscr{C}}$ and $\mathcal{N} : \overline{\mathscr{C}} \to \mathscr{C}$ which we show are monoidal in Propositions 6.8 and 6.15, respectively. These two functors form an equivalence (Theorem 6.22) which constitutes a novel proof of Mac Lane's strictness theorem. Since our proof does not rely on the coherence theorem, it can then be used to *graphically* prove Mac Lane's coherence result (Theorem 6.23) without introducing circularity. Finally, we show how the strictness result extends to the *symmetric* monoidal case by showing that $\overline{\mathscr{C}}$ 'inherits' symmetric monoidal structure from $\mathscr{C}$ (Proposition 6.43), and that this extends to the functors $\mathcal{S}$ and $\mathcal{N}$ (Propositions 6.44 and 6.45, respectively).

TABLE 1.4: Main Contributions of Chapter 6

Contribution	Reference
The 'strictification' $\overline{\mathscr{C}}$ of a monoidal category $\mathscr{C}$	Definition 6.5
Existence of a monoidal functor $\mathcal{S} : \mathscr{C} \to \overline{\mathscr{C}}$	Proposition 6.8
Existence of a monoidal functor $\mathcal{N} : \overline{\mathscr{C}} \to \mathscr{C}$	Proposition 6.15
Elementary proof of the Strictness Theorem	Theorem 6.22
Graphical proof of the Coherence Theorem	Theorem 6.23
$\overline{\mathscr{C}}$ inherits symmetric monoidal structure from $\mathscr{C}$	Proposition 6.43

Chapter 6 is based on the following papers:
• Wilson et al. [107] – *String diagrams for non-strict monoidal categories*

Chapter 7 concludes the thesis, and discusses avenues for future work.

1.4 Related Work

This thesis has its roots in a number of research areas. Part I is built on synthetic categorical treatments of differentiation and touches on mainstream neural networks literature, while Part II is concerned with the combinatorial structure of string diagrams and more foundational questions. However in all cases the work in this thesis is characterized by the heavy use of string diagrams, which are central to our approach. We therefore begin with a brief discussion of the literature for string diagrams.

The use of informal diagrammatic notation has a long history. An early example of digrammatic syntax for logic is due to Peirce [9], while Zanasi [109] notes that diagrammatic notation for signal flow graphs appears in a technical report by Shannon [93]. A more modern example is the notation introduced by Penrose [86] (see also [7]), and developed by Joyal and Street [70], Lafont [77], and others. There exist a number of variants of string diagram notation such as proof nets (see for example Blute et al. [12]), but in this thesis we restrict ourselves to those for symmetric monoidal categories as described in the comprehensive survey due to Selinger [92].

Our definition of categories of polynomial circuits in Chapter 3 is heavily reliant on categorical treatments of differentiation. We chiefly rely on the theory of Reverse Derivative Categories developed by Cockett et al. [29], but this work itself builds on a long history of axiomatisations of (cartesian) differential categories, as for example in [13, 35]. A recent extension of reverse derivative categories also considers the more general monoidal case (as opposed to just cartesian monoidal) [38]. We do not use this generalisation here, but discuss the implications for potential future directions in the conclusion of the thesis. Our definition of $\mathbf{PolyCirc}_S$ also has its roots in the boolean circuits of Lafont [76], but we note the existence of a large number of categorical models of various kinds of circuit [15, 56, 57]. In fact, a key part of our approach is to identify the relationship between the presentation of boolean circuits in [76] with categories of 'reverse differentiable' polynomials defined in [29]. Moreover, this relationship inspires the functional completeness result of Section 3.6, which itself is a discrete analogue to the various universal approximation theorems for neural networks (e.g., [67, 79]).

As well as cartesian (reverse) differential categories, there are several alternative approaches to categorical models of change–we mention them here for completeness. One notable example is the 'change actions' of Alvarez-Picallo [3] (see also Alvarez-Picallo and Ong [4]), as well as a recent work characterising Reverse Mode automatic differentiation [5] using the functorial boxes of Melliès [81].

The primary inspiration for our work on direct learning of parameters of circuit models comes from the circuit extraction technique of Wang et al. [101]. In fact, a later paper by Constantinides [32] also gives good motivation for reconsidering the underlying arithmetic of neural networks for the purposes of hardware acceleration and efficiency. More generally, our work is inspired by the technique of neural network *binarisation* as in Courbariaux et al. [33, 34]. Broadly speaking, the Reverse Derivative Ascent learning algorithm can be considered as a cousin of gradient descent as applied to neural networks [90].

From a categorical perspective, our work builds on the treatment of supervised learning given in Fong et al. [50] (later followed by [96]). A particularly relevant example is the work of Sprunger and Katsumata [97], who use cartesian *forward* derivative categories for computing gradients. However, this work predates the introduction of reverse derivatives [29], and the authors note that "when there are millions of parameters in a machine learning model, [use of the forward derivative] is computationally disastrous". The specific use of the reverse derivative is therefore critical from an efficiency perspective.

The use of lenses and optics for various kinds of learning has also enjoyed increased attention in the literature. Foundational work on properties of various kinds of optics can be found in e.g., [14, 26, 63, 89], while the specific use of optics in the context of machine learning has recently been addressed in the work of Fong and Johnson [49], and Braithwaite and Hedges [23] in the case of bayesian learning.

Our work on developing datastructures for the efficient representation of morphisms of monoidal categories is primarily based on the work of Bonchi et al. [17], which is later developed into a three-part series of papers on rewriting [18–20]. We also draw some inspiration from an earlier work on open graphs by Dixon and Kissinger [41]. This latter work also has an implementation for diagrammatic reasoning in the Quantomatic [75] tool, as well as a related tool, PyZX [74], for working with the ZX-calculus. Another general tool frequently used for quantum natural language processing tasks is DisCoPy [39]. More recent implementations of string diagram representations include the wiring diagrams [85] of Catlab.jl [82], and most recently the 'higher-dimensional' approach of Hadzihasanovic and Kessler [62]. In contrast to these approaches, the work in this thesis is intended firstly to serve as a standard 'reference implementation' which can be easily implemented, and secondly to be high-performance by construction. With respect to this latter point, we demonstrate empirically that our datastructure is able to handle diagrams of 'industrial scale' by taking advantage of highly optimised sparse matrix libraries.

Our work on developing a graphical calculus for non-strict monoidal categories in Chapter 6, while based on relatively fundamental definitions, relates to several existing works in the literature. One such notable example arises in the study of weakly distributive categories of Blute et al. [12], where the authors introduce a diagrammatic calculus with two distinct tensor products and an additional type of wire called a 'thinning link'. It seems that our construction may also arise by omitting the additional structure, but this is not explored in [12]. A similar type-theoretic approach not based on diagrams can also be found in [94], and the idea of using 'adapters' also arises in the reversible circuits of Choudhury et al. [28]. More closely related to our graphical calculus are the scalable graphical calculi described in [24, 25] which contain several similar constructions. However, the authors do not explicitly study the connection to Mac Lane's strictness and coherence theorems. As a final note, we highlight that our construction serves a similar purpose to the 'functorial boxes' of Melliès [81].

Chapter 2

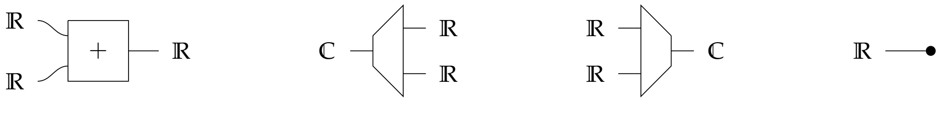
Background

In this chapter, we review the basics of the graphical language of string diagrams for symmetric monoidal categories, as well as how categories can be defined as presentations by generators and equations. Subsequent chapters will make heavy use of both of these concepts. A comprehensive survey of graphical languages for various categories can be found in [92]. We will also assume familiarity with basic category theory, especially with the definitions of symmetric monoidal categories and functors.

2.1 String Diagrams

The categories defined in this thesis will often be stated in terms of a graphical presentation by generators and equations. In this section, we aim to familiarise the reader with such presentations. We begin with intuition, and then proceed to more precise definitions in Section 2.2.

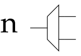
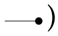
Consider a simple fragment of an arithmetic expression language with operations on real and complex numbers. We might wish to express addition on the reals, 'packing' and 'unpacking' complex numbers into their real and imaginary components, and simply 'discarding' values. Graphically, we can depict these operations in (2.1) in the specified order.



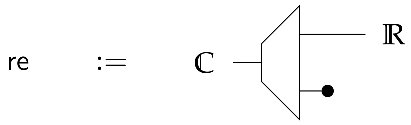
(2.1)

For the purposes of this example, we may think of these operations as functions. [1] Inputs are depicted as dangling wires on the left, and outputs as dangling wires on the

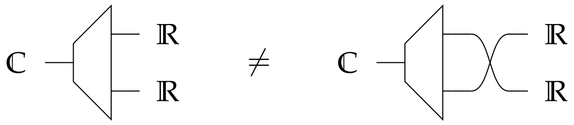
[1] The intuition for such operations as functions does not hold in general. Often morphisms of symmetric monoidal categories can have interpretations which are not functions with inputs and outputs; one such example is the category **FinRel** of finite relations.

right. Note that some operations (e.g., the 'unpacking' operation ) have multiple outputs while others (e.g., the 'discarding' operation ) have none at all.

Naturally we would like to use these operations to construct more complex ones. We can do this by placing our primitive operations on the page and 'wiring them up' as we desire. For example, we might define the function extracting the real component of a complex number re : $\mathbb{C} \to \mathbb{R}$ graphically as follows:

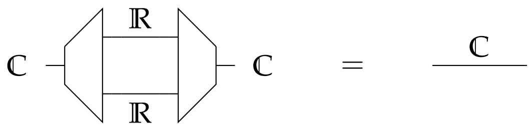


Not every such wiring is permitted, however. In order for a diagram to correspond to a morphism of a symmetric monoidal category, operations must be wired *acyclically*,[2] and wirings must be 'well-typed' (that is, one cannot connect wires labeled A and B when $A \neq B$). In addition, when considering equality of diagrams, the order of inputs and outputs must be respected. For example, the following two diagrams are not the same, because their right boundary wires are in a different order.



In this case, the inequality is clearly desirable: the real and imaginary components of a complex number are obviously not interchangeable.

However, it is often the case that we would like to consider two distinct diagrams to have the same interpretation. To allow for this, we can introduce some equations of diagrams. One such equation we ought to have is that unpacking a complex number into its two components and then 'repacking' them should be the same as doing nothing at all. We can express this with an equation saying that the composition of 'unpacking' and 'packing' is simply the identity.



(2.2)

In fact, we will require a much more general version of this equation when we come to address issues of *strictness* in Chapter 6.

To summarise, we have made three choices which together will form a presentation of a symmetric monoidal category. First, a set $\Sigma_0 := \{\mathbb{C}, \mathbb{R}\}$ of *generating objects* of the category. Second, a set of 'primitive operations' Σ_1 given in (2.1) called *generating*

[2] The acyclicity condition is required for such diagrams to represent a morphism of a symmetric monoidal category. However, *traced* symmetric monoidal categories permit such wirings.

morphisms or just *operations*. Thirdly, a set of equations Σ_2 as in (2.2). These last two components together define the morphisms of the category as those diagrams constructed by permissible wirings from the primitive operations in Σ_1 and considered equal 'up to' the equations in Σ_2. More precisely, two diagrams will represent the same morphism if one can be transformed into the other by deforming it and applying a sequence of rewrites[3] using the equations (2.1).

While we hope the reader now has a clearer intuition for what a presentation is, we must now be more precise. In fact, we have already conflated two distinct concepts: (i) presentations of SMCs by operations and equations and (ii) the graphical language of string diagrams. In order to be precise, we will now review these two concepts.

2.2 Presentations by Generators and Equations

We can now formally define what is meant by a *presentation by generators and equations*. The constructions in this section are well-known, and we recall them here only for completeness. Moreover, in this section we explicitly consider only *strict* symmetric monoidal categories. The objects of these categories will be lists over some generating set, which we define as follows.

Definition 2.1. Fix a set S. The **free monoid** on S, denoted $\mathsf{List}(S)$ is the set whose elements are finite sequences of elements in S. We denote such sequences by $\langle x_1, x_2, \ldots, x_n \rangle$ for $x_i \in S$, and denote the **empty sequence** by I or $\langle\rangle$. We may alternatively call elements of $\mathsf{List}(S)$ **words over** S, or **lists of elements in** S.

We will start by defining 'monoidal signatures', which specify the primitive pieces from which our categories will be made. Next, we define Σ-terms for a given signature Σ. These terms can be thought of as the 'syntax trees' of all possible expressions that can be built by combining the primitive operations of Σ using tensor and composition. We will then define the free strict symmetric monoidal category over a signature, which we think of as the category having generating objects and operations, and equations *only* those of strict symmetric monoidal categories. Finally, we will define a presentation by operations and equations as the *quotient* of the free symmetric monoidal category by some specified additional equations.

Definition 2.2 (Monoidal Signature [70, 73, 92]). A (strict) **monoidal signature** Σ is a pair (Σ_0, Σ_1) where:

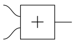
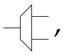
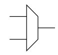
- Σ_0 is a set of **generating objects**,

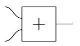
[3] A 'rewrite' of an equation $l = r$ here means finding an occurence of l in a diagram d, and replacing it by r. Note that there are conditions on which such rewrites are legal; a formal treatment of rewriting for symmetric monoidal categories is given by Bonchi et al. [17].

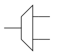
- Σ_1 is a set of **generating morphisms**,

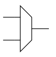
so that there is a function $\tau : \Sigma_1 \to \mathsf{List}(\Sigma_0) \times \mathsf{List}(\Sigma_0)$ taking each generating morphism to its *domain* and *codomain* which are both lists of elements in Σ_0. We will sometimes refer to Σ_0 and Σ_1 together as **generators**, and Σ_1 alone as **operations**.

We have already seen an example of a signature in the previous section:

Example 2.1. *Define* $\Sigma_0 := \{\mathbb{R}, \mathbb{C}\}$ *and* $\Sigma_1 := \{$  ,  ,  , $\}$*, (i.e., the operations depicted in equation* (2.1)*). Define* τ *according to the labeled wires in* (2.1)*:*

$\tau($  $) := (\langle \mathbb{R}, \mathbb{R} \rangle, \langle \mathbb{R} \rangle)$

$\tau($  $) := (\langle \mathbb{C} \rangle, \langle \mathbb{R}, \mathbb{R} \rangle)$

$\tau($  $) := (\langle \mathbb{R}, \mathbb{R} \rangle, \langle \mathbb{C} \rangle)$

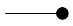
$\tau($ $) := (\langle \mathbb{R} \rangle, \langle \rangle)$

Note that the codomain of the discarding operation *is the empty list* $\langle \rangle$*, corresponding to the unit object of the category.*

Remark 2.3. What we call a *monoidal signature* is called a *tensor scheme* by Joyal and Street [70]. Definition 2.2 uses terminology closer to that of Selinger [92] and Bonchi et al. [18]. In addition, we will often omit an explicit definition of τ when, as in (2.1), its definition is clear from the labels in a diagrammatic representation.

In some cases, it is useful to consider 'polymorphic' operations. For example, the operations of (2.1) define the discarding operation only on $\mathbb{R}$ values. Suppose instead we wish to have a discarding operation for *every* object. We would like to handle this economically, so in such cases we will often define a polymorphic operation ranging over objects. An example of such a definition is below:

Example 2.2 (Example of a polymorphic generating morphism)**.** *For each object* $A \in \Sigma_0$*, there is a morphism*

A 

We take this to mean that there exists a number of such morphisms: one for each object in $A \in \Sigma_0$. In some cases, we may also allow A to range over elements of $\mathsf{List}(\Sigma_0)$.

As in Bonchi et al. [18], we now define Σ-terms: the composites built inductively from primitives by tensor and composition.

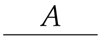
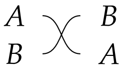
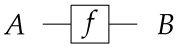
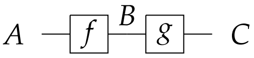
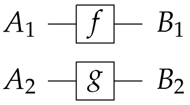
Definition 2.4 (Σ-terms [18])**.** Given a monoidal signature $\Sigma = (\Sigma_0, \Sigma_1)$, a Σ-term is defined inductively as follows:

- id_A is a Σ-term for all $A \in \mathsf{List}(\Sigma_0)$

- $\sigma_{A,B}$ is a Σ-term for all $A, B \in \mathsf{List}(Obj)$
- f is a Σ-term for each $f \in \Sigma_1$
- $f \fatsemi g$ is a Σ-term for Σ-terms $f : A \to B$ and $g : B \to C$
- $f \otimes g : A_1 \otimes A_2 \to B_1 \otimes B_2$ is a Σ-term for Σ-terms $f : A_1 \to B_1$ and $g : A_2 \to B_2$

Remark 2.5. The symmetry σ is assumed to be a Σ-term only in the case of symmetric monoidal categories. In the more general monoidal case, we omit it from the inductive definition.

We can now compare Σ-terms to the graphical language of string diagrams. Following [18, 92], we give a diagrammatic depiction in (2.3) of each of the primitives and composites of Definition 2.4.

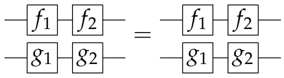
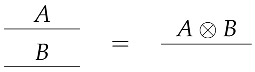
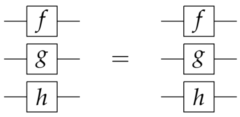
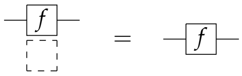
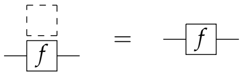
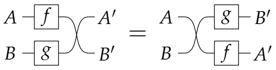
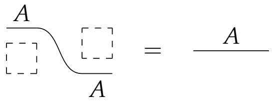
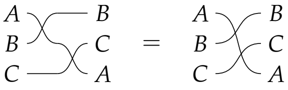
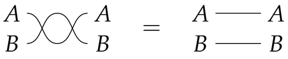
Name	Σ-term	Diagrammatic Syntax	
Unit	$\mathsf{id}_I : I \to I$	(empty)	
Identity	$\mathsf{id}_A : A \to A$		
Symmetry	$\sigma : A \otimes B \to B \otimes A$		(2.3)
Morphism	$f : A \to B$		
Composition	$f \fatsemi g$		
Tensor	$f \otimes g$		

Remark 2.6. In the above we make a minor notational abuse. In particular, objects A in general are elements of $\mathsf{List}(\Sigma_0)$. Thus, although we draw the identity $\mathsf{id}_A : A \to A$ as a single wire, for an element $A = \langle A_1, \ldots, A_n \rangle$ of $\mathsf{List}(\Sigma_0)$ we have the strict equality

$$\mathsf{id}_A = \mathsf{id}_{A_1 \ldots A_n} = \begin{array}{ccc} A_1 & \rule[0.5ex]{3em}{0.4pt} & A_1 \\ & \vdots\, n & \\ A_n & \rule[0.5ex]{3em}{0.4pt} & A_n \end{array}$$

We address the issue of 'bundling' wires in Chapter 6.

Now, in order to define the free symmetric monoidal category, we will quotient Σ-terms by the axioms of strict symmetric monoidal categories. We recall these now, giving a diagrammatic version of each axiom.

Axiom	Σ-term	Diagrammatic Syntax
$\otimes$ Functoriality	$(f_1 \otimes g_1) \fatsemi (f_2 \otimes g_2) = (f_1 \fatsemi f_2) \otimes (g_1 \fatsemi g_2)$	 f_1 f_2 g_1 g_2 = f_1 f_2 g_1 g_2
$\otimes$ Functoriality	$\mathsf{id}_A \otimes \mathsf{id}_B = \mathsf{id}_{A\otimes B}$	
α Naturality	$f \otimes (g \otimes h) = (f \otimes g) \otimes h$	
ρ Naturality	$f \otimes \mathsf{id}_I = f$	
λ Naturality	$\mathsf{id}_I \otimes f = f$	
σ Naturality	$(f \otimes g) \fatsemi \sigma_{A',B'} = \sigma_{A,B} \fatsemi (f \otimes g)$	 A f A' B g B' = A g B' B f A'
Unit Coherence	$\sigma_{A,I} = \mathsf{id}_A$	
Associator Coherence	$(\sigma_{A,B} \otimes \mathsf{id}_C) \fatsemi (\mathsf{id}_B \otimes \sigma_{A,C}) = \sigma_{A,B\otimes C}$	 A B C B C A = A B C B C A
Inverse Law	$\sigma_{A,B} \fatsemi \sigma_{B,A} = \mathsf{id}_A \otimes \mathsf{id}_B$	

(2.4)

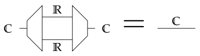
Notice that the pentagon and triangle axioms of monoidal categories are not included above; these become trivial in a strict monoidal category.

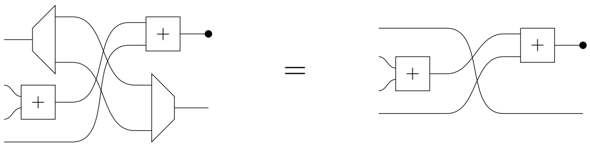
Remark 2.7. Many of the equations of strict symmetric monoidal categories are *implicit* in the syntax of string diagrams. For example, the naturality axiom of the associator for strict monoidal categories says that tensor product of morphisms must be associative on the nose. Since the diagrammatic tensor is simply to place one diagram atop the other, this strict associativity is 'built in'. Thus, we may not consider the diagrammatic syntax and Σ-terms interchangeable until we have quotiented the latter by the axioms of strict symmetric monoidal categories.

We can now define the free symmetric monoidal category on a signature.

Definition 2.8 (Symmetric Monoidal Category freely generated by a signature [18]). The **strict symmetric monoidal category freely generated by** Σ, denoted $\mathbf{Free}_\Sigma$, has objects $\mathsf{List}(\Sigma_0)$, with unit object the empty list $\langle\rangle$. Morphisms are Σ-terms quotiented by the axioms of strict symmetric monoidal categories (2.4). Identities, symmetry, and composition and monoidal product are given by the corresponding Σ-terms in (2.3).

We separate the definition of the free symmetric monoidal category over a signature from the definition of a presentation for two reasons. The first reason is for simple modularity: the latter will be expressed as a quotient of the former. The second more important reason is that reasoning modulo the laws of SMCs can be made simpler by appealing to the coherence theorem of Joyal and Street [70].

This theorem essentially says that for the laws of SMCs, instead of applying individual equational rewrites to Σ-terms, we may reason 'topologically' with a diagram. A simple example of such reasoning is as follows. Returning to the example signature (2.1) and equation (2.2), we may take the tangled diagram below left and immediately apply the equation  to obtain the diagram below right.



The flavour of reasoning modulo the laws of symmetric monoidal categories is therefore somewhat different to equational reasoning with Σ-terms. Instead of performing a sequence of individual equational rewrites, say by first using naturality of σ, we may instead use the *connectivity* of the diagram. This is of course a major selling point of string diagrams: this topological reasoning reduces the amount of book-keeping required in a proof.

Before defining presentations by operations and equations, we make one final definition. Again following [18], we define *symmetric monoidal theories*, which allow us to augment a monoidal signature with *equations*, such as those in (2.2).

Definition 2.9 (Symmetric Monoidal Theory)**.** From [18, Definition 2.1], a **Symmetric Monoidal Theory** Σ is a triple $(\Sigma_0, \Sigma_1, \Sigma_2)$ consisting of a monoidal signature equipped with a set of equations Σ_2. The elements of Σ_2 are pairs of Σ_1-terms $\langle l, r\rangle$ having the same domain and codomain (i.e., with $\tau(l) = \tau(r)$).

This leads us directly to the main definition.

Definition 2.10 (Presentation by Generators and Equations)**.** The **strict symmetric monoidal category presented by generators and equations** $(\Sigma_0, \Sigma_1, \Sigma_2)$ has objects elements of $\mathsf{List}(\Sigma_0)$ and morphisms equivalence classes of Σ_1-terms quotiented by (a) the equations of strict symmetric monoidal categories (2.4) and (b) equations Σ_2.

One special case of such presentations is that of **PROP**s.

Example 2.3 (**PROP**)**.** *A* **PROP** *is a category $\mathscr{C}$ presented by a single object $\Sigma_0 := \{1\}$. The objects of $\mathscr{C}$ are therefore the natural numbers: we identify the unit object–the empty word–with* 0*, while a list of length n is identified with n. The* types *of generating morphisms can be similarity identified, so each generator can be thought of having an* arity *and* coarity *instead of input and output types.*

Examples of useful PROPs include a presentation of boolean circuits due to Lafont [76]. We will define a related family of categories in Chapter 3.

Now that we have defined more clearly what is meant by a presentation, we will see some useful examples in the next section.

2.3 Cartesian Symmetric Monoidal Categories

Categories with *cartesian* symmetric monoidal structure will be particularly important to this thesis. The primary reason is that categories with reverse derivatives–used extensively in Chapter 3–have cartesian structure as a prerequisite. For that reason, we now explicitly define such categories.

We start by recalling the definition of the product.

Definition 2.11 (Product)**.** Let $\mathscr{C}$ be a category and A, B objects of $\mathscr{C}$. The product of A and B in $\mathscr{C}$ is an object denoted $A \times B$ together with projection maps $\pi_0 : A \times B \to A$ and $\pi_1 : A \times B \to B$ such that the following diagram commutes for all objects Q and morphisms f, g

$$\begin{tikzcd}
& Q \arrow[dl, "f"'] \arrow[d, dashed, "\langle f, g \rangle"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\pi_0"] \arrow[r, "\pi_1"'] & B
\end{tikzcd}$$

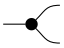
and the 'tupling' morphism $\langle f, g \rangle$ is the unique morphism making the diagram commute.

We will also need the definition of *terminal objects.*

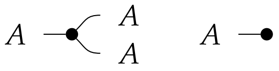
Definition 2.12 (Terminal Object)**.** An object in a category $\mathscr{C}$ is **terminal**, denoted $\mathbb{T}$, if for all objects $A \in \mathscr{C}$, there is a unique morphism called the **terminal map** denoted $! : A \to \mathbb{T}$.

Armed with these standard definitions, we can define cartesian monoidal categories.

Definition 2.13 (Cartesian Monoidal Category)**.** A **cartesian symmetric monoidal category** is a symmetric monoidal category whose tensor is the categorical product × and whose unit is the terminal object $\mathbb{T}$.

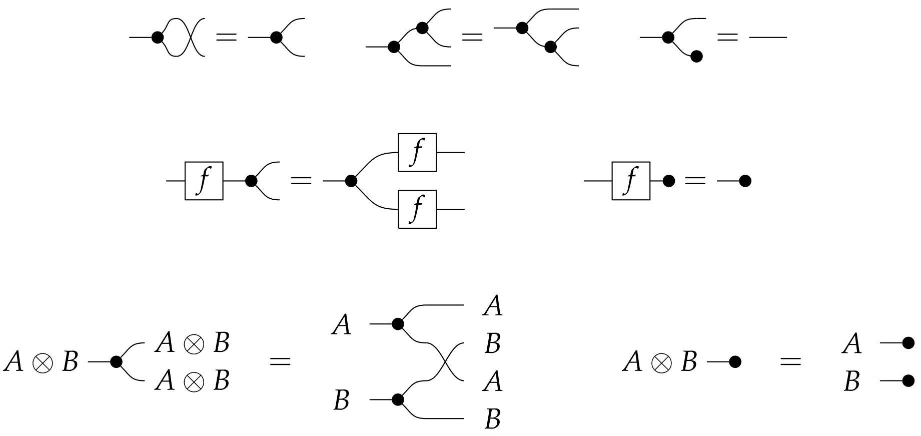
A particularly important theorem by Fox [53] relates cartesian monoidal categories to certain presentations which include cocommutative comonoids. Fox shows that that a category is cartesian if and only if it is equipped with copy  and discard morphisms satisfying certain axioms. Since the content of this thesis is primarily string-diagrammatic, this theorem is invaluable: we recall it now.

Theorem 2.14 (Fox's Theorem, [53])**.** *A category is cartesian iff each object A is equipped with a* copy *and a* discard *map:*



(2.5)

satisfying the following laws



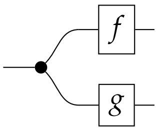
(2.6)

which say that the generators form a cocommutative comonoid, and that they must be natural with respect to the other morphisms of the category.

As a consequence of Fox's theorem, we may ensure that a category presented by generators and equations has products simply by adding the required generators (2.5) and equations (2.6) of Theorem 2.14. We will use this fact in the construction of polynomial circuits in Chapter 3. We may state idea this more formally by defining presentations of *cartesian* categories by generators and equations as follows.

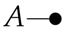
Definition 2.15. Let $(\Sigma_0, \Sigma_1, \Sigma_2)$ be a symmetric monoidal theory, and let ${\Sigma_1}'$ and ${\Sigma_2}'$ be the disjoint unions of Σ_1 and Σ_2 with operations (2.5) and equations (2.6), respectively. The **cartesian category $\mathscr{C}$ presented by generators and equations** $(\Sigma_0, \Sigma_1, \Sigma_2)$ is defined as the symmetric monoidal category presented by generators and equations $(\Sigma_0, {\Sigma_1}', {\Sigma_2}')$.

While we do not prove Fox's theorem in this thesis, it will be helpful to highlight how the string-diagrammatic presentation relates to the categorical product. We therefore give a brief description of this relationship now. First, the *tupling* map can be depicted diagrammatically as follows

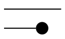
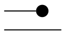
$\langle f, g \rangle$ = 

where in the special case of tupling identity maps we obtain the ‘copy’ generator $\langle \mathsf{id}, \mathsf{id} \rangle =$.

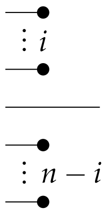
Similarly for the terminal object, the *terminal* map corresponds to the *discard* generator

$!_A$ = 

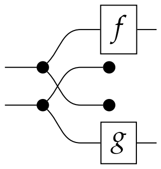
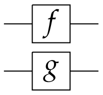
Projections are the tensor product of the identity and discard maps:

$\pi_0 =$  $\pi_1 =$ 

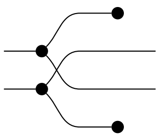
More generally, the i^{th} projection of the n-ary product is the morphism

$\pi_i =$ 

In addition, the tensor product can be written in terms of the tupling of morphisms composed with projections:

$\langle \pi_0 \fatsemi f, \pi_1 \fatsemi g \rangle$ =  =  = $f \times g$

Finally, the symmetry $\sigma : A \times B \to B \times A$ can be constructed similarly.

$\langle \pi_1, \pi_0 \rangle$ =  = = σ

Part I

Circuit Models and Machine Learning

Chapter 3

Polynomial Circuits

3.1 Overview

In this chapter, we will define a *model class* suitable for machine learning. For the purposes of this thesis, this will mean defining a *category* whose role is to define the space of possible models. More specifically, we will define a *family* of categories $\mathbf{PolyCirc}_S$ parametrised by a semiring S where a choice of morphism corresponds to a choice of model. The suitability of $\mathbf{PolyCirc}_S$ as a model class is evaluated in two respects: differentiability and expressivity. In the first case we will show how to equip $\mathbf{PolyCirc}_S$ with *reverse derivative* structure, which will allow models to be trained in a gradient-descent-like procedure we will describe in Chapter 4. In the second case we consider the *expressivity* of the model class. Models are considered 'sufficiently expressive' if, for an arbitrary dataset, there is a model which can faithfully represent it. This will lead us to give a 'functional completeness' theorem analogous to the various 'universal approximation' theorems for neural networks [67, 79].

An important feature of the categories we describe is that morphisms are specified using the graphical language of *string diagrams*. This is beneficial for a number of reasons. First, as a pedagogical tool, it makes the combinatorial structure of a given model clear from its formal specification, without having to rely on informal diagrammatic syntax as in e.g. [72, 100]. Secondly, the use of string diagrams unlocks a number of formal mathematical tools such as the framework of *double pushout rewriting* described in [17–20]. Finally, use of string diagrams will allow us to represent models–morphisms of these categories–using the datastructures we describe in Chapter 5. We therefore have a diagrammatic syntax which is completely formal while also having well-defined datastructures and algorithms for its representation and manipulation.

The model classes defined here also come with benefits to machine learning. In particular, recent neural networks literature contains a number of approaches for improving model performance [33, 34, 101] under the umbrella of 'quantisation' or 'binarisation'. In these approaches, network weights are quantised to small finite sets (for example $\{-1, +1\}$). This reduces the amount of information required to store weights and can improve the efficiency of computing arithmetic operations. In some cases, simply moving from 32 to 16-bit floating-point values can increase performance in terms of operations per second [108], while in extreme cases such as binarisation [33, 34] many operations can be completely elided. However, these approaches typically yield benefits only *after* the training process, and still require higher-precision floating point operations at train time.

By defining model classes which can be *trained* in the low precision regime, we hope to provide new avenues for model design. In contrast to existing neural networks literature which focuses on finding particular model architectures or training methods, our approach allows one to explore the model design space by varying the underlying semiring, and thus the corresponding notion of arithmetic. We will explore the consequences of one such choice–the semiring $\mathbb{Z}_2$–in Chapter 4.

In order to train models defined over an arbitrary semiring, the categories we define will need to have *reverse derivative* structure. Reverse derivatives will allow us to define 'gradient-like' learning algorithms in Chapter 4. However, in order to guarantee that models are sufficiently expressive, it will be necessary to give an alternative, more modular axiomatisation of reverse derivatives. Specifically, this axiomatisation will be in terms of presentations by generators and relations. This more modular approach will be immediately useful in ensuring the categories we define are functionally complete. In fact, we will see that $\mathbf{PolyCirc}_S$ is *not* functionally complete for every choice of semiring S. To address this we define the category $\mathbf{PolyCirc}_S^{=}$, which introduces an additional 'comparator' generator whose inclusion guarantees functional completeness. Moreover, by giving an 'extension theorem' for reverse derivative categories defined in terms of presentations, we will show how to retain reverse derivative structure in $\mathbf{PolyCirc}_S^{=}$.

3.1.1 Relationship to Published Work

Aside from the preliminaries covered in Section 3.3, the content of this chapter consists of the author's individual contributions originally published in [105] and [104]. Parts of this chapter are reproduced from these works verbatim.

3.2 Synopsis

In Section 3.3 we recall background on *reverse derivative categories* (RDCs) and their prerequisites. Our contributions begin in Section 3.4, where we introduce an alternative *graphical* formulation of RDCs more suited to categories defined in terms of presentations by generators and equations. We prove our alternative formulation equivalent in Theorem 3.14. The 'Extension Theorem' (Theorem 3.15) demonstrates how presentations of reverse derivative categories can be extended with new generators and equations so that they retain RDC structure. As a first example of this process, we introduce Cartesian Distributive Categories in Definition 3.18, a precursor to polynomial circuits.

In Section 3.5 we define the category of polynomial circuits $\mathbf{PolyCirc}_S$ (Definition 3.22), We show that $\mathbf{PolyCirc}_S$ can be given reverse derivative structure in Theorem 3.24, and give examples for particular semirings S. In Proposition 3.27, we give an isomorphism between $\mathbf{PolyCirc}_S$ and $\mathbf{POLY}_S$, a category of tuples of polynomials. This isomorphism justifies the interpretation of morphisms of $\mathbf{PolyCirc}_S$ as polynomials.

Finally, in Section 3.6 we define a notion of functional completeness (Definition 3.29) which allows us to formalise how a reverse derivative category has 'sufficient expressive power' to serve as a machine learning model class. We then give a simple condition under which a category may be considered functionally complete in Theorem 3.30, and show that for some choices of semiring S, the category $\mathbf{PolyCirc}_S$ is already functionally complete. However, for some choices of semiring S, we will see that $\mathbf{PolyCirc}_S$ is *not* functionally complete. Thus, in Definition 3.31, we define the category $\mathbf{PolyCirc}_S^{=}$ by extending $\mathbf{PolyCirc}_S$ with an additional generator. This extension will ensure that $\mathbf{PolyCirc}_S^{=}$ is functionally complete. Finally, we will show how the extension $\mathbf{PolyCirc}_S^{=}$ can retain RDC structure in Theorem 3.32, and in Corollary 3.33 we show that it is functionally complete for all choices of semiring S.

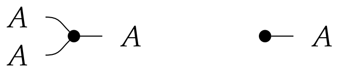
To conclude the chapter, we discuss these definitions from the perspective of machine learning model design. In Section 3.7.1 we show how a number of examples of neural network layers can be considered as morphisms of $\mathbf{PolyCirc}_S$. Finally, in Section 3.7.2 we discuss the impact of the choice of semiring S.

3.3 Preliminaries

In this section, we recall the definition of reverse derivative categories (RDCs). This structure, first introduced in Cockett et al. [29], is a necessary component for the 'gradient'-based learning we will define in Chapter 4.

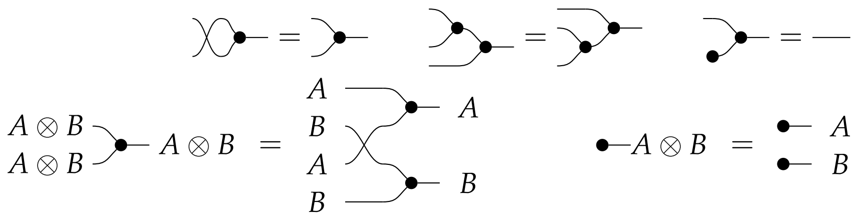
RDCs have some prerequisite structure. Namely, we must first define *cartesian left-additive* categories.

Definition 3.1 (Cartesian Left-Additive Category [13, 29]). A **Cartesian Left-Additive Category** is a cartesian category $\mathscr{C}$ where each object $A \in \mathscr{C}$ is equipped with a commutative monoid



(3.1)

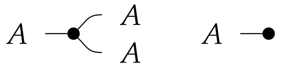
satisfying the equations



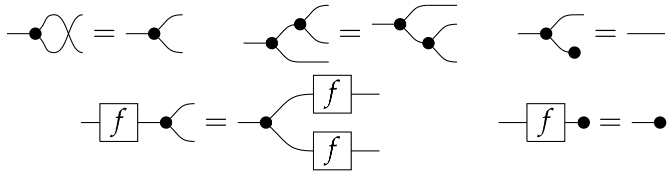
(3.2)

For the purposes of intuition, one may think of the morphism as an open circuit with two inputs whose output is their sum. When we later define polynomial circuits, which will have cartesian left-additive structure, this operation will be interpreted as the polynomial $\langle x_1, x_2 \rangle \mapsto \langle x_1 + x_2 \rangle$.

Note also that since Theorem 2.14 guarantees that any cartesian category comes equipped with a natural comonoid structure, any cartesian left-additive category $\mathscr{C}$ also has the operations and equations of Theorem 2.14 which we repeat here for completeness. That is, operations (2.5)



and equations (2.6)



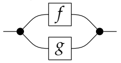
Since both cartesian and cartesian left-additive structure can be given in terms of generators and equations, we may now define the *cartesian left-additive* category presented by a signature. This will make it more convenient to speak of categories with the structure required to define an RDC already 'built-in'. We define such categories in the same way as for *cartesian* categories in Definition 2.15.

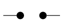
Definition 3.2 (Cartesian Left-Additive Category Presented by Generators and Equations [105]). Let $(\Sigma_0, \Sigma_1, \Sigma_2)$ be a symmetric monoidal theory, and let $\Sigma_1{}'$ and $\Sigma_2{}'$ be the disjoint unions of Σ_1 and Σ_2 with operations (3.1) and equations (3.2) of cartesian left-additive categories, respectively. The **cartesian left-additive category** $\mathscr{C}$ **presented by** $(\Sigma_0, \Sigma_1, \Sigma_2)$ is defined as the cartesian category presented by generators and equations $(\Sigma_0, {\Sigma_1}', {\Sigma_2}')$.

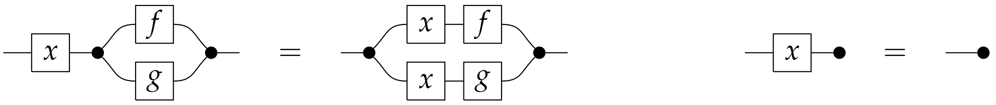
Observe that a cartesian left-additive category presented by generators and equations is indeed cartesian left-additive. It has *cartesian* structure guaranteed by the presence of generators and equations of Fox's theorem–that is, generators and , and equations (2.6). Moreover, it is *cartesian left-additive* because the signature includes the addition and zero generators of (3.1) and the necessary equations of (3.2).

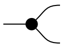
Remark 3.3. The definition of cartesian left-additive structure in this thesis is actually an alternative to the 'standard' definition given in [29, Definition 1] and [13, Definition 1.2.1]. The standard definition instead requires that hom-sets of cartesian left-additive categories come equipped with an addition operation $+$ and zero map 0, forming a commutative monoid satisfying the following axioms:

$$x \,⨟\, (f+g) = (x \,⨟\, f) + (x \,⨟\, g) \qquad\qquad x \,⨟\, 0 = 0$$

where x, f, g are morphisms. We may recover these axioms by defining $f + g :=$ 

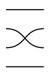
, and the zero morphism as $0 :=$ , so that the axioms can be represented in string-diagrammatic terms as follows:

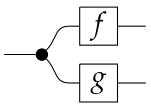


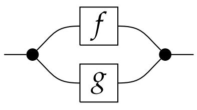
These equalities follow from Definition 3.1 thanks to the naturality of  and as required in Equation (2.6). A proof of the equivalence of Definition 3.1 and [29, Definition 1] can be found in [13, Proposition 1.2.2 (iv)].

With prerequisites in hand, we may now recall the original definition of reverse derivative categories from Cockett et al. [29, Definition 13]. However, we stress that in Definition 3.13 we will provide an alternative graphical definition: we will only use the original definition to prove equivalence of the two. Thus, the reader may safely skip the axioms in the following definition unless particularly interested in details of the proof of equivalence to our re-axiomatisation.

In recalling the original definition we will use the notation of the original paper, which uses several morphisms which we must first specify. We give these in string diagrammatic form below:

ι_0 := ι_1 := ex := 

$\langle f, g\rangle$:= 

$f+g$:= 

We can now proceed by giving the original definition of reverse derivative categories [29, Definition 13].

Definition 3.4 (Reverse Differential Category [29])**.** A **Reverse Differential Category** (RDC) is a cartesian left-additive category equipped with a combinator R of the following type

$$\frac{A \xrightarrow{f} B}{A \times B \xrightarrow[\mathsf{R}[f]]{} A}$$

satisfying axioms [**RD.1**] – [**RD.7**]:

[**RD.1**] $R[f+g] = R[f] + R[g]$ and $R[0] = 0$

[**RD.2**] $\langle a, b+c\rangle \fatsemi \mathsf{R}[f] = \langle a, b\rangle \fatsemi \mathsf{R}[f] + \langle a, c\rangle \fatsemi \mathsf{R}[f]$ and $\langle a, 0\rangle \fatsemi \mathsf{R}[f] = 0$

[**RD.3**] $\mathsf{R}[\mathsf{id}] = \pi_1$ and $\mathsf{R}[\pi_0] = \pi_1 \fatsemi \iota_0$ and $\mathsf{R}[\pi_1] = \pi_1 \fatsemi \iota_1$

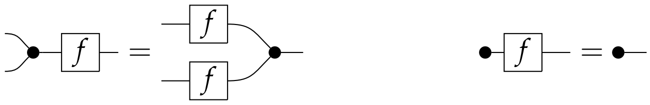
[**RD.4**] $\mathsf{R}[\langle f, g\rangle] = (\mathsf{id} \times \pi_0) \fatsemi \mathsf{R}[f] \times (\mathsf{id} \times \pi_1) \fatsemi \mathsf{R}[g]$ and $\mathsf{R}[!_A] = 0$

[**RD.5**] $\mathsf{R}[f \fatsemi g] = \langle \pi_0, \langle \pi_0 \fatsemi f, \pi_1\rangle\rangle \fatsemi (\mathsf{id} \times \mathsf{R}[g]) \fatsemi \mathsf{R}[f]$

[**RD.6**] $\langle \mathsf{id} \times \pi_0, 0 \times \pi_1\rangle \fatsemi (\iota_0 \times \mathsf{id}) \fatsemi \mathsf{R}[\mathsf{R}[\mathsf{R}[f]]] \fatsemi \pi_1 = (\mathsf{id} \times \pi_1) \fatsemi \mathsf{R}[f]$

[**RD.7**] $(\iota_0 \times \mathsf{id}) \fatsemi \mathsf{R}[\mathsf{R}[(\iota_0 \times \mathsf{id}) \fatsemi \mathsf{R}[\mathsf{R}[f]] \fatsemi \pi_1]] \fatsemi \pi_1$
$$= \mathsf{ex} \fatsemi (\iota_0 \times \mathsf{id}) \fatsemi \mathsf{R}[\mathsf{R}[(\iota_0 \times \mathsf{id}) \fatsemi \mathsf{R}[\mathsf{R}[f]] \fatsemi \pi_1]] \fatsemi \pi_1$$

Remark 3.5. As observed in [29], and despite their definition, ι_0 and ι_1 are not in general the injections of the coproduct. This is essentially because of the lack of naturality axiom in the definition of cartesian left-additive categories: i.e., we do *not* have the following axioms:



Should $\mathscr{C}$ be an RDC where these equations hold, then the left additive structure would coincide with the coproduct by the dual of Fox's theorem (2.14) and would be the coproduct of identity maps.

The intuition behind the reverse derivative is the following approximation.

$$f(x) + \delta_y \approx f(x + \mathsf{R}[f](x, \delta_y))$$

That is, for a given morphism $f : A \to B$, its reverse derivative $\mathsf{R}[f] : A \times B \to A$ approximates how much the input of f should change in order to achieve a given change in output (δ_y). We therefore think of the two 'arguments' to $\mathsf{R}[f]$ as having distinct roles: the first (x) is a *point*, while the second (δ_y) is a *change*.

That the reverse derivative is efficiently computable is essentially the same as the idea behind backpropagation in neural networks. This allows output errors to be 'backpropagated' through the model structure in order to compute a change in parameters. We describe precisely how this process is used in learning in Chapter 4.

Cockett et al. [29] also give two important examples of reverse derivative category which we will refer to later in the thesis. For completeness, we give their definitions now.

Definition 3.6 (**POLY**$_S$ (from [29])). **POLY**$_S$ is the symmetric monoidal category with objects the natural numbers and arrows $m \to n$ the n-tuples of polynomials in m indeterminates:

$$\langle p_1(\vec{x}), \dots, p_n(\vec{x}) \rangle : m \to n$$

with each

$$p_i \in S[x_1, \dots, x_m]$$

where $S[x_1, \dots x_m]$ denotes the polynomial ring in m indeterminates over S.

Remark 3.7. In Section 3.5 we will define a family of categories of 'polynomial circuits' over a semiring, denoted **PolyCirc**$_S$. In addition, we will give an isomorphism **PolyCirc**$_S \cong$ **POLY**$_S$. For that reason we do not give a direct definition of the reverse derivative of **POLY**$_S$ here; it can be found in [29, Example 14].

Definition 3.8 (**Smooth** (from [29])). The category **Smooth** has objects the natural numbers and maps $m \to n$ the smooth functions $f : \mathbb{R}^m \to \mathbb{R}^n$. The reverse derivative of a map $f : m \to n$ in **Smooth** is defined as follows:

$$\mathsf{R}[f](x, \delta_y) := J_f(x)^T \cdot \delta_y$$

where J_f is the jacobian of f.

3.3.1 Forward Derivatives from Reverse Derivatives

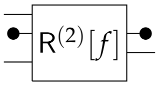
Although we only require reverse derivative structure for machine learning, Cockett et al. [29, Section 3.1] also show that a category $\mathscr{C}$ with reverse derivatives also admits *forward* differential structure. That is, if $\mathscr{C}$ is a reverse differential category, it is also a Cartesian Differential Category (see e.g., [13, 29]).

We do not give a full definition of such categories here. However, in order to simplify our exposition, we will rely on the definition of the induced forward differential operator. We picture this, as for reverse derivatives, as a derivation rule

$$\frac{A \xrightarrow{f} B}{A \times A \xrightarrow[\mathsf{D}[f]]{} B}$$

meaning that every morphism $f : A \to B$ has a *forward derivative* $\mathsf{D}[f] : A \times A \to B$. Note carefully the change in type of D versus R.

Definition 3.9 (Induced forward differential operator [29]). The **forward differential operator** induced by reverse differential structure is defined as

$\mathsf{D}[f] \quad :=$ 

where $\mathsf{R}^{(n)}$ denotes the n-fold application of R, and so $\mathsf{R}^{(2)}[f]$ is the 2-fold composition $\mathsf{R}[\mathsf{R}[f]]$.

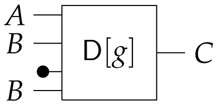
Remark 3.10. The intuition for the *forward* derivative is the approximation

$$f(x + \delta_x) \approx f(x) + \mathsf{D}[f](x, \delta_x)$$

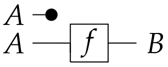
Note that in contrast to the reverse derivative operator R, we think of the forward derivative as mapping a change in input δ_x to a corresponding change in output $\mathsf{D}[f](x, \delta_x)$.

When we give our alternative axiomatisation of reverse derivative categories, we will treat this definition as purely syntactic shorthand to avoid issues of circularity. In order to make this axiomatisation, we will make use of two more such 'syntactic' definitions. We stress that we make no use of the axioms of cartesian differential categories in any of the proofs that follow.

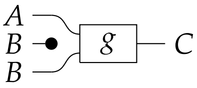
Definition 3.11 (Partial Derivative [29]). Graphically, the partial derivative of $g : A \times B \to C$ with respect to B is defined as follows:

$\mathsf{D}_B[g] \quad :=$ 

Definition 3.12 (Linearity [29]). We say a morphism $f : A \to B$ is **linear** when

$\mathsf{D}[f] \quad =$ 

Further, a morphism $g : A \times B \to C$ is **linear in** B when

$\mathsf{D}_B[g] \quad =$ 

In the next section, we will give our alternative axiomatisation of RDCs. This will allow us to demonstrate its 'modularity' with a theorem showing how new RDCs can be gradually extended with new operations and equations while retaining RDC structure.

3.4 RDCs for Categories Presented by Generators and Equations

We now give our alternative graphical definition of reverse derivative categories. The purpose of this redefinition is two provide a more modular basis from which to construct new reverse derivative categories. More concretely, we would like to take an existing RDC and add new operations to it while retaining the RDC structure. This will be important in Section 3.6, where in order to guarantee that a category satisfies the property of functional completeness we will have to augment it with a new generator.

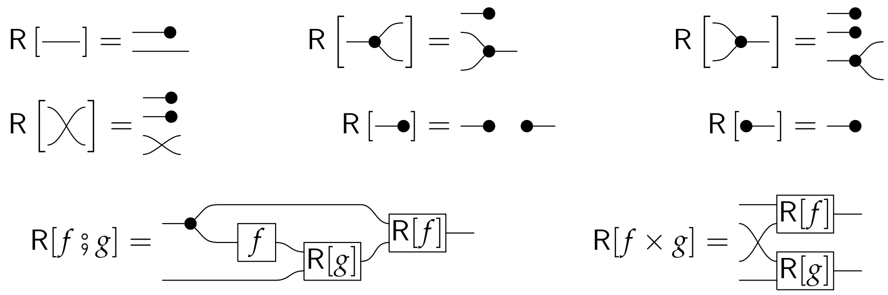
Our recipe for extending RDCs is as follows. We begin with Definition 3.13, which is our alternative axiomatisation of RDCs in terms of categories defined as presentations by generators and equations. Theorem 3.14 shows our redefinition is equivalent to the original. Theorem 3.15 then gives basic conditions under which such presentations can acquire RDC structure. An immediate corollary is that an RDC defined as a presentation by generators and equations can be *extended* to include new generating operations and equations while only requiring that the additional operations and equations be checked for compatibility with the equations of RDCs.

Let us now proceed with our alternative definition of RDCs.

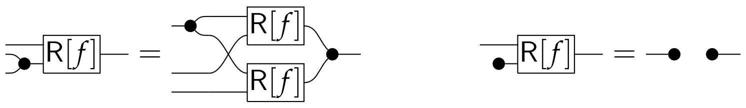
Definition 3.13. A **Reverse Derivative Category** is a cartesian left-additive category equipped with a *reverse differential combinator* R:

$$\frac{A \xrightarrow{f} B}{A \times B \xrightarrow[\mathsf{R}[f]]{} A}$$

satisfying the following axioms:
[ARD.1] (Structural axioms, equivalent to RD.1, RD.3-5 in [29])

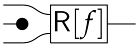


[ARD.2] (Additivity of change, equivalent to RD.2 in [29])



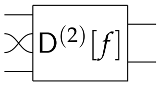
[ARD.3] (Linearity of change, equivalent to RD.6 in [29])

$$\mathsf{D}_B\,[\mathsf{R}[f]] =$$



[ARD.4] (Symmetry of partials, equivalent to RD.7 in [29])

$$\mathsf{D}^{(2)}[f] =$$



The following theorem shows that this definition is equivalent to the original definition of reverse derivative categories given in [29].

Theorem 3.14. *Definition 3.13 is equivalent to [29, Definition 13].*

Proof. The axioms RD.1-7 of [29, Definition 13] imply axioms ARD.1-4 by Proposition A.12. Conversely, axioms ARD.1-4 imply RD.1-7 by Proposition A.15. Therefore ARD.1-4 hold if and only if RD.1-7 hold, so Definition 3.13 is equivalent to [29, Definition 13]. □

We now give a theorem which describes the conditions under which a cartesian left-additive category presented by generators and equations is also a *reverse derivative* category. The idea is straightforward: since axiom ARDC.1 requires that the R operator is defined inductively, one need only check that the *generating* morphisms of a presentation satisfy the axioms of Definition 3.13 and that R is well-defined with respect to the equations.

Theorem 3.15 (Extension Theorem). *Let $\mathscr{C}$ be the cartesian left-additive category presented by generators and equations $(\Sigma_0, \Sigma_1, \Sigma_2)$ If for each operation $s \in \Sigma_1$ there is some $\mathsf{R}[s]$ which is well-defined (see Remark 3.16) with respect to Σ_2, and which satisfies axioms ARD.1-4, then $\mathscr{C}$ is a reverse derivative category.*

Proof. Observe that axioms ARD.1 fix the definition of R on composition, tensor product and the cartesian and left-additive structures. It therefore suffices to show that axioms ARD.2-4 are preserved by composition and tensor product. That is, for morphisms f, g of appropriate types, both $f \mathbin{⨟} g$ and $f \otimes g$ preserve axioms ARD.2-4. Thus, any morphism constructed from generators must also satisfy the axioms ARD.1-4, and $\mathscr{C}$ must be an RDC. We provide the full graphical proofs that ARD.2-4 are preserved by composition and tensor product in Appendix A.1. □

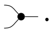
Remark 3.16. In Theorem 3.15, we required that R be well-defined with respect to equations Σ_2. Pragmatically, this means that we must check that $\mathsf{R}[l] = \mathsf{R}[r]$ for each $(l, r) \in \Sigma_2$. This is because the morphisms of the category $\mathscr{C}$ upon which R is defined are actually equivalence classes of diagrams. We must therefore have that $\mathsf{R}[f_1] = \mathsf{R}[f_2]$ for two diagrams $f_1 \sim f_2$ in the same equivalence class.

Example 3.1. *Axiom ARD.1 of Definition 3.13 completely defines the* R *operator on generators of cartesian left-additive categories* (3.1). *Therefore, the cartesian left-additive category presented by* $(\Sigma_0, \{\}, \{\})$ *trivially has RDC structure.*

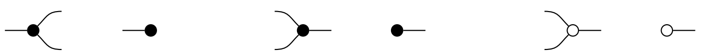
An immediate consequence of Theorem 3.15 is that if we have a presentation of an RDC $\mathscr{C}$, we can 'extend' it with an additional operation s, a chosen reverse derivative $\mathsf{R}[s]$, and equations $\Sigma_2{}'$, so long as R is well-defined with respect to $\Sigma_2{}'$ and the axioms ARD.2-4 hold for $\mathsf{R}[s]$.

Corollary 3.17 (Extension of RDC presentations). *Let* $\mathscr{C}$ *be an RDC presented by generators and equations* $(\Sigma_0, \Sigma_1, \Sigma_2)$, *Now let* $s : A \to B$ *be a new operation not in* Σ_1, *let* $\Sigma_2{}'$ *denote some additional equations involving this operation, and choose some definition of* $\mathsf{R}[s]$. *If* R *is well-defined with respect to* $\Sigma_2{}'$, *and* $\mathsf{R}[s]$ *satisfies axioms ARD.1-4, then the category* $\mathscr{D}$ *presented by* $(\Sigma_0, \Sigma_1 + \{s\}, \Sigma_2 + \Sigma_2{}')$ *is an RDC.*

More simply, this says that an existing RDC can be extended with new 'gadgets', providing they respect the conditions in Theorem 3.15.

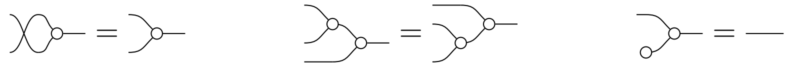
An immediately useful pair of such gadgets are the *multiplication* and *constant one* generators, which will distribute over the addition generator . We call such categories *cartesian distributive*, and define them as follows.

Definition 3.18. A **Cartesian Distributive Category** $\mathscr{C}$ is a cartesian left-additive category such that each object A is equipped with a commutative monoid and unit which distributes over the addition and is annihilated by the zero map . Explicitly, $\mathscr{C}$ has generators



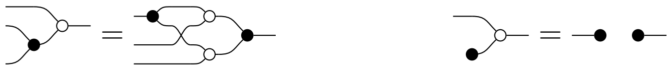
(3.3)

satisfying the *cartesianity* equations (2.6), the *left-additivity* equations (3.2), the *multiplicativity* equations



(3.4)

and the *distributivity* and *annihilation* equations

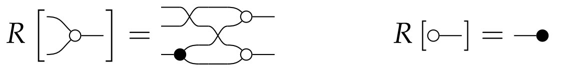


(3.5)

Remark 3.19. In the following sections, it will be useful to refer to particular examples of cartesian distributive categories having additional equations. We will therefore speak of presentations of cartesian *distributive* categories in the same way as we speak of *cartesian* and *cartesian left-additive* categories presented by generators and equations. More explicitly, this will mean that the generators and equations of 3.18 are assumed to be present in a given signature as with Definitions 2.15 and 3.2.

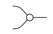
We can think of cartesian distributive categories as an extension of cartesian left-additive categories with an additional multiplication structure. However, note that the reverse derivative operator R on the generators of cartesian left-additive categories is completely defined by Definition 3.13. In contrast, for cartesian *distributive* categories, we must choose the action of R on the generators and . We must then use Theorem 3.15 to show that this indeed defines reverse derivative structure. This will allow us to use cartesian distributive categories as our 'base' category for defining polynomial circuits and their extensions in the next section.

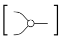
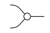
Theorem 3.20. *Let $\mathscr{C}$ be a cartesian left-additive category presented by $(\Sigma_0, \Sigma_1, \Sigma_2)$, and further suppose that $\mathscr{C}$ has RDC structure. Let $\mathscr{D}$ denote the cartesian* distributive *category presented by $(\Sigma_0, \Sigma_1, \Sigma_2)$ with* R *defined as follows.*



(3.6)

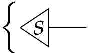
Then $\mathscr{D}$ is an RDC.

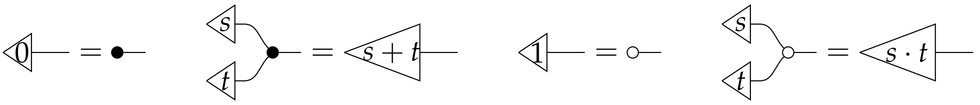
Proof. By Theorem 3.15, it suffices to check that R is well-defined (Proposition A.16) and satisfies axioms ARD.2-4 for the new generators. The generator satisfies the axioms because R [ ] = = R [ ], so it suffices to verify the axioms for  in Propositions A.19, A.20, and A.21 respectively. □

Remark 3.21. The definition of R[] and R[ ] is essentially a string diagrammatic version of the reverse derivative combinator defined on $\mathbf{POLY}_S$. Concretely, if we interpret  as the polynomial $(x_0, x_1) \mapsto x_0 \cdot x_1$, then its reverse derivative is $(x_0, x_1, \delta_y) \mapsto (x_1 \cdot \delta_y, x_0 \cdot \delta_y)$. Note however that cartesian distributive categories are a slightly more general formulation than $\mathbf{POLY}_S$ because a cartesian distributive category need not in general be a PROP.

3.5 Polynomial Circuits

We can now define *polynomial circuits*: the categories which will serve as the basis for our machine learning model class. The morphisms of this category can be thought of as representing polynomials over a commutative semiring S. The addition operation of S corresponds to the cartesian left additive structure, and multiplication to the cartesian distributive. In short, polynomial circuits are essentially cartesian distributive categories with a single generating object and some additional constants and equations. We define them as follows.

Definition 3.22. Let S be a commutative semiring. $\mathbf{PolyCirc}_S$ is the cartesian distributive category presented by objects $\{1\}$, arrows $\Big\{$$: 0 \to 1 \mid s \in S\Big\}$, and the 'constant equations'



(3.7)

for each $s, t \in S$.

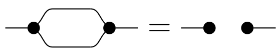
Remark 3.23. The 'constant equations' (3.7) of Definition 3.22 guarantee that addition and multiplication of 'constant' generators ◁s— is the same as that of addition and multiplication of constants in S.

Proposition 3.24. **PolyCirc**$_S$ *is an RDC with* R [◁s—] = —●.

Proof. The type of R [◁s—] : $1 \to 0$ implies that there is only one choice of reverse derivative, namely the unique discard map —●. Furthermore, R is well-defined with respect to the constant equations (3.7) for the same reason. Finally, observe that the axioms ARD.2-4 hold for R [◁s—], precisely in the same way as for R [●—], and so by Theorem 3.15 **PolyCirc**$_S$ is an RDC. □

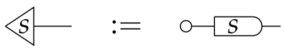
The requirement of Definition 3.22 that we add an axiom for each possible addition and multiplication of constants is somewhat uneconomical. For some significant choices of S, there is an equivalent smaller finite axiomatisation. We demonstrate these with some examples now.

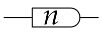
Example 3.2. *Choose $S = \mathbb{Z}_2$. Then the equations of Definition 3.22 reduce to the single equation*

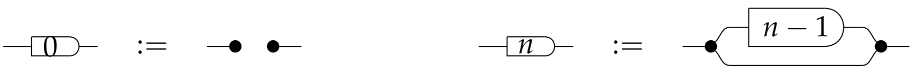


which expresses that $x + x = 0$ for each element x of $\mathbb{Z}_2$. Note that no further equations are needed to define multiplication of constants, which is completely defined on ●— and ○— by the axioms of cartesian distributive categories.

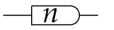
Example 3.3. *When S is the semiring of natural numbers $\mathbb{N}$ with the usual multiplication and addition,* **PolyCirc**$_{\mathbb{N}}$ *can be expressed without any additional equations. The required integer constants can be defined by repeated addition of ○— as follows:*



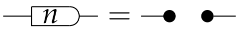
where  *is defined inductively as follows.*



The constant equations for **PolyCirc**$_{\mathbb{N}}$ *can then be derived using the axioms of cartesian distributive categories. This observation in fact makes* **PolyCirc**$_{\mathbb{N}}$ *the free cartesian distributive category on a single generating object.*

Remark 3.25. Note that we can regard the 'repeated addition' morphism  alternatively as multiplication of the input by n. Proving this lemma is a useful step in demonstrating that the inclusion of the constant equations for $\mathbf{PolyCirc}_{\mathbb{N}}$ is redundant.

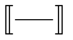
Example 3.4. *In a straightforward generalization of* $\mathbf{PolyCirc}_{\mathbb{Z}_2}$*, we can define* $\mathbf{PolyCirc}_{\mathbb{Z}_n}$ *in the same way, but with the only additional equation as*

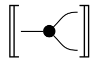


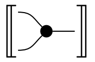
which says algebraically that $(1 + .\overset{n}{.}. + 1) \cdot x = n \cdot x = 0 \cdot x = 0.$

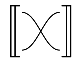
We can now describe the sense in which polynomial circuits can be thought of as polynomials. Concretely, there is an isomorphism $\mathbf{PolyCirc}_S \cong \mathbf{POLY}_S$. This amounts to saying that morphisms of the hom-set $\mathbf{PolyCirc}_S(m, n)$ correspond precisely to n-tuples of polynomials in m indeterminates, for $m, n \in \mathbb{N}$. To make this correspondence clear, we give the interpretation of $\mathbf{PolyCirc}_S$ as polynomials by defining the following functor.

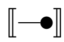
Definition 3.26 ($[\![\cdot]\!] : \mathbf{PolyCirc}_S \to \mathbf{POLY}_S$). Let $[\![\cdot]\!]$ be the strict symmetric identity-on-objects functor defined inductively on generators as follows.

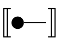
 $= \langle x_0 \rangle$

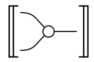
 $= \langle x_0, x_0 \rangle$

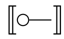
 $= \langle x_0 + x_1 \rangle$

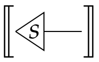
 $= \langle x_1, x_0 \rangle$

 $= \langle \rangle$

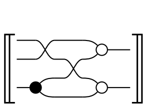
 $= \langle 0 \rangle$

 $= \langle x_0 x_1 \rangle$

 $= \langle 1 \rangle$

 $= \langle s \rangle$

Example 3.5. *For example, consider the polynomial circuit below, which has arity* 3 *and coarity* 2*. It will therefore be represented as a pair (or* 2*-tuple) of polynomials in* 3 *indeterminates, which we name* x_0, x_1, x_2*.*

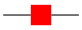
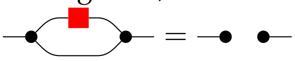
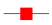
 $= \langle x_1 \cdot x_2, x_0 \cdot x_2 \rangle$

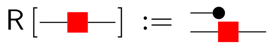
Using this interpretation gives us the following isomorphism.

Proposition 3.27 ($\mathbf{PolyCirc}_S \cong \mathbf{POLY}_S$).

We give a full proof of Proposition 3.27 in Appendix A.2. The essence of the proof is that homsets $\mathbf{PolyCirc}_S(m, n)$ and $\mathbf{POLY}_S$ both have the structure of the free module over the polynomial semiring $S[x_1, \ldots x_m]$. This means that there is a unique module isomorphism between homsets, and since $[\![\cdot]\!]$ is an identity-on-objects functor this is sufficient to complete the proof.

Remark 3.28. Recall that Cockett et al. [29] have already shown $\mathbf{POLY}_S$ to be a reverse derivative category. This of course raises the question: why bother with polynomial circuits at all? The reason is *modularity*. By developing polynomial circuits by incrementally adding algebraic structure and retaining RDC structure using Theorem 3.15, we are now in a position to add additional operations to $\mathbf{PolyCirc}_S$. This will be required in the next section to guarantee the property of 'functional completeness', which will allow us to express any desired function $S^m \to S^n$ as a corresponding (augmented) polynomial circuit. Moreover, the new 'extended' category based on $\mathbf{PolyCirc}_S$ will no longer be isomorphic to $\mathbf{POLY}_S$, and so will need to use Theorem 3.15 once again to gain RDC structure.

Finally, it will be useful to consider the special case of polynomial circuits over a bonafide *ring*, rather than just a semiring. For this case, we introduce additional syntax for negation, which we denote by . Additionally, we must include the equation  which can be thought of as saying that $x - x = 0$ for all $x \in S$. Naturally, we can also define the reverse derivative on  by appeal to Theorem 3.15:



3.6 Functional Completeness

We are now ready to consider the *expressivity* of the model class of polynomial circuits. More concretely, for a given commutative semiring S, we would like to guarantee that for any function between sets $f : S^m \to S^n$, there exists a diagram $d \in \mathbf{PolyCirc}_S(m, n)$ whose interpretation is f. Essentially this would guarantee that we can represent any function with some diagram in $\mathbf{PolyCirc}_S$. We call this property 'functional completeness' after the same property of sets of primitive logic gates in boolean circuits. Note however that in this section we will only show the functional completeness property for *finite* semirings S.

Functional completeness is important for machine learning. It ensures that one can always construct an appropriate model for a given dataset. Consider for example the task of supervised learning: the goal is to discover a function $f : A \to B$ from a dataset of input/output examples in $A \times B$. Considering the latter as the extensional description of a function, it is clear that we need functional completeness to guarantee that there is a circuit model which can faithfully represent any dataset. An analogous property for neural networks has also been studied; see for example the various 'universal approximation' theorems [67, 79].

We will now formally define the property of functional completeness. We will therefore need a category of sets and functions to serve as interpretations of

PolyCirc$_S$. For this we use **FinSet**$_S$: the cartesian monoidal category whose objects are natural numbers, and where morphisms $f : m \to n$ are functions of type $S^m \to S^n$. This category is analogous to **POLY**$_S$, but with hom-sets all functions, not just polynomials.

Definition 3.29. We say a category $\mathscr{C}$ is **functionally complete** with respect to a set S when there a full identity-on-objects functor $\mathsf{Fn} : \mathscr{C} \to \mathbf{FinSet}_S$.

The intuition for this definition is that a category $\mathscr{C}$ is 'functionally complete' when it serves as a syntax for **FinSet**$_S$. That is, the *fullness* of the functor Fn guarantees that any morphism in **FinSet**$_S$ may be expressed as a diagram in **PolyCirc**$_S$. It is not required that Fn be *faithful*, and so we may have unequal diagrams which represent the same function.

In general, **PolyCirc**$_S$ is not functionally complete with respect to S. Take for example the boolean semiring $\mathbb{B}$ with multiplication and addition as AND and OR respectively. It is well known [102] that one cannot construct every function of type $\mathbb{B}^m \to \mathbb{B}^n$ from only these operations.

Nevertheless, there is only one missing ingredient required to make **PolyCirc**$_S$ functionally complete. Namely, the 'comparator' operation, which represents the following function.

$$\texttt{compare}(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

We can state the requirement for the comparator function formally with the following theorem.

Theorem 3.30. *Let S be a finite commutative semiring. A category $\mathscr{C}$ is functionally complete with respect to S if and only if there is a monoidal functor* $\mathsf{Fn} : \mathscr{C} \to \mathbf{FinSet}_S$ *in whose image are the following functions:*

- $\langle\rangle \mapsto s$ *for each* $s \in S$ *(constants)*
- $\langle x, y\rangle \mapsto x + y$ *(addition)*
- $\langle x, y\rangle \mapsto x \cdot y$ *(multiplication)*
- `compare` *(comparison)*

Proof. Suppose $\mathscr{C}$ is functionally complete with respect to S, where S is a finite commutative semiring. Then by definition there is a functor $\mathsf{Fn} : \mathscr{C} \to \mathbf{FinSet}_S$ with each of the required functions in its image.

Now in the reverse direction, we will show that any function can be constructed only from constants, addition, multiplication, and comparison. The idea is that because S is finite, we can simply encode the function table of any function $f : S^m \to S$ as the following expression:

$$x \mapsto \sum_{s \in S^m} \texttt{compare}(s, x) \cdot f(s) \tag{3.8}$$

Further, since $\mathscr{C}$ is cartesian, we may decompose any function $f : S^m \to S^n$ into an n-tuple of functions of type $S^m \to S$. More intuitively, for each of the n outputs, we simply look up the appropriate output in the encoded function table. □

It follows immediately that **PolyCirc**$_S$ is functionally complete with respect to S if and only if one can construct the `compare` function in terms of constants, additions, and multiplications. We illustrate one example of constructing `compare` in this way below.

Example 3.6. **PolyCirc**$_{\mathbb{Z}_p}$ *is functionally complete for prime p. To see why, recall Fermat's Little Theorem [40], which states that*

$$a^{p-1} \equiv 1 (mod\ p)$$

for all $a > 0$. *Consequently, we have that*

$$(p-1) \cdot a^{p-1} + 1 = \begin{cases} 1 & \textit{if } a = 0 \\ 0 & \textit{otherwise} \end{cases}$$

We denote this function as $\delta(a) := (p-1) \cdot a^{p-1} + 1$ *to evoke the dirac delta 'zero indicator' function. To construct the* `compare` *function is now straightforward:*

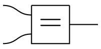
$$\texttt{compare}(x_1, x_2) = \sum_{s \in S} \delta(x_1 + s) \cdot \delta(x_2 + s)$$

Since $\mathbb{Z}_p$ *is a ring, the above can also be written more directly using negation.*

$$\texttt{compare}(x_1, x_2) = \delta(x_1 - x_2)$$

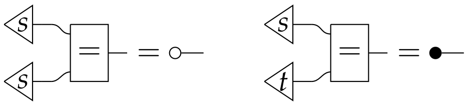
It is not possible in general to construct the `compare` function in terms of multiplication and addition. To guarantee functional completeness it is therefore necessary to *extend* the category of polynomial circuits with an additional comparison operation.

Definition 3.31. Define **PolyCirc**$^{=}_S$ as the cartesian distributive category presented by the same objects, operations, and equations of **PolyCirc**$_S$, plus an additional `compare` operation



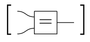
(3.9)

and equations



(3.10)

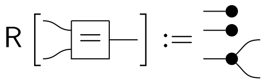
for $s, t \in S$ with $s \neq t$.

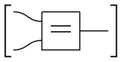
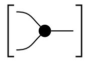
$\mathbf{PolyCirc}_S^{=}$ can be made a reverse derivative category by once again appealing to Theorem 3.15. We must therefore choose an appropriate definition of R  which is both well-defined and satisfies axioms ARD.1-4.

A suggestion for this choice comes from the machine learning literature. In particular, the use of the 'straight-through' estimator in quantised neural networks, as in e.g. [10]. Such models often make use of the dirac delta function, whose gradient is zero almost everywhere. This presents a problem when gradients are 'backpropagated': the zero derivative prevents 'information flow' from deeper layers to shallower ones. To remedy this issue, one instead replaces the 'true' gradient with the *straight-through estimator*. This instead passes through gradients unchanged from deeper layers to shallower ones.

In terms of reverse derivatives, this amounts to setting $\mathsf{R}[\delta] = \mathsf{R}[\mathsf{id}]$. Of course, it is necessary to define R for the full comparator, not just the zero-indicator function δ. We therefore we make the following choice.

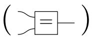
Theorem 3.32. $\mathbf{PolyCirc}_S^{=}$ *is an RDC with* R *as for* $\mathbf{PolyCirc}_S$*, and*



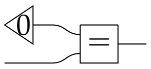
Proof. R is well-defined with respect to the equations (3.10) since both sides of each equation must equal the unique discard morphism . Further, R  satisfies axioms ARD.2-4 in the same way that R  does, and so by Theorem 3.15 $\mathbf{PolyCirc}_S^{=}$ is a reverse derivative category. □

Now the functional completeness of $\mathbf{PolyCirc}_S^{=}$ is a straightforward corollary of Theorem 3.30.

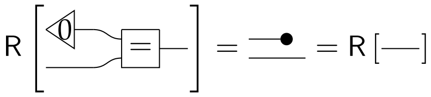
Corollary 3.33. *Fix a finite commutative semiring S.* $\mathbf{PolyCirc}_S^{=}$ *is functionally complete with respect to the underlying set of S.*

Proof. Let $\mathsf{Fn} : \mathbf{PolyCirc}_S^{=} \to \mathbf{FinSet}_S$ be the identity-on-objects strict symmetric monoidal functor defined as for $\mathbf{PolyCirc}_S$, and with Fn  equal to the comparison function. □

Finally, note that we recover the dirac delta function by 'capping' one of the comparator's inputs with the zero constant.

$\delta :=$ 

Taking reverse derivative of this composite yields precisely the 'straight-through' estimator:



3.7 Case Studies

We now discuss the impact of the choice of semiring S from the perspective of machine learning and model design. We begin in Section 3.7.1 by studying several common 'components' of neural network models as morphisms in $\mathbf{PolyCirc}_S$. While usually thought of as differentiable functions $\mathbb{R}^a \to \mathbb{R}^b$, the examples given here demonstrate that many such layers can be described more generally as morphisms in $\mathbf{PolyCirc}_S$ without needing to specify a particular semiring S. This suggests that one may be able to simply re-use existing neural network architectures with a different choice of underlying semiring.

In Section 3.7.2 we will propose two choices of *finite* semiring for machine learning with $\mathbf{PolyCirc}_S$. The implications of the choice of S will offer some new possibilities for model design outside of existing neural network architectures. However, we will also see that for some choices of finite S, re-using neural network architectures may not be effective.

Defining these basic components will prepare us for Chapter 4. There, we will show how several of the morphisms defined here can be combined to define models applied to some benchmark datasets.

3.7.1 Neural Network Layers as Morphisms in $\mathbf{PolyCirc}_S$

Neural Network models are typically expressed as a composition of 'layers'. From the categorical viewpoint, both layers and models are simply morphisms, and so there is no real distinction between the two. However, 'layer' is usually used to suggest a *component* of a larger model, rather than a standalone model by itself. Nevertheless, it will be useful to define some basic layers from which we may define larger models for the experiments in Section 4.6.2. Note that the neural network layers we describe here are all well-known; our contribution here is specifically to show how they can be thought of as morphisms in $\mathbf{PolyCirc}_S$.[1]

We can now define our first layer: the *linear* layer.

[1] For the sake of clarity, we will describe the layers in this section in terms of their interpretations as functions, but note that in each case there is a morphism of $\mathbf{PolyCirc}_S$ whose interpretation yields the given function.

Example 3.7 (Linear Layer [11])**.** *A **linear layer** can conceptually be thought of as the following map*

$$\begin{aligned}\texttt{linear} &: S^{ab} \times S^{a} \to S^{b} \\ \texttt{linear}(M, x) &:= Mx\end{aligned}$$

where M are the coefficients of a $b \times a$ matrix, and Mx denotes matrix-vector multiplication.

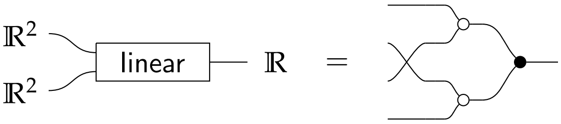
Since this map can be stated purely in terms of additions and multiplications, it can be expressed as a morphism in $\mathbf{PolyCirc}_S$. Consequently, it has a reverse derivative, which we can think of as the following map

$$\begin{aligned}\mathsf{R}[\texttt{linear}](M, x, \delta_y) &: S^{ab} \times S^{a} \times S^{b} \to S^{ab} \times S^{a} \\ \mathsf{R}[\texttt{linear}](M, x, \delta_y) &= (x\delta_y^T, M^T\delta_y)\end{aligned}$$

where here $x\delta_y^T$ denotes the outer product of vectors.

To clarify how `linear` is expressed in $\mathbf{PolyCirc}_S$, we give the following example.

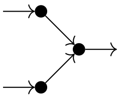
Example 3.8. *When $a = 2$ and $b = 1$, we have the following special case*



Remark 3.34. The theory of linear algebra over semirings is well-studied [58], and so the fact that the `linear` map generalises to $\mathbf{PolyCirc}_S$ is essentially well-known. Nevertheless, observe that while a matrix with coefficients in S is a linear map, the `linear` layer is not.

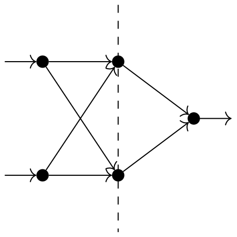
Not only is `linear` defined in $\mathbf{PolyCirc}_S$, but in fact in *any* cartesian distributive category (Definition 3.18). The reverse map of `linear` can therefore be obtained inductively by applying the reverse derivative as outlined in Definition 3.13.

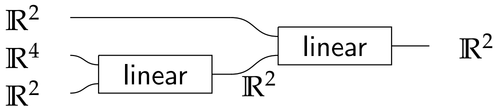
Before giving further examples, it is necessary to highlight an ambiguity in the 'layer' terminology as used in neural networks literature. We will therefore need to clarify the relationship between our string diagrammatic syntax and a 'traditional' informal graphical representation of neural networks. In this representation, the linear layer of Example 3.8 is represented by the following bipartite graph.



Here, the dangling left and right wires represent inputs and outputs, but not including parameters. Nodes represent *values*, edges represent the multiplication of a value by a weight (parameter), and a node's value is determined as the sum of its inputs. Thus, the edges of a fully connected bipartite graph correspond to the matrix coefficients of a linear map.

Unfortunately, the use of the term 'layer' can refer to both the *nodes* of the graph, as well as the linear map it represents. For example, the neural network defined as the composition of two linear layers is often said to have a 'single hidden layer'. Below depicts an example of this in the 'traditional' way (left) and the string-diagrammatic syntax (right):



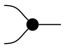


Here, the 'hidden layer' refers to those nodes falling on the dashed line. In string-diagrammatic terms, this refers to the 'internal wire' labeled $\mathbb{R}^2$. Thus the ambiguity in terminology amounts to confusing morphisms and objects. For that reason, *layer* will hereafter refer only to morphisms. Moreover, we will typically mean to indicate that a 'layer' is not intended to be a complete model by itself, but instead a component of a larger model.

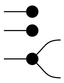
Let us now proceed to define our remaining examples. We will begin with *bias* and *activation* layers, which together with the `linear` layer of Definition 3.7 can be composed to define *dense* layers.

Example 3.9 (Bias Layer)**.** *A* ***bias layer*** *is simply an addition of parameters to inputs.*

`bias` :=



Its reverse derivative is then fixed by Definition 3.13 as follows.



Note that although defined explicitly here, the `bias` layer is not usually considered as a distinct layer in the neural networks literature.

Activation layers are another important class of layer. In neural networks, these are typically nonlinear maps post-composed with other layers. For example, given a neural network layer viewed as a map $f : \mathbb{R}^a \to \mathbb{R}^b$, an activation layer would usually be a map $\alpha : \mathbb{R}^b \to \mathbb{R}^b$.

Example 3.10. *Let A, B be generating objects of a reverse derivative category $\mathscr{C}$. An* ***activation layer*** *is a map $\alpha : A^n \to B^n$.*

Remark 3.35. Activation layers are frequently $1 \to 1$ functions applied 'pointwise'. That is, of the form $\alpha \otimes \ldots \otimes \alpha : A^n \to B^n$ for some chosen activation function $\alpha : A \to B$). However, this is not always the case. For example, in the *softmax* activation function [98, Section 2.8], outputs are normalised so the resulting vector sums to 1. In order to define this, each output depends on *all* of the inputs, and so the map cannot be considered as a pointwise application of a function.

A simple example of an activation function is the identity map. Note that the identity activation is not frequently used in practice, because the composition of two linear layers with identity activations is equivalent to a single linear layer by matrix multiplication.

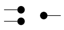
One useful example of an activation layer which is *not* in $\mathbf{PolyCirc}_S$ is the 'Rectified Linear Unit' (`ReLU`) [91, Section 5.24].

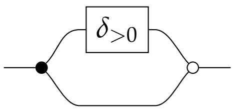
Example 3.11 (ReLU activation [91])**.** *The* `ReLU` *activation function is defined as*

$$\texttt{ReLU}(x) := \delta_{>0}(x) \cdot x$$

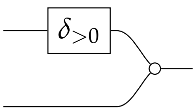
where $\delta_{>0}$ is the positive indicator function, applied pointwise to the elements of x. However, this map has a discontinuity at $x = 0$, and thus is also *not a map in* **Smooth**. *Nevertheless, in the neural networks literature, one typically computes the gradient of* `ReLU` *with the following map.*

$$\langle x, \delta_y \rangle \mapsto \delta_{>0}(x) \cdot \delta_y$$

Although `ReLU` is neither a map of **Smooth** nor $\mathbf{PolyCirc}_S$, we may obtain it by using Theorem 3.15 to extend $\mathbf{PolyCirc}_S$ with a new generator. Add the new generator $\delta_{>0} : S \to S$ and define its reverse derivative as the zero morphism $\mathsf{R}[\delta_{>0}] :=$ . We may now define `ReLU` graphically as follows:



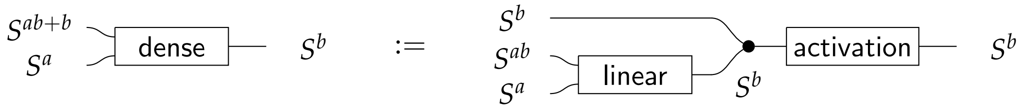
Taking the reverse derivative of this `ReLU` using the inductive definition in 3.13, we obtain



whose interpretation is the desired 'reverse' map as specified in Example 3.11.

Combining linear, bias, and activation layers allows us to define a *dense* layer.

Definition 3.36 (Dense Layer [27])**.** A **dense layer** is a composite of linear, bias, and a chosen activation layer.



The layers defined so far are sufficient for many simple machine learning tasks. For example, in Section 4.6.2 we will build a classifier for the Iris dataset using a composition of two `dense` morphisms. However, solving more complex problems such as image processing tasks will require additional definitions.

We now define two more complex layers which will be useful in solving the MNIST image classification task in Chapter 4. Each of these layers conceptually operates on *images*, thought of as two dimensional arrays of pixels. However, in terms of objects of a Reverse Derivative Category, we represent this two-dimensional array type as a single 'flat' array: an object of the form S^{wh} for an image of width w and height h.

The first such layer we define is a *convolutional* layer. A number of variations on this layer and its use in models occur in the machine learning literature; a survey can be found in Schmidhuber [91, Section 5.8]. The example we give here is therefore just one of many possible variants of what are termed 'convolution' layers. Nevertheless, we will use this specific definition for the purposes of describing our model later.

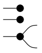
Definition 3.37 (Convolution layer [91])**.** A **2D convolution layer with size** k **convolution kernel** is a map `convolve` $: S^{k^2} \times S^{m^2} \to S^{n^2}$ which convolves a $k \times k$ kernel over an $m \times m$ input image. The output of the convolution layer is an $n \times n$ image where $n = \max(m, k) - \min(m, k) + 1$, i.e., so that we only consider the outputs where kernel and image completely overlap.

Note that the forward map of the convolution layer can be defined completely in terms of copying, addition, and multiplication. We can therefore consider this layer as a map in $\mathbf{PolyCirc}_S$, and obtain its reverse derivative inductively.

Remark 3.38. One often uses convolution layers with multiple input and output *channels*. Channels are often used when the data corresponding to each pixel is multidimensional, for example when one has three distinct *color* channels in the input. A convolution over an image with c_{in} input channels and c_{out} output channels can simply be thought of c_{out} independent convolutions with kernel size $c_{\mathsf{in}} \cdot k^2$ and image size $c_{\mathsf{in}} \cdot m^2$.

In image processing tasks, convolution layers are typically augmented by *max pooling* layers [91, Section 5.11]. Here, the idea is to partition the image into tiles, and apply the max operator to each tile.

Definition 3.39 (Max Pooling layer [91])**.** A **max-pooling layer** `maxpool` : $S^{(kn)^2} \to S^{n^2}$ computes the maximum of each of the n^2 size-$k \times k$ subregions of the input image.

Once again, we can appeal to the extension theorem in order to consider these maps and their reverse derivatives as morphisms in an RDC. In this case, we need an additional operation max : $2 \to 1$ whose reverse derivative we define as . The `maxpool` layer can then be expressed in terms of permutations and the max operation, and so its reverse derivative can be determined inductively.

Remark 3.40. Note that our implementation of layers involving convolution or matrix multiplication operations use specialised subroutines for these purposes, rather than using a full description of the polynomial circuit itself. Notably, the reverse derivative of the convolution layer can itself be expressed as a convolution. We might therefore think of these layers as being pairs of *functions*, rather than pairs of maps in an RDC. We discuss this further in Chapter 4.

3.7.2 Finite Semirings

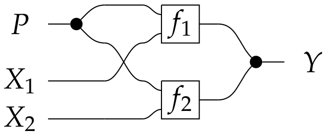
As we have seen, many components used to construct neural network models can be expressed more generally as morphisms in $\mathbf{PolyCirc}_S$. We can of course think of them in the usual 'neural networks' way where $S = \mathbb{R}$. However, in order to actually compute with such morphisms, we must eventually retreat to finite approximations. More concretely, to actually evaluate a neural network considered as a morphism of $\mathbf{PolyCirc}_{\mathbb{R}}$, one usually uses a *floating-point* approximation.

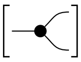
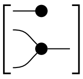
However, such approximations come with several drawbacks. First, floating-point arithmetic is relatively slow compared to e.g. integer operations [69]. Second, the floating point operations of addition and multiplication do not satisfy the semiring axioms and are not even associative, which results in problems of *numerical instability* [60]. Although attempts such as that of Gustafson and Yonemoto [60] exist to provide 'more well-behaved' approximations for the reals, these still do not satisfy the semiring axioms on the nose.

We therefore might reasonably ask if one can simply *start* with a finite semiring S and 'import' existing neural network architectures to the setting $\mathbf{PolyCirc}_S$. In this section, we will discuss the implications for two choices of semiring S on existing neural network layers. Additionally, we will see that varying S in fact allows us to consider new model architectures which are not possible for $S = \mathbb{R}$.

The first example we consider is where $S = \mathbb{Z}_2$; the ring of integers modulo 2. We have already seen in Example 3.6 that $\mathbf{PolyCirc}_{\mathbb{Z}_p}$ is functionally complete for prime p, and so there is no need to consider its extension $\mathbf{PolyCirc}^{=}_{\mathbb{Z}_2}$.

However, using a semiring of modular arithmetic in general introduces a problem: one must be careful not to construct models in which gradients 'wrap around'. Consider for example the model below, which is constructed of two independent sub-models f_1 and f_2 which both use the same parameters P[2] applied to different parts of the input X_1 and X_2.



Since R [] = R [], computing the reverse derivative for P will result in a sum of the input changes computed by $\mathsf{R}[f_1]$ and $\mathsf{R}[f_2]$. In the extreme case when the underlying semiring is $\mathbb{Z}_2$, then when the values of $\mathsf{R}[f_1]$ and $\mathsf{R}[f_2]$ are both 1, the result will 'wrap around', and the resulting change in P will be 0. This is clearly undesirable for learning: here we should prefer that $1 + 1 = 1$ to $1 + 1 = 0$.

Many of the neural network layers of Section 3.7.1 make use of copying. For instance, the parameters of a convolutional layer are the *convolution kernel*. If one regards a convolution as consisting of a tupling of independent circuits (as is possible due to cartesian structure) then it is clear that parameters are shared among *all* of the outputs. This layer therefore exhibits the 'wraparound' problem in $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ because its reverse derivative consists of a large summation. Thus, the direct use of neural network layers in $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ is unlikely to perform well in learning tasks.

We should therefore consider other models in the semiring $S = \mathbb{Z}_2$. In fact, in this setting, it is possible to define models which cannot be directly expressed when $S = \mathbb{R}$. For example, because $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ is functionally complete, we may express the morphism `eval` : $2^n \times n \to 1$ whose 2^n parameters simply encode an entire function table in the sense of Equation (3.8). Note that one can also construct an m-bit output `eval` morphism as the tupling of `eval`; a concrete implementation is given in Section 4.6.3.

Another possible choice of semiring S is the semiring Sat_n as a model of *saturating unsigned integer arithmetic* for a given 'precision' n. The underlying set is simply the finite set $\bar{\mathbf{n}} = \{0 \dots n-1\}$, with addition and multiplication defined as for the naturals but truncated to at most $n - 1$. We define Sat_n as follows, noting that it is equivalent to the semiring $B(n, n-1)$ defined in [2, Example 3] (see also [59]).

Definition 3.41. The semiring Sat_n has as addition and multiplication the operations

$$x_1 + x_2 := \min(n-1, x_1 + x_2) \qquad x_1 \cdot x_2 := \min(n-1, x_1 \cdot x_2)$$

[2] The re-use of weights in this way is known as 'weight-tying' in neural networks literature, but sharing of parameters also occurs in several of the layers we have already defined.

over the set $\bar{\mathbf{n}} := \{0 \ldots n-1\}$

Note that while Sat_n is a commutative semiring, it is certainly *not* a ring: the introduction of inverses means that the associativity axiom of semirings is violated. $\mathbf{PolyCirc}_{\mathsf{Sat}_n}$ is also not functionally complete. Thus, in order to obtain a model class which is functionally complete and is a reverse derivative category, we must use $\mathbf{PolyCirc}^{=}_{\mathsf{Sat}_n}$.

In Chapter 4 we will see examples of models in $\mathbf{PolyCirc}_{\mathbb{R}}$ and $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ applied to real data.

Chapter 4

Machine Learning with Circuits

4.1 Overview

Having defined a family of model classes suitable for machine learning–the category $\mathbf{PolyCirc}_S$ and its extensions–we now turn our attention to algorithms for learning the *parameters* of these models. We may think of morphisms of $\mathbf{PolyCirc}_{\mathbb{R}}$ in particular as neural networks, and a large number of effective techniques under the umbrella of 'gradient descent' are known to work well for training. A survey of these techniques can be found in Ruder [90], but basic idea of gradient descent in supervised learning is to iteratively 'show' example input/output data to a model, and measure its prediction error. Taking the gradient of this error with respect to the *parameters* of the model yields a 'change in parameters'; multiplying this change by a small constant (the *learning rate*) and subtracting from the model parameters constitutes moving in the 'direction of improvement' in parameter space.

In this chapter, we will show that these methods can be generalised to other choices of semiring S using reverse derivative structure. As a first example, we will define a new learning algorithm in terms of morphisms of $\mathbf{PolyCirc}_S$ over an arbitrary semiring S. We will also see how *existing* methods for gradient-based learning of neural networks can be described in terms of morphisms of $\mathbf{PolyCirc}_S$.

The need to define learning in this way addresses a recent need to improve the efficiency of deep learning models. In solving larger and more complex tasks, models have themselves become larger and more complex. This comes with a corresponding increased cost in computational resources: large models require expensive and power-hungry GPGPU hardware to train and run [34, 88].

This increased computational cost creates additional issues in some domains. For example, embedded and mobile devices are power constrained, so the efficiency of running trained models is important to effectively serve these settings. Another example is *privacy-sensitive* applications, where one might wish to *train* a model on private data. This might be the case even when using a pre-trained model: the privacy-sensitive end-user might wish to fine-tune[1] the model on personal data, but for large models this may still require expensive GPU hardware.

Improving efficiency is of course a benefit for its own sake as well. More *efficient* use of computational resources means more complex models can be created, and more difficult problems can be solved.

A recent approach to addressing these issues is that of *binarisation*. In this approach, one typically trains a neural network using $\mathbb{R}$-valued gradients, and then extracts a boolean circuit [101]. Analogously, one can think of this as learning the parameters of a model $f \in \mathbf{PolyCirc}_{\mathbb{R}}$, then somehow extracting a morphism $f' \in \mathbf{PolyCirc}_{\mathbb{Z}_2}$ which has similar performance on a given problem. In contrast, what we propose here is that the parameters of f' may simply be learned directly. This obviates the need for expensive floating point operations, even during the training phase.

The aims of this chapter are therefore twofold. Firstly, we should achieve our goal of defining a learning algorithm which is able to *directly* train models defined as morphisms in $\mathbf{PolyCirc}_S$. Importantly, this will be done without use of values in $\mathbb{R}$ (or more precisely, without floating-point approximations to such values). In this way, we can achieve our stated goal of providing an alternative to existing 'binarisation' approaches for neural networks. Secondly, we should provide a modular, compositional characterisation of learning which captures existing gradient-based techniques for training neural networks. Moreover, we should do this in a *diagrammatic* way: this will allow us to use the datastructures and algorithms we will develop in Part II.

In addition to efficiency benefits, the procedure described here can also be viewed as opening a new avenue in model design. Currently, new problems are typically solved by designing novel architectures. For example, the introduction of convolutional neural networks with max-pooling improved the state of the art on the MNIST problem [91, Section 5.19] In addition, there are even several approaches [45] to using neural networks to discover which architectures perform well. Instead, the idea of Part I of this thesis is that we may also change the underlying *arithmetic* of the model through selection of the semiring S.

[1] See for instance Erhan et al. [46] for details of fine-tuning a pre-trained model.

4.1.1 Relationship to Published Work

The content in this chapter is based on the papers Wilson and Zanasi [104] and Cruttwell et al. [36]. As usual, the preliminary content in Section 4.3 is background material, and does not contain original contributions of the author. Additionally, the contents of Section 4.5 are the result of joint work published in Cruttwell et al. [36][2]. Contributions solely due to the author are contained in Sections 4.4, 4.6, and 4.7. In particular, the contents of 4.4 and 4.7 are based on Wilson and Zanasi [104], and Section 4.6 is based on the implementations and experimental work for [104] and [36], both of which are the author's individual contribution.

4.2 Synopsis

We begin with preliminaries in Section 4.3 where we recall the functoriality of the reverse derivative operator demonstrated in Cockett et al. [29]. We then recall the construction of *lenses*, and discuss how the functoriality of the reverse derivative may be considered as an embedding into the category of lenses.

Our contributions begin in Section 4.4, where we discuss the role of the reverse derivative in learning, and introduce the *Reverse Derivative Ascent* algorithm (Definition 4.6) for learning the parameters of morphisms in $\mathbf{PolyCirc}_{\mathbb{Z}_2}$.

In Section 4.5, we recall the more general framework for learning based on lenses first described in our joint work [36]. We first view Reverse Derivative Ascent within this framework in Example 4.3, and then show that it is also able to capture more complex learning methods from the machine learning literature such as gradient descent with momentum (Example 4.6).

We demonstrate the practicality of our approach in Section 4.6 where we give two case studies solving real-world problems with both boolean circuits and neural networks. Section 4.6.2 demonstrates that our method, when applied to the case of neural networks, gives equivalent results to the same model trained in an existing deep learning framework. Meanwhile, Section 4.6.3 demonstrates that the method of Reverse Derivative Ascent as applied to boolean circuits is able to learn even without the use of real-valued gradients. Both case studies are accompanied by implementations, which can be found in Appendix D.1.

We conclude in Section 4.7 with a discussion of subtleties arising from our implementation, which is in terms of *functions*, not morphisms of $\mathbf{PolyCirc}_S$. This will

[2] Although based on this paper, we use a formulation of the material not involving some constructions appearing in the final published version.

motivate Part II of the thesis, where we show how morphisms of $\mathbf{PolyCirc}_S$ can be represented combinatorially.

4.3 Preliminaries

Before defining learning algorithms in Section 4.4, we recall some preliminaries. We first review the sense in which reverse derivatives are functorial, and then discuss a connection to lenses.

4.3.1 Reverse Derivatives as Lenses

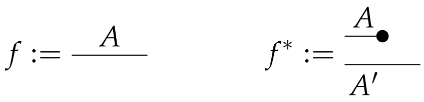
The reverse derivative operator is not a functor on its own. When taking the reverse derivatives of two maps, $f : A \to B$ and $g : B \to C$, one clearly cannot compose their reverse derivatives:

$$\mathsf{R}[f] : A \times B \to A \qquad\qquad \mathsf{R}[g] : B \times C \to B$$

However, there *is* a functorial construction arising from the reverse derivative. Namely, Cockett et al. [29] show that one can construct a category consisting of *pairs* of maps, and a functor from a reverse derivative category into this category of pairs.

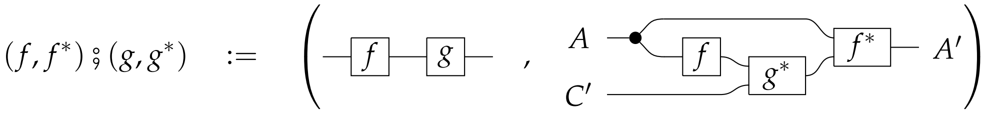
We first recall the definition of this 'category of pairs', $\widetilde{\mathsf{Lin}}^*$, first defined in [29, Example 29].

Definition 4.1 ($\widetilde{\mathsf{Lin}}^*$, [29]). Let $\mathscr{C}$ be a cartesian category. The monoidal category $\widetilde{\mathsf{Lin}(\mathscr{C})}^*$ has objects pairs of objects of $\mathscr{C}$, denoted $\binom{A}{B}$ for $A, B \in \mathscr{C}$. The morphisms of $\widetilde{\mathsf{Lin}(\mathscr{C})}^*$ are pairs of arrows of $\mathscr{C}$ so that $(f, f^*) : \binom{A}{A'} \to \binom{B}{B'}$ is a pair with $f : A \to B$ and $f^* : A \times B' \to A'$. Require also that f^* is *linear* in A (see Definition 3.12). The identity $\mathsf{id} : \binom{A}{A'} \to \binom{A}{A'}$ is the pair (id, π_1), rendered graphically below.



(4.1)

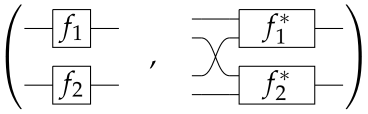
Composition is defined on maps $(f, f^*) : \binom{A}{A'} \to \binom{B}{B'}$ and $(g, g^*) : \binom{B}{B'} \to \binom{C}{C'}$ as the pair



(4.2)

Tensor product of maps $(f_1, f_1^*) : \left(\begin{smallmatrix} A_1 \\ A_1' \end{smallmatrix}\right) \to \left(\begin{smallmatrix} B_1 \\ B_2' \end{smallmatrix}\right)$ and $(f_2, f_2^*) : \left(\begin{smallmatrix} A_2 \\ A_2' \end{smallmatrix}\right) \to \left(\begin{smallmatrix} B_1 \\ B_2' \end{smallmatrix}\right)$ is given by

$$(f_1, f_2^*) \otimes (f_1, f_2^*) \quad := \tag{4.3}$$



Functoriality of the reverse derivative can then be stated as in [29, Proposition 31] as follows:

Proposition 4.2 (Functoriality of the Reverse Derivative [29]). *Let $\mathscr{C}$ be a reverse differential category. For objects A and morphisms f, define $\mathsf{F} : \mathscr{C} \to \widetilde{\mathsf{Lin}(\mathscr{C})}^*$ as follows:*

$$\mathsf{F}(A) := \begin{pmatrix} A \\ A \end{pmatrix} \qquad\qquad \mathsf{F}(f) = (f, \mathsf{R}[f])$$

Then F is a monoidal functor.

We may translate the proof in Cockett et al. [29] to our alternative formulation of RDCs straightforwardly:

Proof of Proposition 4.2. Functoriality of the reverse derivative follows directly from axiom [**ARD.1**] of Definition 3.13. Note that composition and tensor product in $\widetilde{\mathsf{Lin}(\mathscr{C})}^*$ is precisely the same as the reverse derivative of composites $\mathsf{R}[f \fatsemi g]$ and products $\mathsf{R}[f \times g]$, respectively. Moreover, the operation of R on identities gives the reverse maps of identities in $\widetilde{\mathsf{Lin}(\mathscr{C})}^*$. □

Now we may see an obvious parallel to the definition of *bimorphic lenses* defined as for example in Hedges [63, Definition 1].

Definition 4.3 (Bimorphic Lens [63]). Let $\mathscr{C}$ be a cartesian category. The category of **Bimorphic Lenses** is denoted $\mathsf{BiLens}(\mathscr{C})$, and has as objects pairs of objects of $\mathscr{C}$, and morphisms pairs $(f, f^*) : \left(\begin{smallmatrix} A \\ A' \end{smallmatrix}\right) \to \left(\begin{smallmatrix} B \\ B' \end{smallmatrix}\right)$ where $f : A \to B$ and $f^* : A \times B' \to A'$. Identities, composition, and monoidal products are as in Definition 4.1, i.e., equations (4.1), (4.2), and (4.3) respectively. We will call the morphisms of $\mathsf{BiLens}(\mathscr{C})$ **lenses**, and say that a lens $(f, f^*) : \left(\begin{smallmatrix} A \\ A' \end{smallmatrix}\right) \to \left(\begin{smallmatrix} B \\ B' \end{smallmatrix}\right)$ is **simple** when $A = A'$ and $B = B'$. Moreover, we will often refer to the components f and f^* as the **forward** and **reverse** maps, respectively.

It is clear that the definition of BiLens is the same as that of $\widetilde{\mathsf{Lin}}^*$ but without the requirement of linearity in the second component f^* of a morphism (f, f^*). In fact, it appears that this definition also appears in Cockett et al. [29, Example 29] as the 'dual of the simple fibration', although the connection to lenses is not made explicit.

As an immediate consequence, Proposition 4.2 may be extended to a functor to BiLens without changes to its definition. Thus, we may summarise by saying that reverse derivative categories can be embedded into categories of lenses via the reverse derivative.

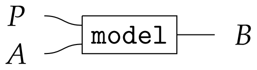
4.4 Reverse Derivative Ascent

We now have all the pieces required to define our first learning algorithm: Reverse Derivative Ascent. Before giving the algorithm itself, we will motivate its design by describing the role of the reverse derivative in supervised learning.

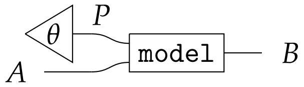
4.4.1 Reverse Derivatives and Learning

The reverse derivative is the key to the learning algorithms presented here. In fact, the intuition for reverse derivatives as computing *changes* leads to a natural definition of an algorithm for learning defined for any model expressible as a morphism of a $\mathbf{PolyCirc}_S$ for some ring S.

In supervised learning, the objective is typically to learn a map $f : A \to B$ from a dataset of examples $a \times b : I \to A \times B$. Based on prior beliefs about the structure of the dataset, one first designs a *model*: a morphism



whose parameters P define a search space over maps of type $A \to B$. [3] More precisely, each choice of parameter $\theta : I \to P$ yields a trained model



which is a map of the desired type. Choices of θ are typically evaluated with respect to some metric such as prediction accuracy.

Example 4.1. *When $P = I$, the model has the trivial parameter space, and so the untrained model is the same as the trained model: nothing can be learned.*

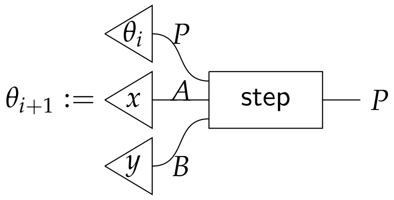
Example 4.2. *Suppose* `model` *is a morphism in $\mathbf{PolyCirc}_{\mathbb{Z}_2}$, and $P = \mathbb{Z}_2$. In this case, the model has a parameter space consisting of just two elements, and so there are only two possible trained models.*

[3] The process of designing such a model is largely beyond the scope of this thesis, but we give some real-world examples in Section 3.7.

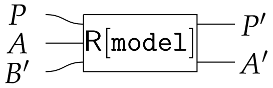
Now, in gradient-based learning, the idea is to successively improve an initial 'guess' of parameters $\theta_0 : I \to P$ by repeatedly showing examples $(x, y) : I \to A \times B$ to the model. Thus, for a single 'step' of learning, we require a map

$$\mathsf{step} : P \times A \times B \to P$$

so that the improved guess θ_{i+1} is given by the following composition.



If we examine the type of the reverse derivative of our model, it is clear that we almost a map of the required type:



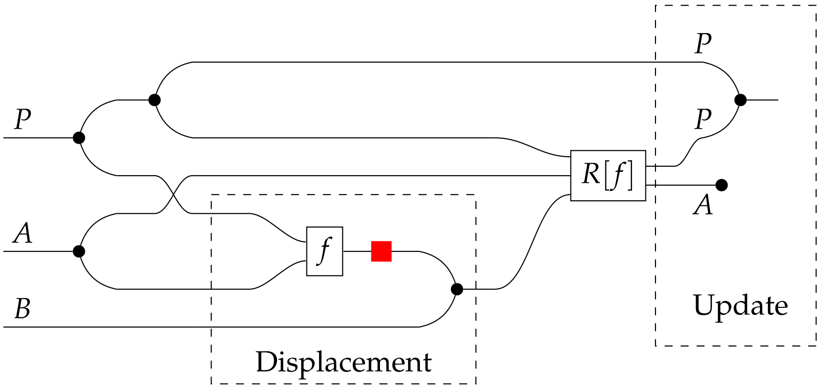
Note that here we have marked certain objects with a prime (e.g. B'). In fact, the reverse derivative requires that $X = X'$ for each object X in the above; we use this notation to distinguish the *roles* of these inputs and outputs. Concretely, the B' input represents a *change in output*, and the two outputs P' and A' represent a *change in input*.

In order to define the inner step of our algorithm, we therefore need two additional pieces. First, a morphism producing a new parameter value P, and second a morphism producing a *change in output* value B'. Making certain selections for these two morphisms yields the Reverse Derivative Ascent algorithm, which we can now define.

4.4.2 Reverse Derivative Ascent

We will define the Reverse Derivative Ascent algorithm in two parts. First, the inner 'step' of the algorithm, which we call `rdaStep`. Repeated applications of this inner step will form the algorithm itself, denoted `rda`.

Definition 4.4 (`rdaStep`)**.** Fix an arbitrary commutative ring S. Let $f : P \times A \to B$ be a morphism of $\mathbf{PolyCirc}_S$, where P represent parameters to be learned, and A represent inputs to the model. We define $\texttt{rdaStep}_f$ as the following morphism



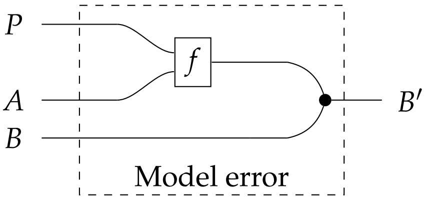
(4.4)

The morphism in 4.4 consists of the reverse derivative of the model f augmented with two choices of morphism: the *displacement* and *update* maps, which we have highlighted in (4.4).

First, the *displacement* computes the model error as the difference between the model prediction $f(\theta, x)$ and the 'true label', b. We may therefore think of the model error as a desired 'change in output' $\delta_y := y - f(\theta, x)$.

Second, the *update* morphism computes an updated parameter simply using the addition morphism ≻•– of cartesian left-additive structure. Intuitively, this makes sense: we take the current parameter value θ_i, and add to it the change in parameters δ_{θ_i} produced by the reverse derivative to obtain our new parameter value $\theta_i + \delta_{\theta_i}$, where δ_{θ_i} is the change in parameters given by the first component of the reverse derivative $\mathsf{R}[f](\theta_i, x, \delta_y)$. Note also that we discard the value of the A' wire; we will not require it for training the model.

Remark 4.5. In Section 4.6 we will give experimental results using Reverse Derivative Ascent for models in $\mathbf{PolyCirc}_{\mathbb{Z}_2}$. Elements of $\mathbb{Z}_2$ are self-inverse, and so we have the equation –🟥– = —. We can therefore express the model error as a generic morphism in any reverse derivative category as in our original paper [104].



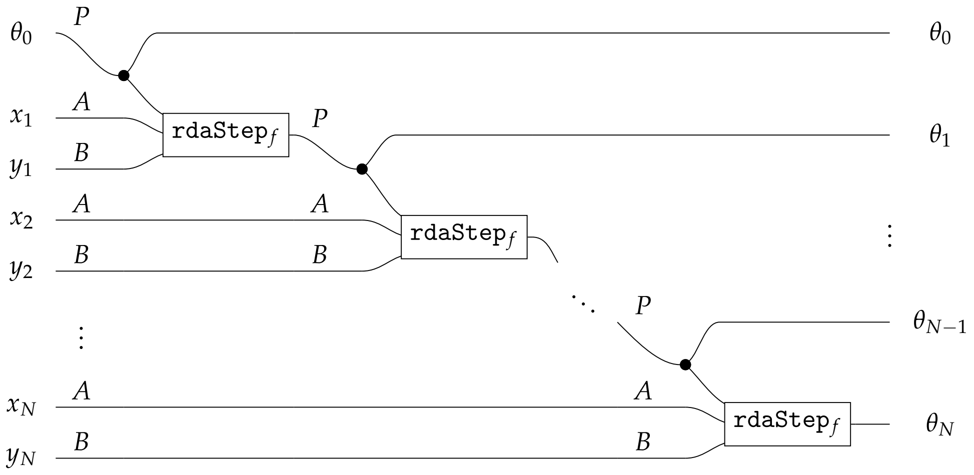
Being defined in any cartesian left-additive category, this is technically a more general formulation. However, this seems unlikely to work except in settings like $\mathbf{PolyCirc}_S$ for rings S with self-inverses.

Now, $\texttt{rdaStep}_f$ represents a single iteration of the full $\texttt{rda}$ algorithm. Its purpose is to map the current 'best guess' parameter $\theta_i : I \to P$ and an input/output example pair

$x \times y : I \to A \times B$ to produce an updated (and hopefully 'better') parameter $\theta_{i+1} : I \to P$.

Of course, a single parameter update is not sufficient for learning: we typically have a dataset of many examples, each of which we would like to show to the model more than once. In this case, we can simply apply the `rdaStep` operation repeatedly to the dataset, choosing new examples each time. This is the 'ascent' part of reverse derivative ascent, and we define it as follows:

Definition 4.6. Let $n \in \mathbb{N}$, and let $(x_i, y_i) \in (A, B)$, denote a sequence of n examples. $\mathtt{rda}_f$ is defined as the following morphism



(4.5)

Remark 4.7. Note that in code, we would typically implement `rdaStep` using a reduce-like operation; our implementation (see Implementation D.1) uses the `scanl` operation in Haskell. We define `rda` here as a string diagram to emphasize its generality as a morphism in a reverse derivative category.

In general, it is not necessary for elements (x_i, y_i) to correspond directly to the elements of the dataset. More commonly, this sequence will be a permutation of the dataset with repetitions as in stochastic gradient descent [90, 6.1].

As a final remark, consider the extreme case of reverse derivative ascent as applied to $\mathbf{PolyCirc}_{\mathbb{Z}_2}$, which we will explore in a case study in Section 4.6.3. Here, values and changes can be thought of as vectors of *bits*. Addition is XOR, and multiplication is AND. Thus, we may consider the reverse derivative as informing us which bits of the parameter bitvector to 'flip'. In contrast to the 'smoother' approach of gradient descent in $\mathbb{R}$, bit flipping is an all-or-nothing affair: there are no 'small' changes that can be made in a given dimension of the parameter vector.

In the next section (4.5) we will see how this algorithm can be reframed into the more general setting of lenses. In particular, `rdaStep` will become the reverse map of a lens. However, note that the algorithm as described can be applied as-is: we will give experimental results for the choice of semiring $S = \mathbb{Z}_2$ on some real-world datasets in Section 4.6.

4.5 Lenses and Learning

In the previous section, we saw how to define the `rda` learning algorithm by iterating the `rdaStep` morphism. A single step of learning consisted of the reverse derivative augmented with two things. First, a 'displacement' map for computing the model error, and secondly an 'update' map computing the updated parameters.

In fact, by allowing the 'update' and 'displacement' maps to vary, we can give a more general recipe for learning in terms of composition of *lenses*. Not only can this more general formulation capture the case of `rda`, but also more complicated existing update schemes from the neural networks literature. One such example of these is the stateful *momentum* gradient update as used in stochastic gradient descent. While this section is restricted to the theoretical formulation of these concepts, in Section 4.6 we give experimental evidence that our approach is correct. More precisely, we include experiments to show (a) that our formulation of the momentum gradient update gives equivalent performance to a model expressed in an existing machine learning framework and (b) that the Reverse Derivative Ascent algorithm is able to learn non-trivial functions from real-world data.

Note that aside from Example 4.3, the content of this section corresponds to the joint work in Cruttwell et al. [36] and is not solely the contribution of the author. We therefore only cover two examples here, and refer to [36] for more detail.

4.5.1 The Learning Step as a Lens

Without further ado, let us define the general recipe for learning. We will first define the general form of *update* and *displacement* maps for a model $f : P \times A \to B$, before defining the composite learner itself in 4.11.

Definition 4.8 (Update Map [36]). Let $S : \mathscr{C} \to \mathscr{C}$ be a strict monoidal functor. An **update map** for parameters P is a morphism of BiLens of the following type

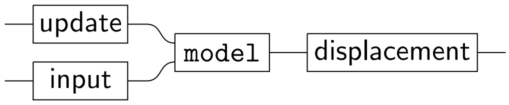
$$\mathsf{update} : \begin{pmatrix} S(P) \times P \\ S(P)' \times P' \end{pmatrix} \to \begin{pmatrix} P \\ P' \end{pmatrix}$$

Remark 4.9. The role of the update map will be the same as in Reverse Derivative Ascent: to update a 'current best guess' parameter θ_i to θ_{i+1} using a *change* in parameters θ_i'. Note however the additional $S(P)$ data: this will allow us to express stateful gradient descent algorithms–in particular, we will use *momentum* gradient descent.

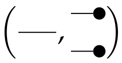
Definition 4.10 (Displacement Map [36]). A **displacement map** for data A is a morphism of BiLens of the following type

$$\mathsf{displacement} : \begin{pmatrix} A \\ A' \end{pmatrix} \to \begin{pmatrix} A \\ A \end{pmatrix}$$

Definition 4.11 (Learner [36]). Fix a reverse differential category $\mathscr{C}$, and choose a model morphism $f : P \times A \to B$, update, and displacement maps. A *learner* is the following composite in BiLens.



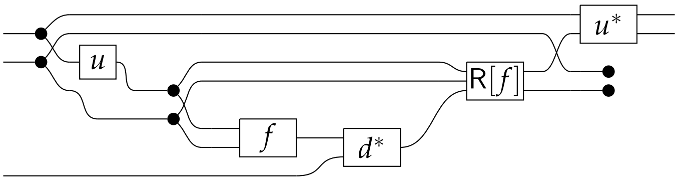
where $\mathsf{input} : \begin{pmatrix} A \\ I \end{pmatrix} \to \begin{pmatrix} A \\ A \end{pmatrix}$ is the lens

input := 

and `model` is the lens $(f, \mathsf{R}[f])$.

Note that because the `model` lens is in the image of the reverse derivative functor, the update and displacement maps must be *simple* lenses. Concretely, these are lenses whose domain and codomain are pairs of the *same* object. In other words, we must have $X' = X$ in all cases: the type of 'changes in X' must be the same as X itself. Nevertheless, we retain this notation to distinguish the differing roles of these two types.

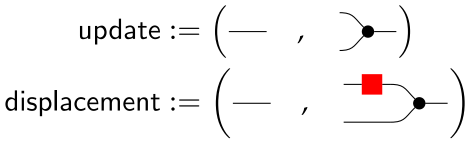
Remark 4.12. The important part of Definition 4.11 is in fact the *reverse* map of the composite lens. The reverse map corresponds to a single 'step' of learning in the same way as the `rdaStep` morphism of Definition 4.6. More concretely, for a model $(f, \mathsf{R}[f])$, update (u, u^*) and displacement (d, d^*), the backwards map of a learner is given by the following diagram:



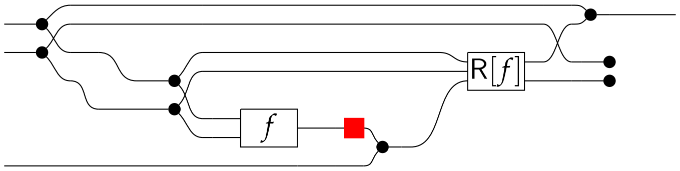
The similarity to the definition of `rdaStep` should now be clear; we will make it precise in Example 4.3.

Example 4.3 (Reverse Derivative Ascent). *Let S be a commutative ring, and suppose $f : P \times A \to B$ is a morphism in* $\mathbf{PolyCirc}_S$*. We may view* `rdaStep` *as the learner with*

$S(P) = I$ and update and displacement maps as follows.



When composed to yield the learner morphism, we obtain the following reverse map:



which is clearly equal to `rdaStep`*.*

Let us now consider examples for the case of neural networks. In order to draw parallels with neural networks terminology, we will give examples of displacement and update maps separately. The reason for this is that while update maps are analogous to learning schemes such as gradient descent and its variants, displacement maps are more like *cost functions*.

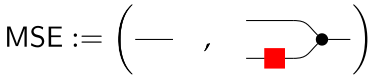
In the lens-based perspective, a displacement map can be thought of as a function computing a difference between model prediction and the true value. The neural networks perspective is somewhat different. When learning the parameters of a neural network with stochastic gradient descent, a single step of learning is usually expressed as follows [90]

$$\theta_{i+1} = \theta_i - \eta \cdot \nabla_\theta \mathsf{cost}(\theta_i, x_i, y_i)$$

where cost : $\mathbb{R}^a \to \mathbb{R}$ is a *cost function*. Now, cost is a composite function involving the forward map of the model, and so computing the gradient with respect to cost computes the reverse derivative of the model in essentially the same way as for lenses. Although these two approaches amount to the same in the learning process, the lens-based perspective makes it clear how the components of learning can be composed in a *modular* fashion.

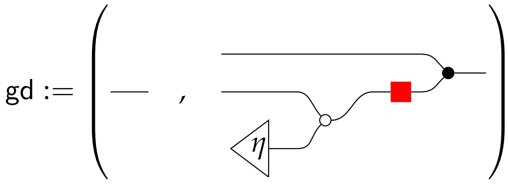
We will now see some concrete examples of displacement and update functions. Our first example is the *mean squared error* displacement map, which arises as the reverse derivative of the mean-squared error cost function applied to a unit change.

Example 4.4 (Mean Squared Error Displacement [36])**.** *The* mean-squared error *displacement denoted* MSE *is defined as the following lens.*



We now give two examples of *update* maps for variants of stochastic gradient descent. We will formulate these as morphisms of $\mathbf{PolyCirc}_S$ for an arbitrary ring S, but to correspond to neural networks we should think of $S = \mathbb{R}$. The first of these is 'basic' stochastic gradient descent without modifications as described in [90, Section 2.2].

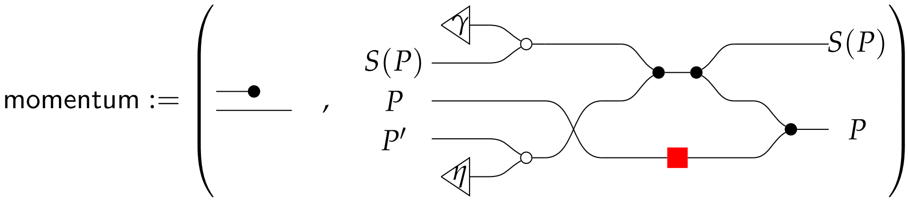
Example 4.5 (Stochastic Gradient Descent [36])**.** *Fix a* ***learning rate*** $\eta : I \to S$*, and choose* $S(P) = I$*. The* ***gradient descent*** *update map is given by the following lens.*



When $S = \mathbb{R}$, one typically selects a small value of $\eta < 1$.

Our next example is a more complex version of stochastic gradient descent using a stateful 'momentum' update. Details of momentum gradient descent can be found in [90, Section 4.1].

Example 4.6 (Stochastic Gradient Descent with Momentum [36])**.** *Fix a small learning rate* $\eta : I \to S$ *and* ***momentum term*** $\gamma : I \to S$*. Momentum gradient update is given by the following lens.*



Remark 4.13. In all the examples above, we've assumed the underlying base category is $\mathbf{PolyCirc}_S$ where S is a *ring*. However, we would like to be able to define learning over arbitrary *semirings*, such as the semiring of 'saturating arithmetic' (Definition 3.41.) For such semirings, consider the setting $\mathbf{PolyCirc}^{=}_S$ of 'functionally complete' polynomial circuits. Here, it is possible to define a difference operation which nevertheless does not respect the usual arithmetic laws (and therefore does not define a ring). Thus, the formulation of learners given here does not preclude the use of semirings: it is merely necessary to find an appropriate displacement map for a given semiring. One such option for computing differences in saturating arithmetic might be the 'clamped difference' operation, i.e., $\langle x, y \rangle \mapsto \mathsf{max}(x - y, 0)$. However, empirical testing of this hypothesis is not addressed in this thesis; we leave this remark as conjecture only.

4.6 Case Studies

We now present two case studies of applying our method to data. These are intended to serve as empirical checks that our theory defines algorithms capable of learning. To that end, we will verify experimentally that the parameters of models expressed as morphisms of **PolyCirc** can be learned by the methods outlined in Sections 4.4 and 4.5.

The first case study in Section 4.6.2 is intended to demonstrate that our theoretical framework correctly captures existing approaches to neural network learning. To verify this, we will express the same model and training scheme in both our framework and Keras [27], an existing deep learning framework. In doing so, we can make a fair comparison: our models should produce comparable results on the same data.

The second case study in Section 4.6.3 demonstrates that our method is able to generalise gradient-based learning beyond the setting of 'real-valued' circuits. In particular, we design models in $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ and show that their parameters can be learned using the Reverse Derivative Ascent algorithm. Note that in this case study there is no use of $\mathbb{R}$ (i.e., floating-point) values at all: computation during training and prediction phases is completely in terms of bit-wise operations such as AND and XOR.

Both our implementations and experimental code can be found in Appendix D.1. We give more detailed information accompanying each of the following case studies.

Before proceeding to experimental details and results, we begin by giving some details of the experimental datasets used.

4.6.1 Datasets

The case studies considered here both give classification models for two datasets: the Iris dataset [44] and the MNIST image classification benchmark [78]. Before describing the details of our experiments, we will give some brief background on these datasets.

The Iris dataset [42] consists of 150 observations from three types of Iris flower. Each observation consists of 4 measurements of petal and sepal length and width. For our purposes, it will serve as a classification problem, where the goal is to predict the

TABLE 4.1: Empirical Results for Neural Network Models

Dataset	Framework	Model	Training	Accuracy %
Iris	Ours	`simple`	Gradient descent	96.00%
Iris	Keras	`simple`	Gradient descent	96.77%
Iris	Ours	`hidden`	Gradient descent	98.00%
Iris	Keras	`hidden`	Gradient descent	98.00%
MNIST	Ours	`convolutional`	Momentum gradient	97.26%
MNIST	Keras	`convolutional`	Momentum gradient	96.38%

flower species from the four measurements. More abstractly, the goal is to find a map $f : \mathbb{R}^4 \to \bar{\mathbf{3}}$, where $\bar{\mathbf{3}}$ denotes the 3-element set of labels. We provide models for this dataset as a test case: even simple models are able to achieve high accuracy, and so we include it here as a basic test that learning is functioning correctly.

The MNIST image classification benchmark is the main focus of our experiments here. The data consists of approximately 60,000 28x28-pixel images representing hand-drawn numeric digits in the range 0 to 9. The MNIST dataset is a more realistic test of solving a 'real-world' problem, requiring a more complex model. However, our goal here is by no means to create a state-of-the-art model; we merely wish to prove our method experimentally.

We can now proceed to describe the first case study.

4.6.2 Neural Networks

We now compare our method to an existing deep learning framework for training neural networks. The objective of these experiments is to provide empirical evidence that the learners described in Section 4.5 faithfully represent gradient descent learning for neural networks. To that end, we compare three models on two datasets implemented in both our framework and that of Keras [27].

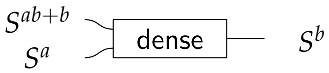
The results of each experiment are given in Table 4.1. These show the models trained in our framework achieve accuracy within 1% of those trained with Keras. We therefore conclude that our approach is indeed able to capture the process of gradient-based learning for neural networks. We give details on experimental setup for the Iris and MNIST datasets in Sections 4.6.2.1 and 4.6.2.2, respectively. See also Implementation D.3 for the implementation of these experiments.

4.6.2.1 Simple and Hidden Models and the Iris Dataset

Our first two experiments use the Iris [42] dataset. We compare two models in our framework to the same models expressed in Keras [27]. In each case, we learn a map of the form $\mathbb{R}^4 \to \mathbb{R}^3$, whose output is one-hot encoded class labels. We define these models now.

Definition 4.14 (`simple` model)**.** `simple` is a single 'dense layer' with sigmoid activation

`simple` :=

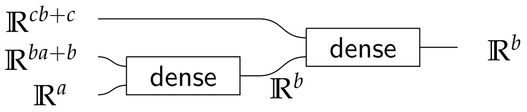


where $a = 4$ and $b = 3$ correspond to the input and input dimension, respectively.

This definition represents one of the simplest possible neural network models: it multiplies the input by a $b \times a$ matrix, and adds the 'bias' term, before scaling outputs into the range $[0, 1]$ with the activation layer to represent probabilities. Nevertheless, this is sufficient to achieve high performance on the Iris dataset.

A modest increase in accuracy can be achieved by composing two such layers.

Definition 4.15 (`hidden` model)**.** Define the model `hidden` as a composition of two dense layers, depicted as follows



so that its parameters are $\mathbb{R}^{cb+c} \times \mathbb{R}^{ba+b}$ with $a = 4$ and $c = 3$ corresponding to the input and output dimensions respectively.

Both models are trained using the mean-squared error displacement and basic gradient update maps described in Section 4.5.

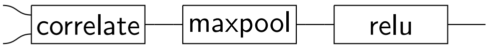
4.6.2.2 Convolutional model and the MNIST dataset

For the MNIST problem, we use a relatively more complex model. Following [27], we use a convolutional network consisting of two convolution layers with max pooling and ReLU activations, followed by a single dense output layer. Note that the `convolutional` model architecture we use here is a simple modification of an example given in the Keras documentation [27] which is already known to work well on the MNIST dataset.

Note also that in this case data is preprocessed by scaling pixel values to the range $[0, 1]$. Since pixel values are encoded as bytes, this is done by simply dividing pixel values by 255.

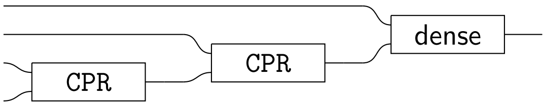
Definition 4.16 (`convolutional` (modified from [27])). The `convolutional` model is defined as the composition of two max-pooling convolutional layers followed by a `dense` 'output' layer. First we define

`CPR` :=



from which we can define the complete model:

`convolutional` :=



The training scheme in this case is *momentum* gradient descent with mean squared error. More specifically, we use the momentum update map of Example 4.6, and the mean-squared-error displacement map of Example 4.4. Complete details are available in Implementation D.3.

4.6.3 Reverse Derivative Ascent

Our second case study has a different goal: to demonstrate empirically that the Reverse Derivative Ascent procedure is able to learn the parameters of models expressed as morphisms in $\mathbf{PolyCirc}_{\mathbb{Z}_2}$. We define two models which we call `eval` and `pseudoLinear`, applied respectively to the Iris and MNIST datasets using the Reverse Derivative Ascent learning scheme described in Definition 4.6.

In contrast to the neural networks case studies, we also preprocess both datasets by transforming numeric inputs into a *single bit* by thresholding. We give a complete description of this preprocessing in the description of each experiment, but it is important to note that this transformed dataset therefore contains a great deal less information than the original. Consequently, the maximum possible accuracy obtainable on each problem is much lower than in the non-quantised case. Moreover, in some cases we also reduce the problem to that of a *binary* classification. It is therefore not appropriate to consider these tasks as equivalent to those described in Section 4.6.2. Our goal here is instead merely to prove experimentally that the method of RDA works in principle: we consider the task of designing models for the full datasets out of scope for this thesis.

We give results of experiments on the Iris and MNIST datasets in Table 4.2. For the 2-class datasets, each class has roughly the same number of examples, so a random classifier would have approximately 50% accuracy. We can therefore see that RDA is able to learn classifiers that perform significantly better than random on the 2-class experiments, even on the relatively complex MNIST dataset. Note however the seemingly low accuracy of the Iris model applied to the full 3-class classification

TABLE 4.2: Empirical Results for Reverse Derivative Ascent

Dataset	Model	Label Encoding	Accuracy %
Iris (2-class)	`eval`	binary	98.0%
Iris (2-class)	`eval`	one-hot	98.0%
Iris	`eval`	binary	73.3%
Iris	`eval`	one-hot	73.3%
MNIST (2 class)	`pseudoLinear`	binary	99.2%

problem. Because of the input transformation, there are many cases of input examples (x_1, y_1) and (x_2, y_2) where $x_1 = x_2$ but $y_1 \neq y_2$. It is therefore impossible to give a perfect classification. In fact, the best possible classifier on this problem can achieve at best 81.3% test accuracy.

Remark 4.17. We also include variations 'binary' and 'one-hot' for each Iris experiment. The two variants refer to to the encoding of the labels for each task. In the 'binary' scheme, the label is simply the binary representation of each numeric class, so 0 becomes 00, 1 becomes 01 and 2 becomes 10. In the one-hot scheme, a label of n classes is encoded as a length n bitvector with the i^{th} bit set to 1.

We now give details on the experimental setup and models used.

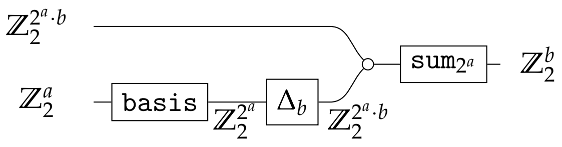
4.6.3.1 The `eval` model and the Iris Dataset

Preprocessing in the Iris model consists of normalisation and thresholding. More precisely, features are normalised to the range $[0, 1]$ then quantised, so that each normalized input feature becomes 1 if above 0.5, and 0 otherwise.

Results for the Iris dataset are given for the `eval` model trained using the Reverse Derivative Ascent algorithm (Definition 4.6). We now describe the `eval` model in more detail.

The `eval` model is an implementation of the function described in Equation (3.8). The idea is that the $2^a \cdot b$ parameters together represent the extensional representation of a function. That is, for a given input a, its b-bit output will simply be looked up in a table of 2^a entries.

Definition 4.18. The `eval` model for a-bit input and b-bit output is given by the following diagram



where

`basis` : $\mathbb{Z}_2^a \to \mathbb{Z}_2^{2^a}$ maps an a-bit value interpreted as an integer into its 2^a bit basis vector, so for example if $a = 3$, the input $(1, 0, 1)$ represents the integer 5, and is mapped to the length $2^a = 8$ output $(0, 0, 0, 1, 0, 0, 0, 0)$.

$\Delta_n : \mathbb{Z}_2^a \to \mathbb{Z}_2^{na}$ is the n-fold copy morphism formed by composition and tensor product of –●< and identity morphisms. Note that by associativity, any choice of these is equal up to the axioms of polynomial circuits.

$\mathtt{sum}_n : \mathbb{Z}_2^{na} \to \mathbb{Z}_2^a$ similarly denotes the n-fold *addition* morphism, formed by composition and tensor of >●– and identity maps.

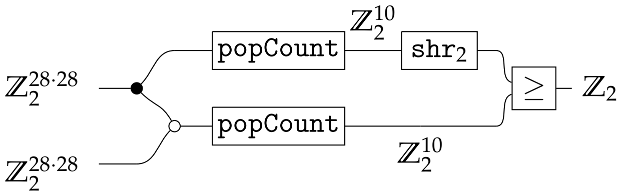
Complete details of the `eval` model can be found in Appendix D.1. See in particular Implementations D.1 and D.2.

4.6.3.2 The `pseudoLinear` model and the MNIST dataset

MNIST data is preprocessed similarly to the Iris dataset. Pixels are set to 1 if they are larger than the mean pixel value, and 0 otherwise. Additionally, we consider only a subset of the classification problem. Specifically, we use only the data of images of digits 0 and 1, with labels in the set $\{0, 1\}$. The trained model will therefore have the type $\mathbb{Z}_2^n \times \mathbb{Z}^{28 \cdot 28} \to \mathbb{Z}_2$ where $n \in \mathbb{N}$ is the number of parameters.

Training is again via the Reverse Derivative Ascent algorithm of Definition 4.6. Results on the MNIST dataset are given for the `pseudoLinear` model, which is defined as follows.

Definition 4.19. The `pseudoLinear` model is given by the following diagram



where:

`popCount` : $\mathbb{Z}_2^a \to \mathbb{Z}_2^{\lceil \log_2(a) \rceil}$ computes an integer representation of the number of bits set in the input

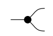
$\mathtt{shr}_n : \mathbb{Z}_2^a \to \mathbb{Z}_2^a$ denotes a logical right shift by n

$\geq : \mathbb{Z}_2^{a+a} \to \mathbb{Z}_2$ denotes comparison of two a-bit integer representations, returning 1 if the first is larger than the second, and 0 otherwise.

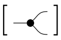
The intuition for the `pseudoLinear` model is that it learns a *mask* of the same size as the input. This mask should represent the 'average' example of the 0 class so that when multiplied by an input of the same class, a large number of bits are set in the output.

4.7 Implementation as Lenses of Functions

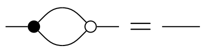
In the previous section, we defined models as morphisms of $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ and demonstrated empirical results using the RDA algorithm on datasets. In the implementation of our experiments, it is of course necessary to actually *evaluate* these morphisms. Consequently, our implementation is actually in terms of lenses of *functions*, rather than lenses of $\mathbf{PolyCirc}_{\mathbb{Z}_2}$. This choice of implementation introduces some subtleties which we address now.

Recall that $\mathsf{Fn} : \mathbf{PolyCirc}_{\mathbb{Z}_2} \to \mathbf{FinSet}_{\mathbb{Z}_2}$ is the functor interpreting a polynomial circuit as tuples of functions. The primitives of our implementation are lenses of the form $(\mathsf{Fn}(f), \mathsf{Fn}(\mathsf{R}[f]))$ for a morphism $f \in \mathbf{PolyCirc}_{\mathbb{Z}_2}$. For example, the primitive lens corresponding to the morphism  is actually the lens given by the pair of maps

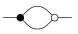
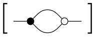
$$x \mapsto (x, x) \qquad\qquad (x, \delta_{y_1}, \delta_{y_2}) \mapsto \delta_{y_1} + \delta_{y_2}$$

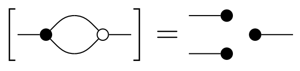
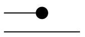
where the backward map is the interpretation of R [] = as a function.

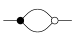
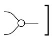
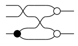
Models are constructed by composition and tensor product of these primitives. However, while this is sufficient to validate our approach experimentally, it hides some important subtleties which we now clarify. The corresponding category of functions for $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ is $\mathbf{FinSet}_{\mathbb{Z}_2}$, which we defined in Section 3.6. In fact, $\mathbf{FinSet}_{\mathbb{Z}_2}$ has a presentation by generators and equations due to Lafont [76]. This presentation is exactly that of $\mathbf{PolyCirc}_{\mathbb{Z}_2}$, augmented with the additional equation



which we can interpret as the equality of polynomials $x^2 = x$.

Interpreting a morphism in $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ as a function therefore amounts to quotienting by this new equation. However, notice that in so quotienting, the category $\mathbf{FinSet}_{\mathbb{Z}_2}$ is no longer an RDC. This is because R is not well-defined with respect to the new equation. More precisely, since we have  = — we must also have that R [] = R [—]. However, we can calculate that

R  R [—] = 

and so clearly R [] ≠ R [—] Thus, we cannot say that $\mathbf{FinSet}_{\mathbb{Z}_2}$ is a reverse derivative category, at least with R [] = .

We must therefore be clear about the precise sense in which our implementation uses reverse derivatives. Since primitives all have the form $(\mathsf{Fn}(f), \mathsf{Fn}(\mathsf{R}[f]))$, we should also require that their compositions and tensor products do as well. This guarantees that the reverse map of each model is genuinely a reverse derivative. More formally, this requirement can be stated as in the following proposition.

Proposition 4.20. *Let $\mathscr{C}$ be a reverse derivative category. Given a strict cartesian monoidal functor $\mathsf{Fn}(\cdot) : \mathscr{C} \to \mathscr{D}$, we have*

$$(\mathsf{Fn}(f), \mathsf{Fn}(\mathsf{R}[f])) \mathbin{;} (\mathsf{Fn}(g), \mathsf{Fn}(\mathsf{R}[g])) = (\mathsf{Fn}(f \mathbin{;} g), \mathsf{Fn}(\mathsf{R}[f \mathbin{;} g]))$$

and

$$(\mathsf{Fn}(f), \mathsf{Fn}(\mathsf{R}[f])) \otimes (\mathsf{Fn}(g), \mathsf{Fn}(\mathsf{R}[g])) = (\mathsf{Fn}(f \times g), \mathsf{Fn}(\mathsf{R}[f \times g]))$$

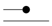
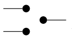
Proof. The two equalities hold straightforwardly because Fn is strict monoidal, and because composition and tensor product of lenses is the same for lenses as for reverse derivatives (Definition 3.13). □

More generally, observe that a strict monoidal functor $F : \mathscr{C} \to \mathscr{D}$ can be lifted into a functor of lenses by its application 'pointwise' to each map in a lens:

$$\begin{aligned} &\mathsf{BiLens}(F) : \mathsf{BiLens}(\mathscr{C}) \to \mathsf{BiLens}(\mathscr{D}) \\ &\mathsf{BiLens}(F)(f, f^*) := (F(f), F(f^*)) \end{aligned}$$

Consequently, a 'composition of interpretations' of polynomial circuits is the same as the 'interpretation of compositions'.

Now we can be confident that a model constructed from primitive lenses of the form $(f, \mathsf{R}[f])$ always has a reverse map consistent with its forward map. However, an immediate caveat is that for a given function $f \in \mathbf{FinSet}_{\mathbb{Z}_2}$, there are actually *many* valid reverse maps. This is because there are lenses (f, f^*) for which there are two distinct polynomials f_1, f_2 with $\mathsf{Fn}(f_1) = \mathsf{Fn}(f_2)$ but $\mathsf{Fn}[\mathsf{R}[f_1]] \neq \mathsf{Fn}[\mathsf{R}[f_2]]$.

Thus, we might ask: for a given function, which of the reverse maps is most appropriate? Informally, we should generally wish to preserve information flow in the reverse direction. For example, given the function corresponding to the identity map , it makes more sense to use  as the reverse derivative than , so that 'information flow' in the reverse direction is not lost.

This gives one motivation for Part II of the thesis. In order to understand the structure of the backwards maps of a given model, one must have access to their underlying graphical structure. However, an implementation based on functions is 'opaque': the internal structure is completely hidden. It would therefore be preferable to have a completely symbolic description of the model.

Moreover, symbolic descriptions are amendable to syntactic transformation. Such transformations have several applications. For example, as in [101], one goal is to evaluate such models on dedicated hardware such as FPGAs for embedded applications. Another example of where symbolic descriptions are useful is in *meta-learning*, where the structure of the model itself is learned.

In Part II of the thesis, we will give datastructures and algorithms for representing morphisms of categories like $\mathbf{PolyCirc}_S$. One aim of this work is to support morphisms of 'industrial scale' consisting of large numbers of generators. In doing so, we hope to provide a more transparent way to represent large circuit models such as those described in this chapter.

Part II

Datastructures for Circuits

Chapter 5

Datastructures and Algorithms

5.1 Overview

We now turn our attention to computer representations of morphisms of symmetric monoidal categories. More specifically, in this chapter we will define a datastructure for representing morphisms of **PROP**s and give algorithms for composition and tensor product. Our guiding motivation is to represent morphisms of $\mathbf{PolyCirc}_S$, the family of categories defined in Chapter 3 whose morphisms correspond to machine learning models. However, we have additional desiderata for the datastructure we present here, which we now motivate.

As noted by Bonchi et al. [18], string-diagrammatic syntax is a natural way to model phenomena in computing which can be thought of as mapping multiple inputs to multiple outputs. We have already seen one example in the machine learning models of Chapters 3 and 4. Other examples exist as well; aside from the obvious case of visual programming languages (as used in e.g. [52, 99]), data such as dependencies and workflows (as in e.g. [51]) can also be naturally thought about in a graphical way. Moreover, Bonchi et al. [18] also note that even the syntax of programming languages–where terms are typically expressed as tree structures–has also enjoyed a generalisation to term *graphs* [87].

Despite this, many phenomena are still modeled using tree-like structures, which can be thought of as representing 'many-to-one' operations. In category-theoretic terms, the use of trees is analogous to working directly with Σ-terms, which obscures the graphical structure of these phenomena. In some cases combinatorial structure is modeled with graphs, but graphs alone do not capture the generality of string diagrams. For example, the software [51] encodes dependency relations as directed acyclic graphs, where nodes represent 'tasks' and edges the inputs and outputs of each task. However, in contrast to a string diagram, such an encoding does not specify the ordering of interfaces to each task: this is analogous to a string diagram in which every generator is commutative. In order to solve this problem, the software places additional burden on the user. However, in a string-diagrammatic approach, the ordered interfaces mean that such data could be represented unambiguously.

We suggest that one reason for the use of trees and graphs instead of string diagrams in general-purpose programming is the lack of clearly specified and simple-to-implement datastructures. Our secondary aim is therefore to make a datastructure suitable for general use when modeling phenomena of the 'many-to-many' type mentioned above. Of course, this should not detract from our main aim: this datastructure should still be a completely formal object. More precisely, we will require that it forms a category, and prove that the operations on it correspond to their categorical counterparts. In any case, this additional secondary aim introduces some new desiderata which we discuss now.

Our first requirement is that the datastructures and algorithms described here should be easy to implement. By way of comparison, tree and graph datastructures are well-understood, with standard references available for many different representations and algorithms. In order for string diagrams to be considered as a replacement, their implementation should be as straightforward as possible. Although we provide a software library (Implementation D.4) for the datastructure and algorithms we define here, our goal here is to ensure that the algorithms themselves are easy to implement without use of this library. This is important because it is not always possible to use a given implementation. For example, if one is constrained by choice of programming language, then it is not necessarily possible to use a library developed for another. It should therefore be possible to implement our datastructure simply and with similar effort as for tree- and graph-like datastructures.

The second requirement is to support 'industrial scale' uses of string diagrams. It is therefore also important to give algorithms which have high performance. Moreover, these algorithms should be able to exploit parallel hardware such as GPUs. In order to achieve this, the datastructures and algorithms we describe are completely in terms of linear-algebraic operations on sparse matrices, which have efficient implementations for both sequential and parallel hardware. This allows the datastructures described in this chapter to support diagrams consisting of millions of generators.

Lastly, we should describe the algorithms themselves in terms of string diagrams. This means that proofs about the datastructure can themselves be diagrammatic. Using the rewriting framework of [18, 19], this means that the datastructure is able to actually represent proofs about the datastructures using the datastructures themselves.

To summarise, we have the following desiderata. We first wish our datastructures and algorithms to have clear, simple definitions which are easy to implement. Secondly, the algorithms should be high performance, both in terms of complexity and real-terms performance of empirical benchmarks. Finally, we should be able to express the algorithms themselves as string diagrams.

5.1.1 Relationship to Published Work

Aside from the preliminaries of Section 5.3, the content of this chapter consists of the author's individual contributions originally published in Wilson and Zanasi [103]. Parts of this chapter are reproduced from this work verbatim.

5.2 Synopsis

We begin with preliminaries in Section 5.3 where we recall the correspondence shown by Bonchi et al. [17] between certain 'open directed hypergraphs' and the free **PROP** on a given signature. We then recall a technique from the parallel programming literature [64] for representing *undirected* hypergraphs as graphs before finally recalling the adjacency matrix representation of a graph.

Our contributions begin in Section 5.4, where in Definition 5.14 we define the **Har** datastructure for representing string diagrams in terms of matrices.

In Section 5.5 we give operations for the construction of **Har**s. We show how to construct 'primitive' **Har**s corresponding to identity, symmetry, and generators in Definitions 5.22, 5.23, and 5.24, respectively. We then define the operations of tensor product (Definition 5.26) and composition (Definition 5.28).

Section 5.6 begins with Theorem 5.29 which demonstrates that **Har**s form a category. Proposition 5.6 shows that every valid **Har** datastructure corresponds to a string diagram, and so one can construct a **Har** by directly constructing its underlying bipartite graph. Finally, in Theorem 5.39 we construct an isomorphism between categories of **Har**s and the free **PROP** on a given signature.

Section 5.7 is devoted to a complexity analysis of **Har** operations. **Har**s over a given signature are shown to enjoy a property of *Bounded Sparsity* (Proposition 5.40), meaning that the matrices used in representing **Har**s are mostly zeros. Using this property, it can then be shown that permutation of **Har**s has linear time complexity (Proposition 5.41), from which it follows that the operations of tensor product and composition are also linear time (Propositions 5.42 and 5.43).

In Section 5.8, we give an empirical analysis of the performance of our **Har** algorithms for tensor and composition. More concretely, we compare our implementation against the wiring diagrams of [82] on some synthetic benchmarks involving composition of large diagrams consisting of millions of generators.

The chapter concludes in Section 5.9 with a discussion of how **Har**s can be extended past the case of **PROP**s to represent morphisms of the free symmetric monoidal category with multiple generating objects.

5.3 Preliminaries

The work in this chapter builds on the correspondence between string diagrams and certain 'open hypergraphs' developed in [17–19]. The essence of our implementation is a representation of these hypergraphs purely in terms of sparse matrices. Moreover, the operations we will define on our representation can be stated purely in terms of linear algebraic operations. For that reason, we will begin by recalling both the open hypergraphs of Bonchi et al. [17] as well as the adjacency matrix representation of graphs.

5.3.1 Open Hypergraphs

The key idea of Bonchi et al. [17] is that string diagrams can be represented as certain hypergraphs with 'interfaces'. More precisely, they draw an isomorphism between such hypergraphs and $\mathbf{Free}_\Sigma$, the **PROP** freely generated by signature Σ. A key benefit of their approach is that the hypergraph representation is 'modulo the laws of SMCs'. The result is that string diagrams are considered purely combinatorially– in terms of their *connectivity*, instead of the geometry of a particular diagram layout as in [70].

In Section 5.4, we will show how such hypergraphs can also be represented as directed bipartite graphs. We therefore begin by recalling the basic definition of directed graph, will shortly be contrasted with the definition of directed *hyper*graphs.

Definition 5.1 (Directed Graph [6])**.** A **directed graph** is a set of vertices V and edges $E \subseteq V \times V$ pairs of vertices. A graph is said to be **bipartite** if V is the disjoint union of two sets $V_0 + V_1$ and each edge (s, t) has either $s \in V_0$ and $t \in V_1$ or $s \in V_1$ and $t \in V_0$.

The edges of a directed graph always connect exactly two nodes. In contrast, a hypergraph has edges which connect *lists* of vertices.

Definition 5.2 (Directed Hypergraph [17])**.** A **directed hypergraph** consists of:

- A set of **hypernodes** V
- A set of **hyperedges** $E \subseteq \mathsf{List}(V) \times \mathsf{List}(V)$

where each edge $e \in E$ is an ordered pair of *lists* of nodes $e = ((s_1, s_2, \ldots, s_m), (t_1, \ldots, t_n))$ where $s_1 \ldots s_m$ are the **source hypernodes** and $t_1 \ldots t_n$ are the **target hypernodes**. We say the **arity** and **coarity** of the edge e are m and n, respectively. If v is a hypernode, we say its **in-degree** $\mathsf{in}(v)$ (resp. **out-degree** $\mathsf{out}(v)$) is the number of edges for which it appears as a target (resp. source). Such hypergraphs form a category [17] denoted **Hyp**.

Under this definition, each of the vertices s_i for $i \in \bar{\mathbf{m}}$ in an edge e is adjacent to every target t_j for $j \in \bar{\mathbf{n}}$, and so directed hypergraphs connect m sources to n targets. Notice also that in the above we used 'hypernode' and 'hyperedge' to distinguish the nodes and edges of hypergraphs from the directed graphs of Definition 5.1.

Remark 5.3. There are a number of ways to generalise graphs to hypergraphs. We say 'directed' hypergraph to distinguish from the undirected hypergraphs of [64] which we define shortly. However, the 'directed' qualifier will usually be omitted, so the reader may assume 'hypergraph' refers to the directed hypergraphs of Definition 5.2.

In order for the hypergraphs of Definition 5.2 to represent string diagrams, two additional pieces are required. The first of these is a labeling of hyperedges with generating morphisms Σ_1.

Definition 5.4 (Σ-Labeled Hypergraph [17])**.** Let Σ be a monoidal signature with a single generating object so that $\Sigma_0 = \{1\}$. A **Σ-labeled hypergraph** is a triple (V, E, L) where (V, E) is a hypergraph and $L : E \to \Sigma_1$ is a labeling of edges as generating morphisms such that the arity and coarity of each e is the same as that of $L(e)$. The category of Σ-labeled hypergraphs is denoted $\mathbf{Hyp}_\Sigma$.

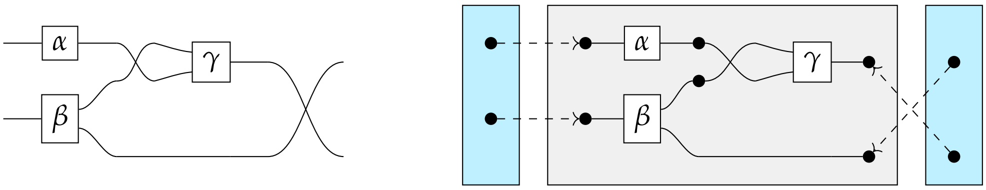
Remark 5.5. Bonchi et al. [17] define the hypergraphs of Definition 5.2 as a category of functors into **FinOrd**, the category of finite sets and functions. The category $\mathbf{Hyp}_\Sigma$ is then defined as the slice category $\mathbf{Hyp}/\Sigma$. In both cases, the morphisms of the category are the 'structure preserving' maps: natural transformations requiring that (for example) order of sources and targets is respected.

There is one final piece to add: the *interfaces* of the hypergraph. For this, Bonchi et al. [17] use *cospans* of hypergraphs. String diagrams are then represented using the following definition.

Definition 5.6 (Open Hypergraphs)**.** An **Open Σ-labeled hypergraph** is a cospan $m \xrightarrow{s} H \xleftarrow{t} n$ where m, n are discrete hypergraphs (having no hyperedges), and H is a Σ-labeled hypergraph. The category of such cospans is denoted $\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_I$, with composition given by pushout.[1] We will often say simply **open hypergraph** when assuming an arbitrary monoidal signature.

[1] Note also this is a (full) subcategory of cospans of hypergraphs, since the feet of the cospan are required to be discrete [17, Definition 3.2].

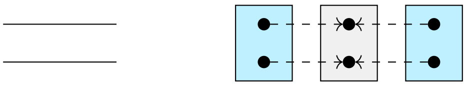
In order to demonstrate how open hypergraphs represent string diagrams, let us proceed with the example below. On the left is a string diagram, and on the right its interpretation as an open hypergraph.



(5.1)

In the depiction above right, the hypergraph H is shown in the central grey box, corresponding to the 'internal wiring' of the diagram. The discrete hypergraphs m, n depicted in blue boxes represent the interfaces of the diagram, while the legs s, t of the cospan represent the left and right 'dangling wires'.

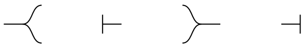
Identities and symmetries are represented as hypergraphs having no hyperedges. For example, the identity morphism of type 2 → 2 is depicted in Equation (5.2) below as a string diagram (left) and hypergraph (right).



(5.2)

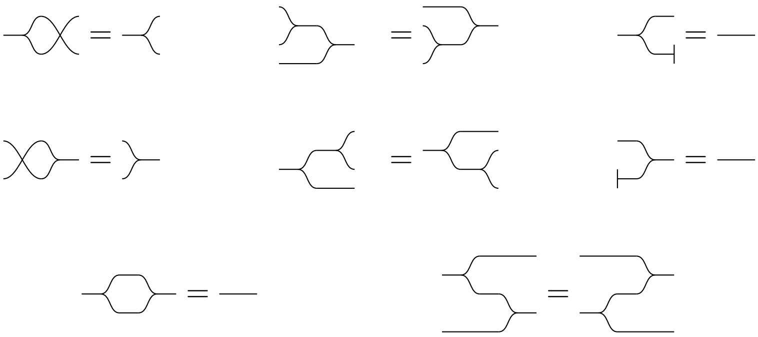
However, not every such hypergraph corresponds to a valid string diagram of $\mathbf{Free}_\Sigma$. In other words, the interpretation of string diagrams as hypergraphs is faithful, but not full. In fact, Bonchi et al. [17] show that there is a correspondence between such hypergraphs and string diagrams generated by Σ augmented by *special frobenius structure.* Graphically, this additional structure is given by the following definition (see also [17, Figure 2]).

Definition 5.7 (Special Frobenius Structure [17]). The theory of Special Frobenius monoids is denoted **Frob**, and consists of generators



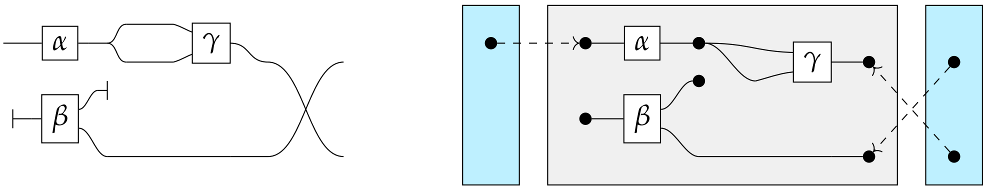
(5.3)

and equations



(5.4)

To clarify the relationship between hypergraphs and string diagrams augmented with this additional structure, observe the following hypergraph which does not correspond to a string diagram of $\mathbf{Free}_\Sigma$, but instead to one of $\mathbf{Free}_{\Sigma+\mathbf{Frob}}$.



(5.5)

Remark 5.8. Intuitively, the correspondence between hypergraphs and frobenius structure can be thought of as follows. The ⊢ and ⊣ morphisms correspond to hypernodes not appearing as a source or target. Similarly, the −⟨ and ⟩− morphisms allow for hypernodes appearing in *multiple* sources or targets.

Naturally however, we would also like to model string diagrams (thought of as morphisms of $\mathbf{Free}_\Sigma$) without the additional frobenius structure. For this reason, Bonchi et al. [17] introduce two conditions on such hypergraphs. We recall the first of these now.

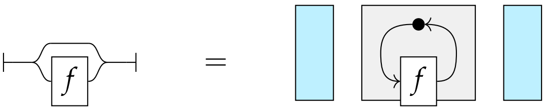
Definition 5.9 (Monogamicity [17]). An open hypergraph $m \xrightarrow{s} (V, E) \xleftarrow{t} n$ is **monogamous** when s, t are monomorphisms, and for each hypernode $v \in V$, we have $|s^{-1}(v)| + \mathsf{in}(v) = 1$ and $|t^{-1}(v)| + \mathsf{out}(v) = 1$ where f^{-1} is the inverse image of the function f.

The *monogamicity* condition ensures that any given hypernode appears exactly once as a source or target, including in the interfaces. In other words, this condition ensures the string diagram can be pictured without use of any of the Frobenius generators in Equation (5.3).

The second condition is that of *acyclicity*, which prevents hypergraphs with 'feedback'.

Definition 5.10 (Acyclicity [17]). A **cycle** in a hypergraph is as defined for graphs. That is, a cycle is a path whose start and end hypernode are equal. A hypergraph is **acyclic** when it has no cycles.

Notice that the condition of acyclicity is required since one can create a hypergraph with cycles by simply composing morphisms of $\mathbf{Free}_{\Sigma+\mathbf{Frob}}$. For example, below left is a string diagram in $\mathbf{Free}_{\Sigma+\mathbf{Frob}}$, and below right is its interpretation as a hypergraph.



(5.6)

Thus the example hypergraph pictured in (5.1) is monogamous acyclic, but the hypergraphs of (5.6) and (5.5) are not.

Now armed with the definition of open hypergraphs, acyclicity, and monogamicity, we can finally state the definition of the category of hypergraphs which will correspond exactly to string diagrams–i.e., morphisms of $\mathbf{Free}_\Sigma$.

Definition 5.11 (Monogamous Acyclic Open Σ-Labeled Hypergraphs [17])**.** Denote by $\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA}$ the subcategory of open Σ-labeled hypergraphs which are monogamous (Definition 5.9) and acyclic (Definition 5.10).

The morphisms of $\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA}$ correspond exactly to string diagrams over Σ, and so there is an isomorphism $\mathbf{Free}_\Sigma \cong \mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA}$ [17].

Now we have described the combinatorial characterisation of string diagrams given in [17], we will review some further background which lays the foundations for our high-performance implementation.

5.3.2 Parallel Hypergraph Processing

Having defined open hypergraphs, we will need a way to implement them which is compatible with our desiderata. There are many such possible representations. For example, in [95], hyperedges are represented directly as pairs of lists. However, this choice of representation comes with drawbacks: code must be written from scratch, and obtaining high performance therefore requires additional tuning.

For our representation here, we would like to leverage existing high performance code for representing graphs and apply it to hypergraphs. We therefore take inspiration from the distributed computing literature. The authors of [64] describe a representation which encodes hypergraphs as certain *bipartite graphs*. The idea here is straightforward. Instead of representing hyperedges and hypernodes as distinct kinds of object, we consider them both as nodes within a bipartite graph.

In the same way that a directed hypergraph can be thought of as generalising a directed graph, an *undirected* hypergraph can be thought of as generalising an (undirected) graph.

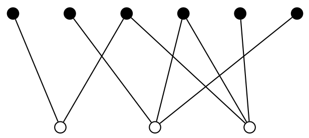
Definition 5.12 (Undirected Hypergraph [64])**.** An **undirected hypergraph** consists of:

- A set of **hypernodes** V
- A set of **hyperedges** $E \subseteq \mathcal{P}(V)$

where $\mathcal{P}(V)$ is the powerset of V.

Observe that in Definition 5.12 there are neither 'source' nor 'target' edges. A hyperedge is instead a *set* of hypernodes, with every hypernode in the set reachable from every other.

The observation of Heintz et al. [64] is that one can achieve high performance in processing undirected hypergraphs by representing them as labeled bipartite graphs. Concretely, vertices are labeled either • or ∘, with the former playing the role of hypernodes, and the latter hyperedges. The bipartite graph pictured below depicts such an encoding, with an edge • → ∘ indicating that the hypernode appears in the hyperedge set.



(5.7)

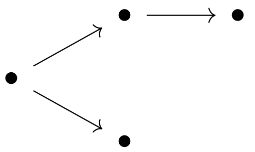
Of course, this representation is missing information: there are neither source nor target hypernodes, and the *order* which hypernodes appear as sources and targets is not present. We will therefore need to adapt this definition to that of *directed* hypergraphs in Section 5.4. In doing so, we will use a representation of graphs as *adjacency matrices*, which we now recall.

5.3.3 Adjacency Matrices and PROPs of Matrices

The representation of directed graphs as adjacency matrices is well known [6]. We recall it here because it will be central to our definition in Section 5.4. We begin by recalling the **PROP** of matrices over a semiring S.

Definition 5.13 ($\mathbf{Mat}_S$)**.** $\mathbf{Mat}_S$ is the category whose objects are the natural numbers and whose morphisms $f : m \to n$ are $n \times m$ matrices with coefficients in S. The $n \times m$ **zero matrix** is denoted $\mathbf{0}_{m,n}$, and the tensor product of matrices $f \otimes g$ is given by the direct sum, i.e. $\begin{vmatrix} f & \mathbf{0} \\ \mathbf{0} & g \end{vmatrix}$. We will refer to the set of $n \times m$ matrices as $\mathbf{Mat}_S(m, n)$, and always write composition of matrices in *diagrammatic* order $f \fatsemi g$ for matrices $f : l \to m$ and $g : m \to n$.

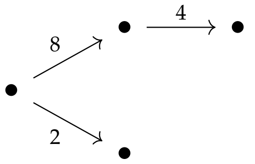
We now recall the adjacency matrix representation of a graph. Let G be a directed graph with K nodes. The adjacency matrix of G is a matrix $\mathbf{Mat}_{\mathbb{B}}(K, K)$ where the i^{th} column denotes the *outgoing* edges of the i^{th} node. For example, consider the graph and its adjacency matrix below:



$$\begin{vmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{vmatrix} \tag{5.8}$$

This representation enjoys a property which will be particularly useful to us later. Namely, there can be exactly one edge of a particular orientation between two nodes of a graph. This will be useful to rule out many non-monogamous hypergraphs.

We can also allow for *labeled* edges by varying the semiring of **Mat**. For example, by considering matrices $\mathbf{Mat}_{\mathbb{N}}(K, K)$ we can consider edges to have labels in the set $\{1, 2, \ldots\}$, with 0 denoting no edge. The choice of the 0 label to represent 'no edge' is important because it will allow us to exploit the *sparsity*. Consider for example the same graph as (5.8) but with labeled edges:

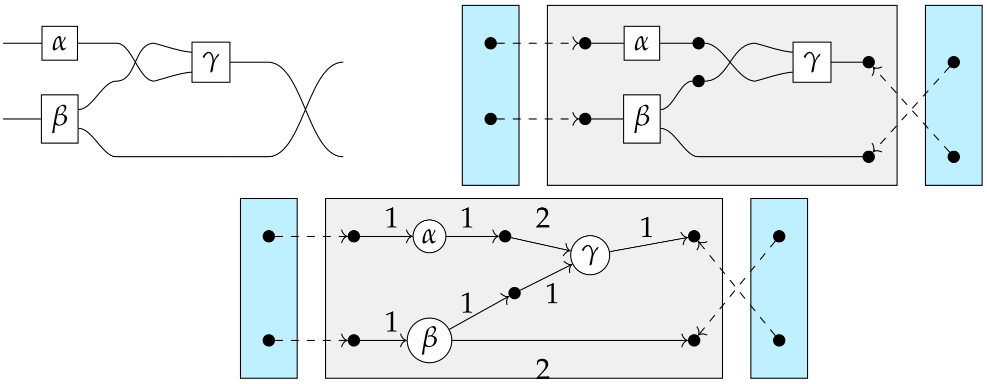


$$\begin{vmatrix} 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \end{vmatrix} \tag{5.9}$$

5.4 The Hypergraph Adjacency Representation

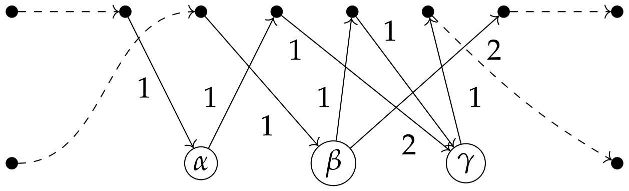
In this section we define our datastructure for representing string diagrams, which we call Hypergraph Adjacency Representations (**Har**s). In order to motivate our definition, we will first develop an example in two steps. We start by adapting the bipartite representation of undirected hypergraphs of Heintz et al. [64] to *directed* hypergraphs, before showing how to encode them in terms of adjacency matrices.

Returning to the example string diagram and hypergraph in Equation (5.1), we now picture its representation as a bipartite graph below the string diagram (left) and hypergraph (right).



(5.10)

We can more easily see that the graph representation (bottom) is bipartite by rearranging nodes and edges as in (5.7).



(5.11)

As with the bipartite representation given by Heintz et al. [64] and depicted in (5.7), we label vertices • and ∘ to denote hypernodes and hyperedges, respectively. We have additionally made a number of changes to account for the differences in our hypergraphs. Namely, our representation has:

- ... *directed edges*, to capture orientation of hyperedges
- ... *interfaces*, which will serve the role of the legs of a cospan in $\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA}$
- ... ∘*-vertex labels* in Σ_1, which serve the role of the hyperedge labeling L from Definition 5.4.
- ... *edge labels* in $\mathbb{N}$, which serve to record the order of hypernodes in hyperedge lists.

With respect to this last point, observe that the two incoming edges to the ∘-vertex labeled γ do not cross each other as in the hypergraph representation. This is meant to highlight that the ordering of these edges is now represented by edge labels, rather than as an explicit list representation attached to the hyperedge.

We will now see how to translate this depiction of a 'bipartite graph with interfaces' into an adjacency matrix representation. Concretely, we will encode the data as 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 first and right boundary nodes last, respectively.

The example hypergraph pictured in (5.10) can then be represented by the following data in which N is shown twice for the sake of 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 also 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 must 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.

5.4.1 Main Definition

We can now define the **Har** datastructure. Note that in the following sections it is assumed that the signature Σ has a single generating object.

Definition 5.14. (Hypergraph Adjacency Representation)
Fix a monoidal signature Σ with a single generating object. A hypergraph adjacency representation of type $m \to n$ is written $\mathbf{Har}_{m,n}$ and consists of the following data:

- **Size** $K \in \mathbb{N}$
- **Labeled Adjacency Matrix** $M \in \mathbf{Mat}_{\mathbb{N}}(K, K)$
- **Left Permutation** $L \in \mathbf{Mat}_{\mathbb{B}}(K, K)$
- **Right Permutation** $R \in \mathbf{Mat}_{\mathbb{B}}(K, K)$
- **Node Labels** $N \in (\{\bullet\} + (\{\circ\} \times \Sigma_1))^K$

Satisfying the following conditions:

1. The graph represented by M is acyclic
2. The matrix $L^T \fatsemi M \fatsemi L$ is ordered such that the **first** m nodes are the **left interface nodes**
3. The matrix $R^T \fatsemi M \fatsemi R$ is ordered such that the **last** n nodes are the **right interface nodes**
4. A node labeled $\bullet$ has no incoming (resp. outgoing) edges if it is a left (resp. right) interface node, and exactly 1 incoming (resp. outgoing) edge otherwise.
5. 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

Remark 5.15. The conditions of **Har**s can be explained as follows. Conditions (1) and (4) ensure that the graph represents a *monogamous acyclic* hypergraph. Condition (5) ensures that the labeling of $\circ$-nodes with generating morphisms Σ_1 is consistent with their arity and coarity. Finally, conditions (2) and (3) ensure that interfaces are consistent with the arity and coarity of the morphism being represented. Notice that the encoding of the interfaces assumes monogamicity. The cospan legs are encoded as permutations, and so a node can appear at most once in an interface. This is in contrast to morphisms of $\mathbf{Free}_{\Sigma+\mathbf{Frob}}$, where the additional frobenius structure allows any $\bullet$-vertex to appear in the interface zero or more times.

5.4.2 Permutation Equivalence and Boundary Orderings

When we define algorithms for composition of **Har**s in Section 5.5, we will discover that composition is only associative up to isomorphism. This is somewhat expected: composition of cospans in $\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA}$ is associative up to unique isomorphism [48], and since our goal is to define a category isomorphic to $\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA}$, we should expect the same. Therefore, in order to form a category of **Har**s, we will quotient by the following equivalence relation.[2]

Definition 5.16 (Permutation Equivalence)**.** $f, g : \mathbf{Har}_{m,n}$ 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 5.17. Note that this definition ensures that if $f \overset{P}{\sim} g$ then the graph represented by g_M is isomorphic f_M and also that the interfaces of f and g are the same.

[2] 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.

Proposition 5.18 (Permutation Equivalence is an Equivalence Relation)**.** *Fix some $f, g \in \mathbf{Har}_{m,n}$. Then there exists an equivalence relation denoted $\sim$, where $f \sim g$ if and only if there exists some permutation matrix $P \in \mathbf{Mat}_{\mathbb{B}}(K, K)$ such that $f \overset{P}{\sim} g$ (c.f. Definition 5.16)*

Proof. We must show that $\sim$ is reflexive, symmetric, and transitive. Reflexivity of $\sim$ is satisfied because $f \overset{\mathsf{id}}{\sim} f$. Symmetry follows because if $f \overset{P}{\sim} g$, then $g \overset{P^T}{\sim} f$. Lastly, transitivity is a consequence of matrix composition: If $f \overset{P}{\sim} g$ and $g \overset{Q}{\sim} h$, then $f \overset{P \mathbin{;} Q}{\sim} h$. □

This definition allows each $f \in \mathbf{Har}_{m,n}$ to be put into equivalent left or right boundary orders by permuting by f_L and f_R, respectively. These particular orderings will come in useful when we define composition and tensor product. Specifically when composing $f \in \mathbf{Har}_{m,n}$ and $g \in \mathbf{Har}_{n,o}$ we will put f into *right* boundary order, and g into *left* boundary order. Taking advantage of monogamicity, this will allow us to compose simply by taking the tensor product of f_M and g_M composed with projections and injections. We discuss this process in detail in Section 5.5. For now, let us define the boundary orderings explicitly.

Definition 5.19. The **left boundary order** of $f \in \mathbf{Har}_{m,n}$ is denoted $L(f)$ and has the following data:

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

Definition 5.20. The **right boundary order** of $f \in \mathbf{Har}_{m,n}$ is denoted $R(f)$ and has the following data:

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

Remark 5.21. Note that by definition $L(f) \overset{f_L}{\sim} f$ and vice-versa, $R(f) \overset{f_R}{\sim} f$.

5.5 Operations on HARs

We can now define algorithms for the operations of composition and tensor product of **Har**s. In addition, we will show how several basic primitive **Har**s can be constructed. Concretely, since **Har**s will form a **PROP**, we will need to define the identity and symmetry **Har**s. It will also be necessary to define the 'singleton' **Har**, which can be thought of as the lifting of a generating morphism in Σ_1 into a **Har**.

We begin by defining the identity **Har**s.

Definition 5.22 (Identity **Har**)**.** The identity $\mathbf{Har}_{m,m}$ is the 4-tuple (M, L, R, N) with

$$M = \mathbf{0}_{m,m} \qquad L = \mathsf{id}_m \qquad R = \mathsf{id}_m \qquad N = \mathbf{0}_{m,1}$$

Note that there are no nodes labeled $\circ$ because the identity map contains no generating morphisms. The Symmetry **Har** has the same (lack of) internal wiring M and node labels N, but with the R boundary map as a permutation.

Definition 5.23 (Symmetry **Har**)**.** The symmetric **Har** of type $m + n \to n + m$ is given by the (M, L, R, N) with

$$M = \mathbf{0}_{m+n,m+n} \qquad L = \mathsf{id}_{m+n} \qquad R = \begin{vmatrix} \mathbf{0} & \mathsf{id}_n \\ \mathsf{id}_m & \mathbf{0} \end{vmatrix} \qquad N = \mathbf{0}_{m+n,1}$$

Note that we could alternatively have chosen to set $R = \mathsf{id}$ and L as a permutation because 'the' symmetry is actually an isomorphism class of **Har**s. We now define the *singleton* **Har**, which corresponds to an individual generating morphism in Σ_1.

Definition 5.24 (Singleton **Har**)**.** Fix a generating morphism $g\colon m \to n \in \Sigma_1$. the **singleton**[3] **Har** is (M, L, R, N) where

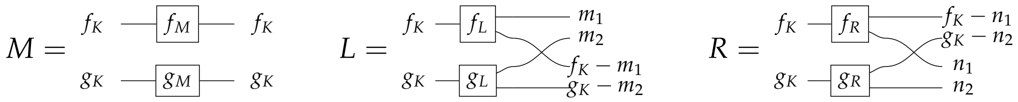
$$M = \begin{vmatrix} 0 & 0 & 0 \\ S & 0 & 0 \\ 0 & T & 0 \end{vmatrix} \qquad L = \mathsf{id}_K \qquad R = \mathsf{id}_K \qquad N = (\mathbf{0}_{1,m}, g, \mathbf{0}_{1,n})$$

and $S \in \mathbf{Mat}_{\mathbb{N}}(1, m)$ is the row vector, $(1, 2, \ldots, m)$ while $T \in \mathbf{Mat}_{\mathbb{N}}(n, 1)$ is the column vector $(1, 2, \ldots, n)$.

Remark 5.25. The singleton **Har** is in left *and* right boundary order, since the nodes S are the ordered left boundary nodes, and the nodes T the ordered right boundary nodes.

We now define two operations on **Har**s: tensor product and composition. We will see that **Har**s form a symmetric monoidal category with these operations in Section 5.6

Definition 5.26 (Tensor Product of **Har**s)**.** Let $f : m_1 \to n_1$ and $g : m_2 \to n_2$ be **Har**s. The tensor product $f \otimes g$ is given by (M, L, R, N), where



and N given by appending f_N and g_N, i.e., the block vector $(f_N \quad g_N)$.

[3] 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.

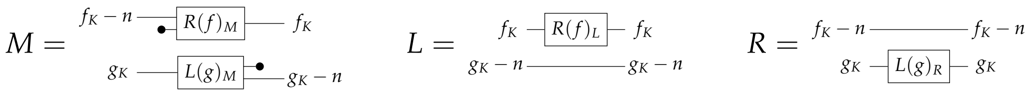
Remark 5.27. Recall that the tensor product in **Mat** is the direct sum, so $(f \otimes g)_M$ is the block matrix

$$(f \otimes g)_M = \begin{vmatrix} f_M & 0 \\ 0 & g_M \end{vmatrix}$$

The definitions for L, R are similar, but with additional bookkeeping to ensure that boundary orderings are respected.

Finally, we can define *composition* of **Har**s.

Definition 5.28 (Composition of **Har**s)**.** Let $f\colon m \to n$ and $g : n \to o$ be **Har**s. Composition $f \,⨟\, g$ is the 4-tuple (M, L, R, N) where



and 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. [4]

In the definition of composition, the morphisms —• and •— represent the projection $\pi_1 : A \times B \to B$ and injection $\iota_0 : A \to A \times B$ morphisms, respectively. The intuition is that $M \,⨟\, \pi_1$ selects the last *columns* of M, and $\iota_0 \,⨟\, M$ as selects the first *rows*. In more graphical terms, we are simply discarding the outgoing edges of the right boundary nodes of f and the *incoming* edges of the *left* boundary nodes of g. Since in both cases there are no such edges, no information is lost. In fact, this operation can be considered as 'gluing' f and g along the boundary as in a pushout of cospans.

5.6 The Category of Hars

We will now show that **Har**s form a **PROP** over a monoidal signature Σ which we denote $\mathbf{Har}_\Sigma$. In addition, we will show that $\mathbf{Har}_\Sigma$ is isomorphic to $\mathbf{Free}_\Sigma$: the free **PROP** on a monoidal signature Σ. In what follows, we will therefore refer to a fixed, arbitrary signature Σ, which is assumed to have a single generating object.

We will first need to define how **Har**s form a category.

Theorem 5.29 (The Category $\mathbf{Har}_\Sigma$)**.** **Har***s form a* **PROP** *with morphisms $m \to n$ the equivalence classes of values in $\mathbf{Har}_{m,n}$ under the relation $\sim$. Identities and symmetries are as given in Definitions 5.22 and 5.23, respectively. Composition and tensor product are as in Definitions 5.28 and 5.26.*

[4] We simply discard the part of the node labels of g corresponding to its left boundary nodes; this does not 'lose information' because their labels are known to be • as a consequence of Definition 5.14.

Proof. We give a graphical proof that composition is assocative up to permutation in Proposition B.6, and so composition is associative for equivalence classes of **Har**s. 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 of matrices is. □

In order to show an isomorphism between $\mathbf{Har}_\Sigma$ and $\mathbf{Free}_\Sigma$, we will construct two functors: Har and Hyp, as pictured below.

$$\begin{tikzcd}
& \mathbf{Free}_\Sigma \arrow[dl, leftrightarrow, "\cong"'] \arrow[dr, "\mathsf{Har}"] & \\
\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA} & & \mathbf{Har}_\Sigma \arrow[ll, "\mathsf{Hyp}"]
\end{tikzcd}$$

The composition of Hyp with the isomorphism $\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA} \cong \mathbf{Free}_\Sigma$ from [17], will form the inverse to **Har**.

Let us now define the functor Har.

Definition 5.30. Denote by $\mathsf{Har} : \mathbf{Free}_\Sigma \to \mathbf{Har}_\Sigma$ the identity-on-objects symmetric monoidal functor defined inductively so that generating morphisms $g \in \Sigma_1$ are mapped to their corresponding singleton **Har**s, and tensor and composition of morphisms is mapped to tensor and composition of **Har**s, respectively.

In fact, *every* morphism of **Har** can be constructed this way. In other words, the singleton **Har**s constructed from generating morphisms of Σ are sufficient as a 'basis' to construct every morphism in $\mathbf{Har}_\Sigma$. This is stated formally in the following proposition.

Proposition 5.31. *The morphisms of* **Har** *are generated by the monoidal signature with a single object and generating morphisms* id, σ, *and* $\{\mathsf{Har}(g) \mid g \in \Sigma_1\}$.

Proof. A **Har** is called a 'permutation **Har**' when it has no nodes labeled $\circ$. Every $h \in \mathbf{Har}_{m,n}$ can be factored as $f \fatsemi p$ for some permutation **Har** p so that $f_M = h_M$ and $f_R = \mathsf{id}$. Further, by the acyclicity property, every $h \in \mathbf{Har}_{m,n}$ with $h_R = \mathsf{id}$ can be decomposed into the form $f \fatsemi (\mathsf{id} \otimes \mathsf{Har}(g) \otimes \mathsf{id})$ for some $g \in \Sigma_1$. Consequently, we can decompose any **Har** into the following form.

$$p_1 \fatsemi (\mathsf{id} \otimes \mathsf{Har}(g_1) \otimes \mathsf{id}) \fatsemi p_2 \fatsemi (\mathsf{id} \otimes \mathsf{Har}(g_2) \otimes \mathsf{id}) \fatsemi \ldots \fatsemi p_N$$

Since h was an arbitrary **Har**, it is clear that any h can be expressed as a composition of such 'layers' of generators separated by permutations. Thus, the morphisms of $\mathbf{Har}_\Sigma$ are freely generated by morphisms id, σ, and $\mathsf{Har}(g)$ for $g \in \Sigma_1$. □

As an immediate consequence of Proposition 5.6, we may also conclude that any **Har** satisfying the conditions described in Definition 5.14 corresponds to a string diagram. Therefore, not only can one construct **Har**s by composition and tensor product of singleton **Har**s, but also by directly specifying the structure of the underlying bipartite graph.

Remark 5.32. Note that the above proof also suggests an algorithm for *decomposition* of **Har**s into terms. To obtain an ordering of generators $g_1 \dots g_N$, one can use a topological algorithm such as Kahn's algorithm [71] on the underlying graph.

In order to prove that $\mathbf{Har}_\Sigma$ is isomorphic to $\mathbf{Free}_\Sigma$, we will essentially show that the functor Har amounts to a 'relabeling of generators'. This can be stated more generally for any monoidal category as the following 'change of basis' theorem.

Theorem 5.33 (Change of Basis). *Fix a monoidal signature Σ, and suppose $\mathscr{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathscr{D}$ are strict symmetric monoidal identity-on-objects functors. If $\mathscr{C}$ is generated by Σ, $\mathscr{D}$ is generated by $\{F(g) \mid g \in \Sigma_1\}$, and for all $g \in \Sigma_1$ we have $G(F(g)) = g$ then $\mathscr{C} \cong \mathscr{D}$.*

Proof. It suffices to show that the functors F and G are inverses. Note that in the following proof we will use juxtaposition for composition to ease the notational burden, so for example fg means $f \mathbin{;} g$ and FG means $F \mathbin{;} G$.

We first check $FG = \mathsf{id}$. Since every morphism $h \in \mathscr{C}$ is formed by composition and tensor product of id, σ, and generators $g \in \Sigma_1$, we proceed by induction:

1. If $h \in \{\mathsf{id}, \sigma\}$ then $FG(h) = h$ because F, G are symmetric monoidal functors.
2. If $h \in \Sigma_1$ then $FG(h) = G(F(h)) = h$ by assumption.
3. If $h = fg$ then $FG(h) = G(F(fg)) = G(F(f)F(g)) = G(F(f))G(F(g)) = fg$ by inductive hypothesis.
4. If $h = f \otimes g$ then $FG(h) = G(F(f \otimes g)) = G(F(f) \otimes F(g)) = G(F(f)) \otimes G(F(g)) = f \otimes g$ by inductive hypothesis.

Now check $GF = \mathsf{id}$. Choose $h \in \mathscr{C}$ and proceed again by induction. Cases 1, 3, and 4 can be shown in the same way as above, so we need only demonstrate that $F(G(h)) = h$. This follows immediately by assumption: since $\mathscr{C}$ is generated by operations Σ_1, we need only check the 'base cases' where $h = F(g)$. Thus, if $h = F(g)$ for $g \in \Sigma_1$ then $F(G(h)) = F(G(F(g))) = F(g) = h$. □

Remark 5.34. Theorem 5.33 amounts to a renaming of generators, and so is analogous to a change of basis in linear algebra.

Now, in order to show that $\mathbf{Har}_\Sigma$ is indeed isomorphic to $\mathbf{Free}_\Sigma$, we will use Theorem 5.33. We have already constructed a functor $\mathsf{Har} : \mathbf{Free}_\Sigma \to \mathbf{Har}_\Sigma$ and shown that $\mathbf{Har}_\Sigma$ is generated by the morphisms $\{\mathsf{Har} \mid g \in \Sigma_1\}$ in Proposition . It therefore remains to construct the functor $\mathsf{Hyp} : \mathbf{Har}_\Sigma \to \mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA}$, and show that its composition with the isomorphism $\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA} \cong \mathbf{Free}_\Sigma$ is inverse to Har on generators Σ_1.

Definition 5.35. We define the functor $\mathsf{Hyp} : \mathbf{Har}_\Sigma \to \mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA}$ on morphisms $h \in \mathbf{Har}_{m,n}$ by constructing a monogamous acyclic hypergraph cospan $m \xrightarrow{s} H \xleftarrow{t} n \in \mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA}$ as follows.

Hypergraph H The graph represented by h_M and h_N consists of vertices v_i labeled $\bullet$, and and vertices e_j labeled $(\circ, g_j)$ for $g_j \in \Sigma_1$. We define H as the hypergraph with hypernodes v_i and labeled hyperedges $e_j = ([s_1, s_2, \ldots, s_A], [t_1, t_2, \ldots, t_B], g_j)$ where s_k is the unique vertex u such that there exists an edge (u, e_j, k) in h_M, and t_k the unique vertex w such that there is an edge (e_j, w, k).

Cospan legs The cospan legs $l : m \to H$ and $r : n \to H$ are constructed from h_L and h_R, respectively. Concretely, if we view each leg as a function mapping a natural number to a node in H, then $l = f_L \mathbin{⨟} \pi_0^{m, f_K - m}$ and $r = f_R \mathbin{⨟} \pi_1^{f_K - n, n}$ with $\pi_0^{X,Y} : X \times Y \to X$ and $\pi_1^{X,Y} : X \times Y \to Y$.

Proposition 5.36. Hyp *is a Symmetric Monoidal Functor*

Proof. It is clear that Hyp preserves identities: $\mathsf{Hyp}(\mathsf{id})$ is mapped to the discrete hypergraph with identity cospans. Similarly, $\mathsf{Hyp}(\sigma)$ is a cospan of the discrete hypergraph whose left leg is identity and right leg is the symmetry, so $\mathsf{Hyp}(\sigma) = \sigma$.

Further, one can verify that the operation of **Har** composition essentially mimics the computation of pushouts of hypergraphs, and Definition 5.28 tells us that the left and right legs of the cospan $\mathsf{Hyp}(f) \mathbin{⨟} \mathsf{Hyp}(g)$ are equal to those of $\mathsf{Hyp}(f \mathbin{⨟} g)$.

Finally, we can see that since the tensor product of $f \otimes g$ of **Har**s is the direct sum of matrices, the hypergraph representation of $\mathsf{Hyp}(f \otimes g)$ coincides with $\mathsf{Hyp}(f) \otimes \mathsf{Hyp}(g)$, and once again Definition 5.26 ensures that the left and right legs of the cospan coincide. □

Corollary 5.37. *There is a symmetric monoidal functor* $\mathsf{Har}^* : \mathbf{Har}_\Sigma \to \mathbf{Free}_\Sigma$ *obtained by composing* Hyp *with the isomorphism* $\mathsf{Cospan}(\mathbf{Hyp}_\Sigma)_{MA} \cong \mathbf{Free}_\Sigma$. *given in [17].*

Moreover, this functor is inverse on generators.

Proposition 5.38. $\mathsf{Har}^*(\mathsf{Har}(g)) = g$ *for* $g \in \Sigma_1$

Proof. Let $g \in \Sigma_1$ be a generating morphism, so $\mathsf{Har}(g)$ is a singleton **Har**. Then $\mathbf{Hyp}(\mathbf{Har}(g))$ gives hypergraph cospan with a single hyperedge labeled g. Under the isomorphism given in [17], this corresponds to the Σ-term consisting of a single generator g (see [18, p. 22]). Thus, the composite functor $\mathsf{Har}^* : \mathbf{Har}_\Sigma \to \mathbf{Free}_\Sigma$ is inverse on generators. □

Theorem 5.39. $\mathsf{Har} : \mathbf{Free}_\Sigma \to \mathbf{Har}_\Sigma$ *is an isomorphism of* **PROP***s*

Proof. We have shown that

- There is a symmetric monoidal functor $\mathsf{Har} : \mathbf{Free}_\Sigma \to \mathbf{Har}_\Sigma$ (Definition 5.30)
- There is a symmetric monoidal functor $\mathsf{Har}^* : \mathbf{Har}_\Sigma \to \mathbf{Free}_\Sigma$ (Corollary 5.37)
- $\mathbf{Har}_\Sigma$ is generated by the singleton **Har**s corresponding to the operations $g \in \Sigma_1$ (Proposition 5.6)
- $\mathsf{Har}^*(\mathsf{Har}(g)) = g$ for $g \in \Sigma_1$. (Proposition 5.38)

Thus, by Theorem 5.33, Har and Har^* form an isomorphism $\mathbf{Free}_\Sigma \cong \mathbf{Har}_\Sigma$, □

5.7 Complexity

We now give the time complexity of the composition and tensor product operations defined in Section 5.5. Since our algorithm is expressed in terms of matrix multiplication, one might expect that the time complexity of the operations presented here to be at best $O(n^{2.37188})$ (at time of writing [43]). However, the matrices representing a **Har** have a high degree of *sparsity*, and so it is possible to significantly improve on this bound.

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

Proposition 5.40. *(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 5.14, vertices in f_M labeled $\bullet$ may have at most one incoming and outgoing edge. Moreover, each vertex v labeled $(\circ, g)$ must have exactly a incoming and b outgoing edges for a generator $g : a \to b$. 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 [61] 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 **Har**s are linear-time operations. To make this clear, we introduce the following proposition:

Proposition 5.41. *(Permutation of* **Har***s has linear complexity)*
Choose some $f \in \mathbf{Har}_\Sigma$ *and a permutation matrix* $P \in \mathbf{Mat}(f_K, f_K)$*. Then* $P \fatsemi f_M$ *and* $f_M \fatsemi P$ *can be computed in linear time.*

Proof. For matrices $A, B \in \mathbf{Mat}(k,k)$, the complexity of Gustavson's sparse matrix multiplication routine [61] is $O(2k + \mathtt{nnz}(A) + m)$. Here $\mathtt{nnz}(A)$ 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 5.40), 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 $\mathtt{nnz}(f_M)$ 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 [61], which can compute $P \fatsemi f_M \fatsemi Q^T$ in $O(\mathtt{nnz}(f_M))$ operations. Since $\mathtt{nnz}(f_M)$ is $O(f_K)$, this operation has linear complexity. □

Using proposition 5.41 we can now show that composition and tensor product have linear time complexity.

Proposition 5.42. *(Tensor Product of* **Har***s* $f \otimes g$ *is* $O(f_K + g_K)$*)*
Given $f \in \mathbf{Har}(m_1, n_1)$ *and* $g \in \mathbf{Har}(m_2, n_2)$*, computation of* $f \otimes g$ *is* $O(f_K + g_K)$*.*

Proof. It is clear from Definition 5.14 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 also. □

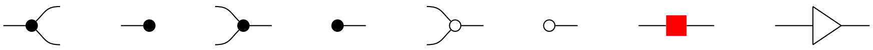
Proposition 5.43. *(Composition of* **Har***s* $f \fatsemi g$ *is* $O(f_K + g_K)$*)*
Given $f \in \mathbf{Har}(m, n)$ *and* $g \in \mathbf{Har}(n, o)$*, computation of* $f \fatsemi g$ *is* $O(f_K + g_K)$*.*

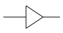
Proof. The proof is similar to that of Proposition 5.42, except that we must include the cost of the operations $R(f)$ and $L(g)$. These operations are linear by proposition 5.41, and so the composition $f \fatsemi g$ must also be $O(f_K + g_K)$. □

5.8 Empirical

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` [82, 83].[5] For implementation and benchmarking code, see Implementation D.4.

The following benchmarks assume a fixed monoidal signature. Specifically, we use the generators of $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ (Definition 3.22). More concretely, we will build morphisms from the following generators:



where the generator  represents a `NOT` gate, alternatively thought of as the map . We choose the generators of $\mathbf{PolyCirc}_{\mathbb{Z}_2}$ since they serve as a syntax for boolean circuits as witnessed by the presentation described by Lafont [76]. We specifically choose such circuits as a real-world example of a presentation in which diagrams will necessarily be very large. For example, a string-diagrammatic representation of a CPU would need (at least) hundreds of thousands of generators.

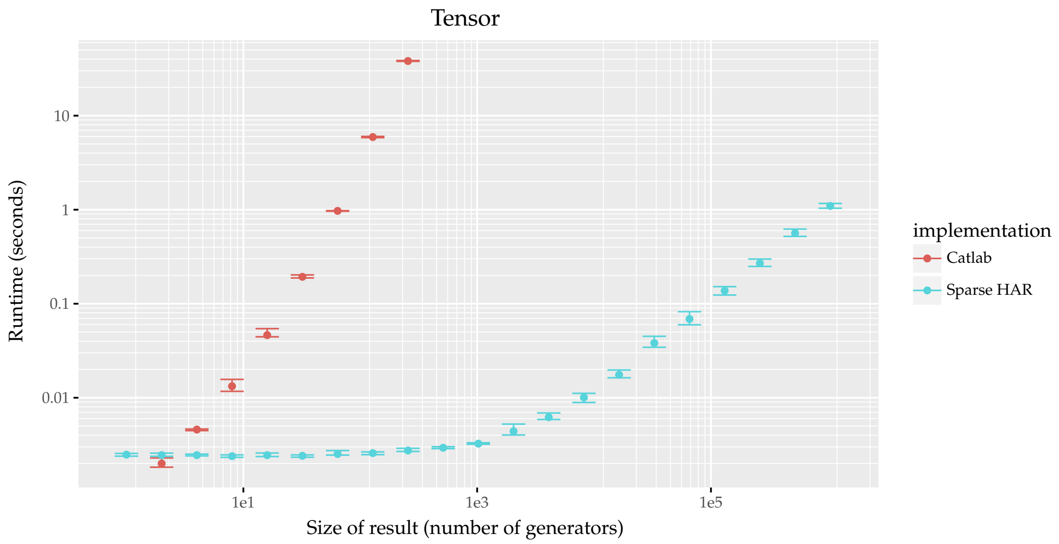
Experiment Details Each benchmark has the same structure: for $k \in \{1 \ldots 20\}$ we construct two string diagrams consisting of 2^{k-1} generators, and then measure the time taken to compute the tensor product or composition of those diagrams. Each measurement is repeated 10 times for each k, and plots of each experiment show the mean computation time with minimum and maximum error bars. In each case, the x-axis is the resulting diagram size (meaning the number of generators), and the y-axis is the time taken to compute a given operation. If a result takes longer than 60 seconds to compute, it is omitted. More details of the experimental setup can be found in Appendix B.2.

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.

5.8.1 Benchmark #1: Repeated Tensor

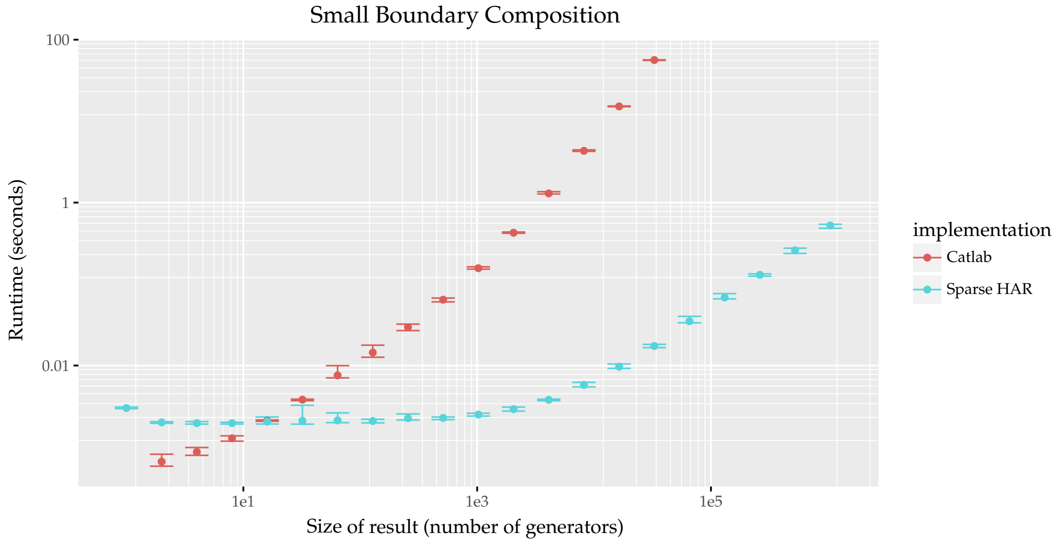
The first experiment measures the performance of the tensor product of large representations. Concretely, let f be the k-fold tensor product of , i.e. $f =$ $\otimes \overset{k}{\ldots} \otimes$. We measure the performance of computing $f \otimes f$.

[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 5.9.



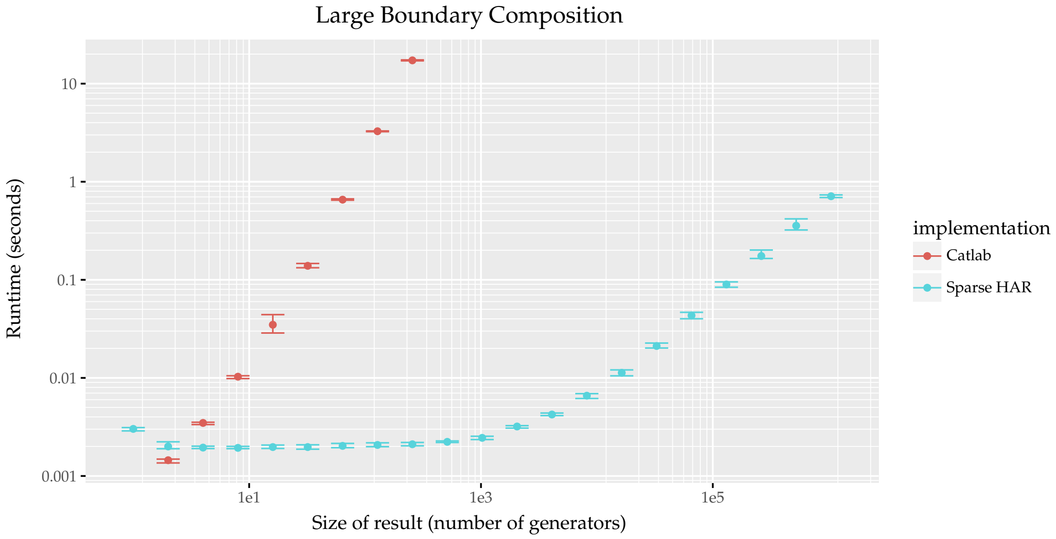
5.8.2 Benchmark #2: Small-Boundary Composition

We measure the performance of composition $m \xrightarrow{f} n \xrightarrow{g} o$ along a small shared boundary, i.e., where $n \ll f_K + g_K$. Concretely, let f be the k-fold composition of —▷—, so that $f =$ —▷— $\fatsemi \overset{k}{\ldots} \fatsemi$ —▷—. We measure the performance of computing $f \fatsemi f$.



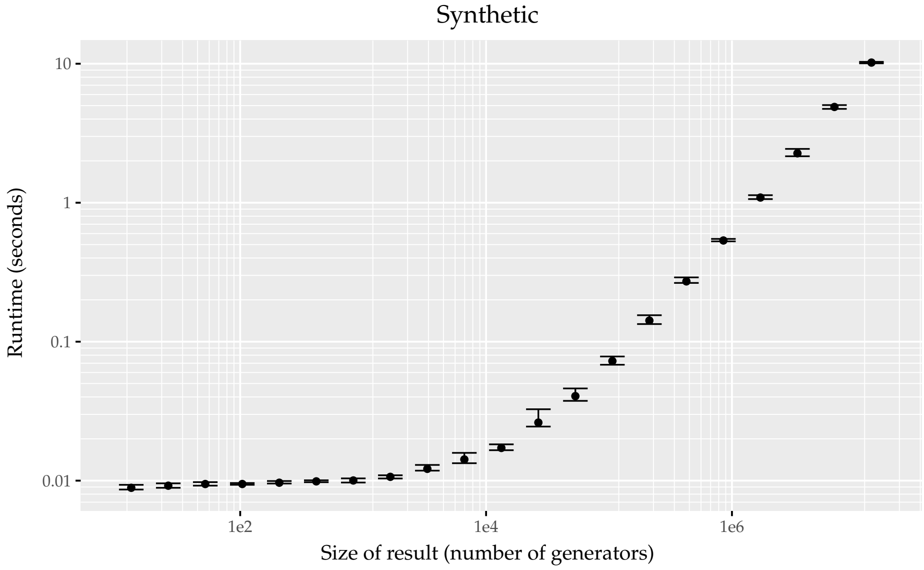
5.8.3 Benchmark #3: Large-Boundary Composition

We measure the performance of composition $m \xrightarrow{f} n \xrightarrow{g} o$ along a *large* shared boundary, i.e. where $n \approx \mathsf{min}(f_K, g_K)$. In particular, let f be the k-fold tensor product of —▷—, then we measure $f \fatsemi f$.



5.8.4 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 a 2^{k-1}-bit adder circuits to form a 2^k-bit adder.



5.9 Extensions to Hars

In this chapter, we defined a datastructure for representing morphisms of the free **PROP** on a given monoidal signature. We also gave algorithms for tensor and composition of such morphisms, and showed how they have time complexity linear in the size of the resulting diagram. However, in general we would like to be able to model a larger class of diagrams. Specifically, it will be useful for the next chapter to consider string diagrams for categories having more than one generating object.

An extension of the definitions in this chapter to categories with multiple generating objects is straightforward. We therefore discuss these extensions informally now. Let Σ now denote a monoidal signature with its set of generating objects Σ_0 no longer constrained to be the 1-element set. One can model string diagrams of $\mathbf{Free}_\Sigma$ by adding additional label information for those **Har** vertices labeled $\bullet$. Specifically, such vertices must additionally be labeled with an element of Σ_0.

However, we must also make sure that object labels of $\bullet$-labeled vertices are compatible with the operations of Σ_1 and the interfaces of the diagram. Thus, two additional constraints on **Har**s are required. First, suppose v is a vertex labeled $(\circ, g)$, with $g : A_1 \otimes \ldots \otimes A_m \to B_1 \otimes \ldots \otimes B_n$. We require for each labeled edge (x, v, i) that the label of x is $(\bullet, A_i)$, and similarly for edges (v, y, j) that labels of y are $(\bullet, B_j)$. Secondly, we must forbid composition of **Har**s having 'incompatible boundaries'. That is, for a **Har** $f : m \to n$ with right interface nodes labeled $A_1 \ldots A_n$ and $g : n \to o$ with left interface nodes labeled $B_1 \ldots B_n$, we require that $A_i = B_i$ for each $i \in \{1 \ldots n\}$.

In summary, we must make the following changes to the **Har** datastructure.

- Node labels N are in the set $((\{\bullet\} \times \Sigma_0) + (\{\circ\} \times \Sigma_1))$
- Vertices labeled $(\circ, g)$, incoming edges $v_1 \ldots v_m$ are 'well-typed' with respect to the generator g.
- Composition of **Har**s having incompatible boundaries is forbidden.

Another useful potential extension is to support string diagrams for signatures augmented with a chosen Special Frobenius structure (Definition 5.7). However, this is not possible without more significant changes to the **Har** datastructure. The reason is that the 'internal wiring' and interfaces of **Har**s are encoded as adjacency matrices. This limits the number of edges between two nodes to at most 1, whereas the hypergraphs corresponding to string diagrams with the additional Frobenius structure have no such restriction. Essentially, the adjacency matrix encoding makes the assumption of monogamicity, which is not present in the Frobenius case. We therefore leave this extension to future work, which we discuss further in Chapter 7.

Chapter 6

Strictness and Coherence

6.1 Overview

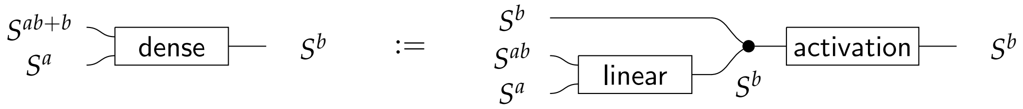
Until now, we have only dealt in *strict* symmetric monoidal categories. However, the final chapter of this thesis is concerned with non-strictness. More precisely, we give a *strict* graphical calculus for reasoning about *non-strict* monoidal categories. Even more precisely, given an arbitrary and potentially non-strict monoidal category $\mathscr{C}$, we give a presentation by generators and equations of its strict equivalent, denoted $\overline{\mathscr{C}}$.

The construction given in this chapter results in a new proof of Mac Lane's well-known strictness theorem [80, p.257]. However, the primary motivation is to augment datastructures such as those described in Chapter 5 with the ability to represent terms of non-strict categories. While Mac Lane's proof guarantees the validity of graphical reasoning for non-strict categories, the precise details are problematic from an implementation perpective. One issue in particular is the reliance on Mac Lane's *coherence* theorem [80] for monoidal categories. In contrast, our proof of the strictness theorem is elementary. This means we are able to give a novel *graphical* proof of the coherence theorem in terms of the strictness theorem.

Reliance on the coherence theorem is problematic for two reasons. Firstly, as observed by Hines [65, Section 1.4] the theorem itself is often mis-stated as 'all diagrams built from coherence isomorphisms commute'. In fact, Hines gives a counterexample to this statement in [65, Definition 2.1]. The correct statement of the coherence theorem in fact only guarantees that certain 'formal' diagrams [80] commute. Thus, reliance on the coherence theorem introduces implementation subtleties which we would like to avoid. The second issue stems from the definition of the tensor product of morphisms in the equivalent strict category. Mac Lane uses the coherence theorem to witness the existence of certain morphisms without needing to define them specifically [80, p.258].

However, in order to implement the strictness construction in code, it is necessary to be completely explicit about the definition of such maps.

In addition to allowing the representation of non-strict terms, the construction presented here has additional benefits applicable to our guiding application of machine learning. Concretely, the 'strictification' of a *strict* category makes the construction of large diagrams more ergonomic by allowing for 'bundling' of wires. Consider for example the machine learning models defined in Part I of the thesis. Recall that we defined models as morphisms of reverse differentiable categories having the form $P \times A \to B$. For example, the 'dense' map of Definition 3.36 was defined as follows



where $P = S^{b \cdot a + b}$. Although pictured as distinct wires, the object corresponding to the two inputs to `dense` is a single natural number $ab + b + a$. Without specifying P explicitly, it is therefore ambiguous which of the inputs are the parameters, and so it is not possible to precisely specify the 'learning step' morphism (4.4) of Reverse Derivative Ascent.

The strictification construction presented here allows for the removal of this ambiguity. In particular, wires may be 'bundled' so that the `dense` morphism can have the form $P \otimes X$, where P and X are *generating* objects of the category. In this case, the domain of the `dense` morphism would genuinely be a *binary* tensor product. Specifying a model would therefore mean simply giving a morphism whose domain is a binary product whose first object represents the parameters. The learning step morphism can then be defined in terms of arrows with binary domains. Categorically speaking, this will require the introduction of a generating object for every object in the 'base' category (in this case $\mathbf{PolyCirc}_S$). We will describe this process formally in Section 6.4.

6.1.1 Relationship to Published Work

Aside from the preliminaries covered in Section 6.3, the content of this chapter consists of the author's individual contributions originally published in [107]. Parts of this chapter are reproduced from this work verbatim.

6.2 Synopsis

As usual, we begin with preliminary background material in Section 6.3. In this case, the background material consists of the definitions of monoidal and symmetric monoidal functors. Although we have so far assumed familiarity with these definitions, we will now need to be precise about the various maps involved.

Section 6.4 marks the beginning of our contributions, where we define the 'graphical language of non-strict monoidal categories'. Concretely, in Definition 6.5 we give a presentation by generators and equations of a category $\overline{\mathscr{C}}$, which we will later prove to be the 'strictification' of an arbitrary monoidal category $\mathscr{C}$. As a first step towards this proof, we construct a functor $\mathcal{S} : \mathscr{C} \to \overline{\mathscr{C}}$ in Definition 6.7, and show it is monoidal in Proposition 6.8.

Section 6.5 begins with the construction of the 'nonstrictification' functor $\mathcal{N} : \overline{\mathscr{C}} \to \mathscr{C}$ in Definition 6.10. The definition of $\mathcal{N}$ is more complex than for $\mathcal{S}$, and so it is necessary to explicitly construct its coherence maps in Definitions 6.12 and 6.14 before proving it is monoidal in Proposition 6.15.

In Section 6.6, it is shown that strictification and non-strictification functors $\mathcal{S}$ and $\mathcal{N}$ constitute a monoidal equivalence between $\mathscr{C}$ and $\overline{\mathscr{C}}$. We first show that $\mathcal{N} \circ \mathcal{S} = \mathsf{id}_{\mathscr{C}}$ in Proposition 6.16, which makes the composite a split idempotent. Finally, in Proposition 6.21, we show that the composite $\mathcal{S} \circ \mathcal{N}$ isomorphic to the identity functor. These two facts together yield Theorem 6.22, constituting a new proof of Mac Lane's Strictness Theorem [80] which crucially does not rely on the coherence theorem.

Section 6.7 exploits our proof of the strictness theorem to give a novel *graphical* proof of Mac Lane's *coherence* theorem [80] (Theorem 6.23). Our proof is essentially the mirror of Mac Lane's, who defines a certain preorder before showing it is free in a particular sense. In contrast, we *define* the category $\mathscr{W}$ (Definition 6.24) to be free, and then use our graphical calculus to show that its strictification $\overline{\mathscr{W}}$ is a preorder (Proposition 6.38.) This implies that $\mathscr{W}$ is a preorder (Corollary 6.39), resulting a novel graphical proof of the Coherence Theorem.

Section 6.8 concludes the chapter by demonstrating how the strictness theorem extends to the *symmetric* monoidal case. Proposition 6.43 shows that if $\mathscr{C}$ is symmetric monoidal then $\overline{\mathscr{C}}$ is as well. In addition, this implies that the functors $\mathcal{S}$ and $\mathcal{N}$ are symmetric monoidal, which is shown in Propositions 6.44 and 6.45, respectively.

6.3 Preliminaries

In the following sections, we will define a pair of functors forming a monoidal equivalence between a monoidal category $\mathscr{C}$ and its strictification $\overline{\mathscr{C}}$. It will therefore be necessary to recall these concepts in detail. We begin with the definition of a *monoidal* functor.

Definition 6.1. Monoidal Functor
Let $(\mathscr{C}, \otimes, I_{\mathscr{C}})$ and $(\mathscr{D}, \bullet, I_{\mathscr{D}})$ be monoidal categories. A *monoidal functor* is a functor $F : \mathscr{C} \to \mathscr{D}$ equipped with natural isomorphisms $\Phi_{X,Y} : F(X) \bullet F(Y) \to F(X \otimes Y)$ and $\phi : I_{\mathscr{D}} \to F(I_{\mathscr{C}})$ such that the following diagrams commute for all objects $A, B, C \in \mathscr{C}$.

$$\begin{tikzcd}
(F(A) \bullet F(B)) \bullet F(C) \arrow[d, "\Phi_{A,B} \bullet \mathrm{id}_{F(C)}"'] & F(A) \bullet (F(B) \bullet F(C)) \arrow[l, "\alpha_{\mathscr{D}}"'] \arrow[d, "\mathrm{id}_{F(A)} \bullet \Phi_{B,C}"] \\
F(A \otimes B) \bullet F(C) \arrow[d, "\Phi_{A \otimes B, C}"'] & F(A) \bullet F(B \otimes C) \arrow[d, "\Phi_{A, B \otimes C}"] \\
F((A \otimes B) \otimes C) & F(A \otimes (B \otimes C)) \arrow[l, "F(\alpha_{\mathscr{C}})"]
\end{tikzcd} \tag{6.1}$$

$$\begin{tikzcd}
F(A) \bullet I_{\mathscr{D}} \arrow[r, "\mathrm{id}_{F(A)} \bullet \phi"] \arrow[d, "\rho_{\mathscr{D}}"'] & F(A) \bullet F(I_{\mathscr{C}}) \arrow[d, "\Phi_{A, I_{\mathscr{C}}}"] & I_{\mathscr{D}} \bullet F(B) \arrow[r, "\phi \bullet \mathrm{id}_{F(B)}"] \arrow[d, "\lambda_{\mathscr{D}}"'] & F(I_{\mathscr{C}}) \bullet F(B) \arrow[d, "\Phi_{I_{\mathscr{C}}, B}"] \\
F(A) & F(A \otimes I_{\mathscr{C}}) \arrow[l, "F(\rho_{\mathscr{C}})"] & F(B) & F(I_{\mathscr{C}} \otimes B) \arrow[l, "F(\lambda_{\mathscr{C}})"]
\end{tikzcd} \tag{6.2}$$

We will also need the definition of an equivalence of categories.

Definition 6.2 (Equivalence of categories)**.** An equivalence is a pair of functors $\mathscr{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathscr{D}$ together with natural isomorphisms $\eta : \mathrm{id}_{\mathscr{C}} \to G \circ F$ and $\epsilon : F \circ G \to \mathrm{id}_{\mathscr{D}}$.

A *monoidal* equivalence of categories requires the natural transformations η and ϵ be *monoidal* natural transformations. We recall these now.

Definition 6.3 (Monoidal Natural Transformation)**.** Let $(F, \Phi, \phi) : (\mathscr{C}, \otimes, I_{\mathscr{C}}) \to (\mathscr{D}, \bullet, I_{\mathscr{D}})$ and $(G, \Gamma, \gamma) : (\mathscr{C}, \otimes, I_{\mathscr{C}}) \to (\mathscr{D}, \bullet, I_{\mathscr{D}})$ be monoidal functors. A **monoidal natural transformation** $\alpha : F \to G$ is a natural transformation such that the following diagrams commute.

$$\begin{tikzcd}
F(A) \bullet F(B) \arrow[r, "\alpha_A \bullet \alpha_B"] \arrow[d, "\Phi_{A,B}"'] & G(A) \bullet G(B) \arrow[d, "\Gamma_{A,B}"] \\
F(A \otimes B) \arrow[r, "\alpha_{A \otimes B}"'] & G(A \otimes B)
\end{tikzcd} \tag{6.3}$$

$$\begin{tikzcd}
 & I_{\mathscr{D}} \arrow[dl, "\phi"'] \arrow[dr, "\gamma"] & \\
F(I_{\mathscr{C}}) \arrow[rr, "\alpha_I"'] & & F(I_{\mathscr{C}})
\end{tikzcd} \tag{6.4}$$

Finally, we may define a *monoidal* equivalence of categories as follows.

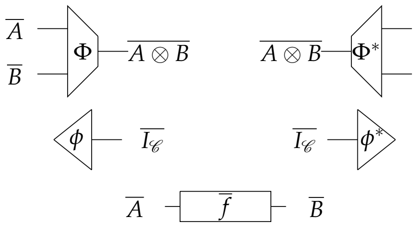
Definition 6.4 (Monoidal Equivalence)**.** Let $\mathscr{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathscr{D}$ be monoidal functors forming an equivalence of categories with natural isomorphisms $\eta : \mathsf{id}_{\mathscr{C}} \to G \circ F$ and $\epsilon : F \circ G \to \mathsf{id}_{\mathscr{D}}$. It is a **monoidal equivalence** when η and ϵ are *monoidal* natural isomorphisms.

6.4 Strictification

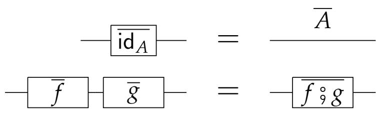
We begin by defining the 'strictification' of an arbitrary monoidal category $\mathscr{C}$ as a presentation by generators and equations of a new, strict monoidal category $\overline{\mathscr{C}}$.

Definition 6.5. Fix an arbitrary monoidal category $\mathscr{C}$. Its 'strictification' $(\overline{\mathscr{C}}, \bullet)$ is the strict monoidal category freely generated by:

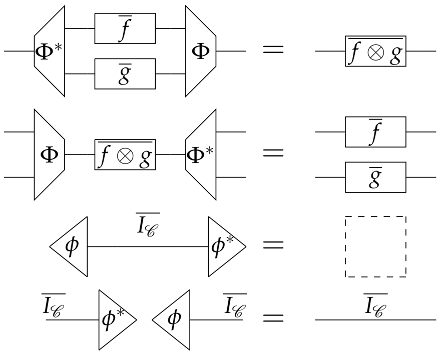
1. Generating objects $\overline{A}$ for each $A \in \mathscr{C}$
2. Generating morphisms (6.5), with $\overline{f} : \overline{A} \to \overline{B}$ for each $f : A \to B \in \mathscr{C}$
3. Functoriality equations (6.6)
4. Adapter equations (6.7), and
5. Monoidal equations (6.8)



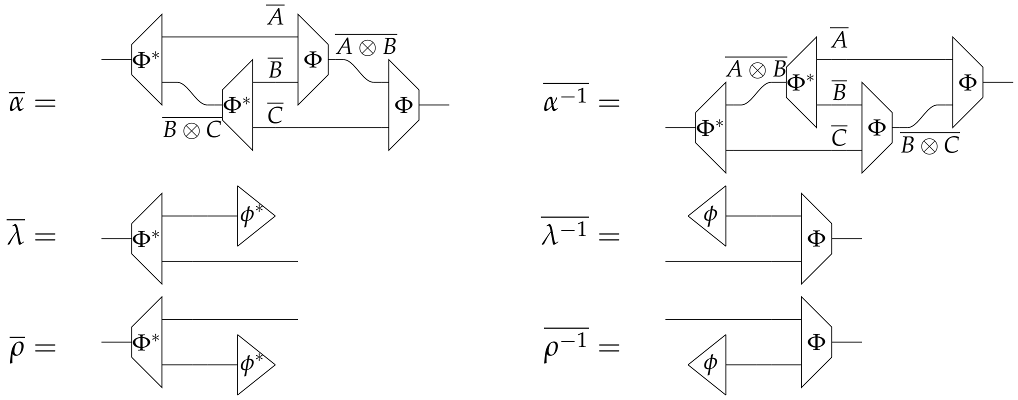
(6.5)



(6.6)



(6.7)



(6.8)

Remark 6.6. The *generating* objects of $\overline{\mathscr{C}}$ are *all* of the objects $\mathscr{C}$. This means that a morphism with m inputs and n outputs will have domain and codomain of the form $\overline{A_1} \bullet \cdots \bullet \overline{A_m}$ and $\overline{B_1} \bullet \cdots \bullet \overline{B_n}$, respectively, where each $\overline{X}$ is an object (not just a generating object) of $\mathscr{C}$. For example, if $A \otimes B$ is an object of $\mathscr{C}$, then $\overline{A \otimes B}$ is a *generating* object of $\overline{\mathscr{C}}$. Note also that since $\overline{\mathscr{C}}$ is strict by construction, we may use string diagrammatic notation without appeal to Mac Lane's strictness theorem.

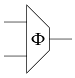
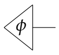
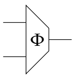
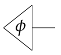
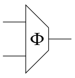
We can now define a monoidal functor from $\mathscr{C}$ to $\overline{\mathscr{C}}$. Note that this functor is one half of a monoidal equivalence, a fact which will be proven in Section 6.6.

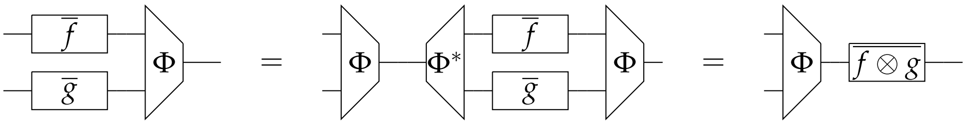
Definition 6.7. Let $\mathcal{S} : \mathscr{C} \to \overline{\mathscr{C}}$ be the *strictification functor* defined on objects and morphisms as $\mathcal{S}(A) := \overline{A}$ and $\mathcal{S}(f) := \overline{f}$, respectively

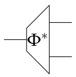
Proposition 6.8. *$(\mathcal{S}, \Phi, \phi)$ is a monoidal functor.*

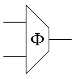
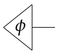
Proof. $\mathcal{S}$ preserves identities and composition (and is therefore a functor) by the functor equations (6.6):

$$\mathcal{S}(\mathsf{id}_A) = \overline{\mathsf{id}_A} = \mathsf{id}_{\overline{A}} \qquad\qquad \mathcal{S}(f \mathbin{⨾} g) = \overline{f \mathbin{⨾} g} = \overline{f} \mathbin{⨾} \overline{g} = \mathcal{S}(f) \mathbin{⨾} \mathcal{S}(g)$$

It is a *monoidal* functor using the adapter generators $\Phi =$  and $\phi =$  from (6.5). We must therefore have that  is a natural isomorphism and  an isomorphism, respectively. This is a straightforward consequence of the adapter equations. First, observe that $\phi \mathbin{⨾} \phi^* = \mathsf{id}$ by definition. The adapter  is natural as a straightforward consequence of the adapter equations (6.7):

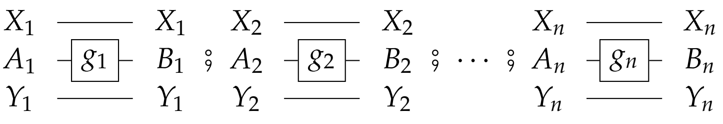


and similarly for its inverse . Finally, for $\mathcal{S}$ to be a monoidal functor we require that the diagrams of (6.1) and (6.2) commute. Again, this is precisely what the monoidal equations (6.8) state, and so the proof is complete. □

Remark 6.9. Notice that $\overline{\mathscr{C}}$ is *defined* by freely adding the requirements of Definition 6.1. Generators  and  and equations (6.7) give the natural isomorphism Φ and isomorphism ϕ, while the commuting diagrams (6.1) and (6.2) are precisely the 'monoidal' equations (6.8).

6.5 Nonstrictification

We can now define the functor $\mathcal{N} : \overline{\mathscr{C}} \to \mathscr{C}$ which will form the other half of a monoidal equivalence between the two categories. The definition of $\mathcal{N}$ is somewhat more involved than for $\mathcal{S}$. To apply the functor we must first decompose a diagram into a 'sequential normal form', consisting of a composition of 'layers'



where each g_i is a generator. This form is alternatively known in the literature; the 'layer' terminology is used by Lafont [77], while Alvarez-Picallo et al. [5] call diagrams of this form 'maximally sequential foliations'. We take advantage of this decomposition to define $\mathcal{N}$ on the individual layers $\mathsf{id}_X \bullet q \bullet \mathsf{id}_Y$ for some generator q, and then freely on composition so that $\mathcal{N}(f \mathbin{⨟} g) = \mathcal{N}(f) \mathbin{⨟} \mathcal{N}(g)$. We give more detail on this decomposition in Appendix C.1.

Definition 6.10. The **nonstrictification** functor $\mathcal{N} : \overline{\mathscr{C}} \to \mathscr{C}$ is defined inductively on objects:

$$\mathcal{N}(I_{\overline{\mathscr{C}}}) := I_{\mathscr{C}} \qquad \mathcal{N}(\overline{A}) := A \qquad \mathcal{N}(\overline{A} \bullet R) := A \otimes \mathcal{N}(R)$$

And on morphisms we give a recursive definition, with the following base cases

$$\mathcal{N}(\mathsf{id}_{I_{\overline{\mathscr{C}}}}) = \mathsf{id}_{I_{\mathscr{C}}}$$

$$\begin{array}{ccc}
\mathcal{N}(\overline{f}) := f & \mathcal{N}(\overline{f} \bullet \mathsf{id}_Y) := f \otimes \mathsf{id}_{\mathcal{N}(Y)} & \mathcal{N}(\mathsf{id}_{\overline{A}} \bullet \overline{f}) := \mathsf{id}_A \otimes f \\
\mathcal{N}(\Phi_{A,B}) := \mathsf{id}_{A\otimes B} & \mathcal{N}(\Phi_{A,B} \bullet \mathsf{id}_Y) := \alpha_{A,B,\mathcal{N}(Y)} & \mathcal{N}(\mathsf{id}_{\overline{A}} \bullet \Phi_{B,C}) := \mathsf{id}_{A\otimes(B\otimes C)} \\
\mathcal{N}(\Phi^*_{A,B}) := \mathsf{id}_{A\otimes B} & \mathcal{N}(\Phi^*_{A,B} \bullet \mathsf{id}_Y) := \alpha^{-1}_{A,B,\mathcal{N}(Y)} & \mathcal{N}(\mathsf{id}_{\overline{A}} \bullet \Phi^*_{B,C}) := \mathsf{id}_{A\otimes(B\otimes C)} \\
\mathcal{N}(\phi) := \mathsf{id}_{I_{\mathscr{C}}} & \mathcal{N}(\phi \bullet \mathsf{id}_Y) := \lambda^{-1}_{\mathcal{N}(Y)} & \mathcal{N}(\mathsf{id}_{\overline{A}} \bullet \phi) := \rho^{-1}_A \\
\mathcal{N}(\phi^*) := \mathsf{id}_{I_{\mathscr{C}}} & \mathcal{N}(\phi^* \bullet \mathsf{id}_Y) := \lambda_{\mathcal{N}(Y)} & \mathcal{N}(\mathsf{id}_{\overline{A}} \bullet \phi^*) := \rho_A
\end{array}$$

where the object Y is assumed to be a *non-empty* list. Additionally, we have a single recursive case $q \in \{\Phi, \phi, \Phi^*, \phi^*, \mathsf{id}_{\overline{Q}}\}$

$$\mathcal{N}(\mathsf{id}_{\overline{A}} \bullet q \bullet r) := \mathsf{id}_A \otimes \mathcal{N}(q \bullet r)$$

and finally we take $\mathcal{N}(f \mathbin{⨟} g) := \mathcal{N}(f) \mathbin{⨟} \mathcal{N}(g)$.

We prove that $\mathcal{N}$ is well-defined with respect to the equations of Definition 6.5 in Appendix C.2, where we also note how $N(f)$ is the same regardless of which 'sequential normal form' decomposition is chosen for f.

Remark 6.11. The definition of $\mathcal{N}$ can be explained more intuitively in terms of programming. If each layer of the sequential normal form is thought of as a list of arrows of $\mathscr{C}$, then the definition of $\mathcal{N}$ is essentially a list recursion in there are separate cases for 1, 2, and n-element lists.

We can now show that $\mathcal{N}$ is a *monoidal* functor. To do this, it is necessary to specify the coherence maps: the natural isomorphism $\Psi_{X,Y} : \mathcal{N}(X) \otimes \mathcal{N}(Y) \to \mathcal{N}(X \bullet Y)$ and isomorphism $\psi : I_{\mathscr{C}} \to \mathcal{N}(I_{\overline{\mathscr{C}}})$ mandated by Definition 6.1.

Definition 6.12. The coherence natural isomorphism for $\mathcal{N}$ is denoted Ψ and is defined recursively as follows.

$$\Psi_{I_{\overline{\mathscr{C}}},I_{\overline{\mathscr{C}}}} := \lambda_{I_{\mathscr{C}}} = \rho_{I_{\mathscr{C}}} \qquad \Psi_{X,I_{\overline{\mathscr{C}}}} := \rho_{\mathcal{N}(X)} \qquad \Psi_{I_{\overline{\mathscr{C}}},Y} := \lambda_{\mathcal{N}(Y)}$$

$$\Psi_{\overline{A},Y} := \mathsf{id}_{A\otimes\mathcal{N}(Y)} \qquad \Psi_{\overline{A}\bullet X,Y} := \alpha^{-1}_{A,\mathcal{N}(X),\mathcal{N}(Y)} \mathbin{⨟} (\mathsf{id}_A \otimes \Psi_{X,Y})$$

Remark 6.13. Note that both $\lambda_{I_{\mathscr{C}}}$ and $\rho_{I_{\mathscr{C}}}$ have the correct type as a choice for $\Psi_{I_{\overline{\mathscr{C}}},I_{\overline{\mathscr{C}}}}$. In fact, they are equal: unitors coincide at the unit object, i.e. $\lambda_{I_{\mathscr{C}}} = \rho_{I_{\mathscr{C}}}$, as noted in [47, Corollary 2.2.5].

Definition 6.14. The coherence isomorphism ψ for $\mathcal{N}$ is defined as follows.

$$\psi := \mathsf{id}_{I_{\mathscr{C}}}$$

Proposition 6.15. *$(\mathcal{N}, \Psi, \psi)$ is a monoidal functor.*

Proof. It is clear that Ψ and ψ are natural isomorphisms since they are both composites of natural isomorphisms. Thus it remains to check the diagrams of Definition 6.1 commute.

The squares (6.2) commute because $\psi = \mathsf{id}$, $\Psi_{A,I_{\overline{\mathscr{C}}}} = \rho$ and $\Psi_{I_{\overline{\mathscr{C}}},B} = \lambda$ by definition.

Now let us check that the hexagon (6.1) commutes. Note that in the following we use that $\mathcal{N}(\alpha_{\overline{\mathscr{C}}}) = \mathsf{id}$, because $\overline{\mathscr{C}}$ is strict, and so the hexagon axiom becomes a pentagon.

We will approach the problem inductively, checking base cases where $A = I$ and $A = \overline{A}$, and finally the inductive step with $A = \overline{A} \bullet R$. Let us begin with $A = I$, and taking the outer path of the hexagon we calculate as follows:

$$\begin{aligned}
&(\mathsf{id}_{I_{\mathscr{C}}} \otimes \Psi_{B,C}) \fatsemi \Psi_{I_{\mathscr{C}},B\bullet C} \fatsemi \Psi^{-1}_{B,C} \fatsemi (\Psi_{I_{\mathscr{C}},B} \otimes \mathsf{id}_{\mathcal{N}(C)})^{-1} \\
&\qquad = (\mathsf{id}_{I_{\mathscr{C}}} \otimes \Psi_{B,C}) \fatsemi \lambda_{\mathcal{N}(B\bullet C)} \fatsemi \Psi^{-1}_{B,C} \fatsemi (\lambda_{\mathcal{N}(B)} \otimes \mathsf{id}_{\mathcal{N}(C)})^{-1} \\
&\qquad = \lambda_{\mathcal{N}(B)\otimes\mathcal{N}(C)} \fatsemi (\lambda_{\mathcal{N}(B)} \otimes \mathsf{id}_{\mathcal{N}(C)})^{-1} \\
&\qquad = \alpha_{I_{\mathscr{C}},\mathcal{N}(B),\mathcal{N}(C)}
\end{aligned}$$

Wherein we expanded the definition of Ψ, then used naturality of $\Psi_{B,C}$ before applying the monoidal triangle lemma of [47, (2.12)].

Now consider the second base case, where A is the 'singleton list' $\overline{A}$. In this case, the hexagon diagram commutes immediately because $\Psi_{\overline{A},B} = \mathsf{id}_{\overline{A}\otimes\mathcal{N}(B)}$ and $\Psi_{\overline{A},B\bullet C} = \mathsf{id}_{\overline{A}\otimes\mathcal{N}(B\bullet C)}$. More explicitly, we calculate as follows, starting again with the outer path of the hexagon and expanding definitions:

$$\begin{aligned}
&(\mathsf{id}_A \otimes \Psi_{B,C}) \fatsemi \Psi_{A,B\bullet C} \fatsemi \Psi^{-1}_{A\bullet B,C} \fatsemi (\Psi_{\overline{A},B} \otimes \mathsf{id}_{\mathcal{N}(C)}) \\
&= (\mathsf{id}_A \otimes \Psi_{B,C}) \fatsemi \mathsf{id}_{A\otimes\mathcal{N}(B\bullet C)} \fatsemi (\alpha^{-1}_{A,\mathcal{N}(B),\mathcal{N}(C)} \fatsemi (\mathsf{id}_A \otimes \Psi_{B,C}))^{-1} \fatsemi (\mathsf{id}_{A\otimes\mathcal{N}(B)} \otimes \mathsf{id}_{\mathcal{N}(C)}) \\
&= (\mathsf{id}_A \otimes \Psi_{B,C}) \fatsemi (\mathsf{id}_A \otimes \Psi_{B,C})^{-1} \fatsemi \alpha_{A,\mathcal{N}(B),\mathcal{N}(C)} \\
&= \alpha_{A,\mathcal{N}(B),\mathcal{N}(C)}
\end{aligned}$$

Finally let us prove the inductive step. Assume that the hexagon commutes for objects R, B, C, giving us the equality

$$\Psi_{R,B\bullet C} \fatsemi \Psi^{-1}_{R\bullet B,C} = (\mathsf{id}_{\mathcal{N}(R)} \otimes \Psi^{-1}_{B,C}) \fatsemi \alpha_{\mathcal{N}(R),\mathcal{N}(B),\mathcal{N}(C)} \fatsemi (\Psi_{R,B} \otimes \mathsf{id}_{\mathcal{N}(C)})$$

We may then rewrite the following subterm of the monoidal hexagon as follows:

$$\mathsf{id}_A \otimes (\Psi_{R,B\bullet C} \fatsemi \Psi^{-1}_{R\bullet B,C}) = \mathsf{id}_A \otimes (\mathsf{id}_{\mathcal{N}(R)} \otimes \Psi^{-1}_{B,C}) \fatsemi \mathsf{id}_A \otimes \alpha_{\mathcal{N}(R),\mathcal{N}(B),\mathcal{N}(C)} \fatsemi \mathsf{id}_A \otimes (\Psi_{R,B} \otimes \mathsf{id}_{\mathcal{N}(C)})$$

We can then rewrite $\mathsf{id}_A \otimes \alpha_{\mathcal{N}(R),\mathcal{N}(B)\mathcal{N}(C)}$ using the monoidal category pentagon axiom, and then use naturality of α to reduce the outer path of the monoidal hexagon until we are left with $\alpha_{A\otimes\mathcal{N}(R),\mathcal{N}(B),\mathcal{N}(C)}$, as required. □

6.6 The Strictness Theorem

Finally, we must check that $\mathcal{S}$ and $\mathcal{N}$ indeed form a monoidal equivalence. This will amount to a proof of Mac Lane's strictness theorem which we state formally in Theorem 6.22. Explicitly, we require the following two diagrams to commute,

$$\begin{tikzcd} A \arrow[r, "\eta_A"] \arrow[d, "f"'] & \mathcal{N}(\mathcal{S}(A)) \arrow[d, "\mathcal{N}(\mathcal{S}(f))"] \\ B \arrow[r, "\eta_B"'] & \mathcal{N}(\mathcal{S}(B)) \end{tikzcd} \quad (6.9) \qquad \begin{tikzcd} \mathcal{S}(\mathcal{N}(A)) \arrow[r, "\epsilon_A"] \arrow[d, "\mathcal{S}(\mathcal{N}(f))"'] & A \arrow[d, "f"] \\ \mathcal{S}(\mathcal{N}(B)) \arrow[r, "\epsilon_B"'] & B \end{tikzcd} \quad (6.10)$$

and further require that both η and ϵ are *monoidal* natural isomorphisms (Definition 6.3.) We begin by showing that η is a monoidal natural transformation.

Proposition 6.16. $\mathcal{N} \circ \mathcal{S} = \mathsf{id}_{\mathscr{C}}$.

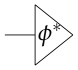
Proof. $\mathcal{N}(\mathcal{S}(f)) = \mathcal{N}(\overline{f}) = f = \mathsf{id}_{\overline{\mathscr{C}}}(f)$ □

Remark 6.17. Note that Proposition 6.16 shows that the composite $\mathcal{N} \circ \mathcal{S}$ is actually *equal* to the identity functor, and thus $\eta_A = \mathsf{id}_A$. This guarantees that η is a monoidal natural isomorphism. In addition, this will make the composite of the two functors a *split idempotent*.

Now we prove (monoidal) naturality of ϵ. This proof is somewhat more involved. Unlike Proposition 6.16, the composite $\mathcal{S} \circ \mathcal{N}$ is merely isomorphic to the identity functor, not equal on the nose. We therefore begin with an inductive definition of ϵ.

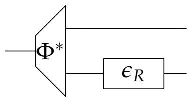
Definition 6.18 ($\epsilon : \mathcal{S} \circ \mathcal{N} \to \mathsf{id}_{\overline{\mathscr{C}}}$)**.** We define the monoidal natural isomorphism $\epsilon : \mathcal{S} \circ \mathcal{N} \to \mathsf{id}_{\overline{\mathscr{C}}}$ for the composite $\mathcal{S} \circ \mathcal{N}$ inductively:

$$\epsilon_{I_{\overline{\mathscr{C}}}} := \phi^* \qquad =$$



$$\epsilon_{\overline{A}} := \mathsf{id}_{\overline{A}} \qquad =$$

$$\epsilon_{\overline{A} \bullet R} := \Phi^* \fatsemi (\mathsf{id}_{\overline{A}} \bullet \epsilon_R) \qquad =$$



Let us first show that ϵ is a natural isomorphism. We will proceed by induction, so we begin by showing the inductive case as a useful lemma.

Proposition 6.19. *If ϵ is natural for f and g, then it is natural for $f \fatsemi g$.*

Proof. Take morphisms $f : X \to Y$ and $g : Y \to Z$. By assumption, we have:

$$\mathcal{S}(\mathcal{N}(f)) = \epsilon_X \fatsemi f \fatsemi \epsilon_Y^{-1} \qquad\qquad \mathcal{S}(\mathcal{N}(g)) = \epsilon_Y \fatsemi g \fatsemi \epsilon_Z^{-1}$$

from which we can derive

$$\begin{aligned} \epsilon_X^{-1} \fatsemi \mathcal{S}(\mathcal{N}(fg)) \fatsemi \epsilon_Z &= \epsilon_X^{-1} \fatsemi \mathcal{S}(\mathcal{N}(f) \fatsemi \mathcal{N}(g)) \fatsemi \epsilon_Z \\ &= \epsilon_X^{-1} \fatsemi \mathcal{S}(\mathcal{N}(f)) \fatsemi \mathcal{S}(\mathcal{N}(g)) \fatsemi \epsilon_Z \\ &= \epsilon_X^{-1} \fatsemi \epsilon_X \fatsemi f \fatsemi \epsilon_Y^{-1} \fatsemi \epsilon_Y \fatsemi g \fatsemi \epsilon_Z^{-1} \fatsemi \epsilon_Z \\ &= f \fatsemi g \end{aligned} \tag{6.11}$$

as required. □

Now we may prove that ϵ is a natural isomorphism.

Proposition 6.20. *$\epsilon : \mathcal{S} \circ \mathcal{N} \to \mathsf{id}_{\overline{\mathscr{C}}}$ is a natural isomorphism.*

Proof. We must show that ϵ is an isomorphism and that diagram (6.10) commutes for all objects in $\mathscr{C}$. The former can be seen by a straightforward induction: observe that ϵ is constructed as a composition of isomorphisms, and so is itself an isomorphism.

It remains to show that diagram (6.10) commutes. Let $f : A \to B$ be a morphism in $\overline{\mathscr{C}}$. By Proposition C.1 we may decompose f into layers $f = t_1 \fatsemi \ldots \fatsemi t_n$ with each layer t_i of the form $\mathsf{id}_X \bullet g \bullet \mathsf{id}_Y$ for some generator g. Proposition 6.19 ensures that if ϵ is natural for two such layers, then ϵ is also natural for their composite. We therefore need only check that ϵ is natural for all such layers.

Let $t = \mathsf{id}_X \bullet g \bullet \mathsf{id}_Y$ be a morphism with g a generator. One can verify naturality of ϵ by a second induction. More completely, one can check graphically that naturality holds for each of the base cases and inductive step in the definition of $\mathcal{N}$ (Definition 6.10). □

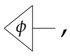
In order to complete the proof that $\mathcal{S}$ and $\mathcal{N}$ form an equivalence, we must now show that ϵ is a *monoidal* natural isomorphism.

Proposition 6.21. *$\epsilon : \mathcal{S} \circ \mathcal{N} \to \mathsf{id}_{\overline{\mathscr{C}}}$ is a monoidal natural isomorphism.*

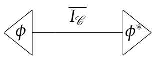
Proof. We must check that diagrams (6.3) and (6.4) commute. The composition of monoidal functors $\mathcal{S} \circ \mathcal{N}$ is a monoidal functor [1] with the following coherence map Γ and isomorphism γ.

$$\Gamma_{X,Y} = (\mathcal{S} \circ \mathcal{N})(X) \bullet (\mathcal{S} \circ \mathcal{N})(Y) \overset{\Phi_{\mathcal{N}(X),\mathcal{N}(Y)}}{\longrightarrow} \mathcal{S}(\mathcal{N}(X) \otimes \mathcal{N}(Y)) \overset{\mathcal{S}(\Psi_{X,Y})}{\longrightarrow} (\mathcal{S} \circ \mathcal{N})(X \bullet Y)$$

$$\gamma = I_{\overline{\mathscr{C}}} \overset{\phi}{\longrightarrow} \mathcal{S}(\mathcal{N}(I_{\overline{\mathscr{C}}})) \overset{\mathcal{S}(\psi)}{\longrightarrow} (\mathcal{S} \circ \mathcal{N})(I_{\overline{\mathscr{C}}})$$

We begin with the latter case, showing the diagram (6.4) commutes, i.e., that $\gamma \mathbin{⨟} \epsilon_{I_{\overline{\mathscr{C}}}} = \mathsf{id}_{I_{\overline{\mathscr{C}}}}$. Observe that $\mathcal{S}(\mathcal{N}(I_{\mathscr{C}})) = \mathcal{S}(I_{\mathscr{C}}) = \overline{I_{\mathscr{C}}}$ and that $\mathcal{S}(\psi) = \mathcal{S}(\mathsf{id}_{I_{\mathscr{C}}}) = \mathsf{id}_{\overline{I_{\mathscr{C}}}}$. Thus, $\gamma = \phi =$ 

, and so

$\psi \mathbin{⨟} \epsilon_{I_{\overline{\mathscr{C}}}} =$ 

$=$ $= \mathsf{id}_{I_{\overline{\mathscr{C}}}}$

as required, so diagram (6.4) commutes.

By similar graphical reasoning, one can also verify inductively that diagram (6.3) commutes. More precisely, one must check that $\Gamma_{X,Y} \mathbin{⨟} \epsilon_{X \bullet Y} = \epsilon_X \bullet \epsilon_Y$ for each of the base cases $\Gamma_{I_{\overline{\mathscr{C}}},Y}$, $\Gamma_{X,I_{\overline{\mathscr{C}}}}$, $\Gamma_{\overline{A},Y}$ and the recursive case $\Gamma_{\overline{A}\bullet X,Y}$. □

We can now give the main result.

Theorem 6.22 (Mac Lane's Strictness Theorem)**.** *Let $\mathscr{C}$ be an arbitrary monoidal category. Then $\mathscr{C}$ is monoidally equivalent to $\overline{\mathscr{C}}$ with the monoidal functors $\mathcal{S}$ and $\mathcal{N}$.*

Proof. $\mathcal{S}$ and $\mathcal{N}$ are monoidal functors by Propositions 6.8 and 6.15, and they form a monoidal equivalence with η and ϵ by Propositions 6.16 and 6.21. Since $\mathscr{C}$ was arbitrary, the proof is complete. □

Note that in contrast to Mac Lane's proof of Theorem 6.22, we make no reference to the coherence theorem. We can therefore make use of the strictness theorem to prove coherence, which is the subject of Section 6.7.

6.7 The Coherence Theorem

We can now use the strictness theorem to give a proof of Mac Lane's *coherence theorem*. Mac Lane's original statement of the coherence theorem is given in two parts. The first part [80, Theorem 1 (p. 166)] defines a preorder $\mathscr{W}$, which is then shown to be free in a certain sense. However, the more recognisable part of the coherence theorem is in fact its corollary [80, p. 169], which is a statement about 'formal diagrams' formed from associators and unitors.

The bulk of the proof is for Mac Lane's Theorem 1. This section is therefore dedicated to giving a novel graphical proof of this theorem. We relegate discussion of Mac Lane's proof of the corollary to Appendix C.3.

Mac Lane's proof of the main theorem begins by defining a certain preorder $\mathscr{W}$ with a single generating object W. $\mathscr{W}$ is then shown to enjoy the following property.

Theorem 6.23. *(Mac Lane's Coherence Theorem [80, p. 166])*
Let $\mathscr{M}$ be an arbitrary monoidal category, and let M be an object of $\mathscr{M}$. Then there is a unique strict monoidal functor $\mathscr{W} \to \mathscr{M}$ such that $W \mapsto M$.

In contrast, we will define the category $\mathscr{W}$ so this unique functor is easy to construct, and then use $\overline{\mathscr{W}}$ to give a *graphical proof* that $\mathscr{W}$ is a preorder. Note that the monoidal functor in question is *strict*, so its coherence maps are identities.

6.7.1 The free monoidal category on one generator

We begin by defining $\mathscr{W}$. Again, we stress that our definition of $\mathscr{W}$ differs from that of Mac Lane. In particular, we define $\mathscr{W}$ to be 'free', and then prove it is a preorder. In contrast, Mac Lane defines $\mathscr{W}$ as a preorder, and then proves its freeness.

Definition 6.24. We define $\mathscr{W}$ as the monoidal category freely generated by a single object W and no morphisms except those required by the definition of a monoidal category. [1]

Remark 6.25. The objects of $\mathscr{W}$ are $I_{\mathscr{W}}$, W, and their tensor products. The arrows are $\mathsf{id}, \rho, \lambda, \alpha$ and their composites and tensor products.

It is now clear that the statement of Mac Lane's Theorem 1 holds for our definition of $\mathscr{W}$:

Proposition 6.26. *Given an arbitrary monoidal category $\mathscr{M}$ and object $M \in \mathscr{M}$, there is a unique strict monoidal functor $\mathscr{W} \to \mathscr{M}$ with $W \mapsto M$.*

[1] Mac Lane denotes the generating object as $(-)$ to suggest an "empty place". We follow the convention of Hines [66] and use W instead.

Proof. Let M be an object of $\mathscr{M}$ and suppose that $\mathsf{U} : \mathscr{W} \to \mathscr{M}$ is a strict monoidal functor such that $W \mapsto M$. Then we must have that $\mathsf{U}(W) = M$ by assumption, and

$$\mathsf{U}(I) = I \qquad \mathsf{U}(X \otimes Y) = \mathsf{U}(X) \otimes \mathsf{U}(Y)$$

on objects, and

$$\mathsf{U}(f) = f, \quad f \in \{\alpha, \lambda, \rho, \mathsf{id}\}$$

$$\mathsf{U}(f \otimes g) = \mathsf{U}(f) \otimes \mathsf{U}(g) \qquad\qquad \mathsf{U}(f \mathbin{;} g) = \mathsf{U}(f) \mathbin{;} \mathsf{U}(g)$$

on morphisms, because U is a strict monoidal functor. But this accounts for all objects and morphisms of $\mathscr{W}$, and so U must be unique. □

In order for this to constitute a proof of the coherence theorem we must now prove that $\mathscr{W}$ is a preorder. Our approach will use the following three lemmas.

1. For any monoidal category $\mathscr{C}$, If $\overline{\mathscr{C}}$ is a preorder, then so is $\mathscr{C}$.
2. $\overline{\mathscr{W}}$ is generated solely by adapters $\{\Phi, \phi, \Phi^*, \phi^*\}$.
3. $\overline{\mathscr{W}}$ is a preorder.

The final result–that $\mathscr{W}$ is a preorder–then follows from the first and third lemmas. Proving the first and second lemmas is straightforward, so we address them now. The third requires more work, and is contained in Section 6.7.2.

Proposition 6.27. *If $\overline{\mathscr{C}}$ is a preorder, then so is $\mathscr{C}$.*

Proof. Let $f, g \in \mathscr{C}(A, B)$ be morphisms. Recall that $\mathcal{N} \circ \mathcal{S} = \mathsf{id}$, and so we can derive $f = \mathcal{N}(\mathcal{S}(f)) = \mathcal{N}(\mathcal{S}(g)) = g$ where we used that $\mathcal{S}(f) = \mathcal{S}(g)$ because $\overline{\mathscr{C}}$ is a preorder. □

The second lemma states that $\overline{\mathscr{W}}$ is generated by adapters.

Proposition 6.28. *$\overline{\mathscr{W}}$ is generated by $\Phi, \phi, \Phi^*, \phi^*, \mathsf{id}$ and equations of Definition 6.5.*

Proof. Arrows of $\overline{\mathscr{W}}$ are by definition either adapters Φ, ϕ, their inverses, or morphisms $\overline{f}$ for some $f \in \mathscr{W}$. But note that all such $f \in \mathscr{W}$ are either $\mathsf{id}, \rho, \lambda, \alpha$ or their composites. It is clear that each of λ, ρ, α can each be written as adapters by equations (6.8), so it remains to show that composites of such morphisms can also be written this way.

That is, we must show that $\mathcal{S}(f \otimes g)$ can be expressed using only adapters and their composites. This can be proved inductively: if $\mathcal{S}(f), \mathcal{S}(g)$ can be expressed using adapters, then so too can compositions $\mathcal{S}(f \fatsemi g) = \mathcal{S}(f) \fatsemi \mathcal{S}(g)$ and tensors $\mathcal{S}(f \otimes g) = \Phi \fatsemi (\mathcal{S}(f) \bullet \mathcal{S}(g)) \fatsemi \Phi^*$.

Thus every morphism of $\overline{\mathscr{W}}$ can be expressed in terms of adapters, and so the category can be said to be *generated* by (only) adapters. □

6.7.2 Graphical proof that $\overline{\mathscr{W}}$ is a preorder

We will now show that $\overline{\mathscr{W}}$ is a preorder. This proof is broken into the following steps.

1. Define for each object a `size` in $\mathbb{N}$ (Definition 6.29)
2. Prove all morphisms in $\overline{\mathscr{W}}$ go between objects of the same size (Proposition 6.30)
3. Define a canonical arrow $\mathtt{can}(A, B)$ between any two objects of the same size (Definition 6.36)
4. Show that any arrow is equal to the canonical one (Proposition 6.37)

Note that we make heavy use of Proposition 6.28, which lets us reason about $\overline{\mathscr{W}}$ purely in terms of adapters and their tensors and composites.

We begin–following Mac Lane–by defining the *size* of an object (the same as Mac Lane's notion of *length* [80, p. 165]) as follows:

Definition 6.29. We define the `size` of an object as the number of occurrences of the generating object W, defined inductively:

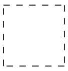
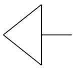
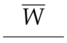
$$\mathtt{size}(I_{\overline{\mathscr{W}}}) := 0 \qquad \mathtt{size}(\overline{I_{\mathscr{W}}}) := 0 \qquad \mathtt{size}(\overline{W}) := 1$$
$$\mathtt{size}(\overline{A \otimes B}) := \mathtt{size}(A) + \mathtt{size}(B) \qquad \mathtt{size}(X \bullet Y) := \mathtt{size}(X) + \mathtt{size}(Y)$$

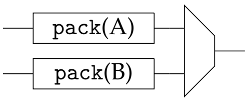
Proposition 6.30 (Morphisms in $\overline{\mathscr{W}}$ preserve `size`)**.** *If $f : A \to B$ is a morphism in $\overline{\mathscr{W}}$, then* $\mathtt{size}(A) = \mathtt{size}(B)$.

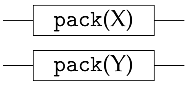
Proof. Induction on morphisms. □

We now define a canonical arrow $\mathtt{can}(A, B)$ between any two objects A and B of the same size. This definition is given as the composite of two morphisms, `pack` and `unpack`, so that $\mathtt{can}(A, B) := \mathtt{unpack}(A) \fatsemi \mathtt{pack}(B)$. We begin by defining `pack` and `unpack`.

Definition 6.31. We define the 'packing' and 'unpacking' morphisms pack and unpack in terms of objects of $\overline{\mathscr{W}}$. Let $A \in \overline{\mathscr{W}}$ be an object. Then $\mathtt{pack}(A)$ is the morphism with codomain A, defined inductively as follows.

$\mathtt{pack}(I_{\overline{\mathscr{W}}}) :=$  $\mathtt{pack}(\overline{I_{\mathscr{W}}}) :=$  $\mathtt{pack}(\overline{W}) :=$ 

$\mathtt{pack}(\overline{A \otimes B}) :=$ 

$\mathtt{pack}(X \bullet Y) :=$ 

In addition, we define $\mathtt{unpack}(A) := \mathtt{pack}(A)^{-1}$.

Remark 6.32. It can be more intuitive to define unpack first, thinking of it as the adapter which removes extraneous $I_{\mathscr{C}}$ objects and 'normalises' the object into a flat array of $\overline{W}$ objects. In this view, pack is the adapter morphism taking a fixed number of $\overline{W}$ objects and assembling them into a certain bracketing, with unit objects inserted as appropriate.

Note that Definition 6.31 implicitly uses that $\overline{\mathscr{W}}$ is a groupoid. We prove this now.

Proposition 6.33. *$\overline{\mathscr{W}}$ is a groupoid.*

Proof. Generators and identities have inverses by Definition 6.5, which allows an inductive definition for tensor and composition, i.e. $(f \mathbin{⨟} g)^{-1} = g^{-1}f^{-1}$ and $(f \bullet g)^{-1} = f^{-1} \bullet g^{-1}$ respectively. □

In order to define the canonical arrow as a composition of pack and unpack, we will need the following lemma. This states that for objects A and B of the same size the $\mathtt{unpack}(A)$ and $\mathtt{pack}(B)$ morphisms are composable.

Proposition 6.34 ($\mathtt{pack}(A) : \overline{W}^{\mathtt{size}(A)} \to A$)**.** *For an object A of size n, the domain of* $\mathtt{pack}(A)$ *is the n-fold $\bullet$-tensoring of $\overline{W}$.*

Proof. Induction on objects. Note the proposition holds for base cases $A \in \{I_{\overline{\mathscr{W}}}, \overline{I_{\mathscr{W}}}, \overline{A}\}$ and is preserved by inductive cases $\mathtt{pack}(\overline{A \otimes B})$ and $\mathtt{pack}(X \bullet Y)$. □

The same holds for unpack as an immediate corollary.

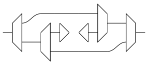
Corollary 6.35 ($\mathtt{unpack}(A) : A \to \overline{W}^{\mathtt{size}(A)}$)**.**

We can now define the canonical arrow between objects of the same size.

Definition 6.36 ($\mathsf{can}(A, B)$)**.** For each pair of objects A, B of the same size, the **canonical arrow** is defined as follows:

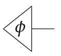
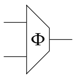
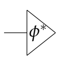
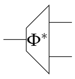
$$\mathsf{can}(A, B) : A \to B$$
$$\mathsf{can}(A, B) := \mathtt{unpack}(A) \mathbin{⨟} \mathtt{pack}(B)$$

Note that the composition of Definition 6.36 is well-typed because $\mathtt{size}(A) = \mathtt{size}(B)$ and therefore $\mathsf{cod}(\mathtt{unpack}(A)) = \overline{W}^{\mathtt{size}(A)} = \overline{W}^{\mathtt{size}(B)} = \mathsf{dom}(\mathtt{pack}(B))$.

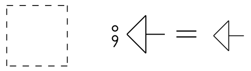
Example 6.1. *The canonical arrow between $W \otimes (I_{\mathscr{W}} \otimes W)$ and $(W \otimes I_{\mathscr{W}}) \otimes W$ is*  *. Note that this is equal to the associator $\alpha_{W,I_{\mathscr{W}},W}$.*

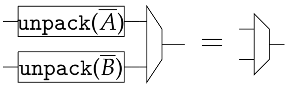
To complete the proof that $\overline{\mathscr{W}}$ is a preorder, we must show that these canonical arrows are the *only* arrows in $\overline{\mathscr{W}}$. We therefore show that every morphism $f : A \to B$ in $\overline{\mathscr{W}}$ is equal to $\mathsf{can}(A, B)$.

Proposition 6.37. $f = \mathtt{unpack}(A) \mathbin{⨾} \mathtt{pack}(B)$ *for all $f : A \to B$ in $\overline{\mathscr{W}}$.*

Proof. By Proposition 6.28 all morphisms in $\mathscr{W}$ are constructed by tensor and composition of adapters. We therefore proceed by induction, starting with the base case where we show that the proposition holds for identities and generators  and . Proofs for inverse generators  and  follow by a symmetric argument.

$$\mathsf{can}(X, X) = \mathtt{unpack}(X) \mathbin{⨾} \mathtt{pack}(X) = \mathtt{pack}(X)^{-1} \mathbin{⨾} \mathtt{pack}(X) = \mathsf{id}_X$$

$$\mathsf{can}(I_{\overline{\mathscr{W}}}, \overline{I_{\mathscr{W}}}) = \mathtt{unpack}(I_{\overline{\mathscr{W}}}) \mathbin{⨾} \mathtt{pack}(\overline{I_{\mathscr{W}}}) =$$ 

$$\mathsf{can}(\overline{A} \bullet \overline{B}, \overline{A \otimes B}) = \mathtt{unpack}(\overline{A} \bullet \overline{B}) \mathbin{⨾} \mathtt{pack}(\overline{A \otimes B}) =$$ 

Now we can prove the inductive step. The composition of canonical morphisms is canonical

$$\begin{aligned}
\mathsf{can}(X, Y) \mathbin{⨾} \mathsf{can}(Y, Z) &= \mathtt{unpack}(X) \mathbin{⨾} \mathtt{pack}(Y) \mathbin{⨾} \mathtt{unpack}(Y) \mathbin{⨾} \mathtt{pack}(Z) \\
&= \mathtt{unpack}(X) \mathbin{⨾} \mathtt{pack}(Y) \mathbin{⨾} \mathtt{pack}(Y)^{-1} \mathbin{⨾} \mathtt{pack}(Z) \\
&= \mathtt{unpack}(X) \mathbin{⨾} \mathtt{pack}(Z) \\
&= \mathsf{can}(X, Z)
\end{aligned}$$

and the tensor product of canonical morphisms is also canonical.

$$\begin{aligned}
\mathsf{can}(X_1, Y_1) \bullet \mathsf{can}(X_2, Y_2) &= \begin{array}{l} -\boxed{\mathtt{unpack}(X_1)}-\boxed{\mathtt{pack}(Y_1)}- \\ -\boxed{\mathtt{unpack}(X_2)}-\boxed{\mathtt{pack}(Y_2)}- \end{array} \\
&= -\boxed{\mathtt{unpack}(X_1 \bullet X_2)}-\boxed{\mathtt{pack}(Y_1 \bullet Y_2)}- \\
&= \mathsf{can}(X_1 \bullet X_2, Y_1 \bullet Y_2)
\end{aligned}$$

□

Proposition 6.38. *$\overline{\mathscr{W}}$ is a preorder.*

Proof. By Proposition 6.30 we know that all morphisms $f : A \to B$ have the property that $\texttt{size}(A) = \texttt{size}(B)$. We then define for any such objects a canonical morphism $\texttt{can}(A, B)$ in Definition 6.36. This canonical isomorphism is unique by Proposition 6.37, and so $\overline{\mathscr{W}}$ is a preorder. □

As an immediate corollary, we have that $\mathscr{W}$ is a preorder.

Corollary 6.39 ($\mathscr{W}$ is a preorder)**.**

Proof. Immediate by Proposition 6.27. □

Since we have proven that $\mathscr{W}$ is a preorder, we can now give an explicit proof of the coherence theorem (Theorem 6.23.) We again stress that our approach here was essentially the reverse of Mac Lane's, who defines $\mathscr{W}$ as a preorder, then shows the existence of the unique strict monoidal functor.

Proof of Theorem 6.23. $\mathscr{W}$ is a preorder by Corollary 6.39, and By Proposition 6.26 there is a unique, strict monoidal functor from $\mathscr{W}$ to an arbitrary monoidal category $\mathscr{M}$ with $W \mapsto M$ for some chosen $M \in \mathscr{M}$. □

Mac Lane's corollary then follows straightforwardly from the main coherence result (Theorem 6.23). The essential idea is to 'export' commuting diagrams from $\mathscr{W}$ to an arbitrary monoidal category by interpreting objects of $\mathscr{W}$ as functors and arrows as natural transformations. The diagrams constructed in this way are the 'formal' diagrams which are guaranteed to commute.

We provide a full exposition of Mac Lane's proof of the corollary in Appendix C.3. The following proposition will allow us to give this exposition *graphically*.

Proposition 6.40. *If $f : \overline{A} \to \overline{B}$ then $\mathcal{S}(\mathcal{N}(f)) = f$.*

Proof. We know that for any $A \in \mathscr{W}$ that $\mathcal{N}(\overline{A}) = A$. Thus for $f : \overline{A} \to \overline{B}$ we have $\mathcal{N}(f) : A \to B$ and thus $\mathcal{S}(\mathcal{N}(f)) : \overline{A} \to \overline{B}$. But $\overline{\mathscr{W}}$ is a preorder, so we have $\mathcal{S}(\mathcal{N}(f)) = f$. □

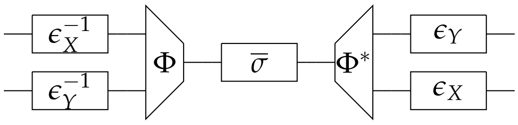
Essentially, Proposition 6.40 shows that any map of the form $f : \overline{A} \to \overline{B}$ refers unambiguously to a morphism of $\mathscr{W}$. We may therefore use the graphical language for non-strict categories to refer to morphisms of $\mathscr{W}$ directly. For a full discussion of the coherence corollary, we now refer the reader to Appendix C.3.

6.8 Symmetric Monoidal Strictness

To conclude the chapter, we briefly discuss how the strictness theorem extends to the *symmetric* monoidal case. We begin by showing how the braiding σ of $\mathscr{C}$ extends to $\overline{\mathscr{C}}$.

Definition 6.41 (Braiding of $\overline{\mathscr{C}}$)**.** Let $\mathscr{C}$ be a symmetric monoidal category. The **braiding** $\sigma_{X,Y}$ in $\overline{\mathscr{C}}$ is defined as

$$\sigma_{X,Y} :=$$

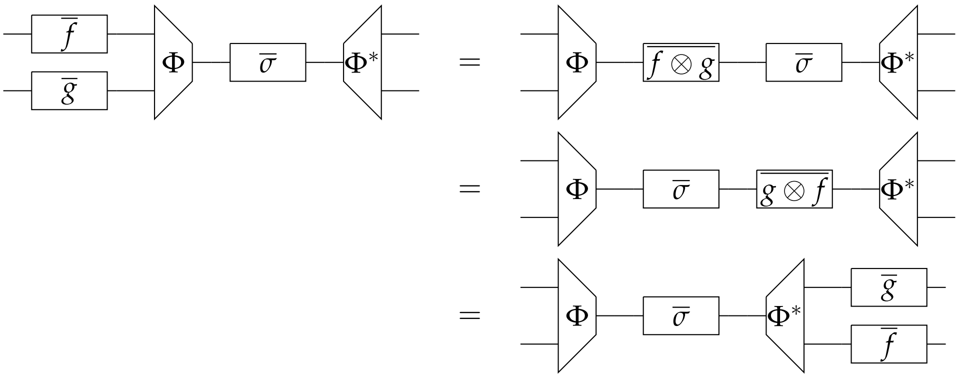


where ϵ is given in Definition 6.18.

Proposition 6.42 (Naturality of braiding)**.** *The braiding in Definition 6.41 is natural.*

Proof. We have that ϵ is natural by Proposition 6.20. The result then follows by applying naturality of ϵ, adapters, and $\overline{\sigma}$. □

Example 6.2. *When $X = \overline{A}$ and $Y = \overline{B}$ naturality follows from naturality of adapters and the braiding in $\mathscr{C}$.*



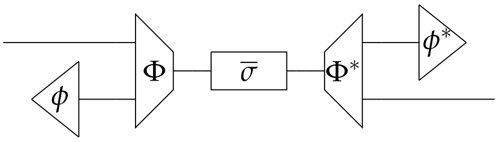
This braiding makes $\overline{\mathscr{C}}$ symmetric monoidal.

Proposition 6.43. *If $\mathscr{C}$ is symmetric monoidal then $\overline{\mathscr{C}}$ is symmetric monoidal.*

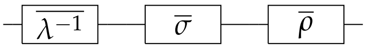
Proof. Let the braiding of $\overline{\mathscr{C}}$ be defined as in 6.41. It is natural by Proposition 6.42, and $\sigma_{X,Y} \fatsemi \sigma_{Y,X} = \mathsf{id}_{X \bullet Y}$ because adapters, ϵ, and $\overline{\sigma}$ are all isomorphisms. Finally, we must show that the unitor coherence and associator coherence axioms of symmetric monoidal categories are satisfied. The unit coherence follows straightforwardly.

Calculating for $\sigma_{X,I_{\overline{\mathscr{C}}}}$ we have

$\sigma_{X,I_{\overline{\mathscr{C}}}} =$



$=$



$=$

where the final step follows from the unitor coherence in $\mathscr{C}$. The case of $\sigma_{X,I_{\overline{\mathscr{C}}}}$ holds in essentially the same way.

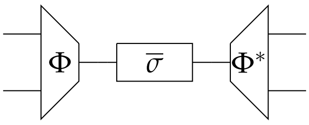
By similar calculations one may show that the associator coherence holds. Essentially, the proof follows by naturality and the associator coherence of $\mathscr{C}$. □

In addition, the monoidal functor $\mathcal{S}$ extends to a *symmetric* monoidal functor.

Proposition 6.44. *If $\mathscr{C}$ is symmetric monoidal, then $\mathcal{S} : \mathscr{C} \to \overline{\mathscr{C}}$ is a symmetric monoidal functor.*

Proof. For $\mathcal{S}$ to be symmetric monoidal, we require that $\sigma_{\mathcal{S}(A),\mathcal{S}(B)} = \Phi \,⨾\, \mathcal{S}(\sigma_{A,B}) \,⨾\, \Phi^*$. This is immediate if we simply apply $\mathcal{S}$, yielding the equality

$\sigma_{\mathcal{S}(A),\mathcal{S}(B)} \quad = \quad \sigma_{\overline{A,B}} \quad =$

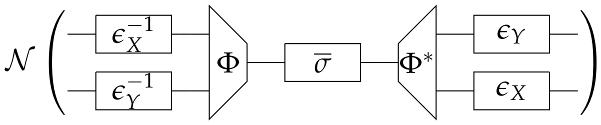


which holds by Definition 6.41. □

Finally, for $\mathscr{C}$ and $\overline{\mathscr{C}}$ to be symmetric monoidally equivalent, we must also have that $\mathcal{N}$ is a symmetric monoidal functor.

Proposition 6.45. *If $\mathscr{C}$ is symmetric monoidal, then $\mathcal{N} : \overline{\mathscr{C}} \to \mathscr{C}$ is a symmetric monoidal functor.*

Proof. One can check by straightforward induction that $\mathcal{N}(\epsilon_X \bullet \epsilon_Y) = \Psi_{X,Y}$. Having done so, the result is immediate:



$= \Psi_{X,Y}^{-1} \,⨾\, \sigma \,⨾\, \Psi_{Y,X}$

as required. □

Chapter 7

Conclusions

In this thesis, we have described a family of 'reverse differentiable' categories called polynomial circuits which are suitable for representing machine learning models. We defined learning algorithms in terms of morphisms of these categories, and gave experimental results on benchmark datasets. In addition, we defined general-purpose datastructures and algorithms able to represent morphisms of polynomial circuits, and demonstrated empirically how they can scale to large diagrams. Finally, we showed how these datastructures can be extended to a represent morphisms of a broader class of *non-strict* monoidal categories, resulting in novel proofs of Mac Lane's strictness and coherence theorems.

However, there remain several avenues for future work, which we discuss now.

7.1 Future work

7.1.1 Presentations with Frobenius Structure and Optic Composition

The first main avenue for future work is to extend the **Har** datastructure to allow for representing morphisms of those presentations of categories where the signature has the form $\Sigma + \mathbf{Frob}$. That is, where the presentation includes a chosen *special frobenius structure* (see Definition 5.7). In these cases, the conditions of monogamicity and acyclicity (Definitions 5.9 and 5.10) are relaxed, and cospans of hypergraphs correspond directly to such morphisms.

The motivation for this work derives from the desire to more efficiently take the reverse derivative of a given diagram, and more generally to compose lenses of diagrams. With the **Har** datastructure, there are two ways this can be done. The first option is to treat the primitives of a diagram as *pairs* of generators and their reverse derivatives, thus building up a pair of diagrams during composition. The second approach is to take the reverse derivative of a diagram by decomposing it into layers, and then apply the reverse derivative composition rule of Definition 3.13.

However, a recent paper by Gavranović [54] shows that these approaches can be improved. In particular, they show that by using composition of *optics* instead of lenses one obtains a diagram which represents a 'more efficient' computation in the sense that there are fewer applications of the forward map.

In order to apply this observation, we suggest combining the diagrams of Boisseau [14] with datastructures representing morphisms of categories whose signatures include a chosen Special Frobenius structure. Essentially, the idea is to first *embed* lenses into categories of optics and peform the more efficient composition there.

From an algorithmic standpoint this is a more complicated approach, sacrificing the ease-of-implementation of the **Har** datastructure. Because the condition of monogamicity must be relaxed, it is no longer possible to directly represent the underlying connectivity of a diagram with an adjacency matrix: the additional Frobenius structure allows for *multiple edges* between nodes. In other words, this means the underlying representation must be changed from a graph to a *multigraph*. In making this change, one must be careful to retain data-parallelism in order for the new approach to achieve high performance. One promising approach to implementing this idea would be to use the ACSets of Patterson et al. [84], which are defined in a 'data-parallel friendly' way.

7.1.2 Designing Model Architectures for $\mathbf{PolyCirc}_S$

Although we provided some initial exploratory experiments for the category $\mathbf{PolyCirc}_{\mathbb{Z}_2}$, there remains a large, unexplored design space for model architectures, as well as choices of the underlying semiring S itself. For instance, when $S = \mathbb{Z}_2$, we observed that the use of 'weight-tying' is likely to lose information in the reverse pass of the model. Model design is therefore much different in this setting than for $S = \mathbb{R}$. It remains to be discovered if there are general principles for learners that work well in any semiring S, or even for a specific choice of semiring.

In addition, some semirings such as finite (Galois) fields and semirings of saturating arithmetic enjoy dedicated hardware acceleration already. Another avenue is therefore to tailor models to these specific cases to exploit their efficiency benefits.

Moreover, we have so far assumed that the 'reverse maps' in such architectures are always reverse derivatives, and thus linear in their second input. An obvious question then is whether non-linear maps *not* in the image of the reverse derivative operator can also perform well as components of machine learning models.

Finally, since we have given a datastructure capable of representing very large model architectures, a promising avenue of research would be to apply ideas of *meta-learning* and automated architecture search (see e.g., [45] for a survey of such techniques.) In doing so, we may also be able to leverage the categorical literature on *rewriting* of string diagrams [18–20].

7.1.3 New Model Classes

Another promising avenue for research is investigating new model *classes*. This amounts to discovering new examples of reverse derivative categories. Of particular interest to this line of research is the recent work by Cruttwell et al. [38] which extends the notion of reverse derivative category to general *monoidal* categories–not just those where the monoidal product is cartesian.

A first candidate for new model classes might be those categories whose morphisms represent maps between *sequences* of elements. The *forward* differential structure of one such category has already been studied by Sprunger and Katsumata [97], so extending this to more efficient reverse derivative structure would be a promising first step. Other potential examples of novel model class are signal flow graphs [16, 22] and even categories modelling quantum computation such as the ZX calculus [30, 31].

Appendix A

Proofs for Chapter 3

A.1 Proofs for Theorem 3.15

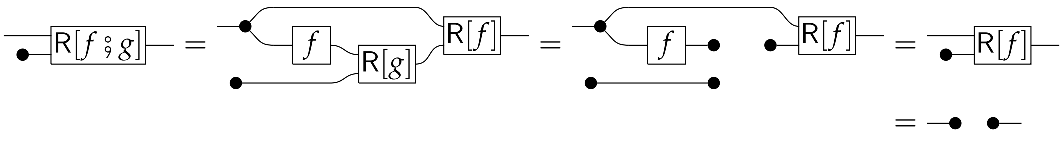
We now prove Theorem 3.15. We split the proof into the following lemmas:

1. ARD.2 is preserved by composition
2. ARD.2 is preserved by tensor product
3. ARD.3/RD.6 is preserved by composition
4. ARD.3/RD.6 is preserved by tensor product
5. ARD.4/RD.7 is preserved by composition
6. ARD.4/RD.7 is preserved by tensor product

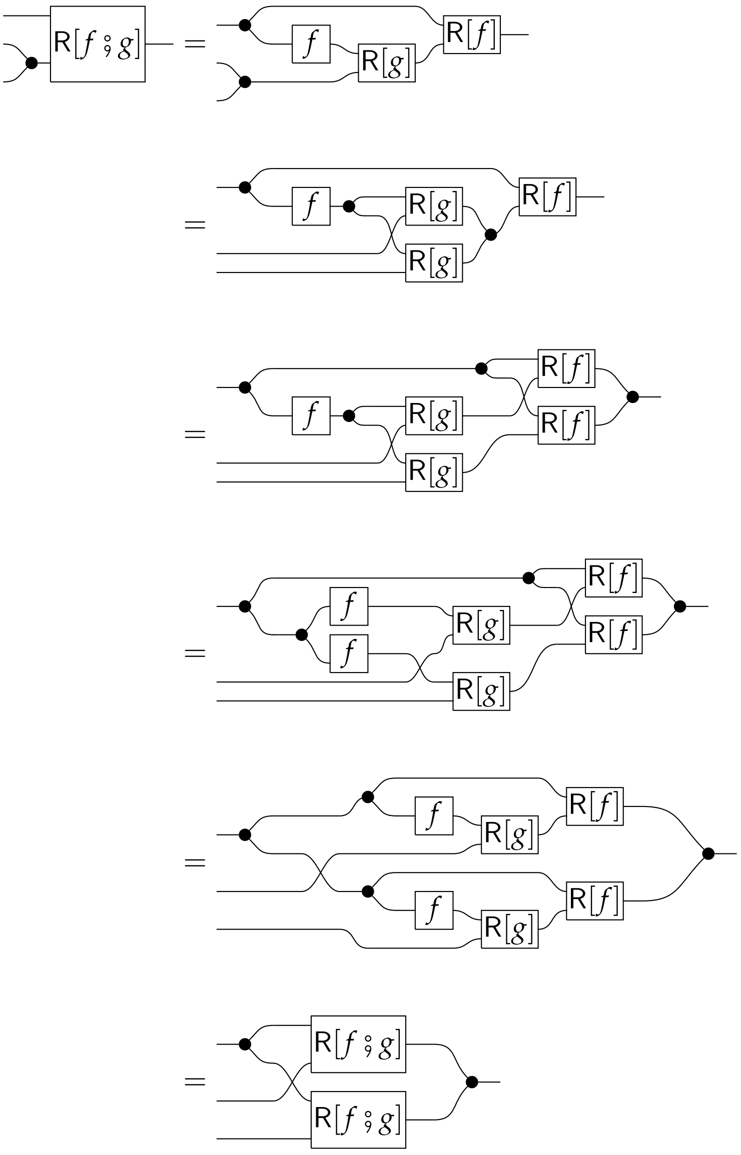
In each case, when we say 'ARD.x is preserved by composition' we mean that if f and and g satisfy ARD.x, then so too does $f \mathbin{;} g$, and likewise for tensor product. Let us now address these lemmas in order.

Lemma A.1. *ARD.2 is preserved by composition*

Proof. Assume that ARD.2 holds for $f : A \to B$ and $g : B \to C$. For the zero case, apply the chain rule and use the hypothesis twice to obtain the result as follows:



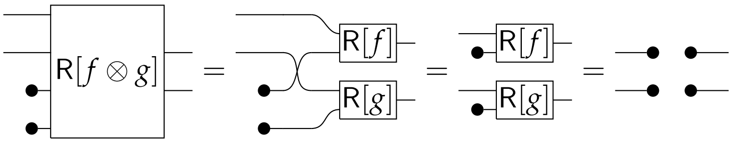
In the additive case we proceed similarly by expanding definitions, applying the hypothesis, and then using associativity and commutativity of —●< to obtain the final result:



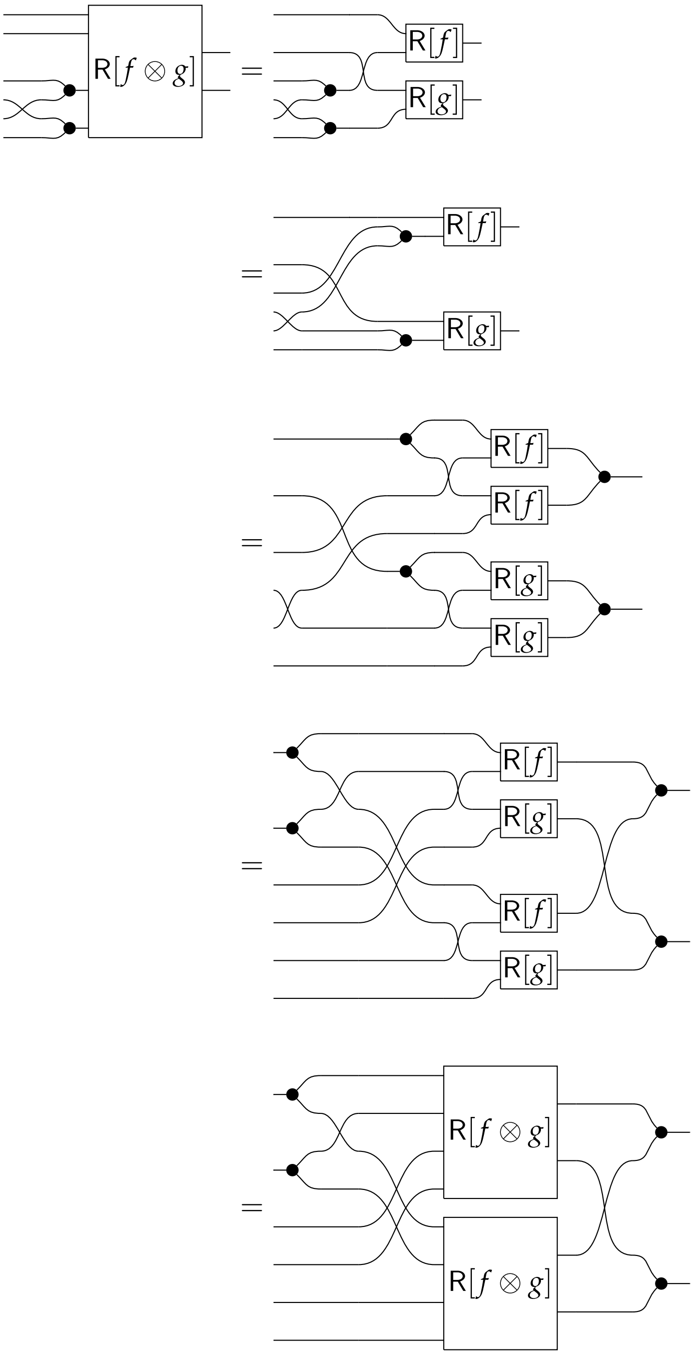
□

Lemma A.2. *ARD.2 (RD.2) is preserved by tensor product*

Proof. For the zero case,



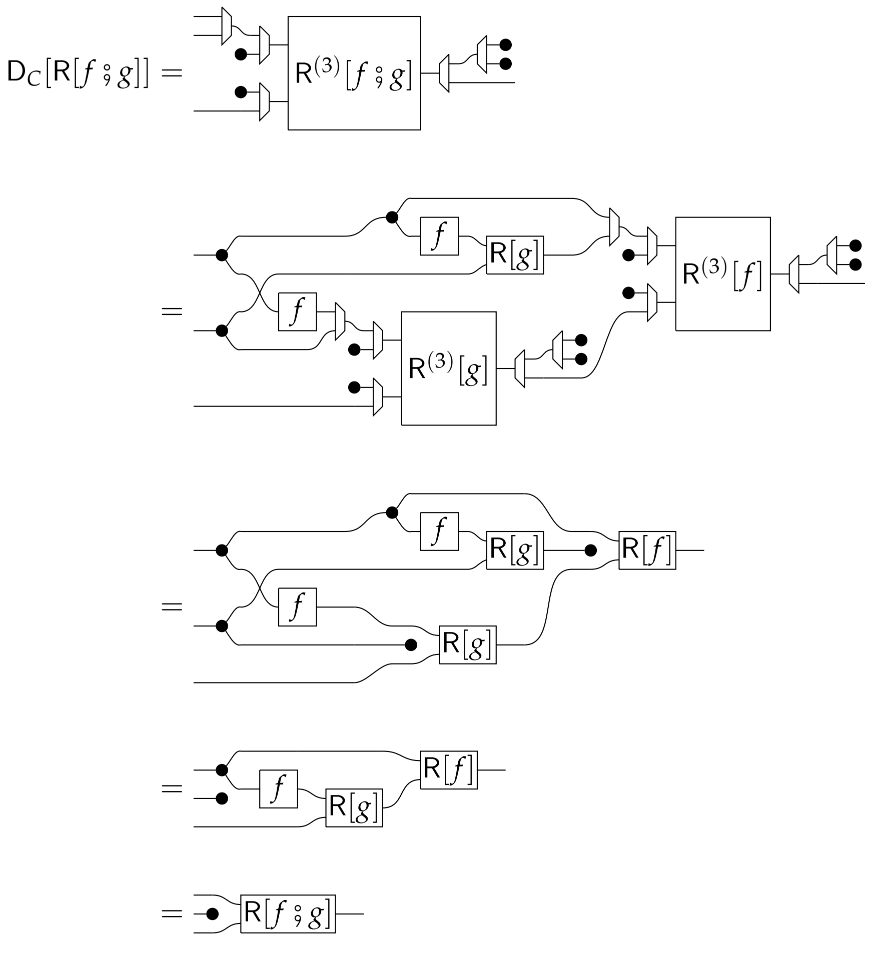
And now in the additive case,



□

Lemma A.3. *ARD.3 (RD.6) is preserved by composition*

Proof. Assume that ARD.3 holds for $f : A \to B$ and $g : B \to C$. Now calculate:

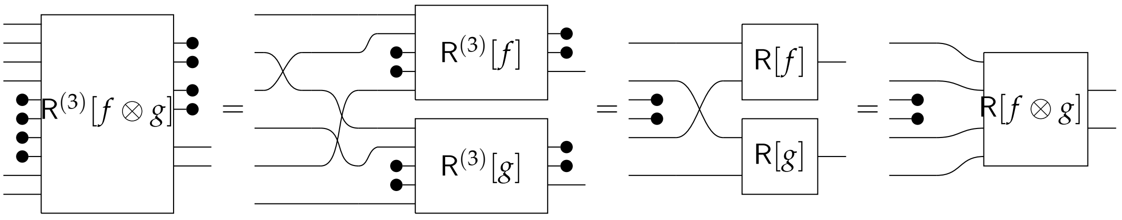


□

We omit much of the tedious calculation in the first step where $\mathsf{R}^{(3)}$ is expanded using repeated application of the chain rule. In the second step we apply the inductive hypothesis, then naturality of —• to finally obtain the result. In addition, we have made use of the notation of Chapter 6 to clarify the role of inputs to $\mathsf{R}^{(3)}$.

Lemma A.4. *ARD.3 (RD.6) is preserved by tensor product*

Proof. Assume that ARD.3 holds for f and g. Then it holds for $f \otimes g$ as follows:

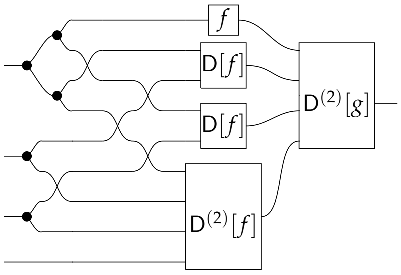


□

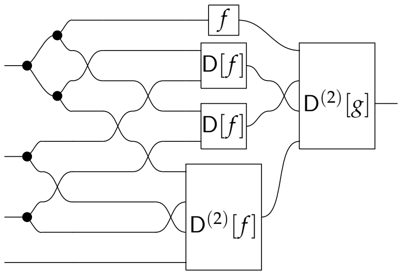
Lemma A.5. *ARD.4 (RD.7) is preserved by composition*

Proof. Assume that ARD.4 holds for $f : A \to B$ and $g : B \to C$. Now we can calculate as follows:

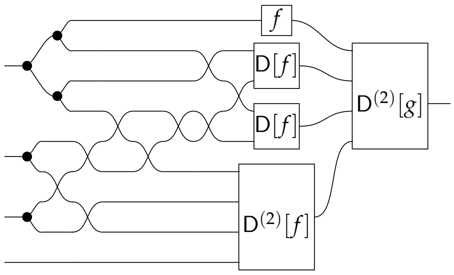
$\mathsf{D}^{(2)}[f \fatsemi g] =$



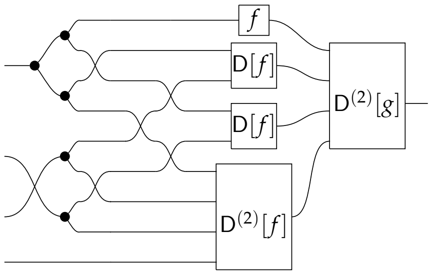
$=$



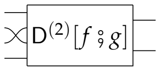
$=$



$=$



$=$

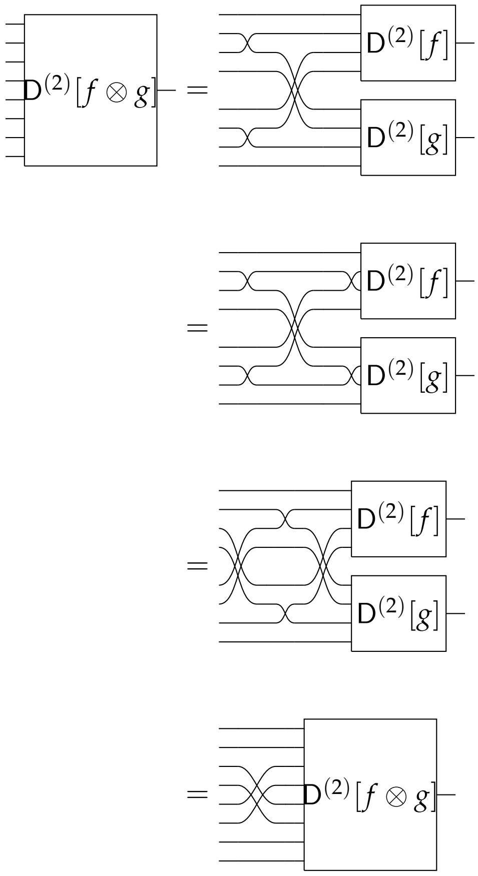


As with the proof for ARD.3 we omit a great deal of tedious expansion and calculation from the first step of the proof, which is obtained simply by expanding $\mathsf{D}^{(2)}[f \fatsemi g]$ in terms of R and using naturality of —• to simplify the result. In remaining steps, we apply of the assumption that ARD.4 holds for f and g, before finally using naturality of −•<. □

Note that although the above derivation is written in terms of D, each step of the proof treats the D operator merely as a syntactic sugar for its definition in terms of R. Each step of the proof thus uses only axioms of RDCs, rather than the forward differential structure defined in terms of it.

Lemma A.6. *ARD.4 (RD.7) is preserved by tensor product*

Proof. Assume that ARD.4 holds for $f : A_1 \to B_1$ and $g : A_2 \to B_2$. Then we may calculate as follows, first expanding the definition of D, and then using the inductive hypothesis to obtain the result:



□

It is now straightforward to prove Theorem 3.15.

Proof. (Proof of Theorem 3.15)

Suppose $\mathscr{C}$ is a category presented by generators and relations which is equipped with a (well-defined) R operator such that axioms ARD.1-4 hold for each generator, and that R is defined on tensor and composition of morphisms as in ARD.1. By the lemmas above, composition and tensor product preserve the remaining axioms ARD.2-4, and so $\mathscr{C}$ is an RDC. □

A.2 Interpretation of PolyCirc$_S$ as Polynomials

We will now show the existence of an isomorphism between **PolyCirc**$_S$ and **POLY**$_S$. The proof presented here is a minor generalisation of the author's own work in [104]. The basic idea is to show that both categories' hom-sets have the structure of the free module over the polynomial semiring. Therefore we recall the definition of a free module now.

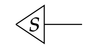
Definition A.7. Following [68, p. 170], let S be a semiring. The free module S^b is the cartesian product of b elements of S, i.e. $S^b = \langle p_1, p_2, ..., p_b \rangle$, with addition defined pointwise, $\langle p_1, p_2, ...p_b \rangle + \langle q_1, q_2, ...q_b \rangle = \langle p_1 + q_1, p_2 + q_2, ...p_b + q_b \rangle$ a zero element $\mathbf{0} = \langle 0, 0, ..., 0 \rangle$ and scalar multiplication $s\langle p_1, p_2, ..., p_b \rangle = \langle sp_1, sp_2, ...sp_b \rangle$

It is clear that the hom-sets of **POLY**$_S$ have this structure

Proposition A.8. *Hom-sets* **POLY**$_S(a, b)$ *have the structure of the free module* S^b *with* S *the polynomial semiring* $S = S[x_1, ..., x_a]$.

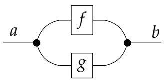
Proof. Immediate from the definition of **POLY**$_S$ □

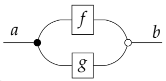
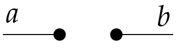
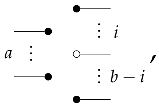
Furthermore, hom-sets of **PolyCirc**$_S$ also have this structure. This implies the existence of a module isomorphism between the hom-sets of **PolyCirc**$_S$ and **POLY**$_S$ which is the basis for the functor $[\![\cdot]\!]$. We begin, however, with some special case examples.

Example A.1. *The hom-set* **PolyCirc**$_S(0, 1)$ *has the structure of the semiring* S*, with every morphism equal to some element*  *in* S.

Example A.2. *Each hom-set* **PolyCirc**$(a, 1)$ *has the structure of the polynomial semiring* $S[x_1, ..., x_a]$*, with indeterminates* $x_1 \ldots x_a$ *given by the projections* $\pi_1 \ldots \pi_a$

Proposition A.9. *Hom-sets* **PolyCirc**(a, b) *have the structure of the free module* $S[x_1, \ldots, x_a]^b$.

Proof. For morphisms $f, g : a \to b$, define addition $f + g :=$  and multiplication

$f * g :=$ , with the zero element defined as $\mathbf{0} :=$ . Then one can verify graphically using equations of polynomial circuits (Definition 3.22) that the module axioms hold. [1] If we define the family of b morphisms $e_i :=$  $, 0 < i \leq b$, we can see that it forms a base: each of the generators of polynomial circuits (Equations (3.3) and (3.7)) can be constructed through addition and scalar multiplication of morphisms e_i and $\mathbf{0}$. □

We are now ready to give the proof of Proposition 3.27.

Proof of Proposition 3.27. By Proposition A.8 and Proposition A.9, there is a module isomorphism between $\mathbf{POLY}_S(a, b)$ and $\mathbf{PolyCirc}_S(a, b)$. Further, because the identity-on-objects functor $[\![\cdot]\!]$ (Definition 3.26) is defined in terms of this bijection, it is a full and faithful functor, and so $\mathbf{POLY}_S \cong \mathbf{PolyCirc}_S$. □

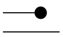
A.3 Proofs for Theorem 3.14

We now prove the lemmas used in the proof of Theorem 3.14. We begin by showing axioms RD.1-7 imply ARD.1-4, and then the converse.

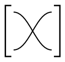
Proposition A.10. *Axioms RD.1 and RD.3-5 of [29, Definition 13] imply axiom ARD.1 of Definition 3.13.*

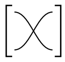
Proof. Each of the structural axioms in ARD.1 can be derived as follows.

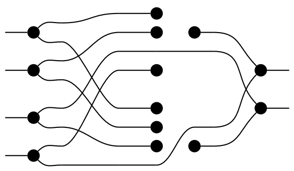
Identity $\mathsf{R}\,[—]$ is derivable from RD.3:

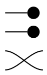
$$\mathsf{R}\,[—] = \mathsf{R}[\mathrm{id}] = \mathsf{R}[\pi_1] =$$ 

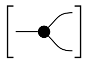
[1] We take *scalar* multiplication of $f : a \to b$ by $g : a \to 1$ as the morphism $f * (g \,⨾\, \Delta^*)$, where Δ^* is the unique $1 \to b$ morphism formed by tensor and composition of and identity.

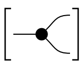
Symmetry R  is derived by applying RD.4 to $\langle \pi_1, \pi_0 \rangle$:

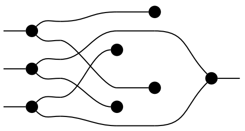
R  $= \mathsf{R}[\langle \pi_1, \pi_0 \rangle] = (\mathsf{id} \times \pi_0) \mathbin{;} \mathsf{R}[\pi_1] + (\mathsf{id} \times \pi_1) \mathbin{;} \mathsf{R}[\pi_0]$

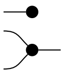
$=$ 

$=$ 

Copy R  is derived by applying RD.4 to $\langle \mathsf{id}, \mathsf{id} \rangle$:

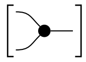
R  $= \mathsf{R}[\langle \mathsf{id}, \mathsf{id} \rangle] = (\mathsf{id} \times \pi_0) \mathbin{;} \mathsf{R}[\mathsf{id}] + (\mathsf{id} \times \pi_1) \mathbin{;} \mathsf{R}[\mathsf{id}]$

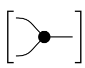
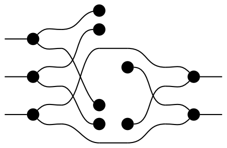
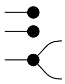
$=$ 

$=$ 

Discard R [—●] is derived from RD.4 directly:

R [—●] $= \mathsf{R}[!] = 0 =$ —● ●—

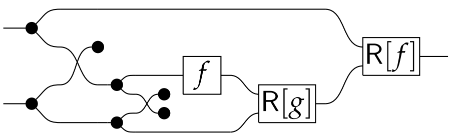
Add R  is derived by applying RD.1 to $\pi_0 + \pi_1$.

R  $= \mathsf{R}[\pi_0 + \pi_1] = \mathsf{R}[\pi_0] + \mathsf{R}[\pi_1] =$  $=$ 

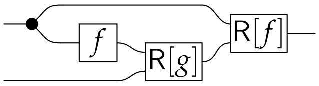
Zero R [●—] = —● follows immediately by uniqueness of the discard map.

Composition $\mathsf{R}[f \mathbin{;} g]$ follows from RD.5 by applying the counit axiom.

$$\mathsf{R}[f \mathbin{;} g] = \langle \pi_0, \langle \pi_0 \mathbin{;} f, \pi_1 \rangle\rangle \mathbin{;} \mathsf{id} \times \mathsf{R}[g] \mathbin{;} \mathsf{R}[f] =$$



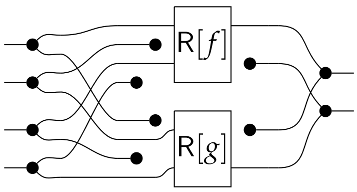
$=$



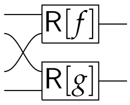
Tensor $\mathsf{R}[f \times g]$ follows from RD.4 via the counit axiom:

$$\mathsf{R}[f \times g] = \mathsf{R}[\langle \pi_0 \mathbin{;} f, \pi_1 \mathbin{;} g \rangle] = (\mathsf{id} \times \pi_0) \mathbin{;} \mathsf{R}[\pi_0 \mathbin{;} f] + (\mathsf{id} \times \pi_1) \mathbin{;} \mathsf{R}[\pi_1 \mathbin{;} g]$$

$=$

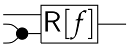
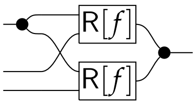
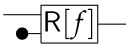
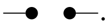


$=$

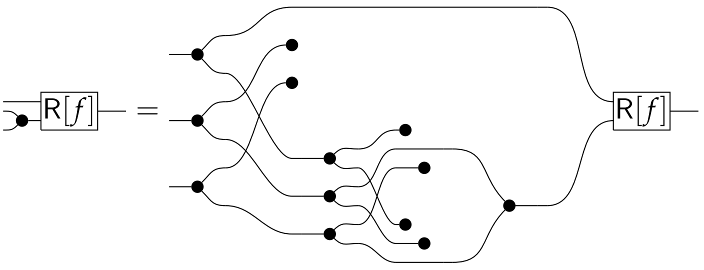


□

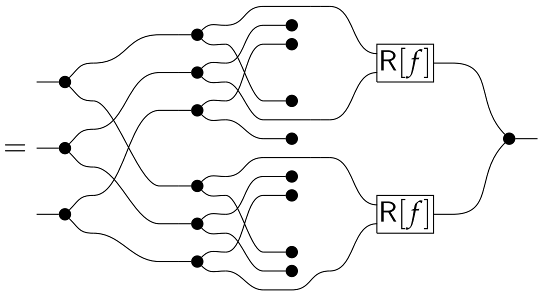
Proposition A.11. *Axiom RD.2 of [29, Definition 13] implies axiom ARD.2 of Definition 3.13*

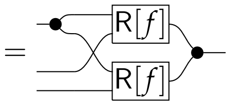
Proof. We must derive that  =  and  = .

The former follows using RD.2 with $a = \pi_0$, $b = \pi_1 \fatsemi \pi_0$ and $c = \pi_1 \fatsemi \pi_1$ as follows.

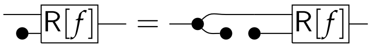


$$= \langle a, b + c\rangle \fatsemi \mathsf{R}[f]$$
$$= \langle a, b\rangle \fatsemi \mathsf{R}[f] + \langle a, c\rangle \fatsemi \mathsf{R}[f]$$

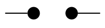




The latter follows again by RD.2 as below



$$= \langle \mathsf{id}, 0\rangle \fatsemi \mathsf{R}[f] = 0 =$$



□

Together, all the axioms RD.1-7 imply the axioms ARD.1-4 of our alternate definition of RDCs.

Proposition A.12. *Axioms RD.1-7 of [29, Definition 13] imply axioms ARD.1-4 of Definition 3.13*

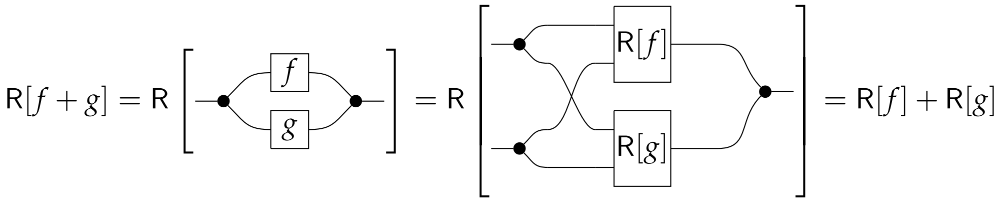
Proof. Axioms RD.1-5 imply axioms ARD.1-2 by Propositions A.10 and A.11. Moreover, ARD.3 and ARD.4 are direct statements of RD.6 and RD.7, so axioms RD.1-7 imply axioms ARD.1-4. □

We now consider the reverse direction, and show that the alternate axioms ARD.1-4 imply those of the original definition RD.1-7.

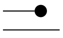
Proposition A.13. *Axiom ARD.1 of 3.13 implies axioms RD.1 and RD.3-5 of [29, Definition 13].*

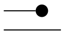
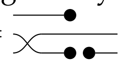
Proof. In each case, we derive each axiom by applying the inductive definition of R given in Definition 3.13 to the left-hand-side.

RD.1

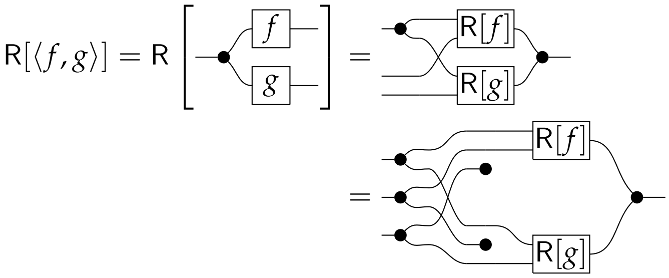


RD.3 RD.3 follows directly from ARD.1.

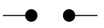
$\mathsf{R}[\mathsf{id}] = \mathsf{R}\,[$ $] =$  $= \mathsf{R}[\pi_1]$

The reverse derivative of projections is given by applying the rule for tensor products. For example, $\mathsf{R}[\pi_0] =$  $=$  $= \pi_1 \,⨟\, \iota_0$. The case for $\mathsf{R}[\pi_1]$ is derived similarly.

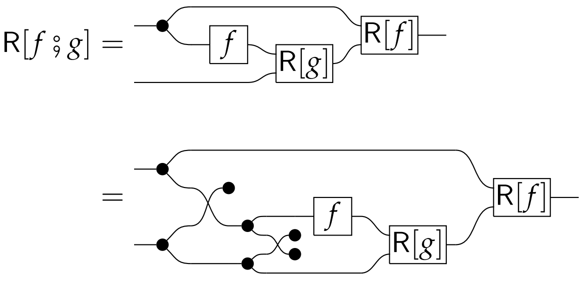
RD.4 On tuples of arrows, we can derive as follows.



$$= (\mathsf{id} \times \pi_0) \,⨟\, \mathsf{R}[f] + (\mathsf{id} \times \pi_1) \,⨟\, \mathsf{R}[g]$$

It is also straightforward that $\mathsf{R}[!] = \mathsf{R}\,[$ $] =$  $= 0$

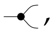
RD.5 The composition rule follows as in Proposition A.10.

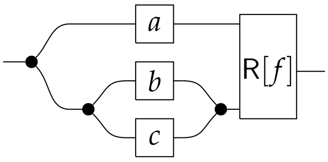


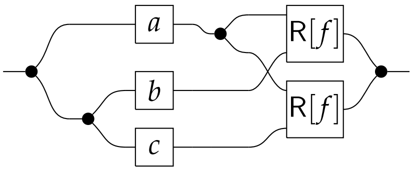
$$= \langle \pi_0, \langle \pi_0 \,⨟\, f, \pi_1 \rangle\rangle \,⨟\, \mathsf{id} \times \mathsf{R}[g] \,⨟\, \mathsf{R}[f]$$

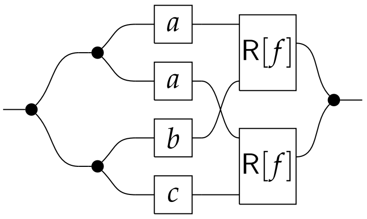
□

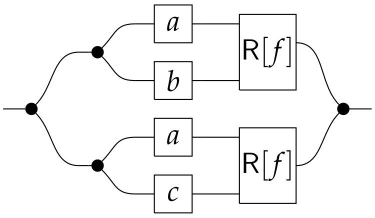
Proposition A.14. *Axiom ARD.2 of Definition 3.13 implies axiom RD.2 of [29, Definition 13].*

Proof. First, using ARD.2 and then naturality, coassociativity and cocommutativity of , derive as follows.

$\langle a, b + c\rangle \mathbin{⨾} \mathsf{R}[f] =$ 

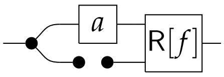
$=$ 

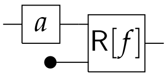
$=$ 

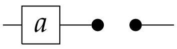
$=$ 

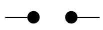
$$= \langle a, b\rangle \mathbin{⨾} \mathsf{R}[f] + \langle a, c\rangle \mathbin{⨾} \mathsf{R}[f]$$

Finally, calculate

$\langle a, 0\rangle \mathbin{⨾} \mathsf{R}[f] =$ 

$=$ 

$=$ 

$=$ 
$= 0$

to complete the proof. □

Proposition A.15. *Axioms ARD.1-4 of Definition 3.13 imply axioms RD.1-7 of [29, Definition 13].*

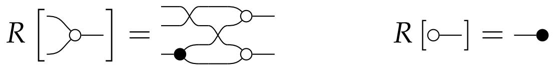
Proof. ARD.1 implies RD.1 and RD.3-5 by Proposition A.13, and ARD.2 implies RD.2 by Proposition A.14. As in A.12, ARD.3 and ARD.4 are direct statements of RD.6 and RD.7, completing the proof. □

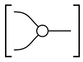
A.4 Proofs for Theorem 3.20

This section contains lemmas used in the proof of Theorem 3.20, which shows that the definition of R on cartesian distributive structure is well-defined (Proposition A.16). and satisfies axioms ARD.2-4 (Propositions A.19, A.20, and A.21, respectively). Note that several of the derivations here are easiest to understand in terms of the 'syntactic' forward differential operator D as described in Appendix A.5

A.4.1 Well-definedness of R for Cartesian Distributive Categories

Proposition A.16. *In a cartesian distributive category, the following choice of* R *is well-defined.*

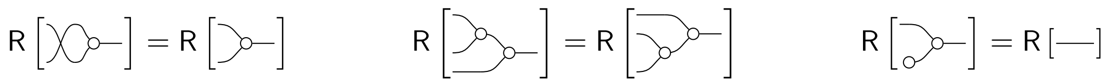


Proof. We check that R  is well-defined by ensuring that R[`lhs`] = R[`rhs`] for each of the additional equations `lhs` = `rhs` in the definition of Cartesian Distributive categories (Definition 3.18).

The commutativity, associativity, and unit equations (3.4) are checked in Proposition A.17.

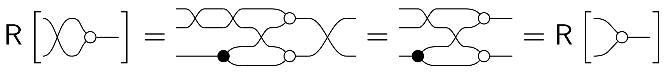
Meanwhile, the distributivity and annihilation equations (3.5) are checked in Proposition A.18, completing the proof. □

Proposition A.17. R *is well-defined for each of the equations in* (3.4)*, so we have the following equalities.*

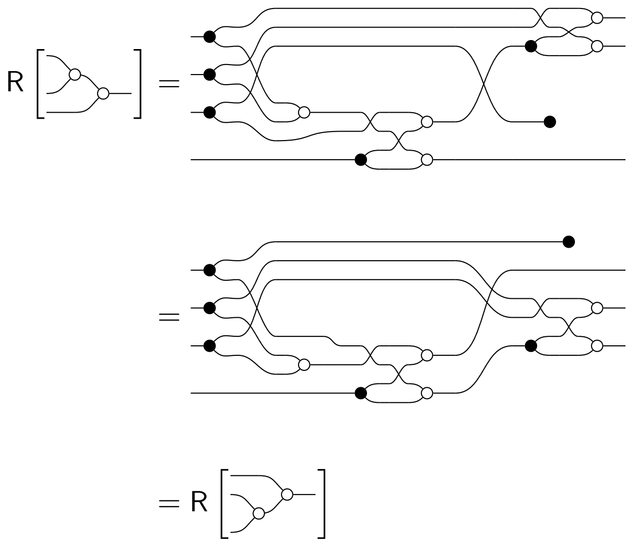


Proof. We check each case separately.

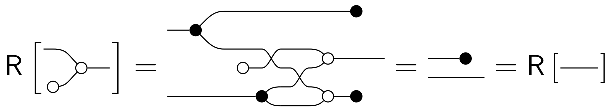
Commutative



Associative



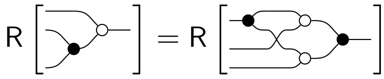
Unit

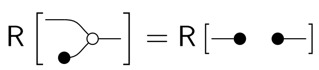


□

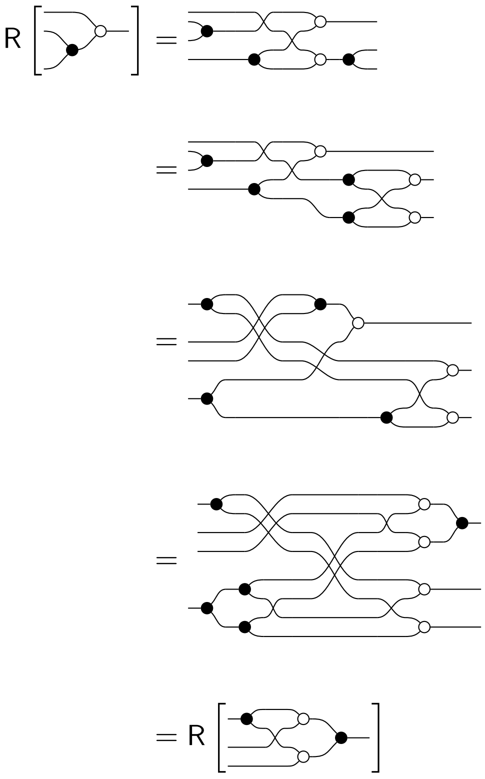
We now check the distributivity and annihilation equations.

Proposition A.18. R *is well-defined for each of the equations in* (3.5)*, so the distributivity and annihilation equations are preserved under* R*.*

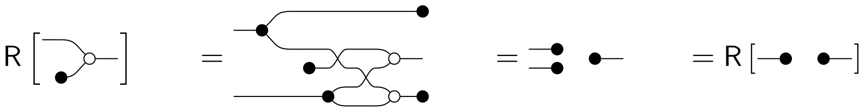




Proof. **Distributivity**

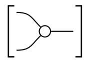


Annihilation

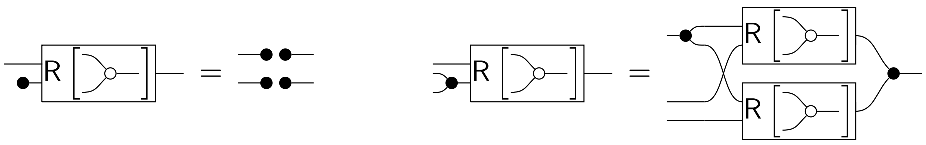


□

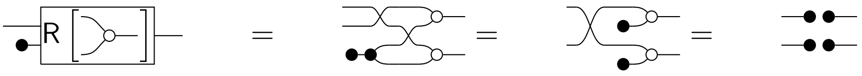
A.4.2 RDC Axioms for Cartesian Distributive Structure

We now verify the axioms of RDCs hold for R . ARD.2 holds by Proposition A.19 below.

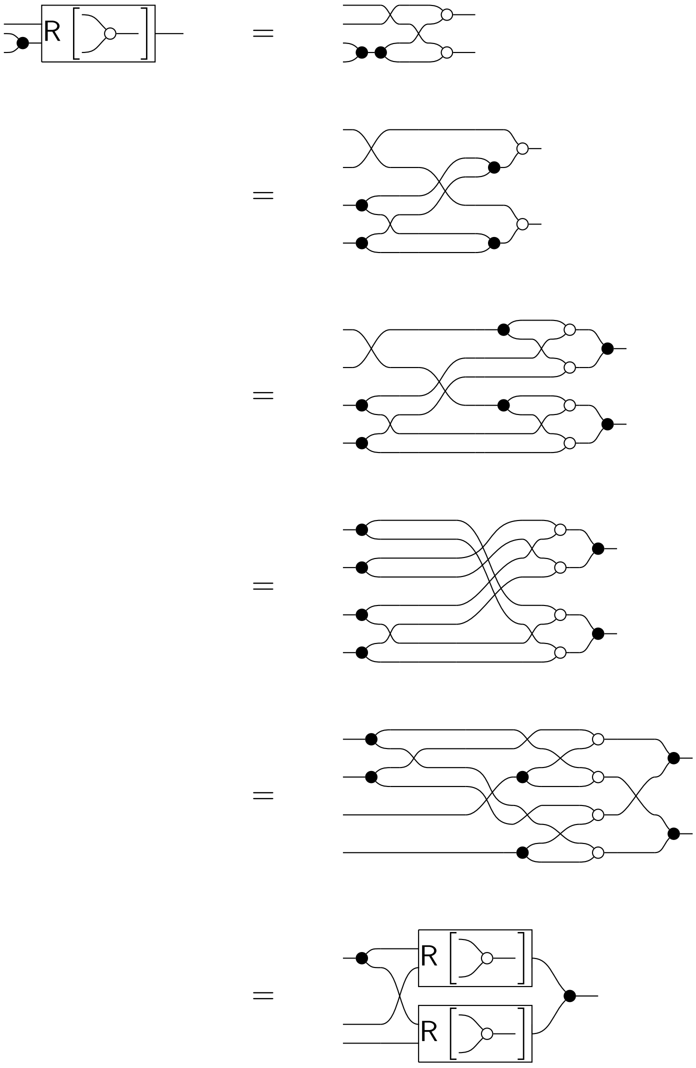
Proposition A.19.



Proof. In the former case, apply the inductive definition of R and use the distributivity axiom.

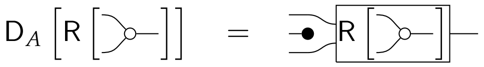


The latter case follows in a similar manner.

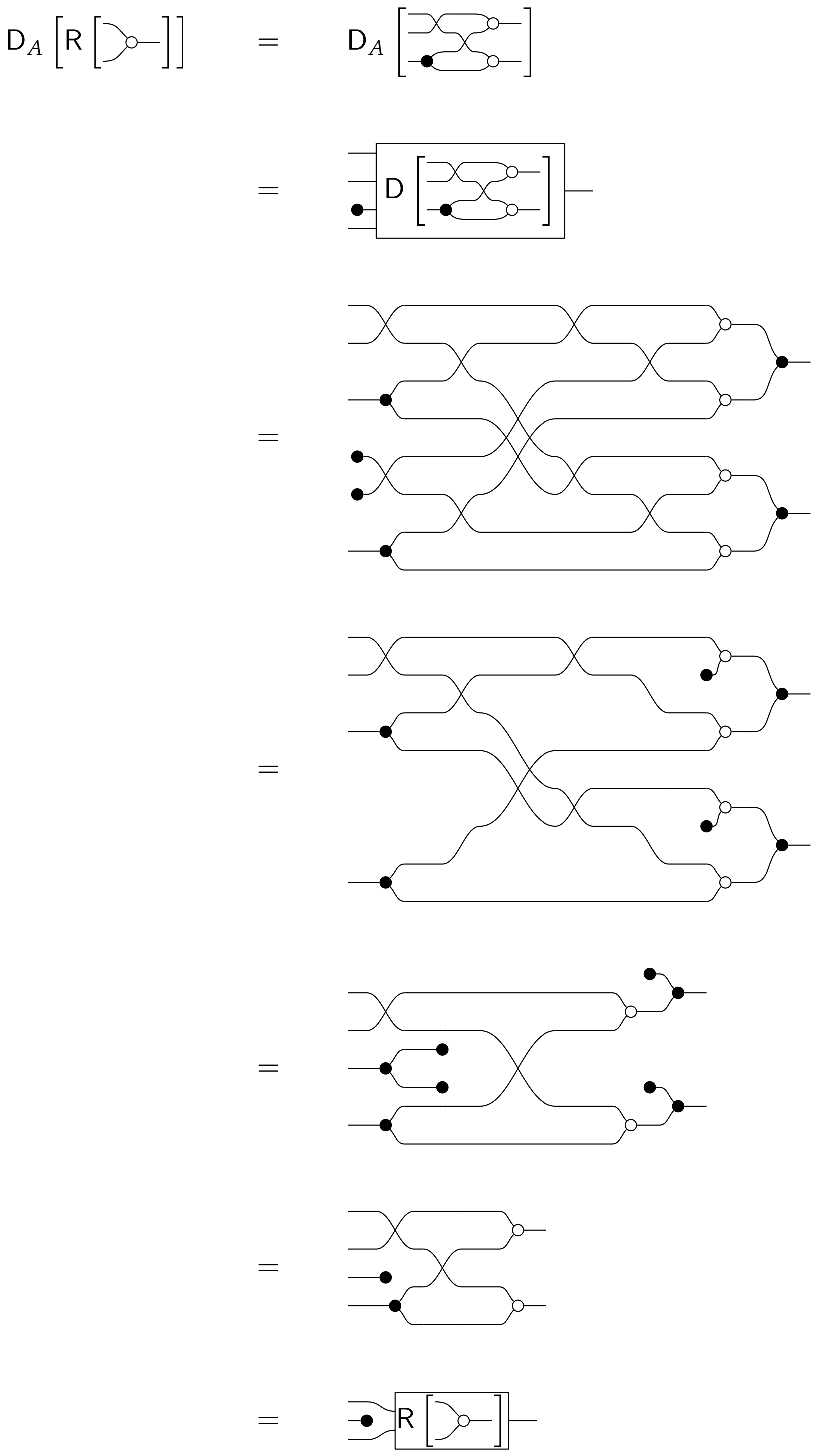


□

ARD.3 holds by Proposition A.20:

Proposition A.20. 

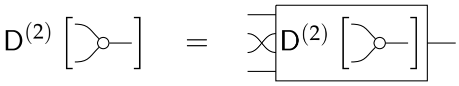
Proof.



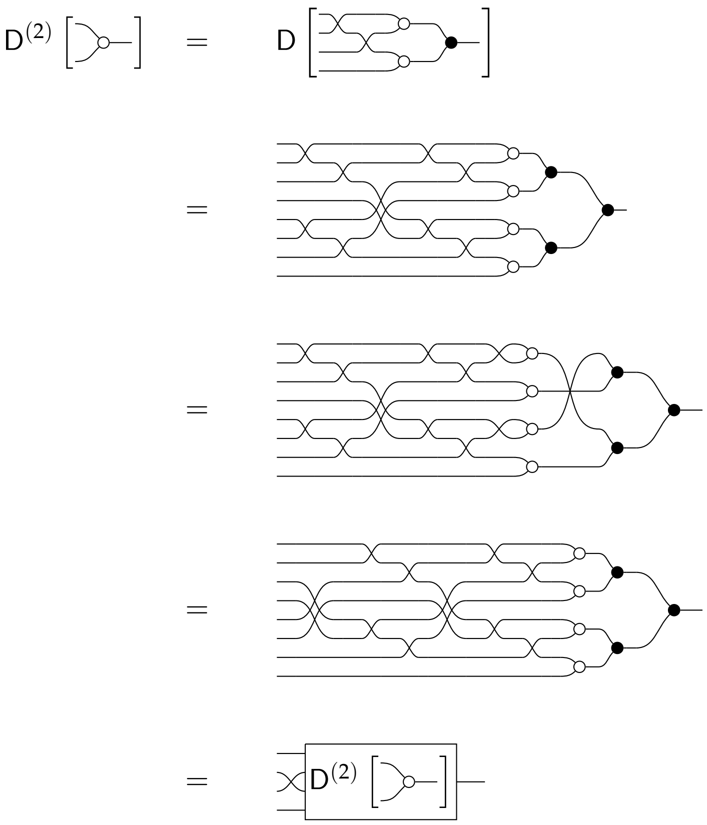
□

Finally, ARD.4 holds by Proposition A.21:

Proposition A.21.



Proof.



□

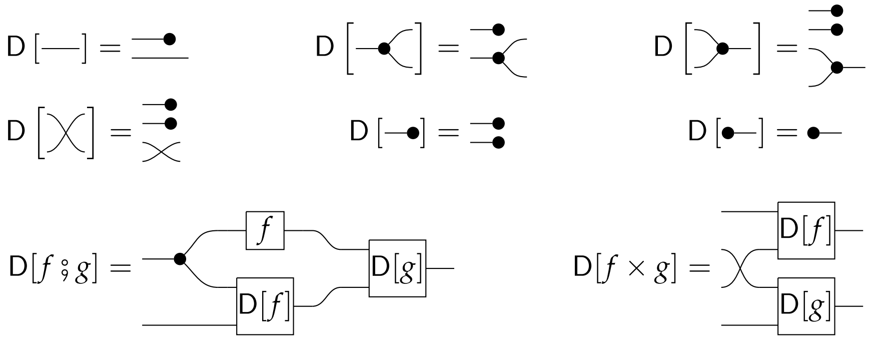
Remark A.22. Note that ARD.4 is the only axiom requiring that is commutative.

A.5 Forward Differential Operator and Linear Maps

Definition 3.9 recalls the induced forward differential operator in terms of reverse differential structure. From this, we may derive an inductive specification of D analogous to that of Definition 3.13.

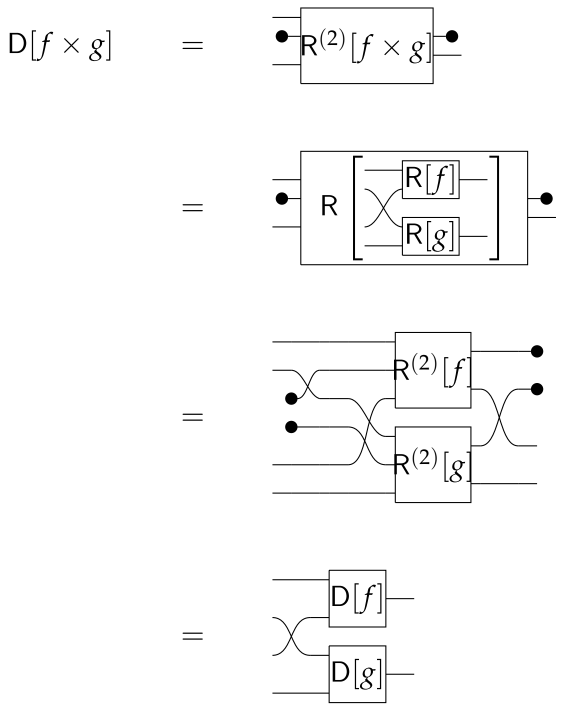
Note that the content of this appendix consists of the contributions of the author first published in [106].

Proposition A.23. *In an RDC with forward differential operator* D *defined in terms of* R *as in Definition 3.9, we have the following equalities.*



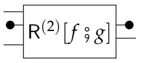
Proof. We proceed by induction. First, observe that one can calculate the value of D on each generating by applying Definition 3.9 directly. The inductive step for tensor and composition is more involved; we give the explicit derivations below.

Tensor Assume that the equality in Definition 3.9 holds for f and g, then calculate as follows.

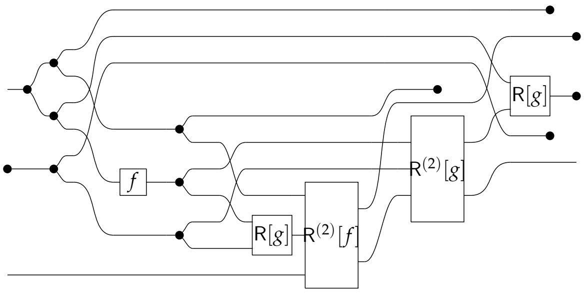


Composition Assume that the equality in Definition 3.9 holds for f and g, then calculate as below.

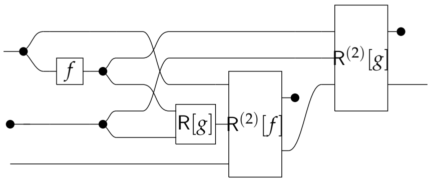
$\mathsf{D}[f \mathbin{;} g]$ =



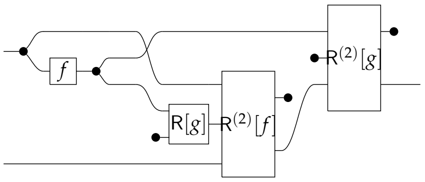
=



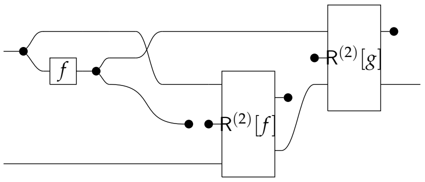
=



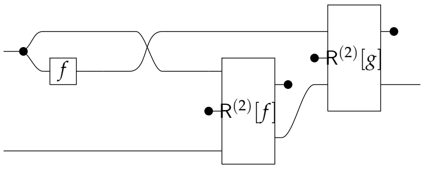
=



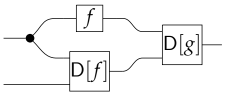
=



=



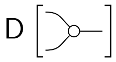
=

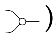
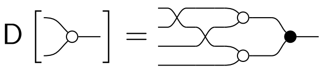


□

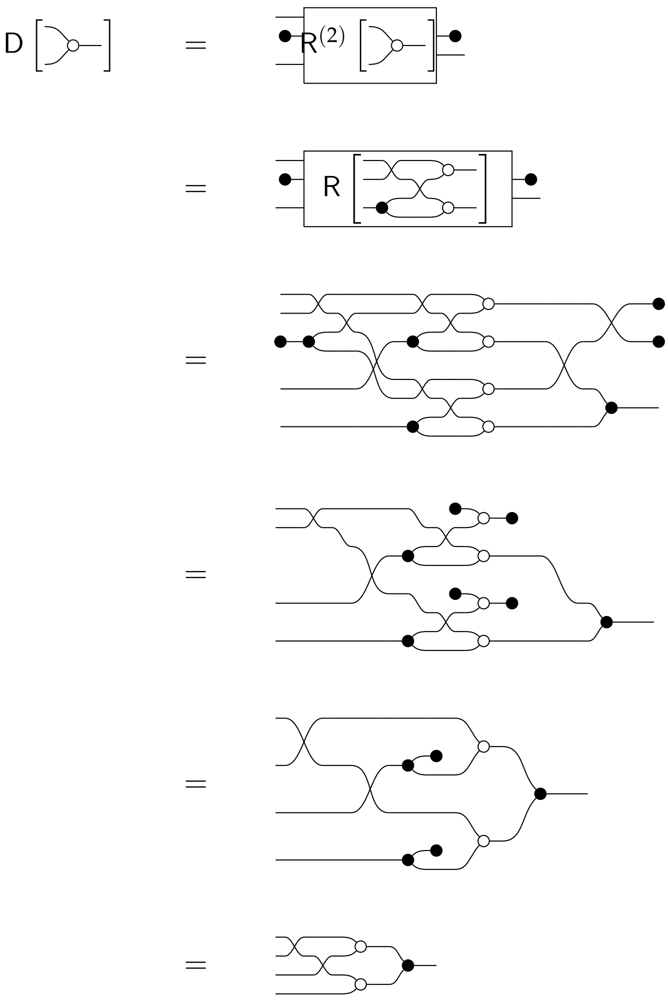
Note that Proposition A.23 is *not* a definition: it is an immediate consequence of the definition of D in terms of R given in Definition 3.9.

A.5.1 Forward Differential Operator on Cartesian Distributive Structure

We can similarly derive the values of D  and D [○—] in a Cartesian Distributive category.

Proposition A.24 (Forward Derivative on ). 

Proof.



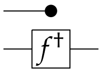
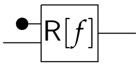
□

Proposition A.25 (Forward Derivative on ○—). D [○—] = ●—

Proof. Observe that because R [○—] = R [●—], we also have $\mathsf{R}^{(2)}$ [○—] = $\mathsf{R}^{(2)}$ [●—] and therefore D [○—] = D [●—] = ●—. □

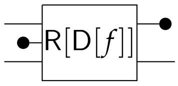
A.5.2 Daggers and Linear Sandwiches

Several of the derivations in this thesis involve taking a reverse derivative of the form $\mathsf{R}[f \fatsemi g \fatsemi h]$, where f and h are linear. The tedium of calculating such reverse derivatives can be alleviated by using the 'linear sandwich' lemma, which we will shortly prove. However, we first need the following proposition.

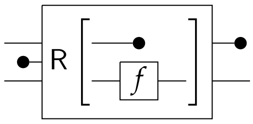
Proposition A.26. *Suppose $f : A \to B$ is a linear arrow in an RDC $\mathscr{C}$. Then* $\mathsf{R}[f] =$  *where $f^\dagger :=$*  *is the dagger defined in [29, Section 3.2].* [2]

Proof. We begin by appeal to [29, Theorems 41 and 42], which allows us to write $\mathsf{R}[f]$ in terms of the 'contextual linear dagger' $\mathsf{D}[f]^{\dagger[A]}$ ([29, Definition 34]).

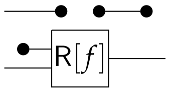
$$\mathsf{R}[f] \;=\; \mathsf{D}[f]^{\dagger[A]} \;=$$



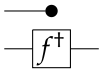
$=$



$=$



$=$

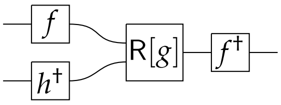


Note that the final step of the derivation uses the definition of †. □

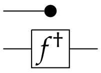
We can now state the linear sandwich lemma.

Lemma A.27 (Linear Sandwich Lemma)**.** *Let $A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$ be morphisms with f and h linear (Definition 3.12). Then*

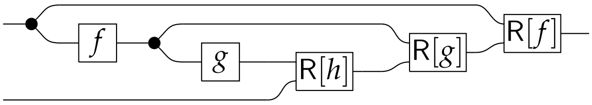
$$\mathsf{R}[f \fatsemi g \fatsemi h] \;=$$



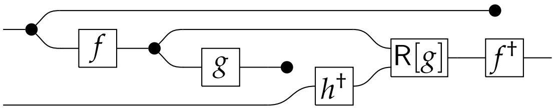
[2] see also [29, Proposition 24]

Proof. If f is linear, then $\mathsf{R}[f] =$  . We can therefore calculate as follows.

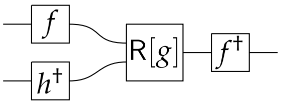
$\mathsf{R}[f \mathbin{⨟} g \mathbin{⨟} h] =$



$=$



$=$



□

Appendix B

Proofs for Chapter 5

B.1 Composition of Hars is associative up to isomorphism

We will now show associativity of composition for morphisms $m \xrightarrow{f} n \xrightarrow{g} o$ in **Har**. Note that since morphisms of **Har** are *equivalence classes* of **Har** representations, we only need show associativity up to permutation. We begin with a useful lemma.

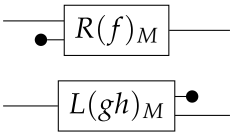
Proposition B.1. $L(f)_R = R(f)_L^T$

Proof. $L(f)_R = f_L^T \mathbin{⨟} f_R = (f_R^T \mathbin{⨟} f_L)^T = R(f)_L^T$ □

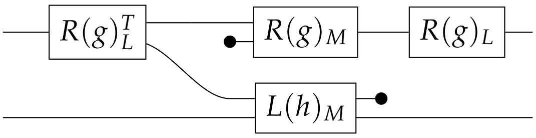
We will now prove associativity up to permutation for each of the components M, L, R, N in separate propositions, which together give the main proof. Note that we will hereafter write fg for $f \mathbin{⨟} g$ to reduce notational noise.

Proposition B.2. $(f(gh))_M \cong ((fg)h)_M$ *with the permutation* $\mathsf{id}_{f_K - n} \otimes R(g)_L^T \otimes \mathsf{id}_{h_K - o}$*:*

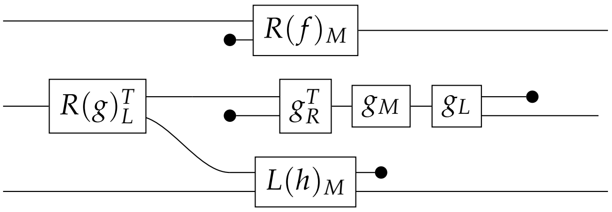
Proof. We will first simplify $(f(gh))_M$ and $((fg)h)_M$ in order to show some commonalities. We begin by unpacking definitions:

$[f(gh)]_M =$ 

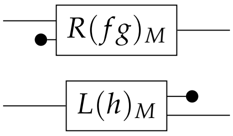
Computing $L(gh)_M$, we obtain

$L(gh)_M =$ 

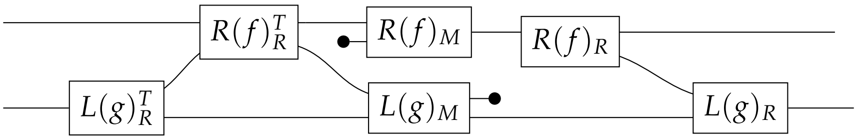
Finally, we can apply $R(g)_M R(g)_L = g_R^T g_M g_L$ to $[f(gh)]_M$, yielding the simplified diagram:

$[f(gh)]_M =$ 

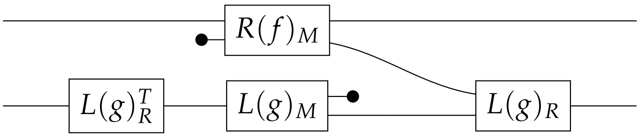
We now turn our attention to $[(fg)h]_M$, which we unpack as follows:

$[(fg)h]_M =$ 

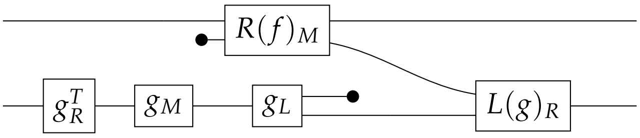
Analogously, we compute $R(fg)_M = (fg)_R^T (fg)_M (fg)_R$:

$R(fg)_M =$ 

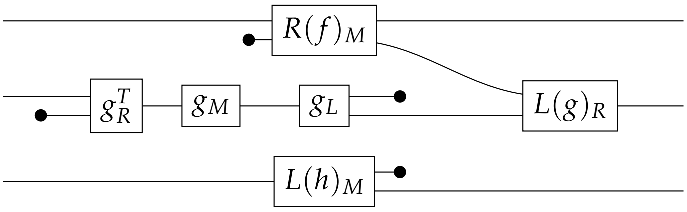
And using that $R(f)_R = \mathsf{id}$,

$R(fg)_M =$ 

Finally using that $L(g)_R^T L(g)_M = g_R^T g_M g_L$:

$R(fg)_M =$ 

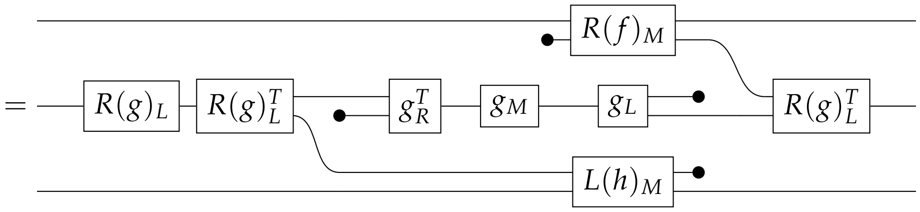
Giving us the simplified diagram for $[(fg)h]_M$:

$[(fg)h]_M =$ 

At this point it is clear that $[(fg)h]_M$ and $[f(gh)]_M$ are related by a permutation. We now complete the proof by showing that $[(fg)h]_M \cong [f(gh)]_M$ for the permutation

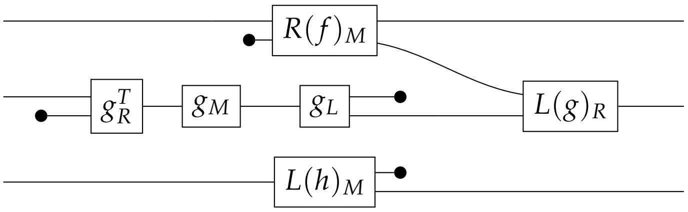
$\mathsf{id} \otimes R(g)_L^T \otimes \mathsf{id}$:

$R(g)_L[f(gh)]_M R(g)_L^T$



Finally, by applying Proposition B.1 we obtain

$R(g)_L[f(gh)]_M R(g)_L^T =$



$= [(fg)h]_M$

□

Proposition B.3. $[f(gh)]_R = (\mathsf{id} \otimes R(g)_L^T \otimes \mathsf{id})[(fg)h]_R$

Proof. By similar graphical reasoning, one can compute $[(fg)h]_R$ and show it equal to $(\mathsf{id} \otimes R(g)_L^T \otimes \mathsf{id})[(fg)h]_R$ □

Proposition B.4. $[f(gh)]_L = (\mathsf{id} \otimes R(g)_L^T \otimes \mathsf{id})[(fg)h]_L$

Proof. Once again, computing $(\mathsf{id} \otimes R(g)_L^T \otimes \mathsf{id})[(fg)h]_R$ allows us to reach $[f(gh)]_L$ □

Proposition B.5. $[f(gh)]_N = [(fg)h]_N(\mathsf{id} \otimes R(g)_L^T \otimes \mathsf{id})$

Proof. Same as for f_M, where f_N is treated as a diagonal matrix. □

Proposition B.6. *(Composition in* **Har** *is associative up to permutation)*

Proof. Immediate from propositions B.2, B.4, B.3, and B.5. □

B.2 Experimental Setup

Our experimental setup uses the ASV benchmarking library for Python, and BenchmarkingTools for Julia. Since each language has its own idiosyncracies, we expect that using a language-specific framework for each implementation will give the fairest results.

B.2.1 Software Versions

TABLE B.1: Software Versions

Software	Version
Python	3.9.4
NumPy	1.20.1
SciPy	1.6.3
Julia	1.6.1
Catlab.jl	0.12.2

B.2.2 Hardware Information

TABLE B.2: Hardware Information

Machine	CPU	CPU Frequency	# cores	System Memory
Dell XPS15 7590	Intel Core i7-9750H	2.60GHz	12	16GB

Appendix C

Proofs for Chapter 6

C.1 Sequential Normal Form

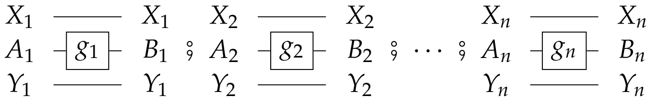
The decomposition described here is well-known in the literature [5, 76, 77]. We nevertheless provide a proof here for completeness.

Proposition C.1. *Let $\mathscr{C}$ be a monoidal category presented by generators and equations. Then any (finite) term t representing a morphism of $\mathscr{C}$ can be factored into 'layers':*

$$(\mathsf{id} \otimes g_1 \otimes \mathsf{id}) \fatsemi (\mathsf{id} \otimes g_2 \otimes \mathsf{id}) \fatsemi \ldots \fatsemi (\mathsf{id} \otimes g_n \otimes \mathsf{id})$$

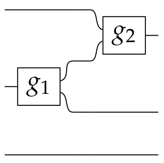
where each g_i is a generating morphism.

This factorization can be diagrammed as follows:



Note that in general $X_i \neq X_{i+1}$ and so on- i.e., the generators need not be "aligned" in this factorization. For example, we can have morphisms like the following:

Example C.1.



Proof. We proceed by induction on terms. Let S_0 denote the set of terms consisting of identities and generators, Then let

$$S_n = S_0 \cup \{t \fatsemi u | t, u \in S_{n-1}\} \cup \{t \otimes u | t, u \in S_{n-1}\}$$

Terms in S_0 are clearly already in sequential normal form, so it remains to prove the inductive case. Assume that any term $w \in S_{n-1}$ has an equivalent term $\hat{w}$ in sequential normal form. Then there are three cases:

1. If $v \in S_{n-1}$, then we have $\hat{v}$ by inductive hypothesis.
2. If $v = t \fatsemi u$, then $\hat{t}$ and $\hat{u}$ exist by inductive hypothesis, and we can form $\hat{v} = \hat{t} \fatsemi \hat{u}$.
3. If $v = t \otimes u$, then $\hat{v} = (\hat{t} \otimes \mathsf{id}) \fatsemi (\mathsf{id} \otimes \hat{u})$

and the proof is complete. □

C.2 Well-Definedness of $\mathcal{N}$

In this appendix we verify that $\mathcal{N}$ is well-defined. This amounts to two things: first that $\mathcal{N}$ is well-defined with respect to the interchange law, and second that it respects the equations of Definition 6.5.

In the former case, sequential normal forms are only unique up to interchange [77], so it must be verified that $\mathcal{N}$ is well-defined with respect to this property. This can be done by checking each of the cases in Definition 6.10. This is straightforward but tedious; each case follows essentially by naturality and [47, Equations 2.12, 2.13].

In the latter case, we need to verify the equations of Definition 6.5. Specifically, for each of the functor, adapter, and monoidal equations $\texttt{lhs} = \texttt{rhs}$, we show that $\mathcal{N}(\texttt{lhs}) = \mathcal{N}(\texttt{rhs})$. We give derivations for these below. Using these derivations one can then check that the equations hold for cases $\mathsf{id} \bullet \texttt{lhs} \bullet \mathsf{id} = \mathsf{id} \bullet \texttt{rhs} \bullet \mathsf{id}$. This means that $\mathcal{N}$ is equal under any rewrite involving the equations of Definition 6.5. These checks are relatively straightforward, but we note that the monoidal equations require the use of the pentagon and triangle axioms, respectively.

We now check that $\mathcal{N}$ is well-defined with respect to the functor, adapter, and monoidal equations of Definition 6.5, beginning with the functor equations (6.6).

$$\mathcal{N}(\overline{\mathsf{id}_A}) = \mathsf{id}_A = \mathcal{N}(\mathsf{id}_{\overline{A}})$$

$$\mathcal{N}(\overline{f} \fatsemi \overline{g}) = \mathcal{N}(\overline{f}) \fatsemi \mathcal{N}(\overline{g}) = f \fatsemi g = \mathcal{N}(\overline{f \fatsemi g})$$

Now the adapter equations (6.7):

$$
\begin{aligned}
\mathcal{N}(\Phi \fatsemi (\overline{f} \bullet \overline{g}) \fatsemi \Phi^*) &= \mathcal{N}(\Phi) \fatsemi \mathcal{N}(\overline{f} \bullet \overline{g}) \fatsemi \mathcal{N}(\Phi^*) \\
&= \mathcal{N}(\overline{f} \bullet \overline{g}) \\
&= \mathcal{N}(\overline{f} \bullet \mathsf{id}) \fatsemi \mathcal{N}(\mathsf{id} \bullet \overline{g}) \\
&= (f \otimes \mathsf{id}) \fatsemi (\mathsf{id} \otimes g) \\
&= f \otimes g \\
&= \mathcal{N}(\overline{f \otimes g})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{N}(\Phi^* \fatsemi \overline{f \otimes g} \fatsemi \Phi^*) &= \mathcal{N}(\Phi^*) \fatsemi \mathcal{N}(\overline{f \otimes g}) \fatsemi \mathcal{N}(\Phi) \\
&= \mathcal{N}(\overline{f \otimes g}) \\
&= f \otimes g \\
&= (f \otimes \mathsf{id}) \fatsemi (\mathsf{id} \otimes g) \\
&= \mathcal{N}(\overline{f} \bullet \mathsf{id}) \fatsemi \mathcal{N}(\mathsf{id} \bullet \overline{g}) \\
&= \mathcal{N}((\overline{f} \bullet \mathsf{id}) \fatsemi (\mathsf{id} \bullet \overline{g})) \\
&= \mathcal{N}(\overline{f} \bullet \overline{g})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{N}(\phi \fatsemi \phi^*) &= \mathcal{N}(\phi) \fatsemi \mathcal{N}(\phi^*) \\
&= \mathsf{id}_{I_{\mathscr{C}}} \fatsemi \mathsf{id}_{I_{\mathscr{C}}} \\
&= \mathsf{id}_{I_{\mathscr{C}}} \\
&= \mathcal{N}(\mathsf{id}_{I_{\overline{\mathscr{C}}}})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{N}(\phi^* \fatsemi \phi) &= \mathcal{N}(\phi^*) \fatsemi \mathcal{N}(\phi) \\
&= \mathsf{id}_{I_{\mathscr{C}}} \fatsemi \mathsf{id}_{I_{\mathscr{C}}} \\
&= \mathsf{id}_{I_{\mathscr{C}}} \\
&= \mathcal{N}(\overline{\mathsf{id}_{I_{\mathscr{C}}}}) \\
&= \mathcal{N}(\mathsf{id}_{\overline{I_{\mathscr{C}}}})
\end{aligned}
$$

Finally the associator/unitor equations (6.8):

$$\begin{aligned}
&\mathcal{N}(\Phi^* \mathbin{;} (\mathsf{id} \bullet \Phi^*) \mathbin{;} (\Phi \bullet \mathsf{id}) \mathbin{;} \Phi) \\
&\qquad = \mathcal{N}(\Phi^*) \mathbin{;} \mathcal{N}(\mathsf{id} \bullet \Phi^*) \mathbin{;} \mathcal{N}(\Phi \bullet \mathsf{id}) \mathbin{;} \mathcal{N}(\Phi) \\
&\qquad = \mathsf{id} \mathbin{;} \mathsf{id} \mathbin{;} \alpha \mathbin{;} \mathsf{id} \\
&\qquad = \alpha \\
&\qquad = \mathcal{N}(\overline{\alpha})
\end{aligned}$$

$$\begin{aligned}
&\mathcal{N}(\Phi^* \mathbin{;} (\Phi^* \bullet \mathsf{id}) \mathbin{;} (\mathsf{id} \bullet \Phi) \mathbin{;} \Phi) \\
&\qquad = \mathcal{N}(\Phi^*) \mathbin{;} \mathcal{N}(\Phi^* \bullet \mathsf{id}) \mathbin{;} \mathcal{N}(\mathsf{id} \bullet \Phi) \mathbin{;} \mathcal{N}(\Phi) \\
&\qquad = \mathsf{id} \mathbin{;} \alpha^{-1} \mathbin{;} \mathsf{id} \mathbin{;} \mathsf{id} \\
&\qquad = \alpha^{-1} \\
&\qquad = \mathcal{N}(\overline{\alpha^{-1}})
\end{aligned}$$

$$\begin{aligned}
\mathcal{N}(\Phi^* \mathbin{;} (\phi^* \bullet \mathsf{id})) &= \mathcal{N}(\Phi^*) \mathbin{;} \mathcal{N}(\phi^* \bullet \mathsf{id}) \\
&= \mathsf{id} \mathbin{;} \lambda \\
&= \lambda \\
&= \mathcal{N}(\overline{\lambda})
\end{aligned}$$

$$\begin{aligned}
\mathcal{N}((\phi \bullet \mathsf{id}) \mathbin{;} \Phi) &= \mathcal{N}(\phi \bullet \mathsf{id}) \mathbin{;} \mathcal{N}(\Phi) \\
&= \lambda^{-1} \mathbin{;} \mathsf{id} \\
&= \lambda^{-1} \\
&= \mathcal{N}(\overline{\lambda^{-1}})
\end{aligned}$$

$$\begin{aligned}
\mathcal{N}(\Phi^* \mathbin{;} (\mathsf{id} \bullet \phi^*)) &= \mathcal{N}(\Phi^*) \mathbin{;} \mathcal{N}(\mathsf{id} \bullet \phi^*) \\
&= \mathsf{id} \mathbin{;} \rho \\
&= \rho \\
&= \mathcal{N}(\overline{\rho})
\end{aligned}$$

$$\begin{aligned}\mathcal{N}((\mathsf{id} \fatsemi \phi) \fatsemi \Phi) &= \mathcal{N}(\mathsf{id} \fatsemi \phi) \fatsemi \mathcal{N}(\Phi)\\ &= \rho^{-1} \fatsemi \mathsf{id}\\ &= \rho^{-1}\\ &= \mathcal{N}(\overline{\rho^{-1}})\end{aligned}$$

Thus $\mathcal{N}$ is well-defined with respect to the monoidal equations.

C.3 Coherence Corollary

This section contains a brief graphical exposition of Mac Lane's corollary to the coherence theorem. Note that we essentially just recount Mac Lane's proof in a graphical way, and so the author's contribution is only pedagogical. In contrast, our proof of the main coherence result in Section 6.7 is novel.

We begin with an informal statement of the coherence corollary. Take a commuting diagram of $\mathscr{W}$, for instance the triangle axiom below left (C.1).

$$\begin{tikzcd} W \otimes (I_{\mathscr{C}} \otimes W) \arrow[rr, "\alpha_{W,I_{\mathscr{C}},W}"] \arrow[dr, "\mathsf{id}_W \otimes \lambda_W"'] & & (W \otimes I_{\mathscr{C}}) \otimes W \arrow[dl, "\rho_W \otimes \mathsf{id}_W"] \\ & W \otimes W & \end{tikzcd} \tag{C.1}$$

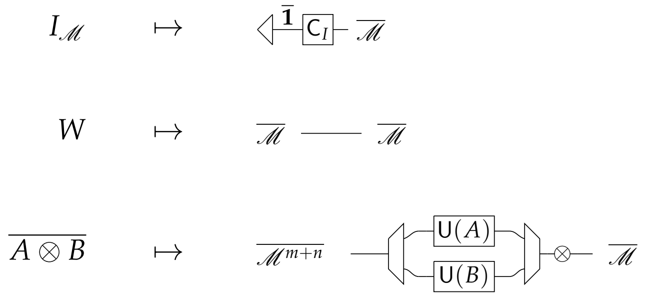
$$\begin{tikzcd} A \otimes (I_{\mathscr{C}} \otimes B) \arrow[rr, "\alpha_{A,I_{\mathscr{C}},B}"] \arrow[dr, "\mathsf{id}_A \otimes \lambda_B"'] & & (A \otimes I_{\mathscr{C}}) \otimes B \arrow[dl, "\rho_A \otimes \mathsf{id}_B"] \\ & A \otimes B & \end{tikzcd} \tag{C.2}$$

The coherence theorem allows one to 'export' this diagram to an arbitrary monoidal category $\mathscr{M}$ by replacing each i^{th} occurrence of W in a vertex with some A_i in $\mathscr{M}$. For instance, let A and B be $\mathscr{M}$ objects, then we substitute the first occurrence of W in each vertex for A, and the second for B, giving us the following commuting diagram in $\mathscr{M}$ on the right (C.2).

Remark C.2. The coherence theorem does **not** say that diagrams in $\mathscr{M}$ whose edges are components of natural transformations all commute; only those which correspond to diagrams in $\mathscr{W}$. In other words, if we have parallel $\mathscr{M}$-arrows $f, g : A \to B$ such that f, g are constructed from associators and unitors, we may not in general conclude that $f = g$.

Now, it is not immediately obvious how even this informal coherence result follows from the statement of Theorem 6.23. Although for some fixed object $M \in \mathscr{M}$ there is a unique, strict monoidal functor $\mathsf{U} : \mathscr{W} \to \mathscr{M}$, this does not let us obtain every diagram we would like. In particular, using U in this way we cannot obtain diagrams with multiple variables such as (C.2)–only those where *every* W is replaced by M.

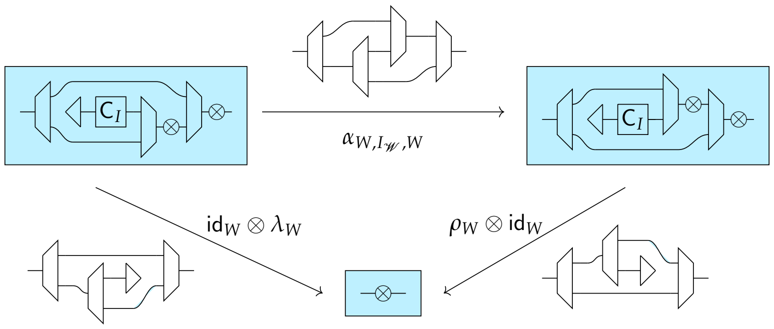
To allow for diagrams with multiple variables, Mac Lane constructs the non-strict monoidal category $\mathsf{It}(\mathscr{M})$. This will allow us to regard objects $A \in \mathscr{W}$ of size m as *functors*. In particular, the unique strict monoidal functor U will map A to a functor $\mathsf{U}(A) : \mathscr{M}^m \to \mathscr{M}$ as follows.



(C.3)

In the above, A and B are assumed to have `size` m and n, respectively. In addition, $-\boxed{\mathsf{C}_I}-$ denotes the strictification of the constant functor $\mathsf{Const}_I : \mathbf{1} \to \mathscr{M}$ mapping the single object of $\mathbf{1}$ to the monoidal unit $I_{\mathscr{M}}$, and $\otimes$ represents the strictification of the tensor product of $\mathscr{M}$.

Now, $\mathsf{U} : \mathscr{W} \to \mathsf{It}(\mathscr{M})$ preserves diagrams since it is a functor, and so we may picture the triangle axiom in $\mathsf{It}(\mathscr{M})$ graphically as below.


C_I
$\otimes$
$\alpha_{W,I_{\mathscr{W}},W}$
C_I
$\otimes$
$\mathsf{id}_W \otimes \lambda_W$
$\rho_W \otimes \mathsf{id}_W$
$\otimes$

The vertices of the diagram above are highlighted in blue and depict functors, while edges depict natural transformations. This transformation of $\mathscr{W}$-objects to functors formalises the intuition of 'replacing the i^{th} occurrence of W in a diagram'. More concretely, for a given diagram in $\mathscr{W}$ with vertices V_i of size n, we now simply make a particular choice of $\mathscr{M}^n$-object for each vertex and apply $\mathsf{U}(V_i) : \mathscr{M}^n \to \mathscr{M}$ to obtain a 'multivariable' diagram in $\mathscr{M}$.

For completeness, we now define $\mathsf{It}(\mathscr{M})$ as in Mac Lane [80] and show how it is monoidal.

Definition C.3. $\mathsf{It}(\mathscr{M})$ (from [80, p. 169])

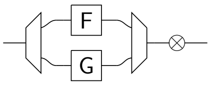
Fix an arbitrary monoidal category $(\mathscr{M}, \otimes, I_{\mathscr{M}}, \alpha, \lambda, \rho)$. Then $\mathsf{It}(\mathscr{M})$ is the category with:

1. Objects: functors $\mathscr{M}^n \to \mathscr{M}$
2. Arrows: natural transformations

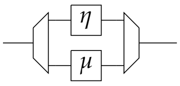
With $\mathscr{M}^n$ denoting the n-fold product $\mathscr{M} \times \overset{n}{\ldots} \times \mathscr{M}$.

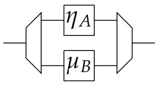
The use of our graphical notation above is justified because $\mathsf{It}(\mathscr{M})$ forms a monoidal category in the following way.

Proposition C.4. $\mathsf{It}(\mathscr{M})$ *is a (non-strict) monoidal category (from [80, p. 169])*
The monoidal unit is the constant functor $\mathsf{Const}_I : \mathbf{1} \to \mathscr{M}$*. The monoidal product* $\square : \mathsf{It}(\mathscr{M}) \times \mathsf{It}(\mathscr{M}) \to \mathsf{It}(\mathscr{M})$ *is defined on objects (functors* F, G*) as:*

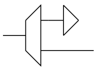
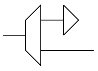
$\mathsf{F}\square\mathsf{G} = \mathscr{M}^{m+n}$  $\mathscr{M}$

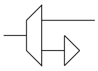
and pointwise on arrows $\eta : \mathsf{F}_1 \to \mathsf{G}_1$ *and* $\mu : \mathsf{F}_2 \to \mathsf{G}_2$ *so that for* $\mathsf{F}_1, \mathsf{G}_1 : \mathscr{M}^m \to \mathscr{M}$ *and* $\mathsf{F}_2, \mathsf{G}_2 : \mathscr{M}^n \to \mathscr{M}$ *the component at* $A \times B \in \mathscr{M}^{m+n}$ *is*

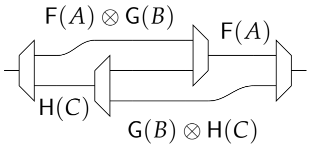
$(\eta\square\mu)_{A\times B} = \Big(\mathsf{F}_1\square\mathsf{F}_2$  $\mathsf{G}_1\square\mathsf{G}_2 \Big)_{A\times B}$

$= \mathsf{F}_1(A) \otimes \mathsf{F}_2(B)$  $\mathsf{G}_1(A) \otimes \mathsf{G}_2(B)$

Associators and unitors are similarly defined pointwise.

$\lambda_{\mathsf{F}_A} = \Big(\mathsf{Const}_{I_{\mathscr{M}}}\square\mathsf{F}$  $\mathsf{F} \Big)_A = I_{\mathscr{M}} \otimes \mathsf{F}(A)$  $\mathsf{F}(A)$

$\rho_{\mathsf{F}_A} = \mathsf{F}(A) \otimes I_{\mathscr{M}}$  $\mathsf{F}(A)$

$\alpha_{\mathsf{F},\mathsf{G},\mathsf{H}_{A,B,C}} =$ 

Proof. Associators and unitors are natural since each of their components is natural. That is, given a natural transformation $\mu : \mathsf{F} \to \mathsf{G}$ we know that $\rho_{\mathsf{F}} \fatsemi \mu = (\mu\square\mathsf{id}) \fatsemi \rho_{\mathsf{G}}$ precisely because the components of both sides are always equal. For example, for all A we have $\rho_{\mathsf{F}_A} \fatsemi \mu = (\mu\square\mathsf{id})_A \fatsemi \rho_{\mathsf{G}_A}$. A similar argument applies to α and λ. Further, the axioms of monoidal categories are satisfied for the same reason: each diagram commutes because all its *components* commute using the monoidal structure of $\mathscr{M}$. □

Finally, we can state the coherence result as in Mac Lane [80].

Corollary C.5. *(from [80, p. 169]) Let $\mathscr{M}$ be a monoidal category. There is a function which assigns to each pair of objects $A, B \in \mathscr{W}$ of* `size` *n a (unique) natural isomorphism $\mathsf{can}_{\mathscr{M}}(A, B) : \mathsf{U}(A) \to \mathsf{U}(B)$ called the canonical map from $\mathsf{U}(A)$ to $\mathsf{U}(B)$, in such a way that the identity arrow $\mathsf{Const}_{I_{\mathscr{M}}} \to \mathsf{Const}_{I_{\mathscr{M}}}$ is canonical (between functors of 0 variables) the identity transformation $\mathsf{id} : \mathsf{id}_{\mathscr{M}} \to \mathsf{id}_{\mathscr{M}}$ is canonical, α, λ, ρ (and their inverses) are canonical, and the composite and $\square$-product of canonical maps is canonical.*

Proof. (from [80, p. 169])
Let $\mathsf{U} : \mathscr{W} \to \mathsf{It}(\mathscr{M})$ be the unique strict monoidal functor mapping W to the identity functor $\mathsf{id} : \mathscr{M} \to \mathscr{M}$ so that U acts on objects as in (C.3). Then U acts on morphisms of $\mathscr{W}$ as follows.

$$\mathsf{id}_{I_{\mathscr{W}}} \mapsto \mathsf{id} \qquad \mathsf{id}_W \mapsto \mathsf{id}$$

$$\lambda_A \mapsto \lambda_{\mathsf{U}(A)} \qquad \rho_A \mapsto \rho_{\mathsf{U}(A)} \qquad \alpha_{A,B,C} \mapsto \alpha_{\mathsf{U}(A),\mathsf{U}(B),\mathsf{U}(C)}$$

$$f \otimes g \mapsto \mathsf{U}(f) \square \mathsf{U}(g) \qquad f \mathbin{⨟} g \mapsto \mathsf{U}(f) \mathbin{⨟} \mathsf{U}(g)$$

Thus $\mathsf{can}_{\mathscr{M}}(A, B) = \mathsf{U}(f)$ for each unique $f : A \to B$ in $\mathscr{W}$. □

Finally, note that by Proposition 6.40, the canonical morphism $\mathsf{can}_{\mathscr{M}}(A, B)$ can be defined as $\mathsf{can}_{\mathscr{M}}(A, B) = (\mathsf{U} \circ \mathcal{N})(\mathsf{can}(\overline{A}, \overline{B}))$. Thus we may use the normal form $\mathsf{can}(A, B)$ to determine the canonical natural isomorphism in $\mathsf{It}(\mathscr{M})$.

Appendix D

Additional Material

D.1 Implementations

Implementation D.1 (Reverse Derivative Ascent Library).
https://github.com/statusfailed/rda

Implementation D.2 (Reverse Derivative Ascent Experiments).
https://github.com/statusfailed/act-2020-experiments

Implementation D.3 (Implementation of Numeric Optics Library and Experiments).
https://github.com/statusfailed/numeric-optics-python

Implementation D.4 (Cartographer-HAR).
https://github.com/statusfailed/cartographer-har

References

[1] Marcelo Aguiar and Swapneel Mahajan. *Monoidal Functors, Species and Hopf Algebras*. CRM monograph series. American Mathematical Society, Providence, RI, November 2010.

[2] Francisco Alarcón and Dan Anderson. Commutative semirings and their lattices of ideals. *Houston Journal of Mathematics*, 20, 1994. URL https://www.math.uh.edu/~hjm/vol20-4.html.

[3] Mario Alvarez-Picallo. Change actions: from incremental computation to discrete derivatives, 2020. URL https://arxiv.org/abs/2002.05256.

[4] Mario Alvarez-Picallo and C. H. Luke Ong. Change actions: Models of generalised differentiation, 2019. URL https://arxiv.org/abs/1902.05465.

[5] Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Functorial string diagrams for reverse-mode automatic differentiation. *CoRR*, abs/2107.13433, 2021. URL https://arxiv.org/abs/2107.13433.

[6] Armen S. Asratian, Tristan M. J. Denley, and Roland Häggkvist. *Bipartite Graphs and their Applications*. Cambridge University Press, 1998.

[7] John C. Baez and Aaron D. Lauda. A prehistory of n-categorical physics. In *Deep Beauty*, pages 13–128. Cambridge University Press, apr 2011. doi: 10.1017/cbo9780511976971.003. URL https://doi.org/10.1017%2Fcbo9780511976971.003.

[8] John C. Baez and Jade Master. Open petri nets. *Mathematical Structures in Computer Science*, 30(3):314–341, mar 2020. doi: 10.1017/s0960129520000043. URL https://doi.org/10.1017%2Fs0960129520000043.

[9] Francesco Bellucci and AHTI Pietarinen. From mitchell to carus: Fourteen years of logical graphs in the making. *Transactions of the Charles S. Peirce Society*, 52, 03 2016. doi: 10.2979/trancharpeirsoc.52.4.02.

[10] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013. URL https://arxiv.org/abs/1308.3432.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2007. ISBN 978-0-387-31073-2.

[12] R.F. Blute, J.R.B. Cockett, R.A.G. Seely, and T.H. Trimble. Natural deduction and coherence for weakly distributive categories. *Journal of Pure and Applied Algebra*, 113(3):229–296, 1996. ISSN 0022-4049. doi: https://doi.org/10.1016/0022-4049(95)00159-X. URL `https://www.sciencedirect.com/science/article/pii/002240499500159X`.

[13] Richard F. Blute, J. R. B. Cockett, and R. A. G. Seely. Cartesian differential categories. *Theory and Applications of Categories*, 22, 2009. URL `https://emis.univie.ac.at/journals/TAC/volumes/22/23/22-23abs.html`.

[14] Guillaume Boisseau. String diagrams for optics, 2020. URL `https://arxiv.org/abs/2002.11480`.

[15] Guillaume Boisseau and Paweł Sobociński. String diagrammatic electrical circuit theory. *Electronic Proceedings in Theoretical Computer Science*, 372:178–191, nov 2022. doi: 10.4204/eptcs.372.13. URL `https://doi.org/10.4204%2Feptcs.372.13`.

[16] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. A categorical semantics of signal flow graphs. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 – Concurrency Theory*, pages 435–450, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44584-6.

[17] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. Rewriting modulo symmetric monoidal structure. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, jul 2016. doi: 10.1145/2933575.2935316. URL `https://doi.org/10.1145%2F2933575.2935316`.

[18] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory i: Rewriting with frobenius structure, 2020. URL `https://arxiv.org/abs/2012.01847`.

[19] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory ii: Rewriting with symmetric monoidal structure, 2021. URL `https://arxiv.org/abs/2104.14686`.

[20] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawe Sobociski, and Fabio Zanasi. String diagram rewrite theory iii: Confluence with and without frobenius, 2021. URL `https://arxiv.org/abs/2109.06049`.

[21] Filippo Bonchi, Pawel Sobociński, and Fabio Zanasi. A Survey of Compositional Signal Flow Theory. In *Advancing Research in Information and Communication*

Technology, volume AICT-600, pages 29–56. 2021. doi: 10.1007/978-3-030-81701-5_2. URL https://hal.inria.fr/hal-03325995. part : TC 1: Foundations of Computer Science.

[22] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. *A Survey of Compositional Signal Flow Theory*, pages 29–56. Springer International Publishing, Cham, 2021. ISBN 978-3-030-81701-5. doi: 10.1007/978-3-030-81701-5_2. URL https://doi.org/10.1007/978-3-030-81701-5_2.

[23] Dylan Braithwaite and Jules Hedges. Dependent bayesian lenses: Categories of bidirectional markov kernels with canonical bayesian inversion, 2022. URL https://arxiv.org/abs/2209.14728.

[24] Titouan Carette and Simon Perdrix. Colored props for large scale graphical reasoning, 2020. URL https://arxiv.org/abs/2007.03564.

[25] Titouan Carette, Yohann D'Anello, and Simon Perdrix. Quantum algorithms and oracles with the scalable ZX-calculus. *Electronic Proceedings in Theoretical Computer Science*, 343:193–209, sep 2021. doi: 10.4204/eptcs.343.10. URL https://doi.org/10.4204%2Feptcs.343.10.

[26] Emma Chollet, Bryce Clarke, Michael Johnson, Maurine Songa, Vincent Wang, and Gioele Zardini. Limits and colimits in a category of lenses. *Electronic Proceedings in Theoretical Computer Science*, 372:164–177, nov 2022. doi: 10.4204/eptcs.372.12. URL https://doi.org/10.4204%2Feptcs.372.12.

[27] Francois Chollet et al. Keras, 2015. URL https://github.com/fchollet/keras.

[28] Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. Symmetries in reversible programming: From symmetric rig groupoids to reversible programming languages, 2021. URL https://arxiv.org/abs/2110.05404.

[29] Robin Cockett, Geoffrey Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon Plotkin, and Dorette Pronk. Reverse derivative categories, 2019.

[30] Bob Coecke and Ross Duncan. A graphical calculus for quantum observables. *Preprint*, 2007.

[31] Bob Coecke and Ross Duncan. Interacting quantum observables. In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, 2008. doi: 10.1007/978-3-540-70583-3_25.

[32] G. A. Constantinides. Rethinking arithmetic for deep neural networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and*

Engineering Sciences, 378(2166):20190051, 2020. doi: 10.1098/rsta.2019.0051. URL https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2019.0051.

[33] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *arXiv:1511.00363 [cs]*, 2015.

[34] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016. URL https://arxiv.org/abs/1602.02830.

[35] G. S. H. CRUTTWELL. Cartesian differential categories revisited. *Mathematical Structures in Computer Science*, 27(1):7091, 2017. doi: 10.1017/S0960129515000055.

[36] G. S. H. Cruttwell, Bruno Gavranović, Neil Ghani, Paul Wilson, and Fabio Zanasi. Categorical foundations of gradient-based learning, 2021. URL https://arxiv.org/abs/2103.01931.

[37] Geoffrey Cruttwell, Jonathan Gallagher, and Dorette Pronk. Categorical semantics of a simple differential programming language. *Electronic Proceedings in Theoretical Computer Science*, 333:289–310, feb 2021. doi: 10.4204/eptcs.333.20. URL https://doi.org/10.4204%2Feptcs.333.20.

[38] Geoffrey Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, and Dorette Pronk. Monoidal reverse differential categories, 2022. URL https://arxiv.org/abs/2203.12478.

[39] Giovanni de Felice, Alexis Toumi, and Bob Coecke. DisCoPy: Monoidal categories in python. *Electronic Proceedings in Theoretical Computer Science*, 333: 183–197, feb 2021. doi: 10.4204/eptcs.333.13. URL https://doi.org/10.4204%2Feptcs.333.13.

[40] Pierre de Fermat. Letter to frénicle de bessy, 1640.

[41] Lucas Dixon and Aleks Kissinger. Open graphs and monoidal theories, 2010. URL https://arxiv.org/abs/1011.4114.

[42] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[43] Ran Duan, Hongxun Wu, and Renfei Zhou. Faster matrix multiplication via asymmetric hashing, 2022. URL https://arxiv.org/abs/2210.10173.

[44] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, USA, 2000. ISBN 978-0-471-05669-0.

[45] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 2018. doi: 10.48550/ARXIV.1808.05377. URL `http://jmlr.org/papers/v20/18-598.html`.

[46] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *J. Mach. Learn. Res.*, 11:625660, mar 2010. ISSN 1532-4435.

[47] P. I. Etingof, Shlomo Gelaki, Dmitri Nikshych, and Victor Ostrik, editors. *Tensor categories*. Number volume 205 in Mathematical surveys and monographs. American Mathematical Society, 2015. ISBN 978-1-4704-2024-6. URL `https://klein.mit.edu/~etingof/egnobookfinal.pdf`.

[48] Brendan Fong. Decorated cospans. 2015. doi: 10.48550/ARXIV.1502.00872. URL `https://arxiv.org/abs/1502.00872`.

[49] Brendan Fong and Michael Johnson. Lenses and learners. *Proceedings of the 8th International Workshop on Bidirectional Transformations*, 2019. doi: 10.48550/ARXIV.1903.03671. URL `https://arxiv.org/abs/1903.03671`.

[50] Brendan Fong, David I. Spivak, and Rémy Tuyéras. Backprop as functor: A compositional perspective on supervised learning. 2017. doi: 10.48550/ARXIV.1711.10455. URL `https://arxiv.org/abs/1711.10455`.

[51] Apache Software Foundation. Apache airflow, 2022. URL `https://airflow.apache.org/`.

[52] Foundry. Nuke, 2022. URL `https://www.foundry.com/products/nuke-family/nuke`.

[53] Thomas Fox. Coalgebras and cartesian categories, Jan 1976. URL `http://dx.doi.org/10.1080/00927877608822127`.

[54] Bruno Gavranović. Space-time tradeoffs of lenses and optics via higher category theory, 2022. URL `https://arxiv.org/abs/2209.09351`.

[55] Dan R. Ghica and Achim Jung. Categorical semantics of digital circuits. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 41–48, 2016. doi: 10.1109/FMCAD.2016.7886659.

[56] Dan R. Ghica and Achim Jung. Categorical semantics of digital circuits. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 41–48, 2016. doi: 10.1109/FMCAD.2016.7886659.

[57] Dan R. Ghica, George Kaye, and David Sprunger. A compositional theory of digital circuits, 2022.

[58] Jonathan S. Golan. *Linear Algebra over a Semiring*, pages 211–221. Springer Netherlands, Dordrecht, 1999. ISBN 978-94-015-9333-5. doi: 10.1007/978-94-015-9333-5_19. URL https://doi.org/10.1007/978-94-015-9333-5_19.

[59] Jonathan S Golan. *Semirings and their Applications*. Springer, Dordrecht, Netherlands, 2010. doi: 10.1007/978-94-015-9333-5.

[60] John L. Gustafson and Isaac T. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2), 2017. doi: 10.14529/jsfi170206.

[61] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250269, September 1978. ISSN 0098-3500. URL https://doi.org/10.1145/355791.355796.

[62] Amar Hadzihasanovic and Diana Kessler. Data structures for topologically sound higher-dimensional diagram rewriting, 2022. URL https://arxiv.org/abs/2209.09509.

[63] Jules Hedges. Limits of bimorphic lenses, 2018. URL https://arxiv.org/abs/1808.05545.

[64] Benjamin Heintz, Rankyung Hong, Shivangi Singh, Gaurav Khandelwal, Corey Tesdahl, and Abhishek Chandra. Mesh: A flexible distributed hypergraph processing system, 2019.

[65] Peter Hines. Identities in modular arithmetic from reversible coherence operations, 2013. URL https://arxiv.org/abs/1304.7128.

[66] Peter Hines. Coherence and strictification for self-similarity, 2015.

[67] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. doi: 10.1016/0893-6080(89)90020-8.

[68] Nathan Jacobson. *Basic Algebra I: Second Edition*. Courier Corporation, December 2012. ISBN 978-0-486-13522-9.

[69] Jeff Johnson. Rethinking floating point for deep learning, 2018. URL https://arxiv.org/abs/1811.01721.

[70] André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in Mathematics*, 88(1):55–112, 1991. ISSN 0001-8708. doi: https://doi.org/10.1016/0001-8708(91)90003-P. URL https://www.sciencedirect.com/science/article/pii/000187089190003P.

[71] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558562, nov 1962. ISSN 0001-0782. doi: 10.1145/368996.369025. URL `https://doi.org/10.1145/368996.369025`.

[72] Lukasz Kaiser, Aidan N. Gomez, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones, and Jakob Uszkoreit. One model to learn them all, 2017. URL `https://arxiv.org/abs/1706.05137`.

[73] Aleks Kissinger. Abstract tensor systems as monoidal categories, 2013. URL `https://arxiv.org/abs/1308.3586`.

[74] Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. In Bob Coecke and Matthew Leifer, editors, Proceedings 16th International Conference on *Quantum Physics and Logic,* Chapman University, Orange, CA, USA., 10-14 June 2019, volume 318 of *Electronic Proceedings in Theoretical Computer Science*, pages 229–241. Open Publishing Association, 2020. doi: 10.4204/EPTCS.318.14.

[75] Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. In *Automated Deduction - CADE-25*, pages 326–336. Springer International Publishing, 2015. doi: 10.1007/978-3-319-21401-6_22. URL `https://doi.org/10.1007%2F978-3-319-21401-6_22`.

[76] Yves Lafont. Towards an algebraic theory of Boolean circuits. *Journal of Pure and Applied Algebra*, 184(2-3):257–310, 2003. ISSN 00224049. doi: 10.1016/S0022-4049(03)00069-0.

[77] Yves Lafont. *Equational reasoning with 2-dimensional diagrams*, volume 909, pages 170–195. 01 2006. ISBN 978-3-540-59340-9. doi: 10.1007/3-540-59340-3_13.

[78] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998. doi: 10.1109/5.726791.

[79] Moshe Leshno et al. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6): 861–867, 1993. doi: 10.1016/s0893-6080(05)80131-5.

[80] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1997. ISBN 978-1-4419-3123-8. doi: 10.1007/978-1-4757-4721-8.

[81] 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. URL `https://doi.org/10.1007/11874683_1`.

[82] Evan Patterson and other contributors. Algebraicjulia/catlab.jl: v0.12.2, May 2021. URL https://doi.org/10.5281/zenodo.4736069.

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

[84] Evan Patterson, Owen Lynch, and James Fairbanks. Categorical data structures for technical computing. 06 2021. URL https://arxiv.org/abs/2106.04703.

[85] Evan Patterson, David I. Spivak, and Dmitry Vagner. Wiring diagrams as normal forms for computing in symmetric monoidal categories. *Electronic Proceedings in Theoretical Computer Science*, 333:49–64, feb 2021. doi: 10.4204/eptcs.333.4. URL https://doi.org/10.4204%2Feptcs.333.4.

[86] Roger Penrose. Applications of negative dimensional tensors. In *Combinatorial Mathematics and its Applications*, 1971. URL https://www.mscs.dal.ca/~selinger/papers/graphical-bib/public/Penrose-applications-of-negative-dimensional-tensors.pdf.

[87] D. PLUMP. *TERM GRAPH REWRITING*, pages 3–61. doi: 10.1142/9789812815149_0001. URL https://www.worldscientific.com/doi/abs/10.1142/9789812815149_0001.

[88] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, pages 1–8, Montreal, Quebec, Canada, 2009. ACM Press. ISBN 978-1-60558-516-1. doi: 10.1145/1553374.1553486.

[89] Mitchell Riley. Categories of optics, 2018. URL https://arxiv.org/abs/1809.00738.

[90] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016. URL https://arxiv.org/abs/1609.04747.

[91] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, jan 2015. doi: 10.1016/j.neunet.2014.09.003. URL https://doi.org/10.1016%2Fj.neunet.2014.09.003.

[92] P. Selinger. A survey of graphical languages for monoidal categories. In *New Structures for Physics*, pages 289–355. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-12821-9_4. URL https://doi.org/10.1007%2F978-3-642-12821-9_4.

[93] Claude E. Shannon. *The Theory and Design of Linear Differential Equation Machines*, pages 514–559. Jan 1942. doi: 10.1109/9780470544242.ch33.

[94] Michael Shulman. A practical type theory for symmetric monoidal categories, 2019. URL https://arxiv.org/abs/1911.00818.

[95] Pawel Sobocinski, Paul W. Wilson, and Fabio Zanasi. CARTOGRAPHER: A Tool for String Diagrammatic Reasoning (Tool Paper). In Markus Roggenbach and Ana Sokolova, editors, *8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*, volume 139 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:7, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-120-7. doi: 10.4230/LIPIcs.CALCO.2019.20. URL http://drops.dagstuhl.de/opus/volltexte/2019/11448.

[96] David I. Spivak. Learners' languages. *Electronic Proceedings in Theoretical Computer Science*, 372:14–28, nov 2022. doi: 10.4204/eptcs.372.2. URL https://doi.org/10.4204%2Feptcs.372.2.

[97] David Sprunger and Shin-ya Katsumata. Differentiable causal computations via delayed trace, 2019. URL https://arxiv.org/abs/1903.01093.

[98] Richard S Sutton and Andrew G Barto. *Reinforcement Learning*. Adaptive Computation and Machine Learning series. Bradford Books, Cambridge, MA, 2 edition, November 2018.

[99] Unity Software. Unity3d, 2022. URL https://unity3d.com/.

[100] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

[101] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. LUTNet: Rethinking Inference in FPGA Soft Logic. *IEEE International Symposium on Field-Programmable Custom Computing Machines*, April 2019. doi: 10.1109/FCCM.2019.00014.

[102] William Wernick. Complete sets of logical functions. *Transactions of the American Mathematical Society*, 51(1):117, 1942. doi: 10.2307/1989982.

[103] Paul Wilson and Fabio Zanasi. The cost of compositionality: A high-performance implementation of string diagram composition, 2021. URL https://arxiv.org/abs/2105.09257.

[104] Paul Wilson and Fabio Zanasi. Reverse derivative ascent: A categorical approach to learning boolean circuits. *Electronic Proceedings in Theoretical*

Computer Science, 333:247–260, feb 2021. doi: 10.4204/eptcs.333.17. URL https://doi.org/10.4204%2Feptcs.333.17.

[105] Paul Wilson and Fabio Zanasi. Categories of differentiable polynomial circuits for machine learning, 2022. URL https://arxiv.org/abs/2203.06430.

[106] Paul Wilson and Fabio Zanasi. An axiomatic approach to differentiation of polynomial circuits. *Journal of Logical and Algebraic Methods in Programming*, 135: 100892, 2023. ISSN 2352-2208. doi: https://doi.org/10.1016/j.jlamp.2023.100892. URL https://www.sciencedirect.com/science/article/pii/S2352220823000469.

[107] Paul Wilson, Dan Ghica, and Fabio Zanasi. String diagrams for non-strict monoidal categories, 2022. URL https://arxiv.org/abs/2201.11738.

[108] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. Integer quantization for deep learning inference: Principles and empirical evaluation, 2020. URL https://arxiv.org/abs/2004.09602.

[109] Fabio Zanasi. Interacting hopf algebras: the theory of linear systems, 2018. URL https://arxiv.org/abs/1805.03032.