!-Logic

First-Order Reasoning for Families of Non-Commutative String Diagrams

David Quick

University College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2015

Acknowledgements

Firstly, I would like to thank Carmel College Darlington, in particular Dr Janice Gorlach, for sparking my interest in Mathematics and inspiring me to study it further. I would like to thank Dr Andrew Ker and Professor Peter Howell for giving me the opportunity not only to study but also teach Mathematics at the University of Oxford. Without their continued support I doubt I would have completed my undergraduate degree.

I would like to thank Professor Bob Coecke and Professor Samson Abramsky for giving me the opportunity to continue my education in the form of a DPhil at this amazing university. Being able to work with such incredible academics has been an absolute privilege. To Dr Aleks Kissinger, who supervised the final years of my DPhil, I cannot express the gratitude I feel. Not only did Aleks pave the way for my research, but he also gave me invaluable advice at every stage along the way.

This thesis would not have been possible without funding from the Engineering and Physical Sciences Research Council. I would also like to thank the administrative staff at both University College and the Department of Computer Science, in particular Julie Sheppard, for helping me navigate the bureaucratic paperwork.

I would like to thank my friends, both in and out of Oxford, without whom I would have surely finished much sooner. Thank you for the many distractions and for those still to come.

Finally, I would like to thank my family. To my parents, I cannot portray the immense debt of gratitude I have for everything you have given me over the past 26 years. Without you I would literally not exist. To my sisters, without whom I would not wish to exist, I thank you for the endless support and encouragement. I can only hope to have given you as much joy as you have given me.

Abstract

Equational reasoning with string diagrams provides an intuitive method for proving equations between morphisms in various forms of monoidal category. !-Graphs were introduced with the intention of reasoning with infinite families of string diagrams by allowing repetition of sub-diagrams. However, their combinatoric nature only allows commutative nodes. The aim of this thesis is to extend the !-graph formalism to remove the restriction of commutativity and replace the notion of equational reasoning with a natural deduction system based on first order logic.

The first major contribution is the syntactic !-tensor formalism, which enriches Penrose's abstract tensor notation to allow repeated structure via !-boxes. This will allow us to work with many noncommutative theories such as bialgebras, Frobenius algebras, and Hopf algebras, which have applications in quantum information theory.

A more subtle consequence of switching to !-tensors is the ability to definitionally extend a theory. We will demonstrate how noncommutativity allows us to define nodes which encapsulate entire diagrams, without inherently assuming the diagram is commutative. This is particularly useful for recursively defining arbitrary arity nodes from fixed arity nodes. For example, we can construct a !-tensor node representing the family of left associated trees of multiplications in a monoid.

The ability to recursively define nodes goes hand in hand with proof by induction. This leads to the second major contribution of this thesis, which is !-Logic (!L). We extend previous attempts at equational reasoning to a fully fledged natural deduction system based on positive intuitionistic first order logic, with conjunction, implication, and universal quantification over !-boxes. The key component of !L is the principle of !-box induction. We demonstrate its application by proving how we can transition from fixed to arbitrary arity theories for monoids, antihomomorphisms, bialgebras, and various forms of Frobenius algebras. We also define a semantics for !L, which we use to prove its soundness.

Finally, we reintroduce commutativity as an optional property of a morphism, along with another property called symmetry, which describes morphisms which are not affected by cylic permutations of their edges. Implementing these notions in the !-tensor language allows us to more easily describe theories involving symmetric or commutative morphisms, which we then demonstrate for recursively defined Frobenius algebra nodes.

Contents

Chapter 1

Introduction

Many real world processes come with natural notions of sequential and parallel composition, these are particularly prevalent in Mathematics, Physics, and Computer Science. Having two such orthogonal notions suggests we are working with something inherently two dimensional. In such cases, reasoning using a standard one dimensional term based syntax is often counter-intuitive and hides the deep underlying structure of the problem. Recently there has been a shift towards equational reasoning using two dimensional graphical calculi, particularly employing different topologies on string diagrams to naturally represent a large variety of monoidal categories.

Many algebraic and coalgebraic structures admit representations where the generating morphisms can have variable arity. For example, an arbitrary arity multiplication operation can be defined to replace the finite arity generators of a monoid. Reasoning about such morphisms involves describing infinite families of string diagrams. *!-Graphs* were introduced as a formalism for dealing with such families of string diagrams by allowing designated subdiagrams to be repeated. Unfortunately, the combinatoric architecture underlying !-graphs restricts to only allow theories in which every morphism is commutative, i.e. independent of the order of wires. Removing this restriction is the first of the two main goals of this thesis. We expand upon work in [28, 30] presenting *!-tensors*, a new formalism still allowing infinite families of diagrams but not making any assumptions about the effect of permuting the wires of a morphism.

Previous attempts at diagrammatic reasoning have been based on substitution, where a !-graph is rewritten by finding a subgraph which can be replaced by an equivalent but generally simpler graph. This form of rewriting can be formalised in category theory as the double pushout graph rewriting method [15, 34]. While rewriting allows for a large number of proofs to be formalised, it crucially does not encapsulate proof by induction on !-boxes. Previous attempts at using proof by induction have been largely ad hoc, with the only attempt to formalise it being in [39] where the technique of *fixing* a !-box was introduced. This effectively freezes the !-box, no longer allowing

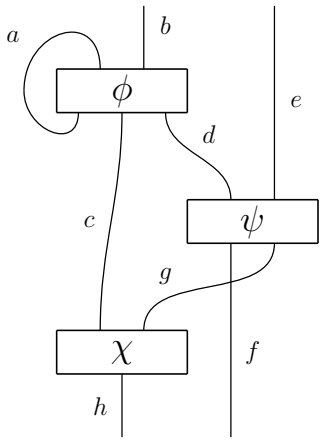
any operations to be applied to it. Fixing was introduced as a temporary measure until a suitable logic could be developed allowing quantification over !-boxes. This thesis introduces such a formal logic for !-tensors which we call *!-logic* (expanding upon work in [29]). !-Logic is based on positive intuitionistic first order logic [18, 11] and allows conjunction, implication, and crucially universal quantification over !-boxes. This allows us to formalise the principle of !-box induction, which is vital if we wish to reason about families of diagrams.

The use of diagrams in physics can be traced back to at least the 1950s. *Feynman Diagrams* [22] and diagrammatic representations of *abstract tensor notation* [40] can both be used as more intuitive ways to represent spatial/temporal structures than the previous one dimensional term based syntax. Abstract tensor notation [40] was developed by Roger Penrose as a notation to represent multi-linear maps in a basis-independent manner. Penrose noted that it was the connectivity that was important rather than the indices themselves and even introduced a graphical representation. The use of such graphical notation was often held back by the technology of the time[41]:

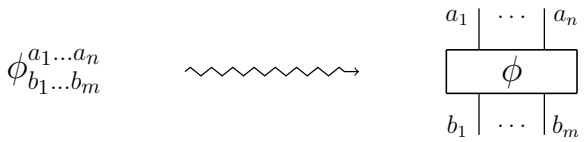
> The notation has been found very useful in practice as it greatly simplifies the appearance of complicated tensor or spinor equations, the various interrelations expressed being discernable at a glance. Unfortunately the notation seems to be of value mainly for private calculations because it cannot be printed in the normal way. (Spinors and Spacetime vol. 1)

Advances in technology have resulted in a new excitement over graphical reasoning. The following shows an example of how a term in abstract tensor notation can be visualised graphically:

$$\phi^{ab}_{acd}\psi^{de}_{fg}\chi^{cg}_{h} \rightsquigarrow \tag{1.1}$$

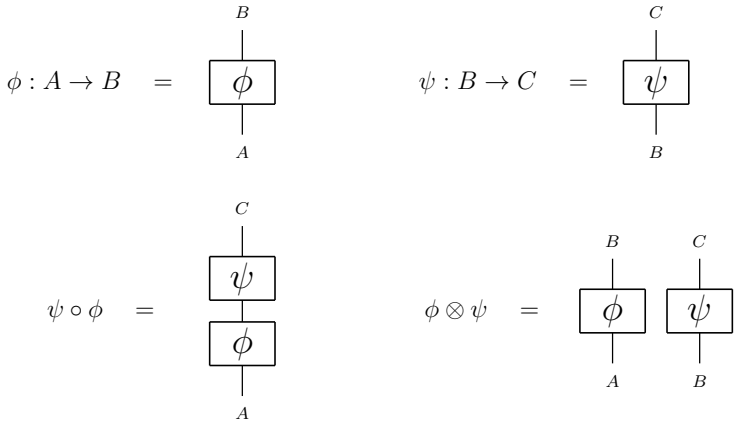


A term of the form $\phi^{a_1 \ldots a_n}_{b_1 \ldots b_m}$, which represents a morphism ϕ with inputs $a_1 \ldots a_n$ and outputs $b_1 \ldots b_m$, can be drawn as a box with input wires attached below and output wires attached above (in their prescribed order).

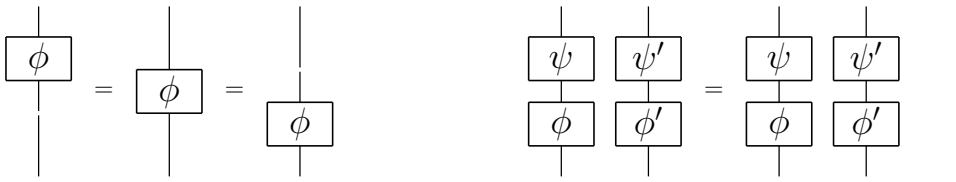


Contraction of repeated names in the tensor expression syntax is shown diagrammatically by plugging an output wire into an input wire.

The categorical foundation for diagrams of this form comes from *monoidal categories*. Given a monoidal category $(\mathcal{C}, \otimes)$, we can choose to represent categorical composition vertically and the monoidal product horizontally:



The axioms of a category and a monoidal category can be seen to be trivial topological deformations of these diagrams. Hence when working pictographically we no longer need concern ourselves with these axioms as they are inherently built-in to the notation. For example, the fact that identities act as units for composition and functoriality of $\otimes$ are seen by:

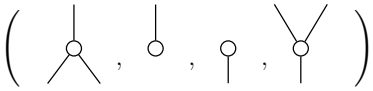


$\phi \circ 1_A \;=\; \phi \;=\; 1_B \circ \phi$ $\qquad$ $(\psi \otimes \psi') \circ (\phi \otimes \phi') = (\psi \circ \phi) \otimes (\psi' \circ \phi')$

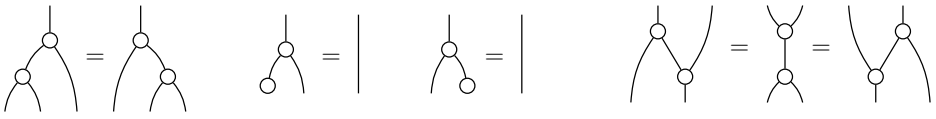
Choosing which kinds of topological deformations are allowed to be applied to diagrams can give us graphical languages representing a variety of additional structures associated with monoidal categories. For example, in [20], Joyal and Street proved that a topology on *progressive planar diagrams* (box and wire diagrams as in (1.1) except not allowing loops or wires to cross) directly corresponds to the axioms of a monoidal category. Their proof demonstrates that progressive planar string diagrams considered up to topological deformations, act as a *free monoidal category*. Here the term 'free' means that two morphisms are equal if and only if they are equal up to the axioms of the monoidal category.

Joyal and Street next demonstrated that allowing wires to cross, extends the topology to a free symmetric monoidal category. They also suggested that the topology of non-progressive diagrams corresponds to either *free symmetric traced* or *compact closed categories*, though a proof for this was not published until [24].

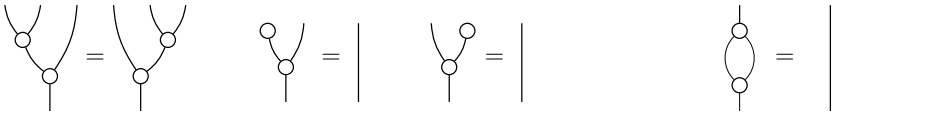
These graphical notations have appeared in fields such as computational linguistics [23], control theory [2, 3], and perhaps most prevalently in categorical quantum mechanics [1, 4, 5, 6]. For example, †-*special commutative Frobenius algebras* (†SCFA) are of particular interest in categorical quantum mechanics. We can describe their structure diagrammatically using four generators (with the dagger operation flipping diagrams vertically)



along with nine equalities:

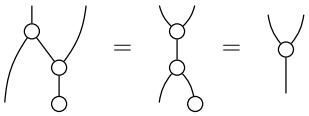


Monoid Laws **Frobenius Law**

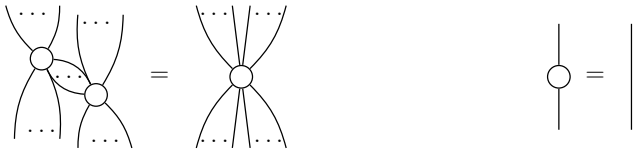


Comonoid Laws **Special Law**

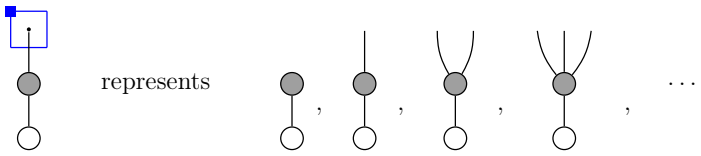
We can then reason with diagrams by rewriting (substituting subgraphs using the axioms):



In the case of a †SCFA, it can be shown that the morphism represented by any connected network of nodes depends only on the number of input and output wires. For example, the above three diagrams, each with one input and two outputs, are all equal. This suggests a more concise representation of a †SCFA, where we quotient over the internal structure of connected networks, simply drawing them as single nodes with the correct number of inputs and outputs. The axioms can then be replaced by a rule allowing two connected nodes to be combined, often referred to as a spider law, along with a rule to simplify the special case of a node with a single input and a single output:

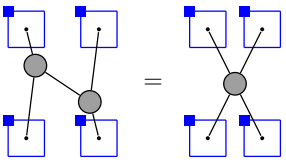


!-Graphs were introduced as a formalism to allow working with such families of diagram equations with repeated structure [14]. A !-graph can contain subgraphs marked by blue boxes (called !-boxes) which are allowed to be repeated.

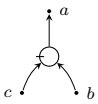


A formal semantics for rewriting using graphs with !-boxes was given in [26] (making use of adhesive categories [35]); this was implemented in the automated proof assistant Quantomatic [27] (see [32] for details).

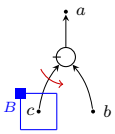
!-Graph equations represent families of concrete equations. For example, the following !-graph equation has as instances all equations allowing two connected gray nodes to be combined to form one node.



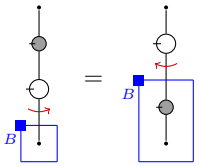
The problem with this existing formalism is its combinatoric representation, which has no means of tracking the order of edges. If we wish to describe noncommutative bialgebras, Hopf algebras, and Frobenius algebras we will need to introduce a new formalism. The first contribution of this thesis is the introduction of two new pieces of diagrammatic notation to keep track of the order of edges. To understand the first, imagine a single fixed arity circular node. We need to order the node's edges, in particular noting which is the first. We do this by adding a tick on the node before the first edge counting clockwise.



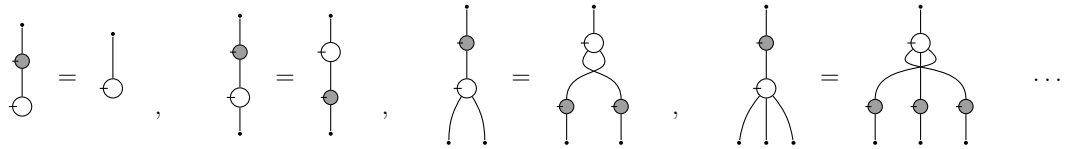
We can now read the edge labels of the above node in order as a, followed by b, followed by c. The second notation deals with families of such diagrams. We need to specify where new copies of edges from a !-box are added. We do this by drawing either clockwise or anticlockwise arrows over edges entering !-boxes. We refer to these arrows as arcs.



The arc in the above diagram stipulates that when creating a new copy of the !-box B, we create a new edge anticlockwise from c. To see how ticks and arcs together allow for noncommutative theories, take the following !-tensor equation:

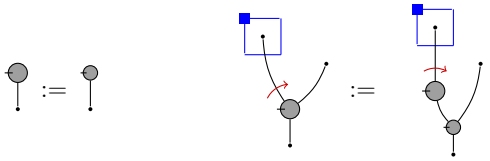


The difference in arc directions dictates the twisting property of this !-tensor equation's instances:

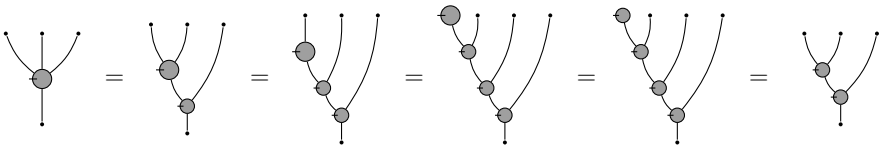


Along with the new diagrams we will introduce a new syntactic representation to replace the combinatoric !-graph formalism. The syntax we introduce is based on Penrose's abstract tensor notation but adapted to deal with compact closed categories and to allow !-boxes.

An added advantage of noncommutativity is the ability to define a node to encapsulate an entire diagram, often useful to reduce the size of large complicated diagrams. Attempting the same for !-graphs assumes the diagram is commutative on inputs and outputs, which is often not the case. Perhaps most significantly, these definitional extensions can be used to recursively define variable arity nodes in terms of fixed arity nodes. As an example, take the following definition of a family of arbitrarily sized left associated trees of comultiplications, defined in terms of a binary comultiplication  and its counit  :

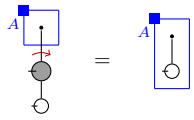


We have drawn the arbitrary arity node slightly larger to help distinguish between them. To see how this definition works we can unfold it on a node with three outputs:

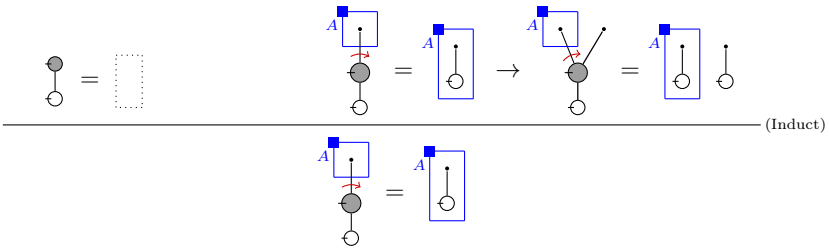


The notion of recursive definition goes hand in hand with proof by induction. We would now hope to be able to pass from statements about diagrams involving the finite arity nodes to statements about families of diagrams involving the arbitrary arity nodes. To do this we introduce !-box induction. Continuing with the comonoid example, suppose we have an additional two equations describing how the fixed arity generators interact with a new white unit:

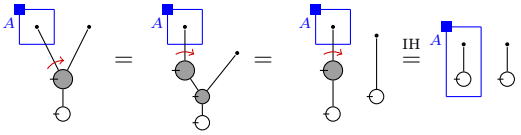
We see that the white unit is copied by the fixed arity comultiplication, so we might suspect the same is true of the arbitrary arity comultiplication. The statement of this can be written using the new !-tensor language:



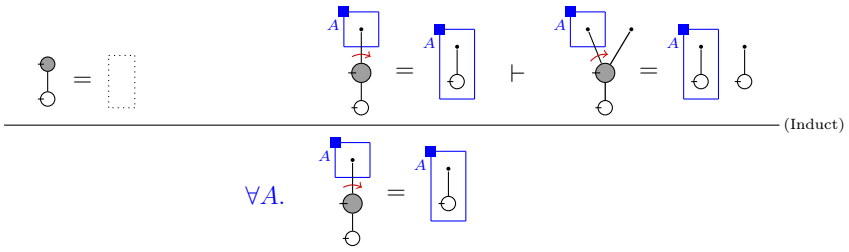
We might attempt to prove this using an induction principle for !-boxes:



Here we have split the problem into a base case with no copies of the !-box and a step case where n copies implies $n+1$ copies. The base case is precisely one of our axioms; for the step case we use the inductive hypothesis and the axioms in a rewriting proof:



Unfortunately, interpreting $\rightarrow$ as 'the set of instances on the left implies the set on the right' we find that this induction is invalid. The step case turns out to be vacuous, as the instances of the right hand side are a subset of those of the left hand side. This is analogous to trying to apply induction to first order logic by the rule $(\forall n.P(n)) \rightarrow (\forall n.P(n+1))$, which is trivially true for any proposition P. The reason for this problem is that the current equational reasoning approach takes a !-tensor equation to mean truth of all instances. This makes it impossible to reason about the individual instances as is necessary when applying induction. Instead we require the ability to talk about !-tensor equations before quantifying over them. With this goal in mind we introduce explicit quantification over !-boxes and use a turnstile symbol, $\vdash$, to denote that formulas on the left entail those on the right. We can hence fix our attempt at !-box induction to the following:



In this thesis we will present such a logic, called !-logic, allowing conjunction, implication, and crucially universal quantification over !-boxes. We build !L as a natural deduction system based on positive intuitionistic first order logic, encompassing !-box induction and prove it to be sound.

Throughout the thesis we present a number of purely diagrammatic proofs for algebraic structures such as monoids, antihomomorphisms, bialgebras, and Frobenius algebras. We often demonstrate a normal form result by transitioning to arbitrary arity nodes. In particular, the Frobenius algebra example presents proofs of normal forms for symmetric, commutative, and special Frobenius algebras which are more elegant than previous versions such as in [37].

In Chapter 2 we summarise the graphical reasoning background material necessary for the remainder of this thesis. We start with the definition of a monoidal category, demonstrating the string diagram topology corresponding to a free monoidal category. From monoidal categories we add additional structure to define *symmetric traced* and *compact closed categories*, presenting each of their free graphical languages.

We continue by describing the transition to string graph notation in the commutative case. String graphs are presented as typed graphs, using dummy vertices to hold wires in place. In section 2.3 we describe how !-boxes allow reasoning with families of string graphs and present the definition of !-graphs, which incorporates !-boxes in the string graph formalism.

Chapter 3 starts the original content of this thesis by developing a language for working with families of noncommutative diagrams called *!-tensors*. We start by adapting the string graph notation to track edge order by adding ticks to nodes. We then develop a syntax for such diagrams based on abstract tensor notation. Next we add !-boxes to describe families of such diagrams and extend the syntax to follow. Since !-boxes are used to add additional copies of a subdiagram, noncommutativity means it is important to prescribe where new edges should be added, which we draw by either clockwise or anticlockwise arcs over edges.

To retrieve the concrete diagrams represented by a !-tensor, we define !-box operations allowing us to create new copies of the contents of !-boxes or remove them entirely. Instantiations are sequences of such operations resulting in a concrete diagram, or if applied to a !-tensor equation resulting in a concrete tensor equation.

One of the obvious benefits of !-tensors is the ability to represent theories with noncommutative nodes such as antihomomorphisms, Frobenius algebras, and Hopf algebras. A much more subtle benefit is the ability to definitionally extend a theory. Suppose we wish to define a new node to represent some diagram; even if each node in the diagram is commutative, there is no reason to believe the combined diagram is. Hence defining such a node in the !-graph formalism makes an assumption about its structure. Having fixed edge orders mean !-tensors allow such definitions, including recursively defined arbitrary arity nodes. We demonstrate this process for trees of (co)multiplications in a number of (co)algebraic structures in section 3.4.1 and see a particularly interesting case for Frobenius algebras in chapter 6.

We conclude chapter 3 by demonstrating how !-tensors can be encoded as !-graphs with some additional data (first seen in [43]). This is of particular importance in adapting the current graphical

proof assistant Quantomatic [27] to allow for noncommutativity. Implementing !-tensor reasoning in this way would allow us to leverage the existing features of Quantomatic, including conjecture synthesis, simplification procedures, and the exporting of diagrammatic proofs as LaTeX code.

Chapter 4 follows up the definition of !-tensors by describing a logic over !-tensor equations. We explain the need for explicit quantification over !-boxes, which has so far been absent in the literature. Previous attempts at reasoning with families of diagrams [39] have treated !-boxes in an ad hoc manner, often relying on the ability to 'fix' a !-box so it is no longer allowed to have operations applied to it. !-Logic rigorously formalises reasoning with families of diagrams by building universal quantification of !-boxes into the proof language, allowing proofs which were previously unachievable.

We start by recursively defining *!-Formulas* from !-tensor equations and three connectives: conjunction, implication, and universal quantification. Then in section 4.2 we introduce *!-Logic* (!L) as a positive intuitionistic logic on !-formulas. The key accomplishment of this is formalising the principle of proof by induction on !-boxes. We demonstrate the newly formalised induction for a few examples in section 4.4 and again for Frobenius algebras in chapter 6.

In section 4.5 we define a semantics for !-formulas which extends to !-logic. This allows us to prove soundness of !L by ensuring that each rule preserves truth with respect to the semantics.

Chapter 5 describes two properties which morphisms can possess. The first is symmetry, which dictates that the morphism is not affected by cyclic permutations of the wires; the second is commutativity, which dictates that the morphism is not affected by any permutation. In each of these cases, the !-tensor notation obscures this edge order independence. To combat this we extend the !-tensor formalism to allow morphisms to be tagged as either symmetric or commutative (both graphically and syntactically) and build in their corresponding equivalences. Finally, we introduce metarules describing when a morphism can be considered symmetric or commutative.

Chapter 6 consists of a case study to demonstrate the power of !-tensors and !-logic. We start by presenting the definition of a (non-commutative) Frobenius algebra, which is not possible using !-graphs. We then use our new ability to recursively define nodes to create an arbitrary arity Frobenius algebra node. Next we use !-box induction to prove a theorem allowing connected nodes with aligned ticks to be combined. We continue by introducing symmetric, commutative, and special Frobenius algebras and using our metarules to transition from arbitrary nodes to symmetric and commutative nodes. In each case we present purely diagrammatic proofs of spider theorems, allowing connected networks of nodes to be combined to a single node. This results in a normal form for each kind of Frobenius algebra.

The final chapter summarises the advancements made in this thesis and then suggests a few topics for further research, specifically in the areas of extending the graphical formalism, implementing !-tensors in an automated theorem prover, and proving completeness of !-Logic.

Chapter 2

Background: Reasoning with Diagrams

This chapter summarises background knowledge on diagrammatic reasoning required to understand the original content of this thesis, which will start in chapter 3. We start by taking the definition of a *category* and extending it to a *monoidal category* which adds the notion of spatial composition (via the tensor product $\otimes$) to the temporal composition of the category ($\circ$). From here we present a variety of different types of category, each with its own graphical notation. The key categories we will be using in future chapters are *symmetric traced categories* and *compact closed categories*. These are the building blocks underlying the graphical language of string diagrams with feedback loops.

The second part of the chapter summarises !-graphs. This is a formalism for working with families of diagrams where we assume all morphisms have commutativity of input/output wires. This allows us to switch to a graph based notation which we represent via string-graphs.

Finally, we give a brief description of graph rewriting, which lets us substitute sections of a diagram using !-graph equations.

2.1 Monoidal Categories

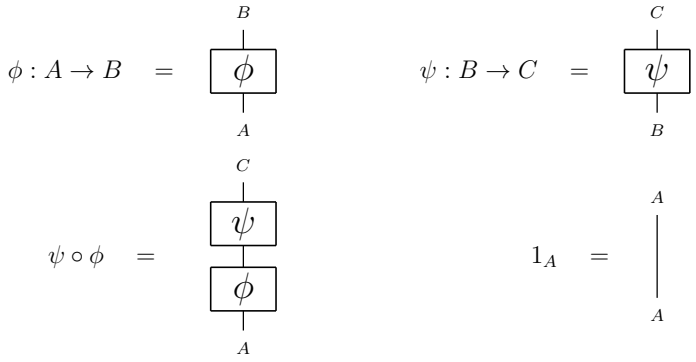
Definition 2.1.1 (Category)**.** A *category* $\mathcal{C}$ consists of:

- a collection of objects, written $\text{ob}(\mathcal{C})$;
- a collection of morphisms $\phi : A \to B$ for every pair of objects A, B, written $\mathcal{C}(A, B)$;
- a morphism $\psi \circ \phi : A \to C$ called the composite, for every pair of morphisms $\phi : A \to B,\ \psi : B \to C$ with common intermediate object;

- a morphism 1_A called the identity for each object A.

such that the composition operation ($\circ$) is associative and has identities (1_A) as units.

We can already represent morphisms in a category in a diagrammatic way. We draw functions as boxes with domain object as an input wire below and codomain object as an output wire above. Composition is then drawn by stacking morphisms vertically and identities are simply bare wires.



Hence the conditions of associativity and units are implicit in the notation:

$$\chi \circ (\psi \circ \phi) = (\chi \circ \psi) \circ \phi \qquad \phi \circ 1_A = \phi = 1_B \circ \phi$$

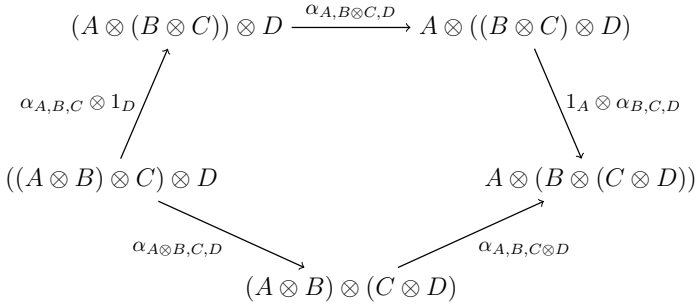
Definition 2.1.2 (Monoidal Category)**.** A *monoidal category* $\mathcal{C}$ is a category $\mathcal{C}$ along with:

- a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, called the tensor product;
- an object I called the tensor unit;
- three natural isomorphisms

$$\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$$
$$\lambda_A : (I \otimes A) \to A$$
$$\rho_A : (A \otimes I) \to A$$

such that the following diagrams commute:

$$\begin{array}{ccc} (A \otimes I) \otimes B & \xrightarrow{\alpha_{A,I,B}} & A \otimes (I \otimes B) \\ & {\scriptstyle \rho_A \otimes 1_B} \searrow \quad \swarrow {\scriptstyle 1_A \otimes \lambda_B} & \\ & A \otimes B & \end{array}$$



We say a monoidal category is *strict* if all three natural isomorphisms are identities. Maclane's coherence theorem [38] tells us that any well-typed equation built from α, λ, ρ, and their inverses holds. As a result of this we can replace any monoidal category with the *strictified* version where the natural isomorphisms act as identities. We can hence drop explicit use of α, λ, ρ, bracketing and (when possible) units, writing $A \otimes B \otimes C$ rather than $(A \otimes (I \otimes B)) \otimes C$.

Having decided that composition using $\circ$ should be drawn as vertical placement, we can naturally set the orthogonal notion of tensor composition as horizontal juxtaposition. Note that the tensor unit, and identities on it, do not appear in our diagrams as they disappear in tensor products $A \otimes I = A$, $\phi \otimes 1_I = \phi$.

Equations such as the following then become implicit in the graphical notation. The first is simply sliding morphisms past each other and the second (a consequence of $\otimes$ being a bifunctor) is trivial:

$(\phi \otimes 1_B) \circ (1_A \otimes \psi) = (1_B \otimes \psi) \circ (\phi \otimes 1_A)$ $\qquad$ $(\psi \otimes \psi') \circ (\phi \otimes \phi') = (\psi \circ \phi) \otimes (\psi' \circ \phi')$

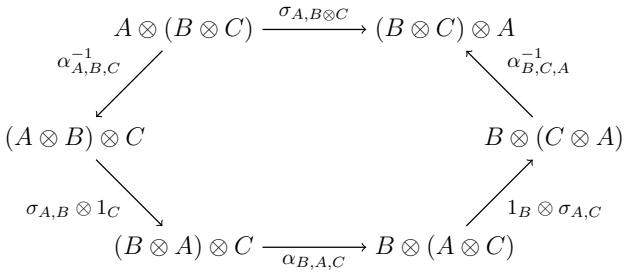
2.1.1 Symmetric Traced Categories

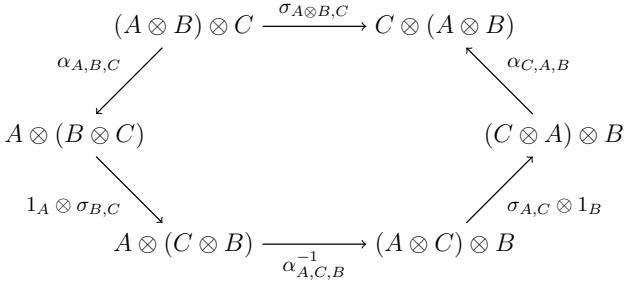
So far our diagrams do not allow wires to cross (i.e. move left/right past each other) nor pass through morphisms. We now present some extensions of monoidal categories allowing such flexibility of wires.

Definition 2.1.3 (Braided Monoidal Category)**.** A *braided monoidal category* $\mathcal{C}$ is a monoidal category $\mathcal{C}$ with an additional natural isomorphism called braiding:

$$\sigma_{A,B} : A \otimes B \to B \otimes A$$

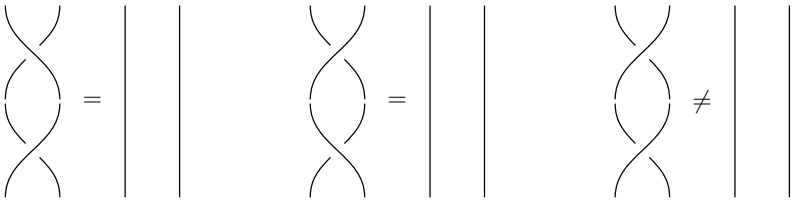
such that the following diagrams commute:



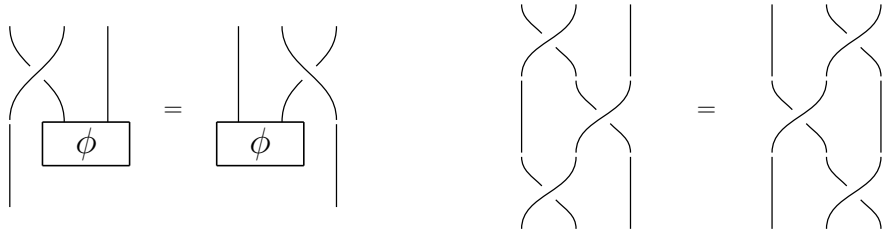


We draw $\sigma_{A,B}$ and its inverse graphically as one wire crossing 'over the top' of the other:

The following diagrams demonstrate that these representations are good choices for satisfying the inverse property. They also demonstrate graphically that our braidings are not self inverse (by which we mean $\sigma_{A,B}$ and $\sigma_{B,A}$ are not inverses) as it may have seemed from the definition.



Again many equations between morphisms in a braided monoidal category become implicit in the graphical notation. We can see that morphisms can now 'pass behind' (or in front of) wires and the Yang-Baxter equation becomes a trivial topological deformation.

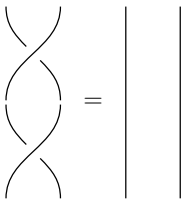


Definition 2.1.4 (Symmetric Monoidal Category)**.** A *symmetric monoidal category* $\mathcal{C}$ is a braided monoidal category $\mathcal{C}$ in which the braidings satisfy, for all objects A, B:

$$\sigma_{A,B}^{-1} = \sigma_{B,A}$$

In this case we refer to the braidings as symmetries.

Diagrammatically this enforces the following:



Note that this diagram still fits into a spatial isotopy by imagining increasing from the three dimensions of braided categories, where a wire in front of another cannot pass through it, to a four dimensional isotopy. This allows us to smoothly deform between the two diagrams above by moving one wire into the fourth dimension to navigate around the other wire. Hence we have seen that categories, monoidal categories, braided monoidal categories, and symmetric monoidal categories are inherently spatial isotopies with the number of dimensions increasing from one to four.

In each of these cases it has been shown that a well-formed equation between morphisms is true from the axioms if and only if it is true diagrammatically under the prescribed spatial isotopy. This is formalised in [20] where Joyal and Street defined progressive polarised diagrams. They demonstrated how up to isomorphism these correspond to symmetric monoidal categories and up to a 3D framed isotopy they correspond to braided monoidal categories. The former case has since been observed to be equivalent to a 4D framed isotopy.

Having removed the inability for wires to pass through each other, we no longer have need for wires passing above/below other wires. Hence to simplify the diagrams we replace the braiding notation with a graphical version of the symmetry isomorphism.

So far all of our diagrams have been progressive (no loops), but there are many physical systems involving feedback which we would like to be able to work with. One way to add loops is via a trace operation.

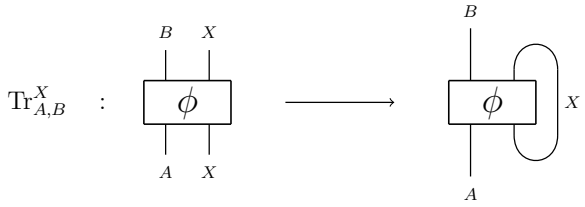
Definition 2.1.5 (Symmetric Traced Category)**.** A *symmetric traced category* $\mathcal{C}$ is a symmetric monoidal category $\mathcal{C}$ with a function called the *trace*:

$$\mathrm{Tr}_{A,B}^{X} : \mathcal{C}(A \otimes X, B \otimes X) \to \mathcal{C}(A, B)$$

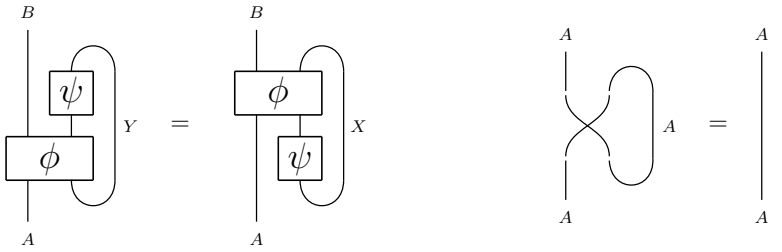
such that the following hold:

1. $\mathrm{Tr}^X((\chi \otimes 1_X) \circ \phi \circ (\psi \otimes 1_X)) = \chi \circ \mathrm{Tr}^X(\phi) \circ \psi$
2. $\mathrm{Tr}^X(\phi \circ (1_A \otimes \psi)) = \mathrm{Tr}^Y((1_B \otimes \psi) \circ \phi)$
3. $\mathrm{Tr}^I(\phi) = \phi$
4. $\mathrm{Tr}^{X \otimes Y}(\phi) = \mathrm{Tr}^X(\mathrm{Tr}^Y(\phi))$
5. $\mathrm{Tr}^X(\psi \otimes \phi) = \psi \otimes \mathrm{Tr}^X(\phi)$
6. $\mathrm{Tr}^X(\gamma_{X,X}) = 1_X$

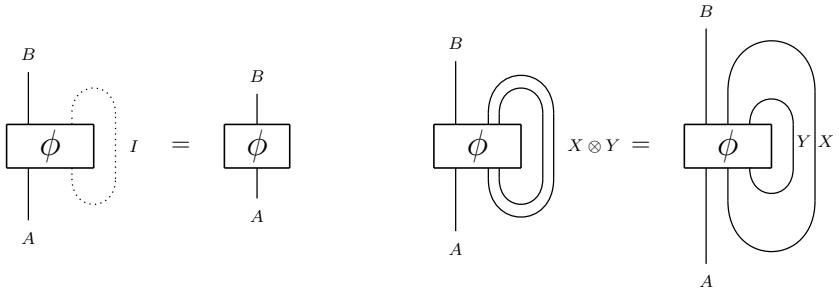
The trace operation has an intuitive notation as a feedback loop connecting the object being *traced out* back to itself.



Then the conditions above become trivial. Conditions 1 and 5 are implicit in the diagrams so not shown here, 2 and 6 become intuitive sliding and yanking conditions.



Conditions 3 and 4 become the following, where for clarity we have drawn a dotted line to represent the tensor unit (not usually drawn) and the object $X \otimes Y$ has been drawn as a double line.

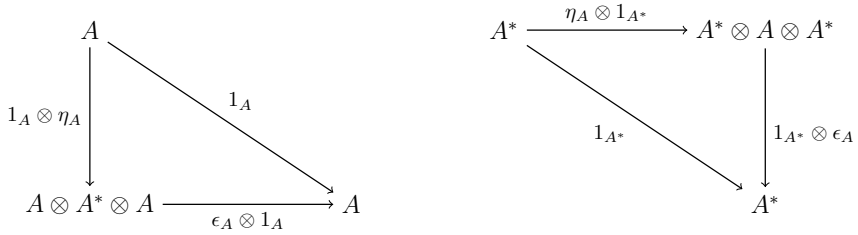


Thus symmetric traced categories are one formalism for adding feedback to planar diagrams. Another such formalism is *compact closed categories* which we present now.

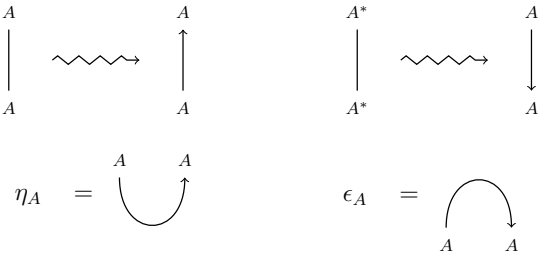
2.1.2 Compact Closed Categories

Many mathematical concepts come with a notion of duals. We will now show how objects with duals can be used to allow wires to reverse direction and so lead to planar diagrams with loops.

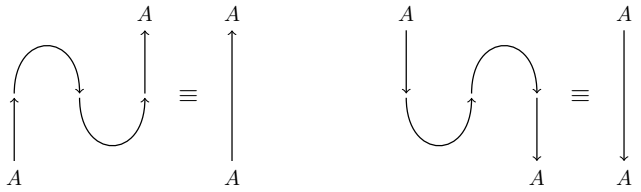
Definition 2.1.6 (Dual Objects)**.** For A, A^* objects in a monoidal category, we say that A^* is the *right dual* of A (equivalently A is the left dual of A^*) if there exist morphisms $\eta_A : I \to A^* \otimes A$, $\epsilon_A : A \otimes A^* \to I$ such that the following diagrams commute:



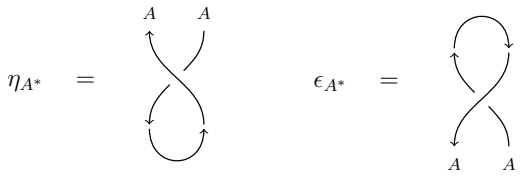
Given an object A which has a right dual A^* we replace the usual undirected wires with wires directed upwards for A and downwards for A^*. The morphisms η and ϵ can then be given intuitive representations as a *cup* and *cap* respectively.



The commutative diagrams then become the following diagrams (aptly named the yanking equations).



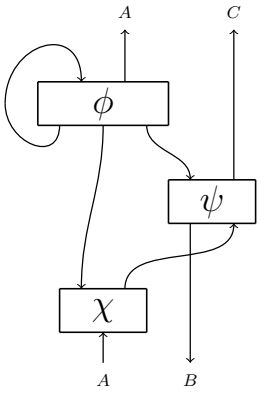
In a braided monoidal category, being the left dual and being the right dual are equivalent, as we can take a cap and cup for A and construct caps and cups for the dual A^*:



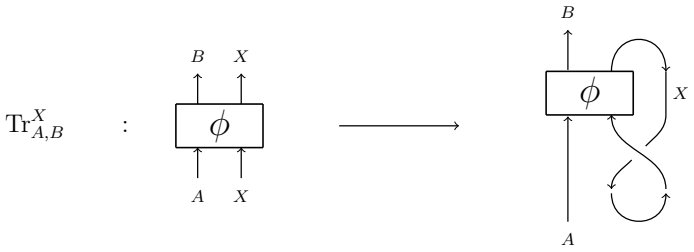
It is easy to see diagrammatically that these satisfy the yanking equation. In this case we refer to A^* as simply being the dual of A as in the following definition.

Definition 2.1.7 (Compact Closed Category)**.** A *compact closed category* is a symmetric monoidal category where every object has a dual.

Diagrams in a compact closed category (CCC) can be drawn as planar diagrams (with loops) where all wires are directed.



The cup and cap of a CCC can be used to construct a trace operation $\mathrm{Tr}^X_{A,B}(\phi) = (1_B \otimes \epsilon_X) \circ (\phi \otimes 1_{X^*}) \circ (1_A \otimes \sigma_{X,X^*}) \circ (1_A \otimes \eta_X)$ or graphically:



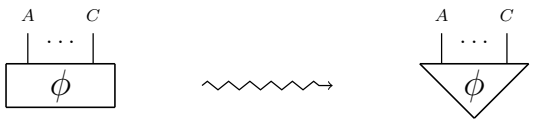
The six trace axioms can then be checked diagrammatically.

Joyal and Street proved conversely that any symmetric traced category can be fully and faithfully embedded in a compact closed category [21]. Hence from now on we will work with planar diagrams with loops which we assume to represent an underlying CCC.

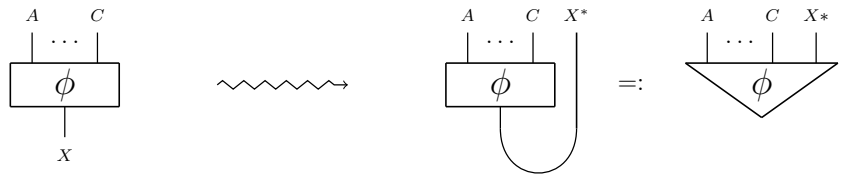
For a thorough guide to graphical languages for a large variety of monoidal categories see [44].

2.2 Commutativity and Graph Notation

A *point* is a morphism ϕ which has the tensor unit as its domain $\phi : I \to X \otimes \ldots \otimes Z$. We have a special notation for such morphisms taking advantage of the fact that (graphically) they have no inputs.

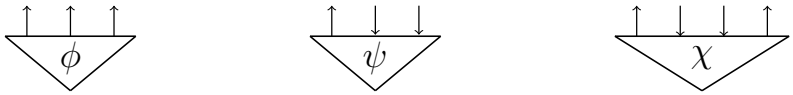


Given a morphism $\phi : X \to A \otimes \ldots \otimes C$ we can use the cups from our compact closed structure to replace the domain object X with its dual in the codomain.



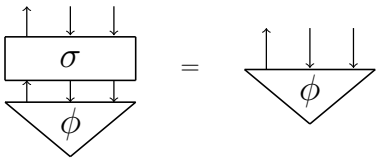
More generally we can replace all objects in the domain with their duals in the codomain. Hence every morphism can be represented as a point. Note that we can return to the original morphism using caps.

For simplicity let us assume that we are working in a CCC which has only a single generating object type X, then objects are tensor products of X and X^* and morphisms are points into these. Since we have only one object type we can drop labels, with arrows pointing up representing X and down representing X^*.

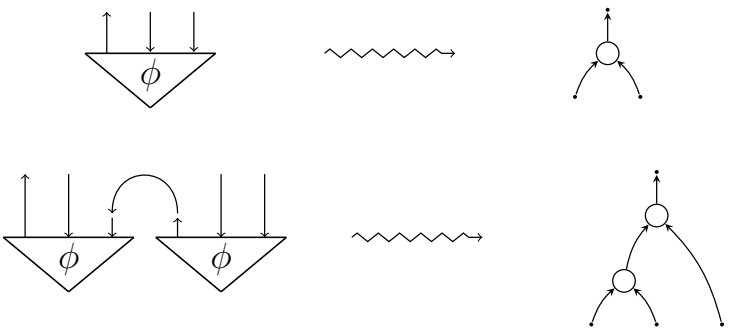


The notation of these points intrinsically allows us to track wires by position. For example, I can describe the 'first', 'second', etc. wire counting from the left, hence assigning a total order on wires.

Definition 2.2.1 (Commutativity)**.** The statement that a point ϕ is *commutative* is the imposition that wire order is unimportant to ϕ. By this we mean that any well-formed equation of the form $\sigma \circ \phi = \phi$, where σ is constructed of symmetries and identities, holds.

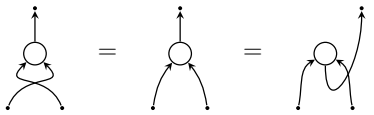


Under the assumption that all points are commutative we can replace points with a graph based notation removing the total order by drawing wires around a circular node.



(2.1)

The ordering of wires is completely meaningless in this new graphical notation. Hence the following are examples of intrinsically equal graphs:



This suggests that we are working with something combinatoric in nature. Diagrams come from a set of nodes and a set of directed edges.

Remark 2.2.2. Commutativity allows us to move edges representing the object X to the top of the node and those representing X^* to the bottom. Hence all nodes can be drawn as having input wires entering from below and output wires exiting above. Similarly, each morphism can be represented by a codomain of outputs X and a domain of inputs X (outputs of type X^*).

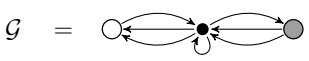
2.2.1 String Graphs

A common definition in graph theory is that of a directed graph, which is made up of a set of vertices and a set of ordered pairs of vertices called edges. This definition does not allow edges to be open at the ends as required for our diagrams. To replace the notion of edges with that of wires (allowing open ends acting as inputs/outputs) we use a form of directed, typed graph called a string graph ([12, 24]) which uses dummy vertices to encode wires. We refer to these dummy vertices as *wire-vertices* and they are assigned a type to represent the object type of that wire. Other vertices are referred to as *node-vertices* and are typed by the name of the morphism they represent. We represent a wire as a chain of wire-vertices. If the start/end of a chain is a wire-vertex then this represents an open input/output wire.

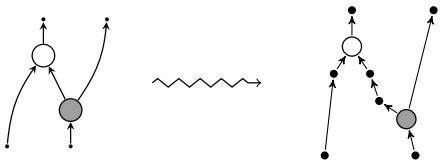
Typing of graphs is done by a graph morphism mapping vertices/edges to a *typegraph* which stores the possible types and restricts which can be connected by edges.

Definition 2.2.3 (Typed-Graphs)**.** A graph G along with a graph morphism $\tau : G \to \mathcal{G}$ is said to be a $\mathcal{G}$-typed graph.

Example 2.2.4. Take the following typegraph as an example:



This dictates that if a graph G is $\mathcal{G}$-typed then any white node must have two inputs and one output and any gray node must have one input and two outputs. The following demonstrates a $\mathcal{G}$-typed string graph encoding of a diagram:



Definition 2.2.5 (Monoidal Signature)**.** A (small, strict) *monoidal signature* is defined to be a 4-tuple $(O, M, \mathrm{dom}, \mathrm{cod})$ where O and M are sets of objects and morphisms respectively and $\mathrm{dom}, \mathrm{cod} : M \to O^*$ assign morphism with lists of objects as domain and codomain.

In the commutative case (which we assume to be in for the rest of this section) we need not worry about the order of elements in the lists and hence can take dom and cod to map into multisets over O. We generally write $\phi : [X_1, \ldots, X_n] \to [Y_1, \ldots, Y_m]$ for the morphism ϕ with domain $[X_1, \ldots, X_n]$ and codomain $[Y_1, \ldots, Y_m]$.

Definition 2.2.6 (Derived Typegraph)**.** Given a monoidal signature $T = (O, M, \mathrm{dom}, \mathrm{cod})$, the derived typegraph $\mathcal{G}_T$ has vertices $O \uplus M$, a self loop for each $X \in O$, an edge $X \to \phi$ for each $X \in \mathrm{dom}(\phi)$, and an edge $\phi \to X$ for each $X \in \mathrm{cod}(\phi)$.

Example 2.2.7. The monoidal signature T with morphisms $\phi : [X, X] \to [X]$ and $\psi : [X, Y] \to [X]$ has derived typegraph $\mathcal{G}_T$ of the form:

Given monoidal signature $T = (O, M, \mathrm{dom}, \mathrm{cod})$, a $\mathcal{G}_T$-typed graph (G, τ) has a set $W(G) := \{v \in G : \tau(v) \in O\}$ of wire-vertices and a set $N(G) := \{v \in G : \tau(v) \in M\}$ of node-vertices.

Definition 2.2.8 (String Graph)**.** Given a monoidal signature T, a $\mathcal{G}_T$-typed graph (G, τ) is called a *string graph* if τ is arity matching for node-vertices (a bijection between the neighbourhoods of v and $\tau(v)$ for each $v \in N(G)$) and each wire-vertex has at most one incoming edge and at most one outgoing edge.

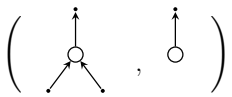
These restriction ensure that node vertices have their correct arities and wire vertices are connected in chains without 'splitting' (since each chain represents a single wire). A wire point with no incoming edge represents an input to the graph and a wire point with no outgoing edge represents an output. We refer to the input vertices and output vertices together as the boundary. The set of wire-vertices in a wire are referred to as the interior of that wire.

Since a chain of wire-vertices is supposed to represent a wire, and we only care about connectivity of diagrams, the number of wire-vertices in a chain has no meaning. We refer to this equivalence as *wire homeomorphism*. Two string graphs are wire homeomorphic if they are equivalent up to the number of wire vertices in each chain. The following diagram demonstrates two wire homeomorphic string graphs:

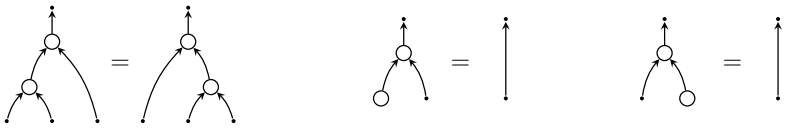
2.2.2 Frobenius Algebras

Frobenius algebras are one example of the advantages of switching to a graphical language. They have been used extensively in the area of categorical quantum mechanics. To define the theory of a commutative Frobenius algebra we first present the theory of a commutative monoid.

A commutative monoid is a commutative associative binary operation with a unit. We can represent it graphically by having two generating morphisms:

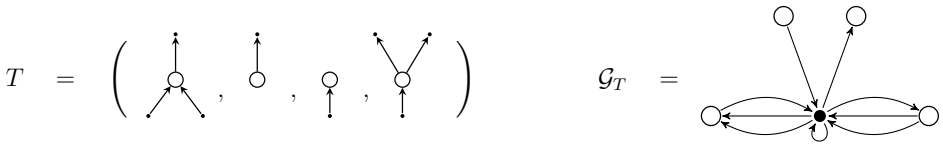


along with three diagrammatic equations:

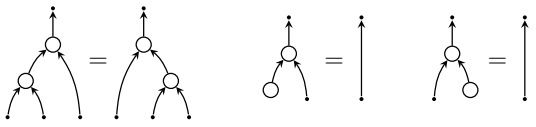


Monoid Laws

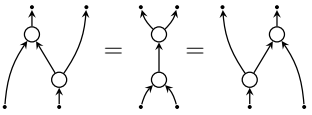
The theory of a commutative Frobenius algebra can be described by the combination of a commutative monoid and a cocommutative comonoid (similar to a monoid except with all diagrams flipped upside down) along with the Frobenius law governing their interaction. Hence the signature and typegraph are:



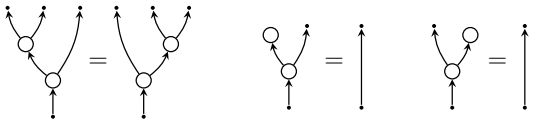
and we prescribe that they satisfy the axioms:



Monoid Laws



Frobenius Law



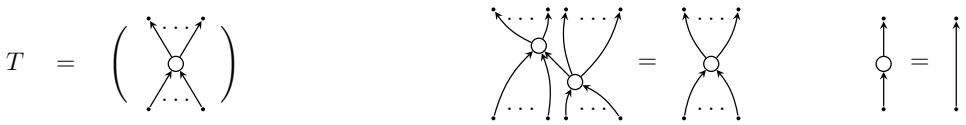
Comonoid Laws

2.3 Families of Diagrams and !-Graphs

Looking back at the definition of a monoidal signature we see that there is no restriction that the set of morphisms be finite. Suppose we have a theory which has infinitely many morphisms, one for each possible number of input edges and each having one output. We could write these as $\phi_0 : [] \to [X]$,

$\phi_1 : [X] \to [X]$, $\phi_2 : [X, X] \to [X]$, ... where we have indexed them based on number of inputs. Rather than working with each individually, we would like to be able to make statements covering all such ϕ_i. To do this we augment the graphical language to allow families of diagrams. Given the infinitely many fixed arity generators we can instead talk about a single *variable arity* generator ϕ. A graphical equation involving ϕ should then be considered to represent an infinite family of graphical equations.

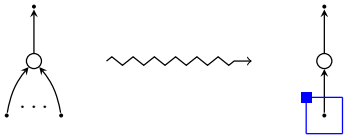
Take the definition of a commutative Frobenius algebra (CFA) as above, we see four generating morphisms and eight equalities. This can actually be replaced by the easier to understand arbitrary arity version. This involves a single generating node with an arbitrary number of inputs and an arbitrary number of outputs (so representing a family of nodes) and only two (families of) diagrammatic equations. The first, often referred to as a spider theorem, allows any two connected nodes to be combined; the second replaces any node with a single input and a single output with an identity wire.



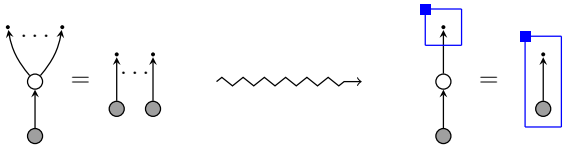
This notation better encapsulates the structure of a CFA by demonstrating that any connected nodes can simply be merged together. However, the ellipisis notation above is clearly not rigorous enough to be used as it is. Instead !-boxes were introduced in [13] and formalised in [26] to represent sections of a graph which can occur zero or more times resulting in a family of graphs.

2.3.1 !-Boxes

We annotate a section of a graph which we allow to be repeated by drawing a blue box called a !-box (pronounced bang-box) around it.

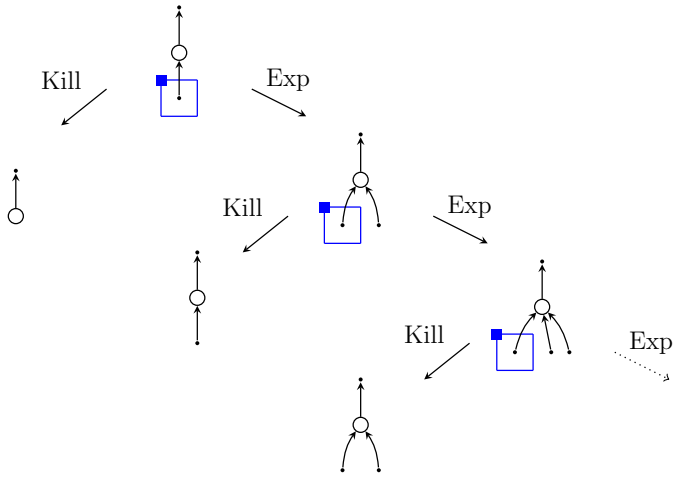


We refer to graphs with !-boxes as !-graphs. Families of equations can similarly be rewritten as !-graph equations.

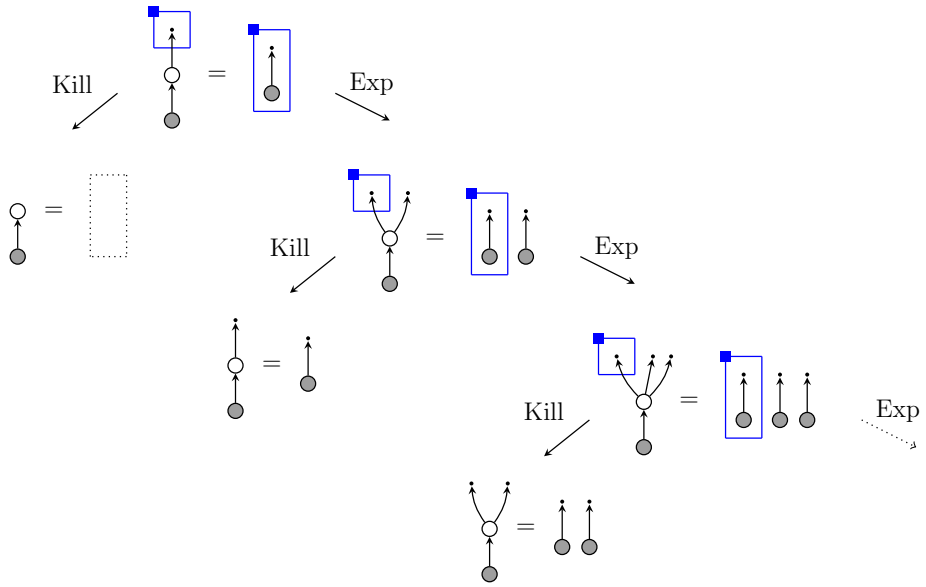


This equation states that a gray node with a single output wire is copied by our arbitrary arity white comultiplication. !-Graphs and !-graph equations should represent (probably infinite) families of *concrete* graphs and *concrete* graph equations respectively. To retrieve the concrete instances of

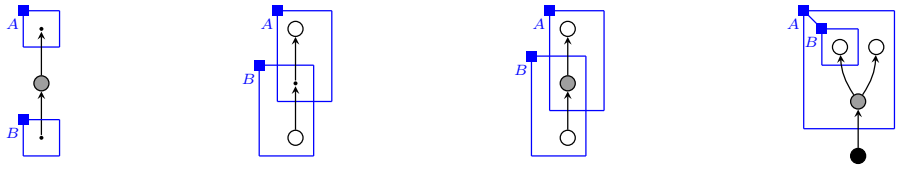
a !-graph or equation we require two !-box operations. The first is *killing* (denoted Kill) which is the operation removing the !-box and all its contents (including wires entering/leaving the !-box) from the graph. The second (denoted Exp) is *expanding*, which creates a new copy of the contents of the !-box and attaches it to the same surrounding nodes. For example, applying them to a !-graph with a single !-box around the input leads to the following tree with concrete graphs at the leaves.



Similarly, the !-box equation representing gray nodes being copied by white leads to a tree with concrete equations at the leaves.



For now we take the meaning of such a !-graph equation to be the set of all concrete instances retrieved by applying !-box operations in any order. Note that !-graphs can have multiple !-boxes (denoted via upper-case Latin alphabet) and they can interact in a few different ways.



In the first case the two !-boxes are in completely separate areas of the diagram, in the second and third they are said to overlap as they share some content and in the final case B is said to be nested in A (as indicated by the line between A and B). Nested means that when expanding the outer !-box a new copy of the inner !-box is created (with a new name). We now give a formal definition of !-graphs.

2.3.2 !-Graphs

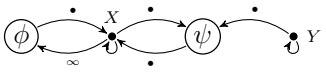
To define !-graphs we must first extend our signature to allow generating nodes with arbitrary arity and similarly extend our definition of a string-graph to match.

Definition 2.3.1 (Compressed Monoidal Signature)**.** A *compressed monoidal signature* is a 4-tuple $(O, M, \mathrm{dom}, \mathrm{cod})$ where O and M are sets of objects and morphisms respectively and $\mathrm{dom}, \mathrm{cod} : M \to (O \times \{\bullet, \infty\})^*$ assign morphism with lists of objects which are either single tagged $(\bullet)$ or variable tagged (∞).

Definition 2.3.2 (Derived Compressed Typegraph)**.** Given a compressed monoidal signature $T = (O, M, \mathrm{dom}, \mathrm{cod})$, the *derived compressed typegraph* $\mathcal{G}_T$ has vertices $O \uplus M$, a self-loop for every $X \in O$, and for every $\phi \in M$:

- one $\bullet$-edge from X to ϕ for each $(X, \bullet)$ in $\mathrm{dom}(\phi)$;
- one $\bullet$-edge from ϕ to X for each $(X, \bullet)$ in $\mathrm{cod}(\phi)$;
- one ∞-edge from X to ϕ for each (X, ∞) in $\mathrm{dom}(\phi)$;
- one ∞-edge from ϕ to X for each (X, ∞) in $\mathrm{cod}(\phi)$.

Example 2.3.3. The compressed monoidal signature T with the two morphisms $\phi : [X^\infty] \to [X^\bullet]$ and $\psi : [Y^\bullet, X^\bullet] \to [X^\bullet]$ results in a typegraph $\mathcal{G}_T$ of the form:



We continue to write $W(G)$ for the wire-vertices (those mapped into O) and $N(G)$ for the node-vertices (those mapped into M). The $\bullet$-tagged edges represent edges of fixed (single) arity whereas ∞-tagged edges represent edges of variable arity. This is important when we transition to !-graphs where we need to be careful not to copy single arity edges.

Definition 2.3.4 (String Graph)**.** A $\mathcal{G}_T$-typed graph (G, τ) is called a *string graph* if τ is arity-matching (a bijection between the $\bullet$-edge neighbourhoods of v and $\tau(v)$, for each $v \in N(G)$) and each wire-vertex in G has at most one incoming edge and at most one outgoing edge.

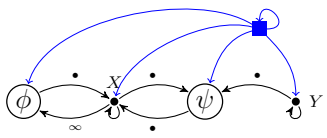
!-boxes denote subsections of a graph which can be repeated many times but typed graphs do not have a built in tool for highlighting such sections. We need to come up with a graph based representation of such a section. We choose to represent a !-box B as a node (of the new type !) with an edge to each vertex contained in B.

Definition 2.3.5 (Open)**.** A subgraph O of a string graph G is said to be *open* if it is not adjacent to any wire-vertices or incident to any fixed-arity edges.

Openness (as described in [39]) encapsulates the property of being able to repeat that part of a graph an arbitrary number of times. It ensures adjacent edges are copied with nodes; fixed arity edges are not copied without the adjacent node; and that wires cannot be partially inside a !-box (to avoid wire splitting).

Definition 2.3.6 (Derived Compressed !-Typegraph)**.** Given a compressed monoidal signature T, the *derived compressed !-typegraph* $\mathcal{G}_{T!}$ is $\mathcal{G}_T$ with the addition of a vertex ! along with edges from ! to every vertex (including itself).

Example 2.3.7. Our previous example of the compressed monoidal signature T with morphisms $\phi : [X^\infty] \to [X^\bullet]$ and $\psi : [Y^\bullet, X^\bullet] \to [X^\bullet]$ results in a !-typegraph $\mathcal{G}_{T!}$ of the form:



If G is $\mathcal{G}_{T!}$-typed, write $!(G)$ for the box-vertices in G (those mapped to !) and $U(G)$ for the full subgraph of G with all vertices except $!(G)$. Given $B \in !(G)$ let $C(B)$ be the full subgraph of G with nodes which have edges from B. So $C(B)$ represents the contents of a !-box B. We use this notation in the !-graph conditions to ensure !-boxes are well-behaved under !-box operations.

Definition 2.3.8 (!-Graph)**.** A $\mathcal{G}_{T!}$-typed graph G is called a *!-graph* if the following hold:

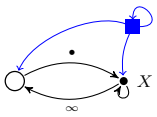
BG1. $U(G)$ is a $\mathcal{G}_T$-typed string graph;

BG2. the full subgraph with vertices $!(G)$ is posetal;

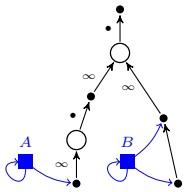
BG3. $\forall B \in !(G)$, $U(C(B))$ is an open subgraph of $U(G)$; and

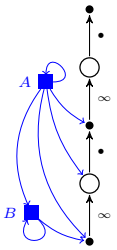
BG4. $\forall B, B' \in !(G)$, if $B' \in C(B)$ then $C(B') \subseteq C(B)$.

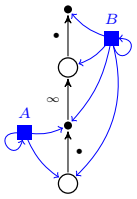
Example 2.3.9. As an example, the arbitrary arity signature for a monoid has a single morphism of type $[X^\infty] \to [X^\bullet]$ and hence the derived !-typegraph is as follows:



Some examples of valid !-graphs are:







We have so far mentioned two !-box operations for a !-box B, namely Kill_B and Exp_B as these are the only operations needed to get all concrete graphs from a !-graph. More generally there can be other valid operations to apply to !-boxes. We present four !-box operations by factorising Exp into the composition of two distinct operations. Drop_B is the operation removing the !-box B but leaving its contents and Copy_B is the operation creating a new copy of not only the contents of B but also B itself (with a new name). It follows that $\text{Exp}_B = \text{Drop}_{B'} \circ \text{Copy}_B$ where B' is the new name assigned to the copy of B created during Copy_B. Since Exp can be written in terms of the others we need only give definitions for Copy, Drop and Kill (similarly many theorems only require proof for this minimal set of operations).

Definition 2.3.10 (!-Box Operations)**.** Our three !-box operations are applied to a $\mathcal{G}_{T!}$-typed graph by:

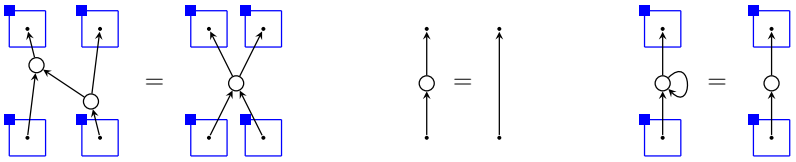
- $\text{Copy}_B(G)$ is defined by a pushout of inclusions in the category of graph morphisms between $\mathcal{G}_{T!}$-typed graphs:

$$\begin{array}{ccc} G\backslash C(B) & \hookrightarrow & G \\ \downarrow & & \downarrow \\ G & \longrightarrow & \text{Copy}_B(G) \end{array}$$

- $\text{Drop}_B(G) := G\backslash B$
- $\text{Kill}_B(G) := G\backslash C(B)$

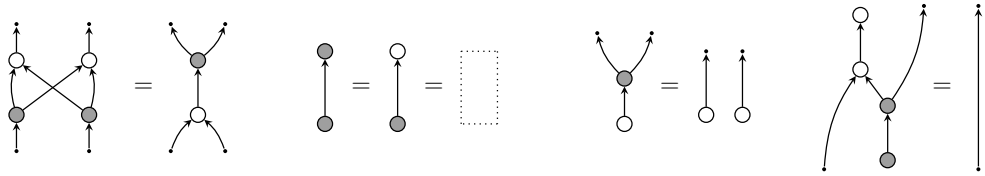
2.3.3 Z/X-Calculus

One particularly prevalent example of the use of !-graphs is in the Z/X-Calculus [5]. We present the phase-free Z/X-Calculus which starts with two †-special commutative Frobenius algebras (†SCFA's). The rules of a †SCFA can be presented using arbitrary arity nodes as:



The first equation, often referred to as a spider diagram, allows us to merge any two connected nodes; the second replaces nodes with a single input and a single output with an identity edge; and the third removes loops.

If we draw one †SCFA (called Z) as white nodes and another (called X) as gray nodes and add some additional rules governing their interaction then we get the phase-free Z/X-calculus. This represents two complementary observables and has applications in quantum information theory. The additional rules are the following four along with any vertically flipped and/or colour flipped versions:

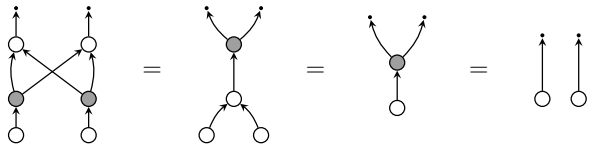


Other similar examples of graphical theories in the area of quantum information theory are the trichromatic calculus [36] and the GHZ/W-calculus [9, 10] both of which are also based on commutative Frobenius algebras. We will work more with Frobenius algebras in chapter 6.

2.4 Rewriting

Reasoning with string-graphs and !-graphs is done via rewriting. If we have an equation $L = R$ then we can choose a direction and decide that whenever L appears as a section of a diagram it should be replaced with R.

Example 2.4.1. Using the Z/X-calculus axioms above we can use rewriting to simplify large diagrams:

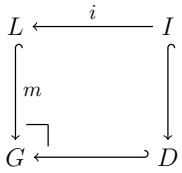


For string-graph rewriting this substitution is implemented via the double-pushout (DPO) graph rewriting method [35, 42]. Here we take a rewrite rule to be a span of graphs,

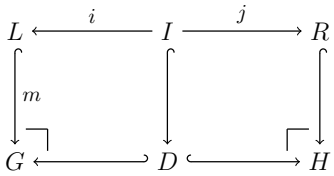
$$L \xleftarrow{\ i\ } I \xrightarrow{\ j\ } R$$

where I is the interface of the rewrite rule (can be thought of as holding the boundary in place as we remove/replace sections of a diagram). The interpretation is that we want to replace instances of L with instances of R.

Definition 2.4.2 (String-Graph Matching)**.** Given the above rule, a matching $m : L \to G$ is an embedding of L in G such that we can find a pushout complement for $m \circ i$:

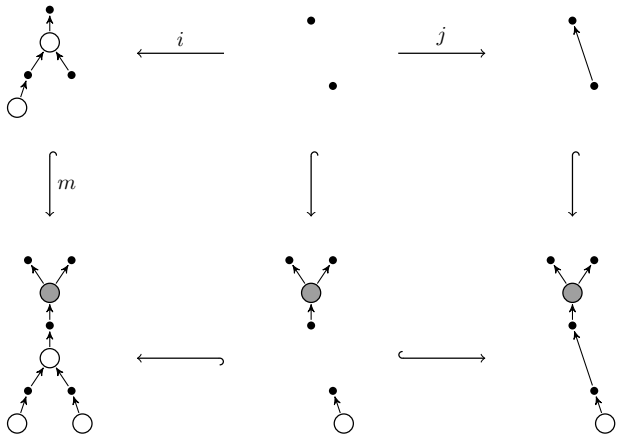


Rewriting G to a new string-graph H is then done by an additional pushout:



from which we conclude $G = H$.

Here is a graphical representation of such a double pushout demonstrating the second equality in example 2.4.1:



In [39] a similar method is described for matching, allowing !-boxes where we may have to apply some !-box operations to the rule before we get a matching.

Chapter 3

!-Tensors

This chapter presents the first original contribution of this thesis, *!-tensors*. !-Tensors remove the restriction of commutativity from the !-graph formalism while still allowing families of diagrams via !-boxes. We start by defining a language for *concrete* tensor diagrams which are those without !-boxes and then extend to allow families of concrete tensors. Along the way we present a few examples of noncommutative theories and demonstrate how noncommutativity allows for recursively defined nodes. We conclude the chapter by demonstrating how !-tensors could be encoded as !-graphs by adding some additional data to the nodes. This is of particular use for implementing !-tensors in Quantomatic [27].

3.1 Tensors

3.1.1 Tensor Diagrams

We continue to work in a compact closed category $\mathcal{C}$ and hence allow planar diagrams with wire crossing and feedback loops. Recall from Section 2.2 that all morphisms can be written as points (morphisms from the tensor unit I). Hence morphisms are of the form $\phi : I \to X \otimes Y^* \otimes \ldots \otimes Y$ with the tensor unit as domain and a tensor product over the generating object types and their duals as codomain.

Recall that a monoidal signature (definition 2.2.5) describes a commutative theory by listing (possibly infinitely many) morphisms, each with a domain and a codomain as words over the generating object types. This ability to split inputs/outputs into a domain/codomain respectively relies on commutativity of the points. For !-tensors to allow noncommutativity, we introduce *compact closed signatures* which generalise monoidal signatures. A compact closed signature describes a set of generating points $\phi : I \to X \otimes Y^* \otimes \ldots \otimes Y$ by listing them as pairs $(\phi, XY^* \ldots Y)$ comprising the morphism name and its codomain written as a word over the generating object types and their

duals. We often write such a morphism in the more intuitive function notation. For example, a binary operation on X is most commonly written $\mu : X \otimes X \to X$ but, using dual types, will be recorded in a compact closed signature as (μ, XX^*X^*).

Definition 3.1.1 (Compact Closed Signature)**.** A *compact closed signature* Σ consists of a set of objects $\mathcal{O} := \{X, Y, \ldots\}$ and a set $\mathcal{M}$ of pairs (ϕ, W), where W is a word in $\{X, X^*, Y, Y^*, \ldots\}$, which we refer to as the morphisms.

Examples 3.1.2. These are the signatures of some of the theories we will revisit throughout this thesis.

- The compact closed signature for a monoid has a single object type X and two morphisms (μ, XX^*X^*) and (η, X) which we refer to as the multiplication and unit respectively. These are more commonly written by the alternative notation:

$$\mu : X \otimes X \to X$$
$$\eta : I \to X$$

  We favour the latter notation for clarity of reading and continue it, where possible, for the remainder of this thesis.

- Adding a homomorphism (or antihomomorphism) to a monoid means adding one additional morphism $\theta : X \to X$ to the signature.

- A bialgebra on X has signature consisting of multiplication, unit, comultiplication, and counit morphisms:

$$\mu : X \otimes X \to X \qquad \delta : X \to X \otimes X$$
$$\eta : I \to X \qquad \epsilon : X \to I$$

- Signatures can be infinite. For example, we will later see how the fixed arity signature of monoids as above can be replaced by one containing a single arbitrary arity generator:

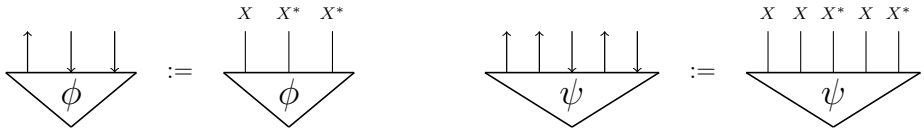
$$\phi_i : X^{\otimes i} \to X \qquad i \in \mathbb{N}$$

  Here ϕ represent a family of morphisms indexed by the number of inputs, each having a single output. This represents the arbitrary arity version of the multiplication μ seen in the monoid signature.

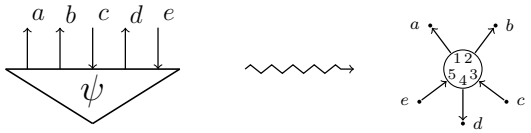
Definition 3.1.3 (Valuation)**.** A *valuation* $[\![-]\!] : \Sigma \to \mathcal{C}$ is a choice of object in $\mathcal{C}$ for each $X \in \mathcal{O}$, and a choice of point $[\![\psi]\!] : I \to X_1 \otimes X_2^* \otimes \ldots \otimes X_n$ for each $(\psi, X_1 X_2^* \ldots X_n) \in \mathcal{M}$.

Given a signature $\Sigma = (\mathcal{O}, \mathcal{M})$ we work in the *free compact closed category* $\mathrm{Free}(\Sigma)$ generated from the objects $\mathcal{O}$ and morphisms $\mathcal{M}$ along with the building blocks of a compact closed category (cups, caps, swaps, identities, $\otimes$, $\circ$) modulo the compact closed axioms. We can uniquely lift any valuation $[\![-]\!] : \Sigma \to \mathcal{C}$ to a functor $[\![-]\!] : \mathrm{Free}(\Sigma) \to \mathcal{C}$.

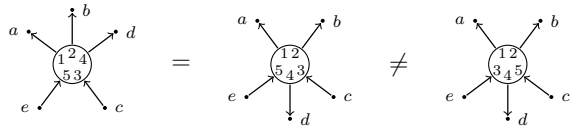
For simplicity we will ignore typing of edges, assuming that there is only a single object type X with dual X^* in our signature. Hence we can drop types from the diagrams by having objects of type X drawn as wires leaving a point and objects of type X^* as wires entering the point.



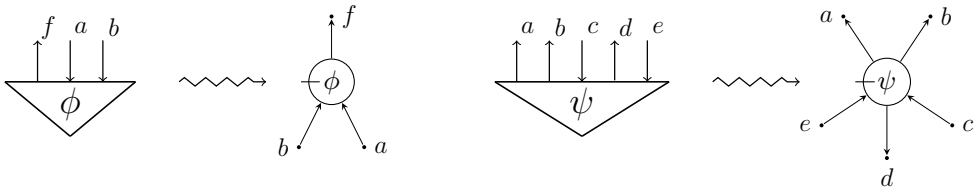
We hope to imitate the transition seen in (2.1) taking points to their graphical notation of nodes. However in this case we do not wish to forget the total order on wires. To keep this information we need to track which wires are considered to be the first, second, etc. We could do this by annotating wires on nodes with numbers as shown in the following diagram:



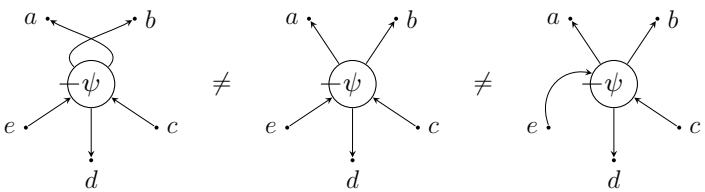
This allows us to commute edges as is possible in the !-graph formalism, so long as we commute the numbers too. But the notation is difficult to write and it is slow to parse.



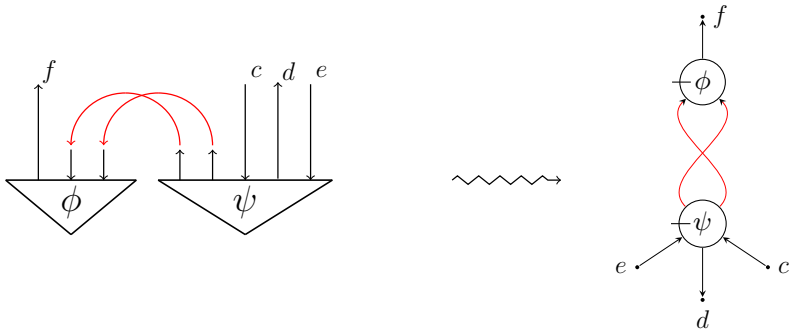
A better notation would be to ban edges from passing each other and count off edges starting from a designated first edges and then following clockwise. We do this by adding a tick to our node, then counting edges clockwise from the tick as first, second, third, and so on.



Note that in the graph case we would usually move the output wire labelled d to the top but that is no longer a valid deformation as we cannot change the order of edges.



Plugging of wires works as expected, caps between objects of type X and X^* connect their corresponding edges:

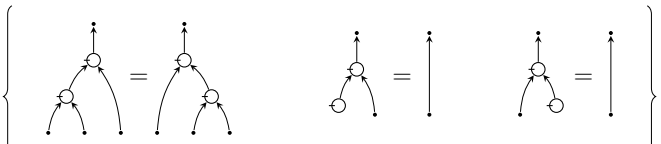


(3.1)

We refer to these new diagrams as tensors (for reasons we will see shortly). Note that we need to name the individual inputs/outputs to distinguish between them (which we do using lower case letters). We will now drop the point notation and switch to these tensor diagrams.

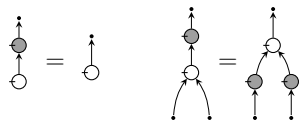
Examples 3.1.4. We can now specify a theory by a signature and some tensor diagram axioms. For the signatures from Example 3.1.2 we can specify corresponding laws:

- The multiplication and unit of a monoid can be written graphically as ( , ). This allows us to replace the equational axioms $\mu \circ (\mu \otimes 1_X) = \mu \circ (1_X \otimes \mu)$ and $\mu \circ (\eta \otimes 1_X) = 1_X = \mu \circ (1_X \otimes \eta)$, called associativity and unit respectively, with their more intuitive diagrammatic counterparts:

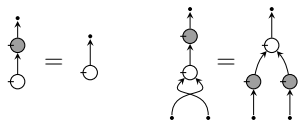
Γ_M = 

Monoid Laws

- Adding  to the monoid above we can get either the theory of homomorphisms or antihomomorphisms on a monoid depending on which of the following axioms we take:



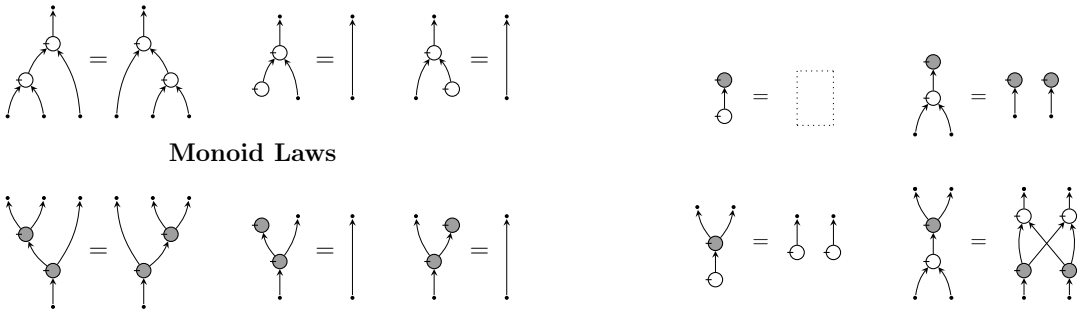
Homomorphism Laws



Antihomomorphism Laws

These are $\theta \circ \eta = \eta$ and $\theta \circ \mu = \mu \circ (\theta \otimes \theta)$ for the homomorphism case and $\theta \circ \eta = \eta$ and $\theta \circ \mu \circ \sigma_{X,X} = \mu \circ (\theta \otimes \theta)$ for the antihomomorphism case (where σ is the swap map).

- A bialgebra is made up of a monoid ( , ) and a comonoid ( , ) along with four axioms governing their interaction:



Monoid Laws

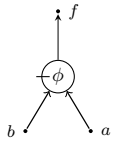
Comonoid Laws

Bialgebra Laws

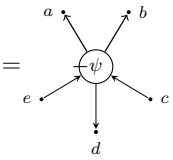
3.1.2 Tensor Notation

Clearly the previous !-graph formalism is unsuitable for tensors, as its combinatoric nature does not store edge order. Instead we will present a syntactic formalism for tensors called tensor expressions which extend Penrose's Abstract Tensor Notation [40]. Nodes are written as their type with a subscript listing edges in clockwise order from the tick. To track whether edges are outputs or inputs we add a hat to outputs $\{\hat{a}, \hat{b}, \ldots\}$ and a check to inputs $\{\check{a}, \check{b}, \ldots\}$.

$\phi_{\hat{f}\check{a}\check{b}}$ =

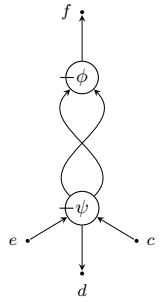


$\psi_{\hat{a}\hat{b}\check{c}\hat{d}\check{e}}$ =



To combine two tensors, as seen in (3.1), the two tensor expressions are written next to each other. Hence the two examples above combine to form:

$\phi_{\hat{f}\check{a}\check{b}}\psi_{\hat{a}\hat{b}\check{c}\hat{d}\check{e}}$ =



The outputs $\hat{a}, \hat{b}$ have been plugged into the inputs $\check{a}, \check{b}$ via contraction of repeated names. Hence a and b do not appear in the tensor diagram notation as they are implicitly bound. This demonstrates the fact that tensor expressions are not unique representations of their corresponding diagrams. After defining tensor expressions we will define an equivalence relation on them which encapsulates the property of representing the same diagram. For example, the two expressions $\phi_{\hat{f}\check{a}\check{b}}\psi_{\hat{a}\hat{b}\check{c}\hat{d}\check{e}}$ and $\phi_{\hat{f}\check{g}\check{h}}\psi_{\hat{g}\hat{h}\check{c}\hat{d}\check{e}}$ are equivalent as they are the same up to renaming bound names (a.k.a. α-conversion).

We need to add a couple of extra generators to our tensor expression language to take care of some special cases. We will write $1_{\hat{a}\check{b}}$ for the single wire from input b to output a and 1 for the empty diagram.

Definition 3.1.5 (Tensor Expression)**.** The set of *tensor expressions* $\mathcal{T}_\Sigma$ for a compact closed signature Σ consists of

- 1
- $1_{\hat{a}\check{b}}$
- $\phi_{\hat{a}\ldots\check{c}}$ $\quad (\phi, X \ldots X^*) \in \Sigma$
- GH $\quad G, H \in \mathcal{T}_\Sigma$

under the condition that neither $\hat{a}$ nor $\check{a}$ occur more than once for any edge name a.

The condition on $\phi_{\hat{a}\ldots\check{c}}$ ensures the arrangement of input/output names matches the possible arities of ϕ in Σ. For example, if $\Sigma = \{(\phi, X), (\phi, XX^*X^*)\}$ then $\phi_{\hat{a}}$ and $\phi_{\hat{c}\check{a}\check{b}}$ are valid but $\phi_{\check{a}\hat{b}\check{c}}$ is not as it requires the morphism (ϕ, X^*XX^*).

Definition 3.1.6 (Tensor Equivalence)**.** Two tensor expressions are equivalent if one can be transformed into the other by any number of α-conversions and identities:

$$(GH)K \equiv G(HK) \qquad GH \equiv HG \qquad G1 \equiv G$$

$$G1_{\hat{b}\check{a}} \equiv G[\check{b} \mapsto \check{a}] \qquad H1_{\hat{a}\check{b}} \equiv H[\hat{b} \mapsto \hat{a}]$$

The first three are associativity, commutativity, and unit rules for the tensor product. Assume for the last two identities that $\check{b}$ and $\hat{b}$ are free in G and H, respectively. These two equivalences demonstrate that plugging an edge into an identity morphism is the same as renaming the edge. We write $G \equiv G'$ to mean G and G' are equivalent tensor expressions.

Remark 3.1.7. We allow equivalent tensor expressions to be substituted for each other in the sense that if $G \equiv H$ then also $GK \equiv HK$ (assuming both expressions are well-formed).

Definition 3.1.8 (Tensor)**.** From definition 3.1.6 we see that $\equiv$ is an equivalence relation. We define a *tensor* to be an $\equiv$-equivalence class of tensor expressions (i.e. a tensor for signature Σ is an element of $\mathcal{T}_\Sigma / \equiv$).

Definitions 3.1.9 (Free/Bound Edges)**.** For a tensor expression G we write free(G) for the set of *free edges* in G (names appearing exactly once in G), we write bound(G) for the set of *bound edges* (names appearing as both $\hat{a}$ and $\check{a}$) and the union of these is edges(G). We naturally extend free($-$) to tensors as it is independent of choice of expression (whereas bound($-$) is not well defined for tensors).

3.1.3 Interpretation

Having defined tensors for a signature Σ, an obvious question is how they relate to morphisms in Free(Σ). We might hope that each morphism has a unique representation as a tensor, but this is

not exactly the case. There is a mismatch between the two formalisms in how they track free wires. Morphisms in Free(Σ) use *position* to refer to wires whereas tensors assign *names* to them. To relate these different approaches we define the notion of being a *canonically named* tensor. Let $\{a_1, a_2, \ldots\}$ be a fixed infinite set of *canonical names*.

Definition 3.1.10 (Canonically Named Tensor)**.** A tensor is *canonically named* if for some $n \in \mathbb{N}$ it has free names exactly one of $\hat{a_i}$ or $\check{a_i}$ for each $i \leq n$.

Theorem 3.1.11. *Canonically-named tensors are in 1-to-1 correspondence with points in* Free($\mathcal{C}$)*.*

Proof. To prove this, we describe the construction depicted in (3.1) in more detail. This technique is very similar to the one employed in [25] but simpler in the compact closed case. First, we interpret a tensor expression as a morphism in the free compact closed category.

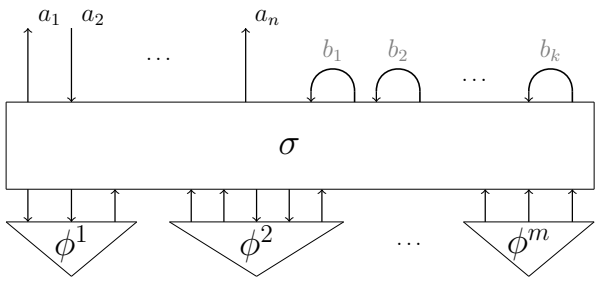
Start with a (canonically named) tensor expression with free names $a_1, \ldots, a_n$ and write $b_1, \ldots, b_k$ for the bound names. The expression is of the form $\phi^1_{\ldots}\phi^2_{\ldots}\ldots\phi^m_{\ldots}$ where $\phi^1, \ldots, \phi^m$ are each generating morphisms, identity wires or the identity tensor. Then, we can interpret the tensor expression as:

$$\phi^1_{\ldots}\phi^2_{\ldots}\ldots\phi^m_{\ldots} \qquad \rightsquigarrow \qquad (1_{a_1} \otimes 1_{a_2} \otimes \ldots \otimes 1_{a_n} \otimes \epsilon_X \otimes \epsilon_X \otimes \ldots \otimes \epsilon_X) \circ \sigma \circ (\phi^1 \otimes \phi^2 \otimes \ldots \otimes \phi^m)$$

where:

1. 1_{a_i} is 1_X if the expression contains a free output $\hat{a_i}$ and 1_{X^*} if it contains a free input $\check{a_i}$,
2. the i-th cap ϵ_X corresponds to the bound name b_i,
3. ϕ^i is either the appropriate generator from Σ, η_X for an identity wire or 1_I for the tensor expression I,
4. σ is the (unique by naturality) map consisting just of symmetries and identities that connects the inputs/outputs of the ϕ^i associated with a given name to the appropriate identity or cap.

We can draw this graphically as:



(3.2)

This shows that a tensor expression uniquely determines a point in the free category. Since a tensor is an equivalence class of tensor expressions, we need to show that this does not depend on the choice

of expression. We can safely ignore bracketing and instances of the empty tensor 1, so if $G \equiv H$ it must be from a number of the following differences (i) the order of ϕ^1 to ϕ^m, (ii) the choice of bound names, or (iii) the number of identity tensors. In each of these cases, we use the axioms of a compact category to show that interpretations of G and H are equal.

For (i), we can use naturality of σ to reshuffle the generators at the bottom, without affecting connectivity. For (ii), we do the same for the caps at the top. For (iii), we can use the compactness equations:

to insert or remove identities, i.e. cups, from the bottom.

Conversely, any expression in the free category can be written in the form of (3.2), at which point one can read off the tensor expression. First, use bifunctoriality to pull all generators and cups to the bottom and all caps to the top. Then, use naturality to sort all of the caps to the right of the output wires. The only freedom is in the order of the generators/cups and the caps, which is captured by $\equiv$. □

Corollary 3.1.12. *For any compact closed signature Σ, a valuation $[\![-]\!] : \Sigma \to \mathcal{C}$ lifts uniquely to an operation which sends canonically named tensors $G \in \mathcal{T}_\Sigma$ to morphisms $[\![G]\!]$ in $\mathcal{C}$.*

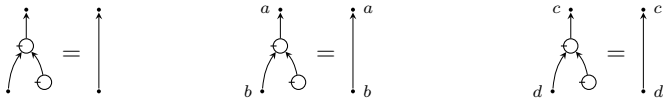
3.1.4 Rewriting

In examples 3.1.4 we have seen diagrammatic theories where we have equations between tensor diagrams. We would like to formalise these and use the graphical axioms to demonstrate equalities of more complicated diagrams. To talk about an equality between tensors we need each side to have the same free names.

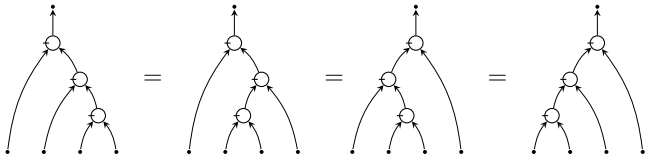
Definition 3.1.13 (Tensor Equation)**.** A tensor equation $G = H$ is *well-formed* if tensors G and H have the same free inputs and the same free outputs.

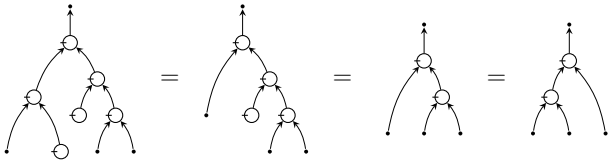
For example, the antihomomorphism law can be written as a tensor equations as $\theta_{\hat{a}\check{d}}\mu_{\hat{d}\check{b}\check{c}} = \mu_{\hat{a}\check{d}\check{e}}\theta_{\hat{d}\check{c}}\theta_{\hat{e}\check{b}}$ as each side has free names $\{\hat{a}, \check{b}, \check{c}\}$. When drawing these out as diagrams we tend to drop the names in favour of tracking wires by their position:

We have already seen this in examples 3.1.4 where the input and output wires have not been named but can be matched between the sides by looking at their position. This is allowed since names do not inherently store any information other than to keep track of open ended wires. So the following equations are all considered to mean the same thing:



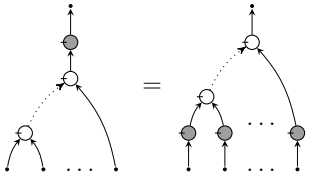
We can use axioms to rewrite more complicated diagrams by substituting one side for another. For example, the axioms of a monoid can be used as in the following rewriting examples:



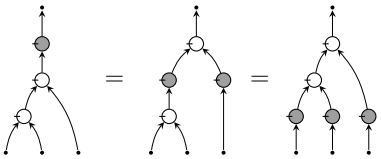


In both cases the diagrams have been rewritten into a normal form of a left associated tree of multiplications.

Conjecture 3.1.14. Homomorphisms can pass through arbitrarily sized left associated trees of multiplications. As suggested by the following illustration:



Rewriting allows us to prove this for any fixed size:



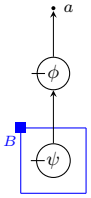
However, we do not currently have the tools to even state the theorem without resorting to ellipses. We now present !-boxes, which allow us to talk about such families of diagram equations. We give the formal statement of the above conjecture in (3.6).

3.2 !-Tensors

3.2.1 !-Tensor Diagrams

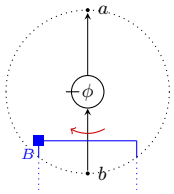
Now that we have non-commutativity we want to allow families of diagrams by adding !-boxes. Graphically we add !-boxes (as for !-graphs) by drawing a blue box around the section of a graph we allow to be repeated. In the syntax we will enclose any nodes inside a !-box B in square brackets with a superscript B, i.e. $[-]^B$.

$\phi_{\hat{a}\check{b}}[\psi_{\check{b}}]^B \quad =$



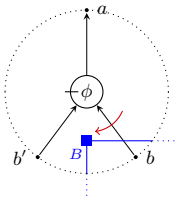
However, it turns out that this diagram is ambiguous. Recall from Section 2.3 that we wish to have an expanding operation which creates a new copy of the contents of B attached to the same nodes. In the !-graph case this was well defined, since position of edges around a node was unimportant. In !-tensor diagrams we need to be more precise. Expanding B in the diagram above, we have options as to where we want the new copy of the bound wire (named b in our expression) to be attached to ϕ. To decide where to attach new copies of edges we add arcs either pointing clockwise or anticlockwise. A clockwise arc means the new copy of b is added to the right in the subscript so we write this as $\hat{a}[\check{b}\rangle^B$. Similarly, anticlockwise expansion is written $\hat{a}\langle\check{b}]^B$ so that new copies are added left, i.e. earlier in the clockwise order. Take the following examples of expansion for a single edge:

$\phi_{\hat{a}[\check{b}\rangle^B} \quad =$

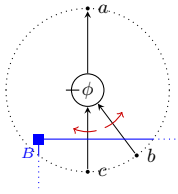


Exp_B

$\phi_{\hat{a}[\check{b}\rangle^B\check{b}'} \quad =$

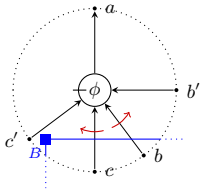


$\phi_{\hat{a}\langle\check{b}]^B[\check{c}\rangle^B} \quad =$



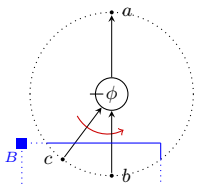
Exp_B

$\phi_{\hat{a}\check{b}'\langle\check{b}]^B[\check{c}\rangle^B\check{c}'} \quad =$



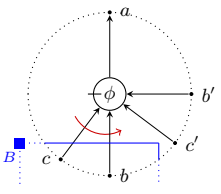
Edges can be grouped so that new copies are expanded together.

$\phi_{\hat{a}\langle\check{b}\check{c}]^B} \quad =$



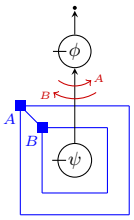
Exp_B

$\phi_{\hat{a}\check{b}'\check{c}'\langle\check{b}\check{c}]^B} \quad =$



Since !-boxes can be nested we may have multiple arcs on an edge.

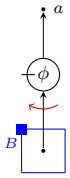
$\phi_{\hat{a}[\langle\check{b}]^A\rangle^B[[\psi_{\hat{b}}]^A]^B}$ =



We have labelled the arcs to demonstrate which !-boxes each corresponds to, but this is not necessary by deciding on the convention that arcs for outer !-boxes are always drawn closer to the node.

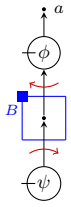
A !-box B might not contain any nodes but still contain edges. We decide that this should always be written explicitly by including $[1]^B$ in the expression. This can be seen in the example of a node taking an arbitrary number of inputs and producing a single output.

$\phi_{\hat{a}[\check{b}\rangle^B}[1]^B$



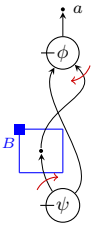
We can immediately take advantage of arcs to describe diagrams involving twisting edges.

$\phi_{\hat{a}[\check{b}\rangle^B\psi_{[\hat{b}\rangle^B}[1]^B$



Exp_B

$\phi_{\hat{a}[\check{b}\rangle^B\check{b}'\psi_{[\hat{b}\rangle^B\hat{b}'}[1]^B$



3.2.2 !-Tensor Expressions

To be precise about !-tensor expressions we will introduce sets of allowed names. We fix disjoint infinite sets $\mathcal{B} = \{A, B, \ldots\}$ and $\mathcal{N} = \{a, b, \ldots\}$ of !-box names and edge names respectively. We often need to refer to directed edge names which we write as $\overline{\mathcal{N}} := \{\hat{a}, \check{a} : a \in \mathcal{N}\}$. Since the subscripts in !-tensor expressions are more than just lists of inputs and outputs as in the tensor case, we give them a formal definition and refer to them as edgeterms.

Definition 3.2.1 (Edgeterm)**.** The set $\mathcal{E}$ of *edgeterms* over our fixed names is defined recursively as:

• $\epsilon \in \mathcal{E}$	(empty edgeterm)
• $\hat{a}, \check{a} \in \mathcal{E}$	$a \in \mathcal{N}$
• $[e\rangle^B, \langle e]^B \in \mathcal{E}$	$e \in \mathcal{E},\ B \in \mathcal{B}$
• $ef \in \mathcal{E}$	$e, f \in \mathcal{E}$

Two edgeterms are equivalent if one can be transformed into the other by:

$$e(fg) \equiv (ef)g \qquad \epsilon e \equiv e \equiv e\epsilon \qquad [\epsilon\rangle^B \equiv \epsilon \equiv \langle\epsilon]^B \tag{3.3}$$

The first two represent associativity and identity of the product, with ϵ as the unit. The last equivalence says that empty edge groups can be ignored.

Remark 3.2.2. We have seen how Exp_B acts on edgeterms in the diagrams of Section 3.2.1. The other !-box operation we want is Kill_B which acts as $\big[[e\rangle^B \mapsto \epsilon, \langle e]^B \mapsto \epsilon\big]$, i.e. removing any edges entering the !-box B by replacing them with the empty edgeterm. From these we can retrieve all possible concrete lists of edges. We will write $\phi_e \in \Sigma$ to say that all possible concrete instances of e have a generator $(\phi, W) \in \Sigma$ of the correct type.

To simplify the definition of !-tensor expressions we start by defining !-pretensor expressions which include ill-formed expressions. We then give the well-formedness conditions in definition 3.2.5.

Definition 3.2.3 (!-Pretensor Expression)**.** The set of *!-pretensor expressions* $\mathcal{T}'_{\Sigma!}$ for a signature Σ is defined recursively as:

- $1 \in \mathcal{T}'_{\Sigma!}$
- $1_{\check{a}\check{b}} \in \mathcal{T}'_{\Sigma!}$ $\quad a, b \in \mathcal{N}$
- $\phi_e \in \mathcal{T}'_{\Sigma!}$ $\quad e \in \mathcal{E}, \phi_e \in \Sigma$
- $[G]^B \in \mathcal{T}'_{\Sigma!}$ $\quad G \in \mathcal{T}'_{\Sigma!},\ B \in \mathcal{B}$
- $GH \in \mathcal{T}'_{\Sigma!}$ $\quad G, H \in \mathcal{T}'_{\Sigma!}$

In order to write the conditions by which a !-pretensor expression is a !-tensor expression we need to refer to which !-boxes surround particular directed edges. We refer to this list of !-boxes (ordered by nesting, inner !-boxes first) as the edge's *context*. Contexts come in two forms, *edge contexts* and *node contexts*.

Definition 3.2.4 (Context)**.** Given a directed edge $a \in \overline{\mathcal{N}}$ in a !-pretensor expression G nested as $[[\phi_{<<a>^{E_1}\ldots>^{E_n}}]^{N_1}\ldots]^{N_m}$ where by $\langle -\rangle$ we mean either clockwise expansions $[-\rangle$ or anticlockwise expansions $\langle -]$.

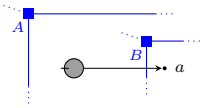
We define the *edge context*, *node context*, and *context* of a respectively as:

$$\begin{aligned}
\mathrm{ectx}_G(a) &:= [E_1, \ldots, E_n] && \text{(edge context)}\\
\mathrm{nctx}_G(a) &:= [N_1, \ldots, N_m] && \text{(node context)}\\
\mathrm{ctx}_G(a) &:= \mathrm{ectx}_G(a).\,\mathrm{nctx}_G(a) && \text{(context)}
\end{aligned}$$

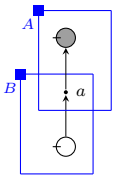
That is, $\mathrm{ectx}_G(a)$ lists the !-boxes containing a that occur as part of a's edgeterm, and $\mathrm{nctx}_G(a)$ lists the rest.

We will write $A \prec_G B$ to mean that the !-box A is nested immediately inside B in G (i.e. without other !-boxes nested between). The reflexive transitive closure of $\prec_G$ is written $\preccurlyeq_G$.

For example, if $\mathrm{ectx}_G(\hat{a}) = [B, \ldots]$ and $\mathrm{nctx}_G(\hat{a}) = [A, \ldots]$ then we know that the node containing $\hat{a}$ is nested inside $A, \ldots$ (and hence so is $\hat{a}$) and the edge $\hat{a}$ additionally enters $B, \ldots$.



This does not require $B \preccurlyeq_G A$, as it could be that A and B simply overlap on the edge a:



Definition 3.2.5 (!-Tensor Expression)**.** The set of *!-tensor expressions* $\mathcal{T}_{\Sigma!}$ for a signature Σ is the set of expressions $G \in \mathcal{T}'_{\Sigma!}$ satisfying the following conditions:

F1. $\hat{a}$ and $\check{a}$ occur at most once for each edge name a

F2. $[-]^A$ occurs at most once for each !-box name A

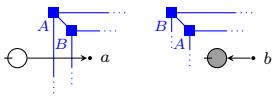
C1. $\mathrm{ectx}_G(a) \cap \mathrm{nctx}_G(a) = \varnothing$ for directed edges $a \in \overline{\mathcal{N}}$ in G

C2. If $\mathrm{ectx}_G(a) = [B_1, \ldots, B_n]$ then $B_1 \prec_G B_2 \prec_G \ldots \prec_G B_n$ and if $B_n \prec_G C$ then $C \in \mathrm{nctx}_G(a)$

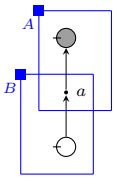
C3. For all bound pairs $\check{a}, \hat{a}$ of edge names in G, there exist lists es, bs of !-box names such that:

$$es.\,\mathrm{nctx}_G(\check{a}) = \mathrm{ectx}_G(\hat{a}).bs \quad \text{and} \quad es.\,\mathrm{nctx}_G(\hat{a}) = \mathrm{ectx}_G(\check{a}).bs$$

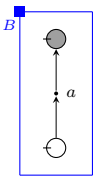
The freshness conditions F1 and F2 ensure that we have not used the same name for more than one (directed) edge or !-box. If a node is in !-box B then any edges attached to it are already in B so it would not make sense to have B in both the $\mathrm{ectx}(a)$ and $\mathrm{nctx}(a)$ for $a \in \overline{\mathcal{N}}$; this is enforced by C1. C2 ensures that edge contexts are compatible with the !-boxes in the rest of the !-tensor expression. For example, $\phi_{[[\hat{a}\rangle^B\rangle^A}$ requires B to be nested in A so does not result in a valid expression when composed with e.g. $[[\psi_{\check{b}}]^A]^B$. Drawing ϕ as a white node and ψ as a gray node, we do not allow:

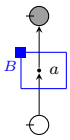


C3 ensures that the two ends of bound edges have consistent contexts. For instance, this is allowed: $\phi_{[\hat{a}\rangle^B}[\psi_{\check{a}}]^B$ but this is not: $\phi_{\hat{a}}[\psi_{\check{a}}]^B$. !-Boxes can simply overlap on an edge, for example, $[\phi_{[\hat{a}\rangle^A}]^B[\phi_{[\check{a}\rangle^B}]^A$ is perfectly valid.



The freedom to pick bs, es allows bound pairs of edges to share some common context, e.g.: $[\phi_{\hat{a}}\psi_{\check{a}}]^B$ (both nodes, and hence the edge, are inside B) or $\phi_{[\hat{a}\rangle^B}\psi_{\langle\check{a}]^B}[]^B$ (only the edge is inside B).





In the second example, B occurs in an edge term, so C2 requires the presence of $[-]^B$ somewhere in the !-tensor expression, hence we append the empty !-box $[]^B$ (actually shorthand for $[1]^B$). In particular, empty !-boxes are meaningful, unlike empty edge groups $[\epsilon\rangle^B$.

In this paper when we write a composition GH, unless otherwise stated, we will assume this forms a well defined !-tensor expression.

We say two !-tensor expressions are equivalent, written $G \equiv H$, if one can be obtained from the other by using the usual tensor equivalences from definition 3.1.6 and/or edgeterm equivalences from (3.2.1). However, we need to generalise the last two tensor equivalences to allow $1_{\hat{b}\check{a}}$ and $1_{\hat{a}\check{b}}$ to appear inside !-boxes:

$$\begin{aligned} G[K_1 \dots [K_n 1_{\hat{b}\check{a}}]^{B_n} \dots]^{B_1} &\equiv G[\check{b} \mapsto \check{a}][K_1 \dots [K_n]^{B_n} \dots]^{B_1} \\ H[K_1 \dots [K_n 1_{\hat{a}\check{b}}]^{B_n} \dots]^{B_1} &\equiv H[\hat{b} \mapsto \hat{a}][K_1 \dots [K_n]^{B_n} \dots]^{B_1} \end{aligned} \tag{3.4}$$

where $\check{b}$ and $\hat{b}$ are free in G and H respectively. These allow identities connected to nodes outside of !-boxes to still be simplified. For example:

$$\phi_{[\hat{a}\rangle^B}[1_{\hat{b}\check{a}}]^B \equiv \phi_{[\hat{b}\rangle^B}[]^B$$

Remark 3.2.6. We allow equivalent !-tensor expressions to be substituted for each other in the sense that $G \equiv H$ implies $GK \equiv HK$ (as with tensor expressions) and $[G]^A \equiv [H]^A$. Thus, $G \equiv H$ implies, for example, that $K_0[K_1 \dots [K_n G]^{B_n} \dots]^{B_1} \equiv K_0[K_1 \dots [K_n H]^{B_n} \dots]^{B_1}$.

Definition 3.2.7 (!-Tensor)**.** From the definition of !-tensor expression equivalence we can see that $\equiv$ is an equivalence relation. We define a *!-tensor* to be an $\equiv$-equivalence class of !-tensor expressions (i.e. a !-tensor for signature Σ is an element of $\mathcal{T}_{\Sigma!}/\equiv$).

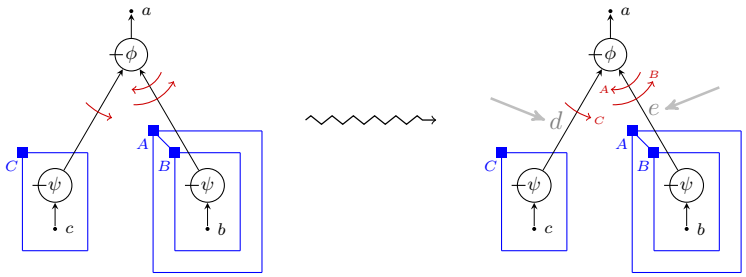
We often switch between talking about !-tensors and !-tensor expressions. This can be done for any notion which is independent of choice of expression. For example, given a !-tensor, we can always choose an expression to represent it before applying some procedure, but two equivalent expressions should give the same result.

Definitions 3.2.8 (Free/Bound Edges)**.** As for tensors we write free(G) for the set of *free edges* in G, bound(G) for the set of *bound edges* and edges(G) for the union. We naturally extend free($-$) to !-tensors as this is independent of choice of expression.

It is !-tensors that correspond to diagrams in our !-box graphical notation.

Theorem 3.2.9. *Any !-tensor can be represented unambiguously using a !-tensor diagram.*

Proof. We show this by providing a general procedure for interpreting a !-tensor diagram as a !-tensor expression, and vice-versa. For the sake of clarity, we demonstrate each step on a worked example. Given a !-tensor diagram, we wish to obtain a unique equivalence class of !-tensor expressions under $\equiv$. Begin by choosing fresh names for all interior edges (we have also added arc labels).



Then, write the !-boxes with nesting as depicted in the diagram:

$$\dots[\dots]^C[\dots[\dots]^B]^A$$

Write each node in the diagram in the location it occurs (w.r.t. !-boxes):

$$\phi_{\dots}[\psi_{\dots}]^C[[\psi_{\dots}]^B]^A$$

Finally, add the edges of each node, reading clockwise from the tick. Edges occurring under a clockwise arrow marked B should be enclosed in $[\dots\rangle^B$, and edges under an anti-clockwise arrow should be enclosed in $\langle\dots]^B$, where the outermost groups are the ones closest to the node in the diagram.

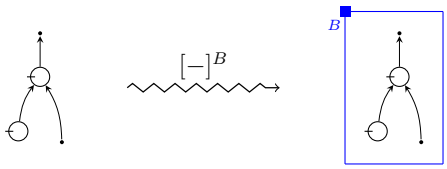
$$\phi_{\hat{a}[\langle\check{e}]^B\rangle^A\langle\check{d}]^C}[\psi_{\hat{d}\check{c}}]^C[[\psi_{\hat{e}\check{b}}]^B]^A$$

The only choices we made in this process were the choice of interior edge names and the order in which to write the individual tensors. However, up to $\equiv$ these are irrelevant. To show that any !-tensor can be represented this way, we simply run the above procedure in reverse. □

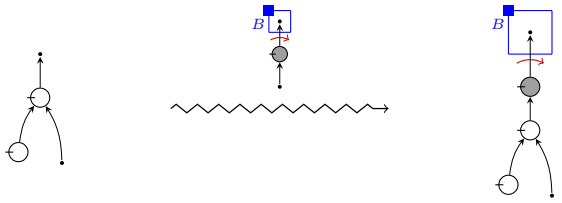
Because of this theorem, we use the terms !-tensor and !-tensor diagram interchangeably, depending on whether we wish to refer to the syntactic vs. graphical notation.

3.2.3 Concatenation

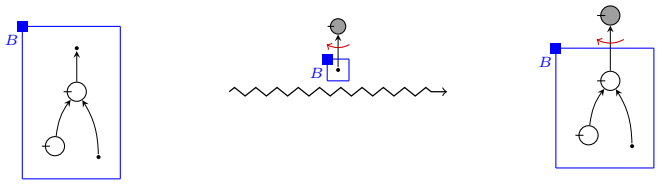
Before we present the operations which can be applied to !-tensors we have one additional construction to define. This relates to the desire to add context to diagrams. One way to do this is by enclosing G in a !-box B to get $[G]^B$.



Another way is to add additional structure using open ended wires.



This example can be done trivially by $G \mapsto FG$, with correctly named free edges, but the same method does not work when F shares some !-boxes with G. Take the following example which we would like to allow:



As !-tensor expressions we cannot take the product as this will not result in a well-formed expression. Writing ϕ for white nodes and ψ for gray nodes we would get $[\phi_{\check{a}\check{b}\check{c}}\phi_{\hat{c}}]^B\psi_{[\check{a}\rangle^B}[1]^B$. The problem being that we end up with two copies of B, which should instead have concatenated to form one. To solve this problem we define a concat operation combining two !-tensors and merging !-boxes where possible.

Definition 3.2.10 (Concat)**.** Take !-tensors G and H, let $B_1, \ldots, B_n$ be their common top-level !-boxes. We can write G in the form $G_0[G_1]^{B_1}[G_2]^{B_2}\ldots[G_n]^{B_n}$ and similarly H in the form $H_0[H_1]^{B_1}[H_2]^{B_2}\ldots[H_n]^{B_n}$. Then the *concatenation* of G and H, written $G{*}H$ is defined by:

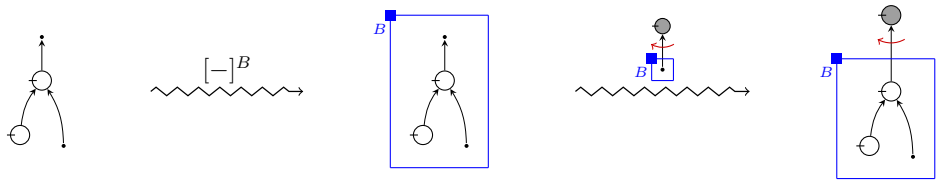
$$G{*}H := G_0H_0[G_1{*}H_1]^{B_1}[G_2{*}H_2]^{B_2}\ldots[G_n{*}H_n]^{B_n}$$

Remark 3.2.11. In the above definition we chose expressions to represent the !-tensors G and H, so we should check our definition is independent of the choice. We can make this simpler by not choosing expressions containing any of $G1, e\epsilon, \epsilon e, [\epsilon\rangle$, or $\langle\epsilon]$ (since they can be simplified to G, e, e, ϵ and ϵ

respectively) and also requiring that $1_{\hat{b}\check{a}}$ only appears if the expression does not contain $\check{b}$ or $\hat{a}$ (else by definition 3.4 we remove $1_{\hat{b}\check{a}}$). Hence the expressions differ only by α-equivalence. Since bound edges remain bound the resulting concatenation only differs by α-equivalence, so concatenation is well-defined.

Remark 3.2.12. As with $G \mapsto GH$ and $G \mapsto [G]^B$, concatenating $G \mapsto G*H$ might not return a well-formed !-tensor. From now on, unless otherwise stated, whenever we write $G*H$ we are assuming that we have a valid !-tensor.

By combining concatenation with addition of !-boxes we can build up !-tensors with added context:



The same operation can be used to describe whether a diagram G appears as a sub-diagram of H, which we refer to as matching. We will say G matches H if there exists some sequence of zero or more !-boxes $B_1, \ldots, B_n$ and a !-tensor F such that $H = [[G]^{B_1} \ldots]^{B_n} * F$.

This is useful in graph rewriting where we may want to replace instances of G with another expression G'. If G matches H then we can write $H = [[G]^{B_1} \ldots]^{B_n} * F$ as above and then apply our rewrite to get $[[G']^{B_1} \ldots]^{B_n} * F$.

Remark 3.2.13. Concatenation is a commutative and associative binary operation with unit 1. It also subsumes the usual product on !-tensors $GH = G*H$.

We have in fact already seen one situation in which we needed to concatenate two !-tensors. In (3.4) we allowed the identity edge $1_{\hat{b}\check{a}}$ to rename $\check{b}$ to $\check{a}$ and the identity edge $1_{\hat{a}\check{b}}$ to rename $\hat{b}$ to $\hat{a}$. These can be written more succinctly using concatenation, so that equivalence of !-tensors can be restated:

Definition 3.2.14 (!-Tensor Equivalence)**.** Two !-tensor expressions are equivalent if one can be transformed into the other by any number of α-conversions, edgeterm equivalences from definition 3.2.1 and identities:

$$(GH)K \equiv G(HK) \qquad GH \equiv HG \qquad G1 \equiv G$$

$$G*[[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n} \equiv G[\check{b} \mapsto \check{a}] \qquad H*[[1_{\hat{a}\check{b}}]^{B_1} \ldots]^{B_n} \equiv H[\hat{b} \mapsto \hat{a}]$$

where $\check{b}$ and $\hat{b}$ are free in G and H respectively.

3.3 Working with !-Tensors

3.3.1 Forests

To better discuss a !-tensor's !-boxes (and the nesting structure on them) we define boxes$(-)$.

Definition 3.3.1 (Box Structure)**.** We define boxes recursively by:

$$\begin{aligned}\text{boxes}([G]^B) &:= \text{boxes}(G) \cup \{C \prec B : \forall C \in \text{boxes}(G) \text{ s.t. } \nexists A,\ C \prec A\}\\ \text{boxes}(GH) &:= \text{boxes}(G) \cup \text{boxes}(H)\\ \text{boxes}(x) &:= \varnothing \qquad \text{otherwise}\end{aligned}$$

For $B \prec A$ we say B is a child of A (or A is a parent of B). We write $\preccurlyeq$ for the reflexive transitive closure of $\prec$. Note that this definition is actually valid on a !-pretensor expression but we are only interested in the !-tensor expression cases.

Theorem 3.3.2. *If $G \in \mathcal{T}_{\Sigma!}$ then* boxes(G) *is a directed forest (cycle-free directed graph where each node has at most one parent).*

Proof. It is clear from the definition that boxes(G) is directed and no !-box can have more than one parent, so let us suppose it contains a cycle. By F2 boxes(GH) cannot create a loop since boxes(G) and boxes(H) are disjoint. The loop must have been created by boxes$([G]^B)$ adding $A \prec B$ when boxes(G) already has $B \preccurlyeq A$. This means B appears in G which is a contradiction to $[G]^B$ being a well formed !-tensor expression (by F2). □

Forests can alternatively be defined as a disjoint union of directed trees. We now present some useful definitions/notations for dealing with forests.

For a subset X of a forest F, we write $\downarrow X$ and $\uparrow X$ for the downward and upward closure of $X \subseteq F$, respectively. For a single element $A \in F$, we write $\downarrow A$ for $\downarrow\{A\}$.

Definition 3.3.3 (Component)**.** If a subset $X \subseteq F$ is both upward and downward closed (i.e. $X = \downarrow X = \uparrow X$) then we say X is a *component* of F. If it contains no proper sub-components, it is called a *connected component*.

We write $F^\top$ for the set of maximal elements of F with respect to $\prec$ (those without parents). Note that for $A \in F^\top$ the set $\downarrow A$ is always a connected component, and for F finite, all connected components are of this form.

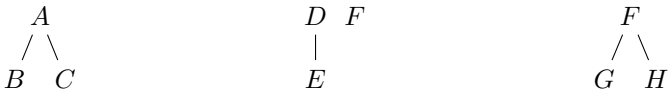
Example 3.3.4. The following diagram represents the forest with maximal elements $\{A, D\}$ and components $\downarrow A = \{\ \overset{A}{\underset{B}{/}}\ \overset{}{\underset{C}{\backslash}}\ \}$ and $\downarrow D = \{\ \overset{D}{\underset{E}{|}}\ \}$.

$$\begin{array}{ccc} A & & D \\ / & \backslash & | \\ B & C & E \end{array}$$

Definition 3.3.5 (Compatible)**.** Two forests F, F' are said to be *compatible*, written $F \triangle F'$, if the intersection $F \cap F'$ is a (possibly empty) component of both F and F'.

Equivalently, F and F' are compatible if and only if there exist forests X, Y, Z such that $F = X \uplus Y$ and $G = Y \uplus Z$. As a consequence, the union of compatible forests is always well-defined ($F \cup F' := X \uplus Y \uplus Z$), and is itself a forest.

So for example, the forest depicted in example 3.3.4 is compatible with forests such as:



but it is not compatible with forests such as:

3.3.2 !-Box Operations

We have seen in Section 2.3.1 for !-graphs that the two !-box operations Exp and Kill allow us to retrieve any concrete instance of a !-graph. We now wish to define equivalent !-box operations on !-tensors.

Before we make the definitions we note that expansion creates new copies of edges and !-boxes which need to have new 'fresh' names assigned to them. We now formalise *renamings* which rename edges and !-boxes, and *fresh renamings* which assign new fresh names during a !-box expansion.

Definition 3.3.6 (Renaming)**.** A *renaming* is a pair of bijections $\mathbf{rn} : \mathcal{B} \to \mathcal{B}$ and $\mathbf{rn} : \mathcal{N} \to \mathcal{N}$, such that $\text{support}(\mathbf{rn}) := \{x \in X : \mathbf{rn}(x) \neq x\}$ is finite for both.

Fresh renamings assign fresh names to specified sets of edge names and !-box names.

Definition 3.3.7 (Fresh Renaming)**.** Given finite sets of !-box names $B \subset \mathcal{B}$ and edge names $N \subset \mathcal{N}$ a renaming $\mathbf{fr}$ is said to be *fresh* for (B, N) if:

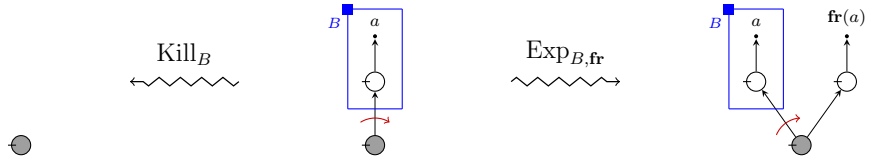
$$N \cap \mathbf{fr}(N) = \varnothing \quad \text{and} \quad B \cap \mathbf{fr}(B) = \varnothing$$

We do not need to worry about existence of such renamings for a given pair (B, N), since both $\mathcal{B}$ and $\mathcal{N}$ are infinite.

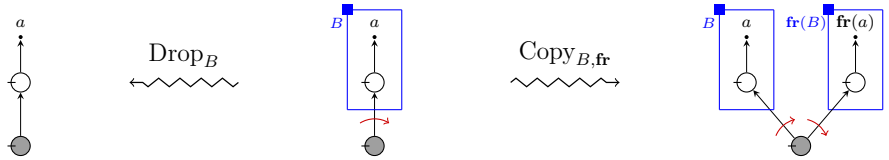
Remark 3.3.8. We add the restriction of finite support to fresh renamings with the intent of always having infinitely many names which they do not affect. This will allow us to use these names for new fresh renamings. For example, given a fresh renaming $\mathbf{fr}$ and a finite set X of names not in $\text{support}(\mathbf{fr})$ we can always find a new fresh renaming equal to $\mathbf{fr}$ on its support but also fresh for X. We call this process *extending* a fresh renaming, in this case to also cover the set X.

Definition 3.3.9 (Fresh for !-Tensor Expressions)**.** We say **fr** is *fresh for* the !-tensor expressions $G_1, \ldots, G_n$ if it is a fresh renaming for $(\bigcup_{i=1}^{n} \text{boxes}(G_i), \bigcup_{i=1}^{n} \text{edges}(G_i))$. This guarantees **fr** can be used to assign fresh names during expansion of a !-box in any G_i.

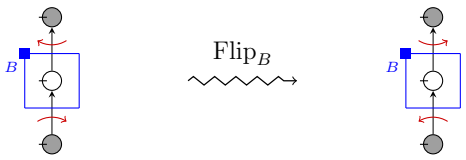
We have already seen the two !-box operations of expanding and killing in section 2.3.1. These allow us to replace any !-box with a finite number of copies of the contents. $\text{Exp}_{B,\mathbf{fr}}$ creates a single new copy of the contents of B freshly named using **fr** and attached as prescribed by the arcs. Kill_B deletes B and all contents from the !-tensor.



These two operations allow us to retrieve any concrete tensor instance of the family of diagrams represented by the !-tensor. However, there are more operations we may wish to apply on !-boxes to create a more powerful proof system. The additional operations used in this thesis are as follows. Copying, written $\text{Copy}_{B,\mathbf{fr}}$, creates a new (freshly named) copy of B and all contents attached as prescribed by the arcs. Dropping, written Drop_B, removes the !-box B leaving the contents as they were.



These operations have all appeared previously for the case of !-graphs. But for !-tensors we have one additional !-box operation which is a result of the arc directions. We wish to be able to 'flip' the direction of all arcs around a !-box which still results in the same concrete instances. We call the !-box operation which flips all of B's clockwise arcs to anticlockwise and vice versa Flip_B.



Definition 3.3.10 (!-Box Operations)**.** A !-box operation Op_B (possibly employing a fresh renaming **fr**) is an operation on !-tensor expressions which acts somewhat trivially on the following cases:

$$\text{Op}_B(GH) := \text{Op}_B(G)\,\text{Op}_B(H) \qquad \text{Op}_B(ef) := \text{Op}_B(e)\,\text{Op}_B(f)$$
$$\text{Op}_B([G]^A) := [\text{Op}_B(G)]^A \qquad \text{Op}_B(\phi_e) := \phi_{\text{Op}_B(e)}$$
$$\text{Op}_B([e\rangle^A) := [\text{Op}_B(e)\rangle^A \qquad \text{Op}_B(\langle e]^A) := \langle \text{Op}_B(e)]^A$$
$$\text{Op}_B(x) := x$$

where $A \neq B$ and $x \in \{1, 1_{\hat{a}\check{b}}, \check{a}, \hat{a}, \epsilon\}$. The !-box operations we are interested in are the following (defined by their actions in the three cases not mentioned above):

$$\begin{aligned}
\mathrm{Flip}_B([G]^B) &:= [G]^B & \mathrm{Kill}_B([G]^B) &:= 1 & \mathrm{Drop}_B([G]^B) &:= G \\
\mathrm{Flip}_B([e\rangle^B) &:= \langle e]^B & \mathrm{Kill}_B([e\rangle^B) &:= \epsilon & \mathrm{Drop}_B([e\rangle^B) &:= e \\
\mathrm{Flip}_B(\langle e]^B) &:= [e\rangle^B & \mathrm{Kill}_B(\langle e]^B) &:= \epsilon & \mathrm{Drop}_B(\langle e]^B) &:= e
\end{aligned}$$

$$\begin{aligned}
\mathrm{Copy}_{B,\mathbf{fr}}([G]^B) &:= [G]^B[\mathbf{fr}(G)]^{\mathbf{fr}(B)} & \mathrm{Exp}_{B,\mathbf{fr}}([G]^B) &:= [G]^B\,\mathbf{fr}(G) \\
\mathrm{Copy}_{B,\mathbf{fr}}([e\rangle^B) &:= [e\rangle^B[\mathbf{fr}(e)\rangle^{\mathbf{fr}(B)} & \mathrm{Exp}_{B,\mathbf{fr}}([e\rangle^B) &:= [e\rangle^B\,\mathbf{fr}(e) \\
\mathrm{Copy}_{B,\mathbf{fr}}(\langle e]^B) &:= \langle \mathbf{fr}(e)]^{\mathbf{fr}(B)}\langle e]^B & \mathrm{Exp}_{B,\mathbf{fr}}(\langle e]^B) &:= \mathbf{fr}(e)\langle e]^B
\end{aligned}$$

These operations are not minimal, for example, expanding can be factorised into two operations:

$$\mathrm{Exp}_{B,\mathbf{fr}} = \mathrm{Drop}_{\mathbf{fr}(B)} \circ \mathrm{Copy}_{B,\mathbf{fr}} \tag{3.5}$$

We wish to prove that the application of a !-box operation to a !-tensor expression results in a !-tensor expression. First we present a couple of lemmas which will help us see how !-box operations affect !-tensors.

Lemma 3.3.11. *For a !-box operation* $\mathrm{Op}_{B,\mathbf{fr}}$ *(possibly using fresh renaming* $\mathbf{fr}$*) we find that* $\mathrm{boxes}(\mathrm{Op}_{B,\mathbf{fr}}(G))$ *depends only on* $\mathrm{boxes}(G)$ *(not the actual content of* G*).*

Proof. We prove this by structural induction on the definition of !-tensor expressions. The cases 1, $1_{\hat{a}\check{b}}$, ϕ_e, GH, and $[G]^A$ where $A \neq B$ are trivial. We check the final case $[G]^B$ individually for each operation.

- $\mathrm{boxes}(\mathrm{Flip}_B([G]^B)) = \mathrm{boxes}([G]^B)$
- $\mathrm{boxes}(\mathrm{Kill}_B([G]^B)) = \mathrm{boxes}(1) = \varnothing$
- $\mathrm{boxes}(\mathrm{Drop}_B([G]^B)) = \mathrm{boxes}(G) = \mathrm{boxes}([G]^B) \backslash \{B\}$
- $\mathrm{boxes}(\mathrm{Copy}_{B,\mathbf{fr}}([G]^B)) = \mathrm{boxes}([G]^B[\mathbf{fr}(G)]^{\mathbf{fr}(B)}) = \mathrm{boxes}([G]^B) \cup \mathbf{fr}(\mathrm{boxes}([G]^B))$
- the $\mathrm{Exp}_{B,\mathbf{fr}}$ case follows from Copy and Drop using (3.5)

□

This lemma allows us to lift !-box operations to act on forests. For example, applying operations to $[[[1]^C[1]^D]^B[1]^E]^A$ then lifting to forests we get:

$$\mathrm{Kill}_B\left(\begin{array}{ccc} & A & \\ & / \quad \backslash & \\ B & & E \\ / \quad \backslash & & \\ C & D & \end{array}\right) = \begin{array}{c} A \\ | \\ E \end{array} \qquad \mathrm{Copy}_{B,\mathbf{fr}}\left(\begin{array}{ccc} & A & \\ & / \quad \backslash & \\ B & & E \\ / \quad \backslash & & \\ C & D & \end{array}\right) = \begin{array}{ccccc} & & A & & \\ & \swarrow & & \searrow & \\ B & & & & B' \quad E \\ / \quad \backslash & & & & / \quad \backslash \\ C & D & & & C' \quad D' \end{array}$$

Where we have written X' as shorthand for $\mathbf{fr}(X)$. We can also see how contexts are affected by !-box operations:

Lemma 3.3.12. *If* $\mathrm{ectx}_G(a) = [E_1, \ldots, E_n]$, $\mathrm{nctx}_G(a) = [N_1, \ldots, N_m]$ *then the following table shows the contexts affected by !-box operations (writing* B' *for* $\mathbf{fr}(B)$*):*

		ectx	nctx
Drop_{E_i}	a	$[E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n]$	
Drop_{N_i}	a		$[N_1, \ldots, N_{i-1}, N_{i+1}, \ldots, N_m]$
Copy_{E_i}	$\mathbf{fr}(a)$	$[E'_1, \ldots, E'_i, E_{i+1}, \ldots, E_n]$	
Copy_{N_i}	$\mathbf{fr}(a)$	$[E'_1, \ldots, E'_n]$	$[N'_1, \ldots, N'_i, N_{i+1}, \ldots, N_m]$
Exp_{E_i}	$\mathbf{fr}(a)$	$[E'_1, \ldots, E'_{i-1}, E_{i+1}, \ldots, E_n]$	
Exp_{N_i}	$\mathbf{fr}(a)$	$[E'_1, \ldots, E'_n]$	$[N'_1, \ldots, N'_{i-1}, N_{i+1}, \ldots, N_m]$

Note that Flip *has no affect on contexts and any edges remaining after a* Kill *operation have not had their contexts altered.*

Proof. Op_B recurses through the definition of !-tensor expressions until it hits $[-]^B, [-\rangle^B$ or $\langle -]^B$ so we only check these cases. Let $e(a)$ be an edgeterm including a, hence $\mathbf{fr}(e(a))$ contains the edge $\mathbf{fr}(a)$. Without loss of generality we take a expanding clockwise in E_i (other direction has similar proof):

- Drop_{E_i} : $[e(a)\rangle^{E_i} \to e(a)$ so the edge context of a has E_i removed.
- Drop_{B_i} : $[\phi_{e(a)} \ldots]^{B_i} \to \phi_{e(a)} \ldots$ so the node context of a has B_i removed.
- Copy_{E_i} : $[e(a)\rangle^{E_i} \to [e(a)\rangle^{E_i}[\mathbf{fr}(e(a))\rangle^{E'_i}$ so the context of a has not changed but we have new edge $\mathbf{fr}(a)$ similar to a except in the edge context has $E_1, \ldots, E_i$ replaced by $E'_1, \ldots, E'_i$ respectively.
- Copy_{B_i} : $[\phi_{e(a)} \ldots]^{B_i} \to [\phi_{e(a)} \ldots]^{B_i}[\mathbf{fr}(\phi_{e(a)} \ldots)]^{B'_i}$ so the context of a has not changed but we have new edge $\mathbf{fr}(a)$ similar to a except the edge context has $E_1, \ldots, E_n$ replaced by $E'_1, \ldots, E'_n$ respectively and the node context has $B_1, \ldots, B_i$ replaced by $B'_1, \ldots, B'_i$ respectively.
- The Exp_{E_i} and Exp_{N_i} cases follow from the previous four by (3.5).

□

From the definitions, the results of !-box operations on !-tensors are clearly !-pretensors. To additionally show that $\mathrm{Op}_{B,\mathbf{fr}}(G)$ is a valid !-tensor expression, i.e. that $\mathrm{Op}_{B,\mathbf{fr}}(G) \in \mathcal{T}_{\Sigma!}$, we need to show that the !-tensor conditions (definition 3.2.5) still hold.

Theorem 3.3.13. *If $G \in \mathcal{T}_{\Sigma!}$ and $\mathrm{Op}_{B,\mathbf{fr}}$ is one of our five !-box operations then $\mathrm{Op}_{B,\mathbf{fr}}(G) \in \mathcal{T}_{\Sigma!}$.*

Proof. For each !-box operation the conditions F1-2 are trivial since new edges/!-boxes have new names created by a fresh renaming. Hence we will only check conditions C1-3 for each of our !-box operations. We will use lemma 3.3.12 to see how the contexts are affected.

-Flip

Contexts are not affected by Flip_B so the conditions C1-3 are trivially respected by Flip_B.

-Kill

C1: If $a \in \mathrm{edges}(\mathrm{Kill}_B(G))$ then contexts were not affected by Kill_B so the same condition holds.

C2: Suppose we have $a \in \mathrm{edges}(\mathrm{Kill}_B(G))$ with edge context $[E_1, \ldots, E_n]$. This must have come from $a \in \mathrm{edges}(G)$ with edge context $[E_1, \ldots, E_n]$, hence $E_1 \prec_G \ldots \prec_G E_n$ (by C2 on G), with no E_i nested inside B. Then, $E_1 \prec_{\mathrm{Kill}_B(G)} \ldots \prec_{\mathrm{Kill}_B(G)} E_n$.
If we also have $E_n \prec_{\mathrm{Kill}_B(G)} C$ then $E_n \prec_G C$ and so $C \in \mathrm{ctx}_G(a)$ which (since C is not nested in B) implies $C \in \mathrm{ctx}_{\mathrm{Kill}_B(G)}(a)$.

C3: If $\hat{a}, \check{a} \in \mathrm{edges}(\mathrm{Kill}_B(G))$ they must be from $\hat{a}, \check{a} \in \mathrm{edges}(G)$ and $\mathrm{ectx}, \mathrm{nctx}$ were not affected by Kill_B so the condition still holds.

-Drop:

C1: Again trivial since ectx and nctx have only lost !-boxes.

C2: If $a \in \mathrm{edges}(\mathrm{Drop}_B(G))$ has edge context $[E_1, \ldots, E_n]$ then $a \in \mathrm{edges}(G)$ could have the same edge context or contain B, i.e $[E_1, \ldots, B, \ldots, E_n]$, in which case nesting in G would be $E_1 \prec_G \ldots \prec_G B \prec_G \ldots \prec_G E_n$. In either case, the nesting in $\mathrm{Drop}_B(G)$ is $E_1 \prec_{\mathrm{Drop}_B(G)} \ldots \prec_{\mathrm{Drop}_B(G)} E_n$ since B is removed.
If we also have $E_n \prec_{\mathrm{Drop}_B(G)} C$ then either $E_n \prec_G C$ or $E_n \prec_G B \prec_G C$. In either case $C \in \mathrm{ctx}_G(a)$ which implies $C \in \mathrm{ctx}_{\mathrm{Drop}_B(G)}(a)$.

C3: If $\hat{a}, \check{a} \in \mathrm{edges}(\mathrm{Drop}_B(G))$ then $\hat{a}, \check{a} \in \mathrm{edges}(G)$ and $\mathrm{ectx}, \mathrm{nctx}$ only lost the !-box B so the condition still holds by removing B from es, bs.

-Copy

C1: Edges in $\mathrm{Copy}_{B,\mathbf{fr}}(G)$ are either edges from G or fresh names for edges in G. For the former $\mathrm{ectx}, \mathrm{nctx}$ have not been changed so the condition holds. For the latter we know from lemma 3.3.12 that !-boxes in $\mathrm{ectx}, \mathrm{nctx}$ have only been replaced by fresh versions of themselves hence are still distinct.

C2: Take $a \in \mathrm{edges}(G)$:

- For a in $\mathrm{Copy}_{B,\mathbf{fr}}(G)$ we have $\mathrm{ectx}_{\mathrm{Copy}_{B,\mathbf{fr}}(G)}(a) = \mathrm{ectx}_G(a)$ and $E_i \prec_G E_{i+1} \implies E_i \prec_{\mathrm{Copy}_{B,\mathbf{fr}}(G)} E_{i+1}$ so the conditions hold.
- $\mathrm{ectx}_{\mathrm{Copy}_{B,\mathbf{fr}}(G)}(\mathbf{fr}(a)) = [E'_1, \ldots, E'_i, E_{i+1}, \ldots, E_n]$ where $\mathrm{ectx}_G(a) = [E_1, \ldots, E_n]$ and $B = E_i$. From the definition of $\mathrm{Copy}_{B,\mathbf{fr}}$ we see E'_i $(= B')$ must have been created inside E_{i+1}. For the other !-boxes, it is clear from $\mathrm{ectx}_G(a)$ that $E'_j \prec_{\mathrm{Copy}_{B,\mathbf{fr}}(G)} E'_{j+1}$ for $j < i$ and $E_j \prec_{\mathrm{Copy}_{B,\mathbf{fr}}(G)} E_{j+1}$ for $j > i$.

  If the top !-box in $\mathrm{ectx}_{\mathrm{Copy}_{B,\mathbf{fr}}(G)}(\mathbf{fr}(a))$ is nested inside C then $E_n \prec_G C$. Then (by C2 applied to G) $C \in \mathrm{ctx}_G(a)$ which implies $C \in \mathrm{ctx}_{\mathrm{Copy}_{B,\mathbf{fr}}(G)}(\mathbf{fr}(a))$ (since C is not nested in B).

C3: For $\hat{a}, \check{a} \in \mathrm{edges}(\mathrm{Copy}_{B,\mathbf{fr}}(G))$ where $\hat{a}, \check{a} \in \mathrm{edges}(G)$, ectx and nctx are not affected by $\mathrm{Copy}_{B,\mathbf{fr}}$ so the condition still holds.

For $\mathbf{fr}(\hat{a}), \mathbf{fr}(\check{a}) \in \mathrm{edges}(\mathrm{Copy}_{B,\mathbf{fr}}(G))$ where $\hat{a}, \check{a} \in \mathrm{edges}(G)$, there exist !-boxes $E_i, N_i \in \mathcal{B}$ such that:

$$[E_1, \ldots, E_n].\,\mathrm{nctx}_G(\check{a}) = \mathrm{ectx}_G(\hat{a}).[N_1, \ldots, N_m]$$

$$[E_1, \ldots, E_n].\,\mathrm{nctx}_G(\hat{a}) = \mathrm{ectx}_G(\check{a}).[N_1, \ldots, N_m]$$

Since the edge was copied, B must be in the contexts of $\hat{a}$ and $\check{a}$. We need to show that there exist es, bs such that:

$$es.\,\mathrm{nctx}_G(\mathbf{fr}(\check{a})) = \mathrm{ectx}_G(\mathbf{fr}(\hat{a})).bs \quad \text{and} \quad es.\,\mathrm{nctx}_G(\mathbf{fr}(\hat{a})) = \mathrm{ectx}_G(\mathbf{fr}(\check{a})).bs$$

We do this by considering 4 cases:

- If $B \in \mathrm{ectx}_G(\check{a}) \cap \mathrm{ectx}_G(\hat{a})$ then $B = E_i$ for some i, and the condition holds by letting $es := [E'_1, \ldots, E'_i, E_{i+1}, \ldots, E_n]$ and $bs := [N_1, \ldots, N_m]$.
- Similarly, if $B \in \mathrm{nctx}_G(\check{a}) \cap \mathrm{nctx}_G(\hat{a})$ then $B = N_i$ for some i, and the condition holds by letting $es := [E'_1, \ldots, E'_m]$ and $bs := [N'_1, \ldots, N'_i, N_{i+1}, \ldots, N_n]$.
- If $B \in \mathrm{ectx}_G(\check{a}) \cap \mathrm{nctx}_G(\hat{a})$, the condition holds with $es := [E'_1, \ldots, E'_m]$ and $bs := [N_1, \ldots, N_m]$.
- Otherwise $B \in \mathrm{nctx}_G(\check{a}) \cap \mathrm{ectx}_G(\hat{a})$, in which case the condition holds with $es := [E'_1, \ldots, E'_m]$ and $bs := [N_1, \ldots, N_m]$.

-Expand

We note that $\mathrm{Exp}_{B,\mathbf{fr}} = \mathrm{Drop}_{\mathbf{fr}(B)} \circ \mathrm{Copy}_{B,\mathbf{fr}}$ so this case follows from the previous cases.

□

Lemma 3.3.14. *For* Op_B *one of our five !-box operations and* G *and* H *two !-tensors expressions* $\mathrm{Op}_B(G*H) = \mathrm{Op}_B(G)*\mathrm{Op}_B(H)$

Proof. Let $G = G_0[G_1]^{B_1}[G_2]^{B_2}\dots[G_n]^{B_n}$ and $H = H_0[H_1]^{B_1}[H_2]^{B_2}\dots[H_n]^{B_n}$ where $B_1,\dots,B_n$ are the shared top-level !-boxes. We prove the lemma by induction on the maximum shared nesting depth of !-boxes in G, H. The base case is trivial, for the step case if B is not one of the shared !-boxes $B_1,\dots,B_n$, then we find:

$$\begin{aligned}
\mathrm{Op}_B(G{*}H) &= \mathrm{Op}_B(G_0H_0[G_1{*}H_1]^{B_1}\dots[G_n{*}H_n]^{B_n})\\
&= \mathrm{Op}_B(G_0H_0)[\mathrm{Op}_B(G_1{*}H_1)]^{B_1}\dots[\mathrm{Op}_B(G_n{*}H_n)]^{B_n}\\
&= \mathrm{Op}_B(G_0)\,\mathrm{Op}_B(H_0)[\mathrm{Op}_B(G_1){*}\mathrm{Op}_B(H_1)]^{B_1}\dots[\mathrm{Op}_B(G_n){*}\mathrm{Op}_B(H_n)]^{B_n}\\
&= \mathrm{Op}_B(G_0)[\mathrm{Op}_B(G_1)]^{B_1}\dots[\mathrm{Op}_B(G_n)]^{B_n}{*}\mathrm{Op}_B(H_0)[\mathrm{Op}_B(H_1)]^{B_1}\dots[\mathrm{Op}_B(H_n)]^{B_n}\\
&= \mathrm{Op}_B(G){*}\mathrm{Op}_B(H)
\end{aligned}$$

where the third equality uses the inductive hypothesis. For the step case if B is one of the shared !-box we have to check $\mathrm{Op}_B([G{*}H]^B) = \mathrm{Op}_B([G]^B){*}\mathrm{Op}_B([H]^B)$. This is trivial for Flip, Kill and Drop so we check $\mathrm{Copy}_{B,\mathbf{fr}}$ (with $\mathbf{fr}$ fresh for $[G{*}H]^B$) and then $\mathrm{Exp}_{B,\mathbf{fr}}$ follows.

$$\begin{aligned}
\mathrm{Copy}_{B,\mathbf{fr}}([G{*}H]^B) &= [G{*}H]^B[\mathbf{fr}(G{*}H)]^{\mathbf{fr}(B)}\\
&= [G]^B{*}[H]^B{*}[\mathbf{fr}(G)]^{\mathbf{fr}(B)}{*}[\mathbf{fr}(H)]^{\mathbf{fr}(B)}\\
&= [G]^B{*}[\mathbf{fr}(G)]^{\mathbf{fr}(B)}{*}[H]^B{*}[\mathbf{fr}(H)]^{\mathbf{fr}(B)}\\
&= \mathrm{Copy}_{B,\mathbf{fr}}([G]^B){*}\mathrm{Copy}_{B,\mathbf{fr}}([H]^B)
\end{aligned}$$

□

We wish to lift !-box operations to work on !-tensors, rather than !-tensor expressions. We prove this possible in theorem 3.3.17. First we need to treat the $\mathrm{Copy}_{B,\mathbf{fr}}$ case carefully. This is due to the use of fresh renamings which depend on bound names so cannot trivially be lifted to !-tensors (which are classes of expressions each with their own set of bound names). We start by checking the !-tensor represented by $\mathrm{Copy}_{B,\mathbf{fr}}(G)$ is independent of the value of $\mathbf{fr}$ on $\mathrm{bound}(G)$.

Lemma 3.3.15. *If* $\mathbf{fr}, \mathbf{fr}'$ *are fresh for a !-tensor expression* G *and agree on* $\mathrm{boxes}(G)$ *and* $\mathrm{free}(G)$ *then* $\mathrm{Copy}_{B,\mathbf{fr}}(G) \equiv \mathrm{Copy}_{B,\mathbf{fr}'}(G)$.

Proof. For $a \in \mathrm{bound}(G)$ condition C3 tells us that $B \in \mathrm{ctx}_G(\hat{a}) \Leftrightarrow B \in \mathrm{ctx}_G(\check{a})$ and so copying creates fresh versions of $\hat{a}$ and $\check{a}$ in tandem. Hence the two expressions differ only on certain bound names. Suppose they differ by $\mathrm{Copy}_{B,\mathbf{fr}}(G)$ having a bound edge named a where $\mathrm{Copy}_{B,\mathbf{fr}'}(G)$ has bound edge named a'. We can choose a completely fresh name b, then by α-conversion a and a' can each be replaced by b. By repeating this for each conflicting bound name, α-equivalence eventually gives us $\mathrm{Copy}_{B,\mathbf{fr}}(G) \equiv \mathrm{Copy}_{B,\mathbf{fr}'}(G)$. □

Now we attempt to define the operation $\mathrm{Copy}_{B,\mathbf{fr}}$ on !-tensors using the following procedure.

Definition 3.3.16 (Copy on !-Tensors)**.** Given a !-tensor G and a fresh renaming $\mathbf{fr}$ for the pair $(\text{boxes}(G), \text{free}(G))$:

- Choose an expression representing G, say G_1, making sure $\text{bound}(G_1) \cap \text{support}(\mathbf{fr}) = \varnothing$
- Choose an extension $\mathbf{fr}_1$ of $\mathbf{fr}$ which is fresh for G_1
- Define $\text{Copy}_{B,\mathbf{fr}}(G)$ to be $\text{Copy}_{B,\mathbf{fr}_1}(G_1)$

These choices are possible by α-conversion and since $\mathbf{fr}$ has finite support. Now we need to check this is well defined by making sure the result is independent of our choice of expression G_1 and fresh extension $\mathbf{fr}_1$. Suppose G_2 and $\mathbf{fr}_2$ are another such expression and fresh extension. We can then choose $\mathbf{fr}'$, an extension of $\mathbf{fr}$ also free for $\text{bound}(G_1) \cup \text{bound}(G_2)$. By lemma 3.3.15 $\text{Copy}_{B,\mathbf{fr}_1}(G_1) \equiv \text{Copy}_{B,\mathbf{fr}'}(G_1)$ and $\text{Copy}_{B,\mathbf{fr}_2}(G_2) \equiv \text{Copy}_{B,\mathbf{fr}'}(G_2)$ so our problem is reduced to showing that $\text{Copy}_{B,\mathbf{fr}'}(G_1) \equiv \text{Copy}_{B,\mathbf{fr}'}(G_2)$. This case is treated as the other cases below.

Theorem 3.3.17. *Let* $\mathbf{fr}$ *be a fresh renaming for the !-tensor expressions* G, H*. Then* $G \equiv H$ *implies* $\text{Op}_{B,\mathbf{fr}}(G) \equiv \text{Op}_{B,\mathbf{fr}}(H)$*.*

Proof. We need to check our enforced equivalences still hold after Op_B. It is clear from the definitions of $\text{Op}_B(GH), \text{Op}_B(ef), \text{Op}_B(1), \text{Op}_B(\epsilon)$ that associativity/commutativity/unit conditions are preserved. We check the other cases:

- When $A \neq B$ we find

$$\text{Op}_B([\epsilon\rangle^A) \equiv [\text{Op}_B(\epsilon)\rangle^A \equiv [\epsilon\rangle^A \equiv \epsilon \equiv \ldots \equiv \text{Op}_B(\langle\epsilon]^A)$$

  When $A = B$ we need to check each operation individually:

$$\text{Flip}_B([\epsilon\rangle^B) \equiv \langle\epsilon]^B \equiv \epsilon \equiv [\epsilon\rangle^B \equiv \text{Flip}_B(\langle\epsilon]^B)$$

$$\text{Kill}_B([\epsilon\rangle^B) \equiv \epsilon \equiv \text{Kill}_B(\langle\epsilon]^B)$$

$$\text{Drop}_B([\epsilon\rangle^B) \equiv \epsilon \equiv \text{Drop}_B(\langle\epsilon]^B)$$

$$\text{Copy}_{B,\mathbf{fr}}([\epsilon\rangle^B) \equiv [\epsilon\rangle^B[\epsilon\rangle^{\mathbf{fr}(B)} \equiv \epsilon \equiv \langle\epsilon]^{\mathbf{fr}(B)}\langle\epsilon]^B \equiv \text{Copy}_{B,\mathbf{fr}}(\langle\epsilon]^B)$$

$$\text{Exp}_{B,\mathbf{fr}}([\epsilon\rangle^B) \equiv [\epsilon\rangle^B\epsilon \equiv \epsilon \equiv \epsilon\langle\epsilon]^B \equiv \text{Exp}_{B,\mathbf{fr}}(\langle\epsilon]^B)$$

- From definition 3.2.14 we need to check $G*[[1_{\hat{b}\check{a}}]^{B_1}\ldots]^{B_n} \equiv G[\check{b} \mapsto \check{a}]$ is preserved given free name $\check{b}$ in G.

  If $B \notin [B_1, \ldots, B_n]$ then neither $\hat{b}$ nor $\check{b}$ is affected by Op_B so using lemma 3.3.14 we get:

$$\begin{aligned}\text{Op}_B(G*[[1_{\hat{b}\check{a}}]^{B_1}\ldots]^{B_n}) &\equiv \text{Op}_B(G)*\text{Op}_B([[1_{\hat{b}\check{a}}]^{B_1}\ldots]^{B_n}) \\ &\equiv \text{Op}_B(G)*[[1_{\hat{b}\check{a}}]^{B_1}\ldots]^{B_n} \\ &\equiv \text{Op}_B(G)[\check{b} \mapsto \check{a}] \\ &\equiv \text{Op}_B(G[\check{b} \mapsto \check{a}])\end{aligned}$$

otherwise $B = B_i$ for some $i \leq n$. Then each operation needs to be checked individually:

- $\mathrm{Flip}_B(G{*}[[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n})$

  $\equiv \mathrm{Flip}_B(G){*}\mathrm{Flip}_B([[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n})$

  $\equiv \mathrm{Flip}_B(G){*}[[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n}$

  $\equiv \mathrm{Flip}_B(G)[\check{b} \mapsto \check{a}]$

  $\equiv \mathrm{Flip}_B(G[\check{b} \mapsto \check{a}])$

- $\mathrm{Kill}_B(G{*}[[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n})$

  $\equiv \mathrm{Kill}_B(G){*}\mathrm{Kill}_B([[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n})$

  $\equiv \mathrm{Kill}_B(G){*}[[1]^{B_{i+1}} \ldots]^{B_n}$

  $\equiv \mathrm{Kill}_B(G)$

  $\equiv \mathrm{Kill}_B(G[\check{b} \mapsto \check{a}])$

  where the last equivalence is true since Kill_B will delete the edge into B whether it is named $\check{b}$ or $\check{a}$.

- $\mathrm{Drop}_B(G{*}[[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n})$

  $\equiv \mathrm{Drop}_B(G){*}\mathrm{Drop}_B([[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n})$

  $\equiv \mathrm{Drop}_B(G){*}[[[[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_{i-1}}]^{B_{i+1}} \ldots]^{B_n}$

  $\equiv \mathrm{Drop}_B(G)[\check{b} \mapsto \check{a}]$

  $\equiv \mathrm{Drop}_B(G[\check{b} \mapsto \check{a}])$

  where the last equivalence is true since Drop_B will act the same whether our edge is named $\check{b}$ or $\check{a}$.

- Abbreviating $\mathbf{fr}(x)$ as x':

  $\mathrm{Copy}_{B,\mathbf{fr}}(G{*}[[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n})$

  $\equiv \mathrm{Copy}_{B,\mathbf{fr}}(G){*}\mathrm{Copy}_{B,\mathbf{fr}}([[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n})$

  $\equiv \mathrm{Copy}_{B,\mathbf{fr}}(G){*}[[[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_i}[[1_{\hat{b'}\check{a'}}]^{B'_1} \ldots]^{B'_i} \ldots]^{B_n}$

  $\equiv \mathrm{Copy}_{B,\mathbf{fr}}(G){*}[[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n}{*}[[1_{\hat{b'}\check{a'}}]^{B'_1} \ldots]^{B_n}$

  $\equiv \mathrm{Copy}_{B,\mathbf{fr}}(G)[\check{b} \mapsto \check{a}][\check{b'} \mapsto \check{a'}]$

  $\equiv \mathrm{Copy}_{B,\mathbf{fr}}(G[\check{b} \mapsto \check{a}])$

  where the last equivalence is true since moving a rename to after $\mathrm{Copy}_{B,\mathbf{fr}}$, you have to do both the original rename and the fresh version.

– $\mathrm{Exp}_{B,\mathbf{fr}}$ follows by applying $\mathrm{Drop}_{\mathbf{fr}(B)}$ to the above case:

$$\begin{aligned}\mathrm{Exp}_{B,\mathbf{fr}}(G*[[1_{\hat{b}\check{a}}]^{B_1}\ldots]^{B_n}) &= \mathrm{Drop}_{\mathbf{fr}(B)}(\mathrm{Copy}_{B,\mathbf{fr}}(G*[[1_{\hat{b}\check{a}}]^{B_1}\ldots]^{B_n})) \\ &= \mathrm{Drop}_{\mathbf{fr}(B)}(\mathrm{Copy}_{B,\mathbf{fr}}(G[\check{b}\mapsto\check{a}])) \\ &= \mathrm{Exp}_{B,\mathbf{fr}}(G[\check{b}\mapsto\check{a}])\end{aligned}$$

- The proof for $H*[[1_{\hat{a}\check{b}}]^{B_1}\ldots]^{B_n} \equiv H[\hat{b}\mapsto\hat{a}]$ is similar.
- Finally, α-equivalence is trivial since for $a\in\mathcal{N}$, $B\in\mathrm{ctx}_G(\hat{a})\Leftrightarrow B\in\mathrm{ctx}_G(\check{a})$ and so a bound wire $\hat{a},\check{a}$ under a !-box operation may result in bound wires $\hat{a},\check{a}$ and possibly $\mathbf{fr}(\hat{a}),\mathbf{fr}(\check{a})$ each of which can be renamed freely using α-conversion.

□

3.3.3 Instantiation

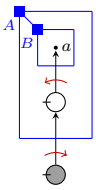
Definition 3.3.18 (Fresh for !-Tensors)**.** We say the renaming $\mathbf{fr}$ is *fresh for* the !-tensors $G_1,\ldots,G_n$ (with compatible !-box structures) if it is a fresh renaming on $(\bigcup_{i=1}^n \mathrm{boxes}(G_i), \bigcup_{i=1}^n \mathrm{free}(G_i))$. By the above theorems this means $\mathrm{Copy}_{B,\mathbf{fr}}$ is a well-defined !-box operation on each G_i, hence so is $\mathrm{Exp}_{B,\mathbf{fr}}$.

Definition 3.3.19 (Partial Instantiation)**.** A *partial instantiation* of a !-tensor is a sequence of zero or more $\mathrm{Kill}_B, \mathrm{Exp}_{B,\mathbf{fr}}$ operations, each with B in its domain and $\mathbf{fr}$ fresh for its domain.

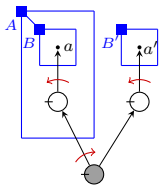
Definition 3.3.20 (Instantiation, Concrete Instance)**.** An *instantiation* of a !-tensor G is a partial instantiation i such that $i(G)$ is a concrete tensor. We write $\mathrm{Inst}(G)$ for the set of instantiations of G and call $i(G)$ a *concrete instance* of G.

Example 3.3.21. Take the following !-tensor as an example:

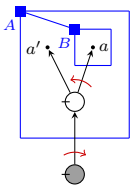
G :=



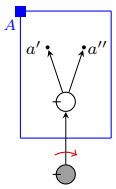
The function $\mathbf{fr} = [A\leftrightarrow A', B\leftrightarrow B', a\leftrightarrow a']$ is fresh for G so can be used when expanding a !-box as in the following partial instantiations:



$\mathrm{Exp}_{A,\mathbf{fr}}$

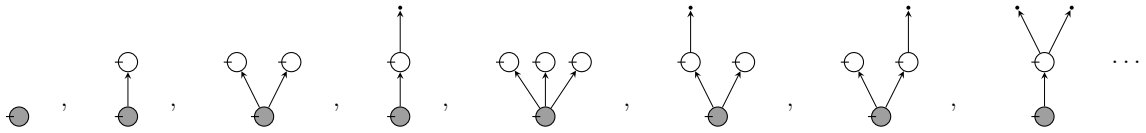


$\mathrm{Exp}_{B,\mathbf{fr}}$

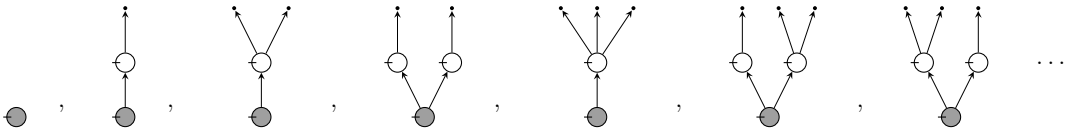


$\mathrm{Kill}_B \circ \mathrm{Exp}_{B,\mathbf{fr}'} \circ \mathrm{Exp}_{B,\mathbf{fr}}$

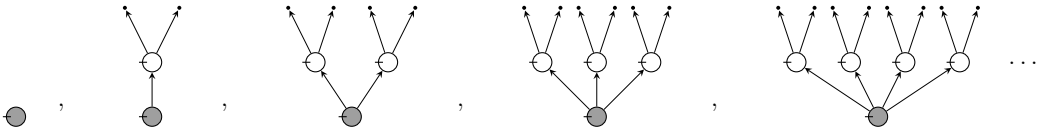
Each concrete instance of G is made up of a gray node with edges to any number of white nodes, each with an arbitrary number of outputs:



The partial instantiations also lead to some interesting subsets of this family of diagrams. For example, the second partial instantiation above represents all diagrams where each white node has at least one output:



whereas the third partial instantiation represents the diagrams where each white node has exactly two outputs:



To work with instantiations we will show that they admit a normal form where !-boxes are dealt with from the top down. We use a number of lemmas regarding commuting !-box operations which are found in appendix A.

Theorem 3.3.22. *Given an instantiation $i \in Inst(G)$ and a top-level !-box A in G (i.e one with no parent !-box), i can be rewritten as $\mathbf{rn} \circ i' \circ \mathrm{Kill}_A \circ \mathrm{Exp}_A^n$ where $\mathbf{rn}$ is a renaming and $i' \in Inst(\mathrm{Kill}_A \circ \mathrm{Exp}_A^n(G))$.*

Proof. Clearly Kill_A must already occur to the left of any instance of Exp_A, so it suffices to show that these two operations can be commuted to the right past operations on a different !-box B. If B is not nested in A, then by lemma A.1.1 we can reorder $\mathrm{Op}'_A \circ \mathrm{Op}_B = \mathbf{rn} \circ \mathrm{Op}_B \circ \mathrm{Op}'_A$ for some appropriate fresh renamings and a renaming $\mathbf{rn}$.
Otherwise B is nested in A, so we can use one of the following equations proved in lemma A.1.2 and lemma A.1.3 respectively:

- $\mathrm{Kill}_A \circ \mathrm{Op}_B(G) = \mathrm{Kill}_A(G)$
- $\mathrm{Exp}_A \circ \mathrm{Op}_B(G) = \mathbf{rn} \circ \mathrm{Op}_{\mathbf{fr}(B)} \circ \mathrm{Op}_B \circ \mathrm{Exp}_{A,\mathbf{fr}}(G)$

for some appropriate fresh renamings and a renaming $\mathbf{rn}$.
Any renamings introduced can be commuted to the left via:

$$\mathrm{Op}_{C,\mathbf{fr}_C} \circ \mathbf{rn} = \mathbf{rn} \circ \mathrm{Op}_{C,\mathbf{rn}^{-1} \circ \mathbf{fr}_C \circ \mathbf{rn}}$$

which is proved in lemma A.0.2. Repeating this procedure eventually results in all operations on A being at the right and all renamings, which can then be combined into one, being at the left. □

Notation 3.3.23. We will write KE_A^n as a shorthand for $\mathrm{Kill}_A \circ \mathrm{Exp}_A^n$.

Corollary 3.3.24. *Given a total order on !-box names, !-tensor instantiations admit a normal form, up to renaming.*

Proof. Given an instantiation of G we can chose the first (by the total order on !-box names) top-level !-box A, and move its operations (of the form KE_A^n) completely to the right, moving any renamings completely to the left. The rest is then an instantiation of $\mathrm{KE}_A^n(G)$, so we can repeat the process. Termination of this procedure can be shown since each step removes a !-box and only adds !-boxes with fewer levels of nesting. □

3.3.4 !-Tensor Equations

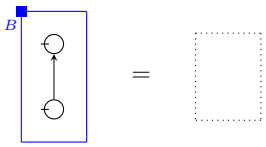
To properly reason with families of diagram equations we need to talk about equality of !-tensors. Recall the definition of well-formed tensor equations from definition 3.1.13:

Definition 3.3.25 (Tensor Equation)**.** A tensor equation $G = H$ is *well-formed* if tensors G and H have the same free inputs and the same free outputs.

To extend this definition to allow !-tensors we need to be careful about free edges and !-boxes. The intended interpretation of the !-tensor equation $G = H$ is that after applying any !-box operations to each side we are still left with a valid !-tensor equation. From this it follows that any concrete instantiation should result in a valid tensor equation. Hence we need free edges to always be expanded in the same way on each side of the equation. This property is encapsulated by the following definition.

Definition 3.3.26 (!-Tensor Equation)**.** $G = H$ is a *well-formed* !-tensor equation if the !-boxes are compatible i.e $\mathrm{boxes}(G) \triangle \mathrm{boxes}(H)$, G and H have the same free inputs and outputs, and those free edges have the same !-boxes appearing in their contexts in G as they do in H.

Remark 3.3.27. The definition of a well-formed !-tensor equation allows for !-boxes to appear on only one side of the equation. This may seem counter-intuitive, but one example of a useful equation with this property would be one allowing us to remove an arbitrary number of copies of some *scalar* (diagram with no inputs or outputs).



If desired we can always avoid this by including empty !-boxes. The above equation, for example, can have the right hand side 1 replaced with $[1]^B$.

Definition 3.3.28 (!-Box Operation on an Equation)**.** If the renaming **fr** is fresh for G, H we say it is fresh for the equation $G = H$. We can then apply a !-box operation $\mathrm{Op}_{B,\mathbf{fr}}$ to $G = H$ resulting in $\mathrm{Op}_{B,\mathbf{fr}}(G) = \mathrm{Op}_{B,\mathbf{fr}}(H)$.

Note that since G and H are allowed to have different !-boxes, it may be that $\mathrm{Op}_{B,\mathbf{fr}}$ acts as the identity on one side of the equation. Having defined !-box operations on !-tensor equations we can define instantiations of them in the same way as for !-tensors.

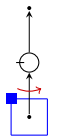
Definition 3.3.29 (Partial Instantiation)**.** A *partial instantiation* of a !-tensor equation is a sequence of zero or more $\mathrm{Kill}_B, \mathrm{Exp}_{B,\mathbf{fr}}$ operations, each with B in its domain and **fr** fresh for its domain.

Definition 3.3.30 (Instantiation, Concrete Instance)**.** An *instantiation* of a !-tensor equation $G = H$ is a partial instantiation i such that $i(G)$ and $i(H)$ are concrete tensors. We write $\mathrm{Inst}(G = H)$ for the set of instantiations of $G = H$ and call $i(G = H)$ a *concrete instance* of $G = H$.

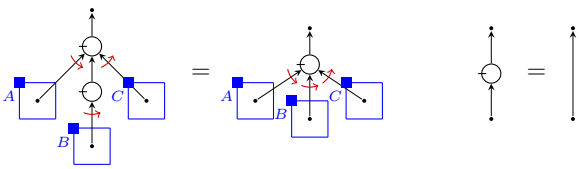
Partial instantiations take !-tensor equations to !-tensor equations, whereas instantiations take them to tensor equations.

Remark 3.3.31. Given $i \in \mathrm{Inst}(G = H)$ we know $i(G)$ is a concrete tensor so we may expect i to be an instantiation of G. This is not always the case since G and H can contain different !-boxes and hence some operations $\mathrm{Op}_{B,\mathbf{fr}}$ in i might not have B in their domain on G. These operations act as the identity and so we need not worry about them. We write $i|_G$ for i restricted to the operations $\mathrm{Op}_{B,\mathbf{fr}}$ which have B in their domain when applied to G. Hence for any $i \in \mathrm{Inst}(G = H)$ we have $i|_G \in \mathrm{Inst}(G)$ and $i|_H \in \mathrm{Inst}(H)$ and can write the equation $i(G = H)$ as an instantiation of both sides $i|_G(G) = i|_H(H)$.

For an example, let us start with the arbitrary arity version of the theory of a monoid. We have a single generator representing multiplication which allows an arbitrary number of inputs and has a single output:

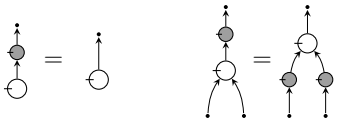


and two !-tensor equations:



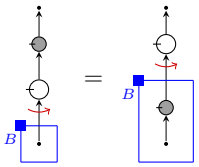
Monoid Laws

The first allows any two connected nodes to be combined, often referred to as a spider theorem, while the second replaces a node with a single input and a single output with an identity wire. We can extend this definition by adding a homomorphism  satisfying:



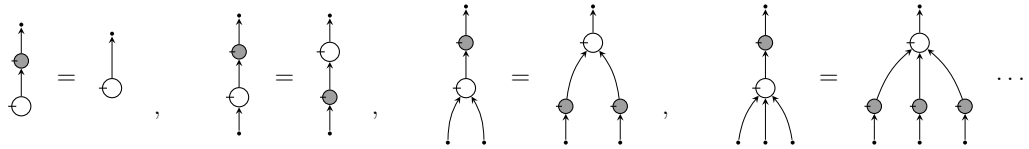
Homomorphism Laws

As suggested in conjecture 3.1.14 we believe that the homomorphism can pass through our arbitrary arity multiplication. We can now state this as a !-tensor equation:

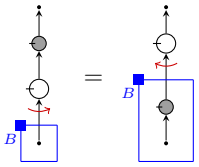


We can apply !-box operations such as Kill_B and $\mathrm{Exp}_{B,\mathbf{fr}}$ to this equation to get new !-tensor equations:

Repeated application results in the concrete instances:

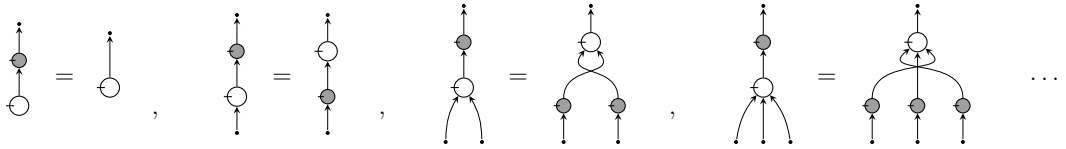


Similarly, in the theory of an antihomomorphism we hope to prove the theorem:

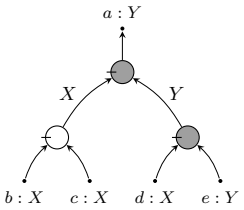


(3.6)

which leads to concrete instances with a twisting property:



Remark 3.3.32. At the beginning of this chapter we decided to ignore typing of edges in order to keep the definitions/proofs easy to understand. We now give a brief description of the changes to make to allow typing. Typed diagrams differ by having labelling on wires with their object type (both for free and bound wires).



To keep track of edge typing in tensor expressions we need edge names to carry a type. We do this by having an infinite set of edge names for each type. If our signature has generating object types $\mathcal{O} = \{X, Y, \ldots\}$, then we split the edge names $\mathcal{N}$ into disjoint infinite sets $\mathcal{N}_X, \mathcal{N}_Y, \ldots$ one for each generator. We define a typing function $\tau : \mathcal{N} \to \mathcal{O}$ by $\tau(a) = X$ where $a \in \mathcal{N}_X$.

From then on there are only a few minor alterations to definitions:

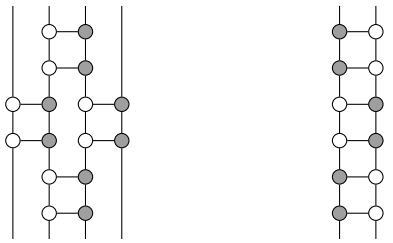
- To guarantee wires have unique types we add the restriction that $\tau(a) = \tau(b)$ for $1_{\hat{a}\check{b}}$ to be valid in a tensor expression.
- For canonically named tensors we now require a list of canonical names $\{a_1^X, a_2^X, \ldots\} \subset \mathcal{N}_X$ for each object type X. A tensor is *canonically named* if for some $n \in \mathbb{N}$, for each $i \leq n$ it has either an input or output a_i^X for some object X.
- !-Tensor expressions and !-tensors are unchanged (other than using the new typed edge names).
- To guarantee !-box operations respect edge typing we add an extra condition to being a fresh renaming ensuring **fr** preserves typing.

$$\tau(\mathbf{fr}(a)) = \tau(a) \quad \forall a \in \mathcal{N}$$

Remark 3.3.33. As with tensor equations, we prefer to track wires by position rather than edge names. We will also extend this to dropping !-box names whenever the correspondence is clear from position.

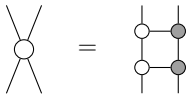
3.4 Definitional Extension

Families of diagrams with twisting are an obvious advantage of !-tensors, but another much more subtle advantage is the ability to definitionally extend a theory by defining a node to replace some larger structure. To see an example of the need for noncommutativity we will attempt such a definition in a commutative formalism. Take the following two diagrams where edges can freely commute around a node:

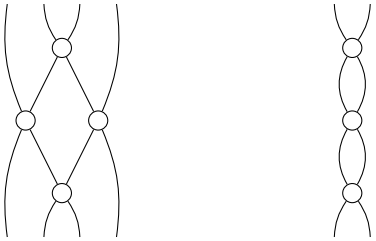


These have a lot of repeated structure. We see multiple instances of the following sub-diagram:

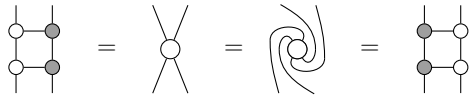
We might hope to define a new node $\phi : X \otimes X \to X \otimes X$ to replace it:



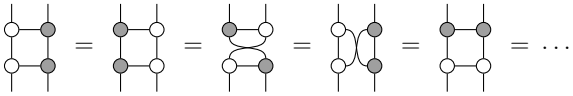
so that using the new notation the diagrams simplify to



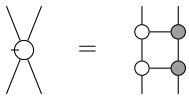
Unfortunately we have made a number of assumptions in our definition. In the second diagram we have the rotated version of the repeated structure, but this is not apparent in our new notation. Commutativity of the new node assumes we can move edges around as we want leading to an equality between two different diagrams.



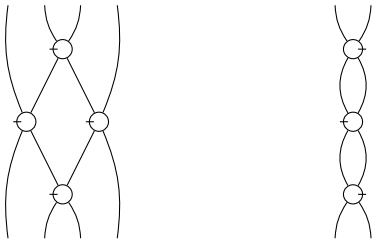
In fact we have assumed a large number of diagrams to be equal:



The problem here is that even though all generators are commutative, this does not imply commutativity of the diagrams they form. Noncommutativity lets us overcome this difficulty, allowing the following well-defined node to replace the repeated sections:



It is now clear that the tick identifies the two surrounding edges as those from white nodes and edges cannot be commuted. The earlier examples can now be simplified to:

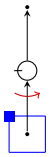


!-Tensors are the only formalism allowing families of diagrams and definitional extension of a theory.

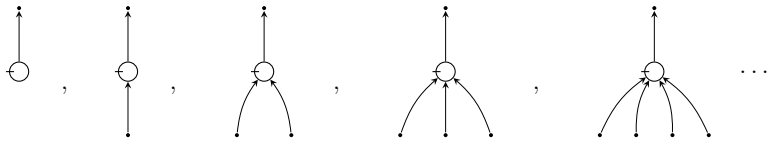
3.4.1 Recursive Definitions

One particularly useful application of definitional extension is the ability to replace fixed arity generators in a theory with arbitrary arity versions. Taking the example of a monoid, we would like to define an arbitrary arity node to represent the left associated trees of multiplication operations:

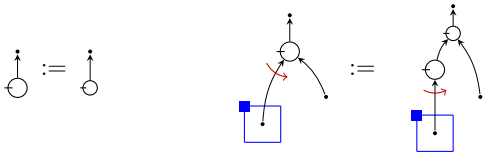
Then the !-tensor:



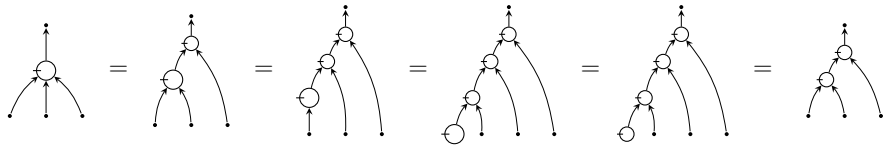
represents the family of such nodes with an arbitrary number of inputs. Concrete instances are retrieved via kill and expand operations.



The use of Kill and Exp to retrieve concrete instances suggests a more precise method for defining arbitrary arity nodes. We can give a recursive definition of the !-tensor G by defining a base case $\text{Kill}_B(G)$ and then defining $\text{Exp}_{B,fr}(G)$ in terms of G, i.e:

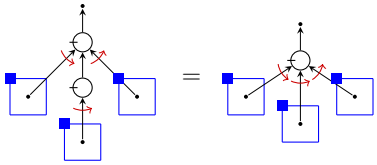


Here the base case is defined to be the monoidal unit  , then extra inputs are added via the monoidal multiplication  . We can check that this works as expected by unfolding the definition for a node with three inputs:



where the last step is by the unit law.

We can use the new arbitrary arity nodes to make statements such as the following, claiming that any two connected nodes can be combined (often referred to as a spider theorem):



which we will prove from the recursive definition in section 4.4 once we have developed a formal logic for working with !-tensors. We will also prove rules such as the following for homomorphisms and antihomomorphisms respectively.

3.5 Encoding !-Tensors as !-Graphs

In this section we demonstrate how !-tensors can be encoded as !-graphs with some additional data to keep track of the order of edges around nodes. This is of particular interest for adapting the semi-automated theorem prover Quantomatic to work with non-commutative theories. We will elaborate on this idea in section 7.1.1, but for now we simply present the mechanics of the encoding.

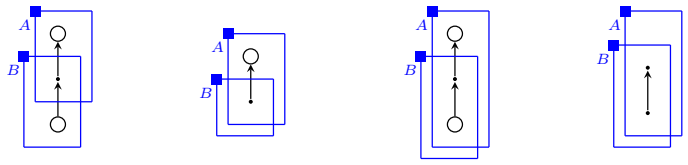
3.5.1 Simple Overlap and Neighbourhood Orders

We know from theorem 3.3.2 that the !-boxes of a !-tensor form a directed forest, but definition 2.3.8 tells us that for !-graphs they are merely posetal. Hence if we wish to encode !-tensors as !-graphs with additional data, we know that the result will be in a proper subset of all possible !-graphs. We define this restriction, which we refer to as simple overlap, by enforcing that the contents of !-boxes only overlap on the interiors of wires along with a condition to ensure they are consistent at boundaries.

Definition 3.5.1 (Simple Overlap)**.** Given a pair of non-nested !-boxes B and B', we say they *overlap simply* if $C(B) \cap C(B')$ consists of only the interior of zero or more wires, where at least one endpoint is a node-vertex and any node-vertex endpoints are in either $C(B)$ or $C(B')$.

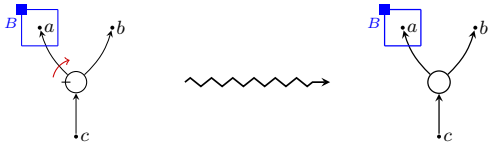
Definition 3.5.2 (!-Graph with Simple Overlap)**.** A !-graph where any two non-nested !-boxes overlap simply, is called a *!-graph with simple overlap*.

This is not quite the same as the notion of trivial overlap as defined in [31] where it was shown that !-graphs with trivial overlap (BGTO) can be encoded using a context-free grammar. The difference here is that we allow non-nested overlap at a boundary rather than just between two node-vertices.

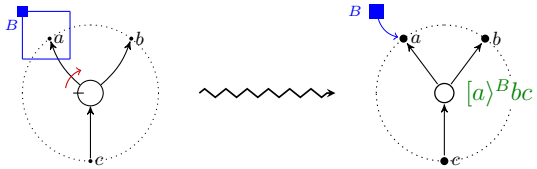


Here, the first two !-graphs have only simple overlap, though the second is non-trivial; the final two have non-simple overlap.

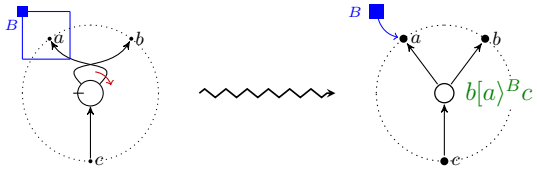
Now we wish to encode !-tensors as !-graphs without losing any information. It is clear that edge orders and expansion directions are lost when naively converting a !-tensor to a !-graph.



Taking a !-tensor expression for the left hand side, $\phi_{[\hat{a}\rangle^B \hat{b}\check{c}}[]^B$, we can see that all of the lost information is stored in ϕ's edgeterm $[\hat{a}\rangle^B \hat{b}\check{c}$. Imagine encoding a single node as a !-graph (as in definition 2.3.8). It is clear that adding the edgeterm as additional data ensures no information is lost. In fact we no longer even require the hats and checks since it is clear from the !-graph node which edges are inputs and which are outputs. we see this from the following diagrams of single nodes where we fixed edge positions and draw edgeterms on nodes.



We can check that a different !-tensor has a different encoding. Even though the following !-tensor results in the same !-graph as above, the extra data can distinguish them.



Hence we wish to encode !-tensors as !-graphs where each node has an associated edgeterm over its adjacent edges. We refer to this edgeterm information as a *neighbourhood order* on the !-graph.

Definition 3.5.3 (Neighbourhood Order)**.** Given a !-graph with simple overlap G, a *neighbourhood order* on G is a function $\mathrm{nhd} : N(G) \to \mathcal{E}_{W(G)}$ satisfying $\forall v \in N(G)$:

- $\mathrm{nhd}(v)$ is an edgeterm with edge names $\{w \in W(G) : w \to v \vee v \to w\}$,
- The !-boxes in $\mathrm{ectx}_{nhd(v)}\, a$ are precisely those with edges to a but not to v (ordered by nesting order).

Our intention is to encode !-tensors as pairs (G, nhd) where G is a !-graph (with simple overlap) and nhd is a neighbourhood order on G. We need to expand our five !-box operations to work on these !-graphs with neighbourhood orders. We have already seen the definitions of most !-box operations on !-graphs in definition 2.3.10. From !-tensors we gained one additional operation called Flip_B, but this is related to expansion directions so we decide that it acts as the identity map on !-graphs. We also know how to apply !-box operations to edgeterms from definition 3.3.10.

Definition 3.5.4 (!-Box Operations)**.** For Op_B any !-box operation and (G, nhd) a !-graph with neighbourhood order,

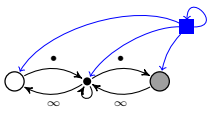
$$\mathrm{Op}_B(G, \mathrm{nhd}) := (\mathrm{Op}_B(G), \mathrm{Op}_B \circ \mathrm{nhd})$$

3.5.2 The Encoding Map

We now wish to define a map $\mathcal{I}$ taking !-tensors for some signature to !-graphs with neighbourhood orders. To do this we first need to fix the signatures. Recall that !-tensors are defined based on a compact closed signature (definition 3.1.1) whereas !-graphs are based on a compressed monoidal signature (definition 2.3.1). The former list all concrete morphisms as words over X, X^* (using compact closed structure) while the latter lists families of morphisms with seperate domains and codomains over $(X, \bullet), (X, \infty)$ where ∞-tagged objects represent variable arity. These are clearly not equivalent, so we will fix a compact closed signature Σ and a corresponding compressed monoidal signature $\mathcal{I}(\Sigma)$, to work with in the remainder of this section.

Suppose we have morphisms of the form (ϕ, w) where ϕ is the morphism's name and w is a word over the set $\{X, X^*\} \times \{\bullet, \infty\}$ (i.e. families of words in X, X^*). From this we can define the compact closed signature Σ to have morphisms $(\phi, i(w))$ for each $i(w)$, concrete instances of w (i.e. (X, ∞) can be replaced by any number of X's). We define the compressed monoidal signature $\mathcal{I}(\Sigma)$ to have morphisms ϕ with $\mathrm{cod}(\phi) = w|_X$ (i.e only the output parts of w) and $dom(\phi) = w|_{X^*}$ (i.e the input parts of w written as a word over X). For the rest of the section we will suppose we have such a pair of well-defined signatures Σ and $\mathcal{I}(\Sigma)$.

For an example, take the signature for a bialgebra. This can be represented as a compact closed signature Σ with one morphism $(\phi, X(X^*)^i)$ for each $i \in \mathbb{N}$ and one morphism $(\psi, (X)^i X^*)$ for each $i \in \mathbb{N}$. The corresponding compressed monoidal category $\mathcal{I}(\Sigma)$ has (families of) morphisms $\phi : [X^\infty] \to [X^\bullet]$ and $\psi : [X^\bullet] \to [X^\infty]$. Drawing ϕ and ψ as white and gray nodes respectively this results in !-typegraph $\mathcal{G}_{\mathcal{I}(\Sigma)!}$ of the form:



We now start by defining a map taking a !-tensor expression for signature Σ to an $\mathcal{I}(\Sigma)$-typed !-graph, as described in section 2.3.2. The !-tensor formalism avoids the need to name nodes, but we will need to assign them with unique names if we want to keep track of them during the conversion. We do this using an indexing set. Given a !-tensor expression G, we say J indexes the nodes if the elements $j \in J$ are in 1-to-1 correspondence with the nodes in G (i.e. the subexpressions of the form ϕ_e). If j corresponds to a node ϕ_e we will write N_j to refer to the expression ϕ_e.

Example 3.5.5. We can index the !-tensor expression $G = \phi_{\langle \check{a}]^A}[\phi_{\hat{a}[\check{b}\rangle^B}[1]^B]^A 1_{\hat{d}\check{c}}$ using the natural numbers reading them from left to right so that $J = \{1, 2\}$ and the corresponding nodes are:

$$N_1 = \phi_{\langle \check{a}]^A} \qquad\qquad N_2 = \phi_{\hat{a}[\check{b}\rangle^B}$$

Note that not all atomic subexpressions are indexed, only those representing a node.

In converting a !-tensor expression to a !-graph, the node N_j can be labelled j.

Definition 3.5.6 (Encoding on Expressions)**.** We define $\mathcal{I}$ taking !-tensor expressions to !-graphs with neighbourhood orders recursively by:

- $\mathcal{I}(1) := \{\}$
- $\mathcal{I}(1_{\hat{b}\check{a}}) := \{a \to b\}$
- $\mathcal{I}(\phi_e = N_j) := \{a \to j : \check{a} \in e\} \cup \{j \to a : \hat{a} \in e\} \cup \{B \to a : B \in \mathrm{ectx}_e(a)\}$
- $\mathcal{I}(GH) := \mathcal{I}(G) \cup \mathcal{I}(H)$
- $\mathcal{I}([G]^B) := \mathcal{I}(G) \cup \{B \to x : x \in U(\mathcal{I}(G))\} \cup \{B \to B' : B' \leq B\}$

The typing function on vertices maps $j \to \phi$ where $N_j = \phi_e$, !-boxes to !, and edge names a to their edge type $\tau(a)$. The neighbourhood order is $nhd(j) := e'$ for each $N_j = \phi_e$ where e' is e without the hats or checks.

Example 3.5.7. If we index $G = \phi_{\langle \check{a}]^A [\phi_{\hat{a}[\check{b}\rangle^B}[1]^B]^A 1_{\hat{d}\check{c}}}$ using $J = \{1, 2\}$ as in the previous example, the atomic subexpressions become:

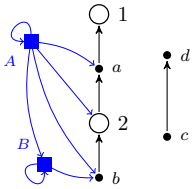
$\mathcal{I}(N_1) =$ 

$\mathcal{I}(N_2) =$ 

$\mathcal{I}(1) =$

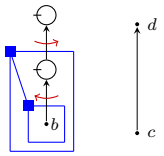
$\mathcal{I}(1_{\hat{d}\check{c}}) =$ 

Going through the recursive definition, these combine to the !-graph we would expect:

$\mathcal{I}(\phi_{\langle \check{a}]^A [\phi_{\hat{a}[\check{b}\rangle^B}[1]^B]^A 1_{\hat{d}c}}) =$ 

With neighbourhood order $nhd = [1 \mapsto \langle a]^A, 2 \mapsto a[b\rangle^B]$.

It is easy to see how this encodes the !-tensor:



Now we verify the definition acts as expected.

Theorem 3.5.8. *Given $G \in \mathcal{T}_\Sigma$, $\mathcal{I}(G)$ is a $\mathcal{G}_{\mathcal{I}(\Sigma)!}$-typed !-graph with simple overlap and a neighbourhood order.*

Proof. Let J index the nodes in G.

- We first go through the four conditions of definition 2.3.8 to check we have a $\mathcal{G}_{\mathcal{I}(\Sigma)!}$-typed !-graph:

  BG1: We wish to show $U(\mathcal{I}(G))$ is a $\mathcal{G}_{\mathcal{I}(\Sigma)}$-typed string graph. To do this we check the typing function is arity matching and wire-vertices have at most one incoming edge and at most one outgoing edge.

  Given $\phi_e = N_j$, we must have a morphism (ϕ, w) in $\mathcal{I}(\Sigma)$ where concrete instances of e are instances of w. Hence the $\bullet$-edges adjacent to j in $U(\mathcal{I}(G))$ come from (and hence are in bijection with) the $\bullet$-tagged objects in w. Now from the definition of a derived typegraph, $\mathcal{G}_{\mathcal{I}(\Sigma)}$ contains a node ϕ with $\bullet$-edges to/from the $\bullet$-tagged objects in w. Hence we have a bijection between the $\bullet$-edge neighbourhoods of j (in $U(\mathcal{I}(G))$) and ϕ (in $\mathcal{G}_{\mathcal{I}(\Sigma)}$) as required.

From the definition of $\mathcal{I}$, wire vertices come directly from edge names. The unique (possible) occurrence of $\hat{a}$ results in a wire $\_ \to a$ and the unique (possible) occurrence of $\check{a}$ results in a wire $a \to \_$. Hence each a can have at most one incoming and one outgoing edge.

BG2: The full subgraph $!(G)$ is the reflexive, transitive closure of the 'child of' relation on !-boxes. Hence it is reflexive, transitive, and antisymmetric (the 'child of' relation is cycle-free).

BG3: Take $B \in !(G)$ and write X for $U(C(B))$, so we need to show that X is an open subgraph of $U(G)$. We first show that any wire-vertex adjacent to a vertex in X is in X. From the definition of $\mathcal{I}$, the only time an edge is added from a box-vertex to a node-vertex j, there are also edges added to each neighbour of j. Any two adjacent wire-vertices must come from $\mathcal{I}(1_{\hat{a}\check{b}})$, so that $B \to a \Leftrightarrow B \to b$.

Incident edges can only come from node-vertices not in X with adjacent wire-vertices in X. Hence the wire comes from a directed edge with B in its edge context which means it is ∞-tagged in the typegraph.

BG4: An edge between box-vertices, $A \to B$ must be added by $\mathcal{I}([H]^A)$ during the recursive definition, where B and $C(B)$ are already in $\mathcal{I}(H)$. Hence we get edges $A \to x$ for all $x \in C(B)$ meaning $C(B) \subseteq C(B')$.

- Next we check that any overlap of non-nested !-boxes B, B' is simple.

  Suppose j is a node-vertex such that $B \to j$ and $B' \to j$. From the definition of $\mathcal{I}$ we see that B and B' must appear in the node context of N_j and so one is nested inside the other. This contradiction proves that $C(B) \cap C(B')$ contains only wire vertices.

  Suppose a is a wire-vertex such that $B \to a$ and $B' \to a$. If a is adjacent to another wire-vertex b, then this edge must have come from $1_{\hat{b}\check{a}}$ or $1_{\hat{a}\check{b}}$ so a and b are nested inside the same !-boxes. Hence the intersection $C(B) \cap C(B')$ contains only the interiors of zero or more wires.

  Suppose both endpoints are wire-vertices. Hence the edges have come from identity wires $1_{\hat{b}\check{a}}$ which must have B and B' in its context which would require them to be nested.

  Suppose a wire-vertex $a \in C(B) \cap C(B')$ is adjacent to the node-vertex j. If j was neither in $C(B)$ nor $C(B')$, then both B and B' must occur in the edge context of a and hence would be nested. We conclude that j must occur in exactly one of $C(B)$ or $C(B')$.

- Finally, we wish to check that nhd is a neighbourhood order on $\mathcal{I}(G)$. For $N_j = \phi_e$ the incoming edges a to j come from all inputs $\check{a}$ in e and the outgoing edges b come from all outputs $\hat{b}$ in e, hence the neighbourhood of j is made up of edge names in e as required. Also,

for the node $N_j = \phi_e$, !-boxes with edges to a (an edge name in e) but not to j are exactly those which appear in $\mathrm{ectx}_e(a) = \mathrm{ectx}_{\mathrm{nhd}(\phi)}(a)$.

□

Hence $\mathcal{I}$ takes a !-tensor expression and returns a correctly typed !-graph with a neighbourhood order. By the following theorem, the definition of $\mathcal{I}$ can be lifted from specific !-tensor expressions to !-tensors (equivalence classes of !-tensor expressions).

Theorem 3.5.9. *$\forall G, H \in \mathcal{T}_\Sigma$, $G \equiv H$ if and only if $\mathcal{I}(G)$ and $\mathcal{I}(H)$ are equivalent up to renaming and wire homeomorphism.*

Proof. In both formalisms bound variables can be renamed freely, so we will not worry about names here. We prove first that all !-tensor equivalences from definitions 3.2.1 and 3.1.6 are preserved through $\mathcal{I}$. Then we check that if $\mathcal{I}(G)$ and $\mathcal{I}(H)$ are equivalent up to wire homeomorphism then $G \equiv H$.

- Since edgeterms are copied directly from !-tensors to !-graphs, the edgeterm equivalences are still preserved. The associativity, commutativity, and identity equivalences on !-tensor expressions have no affect on the graphical formalism so these are also preserved. The only things left to check are the two equivalences involving inserting identity wires $1_{\hat{b}\check{a}}$ using concatenation. These two conditions come down to wire homeomorphism in the !-graph framework. First we are given that $\check{b}$ exists in G and (for some !-boxes $B_1, \ldots, B_n$) we look at $G * [[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n}$ under $\mathcal{I}$. This becomes a graph with edges $a \to b \to x$ for some x and since a and b are both wire-vertices this is wire homeomorphic to $G[\check{b} \mapsto \check{a}]$. The other case is similar but with arrows reversed.

- For the other direction suppose $\mathcal{I}(G)$ and $\mathcal{I}(H)$ differ only by a single wire homeomorphism, so there exist wire-vertices a, b and a vertex c in $\mathcal{I}(G)$ with $a \to b \to c$, and $\mathcal{I}(H)$ is $\mathcal{I}(G)$ but with $a \to b \to c$ replaced with $a \to c$. From the definition of $\mathcal{I}$ we see that there exists some G' and !-boxes $B_1, \ldots, B_n$ such that $G = G' * [[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n}$ (this is the only way two wire vertices can be connected) and also $\check{b}$ must exist in G'. H must be the same as G' except the edge $\check{b}$ is replaced by the edge $\check{a}$.

$$\begin{aligned} G &\equiv G' * [[1_{\hat{b}\check{a}}]^{B_1} \ldots]^{B_n} \\ &\equiv G'[\check{b} \mapsto \check{a}] \\ &\equiv H \end{aligned}$$

□

We have shown $\mathcal{I}$ is injective on !-tensors and hence is a bijection onto its image. We can hence take any !-tensor G and work with it in the form of $\mathcal{I}(G)$. To work as an encoding we would hope that !-box operations are equivalent in each formalism. Applying the !-box operation Op_B to $\mathcal{I}(G)$ and then returning to the !-tensor formalism should result in a !-tensor equivalent to $\mathrm{Op}_B(G)$. By the previous theorem, $\mathcal{I}$ has a left inverse $\mathcal{I}^{-1}$ so we need only check that $\mathrm{Op}_B \circ \mathcal{I} = \mathcal{I} \circ \mathrm{Op}_B$ and it then follows that $\mathcal{I}^{-1} \circ \mathrm{Op}_B \circ \mathcal{I} = \mathrm{Op}_B$.

Theorem 3.5.10. $\mathrm{Op}_B(\mathcal{I}(G)) = \mathcal{I}(\mathrm{Op}_B(G))$ *for any !-box operation* Op_B *and* $G \in \mathcal{T}_\Sigma$.

Proof. This can be shown by case analysis on the recursive definition of $\mathcal{T}_\Sigma$ going through each !-box operation Copy_B, Flip_B, Kill_B, and Drop_B (from which Exp_B also follows). Most cases are trivial, the interesting case is showing $\mathrm{Op}_B(\mathcal{I}([G]^B)) = \mathcal{I}(\mathrm{Op}_B([G]^B))$ for each operation:
$\mathrm{Copy}_B(\mathcal{I}([G]^B))$ is defined by a pushout of the inclusion $1 \hookrightarrow \mathcal{I}([G]^B)$ with itself, so equals the disjoint union of two copies of $\mathcal{I}([G]^B)$. Hence if $\mathbf{fr}$ assigns fresh names to to the edges and !-boxes of $[G]^B$, then:

$$\begin{aligned}
\mathrm{Copy}_B(\mathcal{I}([G]^B)) &= \mathcal{I}([G]^B) \cup \mathcal{I}([\mathbf{fr}(G)]^{\mathbf{fr}(B)}) \\
&= \mathcal{I}([G]^B[\mathbf{fr}(G)]^{\mathbf{fr}(B)}) \\
&= \mathcal{I}(\mathrm{Copy}_B([G]^B)) \\
\mathrm{Flip}_B(\mathcal{I}([G]^B)) &= \mathcal{I}([G]^B) \\
&= \mathcal{I}(\mathrm{Flip}_B([G]^B)) \\
\mathrm{Kill}_B(\mathcal{I}([G]^B)) &= \mathrm{Kill}_B(\mathcal{I}(G) \cup \{B \to x : x \in U(\mathcal{I}(G))\} \cup \{B \to B' : B' \leq B\}) \\
&= \{\} \\
&= \mathcal{I}(1) \\
&= \mathcal{I}(\mathrm{Kill}_B([G]^B)) \\
\mathrm{Drop}_B(\mathcal{I}([G]^B)) &= \mathrm{Drop}_B(\mathcal{I}(G) \cup \{B \to x : x \in U(\mathcal{I}(G))\} \cup \{B \to B' : B' \leq B\}) \\
&= \mathcal{I}(G) \\
&= \mathcal{I}(\mathrm{Drop}_B([G]^B))
\end{aligned}$$

□

Corollary 3.5.11. *The concrete instances of* $\mathcal{I}(G)$ *are the concrete instances of* G *with* $\mathcal{I}$ *applied to them.*

Chapter 4

A Formal Logic

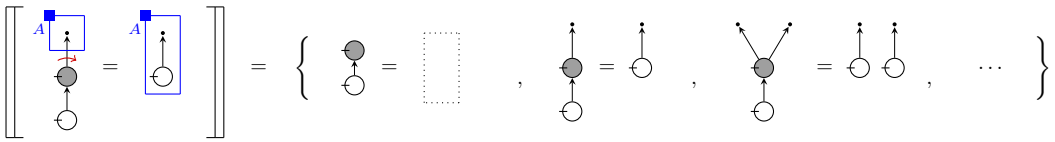
The second major contribution of this thesis is !-logic, a logic for !-tensors. Previous attempts at diagrammatic reasoning have relied on rewriting via substitution. The equation $L = R$ implies we can replace any instance of L as a section of a diagram with R. We replace the previous methods for equational reasoning (which lacked a formal logical framework) with a first order logic with conjunction, implication, and universal quantification over !-boxes. In section 4.2 we present the !-logic rules (referred to as !L), culminating in a !-box induction rule.

After looking through a few examples of proof by induction for recursively defined nodes, we conclude this chapter by defining a semantics for !-logic and proving soundness with respect to this semantics.

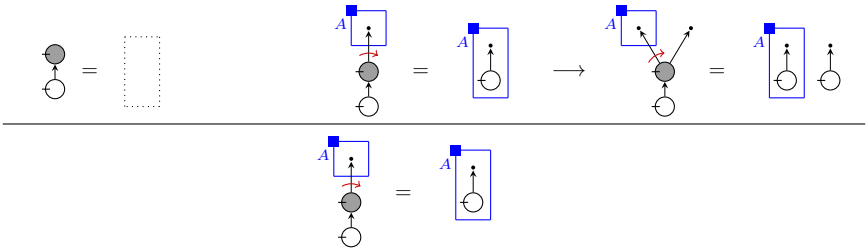
4.1 !-Formulas

4.1.1 Quantification

In [39] an attempt was made to develop a formal logic for !-graphs. A !-graph equation $G = H$ was taken to mean that any instantiation of that equation holds. Suppose we take the same definition for !-tensor equations.



To prove this equation from some set of concrete axioms we may hope to use a form of induction:



(Induct)

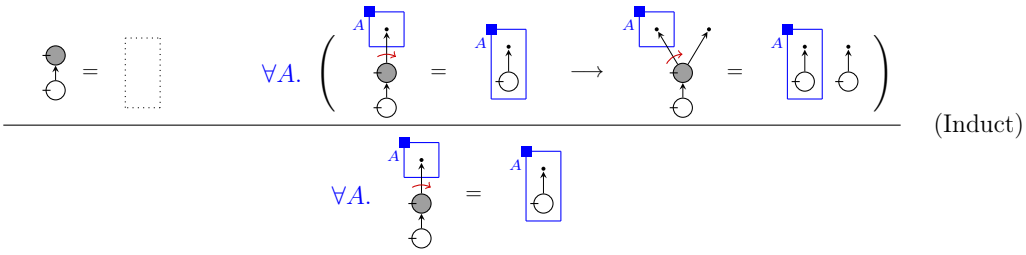
Here we have split into a base case with no copies of A and a step case where n copies implies $n+1$ copies. Unfortunately, interpreting $\rightarrow$ as 'the set of instances on the left implies the set on the right' we find that this induction is invalid. The step case turns out to be vacuous, as the instances of the right hand side are a subset of those of the left hand side. This is analogous to trying to apply induction to first order logic by the rule:

$$\frac{P(0) \qquad \forall n\ P(n) \quad \longrightarrow \quad \forall n\ P(n+1)}{\forall n\ P(n)} \text{ (Induct)}$$

It is clear that quantification is playing an important role here. The corrected version of induction for first order predicate logic is:

$$\frac{P(0) \qquad \forall n \left(\ P(n) \ \longrightarrow \ P(n+1) \ \right)}{\forall n\ P(n)} \text{ (Induct)}$$

We have corrected the induction by pulling the quantification outside of the step case implication. Hence in the quest for !L we no longer take a !-tensor equation to mean truth of its instances, instead introducing quantification over !-boxes. We hope to correct the previous attempt at !-box induction by analogy with first order logic, where now the left hand side of the implication does not assume truth of every instance.



(Induct)

However, things get more interesting when we bring !-box nesting into the picture. If G and H have a single !-box B, then it is clear that $\forall B.(G = H)$ should be interpreted as 'any instantiation i of B (i.e. KE_B^n for some $n \in \mathbb{N}$) results in the correct concrete equation $i(G) = i(H)$'. It is less clear how we should interpret quantification over B if it is nested inside another !-box A. B is implicitly bound to operations applied to A which create new copies of, and eventually delete, B. Hence if we want to quantify over B we must also quantify over any parents and children. So quantification is done over connected components of the !-box structure. Any such component can be written as $\downarrow A$ for some !-box A and so we simply write $\forall A.X$ for quantification over the component $\downarrow A$ in X.

We make the interpretation of quantification formal when we describe semantics in section 4.5. First we define !-formulas which are the building blocks of our logic.

4.1.2 !-Formulas

Definition 4.1.1 (!-Formulas)**.** The set of *!-formulas*, $\mathcal{F}_\Sigma$, for a signature Σ is defined inductively as:

$$\begin{aligned}
&\bullet\ G = H \in \mathcal{F}_\Sigma && G, H \in \mathcal{T}_{\Sigma!},\ G = H \text{ well-formed} \\
&\bullet\ X \wedge Y \in \mathcal{F}_\Sigma && X, Y \in \mathcal{F}_\Sigma,\ \mathrm{boxes}(X) \triangle \mathrm{boxes}(Y) \\
&\bullet\ X \to Y \in \mathcal{F}_\Sigma && X, Y \in \mathcal{F}_\Sigma,\ \mathrm{boxes}(X) \triangle \mathrm{boxes}(Y) \\
&\bullet\ \forall A.X \in \mathcal{F}_\Sigma && X \in \mathcal{F}_\Sigma,\ A \in \mathrm{boxes}(X)^\top
\end{aligned}$$

where boxes$(-)$ is defined recursively on !-formulas by:

- $\mathrm{boxes}(G = H) := \mathrm{boxes}(G) \cup \mathrm{boxes}(H)$
- $\mathrm{boxes}(X \wedge Y) := \mathrm{boxes}(X) \cup \mathrm{boxes}(Y)$
- $\mathrm{boxes}(X \to Y) := \mathrm{boxes}(X) \cup \mathrm{boxes}(Y)$
- $\mathrm{boxes}(\forall A.X) := \mathrm{boxes}(X) \setminus \downarrow A$

Remark 4.1.2. Note that the set $\mathcal{F}_\Sigma$ is defined inductively, relying on a simultaneous recursive definition of boxes$(-)$. This is non-circular, since the inductive steps always rely on calls to boxes$(-)$ on strictly smaller formulas. Unsurprisingly, this style of definition is referred to as induction-recursion [17].

Remark 4.1.3. The function boxes$(-)$ returns only the free !-boxes of a formula. Quantification $\forall A.X$ means the !-boxes in $\downarrow A$ are now bound. As with bound edge names this means we can use α-conversion to rename them.

Definition 4.1.4 (Free Edges)**.** We write free(X) for the set of *free edges* in X (i.e. the union of free$(-)$ applied to each atomic part of X).

To instantiate !-formulas we need to be able to apply !-box operations to them.

Definition 4.1.5 (Fresh for !-Formulas)**.** We say $\mathbf{fr}$ is *fresh for* X if it is a fresh renaming on $(\mathrm{boxes}(X), \mathrm{free}(X))$.

Definition 4.1.6 (Operations on $\mathcal{F}_\Sigma$)**.** For $\mathrm{Op}_{B,\mathbf{fr}}$ one of the !-box operations Flip_B, Kill_B, Drop_B,

$\mathrm{Copy}_{B,\mathbf{fr}}$ or $\mathrm{Exp}_{B,\mathbf{fr}}$ and $\mathbf{fr}$ fresh for the domain:

- $\mathrm{Op}_{B,\mathbf{fr}}(G = H) := \mathrm{Op}_{B,\mathbf{fr}}(G) = \mathrm{Op}_{B,\mathbf{fr}}(H)$
- $\mathrm{Op}_{B,\mathbf{fr}}(X \wedge Y) := \mathrm{Op}_{B,\mathbf{fr}}(X) \wedge \mathrm{Op}_{B,\mathbf{fr}}(Y)$
- $\mathrm{Op}_{B,\mathbf{fr}}(X \to Y) := \mathrm{Op}_{B,\mathbf{fr}}(X) \to \mathrm{Op}_{B,\mathbf{fr}}(Y)$
- $\mathrm{Op}_{B,\mathbf{fr}}(\forall A.X) := \begin{cases} \forall A.X & B \in\!\downarrow\! A \\ \forall A.\, \mathrm{Op}_{B,\mathbf{fr}'}(X) & B \notin\!\downarrow\! A \end{cases}$

Remark 4.1.7. In the recursive definition of $\mathrm{Op}_{B,\mathbf{fr}}(\forall A.X)$ for $B \notin\!\downarrow\! A$ we have used a new fresh renaming $\mathbf{fr}'$. The reason for this is that we now need it to be fresh for the contents of the !-boxes in $\downarrow A$ even though it will not be applied to them as they are necessarily not nested inside B. By α-equivalence we can take A not to share any names with support($\mathbf{fr}$), then we can always extend $\mathbf{fr}$ to be fresh on $\downarrow A$.

Theorem 4.1.8. *!-Box operations preserve the property of being a !-formula.*

Proof. We prove this using structural induction on !-formulas.

- If $G = H$ is a !-formula then G and H have the same free edges in the same !-boxes. Hence $\mathrm{Op}_B(G)$ and $\mathrm{Op}_B(H)$ have the same free edges (a or $\mathbf{fr}(a)$ for a free in $G = H$) and these are in the same !-boxes.
- For the next two cases we have boxes(X) and boxes(Y) compatible. Op_B takes the unique connected component S containing B and replaces it with $\mathrm{Op}_B(S)$. This can only have gained fresh !-box names so boxes($\mathrm{Op}_B(X)$) and boxes($\mathrm{Op}_B(Y)$) are still compatible.
- If $B \in\!\downarrow\! A$ then the final case is trivial. If $B \notin\!\downarrow\! A$ then the component $\downarrow A$ is not affected by Op_B so is still a component of $\mathrm{Op}_B(X)$.

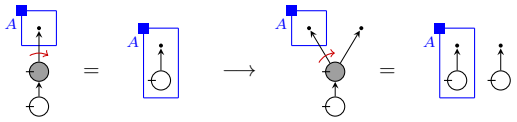
□

Having defined !-box operations on !-formulas we can define the set of instantiations, resulting in concrete formulas, as expected.

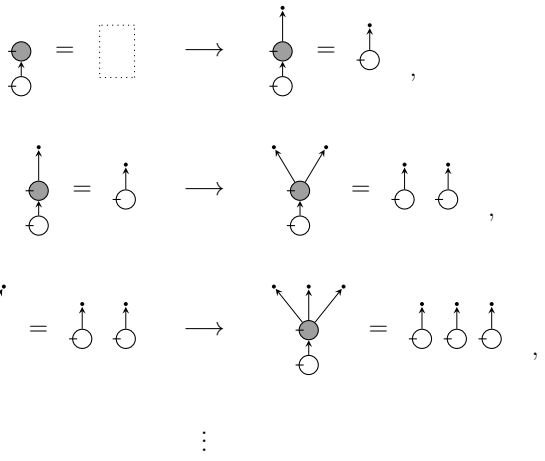
Definition 4.1.9 (Partial Instantiation)**.** A *partial instantiation* of a !-formula is a sequence of zero or more $\mathrm{Kill}_B, \mathrm{Exp}_{B,\mathbf{fr}}$ operations each with B in its domain and $\mathbf{fr}$ fresh for its domain.

Definition 4.1.10 (Instantiation, Concrete Instance)**.** An *instantiation* of a !-formula X is a partial instantiation i such that $i(X)$ is a concrete formula. We write $\mathrm{Inst}(X)$ for the set of instantiations of X and call $i(X)$ a *concrete instance* of X.

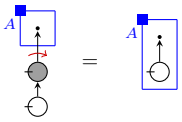
Example 4.1.11. The following !-formula (taken from the inductive step of our proposed !-box induction):



has concrete instances:



As we might expect, and will see later, quantifying over A means requiring each of these to be true, which when combined with a base case clearly implies truth of every concrete instance of:



We have seen how !-box operations can be applied to a !-formula X. We now show that any combination of !-box operations i resulting in a concrete !-formula $i(X)$ can be rewritten as an instantiation, up to renaming. We use a number of lemmas from appendix A concerning reordering !-box operations.

Theorem 4.1.12. *Given a sequence i of !-box operations such that $i(X)$ is a concrete !-formula, $\exists i' \in Inst(X)$ and a renaming* **rn** *such that $i(X) = \mathbf{rn} \circ i'(X)$.*

Proof. First we note that any operations Op_B without B in their domain when applied to X act trivially and can hence be removed. Similarly, Drop_B operations can be removed by rewriting $\mathrm{Drop}_B = \mathbf{rn} \circ \mathrm{Kill}_B \circ \mathrm{Exp}_{B,fr}$ for some renaming **rn** as shown in lemma A.2.2. Note that any renamings introduced can be pushed to the left by lemma A.0.2 which states:

$$\mathbf{rn} \circ \mathrm{Op}_{\mathbf{rn}^{-1}(C),\mathbf{rn}^{-1} \circ \mathbf{fr} \circ \mathbf{rn}}(G)$$

Now suppose j is a combination of Kill, Exp, Copy, and Flip operations such that $j(X)$ is a concrete !-formula. We wish to show that j can be rewritten to an instantiation, that is, a composition of only Kill and Exp operations. We do this by repeatedly applying the following rewriting procedure

to j:

If j has no Copy or Flip operations then we are done, else we go through one of the following four cases based on the left-most Copy or Flip operation:

- $\mathrm{Copy}_{B,\mathbf{fr}}$ is the left-most such operation with $j = i \circ \mathrm{Copy}_{B,\mathbf{fr}} \circ j'$ and B has a parent !-box in $\mathrm{Copy}_{B,\mathbf{fr}} \circ j'(X)$:

  Let A be the top level !-box containing B. By theorem 3.3.22 we can write $i = \mathbf{rn} \circ i' \circ \mathrm{KE}_A^n$ for a renaming $\mathbf{rn}$, so that:

$$\begin{aligned} j &= i \circ \mathrm{Copy}_{B,\mathbf{fr}} \circ j' \\ &= \mathbf{rn} \circ i' \circ \mathrm{KE}_A^n \circ \mathrm{Copy}_{B,\mathbf{fr}} \circ j' \\ &= \mathbf{rn} \circ i' \circ \mathrm{Copy}_{B_1} \circ \ldots \circ \mathrm{Copy}_{B_n} \circ \mathrm{KE}_A^n \circ j' \end{aligned}$$

  where $B_1, \ldots B_n$ are the copies of B created by KE_A^n and the final step is by repeated use of lemma A.2.3.

  Now we can restart the procedure on $i' \circ \mathrm{Copy}_{B_1} \circ \ldots \circ \mathrm{Copy}_{B_n} \circ \mathrm{KE}_A^n \circ j'$, knowing that the new left-most copying operation Copy_{B_1} has lower nesting depth.

- $\mathrm{Copy}_{B,\mathbf{fr}}$ is the left-most such operation with $j = i \circ \mathrm{Copy}_{B,\mathbf{fr}} \circ j'$ and B has no parent !-box in $\mathrm{Copy}_{B,\mathbf{fr}} \circ j'(X)$:

  Then $\mathbf{fr}(B)$ also has no parent !-box in $\mathrm{Copy}_{B,\mathbf{fr}} \circ j'(X)$. By theorem 3.3.22 we can write $i = \mathbf{rn} \circ i' \circ \mathrm{KE}_{\mathbf{fr}(B)}^n$ for a renaming $\mathbf{rn}$, so that:

$$\begin{aligned} j &= i \circ \mathrm{Copy}_{B,\mathbf{fr}} \circ j' \\ &= \mathbf{rn} \circ i' \circ \mathrm{KE}_{\mathbf{fr}(B)}^n \circ \mathrm{Copy}_{B,\mathbf{fr}} \circ j' \\ &= \mathbf{rn} \circ i' \circ \mathrm{Exp}_B^n \circ j' \end{aligned}$$

  where the final step is by repeated use of lemma A.2.3.

  Now we can restart the procedure on $i' \circ \mathrm{Exp}_{\mathbf{fr}(B)}^n \circ j'$, knowing that we have removed the left-most Copy operation.

- Flip_B is the left-most such operation with $j = i \circ \mathrm{Flip}_B \circ j'$ and B has a parent !-box in $\mathrm{Flip}_B \circ j'(X)$:

  Let A be the top level !-box containing B. By theorem 3.3.22 we can write $i = \mathbf{rn} \circ i' \circ \mathrm{KE}_A^n$ for a renaming $\mathbf{rn}$, so that:

$$\begin{aligned} j &= i \circ \mathrm{Flip}_B \circ j' \\ &= \mathbf{rn} \circ i' \circ \mathrm{KE}_A^n \circ \mathrm{Flip}_B \circ j' \\ &= \mathbf{rn} \circ i' \circ \mathrm{Flip}_{B_1} \circ \ldots \circ \mathrm{Flip}_{B_n} \circ \mathrm{KE}_A^n \circ j' \end{aligned}$$

where $B_1, \ldots B_n$ are the copies of B created by KE^n_A and the final step is by repeated use of lemma A.2.1.

Now we can restart the procedure on $i' \circ \mathrm{Flip}_{B_1} \circ \ldots \circ \mathrm{Flip}_{B_n} \circ \mathrm{KE}^n_A \circ j'$, knowing that the new left-most flipping operation Flip_{B_1} has lower nesting depth.

- Flip_B is the left-most such operation with $j = i \circ \mathrm{Flip}_B \circ j'$ and B has no parent !-box in $\mathrm{Flip}_B \circ j'(X)$:

  By theorem 3.3.22 we can write $i = \mathbf{rn} \circ i' \circ \mathrm{Kill}_B \circ \mathrm{Exp}_{B,\mathbf{fr}_n} \circ \ldots \circ \mathrm{Exp}_{B,\mathbf{fr}_1}$ for a renaming $\mathbf{rn}$, so that:

$$\begin{aligned} j &= i \circ \mathrm{Flip}_B \circ j' \\ &= \mathbf{rn} \circ i' \circ \mathrm{Kill}_B \circ \mathrm{Exp}_{B,\mathbf{fr}_n} \circ \ldots \circ \mathrm{Exp}_{B,\mathbf{fr}_1} \circ \mathrm{Flip}_B \circ j' \\ &= \mathbf{rn} \circ i' \circ \mathrm{Kill}_B \circ \mathrm{Exp}_{B,\mathbf{fr}'_1} \circ \ldots \circ \mathrm{Exp}_{B,\mathbf{fr}'_n} \circ j' \end{aligned}$$

  where the final step is by lemma A.2.1, introducing new fresh renamings $\mathbf{fr}'_i$, each of which agrees with $\mathbf{fr}_i$ on the contents of B.

  Now we can restart the procedure on $i' \circ \mathrm{Kill}_B \circ \mathrm{Exp}_{B,\mathbf{fr}_1} \circ \ldots \circ \mathrm{Exp}_{B,\mathbf{fr}_n} \circ j'$, knowing that we have removed the left-most Flip operation.

To see that this process terminates, note that each iteration either eliminates a Copy or Flip operation at depth 0, or replaces one at depth k with operations at depth $k-1$. □

4.2 The Rules of !L

!L is presented in terms of sequents which are of the form $\Gamma \vdash Y$, where Γ is a finite sequence $X_1, X_2, \ldots, X_n$. We interpret this as the !-formula Y following from the conjunction of the !-formulas $X_1, X_2, \ldots, X_n$. We always assume the !-formulas of a sequent have compatible !-box structures. The core rules of !L are taken from those of positive intuitionistic logic with (Cut):

$$\frac{}{X \vdash X}\ (\text{Ident}) \qquad \frac{\Gamma \vdash Y}{\Gamma, X \vdash Y}\ (\text{Weaken}) \qquad \frac{\Gamma, X, Y, \Delta \vdash Z}{\Gamma, Y, X, \Delta \vdash Z}\ (\text{Perm}) \qquad \frac{\Gamma, X, X \vdash Y}{\Gamma, X \vdash Y}\ (\text{Contr})$$

$$\frac{\Gamma \vdash X \qquad \Delta \vdash Y}{\Gamma, \Delta \vdash X \wedge Y}\ (\wedge I) \qquad \frac{\Gamma \vdash X \wedge Y}{\Gamma \vdash X}\ (\wedge E_1) \qquad \frac{\Gamma \vdash X \wedge Y}{\Gamma \vdash Y}\ (\wedge E_2)$$

$$\frac{\Gamma \vdash X \to Y}{\Gamma, X \vdash Y}\ (\to E) \qquad \frac{\Gamma, X \vdash Y}{\Gamma \vdash X \to Y}\ (\to I) \qquad \frac{\Gamma \vdash X \qquad \Delta, X \vdash Y}{\Gamma, \Delta \vdash Y}\ (\text{Cut})$$

The rules for introducing and eliminating $\forall$ are also analogous to the usual rules. We take a renaming $\mathbf{rn} : \mathcal{B} \to \mathcal{B}$ for !-boxes with $\text{support}(\mathbf{rn}) = {\downarrow}A$. Write $\mathbf{rn}(X)$ for the application of that renaming to a !-formula, then:

$$\frac{\Gamma \vdash \mathbf{rn}(X)}{\Gamma \vdash \forall A.X}\ (\forall I) \qquad \frac{\Gamma \vdash \forall A.X}{\Gamma \vdash \mathbf{rn}(X)}\ (\forall E)$$

where in the case of $(\forall I)$ we also require that $\mathbf{rn}({\downarrow}A)$ is disjoint from $\text{boxes}(\Gamma)$.

To these core logical rules, we add rules capturing the fact that $=$ is an equivalence relation and a congruence:

$$\frac{}{\Gamma \vdash G = G}\ (\text{Refl}) \qquad \frac{\Gamma \vdash G = H}{\Gamma \vdash H = G}\ (\text{Symm}) \qquad \frac{\Gamma \vdash G = H \qquad \Gamma \vdash H = K}{\Gamma \vdash G = K}\ (\text{Trans})$$

$$\frac{\Gamma \vdash G = H}{\Gamma \vdash [G]^A = [H]^A}\ (\text{Box}) \qquad \frac{\Gamma \vdash G = H}{\Gamma \vdash G{*}F = H{*}F}\ (\text{Concat})$$

The (Box) and (Concat) rules allow an equation to be applied to a sub-expression as we will prove in theorem 4.3.2.

The main utility of universal quantification is to control the application of !-box operations. In order to start instantiating a !-box (or one of its children), it must be under a universal quantifier:

$$\frac{\Gamma \vdash \forall A.X}{\Gamma \vdash \text{Flip}_B(X)}\ (\text{Flip}) \qquad \frac{\Gamma \vdash \forall A.X}{\Gamma \vdash \text{Kill}_B(X)}\ (\text{Kill}) \qquad \frac{\Gamma \vdash \forall A.X}{\Gamma \vdash \text{Drop}_B(X)}\ (\text{Drop})$$

$$\frac{\Gamma \vdash \forall A.X}{\Gamma \vdash \text{Copy}_B(X)}\ (\text{Copy}) \qquad \frac{\Gamma \vdash \forall A.X}{\Gamma \vdash \text{Exp}_B(X)}\ (\text{Exp})$$

where $B \preccurlyeq A$ in $\text{boxes}(X)$. These rule-schemes along with $(\forall I)$ play an analogous role to the substitution of a universally-quantified variable for an arbitrary term.

Remark 4.2.1. !-Box operations do not make up part of the underlying object language and are actually applied at the meta level. For example, in $\Gamma \vdash \text{Copy}_B(X)$, the expression $\text{Copy}_B(X)$ represents the !-formula resulting from applying Copy_B to the !-formula X. This is why we refer to (Copy) as a rule-scheme rather than a rule and similarly for the other !-box operation rule-schemes.

The final and possibly most significant rule-scheme of !L is *!-box induction*, which allows us to introduce new !-boxes. For a top-level !-box A, we have:

$$\frac{\Gamma \vdash \text{Kill}_A(X) \qquad \Delta, X \vdash \forall B_1. \cdots \forall B_n.\, \text{Exp}_A(X)}{\Gamma, \Delta \vdash X}\ (\text{Induct})$$

where A does not occur free in Γ or Δ and B_1 to B_n are the fresh names of children of A coming from its expansion.

Remark 4.2.2. The order of $B_1, \dots B_n$ above is unimportant as we can use $(\forall E)$ and $(\forall I)$ to reorder them.

4.3 Derived Rules

As described in definition 4.5.4, a sequent $Y_1, \dots, Y_n \vdash X$ means the !-formula X follows from the conjunction of the list of !-formulas $Y_1, \dots, Y_n$. By the rule (Perm) we can freely commute the elements of this list, meaning it acts as a multiset. Then by (Contr) we can remove repetition ensuring the assumptions act as the set $\{Y_1, \dots Y_n\}$.

As seen in section 3.1.1, diagrammatic theories are made up of a number of generators and a set Γ of diagrammatic equations. By the above we can then use Γ in a sequent by writing $\Gamma \vdash X$. If X is one of the equations of Γ then we would trivially expect $\Gamma \vdash X$ to be true as X is one of the assumptions. We now show how this is achieved in !L and give it the name (Assm) which we use throughout the rest of this thesis.

Theorem 4.3.1. *If $X \in \Gamma$ then*

$$\frac{}{\Gamma \vdash X}\ (Assm)$$

Proof. Let Γ be the list $Y_1, \dots Y_i, X, Y_{i+1}, \dots Y_n$ then by repeated application of (Weaken) and (Perm) we prove the claim:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X \vdash X}\ \text{(Ident)}}{X \vdash X}\ \text{(Weaken)}}{\vdots}}{X, Y_1, \dots, Y_n \vdash X}\ \text{(Weaken)}}{\vdots}\ \text{(Perm)}}{\Gamma \vdash X}\ \text{(Perm)}$$

□

We often have multiple ways to represent the exact same !-formula but wish to add an additional step to a proof tree to clarify rewriting. When we wish to do this we will write it as a rule $(\equiv)$. For example, the following diagrams demonstrate the use of $(\equiv)$ for equivalent expressions for a !-tensor and definitions of !-box operations respectively.

$$\frac{\Gamma \vdash GH = K}{\Gamma \vdash HG = K}\ (\equiv) \qquad\qquad \frac{\Gamma \vdash \mathrm{Drop}_B([G]^B) = H}{\Gamma \vdash G = H}\ (\equiv)$$

Suppose we have $\Gamma \vdash G = G'$ and G matches a !-tensor H, by which we mean $\exists B_1, \ldots, B_n \in \mathcal{B}$ and $\exists F \in \mathcal{T}_\Sigma$ such that $H = [[G]^{B_n} \ldots]^{B_1} * F$. We would expect to be able to substitute the copy of G in H with G'. We can now prove a rule stating this:

Theorem 4.3.2. *If G matches H as witnessed by $H = [[G]^{B_n} \ldots]^{B_1} * F$, then:*

$$\frac{\Gamma \vdash G = G'}{\Gamma \vdash H = H'} \textit{ (Subst)}$$

where $H' = [[G']^{B_n} \ldots]^{B_1} * F$, i.e. H with G' substituted for G.

Proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash G = G'}{\vdots}\,(\text{Box})}{\Gamma \vdash [[G]^{B_n} \ldots]^{B_1} = [[G']^{B_n} \ldots]^{B_1}}\,(\text{Box})}{\Gamma \vdash [[G]^{B_n} \ldots]^{B_1} * F = [[G']^{B_n} \ldots]^{B_1} * F}\,(\text{Concat})}{\Gamma \vdash H = H'}\,(\equiv)$$

□

It may seem unusual that the !-box operation rules (Op) lose quantification, since we may wish to apply more !-box operations. We now show how the current rules are enough to prove the quantified versions. We bring back not only quantification over $\downarrow B$ but also over any new components created by Op_B:

Theorem 4.3.3. *If $\mathrm{Op}_{B,\mathbf{fr}}$ is a !-box operation for $B \in \downarrow A$ with $\mathbf{fr}$ fresh for a !-formula X, then:*

$$\frac{\Gamma \vdash \forall A.X}{\Gamma \vdash \forall A_1. \cdots \forall A_n. \mathrm{Op}_{B,\mathbf{fr}}(X)} \textit{ ($\forall$ Op)}$$

where $\downarrow A_1, \ldots, \downarrow A_n$ are the connected components of $\mathrm{Op}_{B,\mathbf{fr}}(\downarrow A)$ (i.e. those created by $\mathrm{Op}_{B,\mathbf{fr}}$ in $\mathrm{Op}_{B,\mathbf{fr}}(X)$).

Proof. If we directly applied the rule ($\mathrm{Op}_{B,\mathbf{fr}}$), the result may contain !-boxes which already appear in Γ, preventing us from using ($\forall$ I) to bring back the quantification. Hence we need to be more precise about !-box names.

Let $\mathbf{rn} : \mathcal{B} \to \mathcal{B}$ rename the !-boxes of $\downarrow A$ to fresh names and act as identity elsewhere. We can extend $\mathbf{fr}$ to a new fresh renaming $\mathbf{fr}'$ which is also fresh on $\mathbf{rn}(\downarrow A)$. By alpha equivalence, we use $\mathbf{rn}$ to replace $\forall A.X$ with $\forall\, \mathbf{rn}(A).\, \mathbf{rn}(X)$ which removes the problem of names clashing.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \forall A.X}{\Gamma \vdash \forall\, \mathbf{rn}(A).\, \mathbf{rn}(X)}\,(\equiv)}{\Gamma \vdash \mathrm{Op}_{\mathbf{rn}(B),\mathbf{fr}'}\, \mathbf{rn}(X)}\,(\mathrm{Op})}{\vdots}\,(\forall\ \mathrm{I})}{\Gamma \vdash \forall A_1. \cdots \forall A_n. \mathrm{Op}_{B,\mathbf{fr}}\, X}\,(\forall\ \mathrm{I})$$

where the final steps are valid since $\mathbf{rn}$ and $\mathbf{fr}$ both chose fresh names not appearing in Γ. □

Theorem 4.3.4. *The rule (Exp) can be derived from the other rules in !L.*

Proof. We show that it can be derived from (Copy) and (Drop) using the identity $\mathrm{Exp}_{B,\mathbf{fr}} = \mathrm{Drop}_{\mathbf{fr}(B)} \circ \mathrm{Copy}_{B,\mathbf{fr}}$. We have two cases to check:
If $B \neq A$ then:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \forall A.X}{\Gamma \vdash \forall A.\,\mathrm{Copy}_{B,\mathbf{fr}}(X)}\,(\forall\,\mathrm{Copy})}{\Gamma \vdash \mathrm{Drop}_{\mathbf{fr}(B)} \circ \mathrm{Copy}_{B,\mathbf{fr}}(X)}\,(\mathrm{Drop})}{\Gamma \vdash \mathrm{Exp}_{B,\mathbf{fr}}(X)}\,(\equiv)$$

If $B = A$ then:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \forall A.X}{\Gamma \vdash \forall A.\forall\,\mathbf{fr}(A).\,\mathrm{Copy}_{B,\mathbf{fr}}(X)}\,(\forall\,\mathrm{Copy})}{\Gamma \vdash \forall\,\mathbf{fr}(A).\,\mathrm{Copy}_{B,\mathbf{fr}}(X)}\,(\forall\ \mathrm{E})}{\Gamma \vdash \mathrm{Drop}_{\mathbf{fr}(B)} \circ \mathrm{Copy}_{B,\mathbf{fr}}(X)}\,(\mathrm{Drop})}{\Gamma \vdash \mathrm{Exp}_{B,\mathbf{fr}}(X)}\,(\equiv)$$

□

Theorem 4.3.5. *For i a partial instantiation of the !-formula $Y_1 \to (\ldots(Y_n \to X))$ we get:*

$$\dfrac{Y_1, \ldots, Y_n \vdash X}{i(Y_1), \ldots, i(Y_n) \vdash i(X)}\,\textit{(Partial)}$$

Proof. Since i is made up of !-box operations we need only show the above for a general !-box operation $i = \mathrm{Op}_{B,\mathbf{fr}}$ then repeated application gives the general theorem. Let B be in the component $\downarrow A$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Y_1, \ldots, Y_n \vdash X}{\vdots}\,(\to \mathrm{I})}{\vdash Y_1 \to (\ldots(Y_n \to X))}\,(\to \mathrm{I})}{\vdash \forall A.(Y_1 \to (\ldots(Y_n \to X)))}\,(\forall\ \mathrm{I})}{\vdash \mathrm{Op}_{B,fr}(Y_1 \to (\ldots(Y_n \to X)))}\,(\mathrm{Op})}{\vdash \mathrm{Op}_{B,fr}(Y_1) \to (\ldots(\mathrm{Op}_{B,fr}(Y_n) \to \mathrm{Op}_{B,fr}(X)))}\,(\equiv)}{\vdots}\,(\to E)}{\mathrm{Op}_{B,\mathbf{fr}}(Y_1), \ldots, \mathrm{Op}_{B,\mathbf{fr}}(Y_n) \vdash \mathrm{Op}_{B,\mathbf{fr}}(X)}\,(\to E)$$

□

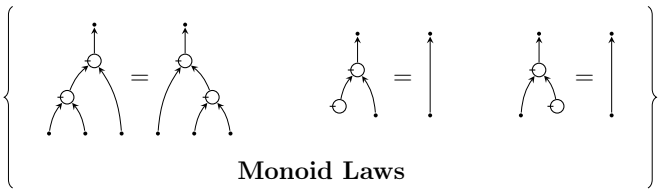
We now use these new rules to present a few examples of the use of induction when transitioning from a fixed arity theory to an arbitrary arity one.

4.4 Induction Examples

4.4.1 Monoid

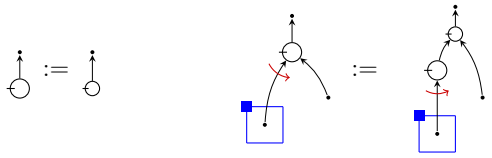
Recall the finite arity theory of a monoid from example 3.1.4. There are two generating morphisms: ( , ) and three diagrammatic axioms:

Γ_M =



Monoid Laws

As seen in section 3.4.1 we can recursively define an arbitrary arity multiplication operation from the finite arity generators:



From this we wish to prove that any two connected nodes can be combined to a single node. We will now prove this using !L, starting with a lemma.

Lemma 4.4.1.

$$\frac{}{\Gamma_M \vdash \forall A.\forall B.\ \ A \quad B = A \quad B} \ (4.4.1)$$

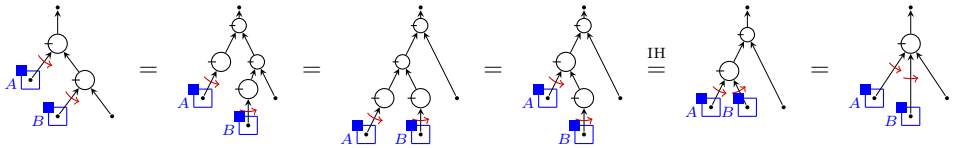
Proof. We present part of the proof tree:

$$\dfrac{\dfrac{\dfrac{}{\Gamma_M \vdash \ =\ } \ (\text{Assm})}{\Gamma_M \vdash \ A \quad =\ A} \ (\text{Concat})}{\Gamma_M \vdash \ A \quad =\ A} \ (\equiv) \qquad \dfrac{}{\Gamma_M ,\ A \quad B = A \quad B \ \vdash\ A \quad B = A \quad B} \ \textbf{(step)}$$

$$\frac{\Gamma_M \vdash \ A \quad B = A \quad B}{\Gamma_M \vdash \forall A.\forall B.\ \ A \quad B = A \quad B} \ (\forall I) \qquad (\text{Induct})$$

where we have proved the base case but left the step case (marked **step**) to be proved by a rewrite sequence:

step:



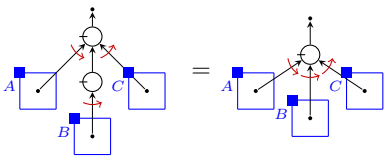
The equality labelled as IH follows from the inductive hypothesis by concatenation with the monoidal multiplication. □

Proof trees often become unfeasibly large and difficult to write down, so we present our proofs using rewriting sequences. Each equality in a rewrite sequence will be provable from the assumptions using !L. We will often drop the names of !-boxes and keep track of them using their position.

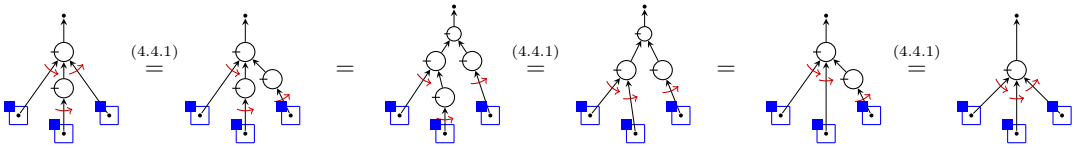
Using lemma 4.4.1 we can prove the more general case using rewriting.

Theorem 4.4.2.

$$\frac{}{\Gamma_M \vdash \forall A.\forall B.\forall C.\ \ \ =} \ (4.4.2)$$

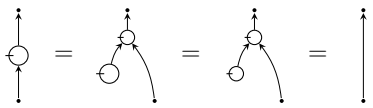


Proof.

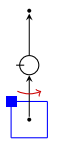


Where the first equality comes from $\mathrm{Exp}_A(4.4.1)$ by (Concat); the third comes from lemma 4.4.1 by (Concat); and the final equality is $\mathrm{Copy}_A(4.4.1)$. These are all valid since A is quantified over in the lemma. □

Remark 4.4.3. With the aim to simplify diagrams we can now always combine connected nodes until we have only one node. At this point we might be tempted to think we have as simple a diagram as possible. However, in the case of a single node with a single input we can simplify further:

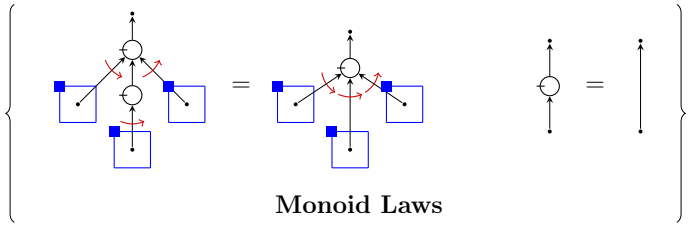


Theorem 4.4.4. *The finite arity theory of monoids Γ_M can be replaced by the arbitrary arity version $\Gamma_{M!}$ made up of a single arbitrary arity generator:*

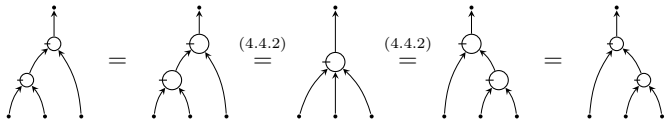


along with two !-tensor equations:

$\Gamma_{M!}$ =



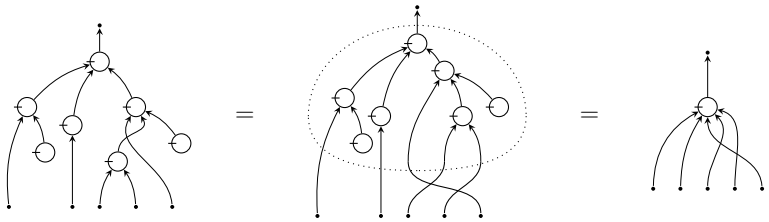
Proof. We have seen how the arbitrary arity node can be defined in terms of fixed arity generators and proved the equations in $\Gamma_{M!}$. Now we need only check that $\Gamma_{M!}$ covers all of the axioms in Γ_M. For example, the associativity axiom can be shown by:



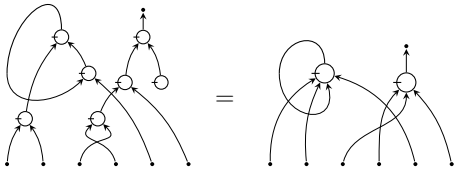
where the second and third equalities follow by applying $\mathrm{KE}^1_C \circ \mathrm{KE}^2_B \circ \mathrm{Kill}_A$ and $\mathrm{Kill}_C \circ \mathrm{KE}^2_B \circ \mathrm{KE}^1_A$ respectively to theorem 4.4.2. Similarly, the unit laws can be shown via rewriting:

□

As it is stated, theorem 4.4.2 allows us to combine any two connected nodes (so long as we keep track of edge ordering). This type of theorem allowing connected nodes to be merged together is often refered to as a spider theorem. By induction we can combine any finite connected network of nodes (again keeping track of edge order). The process of reducing an arbitrary diagram of connected nodes to a single node is as follows: first we identify a spanning tree in the diagram; second we use naturality to move all crossings (and possibly loops) out of the spanning tree; and finally, by repeated use of theorem 4.4.2, we can combine all nodes into one.

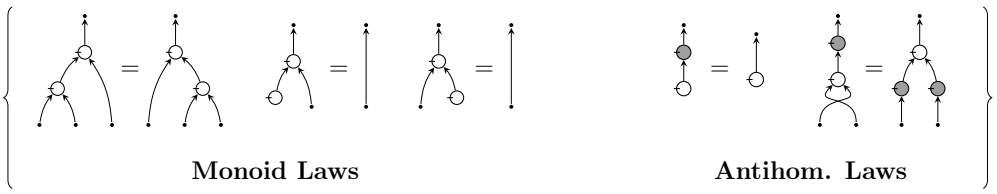


It is now clear to see the advantages of moving to the arbitrary arity theory. Complicated diagrams can be reduced to disconnected nodes:



4.4.2 Antihomomorphism

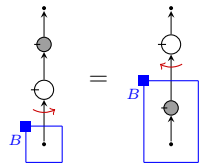
The fixed arity theory Γ_{AH} of an antihomomorphism ( ) for a monoid (,) is given by diagrammatic axioms:

$\Gamma_{AH} =$ 

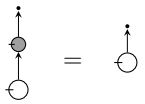
As in the previous section, we can replace the finite monoid part with its arbitrary arity counterpart. We can then prove the theorem suggested in (3.6):

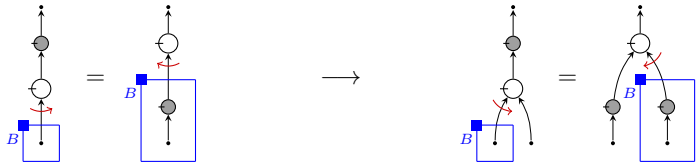
Theorem 4.4.5.

(4.4.5)

$\Gamma_{AH} \vdash \forall B.$ 

Proof. By induction on B the lemma breaks down into two cases:

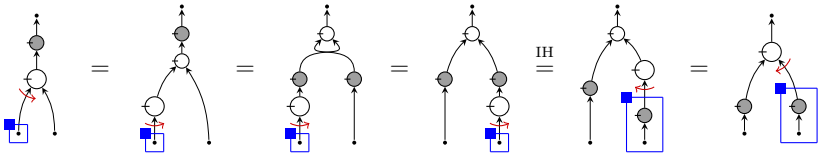

(**base**)



(**step**)

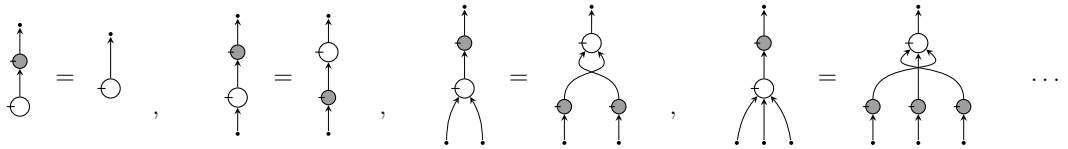
The base case trivially holds by (Assm); we show the step case by rewriting:

step:

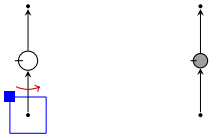


where the equality labelled as IH follows from the inductive hypothesis via the rule (Concat). □

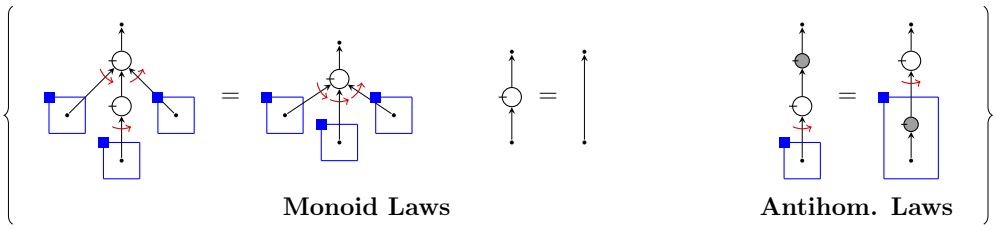
Hence by instantiating we have a proof for tensor equations such as:



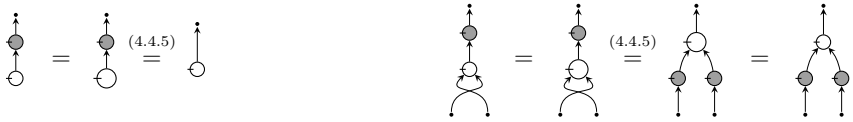
Theorem 4.4.6. *The finite arity theory of antihomomorphisms Γ_{AH} can be replaced with the arbitrary arity version $\Gamma_{AH!}$ made up of two generators:*



along with three !-tensor equations:

$\Gamma_{AH!} =$ 

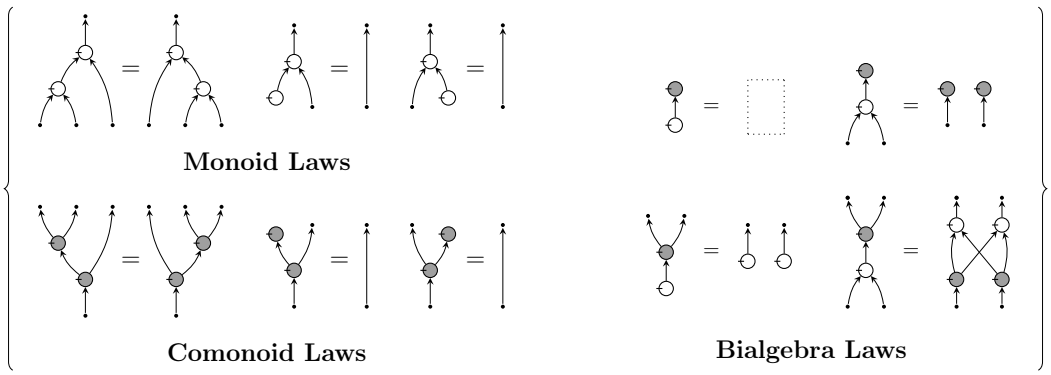
Proof. We have seen how these follow from the fixed arity theory. Now we need only check the axioms in Γ_{AH} follow from those in $\Gamma_{AH!}$. The monoid laws were shown in theorem 4.4.4 to follow from $\Gamma_{M!}$ which is a subset of $\Gamma_{AH!}$, so we need only check the antihomomorphism laws:



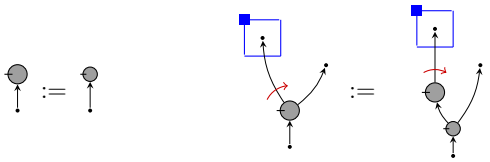
□

4.4.3 Bialgebra

The theory of a bialgebra consists of a monoid () and a comonoid () with the bialgebra laws governing their interaction:

$\Gamma_{BA} =$ 

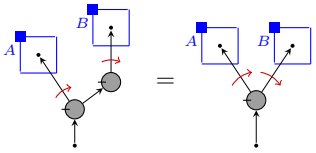
As in the previous two sections, we can replace the finite monoid part with its arbitrary arity counterpart. But in this case we can do the same with the comonoid. The recursive definition of a left associated tree of comultiplications is:



from which we prove the same lemma and theorem as in the monoid case:

Lemma 4.4.7.

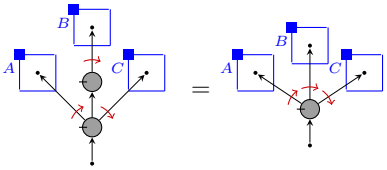
$$\Gamma_{BA} \vdash \quad \forall A.\forall B. \quad \text{(4.4.7)}$$



Proof. Proof by induction, similar to the proof of lemma 4.4.1 except with the diagrams flipped upside down and white nodes replaced with gray nodes. □

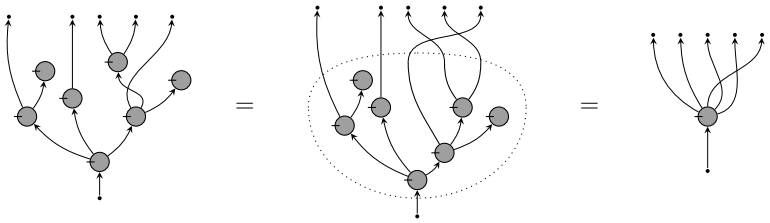
Theorem 4.4.8.

$$\Gamma_{BA} \vdash \forall A.\forall B.\forall C. \quad \text{(4.4.8)}$$

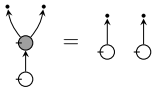


Proof. Proof by rewriting using lemma 4.4.7, similarly to the proof of theorem 4.4.2 except with the diagrams flipped upside down and white nodes replaced with gray nodes. □

Hence similarly to the monoid case, connected planar tree-like networks of comultiplications can be replaced by a single node:



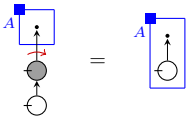
This already lets us replace the monoid and comonoid parts with arbitrary arity versions. We hope to do the same for the four bialgebra laws. Take the axiom stating that gray comultiplications copy white units:



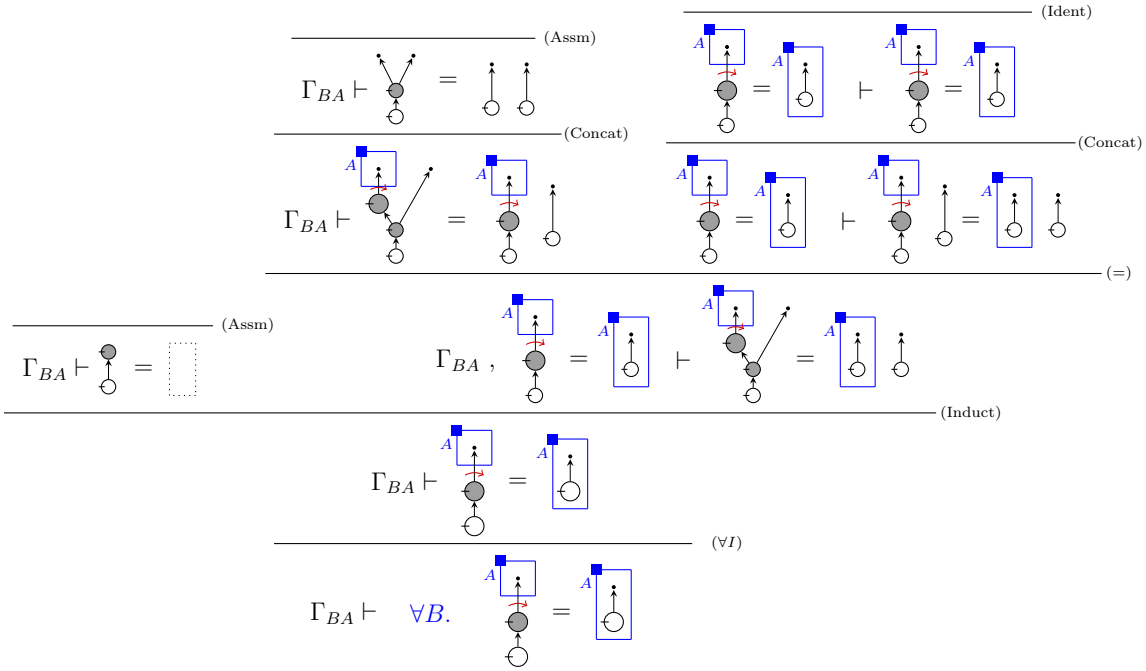
We hope to replace this with an arbitrary arity version, as seen in section 4.1.1.

Theorem 4.4.9.

$$\Gamma_{BA} \vdash \quad \forall A. \quad \text{(4.4.9)}$$



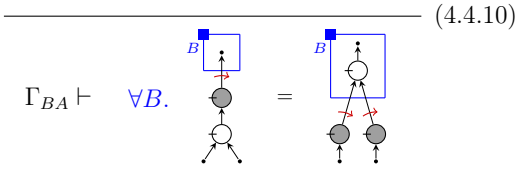
Proof. We use this example to demonstrate how a proof can be presented as a proof tree:



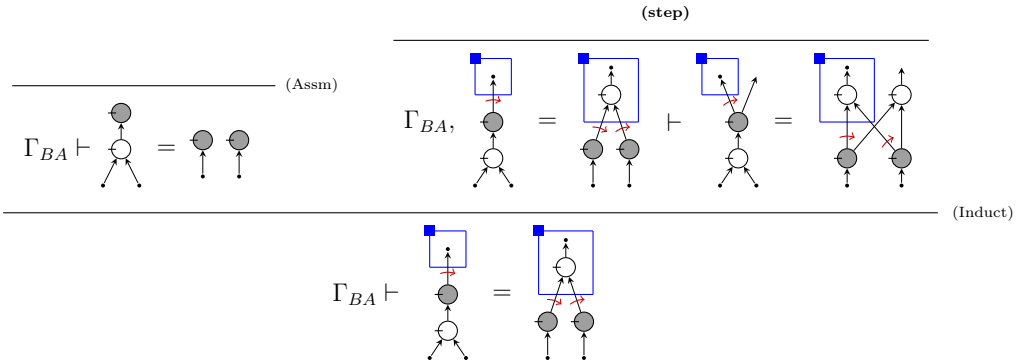
□

This theorem can be used to replace two of the bialgebra equations (the two concerning white units) but we can generalise even further to replace all four axioms by one !-tensor equation.

Lemma 4.4.10.

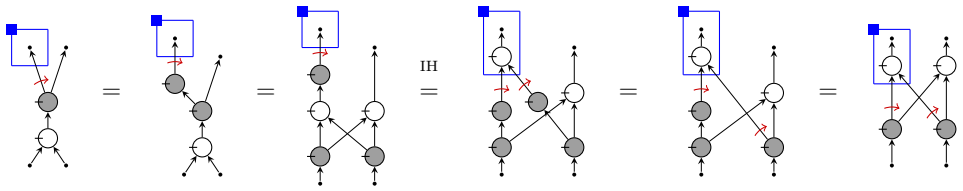


Proof. By applying induction on B the lemma breaks down into two cases:



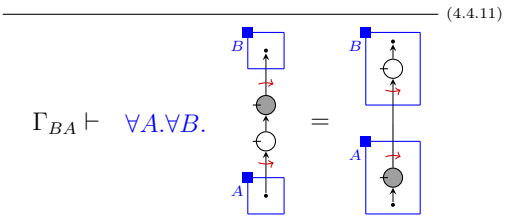
where we prove the step case (marked **step**) by rewriting:

step:

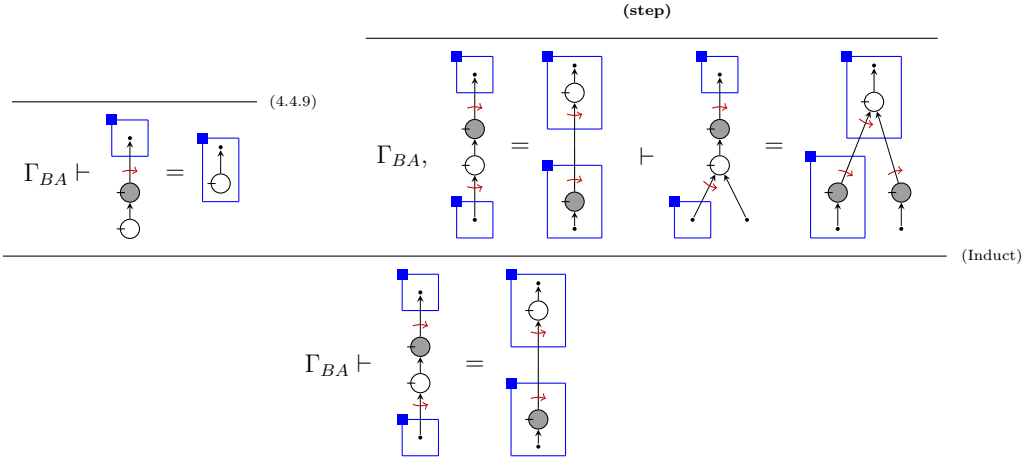


The equality labelled as IH follows from the inductive hypothesis via the rule (Concat). □

Theorem 4.4.11.

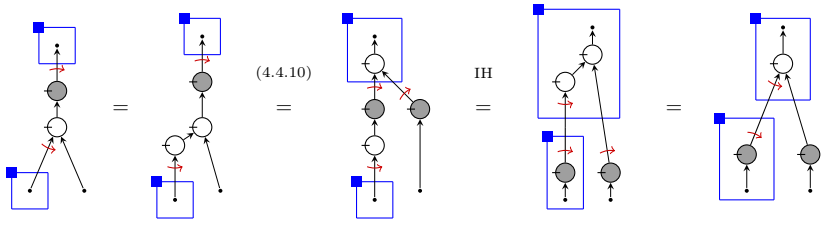


Proof. By applying induction on A the lemma breaks down into two cases:



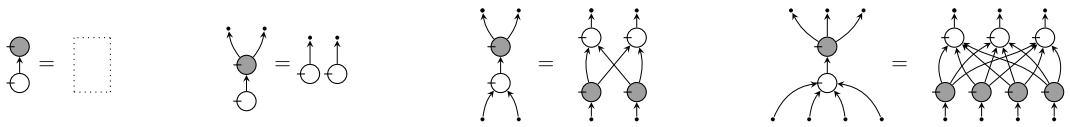
The base case is theorem 4.4.9, so we only need to prove the step case, which we do by rewriting:

step:



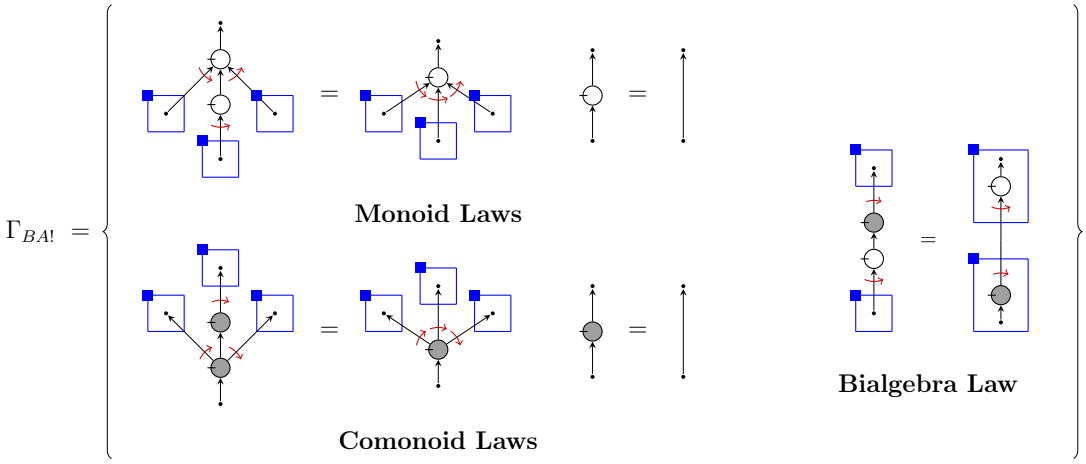
The second and third equalities follow from lemma 4.4.10 and the inductive hypothesis repectively via (Concat). □

Hence by instantiating we have a proof allowing any multiplication to pass through a comultiplication as in the following examples:



Theorem 4.4.12. *The fixed arity theory of bialgebras Γ_{AB} can be replaced with the arbitrary arity version made up of two generators:*

along with five rules:



Proof. We have seen how these follow from the fixed arity theory. Now we need only check the fixed arity bialgebra axioms follow from $\Gamma_{BA!}$. This is trivial as each one is simply an instance of theorem 4.4.11. Specifically we used the four instantiations $\mathrm{Kill}_B \circ \mathrm{Kill}_A$, $\mathrm{Kill}_B \circ \mathrm{KE}_A^2$, $\mathrm{KE}_B^2 \circ \mathrm{Kill}_A$ and $\mathrm{KE}_B^2 \circ \mathrm{KE}_A^2$. □

4.5 Semantics

Recall that a valuation for a signature Σ is a map $[\![-]\!] : \Sigma \to \mathcal{C}$ into a compact closed category and can uniquely extend to assign a morphism in $\mathcal{C}$ to each tensor. This allows us to assign a truth value to any concrete equation $G = H$ by saying it is true if $[\![G]\!]$ and $[\![H]\!]$ are the same morphism in $\mathcal{C}$. Hence we start by defining semantics on concrete equations:

$$[\![G = H]\!] := \begin{cases} T & \text{if } [\![G]\!] = [\![H]\!] \\ F & \text{otherwise} \end{cases} \tag{4.1}$$

We hope to build this up to !-formulas. We have defined the application of !-box operations and hence instantiations to !-formulas. Applying $i \in \mathrm{Inst}(X)$ to X should result in a truth value and hence we can think of $[\![X]\!]$ as a map from $\mathrm{Inst}(X)$ to truth values. Equivalently X can be defined as the set of instantiations i for which $i(X)$ holds. We adopt the second approach and hence atomic formulas have a natural interpretation.

Definition 4.5.1 (Atomic Formula Semantics)**.** For an atomic !-formula $G = H$ and a valuation $[\![-]\!] : \Sigma \to \mathcal{C}$, we let:

$$[\![G = H]\!] = \left\{ i \in \mathrm{Inst}(G = H) \;\middle|\; [\![i(G)]\!] = [\![i(H)]\!] \right\} \tag{4.2}$$

So in the concrete case, true equations get mapped to the set containing the identity instantiation id and false equations become the empty set. This can agree with our suggested interpretation of concrete !-formula equations if we define true and false by $T = \{\mathrm{id}\}$ and $F = \varnothing$.

Definition 4.5.2 (!-Formula Semantics)**.** The interpretation $[\![-]\!]$ of a !-formula is defined recursively by (atomic case above):

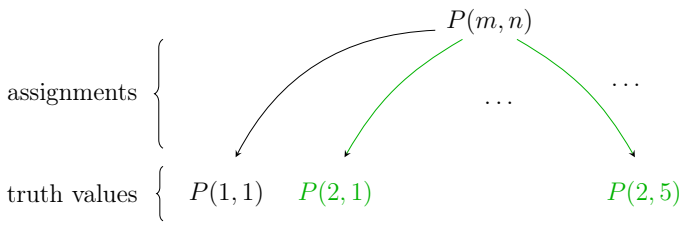
$$[\![X \wedge Y]\!] := \left\{ \; i \in \mathrm{Inst}(X \wedge Y) \;\middle|\; i|_X \in [\![X]\!] \wedge i|_Y \in [\![Y]\!] \; \right\}$$
$$[\![X \to Y]\!] := \left\{ \; i \in \mathrm{Inst}(X \to Y) \;\middle|\; i|_X \in [\![X]\!] \to i|_Y \in [\![Y]\!] \; \right\}$$
$$[\![\forall A.X]\!] := \left\{ \; i \in \mathrm{Inst}(\forall A.X) \;\middle|\; \forall j \in \mathrm{Inst}(i(X)) \quad j \circ \; i \in [\![X]\!] \; \right\}$$

When we write $i|_X$ we mean the restriction to only !-box operations which act on X (i.e. removing those acting as the identity). This ensures that $i|_X$ is an instantiation of X.

Remark 4.5.3. In the final case above we used the fact that !-box operations on different components commute. The concrete formula $i(\forall A.X)$ can equivalently be written as $\forall A.(i(X))$, demonstrating that $\mathrm{boxes}(i(X)) = \downarrow A$. Hence universal quantification over $\mathrm{Inst}(i(X))$ is quantifying over possible instantiations of $\downarrow A$ as we would expect.

This is analogous to a similar style of interpretation in first order predicate logic [19]. Here a predicate in some variables is treated as the set of assignments of those variables resulting in a correct statement. Our !-boxes take the role of variables and instantiating them is the equivalent of applying an assignment.

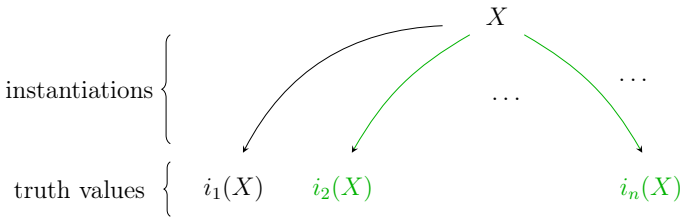
For example, working in $\mathbb{N}$, if we have the statement '$m.n$ is an even number' which we refer to as $P(m, n)$, we can take $[\![P(m, n)]\!]$ to be the set of assignments of natural numbers to m and n resulting in a true statement, i.e. those highlighted green in the following illustration.



$[\![\forall n P(m,n)]\!]$ is the set of assignments of natural numbers to m such that for all assignments of n we have correct statement $P(m,n)$ (in this case we get all even assignments of m).

In the statement $\forall m \forall n P(m,n)$ the quantification over m leaves us with no variables to assign values to, so the only assignment is the trivial identity assignment. As suggested by our interpretation, $[\![\forall m \forall n P(m,n)]\!]$ should equal $\{id\}$ if and only if $id(\forall m \forall n P(m,n))$ is true; in other words if and only if $P(m,n)$ is true for all m and n (in this case it is false and we are left with the empty set).

Replacing variables with !-box components and assignments with instantiations we see how this compares with our semantics. For X containing two non-nested !-boxes A and B, $[\![X]\!]$ is the set of instantiations leading to correct concrete equations which can be drawn similarly to above:



$[\![\forall B.X]\!]$ is the set of instantiations i of $\forall B.X$ such that $i(\forall B.X)$ is a correct formula. These must instantiate the only remaining !-box A and hence are instantiations i of A such that for all instantiations j of $i(X)$ we have correct formula $j \circ\ i(X)$.

Quantifying over A leaves us with no free !-boxes to instantiate so the only instantiation is the trivial identity function. As suggested by our interpretation, $[\![\forall A.\forall B.X]\!]$ should equal $\{id\}$ if and only if $id(\forall A.\forall B.X)$ is true; in other words if and only if all instantiations of X are true.

Sequents on the other hand will always be interpreted as truth values. Namely whether or not the !-formulas on the left of the turnstyle imply that on the right. To define this we push all sequents right past the turnstyle and quantify over all remaining free !-boxes.

Definition 4.5.4 (Sequent Semantics)**.**

$$[\![X_1, \ldots, X_n \vdash Y]\!] := [\![\forall A_1 \ldots \forall A_m.((X_1 \wedge \ldots \wedge X_n) \to Y)]\!]$$

where $\downarrow A_1, \ldots \downarrow A_m$ are the connected components of $(X_1 \wedge \ldots \wedge X_n) \to Y$.

Theorem 4.5.5. *The definition above is independent of the chosen order of* $A_1, \ldots, A_m$.

Proof. We verify $[\![\forall A.\forall B.X]\!] = [\![\forall B.\forall A.X]\!]$, from which the result follows. By expanding the definition of semantics twice:

$$\begin{aligned}[\![\forall A.\forall B.X]\!] &= \{i \in \mathrm{Inst}(\forall A.\forall B.X) \mid \forall j \in \mathrm{Inst}(i(\forall B.X))\ j \circ\ i \in [\![\forall B.X]\!]\}. \\ &= \{i \in \mathrm{Inst}(\forall A.\forall B.X) \mid \forall j \in \mathrm{Inst}(\forall B.(i(X))\ \forall k \in \mathrm{Inst}(j(i(X)))\ k \circ\ j \circ\ i \in [\![X]\!]\}.\end{aligned}$$

Suppose i is in this set. We wish to show $i \in [\![\forall B.\forall A.X]\!]$. Hence given $k' \in \mathrm{Inst}(i(\forall A.X))$ and $j' \in \mathrm{Inst}(k'(i(X)))$ we need to verify $j' \circ k' \circ\ i \in [\![X]\!]$.
We have $j' \circ k' \in \mathrm{Inst}(i(X))$. By lemma A.1.1 we can commute all operations in j' through k' only changing fresh renamings. Hence we have $k \circ\ j = j' \circ k' \in \mathrm{Inst}(i(X))$ where $k \in \mathrm{Inst}(j(i(X)))$ and $j \in \mathrm{Inst}(\forall B.i(X))$.
$i \in [\![\forall A.\forall B.X]\!]$ implies $j' \circ k' \circ\ i = k \circ\ j \circ\ i \in [\![X]\!]$ as required.
Hence $[\![\forall A.\forall B.X]\!] \subseteq [\![\forall B.\forall A.X]\!]$ and by symmetry the reverse is true. This proves order of quantification is unimportant. □

4.6 Soundness

Now that we have a semantic interpretation of sequents, we hope to check that !L is sound with respect to it. i.e. If $\Gamma \vdash X$ is derivable in !L, then $[\![\Gamma \vdash X]\!]$ is true for any valuation $[\![-]\!]$ into a compact closed category.

To prove this, it suffices to show that each rule in our logic respects $[\![-]\!]$, i.e. truth of the conclusion (with respect to $[\![-]\!]$) should follow from truth of the premises. Before we prove this we present some useful notation and a lemma to make the proof easier to follow.

If $\mathrm{boxes}(X)$ is a component of $\mathrm{boxes}(Y)$ and $i \in \mathrm{Inst}(Y)$ we will write $i \vDash X$ as shorthand for $i|_X \in [\![X]\!]$. Using this notation, the interpretation can be rewritten as follows:

$$\begin{aligned}
i \vDash G = H &\iff [\![i(G)]\!] = [\![i(H)]\!] && i \in \mathrm{Inst}(G = H) \\
i \vDash X \wedge Y &\iff i \vDash X \wedge i \vDash Y && i \in \mathrm{Inst}(X \wedge Y) \\
i \vDash X \to Y &\iff i \vDash X \to i \vDash Y && i \in \mathrm{Inst}(X \to Y) \\
i \vDash \forall A.X &\iff \forall j \in \mathrm{Inst}(i(X)) \quad j \circ\ i \vDash X && i \in \mathrm{Inst}(\forall A.X))
\end{aligned}$$

Lemma 4.6.1. *Given a !-formula X with* $\mathrm{boxes}(X)$ *comprised of connected components $\downarrow B_1, \ldots \downarrow B_n$:*

$$i \vDash X \quad \forall i \in \mathit{Inst}(X) \quad \iff \quad [\![\forall B_1 \ldots \forall B_n.X]\!] = \{\mathrm{id}\} = T$$

Proof. Suppose we have the left hand side and take $j \in \mathrm{Inst}(\forall B_n.X)$. If k is a concrete instantiation of $j(X)$ then $k \circ\ j$ is a concrete instantiation of X and so $k \circ\ j \vDash X$. Hence we have shown for any $j \in \mathrm{Inst}(\forall B_n.X)$ that $j \vDash \forall B_n.X$.

Conversely, suppose we are given that $\forall j \in \mathrm{Inst}(\forall B_n.X)\ j \vDash \forall B_n.X$ and we have $i \in \mathrm{Inst}(X)$. Then we can pull the operations on $\downarrow B_n$ to the left in i to rewrite it as $i = k \circ\ j$ where $j \in \mathrm{Inst}(\forall B_n.X)$ and $k \in \mathrm{Inst}(j(X))$. By the assumption we now have $j \vDash \forall B_n.X$ and hence $k \in \mathrm{Inst}(j(X))$ implies $k \circ\ j \vDash X$, i.e. $i \vDash X$.
We have shown the equivalence of $i \vDash X \quad \forall i \in \mathrm{Inst}(X)$ and $i \vDash \forall B_n.X \quad \forall i \in \mathrm{Inst}(\forall B_n.X)$. By repeated application we can prove the lemma:

$$\begin{aligned} i \vDash X \quad \forall i \in \mathrm{Inst}(X) &\Leftrightarrow i \vDash \forall B_n.X \quad \forall i \in \mathrm{Inst}(\forall B_n.X) \\ &\vdots \\ &\Leftrightarrow i \vDash \forall B_1 \ldots \forall B_n.X \quad \forall i \in \mathrm{Inst}(\forall B_1 \ldots \forall B_n.X) = \{\mathrm{id}\} \\ &\Leftrightarrow \mathrm{id} \vDash \forall B_1 \ldots \forall B_n.X \\ &\Leftrightarrow [\![\forall B_1 \ldots \forall B_n.X]\!] = \{\mathrm{id}\} = T \end{aligned}$$

□

We can now prove soundness as described at the start of the section

Remark 4.6.2. Since edge names do not inherently carry any meaning, for a concrete formula X and a renaming $\mathbf{rn}$, we have that $\mathbf{rn}(X)$ and X are interchangeable. From this we deduce that $[\![\mathbf{rn}(X)]\!] = [\![X]\!]$ for any !-formula X and also that if $i \Vdash X$ then $\mathbf{rn} \circ i \Vdash X$.

Theorem 4.6.3 (Soundness)**.** *If $\Gamma \vdash X$ is derivable in !L, then $[\![\Gamma \vdash X]\!]$ is true for any valuation $[\![-]\!]$ into a compact closed category $\mathcal{C}$.*

Proof. We check for an arbitrary valuation $[\![-]\!] : \Sigma \to \mathcal{C}$, that the rules of !L respect truth of $[\![-]\!]$. Throughout we will write K for the conjunction of the !-formulas in Γ and K' for the conjunction of those in Δ. By lemma 4.6.1, to check that $[\![\Gamma \vdash X]\!]$ is true, it suffices to check that, for all $i \in \mathrm{Inst}(K \to X)$, $i \vDash K \to X$. We now check each rule individually:

- (Ident) Given $i \in \mathrm{Inst}(X \to X)$, we need to show $i \vDash X \to X$. This is equivalent to $i \vDash X \to i \vDash X$, which is trivially true.
- (Weaken) Given $i \in \mathrm{Inst}((K \wedge X) \to Y)$, the premise of (Weaken) states that $i \vDash K \to Y$. Now if $i \vDash K \wedge X$, then $i \vDash K$, so $i \vDash Y$ by the premise. Thus $i \vDash (K \wedge X) \to Y$.
- The rules (Perm) and (Contr) follow from associativity, commutativity, and idempotence of $\wedge$.
- $(\wedge I)$ Given $i \in \mathrm{Inst}((K \wedge K') \to (X \wedge Y))$, the premises of $(\wedge I)$ state $i \vDash K \to X$ and $i \vDash K' \to Y$. Now if $i \vDash K \wedge K'$, we have $i \vDash K$ and hence $i \vDash X$. We also have $i \vDash K'$ and hence $i \vDash Y$. Thus $i \vDash X \wedge Y$.

- ($\wedge E1$) Given $i \in \text{Inst}(K \to X)$. We can apply i to $K \to (X \wedge Y)$ but there may be free !-boxes left so let $j \in \text{Inst}(i(K \to (X \wedge Y)))$. Then our premise states that $j \circ\ i \vDash K \to (X \wedge Y)$. Now, if $i \vDash K$ then $j \circ\ i \vDash K$ so the premise implies $j \circ\ i \vDash X \wedge Y$ from which we conclude $j \circ\ i \vDash X$ and so $i \vDash X$.

- ($\wedge E2$) is similar to ($\wedge E1$).

- ($\to E$) Given $i \in \text{Inst}((K \wedge X) \to Y)$, the premise states $i \vDash K \to (X \to Y)$. Now if $i \vDash K \wedge X$ then we can firstly conclude $i \vDash K$, so by the premise $i \vDash X \to Y$, and secondly we can conclude $i \vDash X$ and hence deduce $i \vDash Y$.

- ($\to I$) is the same as ($\to E$) in reverse.

- (Cut) Given $i \in \text{Inst}(K \wedge K' \to Y)$. We can apply i to $K \to X$ or to $(K' \wedge X) \to Y$ but in each case we may be left with free !-boxes (from those components in X which are not in K, K' or Y). Let j instantiate these extra !-boxes so that both $j \circ\ i(K \to X)$ and $j \circ\ i((K' \wedge X) \to Y)$ are concrete !-formulas. Then the premises tell us that $j \circ\ i \vDash K \to X$ and $j \circ\ i \vDash (K' \wedge X) \to Y$. Now if $i \vDash K \wedge K'$ then $j \circ\ i \vDash K \wedge K'$, which firstly gives us $j \circ\ i \vDash K$ which by our first premise implies $j \circ\ i \vDash X$. Secondly, it gives us $j \circ\ i \vDash K'$ which combined with the previous statement shows us $j \circ\ i \vDash K' \wedge X$ and hence, by the second premise, $j \circ\ i \vDash Y$ i.e. $i \vDash Y$.

- ($\forall I$) Given $i \in \text{Inst}(K \to \forall A.X)$, we need to show that if $i \vDash K$ then given an arbitrary $j \in \text{Inst}(i(X))$ we have $j \circ\ i \vDash X$. First note that without loss of generality any !-box names on operations in i are disjoint from $\mathbf{rn}(\downarrow A)$. This is possible because $\mathbf{rn}(\downarrow A)$ must already be disjoint from boxes(Γ) (by side-condition) and it must be disjoint from boxes($\forall A.X$) = boxes(X)$\setminus \downarrow A$ by injectivity of $\mathbf{rn}$. The only other !-box names in i are those introduced during instantiation, which can be freely chosen.
  We now have that $\mathbf{rn}(j) \in \text{Inst}(i(\mathbf{rn}(X)))$ (since $\mathbf{rn}$ has no affect on operations in i) and hence $\mathbf{rn}(j) \circ\ i \in \text{Inst}(K \to \mathbf{rn}(X))$ which by the premise implies $\mathbf{rn}(j) \circ\ i \vDash K \to \mathbf{rn}(X)$. Now since $\mathbf{rn}$ is identity everywhere except $\downarrow A$, we can rewrite this as $\mathbf{rn}(j \circ\ i) \vDash \mathbf{rn}(K \to X)$ which is equivalent to $j \circ\ i \vDash K \to X$. Now since $i \vDash K$ we can conclude $j \circ\ i \vDash X$.

- ($\forall E$) Given $i \in \text{Inst}(K \to \mathbf{rn}(X))$ and that $i \vDash K$, the premise tells us $i \vDash \forall A.X$. If we remove some operations from i to get $i' = i|_{\forall A.X}$ we still have $i' \vDash \forall A.X$. By definition this means for any j in $\text{Inst}(i'(X))$ we have $j \circ\ i' \vDash X$. Renaming both sides gives $\mathbf{rn}(j \circ\ i') \vDash \mathbf{rn}(X)$ but since $\mathbf{rn}$ is identity except on $\downarrow A$ this is equivalently written as $\mathbf{rn}(j) \circ\ i' \vDash \mathbf{rn}(X)$. Since j is free to be chosen, we can take it so that $(\mathbf{rn}(j) \circ\ i')|_{\mathbf{rn}(X)} = i|_{\mathbf{rn}(X)}$. Hence $i \vDash \mathbf{rn}(X)$.

- The rules (Refl), (Symm), and (Trans) reduce to the equivalent properties of equality of morphisms in the category $\mathcal{C}$.

- (Box) We are given $i \in \text{Inst}(K \to [G]^A = [H]^A)$, with $i \vDash K$. For the !-formulas to be compatible we must have $\downarrow A$ disjoint from boxes(K). Hence A is a top-level !-box in $K \to [G]^A = [H]^A$ so that we can write $i = \mathbf{rn} \circ i' \circ \text{KE}_A^n$ for a renaming $\mathbf{rn}$, with $i' \in \text{Inst}(\text{KE}_A^n(K \to [G]^A = [H]^A))$. Writing fr_1 to fr_n for the successive Exp_A operations we can prove $i \vDash [G]^A = [H]^A$:

$$\begin{aligned}
[\![i([G]^A)]\!] &= [\![i'(\mathbf{fr}_1(G) \dots \mathbf{fr}_n(G))]\!] \\
&= [\![i'(\mathbf{fr}_1(G)) \dots i'(\mathbf{fr}_n(G))]\!] \\
&= [\![i'(\mathbf{fr}_1(H)) \dots i'(\mathbf{fr}_n(H))]\!] \\
&= [\![i'(\mathbf{fr}_1(H) \dots \mathbf{fr}_n(H))]\!] \\
&= [\![i([H]^A)]\!]
\end{aligned}$$

  where we have dropped the renaming $\mathbf{rn}$ using remark 4.6.2 the third equality follows from the premise and the fact that if $[\![G]\!] = [\![G']\!]$ and $[\![H]\!] = [\![H']\!]$ for concrete tensors G and H then $[\![GH]\!] = [\![G'H']\!]$.

- (Concat) Given $i \in \text{Inst}(K \to G{*}F = H{*}F)$, such that $i \vDash K$ the premise tells us $[\![i(G)]\!] = [\![i(H)]\!]$ so we can check $i \vDash G{*}F = H{*}F$:

$$\begin{aligned}
[\![i(G{*}F)]\!] &= [\![i(G){*}i(F)]\!] \\
&= [\![i(G)i(F)]\!] \\
&= [\![i(H)i(F)]\!] \\
&= [\![i(H){*}i(F)]\!] \\
&= [\![i(H{*}F)]\!]
\end{aligned}$$

- (Kill) We are given $i \in \text{Inst}(K \to \text{Kill}_B(X))$ such that $i \vDash K$. We look at $i \circ \text{Kill}_B$. We can pull the operations on boxes$(K \to \forall A.X)$ to the right to get, for $\mathbf{rn}$ a renaming, $i \circ \text{Kill}_B = \mathbf{rn} \circ j \circ i'$ where $i' = i|_{K \to \forall A.X}$ and $j \in \text{Inst}(i'(K \to X))$. Since $i' \vDash K$ the premise tells us $i' \vDash \forall A.X$. Hence $j \circ i' \vDash X$ which by remark 4.6.2 implies $\mathbf{rn} \circ j \circ i' \vDash X$ i.e. $i \circ \text{Kill}_B \vDash X$ which is equivalent to $i \vDash \text{Kill}_B(X)$ as required.

- The proofs for (Flip), (Drop), and (Copy) are similar to that of (Kill) except we use theorem 4.1.12 to rewrite $i \circ \text{Flip}_B$, $i \circ \text{Drop}_B$, and $i \circ \text{Copy}_B$ respectively as only Exp and Kill operations to make sure they form valid instantiations. Note, theorem 4.1.12 introduces a renaming function at the left so

- (Exp) We can prove this similarly to (Kill) or note from theorem 4.3.4 that it is not necessary.

- (Induct) We wish to show from the premises of (Induct) that for any instantiation $i \in \text{Inst}((K \wedge K') \to X)$ that if $i \vDash K$ and $i \vDash K'$ then $i \vDash X$. Since (Induct) is done on a top-level !-box

A, by theorem 3.3.22, we can rewrite i as $\mathbf{rn} \circ j \circ \mathrm{KE}^n_A$, where $\mathbf{rn}$ is a renaming and j does not contain A. Renaming is unimportant so we proceed for arbitrary expressions of the form $j \circ \mathrm{KE}^n_A$ by induction on the natural number n.

Base case: For $j \circ \mathrm{Kill}_A$ if $j \circ \mathrm{Kill}_A \vDash K$, then since A is not free in K, $j \vDash K$. So, by the first premise, $j \vDash \mathrm{Kill}_A(X)$. Thus $j \circ \mathrm{Kill}_A \vDash X$, as required.

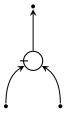
Step case: For fixed natural number n, the inductive hypothesis tells us any expression of the form $i := j \circ \mathrm{KE}^n_A$ we have $i \vDash (K \wedge K') \to X$. Now take an arbitrary expression $i' = j' \circ \mathrm{KE}^n_A \circ \mathrm{Exp}_{A,\mathbf{fr}}$ (which has one additional expansion of A) and suppose $i' \vDash K \wedge K'$. j' may contain operations on newly created !-boxes from $\mathrm{Exp}_{A,\mathbf{fr}}$, but since these new !-boxes do not overlap with the others their operations commute. Hence we can pull the operations from j' which act on $\mathbf{fr}(\downarrow A)$ to the right and call them j'' so $i' = j \circ \mathrm{KE}^n_A \circ j'' \circ \mathrm{Exp}_{A,\mathbf{fr}}$ for some j. Since A is not free in $K \wedge K'$ we can reduce $i' \vDash K \wedge K'$ down to $j \circ \mathrm{KE}^n_A \vDash K \wedge K'$. Now the inductive hypothesis tells us $j \circ \mathrm{KE}^n_A \vDash X$ which by the second (Induct) premise implies $j \circ \mathrm{KE}^n_A \vDash \forall B_1 \ldots \forall B_n . \mathrm{Exp}_{A,\mathbf{fr}}(X)$. This quantification over $B_1, \ldots, B_n$ allows any instantiation of these !-boxes, such as j'', and so trivially implies $j \circ \mathrm{KE}^n_A \circ j'' \vDash \mathrm{Exp}_{A,\mathbf{fr}}(X)$ and hence $j \circ \mathrm{KE}^n_A \circ j'' \circ \mathrm{Exp}_{A,\mathbf{fr}} \vDash X$ i.e. $i' \vDash X$ as required.

□

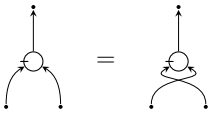
Chapter 5

Node Types

We have seen how !-logic allows us to formalise proof in the language of !-tensors which are designed for working with noncommutative theories. We would like to allow commutative theories to use the same natural deduction system. Suppose we have a commutative morphism with two inputs and a single output. We can represent this as a !-tensor node:



however, this has fixed the order of edges. To enforce commutativity of a generator we would need to add new axioms. For example, in this case we would add an axiom allowing the inputs to be reversed:



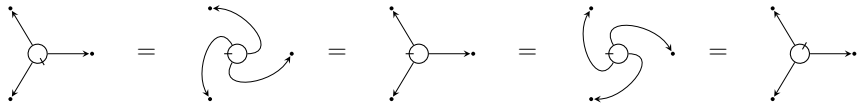
We would prefer not to have to deal with such axioms every time we wish to commute edges, instead having them built into the notation. In this chapter we introduce a new node type to take advantage of commutativity and another for nodes with the property of symmetry, which ensures they do not need the tick on their node. This allows not only commutative theories but theories involving a variety of generators: some noncommutative, some symmetric and some commutative.

5.1 Symmetric Morphisms

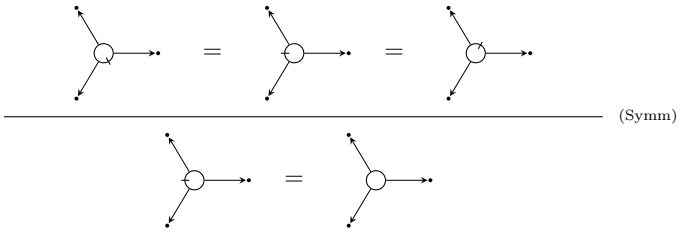
Symmetric morphisms are characterised by the property that cyclic permutations of edges have no effect on the morphism. For such morphisms we wish to remove the tick, resulting in a new notation as a circular node. Before looking at the general case let us attempt such a transition for a fixed arity generator.

Take the example of a morphism with no inputs but three outputs: $\phi : I \to X \otimes X \otimes X$. We refer to such a morphism with only outputs as a state. Symmetry prescribes that the three outputs can undergo cyclic permutations without affecting the state. We can visualise this in point notation and !-tensor notation, representing ϕ as a white node:

Another way to see this in !-tensor notation is that the tick position can pass through edges. In the following diagram the second and third equalities are by symmetry:



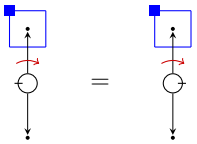
This suggests that tick position no longer has any relevance, so we may remove the tick entirely. We will refer to nodes without ticks, i.e. circles, as *symm-nodes*. Hence we would hope to implement the following rule stating that if the tick position on ϕ is irrelevant, then it can always be replaced by a symm-node:



It is clear to see how the new notation is inherently symmetric. To represent the fact that ϕ is a symmetric morphism in the !-tensor language, we could tag it with a superscript $\phi \to \phi^s$. We can then add a new equivalence to !-tensor expressions allowing cyclic permutation of edges. In this case it is sufficient to allow $\phi^s_{\hat{a}\hat{b}\hat{c}} \equiv \phi^s_{\hat{c}\hat{a}\hat{b}}$ for any edgenames a, b, c. Having symmetry built in to !-tensor equivalence in this way allows us to quotient out the unnecessary information in edge orders of ϕ.

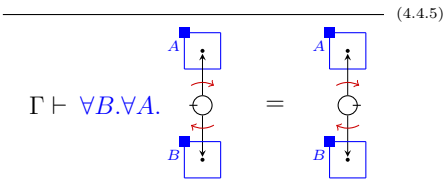
While this rule for introducing symm-nodes could be added to !L, the same is not true for the more general arbitrary arity generator. Suppose we have extended ϕ to represent a family of generator states, each with no inputs and indexed by the number of outputs, $\phi_i : I \to X^{\otimes i}$. i.e. the family represented by:

where we also have a rule allowing the tick to 'pass through' a single edge:

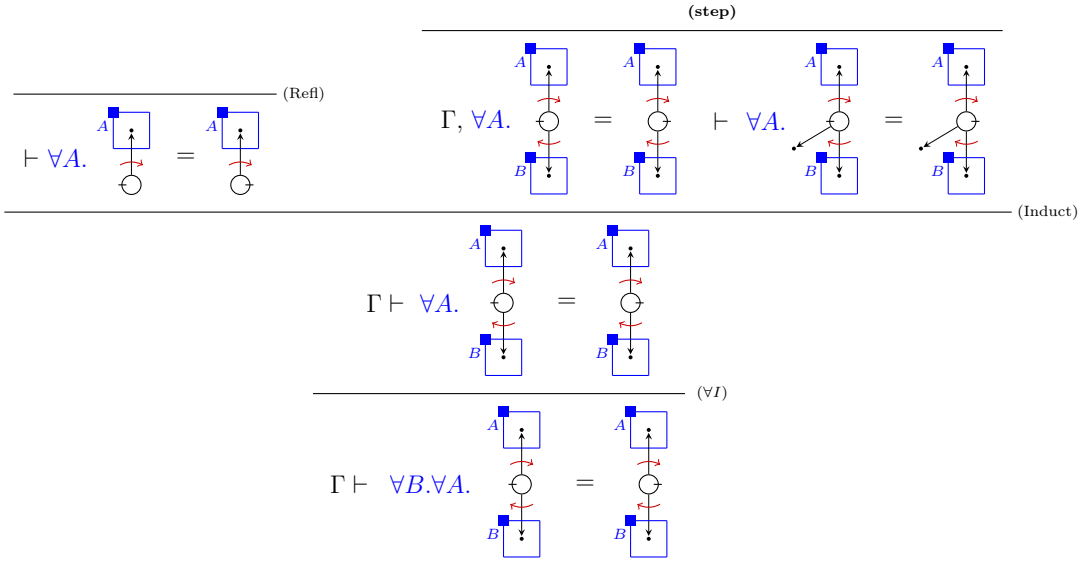


By induction we can prove a generalisation allowing the tick to pass through an arbitrary number of edges. Writing Γ for the set of axioms (in this case only symmetry) we present the proof graphically:

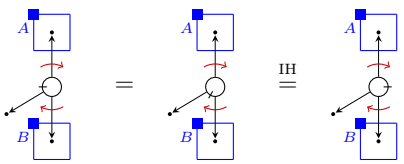
Theorem 5.1.1.



Proof. We prove this theorem, of the form $\forall B.\forall A.X$ by applying induction on B to the !-formula $\forall A.X$:

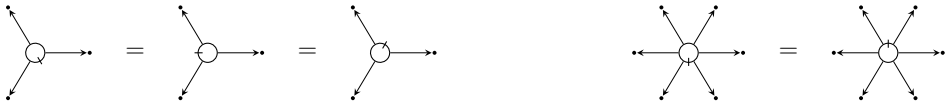


where we prove the step case by rewriting:



The first equality is from the symmetry axiom allowing the tick to pass through a single edge. The second equality comes from the inductive hypothesis, by applying $\mathrm{Flip}_A \circ \mathrm{Exp}_A \circ \mathrm{Flip}_A$, which is valid since A is quantified over. □

The concrete instances demonstrate how a tick can be placed at any location on any concrete instance of the state ϕ:



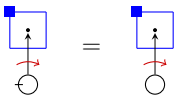
There are also many partial instantiations which demonstrate how a tick can move around a variable arity node, such as:

We may hope that any such equation (simply repositioning the tick around a state of type ϕ) is an instance of theorem 5.1.1. However, this is not the case. For example, the following two equations, which we expect to hold, are not partial instances.

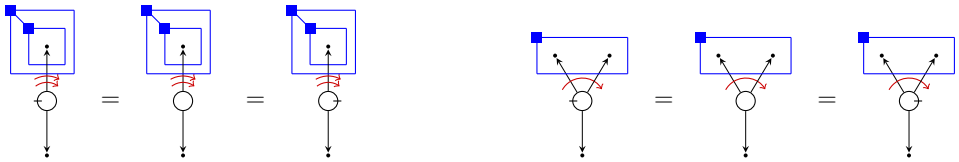
It is clear that we cannot retrieve these as instances since no !-box operations will increase the depth of !-box nesting or the number of edges in the single !-box. While these cases do not directly follow from theorem 5.1.1 by (Partial), each can be shown via !-box induction. Unfortunately it is clear that there is no single rule which has as instances all possible tick repositionings.

Similarly, we would like to be able to write a single rule specifying that any node of type ϕ can be replaced by a symm-node (i.e. a circular node, since tick position is irrelevant). The concrete instances would be rules such as:

Each of these fixed arity cases is a concrete instance of:



Yet, as previously, this !-tensor equation does not cover all cases of switching to symm-nodes as partial instances. For example, neither of the following is an instance (though they can each be deduced by induction):



Again, no single rule could possibly cover all cases, so we cannot implement the conversion from arbitrary nodes to symm-nodes in !L. We can, however, implement symmetric nodes in our theory by tagging morphisms which are known to be symmetric. We can then add new !-tensor equivalences to represent the fact that tick position is no longer important.

We write ϕ^s_e rather than ϕ_e to represent that the morphism ϕ is symmetric. Then, to enforce the ability to arbitrarily apply cyclic permutations to the edges, we add the !-tensor equivalence:

$$\phi^s_{ef} \equiv \phi^s_{fe} \tag{5.1}$$

Theorem 5.1.2. *The equivalence $\equiv$ is still preserved by !-box operations.*

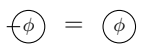
Proof. Given $\phi^s_{ef} \equiv \phi^s_{fe}$ and the !-box operation Op_B (possibly employing a fresh renaming) we check:

$$\mathrm{Op}_B(\phi^s_{ef}) \equiv \phi^s_{\mathrm{Op}_B(e)\,\mathrm{Op}_B(f)} \equiv \phi^s_{\mathrm{Op}_B(f)\,\mathrm{Op}_B(e)} \equiv \mathrm{Op}_B(\phi^s_{fe})$$

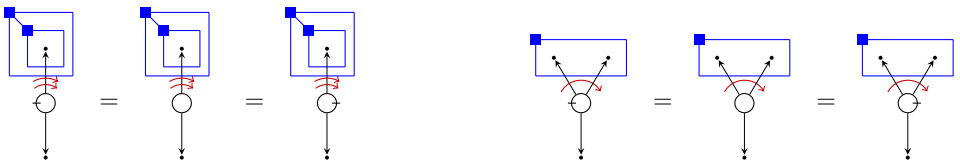
□

To introduce symm-nodes to a theory we would like to use a metarule describing when it is valid to switch a morphism to symm-nodes. In the case of the (family of) states ϕ above, from theorem 5.1.1, the metarule should deduce that any term ϕ_e can be replaced by ϕ^s_e, allowing all arbitrary nodes of type ϕ to be replaced by symm-nodes.

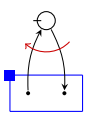
We will write this rule as:



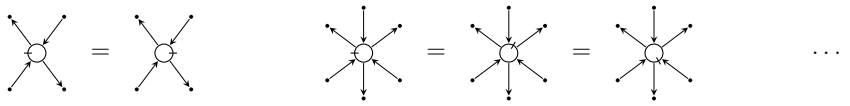
by which we do not mean $\phi_\epsilon = \phi^s_\epsilon$ as it appears but that $\phi_e = \phi^s_e$ for any valid edgeterm e. This equation is clearly not a !-formula, but it makes the earlier difficult cases trivial:



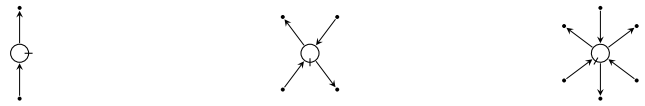
The state ϕ has only output edges, all of the same type X, so any cyclic permutation $\phi^s_{ef} \to \phi^s_{fe}$ results in a valid state. More generally, a generating morphism might not allow every arrangement of edges but still be independent of valid choice of tick position. For example, the family of morphisms generated by:



may have equalities:



so that any valid choices for tick position around the node are equal. Note that there are tick positions which are not valid such as:

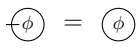


Therefore we need to adapt our idea of symmetry accordingly, to allow for cases where not all cyclic permutations of edges are valid. We define a morphism to be *symmetric* in the theory Γ if any two concrete instances which differ only by a cyclic permutation can be shown to be equal from Γ.

Definition 5.1.3 (Symmetric Morphism)**.** We say ϕ is a *symmetric morphism* in theory Γ if for any concrete edgeterms e, f we have $\Gamma \vdash \phi_{ef} = \phi_{fe}$ (if it is a valid !-formula).

It is clear to see how this relates to equivalence of symmetric nodes as described in (5.1).

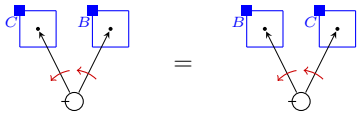
Definition 5.1.4 (Symmetry Metarule)**.** If ϕ is symmetric in Γ then the metarule (Symm) deduces that:



This agrees with our suggestion for the state ϕ where any concrete case of $\Gamma \vdash \phi_{ef} = \phi_{fe}$ follows as an instance of theorem 5.1.1. Hence the meta rule allows us to draw nodes as circles without ticks. We could equivalently use a sequence of instances of the symmetry axiom for ϕ to show each concrete instance.

Example 5.1.5. Take the arbitrary arity theory of a monoid $\Gamma_{M!}$ from section 4.4. Nodes have a single output after the tick followed by an arbitrary number of inputs. The only cyclic permutation of edges taking the single output back to its original position is the trivial permutation and hence we vacuously have that the morphism is symmetric and nodes with ticks can be replaced by circular nodes. This is not surprising since the single output edge uniquely determines where the tick should be positioned, so it should not be necessary in the graphical notation.

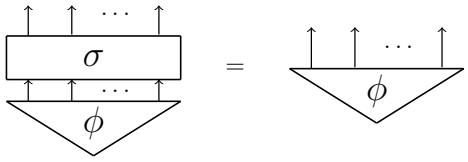
The most interesting cases of symmetry are those where the generating morphism is of the form $X = \phi_{[e\rangle^B}$ or $X = \phi_{\langle e]^B}$ (such as the state ϕ above and the frobenius algebra case we will present in section 6.2). In these cases the condition of symmetry reduces to correctness of all concrete instances of $\mathrm{Kill}_A \circ \mathrm{Copy}_{A,\mathbf{fr}_C} \circ \mathrm{Copy}_{A,\mathbf{fr}_B}(X) = \mathrm{Kill}_A \circ \mathrm{Copy}_{A,\mathbf{fr}_B} \circ \mathrm{Copy}_{A,\mathbf{fr}_C}(X)$. For example, this states that symmetry of ϕ follows from:



where $\mathbf{fr}_B$ takes A (along with its contents) to B and $\mathbf{fr}_C$ takes A to C. From which the metarule (Symm) deduces that $\phi_e = \phi_e^s$ for any valid edgeterm e.

5.2 Commutative Morphisms

Commutativity of a morphism states that wire order is unimportant. For example, taking the family of states $\phi_i : I \to X^{\otimes i}$ as in the previous section, commutativity of the family ϕ means that any well-formed equality of the form $\sigma \circ \phi_i = \phi_i$, where σ is constructed from only symmetries and identities, holds:



We can represent commutativity of ϕ in !-tensor notation by allowing any rearrangements of the edges around a node of type ϕ:

As in the symmetric case, we will tag *commutative morphisms* and then allow them in the theory by replacing arbitrary nodes with what we call *comm-nodes*. A comm-node is drawn without a tick since edge order is clearly not important; however, we also draw the circle in bold to represent that edges can commute arbitrarily. Then we define some new !-tensor equivalences to implement commutativity of edges around these nodes. In this case the equivalence should allow any edges to be rearranged, which we implement on the level of edgeterms. Hence two edgeterms are equivalent up to commutativity, written $\equiv_c$, if they are equivalent under any of the previous edgeterm equivalences along with the additional equivalence:

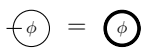
$$ef \equiv_c fe$$

From this it follows that $efgh \equiv_c egfh$ so by transitivity of $\equiv_c$ any rearrangement of edges is equivalent. We use this to write the definition of commutativity of a morphism:

Definition 5.2.1 (Commutative Morphism)**.** We say the morphism ϕ is *commutative* in theory Γ if $\Gamma \vdash \phi_e = \phi_f$ for any concrete edgeterms e and f valid for ϕ such that $e \equiv_c f$.

Note that commutativity of a morphism talks only about the concrete instances similarly to the definition of a symmetric morphism. We tag commutative morphisms with a superscript c, so our metarule is:

Definition 5.2.2 (Commutativity Metarule)**.** If ϕ is commutative in Γ then the metarule (Comm) deduces that:



i.e. $\phi_e = \phi_e^c$ for any valid edgeterm e.

We now have not only lost the tick but also drawn the node in bold to indicate that edges commute. Since comm-nodes are independent of edge order, we then introduce a new !-tensor equivalence:

$$e \equiv_c f \quad \Longrightarrow \quad \phi_e^c \equiv \phi_f^c \tag{5.2}$$

This is enough to arbitrarily rearrange edges around any concrete instance of ϕ, but it transpires this is not the only equivalence needed for comm-nodes. Recall that the notation $[e\rangle^B$ dictates that expansions of the !-box B should add new copies of e clockwise around the node. Since edge order is redundant, we expect direction to be redundant too. We can see this from the following equivalences:

$$\mathrm{Copy}_{B,\mathbf{fr}}([e\rangle^B) \equiv_c [e\rangle^B[\mathbf{fr}(e)\rangle^{\mathbf{fr}(B)} \equiv_c [\mathbf{fr}(e)\rangle^{\mathbf{fr}(B)}[e\rangle^B$$
$$\mathrm{Copy}_{B,\mathbf{fr}}(\langle e]^B) \equiv_c \langle \mathbf{fr}(e)]^{\mathbf{fr}(B)}\langle e]^B \equiv_c \langle e]^B\langle \mathbf{fr}(e)]^{\mathbf{fr}(B)}$$

so clockwise expanding groups can be expanded anti-clockwise and vice versa. We remove this redundancy from !-tensors by adding a new equivalence saying clockwise expanding groups and anti-clockwise expanding groups are interchangeable in commutative edgeterms.

$$[e\rangle^B \equiv_c \langle e]^B \tag{5.3}$$

To avoid this redundancy in the !-tensor expression notation we can write $\langle e\rangle^B := [e\rangle^B \equiv_c \langle e]^B$.

Similarly, the ability to group edges functions only to enforce an ordering on their expansions, so we also expect these to be redundant. The concrete instances of $\langle ef\rangle^B$ are equivalent to the concrete instances of $\langle e\rangle^B\langle f\rangle^B$ up to commutativity. Hence we add another equivalence:

$$\langle ef\rangle^B \equiv_c \langle e\rangle^B\langle f\rangle^B \tag{5.4}$$

This allows all edge groups to be split apart into single edges. Taking this ability to ungroup edges along with removing their directions allows us to entirely remove the arc notation around

comm-nodes. Hence if all morphisms are commutative, we no longer have ticks or expansion arcs and so our graphical language has been reduced to that of !-graphs.

Note that we are still allowing each of these commutative edgeterm equivalences on commutative morphisms by $e \equiv_c f \;\Rightarrow\; \phi^c_e \equiv \phi^c_f$.

Theorem 5.2.3. *The equivalence $\equiv$ is still preserved by !-box operations.*

Proof. We need to check that if $e \equiv_c f$, then $\mathrm{Op}_A(\phi^c_e) \equiv \mathrm{Op}_A(\phi^c_f)$ or equivalently $\phi^c_{\mathrm{Op}_A(e)} \equiv \phi^c_{\mathrm{Op}_A(f)}$ which we show by checking $\mathrm{Op}_A(e) \equiv_c \mathrm{Op}_A(f)$. We must check each of our three enforced equivalences for $\equiv_c$:

- The first commutativity equivalence is $ef \equiv_c fe$ so we check:

$$\mathrm{Op}_A(ef) \equiv_c \mathrm{Op}_A(e)\,\mathrm{Op}_A(f) \equiv_c \mathrm{Op}_A(f)\,\mathrm{Op}_A(e) \equiv_c \mathrm{Op}_A(fe)$$

- The second equivalence is $[e\rangle^B \equiv_c \langle e]^B$. This is simple to check if $A \neq B$:

$$\mathrm{Op}_A([e\rangle^B) \equiv_c [\mathrm{Op}_A(e)\rangle^B \equiv_c \langle \mathrm{Op}_A(e)]^B \equiv_c \mathrm{Op}_A(\langle e]^B)$$

  else if $A = B$ it needs to be shown separately for each !-box operation:

  - $\mathrm{Flip}_B([e\rangle^B) \equiv_c \langle e]^B \equiv_c [e\rangle^B \equiv_c \mathrm{Flip}_B(\langle e]^B)$
  - $\mathrm{Kill}_B([e\rangle^B) \equiv_c \epsilon \equiv_c \mathrm{Kill}_B(\langle e]^B)$
  - $\mathrm{Drop}_B([e\rangle^B) \equiv_c e \equiv_c \mathrm{Drop}_B(\langle e]^B)$
  - $\mathrm{Copy}_{B,\mathbf{fr}}([e\rangle^B) \equiv_c [e\rangle^B[\mathbf{fr}(e)\rangle^{\mathbf{fr}(B)} \equiv_c \langle e]^B\langle \mathbf{fr}(e)]^{\mathbf{fr}(B)} \equiv_c \langle e]^{\mathbf{fr}(B)}\langle e]^B \equiv_c \mathrm{Copy}_{B,\mathbf{fr}}(\langle e]^B)$
  - The $\mathrm{Exp}_{B,\mathbf{fr}}$ case follows by applying $\mathrm{Drop}_{\mathbf{fr}(B)}$ to the Copy case.

- The final equivalence is $\langle ef\rangle^B \equiv_c \langle e\rangle^B\langle f\rangle^B$. This is simple to check if $A \neq B$:

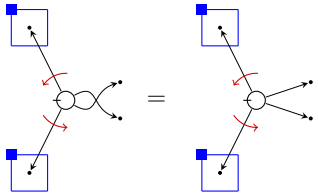
$$\mathrm{Op}_A(\langle ef\rangle^B) \equiv_c \langle \mathrm{Op}_A(e)\,\mathrm{Op}_A(f)\rangle^B \equiv_c \langle \mathrm{Op}_A(e)\rangle^B\langle \mathrm{Op}_A(f)\rangle^B \equiv_c \mathrm{Op}_A(\langle e\rangle^B\langle f\rangle^B)$$

  else if $A = B$ it needs to be shown separately for each !-box operation:

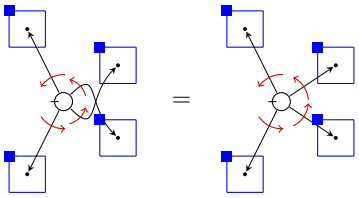
  - $\mathrm{Flip}_B(\langle ef\rangle^B) \equiv_c \langle ef\rangle^B \equiv_c \langle e\rangle^B\langle f\rangle^B \equiv_c \mathrm{Flip}_B(\langle e\rangle^B\langle f\rangle^B)$
  - $\mathrm{Kill}_B(\langle ef\rangle^B) \equiv_c \epsilon \equiv_c \mathrm{Kill}_B(\langle e\rangle^B\langle f\rangle^B)$
  - $\mathrm{Drop}_B(\langle ef\rangle^B) \equiv_c ef \equiv_c \mathrm{Drop}_B(\langle e\rangle^B\langle f\rangle^B)$
  - $\mathrm{Copy}_{B,\mathbf{fr}}(\langle ef\rangle^B) \equiv_c \langle ef\rangle^B\langle \mathbf{fr}(e)\,\mathbf{fr}(f)\rangle^{\mathbf{fr}(B)}$
    $\equiv_c \langle e\rangle^B\langle \mathbf{fr}(e)\rangle^{\mathbf{fr}(B)}\langle f\rangle^B\langle \mathbf{fr}(f)\rangle^{\mathbf{fr}(B)}$
    $\equiv_c \mathrm{Copy}_{B,\mathbf{fr}}(\langle e\rangle^B\langle f\rangle^B)$
  - The $\mathrm{Exp}_{B,\mathbf{fr}}$ case follows by applying $\mathrm{Drop}_{\mathbf{fr}(B)}$ to the Copy case.

□

To see an example we return to the state ϕ as in the previous section but add an additional axiom enforcing that any two neighbouring edges can commute:

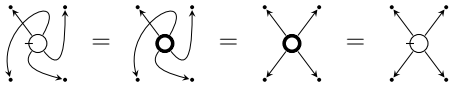


This allows any two adjacent edges to be commuted, hence sequences of instances of this allow arbitrary rearrangements of edges around any fixed arity instance of the state ϕ. In fact, using induction, we can prove the following rule covering all concrete instance of the equivalence $\phi^c_{efgh} \equiv \phi^c_{egfh}$ as suggested above:

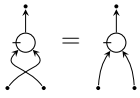


from which our metarule deduces:

Hence many equations become trivial by switching to comm-nodes (remembering that bold nodes do not track edge order):

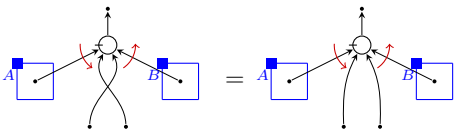


For another example let us take the theory of commutative monoids. Here we add the new axiom:

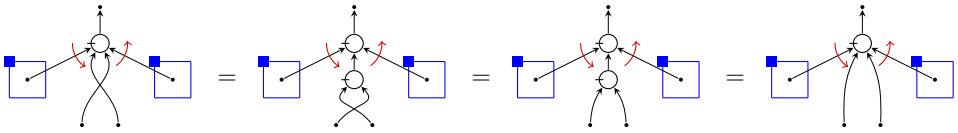


to the fixed arity theory Γ_M. We can replace the fixed arity theory with the arbitrary arity version as in section 4.4 and replace the commutativity rule with its arbitrary arity counterpart:

Theorem 5.2.4.

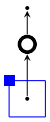


Proof. Which we prove by rewriting:

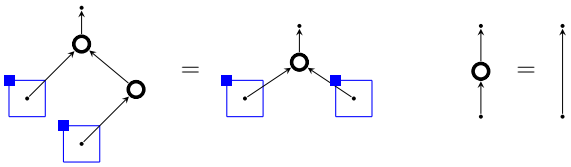


where the first and third equalities use theorem 4.4.2 which states that two connected nodes can be combined. □

Now any rearrangement of the edges of an instance of the arbitrary arity commutative monoid node must take the single output to itself. The remaining inputs can be rearranged in any way. By a sequence of instances of theorem 5.2.4 we can show that any reorderings are equal. Hence our metarule tells us that the arbitrary arity commutative monoid morphisms can be drawn as comm-nodes (bold circles) and hence all arcs can be removed. This makes theorem 5.2.4 trivial and hence the theory of commutative monoids can be described by a single (commutative) generator:



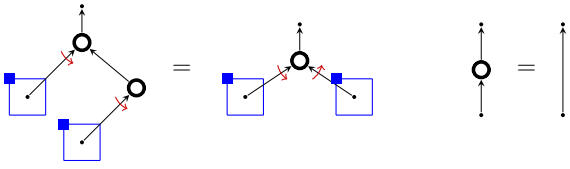
with rules:



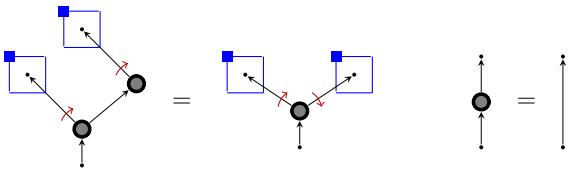
Commutative Monoid Laws

5.3 Commutative Bialgebra

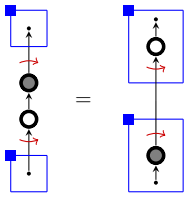
A (co)commutative bialgebra can be described by the theory of a commutative monoid along with a similar theory of a cocommutative comonoid and the interactions of the two as described in section 4.4:



Commutative Monoid Laws

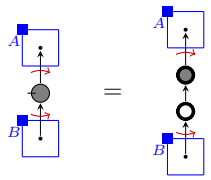


Commutative Comonoid Laws



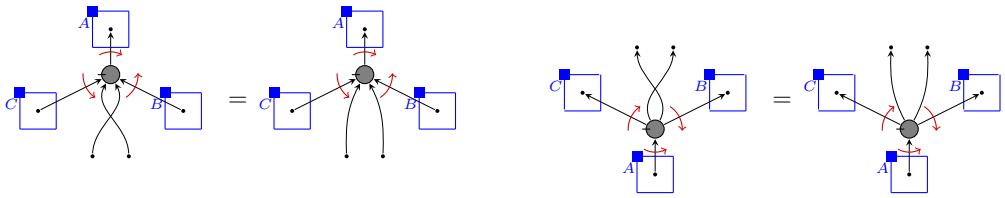
Bialgebra Law

(Co)commutative bialgebras appear in a variety of fields and have been shown to admit a normal form where all comultiplications are pushed to the bottom and all multiplications are pushed to the top [33, 16]. Here we demonstrate how we can replace the two generators with a single generator by defining the new morphism:

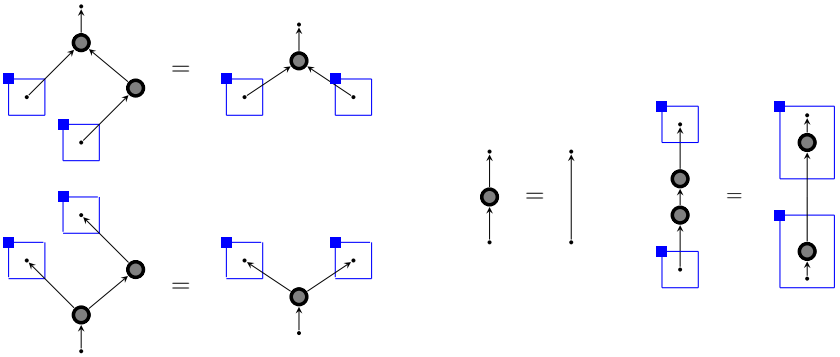


Note that this must be defined as an arbitrary node (with tick) even though it is defined in terms of commutative morphisms. The concrete instances of this new node are of the form:

Any rearranging of the edges must rearrange the inputs and separately rearrange the outputs. Hence commutativity follows from sequences of instances of the following two equations, which follow trivially from (co)commutativity of the (co)monoid:



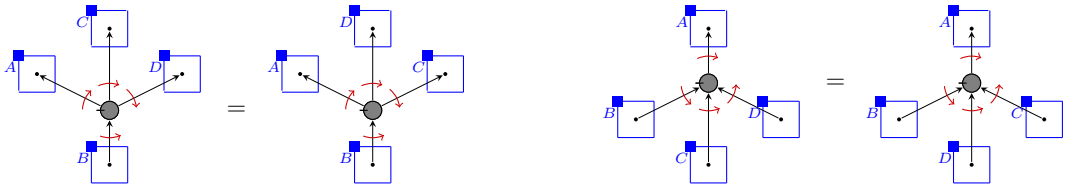
So by the metarule (Comm) we can replace the arbitrary nodes with commnodes and drop arcs, meaning the theory can be described more succinctly by the rules:



(Co)Commutative Bialgebra Laws

In each of our examples of the use of the metarule (Comm), each !-box in the generator contained a different edge type (e.g. input vs output). This has ensured that edges can only be rearranged with other edges from the same !-box. If this is the case, then commutativity of the morphism

can be deduced from proofs for each !-box A in the generator that $\mathrm{Copy}_{A,\mathbf{fr}'} \circ \mathrm{Copy}_{A,\mathbf{fr}''}(X) = \mathrm{Copy}_{A,\mathbf{fr}''} \circ \mathrm{Copy}_{A,\mathbf{fr}'}(X)$. For example, in the bialgebra case this means showing each of:



In the next chapter we demonstrate another example of switching to symm-nodes and another of switching to comm-nodes, in each case for frobenius algebras.

Chapter 6

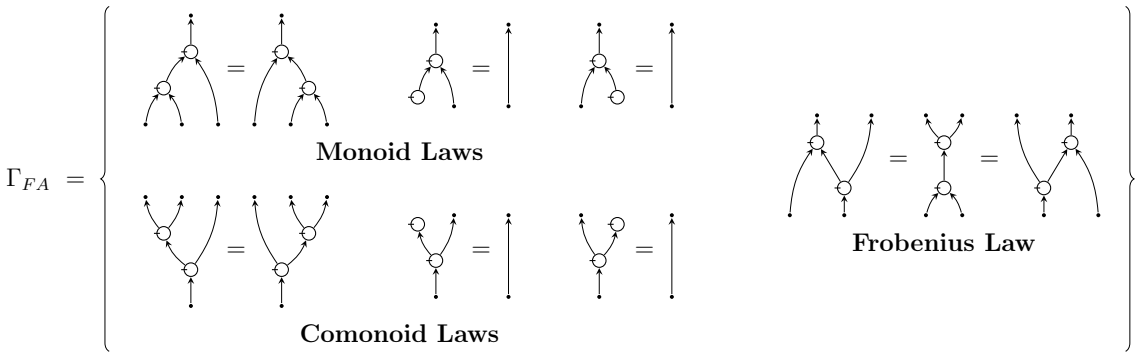
Working Example: Frobenius Algebras

In this chapter we work through the example of Frobenius algebras including symmetric, commutative, and special variations. Frobenius algebras appear in a large variety of different areas particularly in quantum information theory. Commutative Frobenius algebras (CFAs) have been shown to play an important role in topological quantum field theories [37] and to describe classical data in [7]. They lend themselves to a graphical notation and are the building blocks of the Z/X-calculus [8] and GHZ/W-calculus [9], which are key to understanding various principles in categorical Quantum mechanics.

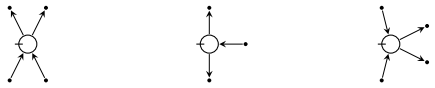
Starting with the standard fixed arity definition of a Frobenius algebra, we define arbitrary arity nodes to generalise the theory. We then demonstrate (and prove by induction) the equations which hold for symmetric, commutative, and special Frobenius algebras, taking advantage of the new node types defined in chapter 5. In each case we present a normal form result allowing arbitrary diagrams to be reduced to disconnected nodes.

6.1 Arbitrary Arity Input/Output Nodes

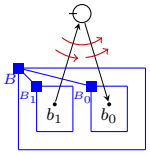
A Frobenius algebra can be described by the combination of a monoid ( ,) and a comonoid ( ,) along with the Frobenius law which governs their interaction. Writing Γ_{FA} for the set of axioms we get:



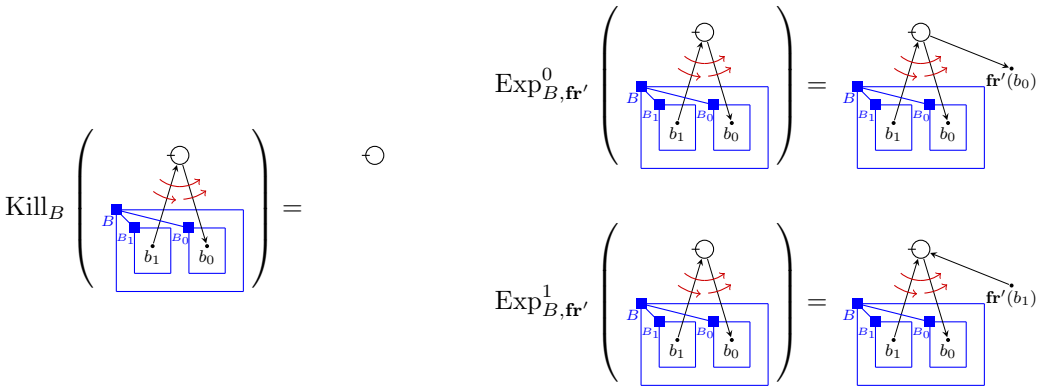
We might hope that we can replace the generators and rules of Γ_{FA} with an arbitrary arity version as in the monoid and bialgebra examples of section 4.4. It transpires this is possible but using a more complicated type of arbitrary arity node. So far we have only looked at nodes with either an arbitrary number of inputs (fixed outputs) or an arbitrary number of outputs (fixed inputs). For the Frobenius algebra case, we will require nodes which have arbitrary arrangements of inputs and outputs, for example:



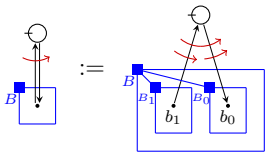
Clearly it is not possible to describe the family of such generators using a node with only a single !-box. Fortunately it can be represented by using the !-box nesting demonstrated in the following diagram:



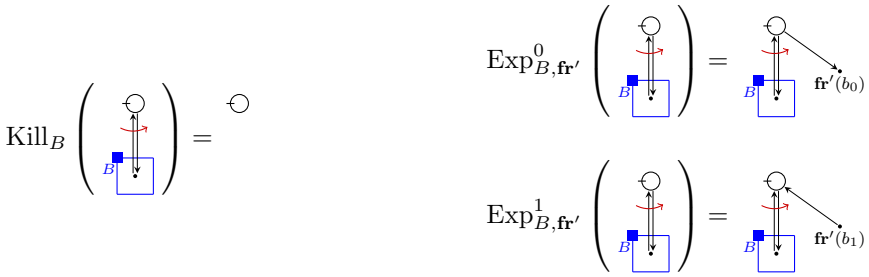
Having these nested !-boxes results in many options for partial instantiations. To see how the concrete instances of this are all possible arbitrary arrangements of input and output edges, we only need three (sequences of) !-box operations. The first is $Kill_B$ which removes all three !-boxes. The second creates a single additional output edge (positioned anticlockwise) and can be achieved by $\mathrm{Kill}_{\mathbf{fr}(B_0)} \circ \mathrm{Exp}_{\mathbf{fr}(B_0),\mathbf{fr}_0} \circ \mathrm{Kill}_{\mathbf{fr}(B_1)} \circ \mathrm{Exp}_{B,\mathbf{fr}}$. We abbreviate this sequence to $\mathrm{Exp}^0_{B,\mathbf{fr}'}$ which demonstrates that we are creating a new copy of the contents of B_0. The fresh renaming $\mathbf{fr}'$ need only take b_0 to $\mathbf{fr}_0(\mathbf{fr}(b_0))$. The final operation written $\mathrm{Exp}^1_{B,\mathbf{fr}'}$ creates one additional copy of the contents of B_1. This creates a new input edge (anticlockwise) and is achieved by applying $\mathrm{Kill}_{\mathbf{fr}(B_1)} \circ \mathrm{Exp}_{\mathbf{fr}(B_1),\mathbf{fr}_1} \circ \mathrm{Kill}_{\mathbf{fr}(B_0)} \circ \mathrm{Exp}_{B,\mathbf{fr}}$.



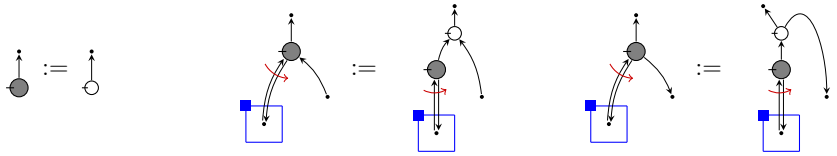
To make the diagrams easier to read we will often employ a shorthand notation, drawing only a single !-box but still two edges into it.



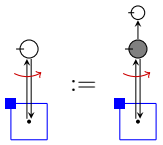
Hence the !-box operations can be drawn as:



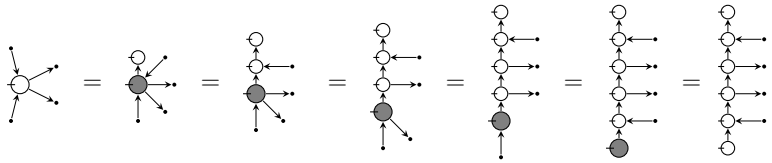
These three operations suggest the possibility of recursive definition of such nodes. We can give definitions for the base case of no edges and then define addition of a single input and of a single output. Recall from the monoid example that we used a single free edge at the top of the node to add extra structure. We do the same for Frobenius algebras by first recursively defining a node with a single output followed by an arbitrary arrangement of edges. Using the same shorthand as above we define:



The unit is used to define a node with only the top output, then we use multiplies and comultiplies respectively to add inputs and outputs. Finally, we plug the top output using the Frobenius counit so that completely arbitrary arrangements are possible.



Expanding these definitions we can see what form such an arbitrary node takes:



6.1.1 Induction on Arbitrary Input/Output

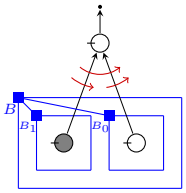
Given the above recursive definition, a natural question is whether we can prove using induction on B appearing in the form $[[\ldots]^{B_0}[\ldots]^{B_1}]^B$. We would suspect that from $\vdash \mathrm{Kill}_B(X)$, $X \vdash \mathrm{Exp}^0_{B,\mathbf{fr}}(X)$ and $X \vdash \mathrm{Exp}^1_{B,\mathbf{fr}}(X)$ we could deduce $\vdash X$ (and hence $\vdash \forall B.X$). We will refer to these three sequents respectively as the **base** case, **step 0** case and **step 1** case.

Theorem 6.1.1. *Suppose the !-formula X contains the subexpression $[[\ldots]^{B_0}[\ldots]^{B_1}]^B$ (so the only contents of B are B_0, B_1, and their contents). If B, B_0, and B_1 do not appear in Γ then we can apply the following induction rule:*

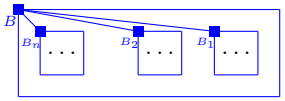
$$\frac{\Gamma \vdash \mathrm{Kill}_B(X) \qquad \Gamma, X \vdash \mathrm{Exp}^0_B(X) \qquad \Gamma, X \vdash \mathrm{Exp}^1_B(X)}{\Gamma \vdash X}\ (Induct^{0,1})$$

Proof. The proof is given in Appendix B. □

Remark 6.1.2. This new form of induction is not restricted to arbitrary arrangements of input and output edges. For example, for a white node taking an arbitrary number of inputs we could use (Induct ^{0,1}) to prove things about:



Extensions could also be given if the !-box B contains more !-boxes, so long as it only contains !-boxes (no nodes or edges directly inside B):

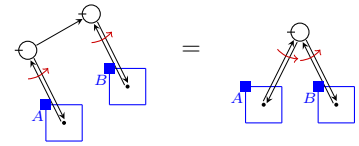


In this case we would need one step case for each of B_1 to B_n. These would represent creating a new copy of the contents of each !-box.

6.1.2 Combining Frobenius Nodes

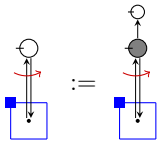
We now wish to prove that two Frobenius nodes with aligned ticks can be combined to form a single node.

$$\frac{}{\Gamma_{FA} \vdash \quad \forall A.\forall B.}\ (6.1.4)$$



where we have used the shorthands:

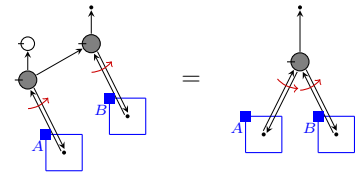
Recall that our Frobenius nodes are defined via:



where the gray nodes are recursively defined. We first prove a lemma about combining the gray nodes.

Lemma 6.1.3.

$$\frac{}{\Gamma_{FA} \vdash \quad \forall A.\forall B.}\ (6.1.3)$$

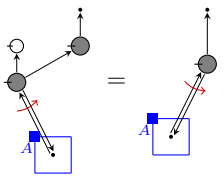


Proof. By applying our new form of induction on B (and its nested !-boxes):

$$\frac{\Gamma_{FA} \vdash \mathrm{Kill}_B(X) \qquad \Gamma_{FA}, X \vdash \mathrm{Exp}^0_B(X) \qquad \Gamma_{FA}, X \vdash \mathrm{Exp}^1_B(X)}{\Gamma_{FA} \vdash X}\ (\mathrm{Induct}^{0,1})$$

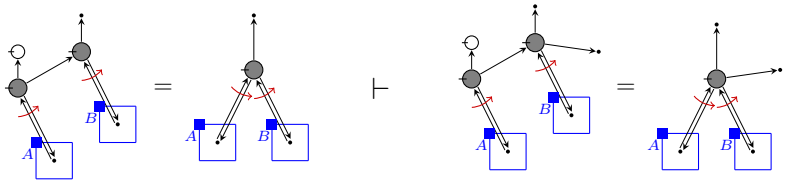
the lemma breaks down into three cases:

$$\Gamma_{FA} \quad \vdash$$



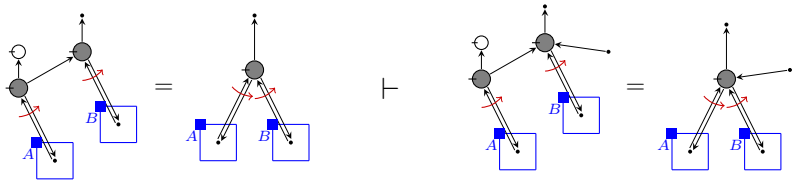
(**base**)

$$\Gamma_{FA}\ ,$$



(**step 0**)

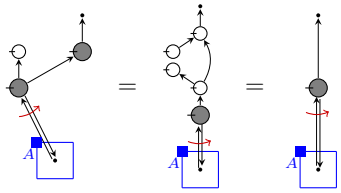
Γ_{FA} ,



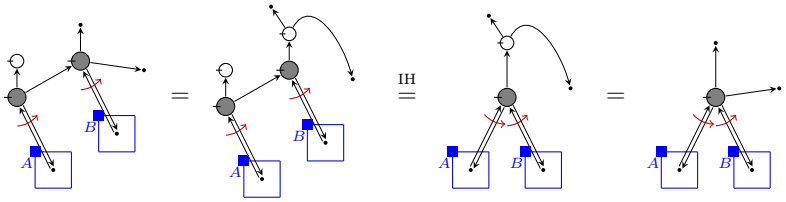
(**step 1**)

...each of which has a simple rewriting proof:

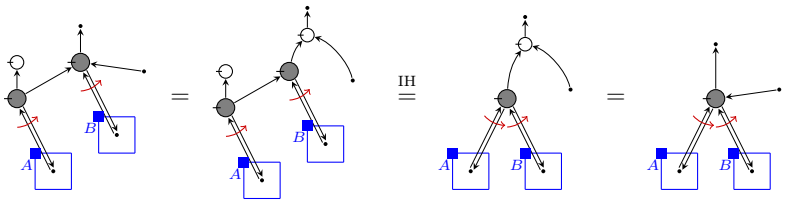
base:



step 0:



step 1:

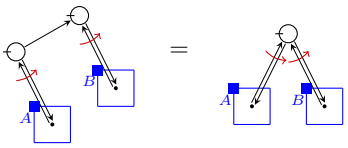


The equalities labelled as IH follow from the inductive hypothesis via the rule (Concat) with the Frobenius comultiply and multiply respectively. □

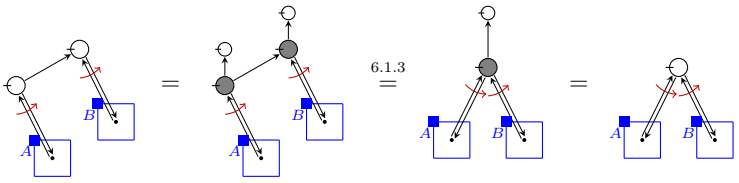
From which our earlier claim follows by rewriting:

Theorem 6.1.4.

$$\frac{}{\Gamma_{FA} \vdash \quad \forall A.\forall B.} \quad (6.1.4)$$



Proof.

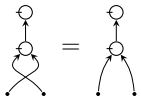


□

Note that this theorem allows us to combine two nodes with an edge from the first position of one node to the last position of the other (clockwise from the tick). We could similarly prove the ability to combine nodes if we have an edge from the last position of one node to the first of the other. To prove more interesting theorems we now start to look at different classes of Frobenius algebras.

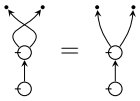
6.2 Symmetric Frobenius Algebras

If a Frobenius algebra also satisfies the symmetry axiom:



then we refer to it as a symmetric Frobenius algebra. We write Γ_{SFA} for the axioms of Γ_{FA} along with the symmetry axiom.

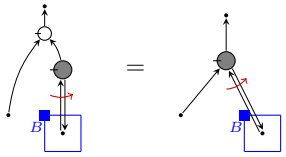
Remark 6.2.1. In a Frobenius algebra the symmetry axiom implies its codiagram:



As the name suggests, nodes in symmetric Frobenius algebras satisfy a property that edges are able to 'pass through the tick'. We hope to employ the metarule (Symm) to deduce that the family of symmetric Frobenius algebra morphisms can be expressed using symm-nodes. We prove the ability to pass an edge through the tick by the following theorems (keeping the previous section's notation for B).

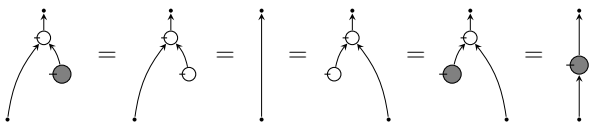
Lemma 6.2.2.

$$\frac{}{\Gamma_{SFA} \vdash \quad \forall B.} \quad (6.2.2)$$

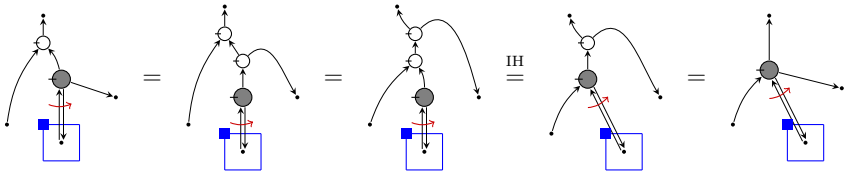


Proof. By induction on B we can break this into three cases, each of which has a simple rewriting proof:

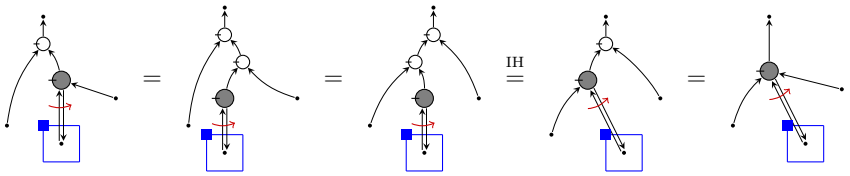
base:



step 0:



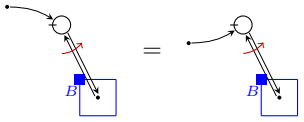
step 1:



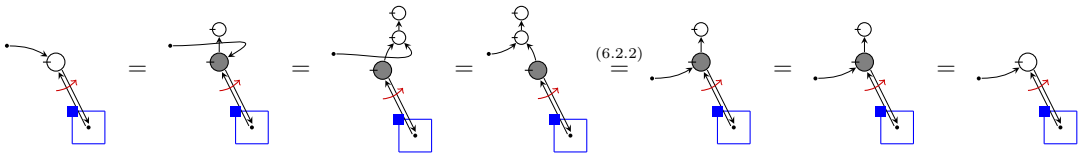
where the two equalities labelled IH follow from the inductive hypothesis via the rule (Concat) with the Frobenius comultiply and multiply respectively. □

Theorem 6.2.3.

$$\frac{}{\Gamma_{SFA} \vdash \quad \forall B.} \quad (6.2.3)$$



Proof.

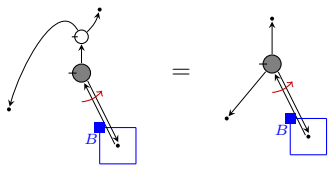


□

Similarly, for output edges:

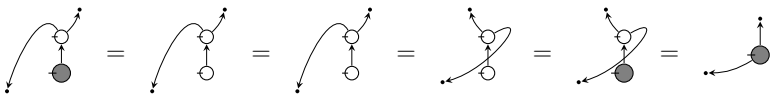
Lemma 6.2.4.

$$\frac{}{\Gamma_{SFA} \vdash \quad \forall B.} \quad (6.2.2)$$

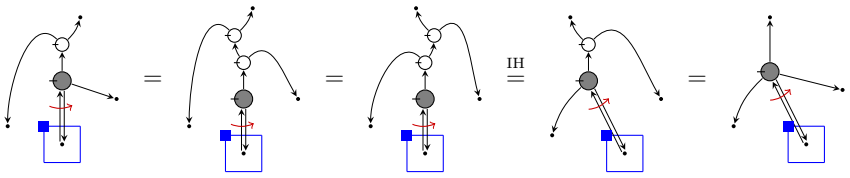


Proof. As above, by induction on B, we can break this into three cases which we prove separately by rewriting:

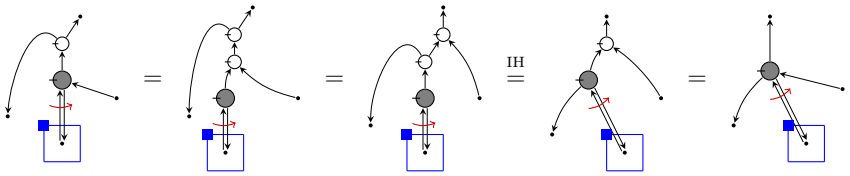
base:



step 0:



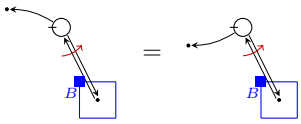
step 1:



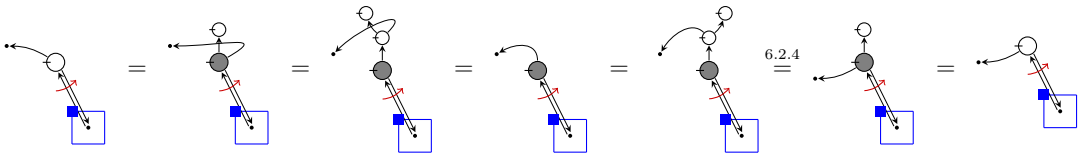
□

Theorem 6.2.5.

$$\frac{}{\Gamma_{SFA} \vdash \quad \forall B.} \quad (6.2.5)$$



Proof.



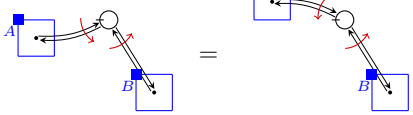
□

From this we would expect to be able to pass any arrangement of input and output edges through the tick. Using our standard shorthand notation:

this can be written as follows and proved via induction:

Theorem 6.2.6.

$$\frac{}{\Gamma_{SFA} \vdash \quad \forall A. \forall B.} \quad (6.2.6)$$

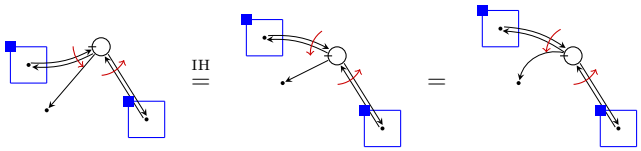


Proof. Writing X for the !-tensor equation above, we wish to prove $\Gamma_{SFA} \vdash \forall A. \forall B. X$. We start by applying $(Induct_A^{0,1})$ on the !-formula $\forall B. X$ to break the goal into three cases:

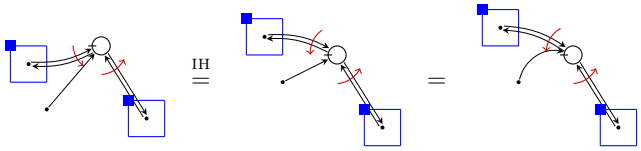
$$\dfrac{\dfrac{\Gamma_{SFA} \vdash \forall B.\,\mathrm{Kill}_A(X) \qquad \Gamma_{SFA}, \forall B.X \vdash \forall B.\,\mathrm{Exp}^0_A(X) \qquad \Gamma_{SFA}, \forall B.X \vdash \forall B.\,\mathrm{Exp}^1_A(X)}{\Gamma_{SFA} \vdash \forall B.X}\,(\mathrm{Induct}^{0,1})}{\Gamma_{SFA} \vdash \forall A.\forall B.X}\,(\forall I)$$

The base case is trivially true, the step cases have simple rewriting proofs:

step 0:

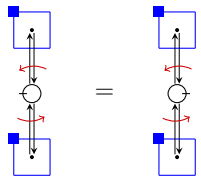


step 1:

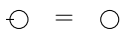


Note that the IH steps come from applying $\mathrm{Flip}_B \circ \mathrm{Exp}^0_B \circ \mathrm{Flip}_B$ and $\mathrm{Flip}_B \circ \mathrm{Exp}^1_B \circ \mathrm{Flip}_B$ respectively to the induction hypothesis. This explains the need to keep the quantification of B while applying the induction principle. □

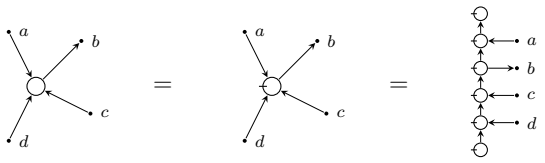
Another way to write this demonstrates how we have shown ticks can be positioned arbitrarily on symmetric Frobenius algebra nodes:



Hence applying the metarule (Symm) we can replace arbitrary symmetric Frobenius algebra nodes with symm-nodes:



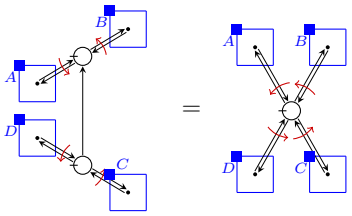
When we need to go back to the recursive definition we can always reintroduce the tick at any location and then expand the definition out from there.



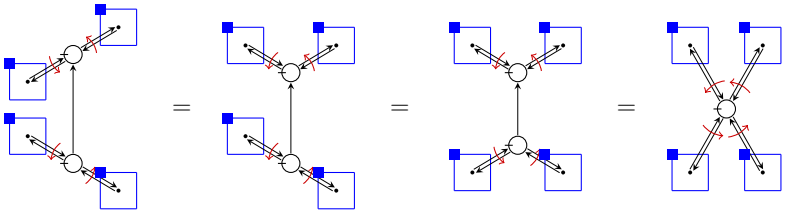
Theorem 6.1.4 now tells us that any two connected nodes can be combined (since tick positioning is irrelevant) but we cannot change the order of edges. Formally, we could prove this for arbitrary nodes by rewriting:

Theorem 6.2.7.

$$\Gamma_{SFA} \vdash \quad \forall A.\forall B.\forall C.\forall D. \qquad (6.2.7)$$

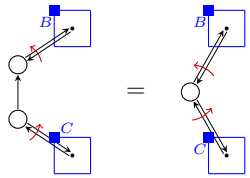


Proof. By rewriting:

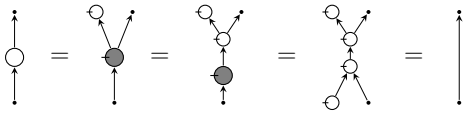


where the third equality comes from theorem 6.1.4 by the partial instantiation $\mathrm{Copy}_B \circ \mathrm{Copy}_A$. □

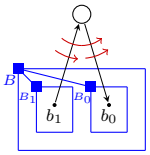
However, using the new symm-nodes directly in theorem 6.1.4 we get the much simpler symmetric presentation of the same rule:



Remark 6.2.8. Similarly to the monoid case of remark 4.4.3 we can simplify an arbitrary arity node with single input and single output to an identity wire:

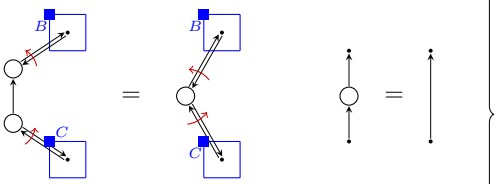


Theorem 6.2.9. *The finite arity theory of symmetric Frobenius algebras Γ_{SFA} can be replaced by the arbitrary arity version $\Gamma_{SFA!}$ made up of a single (symmetric) arbitrary arity generator:*

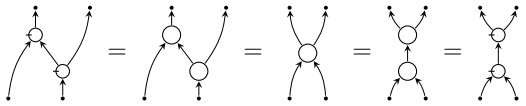


along with the !-tensor equations:

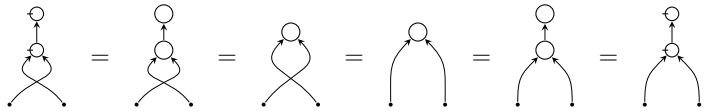
$$\Gamma_{SFA!} = \left\{ \quad \text{SFA Laws} \quad \right\}$$



Proof. We have seen how the arbitrary arity node can be defined in terms of fixed arity generators and proved the equations in $\Gamma_{SFA!}$. Now we need only check that $\Gamma_{SFA!}$ covers all of the axioms in Γ_{SFA}. For example, the first half of the Frobenius law can be shown by:

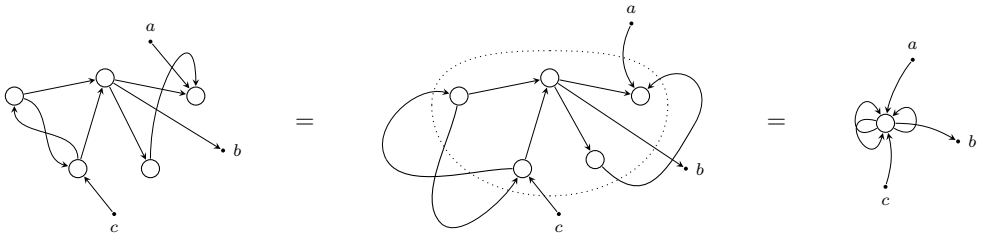


where the second and third equalities are instances of theorem 6.2.7. The symmetry law can be shown by:



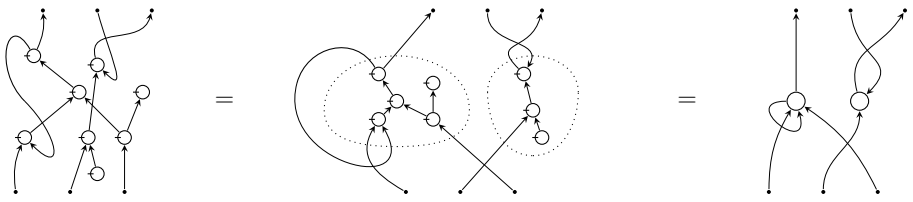
where the second and fourth equalities are instances of theorem 6.2.7 and the third, an instance of theorem 6.2.6, appears trivial in our new notation. The other laws have similar proofs. □

Repeated application of theorem 6.2.7 tells us that we can combine any number of such nodes so long as we keep track of the order of edges. Specifically any planar, acyclic, connected network of symmetric Frobenius algebra nodes can be combined to a single node. The following illustration demonstrates this process by deforming a connected network until we can easily see a planar, acyclic sub-diagram which can be combined to a single node.



(6.1)

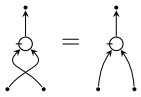
It is then clear how transitioning from fixed to arbitrary arity removed unnecessary structure. The following diagram demonstrates how we can use theorem 6.2.7 to simplify any diagram built from the fixed arity Frobenius algebra generators $\{$  ,  ,  ,  $\}$.



The process is to find a spanning tree for each connected component, then use naturality and the compact closed structure to pull out any loops or wire crossings. We then collapse each spanning tree to an arbitrary arity symmetric Frobenius node, being careful to preserve the order on edges entering/leaving the planar, acyclic section containing the spanning tree.

6.3 Commutative Frobenius Algebras

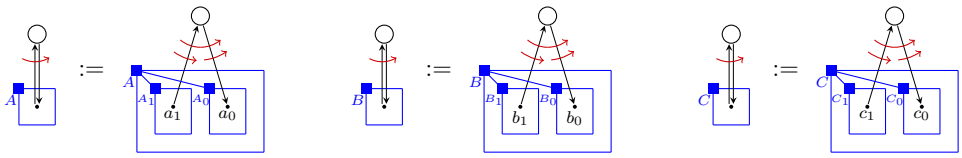
If a Frobenius algebra also satisfies commutativity



then we refer to it as a commutative Frobenius algebra. We write Γ_{CFA} for the theory of commutative Frobenius algebras (i.e. Γ_{FA} with the commutativity axiom).

Remark 6.3.1. It follows from the Frobenius law that commutativity of the multiplication operation (as above) implies cocommutativity of the comultiplication. Commutativity also subsumes the property of symmetry and hence for the rest of this section we can switch from nodes with ticks to plain circular nodes.

Commutativity specifies that edge order is unimportant on the generating nodes. We now demonstrate that, by the metarule (Comm), this extends to commutativity of the arbitrary arity nodes letting us replace arbitrary nodes with comm-nodes. Throughout this section we use our standard notation for arbitrary in/out !-boxes A, B and C.

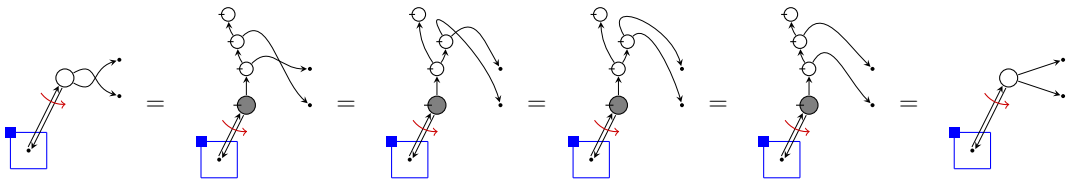


We wish to prove that any two neighbouring edges can commute past each other. There are three cases to check: commuting two inputs; commuting two outputs; and commuting an input with an output.

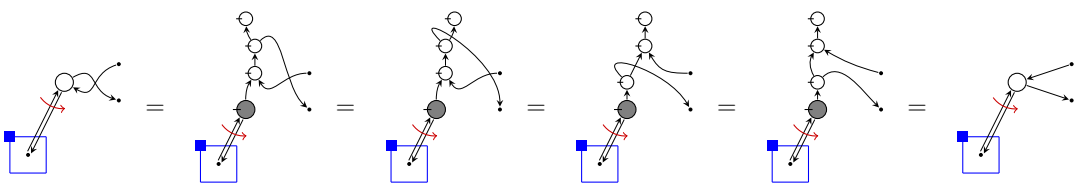
Lemma 6.3.2. *Any two neighbouring edges on a commutative Frobenius algebra node commute.*

Proof. We show each case individually by rewriting:

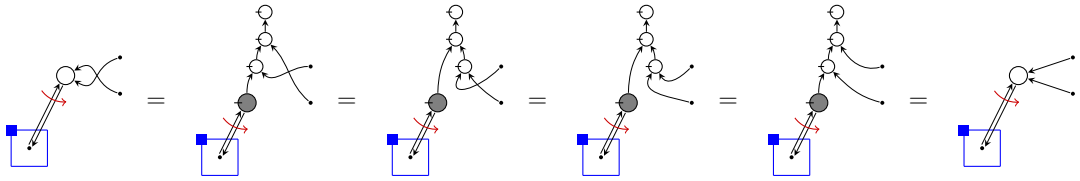
out edges commute:



out edges commute with in:



in edges commute:

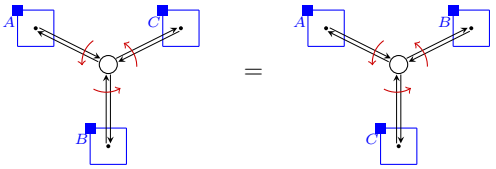


□

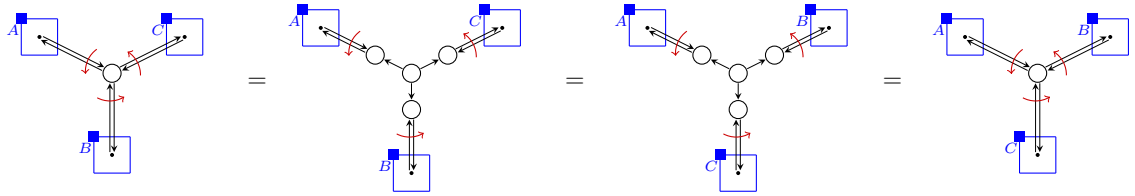
From this we would expect to be able to pass any arrangement of input and output edges through each other. We could prove this by induction: first proving that a single output can pass through an arbitrary arrangement, then similarly for a single input. Finally, we would conclude that arbitrary arrangements can commute by a final induction proof. However, in this case we can avoid induction and in fact prove commutativity of arbitrary arrangements of edges using only theorem 6.1.4 and one of the cases above. Note that for ease of readability we have not fixed the position of free edges/!-boxes.

Theorem 6.3.3.

$$\frac{}{\Gamma_{CFA} \vdash \forall A.\forall B.\forall C.} \quad (6.3.3)$$

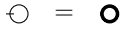


Proof.

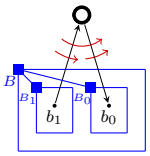


where the first and third steps come from repeated use of theorem 6.1.4 and the second step is one of the instances of lemma 6.3.2 applied to the centre node. □

Since edges can now commute at will, we deduce that all instances of commutative Frobenius algebra nodes are commutative, so (Comm) lets us switch to comm-nodes which we denote diagrammatically by drawing the nodes in bold.

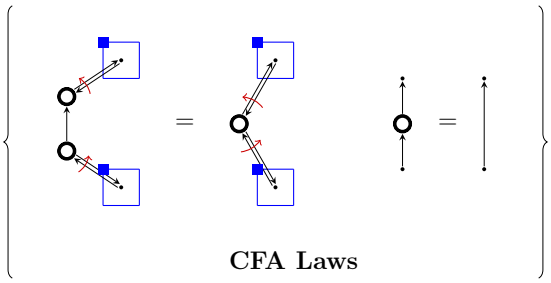


Theorem 6.3.4. *The finite arity theory of commutative Frobenius algebras Γ_{CFA} can be replaced by the arbitrary arity version $\Gamma_{CFA!}$ made up of a single (commutative) arbitrary arity generator:*



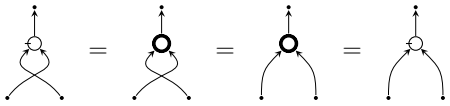
along with the !-tensor equations:

$\Gamma_{CFA!} =$



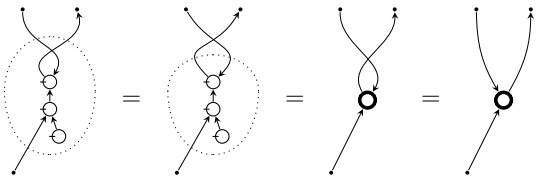
CFA Laws

Proof. By theorem 6.2.9 we need only check the commutativity part. We have seen how theorem 6.3.3 follows from Γ_{SFA}, then we trivially check:

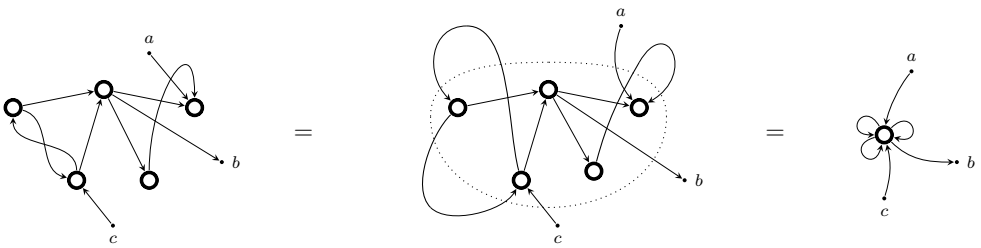


where the second equality is an instance of theorem 6.3.3. □

This allows us to generalise the process described in (6.1). We no longer need to restrict to planar diagrams. Any crossings inside a sub-diagram could be pulled out, then removed by commuting the edges:

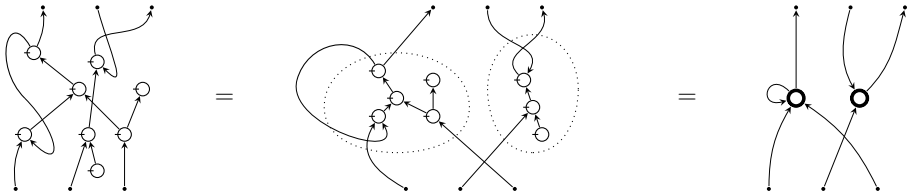


Hence any acyclic, connected network of commutative Frobenius algebra nodes can be combined to a single node as illustrated below.



(6.2)

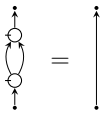
Again this is a clear advantage over dealing with the fixed arity theory. We can now reduce any diagram in a fixed arity commutative Frobenius algebra to disconnected commutative arbitrary arity nodes:



The process is to find a spanning tree for each connected component, then use naturality and the compact closed structure to pull out any loops. We then collapse each spanning tree to an arbitrary arity commutative Frobenius node.

6.4 Special Frobenius Algebras

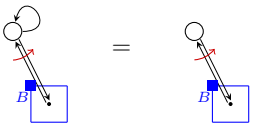
If a commutative Frobenius algebra satisfies the condition:



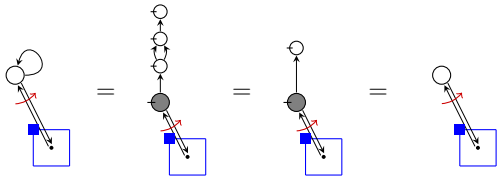
then we refer to it as a special commutative Frobenius algebra (SCFA), writing Γ_{SCFA} for the set of axioms. Speciality together with symmetry allows us to drop adjacent loops from nodes.

Theorem 6.4.1.

$$\frac{}{\Gamma_{SCFA} \vdash \quad \forall B.} \quad (6.4.1)$$



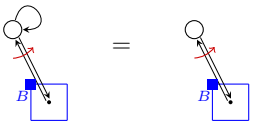
Proof.



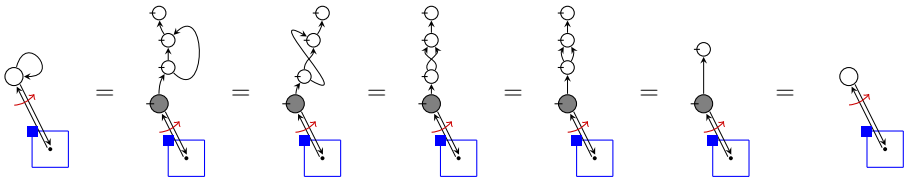
□

Theorem 6.4.2.

$$\frac{}{\Gamma_{SCFA} \vdash \quad \forall B.} \quad (6.4.2)$$

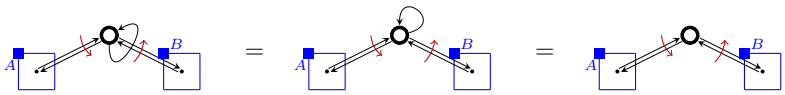


Proof.



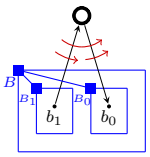
□

Adding commutativity we can trivially make any loop into an adjacent loop and hence loops can be dropped altogether.



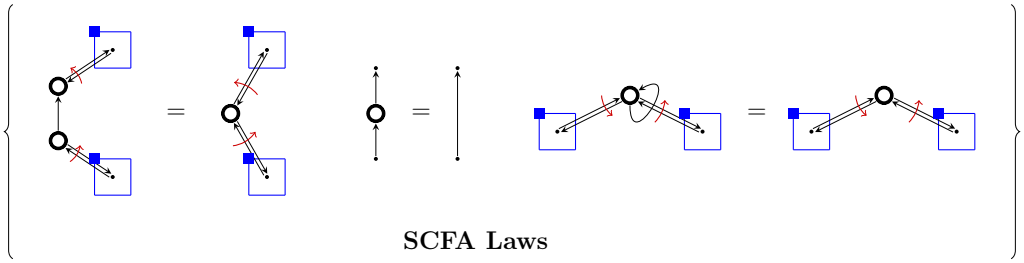
(6.3)

Theorem 6.4.3. *The finite arity theory of special commutative Frobenius algebras* Γ_{SCFA} *can be replaced by the arbitrary arity version* $\Gamma_{SCFA!}$ *made up of a single arbitrary arity generator:*

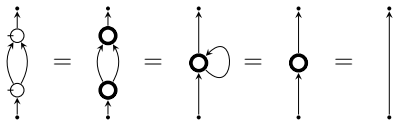


along with the !-tensor equations:

$\Gamma_{SCFA!}$ =

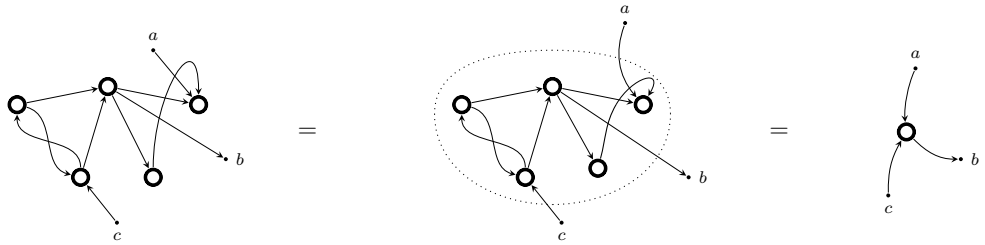


Proof. We have seen how $\Gamma_{SCFA!}$ follows from Γ_{SCFA}, so by theorem 6.3.4 we only need to check the special law follows from $\Gamma_{SCFA!}$:



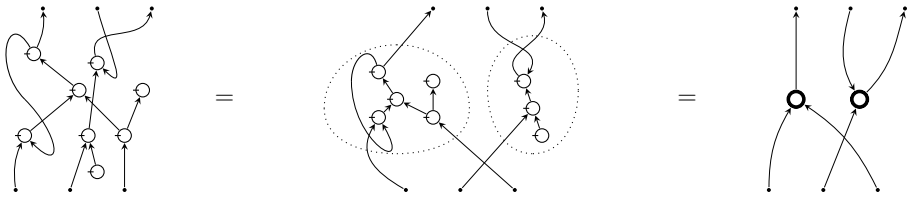
where the second, third, and fourth equalities are from theorem 6.2.7, theorem 6.3 and remark 6.2.8 respectively. □

This allows us to drop the acyclic condition from the process in (6.2) and conclude that any connected network of nodes in a special commutative Frobenius algebra can be combined to a single node:



(6.4)

As expected this allows us to reduce any diagram in a SCFA to disconnected nodes without loops:



(6.5)

Chapter 7

Conclusions

The first major contribution of this thesis is a formalism for families of string diagrams called the !-tensor formalism. The previous (combinatoric) !-graph formalism allowed descriptions of infinite families of diagrams but was restricted to nodes which were independent of the order of edges (called commutative nodes). !-Tensors extend the graphical notation to noncommutative nodes while still allowing reasoning with infinite families of diagrams. This allowed us to describe a larger variety of theory, such as those involving a twisting property where edges need to reverse order between two nodes.

To reason with !-tensors we presented a syntactic formalism, based on Penrose's abstract tensor notation. We then defined !-box operations which allow us to retrieve the concrete diagrams represented by a !-tensor. They similarly allow us to retrieve all concrete diagram equations represented by a !-tensor equation.

!-Tensors allow more than just working with noncommutative theories, another of their major features is the ability to definitionally extend a theory. In section 3.4 we demonstrated the ability to define new nodes to replace diagrams, including recursively defining families of nodes, such as trees of multiplication operations. We presented a few examples of how multiple fixed arity generators in a theory can be replaced by a single arbitrary arity generator. For example, monoids, antihomomorphisms, and bialgebras each admit a node representing the family of left associated trees of multiplications. Definitional extension was not possible in the !-graph formalism since the newly defined nodes would always be assumed to be commutative.

Next we developed !-logic (!L) as a framework to reason with !-tensors. !-Formulas were defined to have !-tensor equations as atoms and the connectives: conjunction, implication, and universal quantification over !-box components. We then presented the rules of !L in the form of a natural deduction system over sequents $\Gamma \vdash X$. We started with rules analogous to those of positive intuitionistic first order logic and concluded with !-box induction, which is the powerhouse of !L.

In section 4.4 we illustrated the power of !-box induction for a few examples of recursively defined arbitrary arity nodes where we proved that a fixed arity theory can be replaced by an (often simpler) arbitrary arity theory.

To give an interpretation to !L we defined a semantics $[\![-]\!]$ for !-formulas and sequents based on valuations of the generators. This allowed us to show soundness of !L by checking each rule preserved truth of sequents with respect to $[\![-]\!]$.

While transitioning to a non-commutative formalism has many clear advantages, if we wish to work with commutative morphisms, the additional structure only gets in the way. We demonstrated in chapter 5 that no single !-tensor equation can cover all equalities expected of a commutative morphism. Hence we decided to implement commutativity at the level of !-tensors by augmenting the ability to tag those nodes which represent commutative morphisms. The ability for edges to commute around such nodes is then built in to !-tensor notation. Diagrammatically, we introduced circular nodes to represent symmetric morphisms (those where edges can undergo cyclic permutations) and bold circles to represent commutative morphisms. We then presented metarules which describe conditions under which nodes can be replaced by their symmetric or commutative counterparts.

We concluded by working through the example of Frobenius algebras which exemplify many of the advancements made in this thesis. Frobenius algebras admit a particularly interesting variable arity node which allows arbitrary arrangements of input and output edges. We demonstrated how this family of nodes could be described using nested !-boxes and defined recursively. This is only possible by using !-tensors.

Having described the tools to talk about frobenius algebras we could then prove many of their fundamental properties using !L, particularly !-box induction. We first proved a spider theorem allowing two connected nodes with aligned ticks to be merged. In the example of a symmetric Frobenius algebra, we then proved that the theory can be described by a single symmetric generator and a spider law allowing nodes to be combined. From this we demonstrated how any planar, acyclic, connected network of symmetric Frobenius algebra nodes can be combined to a single node. Similarly, for commutative Frobenius algebras we removed the restriction that edges cannot commute, hence showing that any acyclic, connected network of commutative Frobenius algebra nodes can be combined to a single commutative node.

7.1 Future Work

7.1.1 Implementation

We have already suggested one clear avenue of future work in section 3.5 where we stated that it would be useful to implement non-commutativity of nodes in the graphical proof assistant Quantomatic [27]. The idea for Quantomatic arose when graphical languages became useful in areas such as quantum

information theory. As diagrams grew larger and more complex it became difficult to work with them rewriting by hand. Quantomatic was introduced as a means to automate parts of this process. The user provides a set of axioms and then draws a diagram to be rewritten and interactively applies rewrites to the diagram.

For an example, take the theory of a commutative bialgebra, as presented in section 5.3. We can implement the bialgebra law as can be seen in the Figure 7.1.

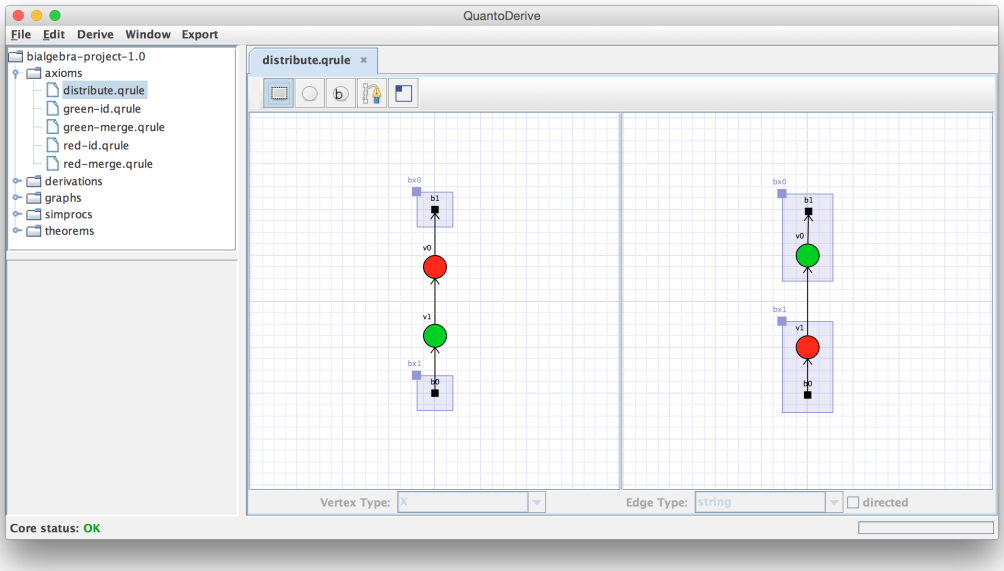


Figure 7.1: Bialgebra Law in Quantomatic

We use Quantomatic to explore the tree of possible rewrites of a diagram. Figure 7.2 demonstrates a diagram in the process of being rewritten. Note that on the left hand side we can see the different rewrite paths we have attempted to take.

Rewriting can be achieved by manually choosing each step or by applying a simplification procedure (simproc). A simproc is a Poly/ML function which iteratively chooses which rewrite to apply to the given graph.

The current version of Quantomatic is based on !-graphs and hence has a combinatoric architecture. We would like to implement a similar semi-automated theorem prover for !-tensors. There are two options for this: A term rewriting program could be built from scratch based on !-tensor expressions; or we could adapt the code for Quantomatic to also allow noncommutative nodes. The former may provide some efficiency advantages over the latter but involves a large amount of work and we would prefer to have one program to cover both. To that end we developed the theory in section 3.5 which demonstrates how !-tensors can be encoded as !-graphs with some additional data about each node called a neighbourhood order.

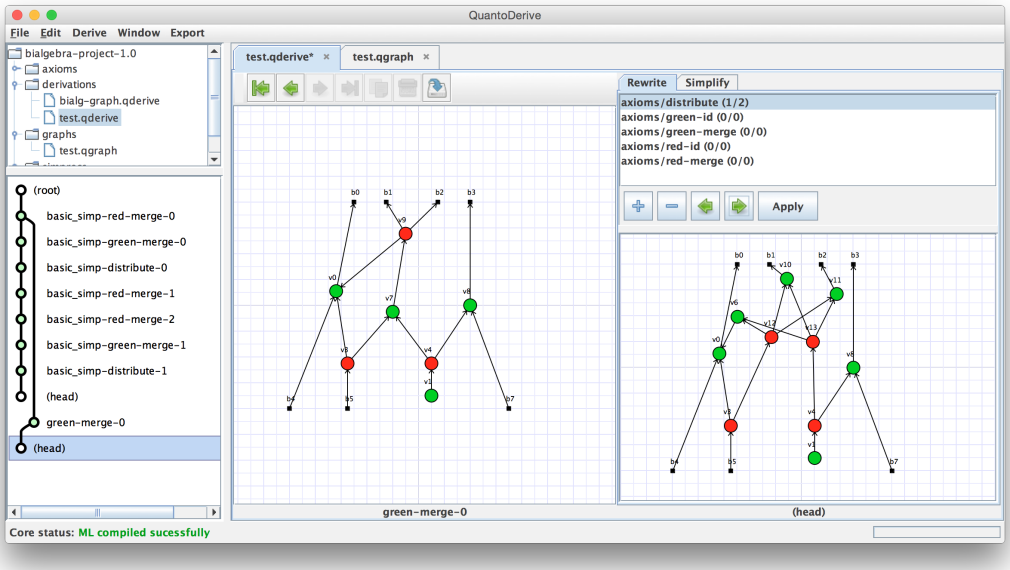
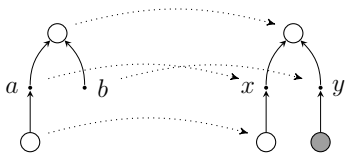


Figure 7.2: Proof Derivation in Quantomatic

It is simple to adapt quantomatic to understand neighbourhood orders and check that a !-tensor encoded as a !-graph with neighbourhood order is well formed. For matching we need to be more careful. For instance the following is a !-graph matching:



Suppose we have a neighbourhood order on these with the top nodes having edgeterms ab and yx for the left and right hand side respectively. Drawing as !-tensors it is clear we no longer have a match:

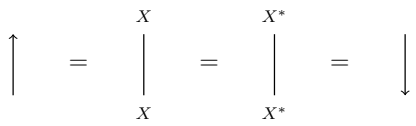
Instead we need to slightly restrict the notion of matching from [39] to respect edgeterms. A matching from (G, nhd_G) to (K, nhd_K) should be a !-graph matching $i : G \to K$, satisfying:

$$i(\mathrm{nhd}_G(v)) = \mathrm{nhd}_K(i(v)) \qquad \forall v \in N(G)$$

The next step is to develop an efficient matching algorithm and implement this along with !-tensor substitution. We leave this as future work.

7.1.2 Self Dual Objects

One area which would benefit from being formalised is the ability for object types to be self dual. If we are given that $X = X^*$, then wire direction for X becomes meaningless, hence it would make sense to replace directed wires with undirected wires.



To reflect this we would like to drop the direction notation in tensor and !-tensor expressions. We could do this by replacing $\hat{a}$ and $\check{a}$ with $\bar{a}$ when the type of a is self dual. The definitions would need to reflect this with the new identity edge equivalence $1_{\bar{a}\bar{b}} \equiv 1_{\bar{b}\bar{a}}$.

Writing $\#\hat{a}$ for the number of occurrences of $\hat{a}$ we could define an edgename a in G to be

- *free* in G if $\#\hat{a} + \#\check{a} + \#\bar{a} = 1$
- *bound* in G if $(\#\hat{a}, \#\check{a}, \#\bar{a}) = (1, 1, 0)$ or $(0, 0, 2)$
- *invalid* in G otherwise

Then conditions F1-2 and C1-3 would need to be updated. For example, F1 would require that each $a \in \mathcal{N}$ must be valid in G.

A bound undirected edge a would have two contexts, one for each occurrence of $\bar{a}$. But the consistency conditions remain the same except for using these two contexts instead of $\mathrm{ctx}(\hat{a})$ and $\mathrm{ctx}(\check{a})$.

7.1.3 Completeness

Having shown soundness of !L the obvious next step is completeness. The statement of completeness is:

Conjecture 7.1.1 (Completeness)**.** If $[\![\Gamma \vdash X]\!]$ is true for any compact closed category $\mathcal{C}$, then $\Gamma \vdash X$ is derivable in !L.

String diagrams (and hence tensors) are known to be sound and complete for compact closed categories. i.e. Concrete !-tensor equations are true in all models if and only if they are identical tensors. Hence for general !-tensor equations the problem reduces to deciding whether two !-tensors with corresponding !-boxes have identical instances. This covers atomic !-formulas, but when we involve conjunction, implication, and universal quantification, things become less clear. We leave it as future work to either prove completeness of !L (or possibly an extension of !L) or to prove no such complete extension exists.

Appendix A

Reordering !-Box Operations

This appendix presents some useful lemmas describing when and how !-box operations can be commuted past each other. These lemmas are used in the proofs of theorem 3.3.22, theorem 4.1.12, and the (Copy), (Drop), and (Flip) cases of theorem 4.6.3.

Proving that two different sequences of operations are equivalent on a !-tensor expression G is generally done by structural induction on G. Take the example of (3.5) which states that the expression $\mathrm{Exp}_{B,\mathbf{fr}}(G)$ can equivalently be written $\mathrm{Drop}_{\mathbf{fr}(B)} \circ \mathrm{Copy}_{B,\mathbf{fr}}(G)$. Most of the cases in the inductive definition of a !-tensor expression are trivial to check, with the interesting cases being $[G]^B$, $\langle e]^B$, and $[e\rangle^B$. For example, the $[G]^B$ case is shown by the following, where each step is from the definition of a !-box operation.

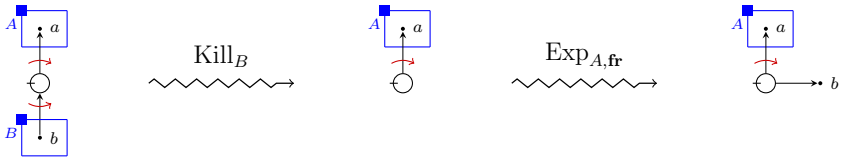
$$\mathrm{Drop}_{\mathbf{fr}(B)} \circ \mathrm{Copy}_{B,\mathbf{fr}}([G]^B) = \mathrm{Drop}_{\mathbf{fr}(B)}([G]^B[\mathbf{fr}(G)]^{\mathbf{fr}(B)}) = [G]^B\,\mathbf{fr}(G) = \mathrm{Exp}_{B,\mathbf{fr}}([G]^B)$$

The proofs for $\langle e]^B$ and $[e\rangle^B$ are similar. For more complicated equations involving multiple operations on different and possibly nested !-boxes, the proofs become tedious and so we will omit them here. We do, however, need to be careful about renamings, in particular checking appropriate fresh renamings exist for each operation. In the above example we know $\mathbf{fr}$ is fresh for G, since it is used in $\mathrm{Exp}_{B,\mathbf{fr}}$ when applied to G, and so $\mathrm{Copy}_{B,\mathbf{fr}}$ is a valid !-box operation on G. In the more advanced cases below we will often introduce new fresh renamings and will need to check they are fresh on their domains.

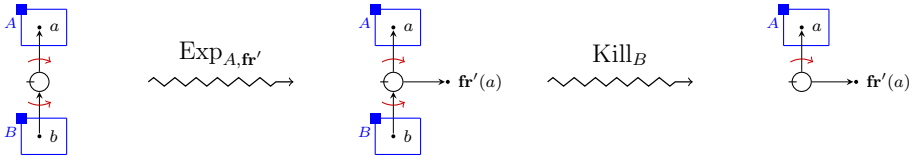
Remark A.0.1. The fresh renaming $\mathbf{fr}$ in $\mathrm{Op}_{A,\mathbf{fr}}(G)$ is only ever applied to names contained in A, by which we mean edges a such that $A \in \mathrm{ctx}_G(a)$ and !-boxes $B \preccurlyeq_G A$. Hence when defining such a fresh renaming $\mathbf{fr}$ for use in $\mathrm{Op}_{A,\mathbf{fr}}(G)$ we often only explicitly give its value on the contents of A, stating that it is fresh elsewhere. By this we mean that fresh names are chosen which are not in G or the image of $\mathbf{fr}$ on the contents of A. If we write $C(A)$ for the contents of A (both edge and box names), then we may define a fresh renaming as follows:

$$\mathbf{fr}(x) = \begin{cases} x' & x \in C(A) \\ \text{fresh} & \text{otherwise} \end{cases}$$

In some cases it will not be possible to rearrange the operations and get the exact same result. For example, suppose we are given $\text{Exp}_{A,\mathbf{fr}} \circ \text{Kill}_B$ acting on the !-tensor G below, where $fr(a) = b$:



If we wish to apply the operation on A first, then we may hope this is equivalent to $\text{Kill}_B \circ \text{Exp}_{A,\mathbf{fr}'}(G)$ for some appropriate fresh renaming $\mathbf{fr}'$. The following diagram demonstrates what affect this has:



For the two to be equivalent we require $\mathbf{fr}'(a) = b$ which is not a valid fresh renaming on G. In these cases, where names prevent operation reordering, we use renamings to correct the names after reordering. Hence in this case we show $\text{Exp}_{A,\mathbf{fr}} \circ \text{Kill}_B(G) = \mathbf{rn} \circ \text{Kill}_B \circ \text{Exp}_{A,\mathbf{fr}'}(G)$ where $\mathbf{rn}$ need only take $\mathbf{fr}'(a)$ to b. Again we need to be careful that our chosen renaming is a well-defined bijection.

A useful lemma is that which allows us to commute a renaming to the left past any !-box operations:

Lemma A.0.2. *Renamings can be pushed left past !-box operations using the following equality:*

$$\text{Op}_{C,\mathbf{fr}} \circ \mathbf{rn}(G) = \mathbf{rn} \circ \text{Op}_{\mathbf{rn}^{-1}(C),\mathbf{rn}^{-1} \circ \mathbf{fr} \circ \mathbf{rn}}(G)$$

Proof. As stated above we will omit the structural induction on !-tensor expressions which is simple but tedious. We are left to prove that the !-box operation $\text{Op}_{\mathbf{rn}^{-1}(C),\mathbf{rn}^{-1} \circ \mathbf{fr} \circ \mathbf{rn}}$ is well-defined on G, i.e. that $\mathbf{rn}^{-1} \circ \mathbf{fr} \circ \mathbf{rn}$ is fresh for G.

Suppose, for a contradiction, that $\mathbf{rn}^{-1} \circ \mathbf{fr} \circ \mathbf{rn}(\text{edges}(G)) \cap \text{edges}(G) \neq \varnothing$. Hence there exist a and b in $\text{edges}(G)$ such that $\mathbf{rn}^{-1} \circ \mathbf{fr} \circ \mathbf{rn}(a) = b$, or equivalently $\mathbf{fr}(\mathbf{rn}(a)) = \mathbf{rn}(b)$.

This contradicts $\mathbf{fr}(\mathbf{rn}(\text{edges}(G))) \cap \mathbf{rn}(\text{edges}(G)) = \varnothing$ as enforced by freshness of $\mathbf{fr}$ on $\mathbf{rn}(G)$. A similar contradiction proof shows that $\mathbf{rn}^{-1} \circ \mathbf{fr} \circ \mathbf{rn}(\text{boxes}(G)) \cap \text{boxes}(G) = \varnothing$, so we can conclude that $\mathbf{rn}^{-1} \circ \mathbf{fr} \circ \mathbf{rn}$ is indeed fresh for G. □

A.1 Instantiations

To work with instantiations we wish to show that, given a total order on !-box names, they admit a normal form, up to renaming. The normal form deals with !-boxes from the top down, so operations on parents are always applied first. To do this we now present a couple of lemmas about reordering operations, which are used in the proof given in corollary 3.3.24. We wish to show that, up to renaming, operations on non-nested !-boxes can be commuted (possibly changing their fresh renamings) and that operations on a !-box A can commute right past operations on !-boxes nested in A.

Lemma A.1.1. *Given a !-tensor G and partial instantiation $\mathrm{Op}'_{A,\mathbf{fr}_A} \circ \mathrm{Op}_{B,\mathbf{fr}_B}$ where A and B are not nested inside each other there exist fresh renamings $\mathbf{fr}'_B, \mathbf{fr}'_A$, and $\mathbf{rn}$ such that:*

$$\mathrm{Op}'_{A,\mathbf{fr}_A} \circ \mathrm{Op}_{B,\mathbf{fr}_B}(G) = \mathbf{rn} \circ \mathrm{Op}_{B,\mathbf{fr}'_B} \circ \mathrm{Op}'_{A,\mathbf{fr}'_A}(G)$$

Proof. Since A and B are not nested inside each other, it is straightforward to show that applying operations in either order yields the same resulting !-tensor expression, up to renaming of some edges. Thus, it only remains to define the new renamings to ensure that the final edge names are correct. Let $\mathbf{fr}'_A$ and $\mathbf{fr}'_B$ act the same as $\mathbf{fr}_A$ and $\mathbf{fr}_B$, respectively, on box names. For edge names, let $\mathbf{fr}'_A$ be fresh for G taking a to a_1, then let $\mathbf{fr}'_B$ be fresh for $\mathrm{Op}'_{A,\mathbf{fr}'_A}(G)$ by taking each a to a new name a_2.

The resulting !-tensor expression $\mathrm{Op}_{B,\mathbf{fr}'_B} \circ \mathrm{Op}'_{A,\mathbf{fr}'_A}(G)$ may have names of the form a_1, a_2, and a_{12}, where a subscript 1 indicates the name results from the operation on A and a subscript 2 indicates the name results from the operation on B. Finally, we fix a bijection $\mathbf{rn}$ such that for the names in $\mathrm{Op}_{B,\mathbf{fr}'_B} \circ \mathrm{Op}'_{A,\mathbf{fr}'_A}(G)$:

$$\mathbf{rn} = \begin{cases} a_1 \mapsto \mathbf{fr}_A(a) \\ a_2 \mapsto \mathbf{fr}_B(a) \\ a_{12} \mapsto \mathbf{fr}_A(\mathbf{fr}_B(a)) \end{cases}$$

To see that such a bijection exists, we go through the cases for each operation. If both operations are killing, then $\mathbf{rn}$ can be the identity since none of the above names exist in $\mathrm{Op}_{B,\mathbf{fr}'_B} \circ \mathrm{Op}'_{A,\mathbf{fr}'_A}(G)$. If one operation is killing, then $\mathbf{rn}$ only needs to be defined as $a_1 \mapsto \mathbf{fr}_A(a)$ or $a_2 \mapsto \mathbf{fr}_B(a)$. Finally, if both operations are expansions, then such a bijection exists because the images of $\mathbf{fr}_A$, $\mathbf{fr}_B$, and $\mathbf{fr}_A \circ \mathbf{fr}_B$ in $\mathrm{Op}'_{A,\mathbf{fr}_A} \circ \mathrm{Op}_{B,\mathbf{fr}_B}(G)$ are disjoint. It is now possible to show through straightforward but tedious case analysis that the lemma holds. □

Given a !-tensor G and a partial instantiation $\mathrm{Op}'_{A,\mathbf{fr}_A} \circ \mathrm{Op}_{B,\mathbf{fr}_B}$ where B is nested in A we hope to rewrite the partial instantiation so that the operation on A is applied first. We split this into two cases based on the operation on A:

Lemma A.1.2. *If B is nested inside A in the !-tensor expression G, then:*

$$\mathrm{Kill}_A \circ \mathrm{Op}_{B,\mathbf{fr}_B}(G) = \mathrm{Kill}_A(G)$$

Proof. It is clear that killing A will erase any effects resulting from the operation $\mathrm{Op}_{B,\mathbf{fr}_B}$. We omit the tedious structural induction proof. □

Lemma A.1.3. *If we are given the partial instantiation $\mathrm{Exp}_{A,\mathbf{fr}_A} \circ \mathrm{Op}_{B,\mathbf{fr}_B}$ on G, where B is nested inside A, there exist renamings $\mathbf{fr}'_A$, $\mathbf{fr}'_B$, $\mathbf{fr}''_B$, $\mathbf{rn}$ such that:*

$$\mathrm{Exp}_{A,\mathbf{fr}_A} \circ \mathrm{Op}_{B,\mathbf{fr}_B}(G) = \mathbf{rn} \circ \mathrm{Op}_{\mathbf{fr}'_A(B),\mathbf{fr}''_B} \circ \mathrm{Op}_{B,\mathbf{fr}'_B} \circ \mathrm{Exp}_{A,\mathbf{fr}'_A}(G)$$

Proof. Again, it is straightforward to check that the sequences of operations $\mathrm{Exp}_{A,\mathbf{fr}_A} \circ \mathrm{Op}_{B,\mathbf{fr}_B}$ and $\mathrm{Op}_{\mathbf{fr}'_A(B),\mathbf{fr}''_B} \circ \mathrm{Op}_{B,\mathbf{fr}'_B} \circ \mathrm{Exp}_{A,\mathbf{fr}'_A}$ have the same result on !-tensors up to renaming of some edges. We now define the renamings $\mathbf{fr}'_A$, $\mathbf{fr}'_B$, $\mathbf{fr}''_B$, and $\mathbf{rn}$ for equality to hold.
Similarly to the proof of lemma A.1.1, we will tag names affected by operations on A with the subscript 1 and those affected by operations on B with the subscript 2. However, unlike in lemma A.1.1, we also need to apply the renaming to box names (since !-boxes can be nested inside both A and B).

Let $\mathbf{fr}'_A$ take each edge or !-box name x inside G to fresh name x_1. We then define $\mathbf{fr}'_B$ fresh on $\mathbf{fr}'_A(G)$ to take the name x to fresh name x_2. We need to be more careful when defining $\mathbf{fr}''_B$ as we hope to take x to x_2 but this is not necessarily fresh on names inside B. Fortunately it is fresh on the contents of B_1 ($= \mathbf{fr}'_A(B)$) which is where it will be applied during $\mathrm{Op}_{\mathbf{fr}'_A(B),\mathbf{fr}''_B}$. Hence we define $\mathbf{fr}''_B$ on edge and box names by:

$$\mathbf{fr}''_B(x) := \begin{cases} x_2 & x \in C(B_1) \\ \text{fresh} & \text{otherwise} \end{cases}$$

The resulting !-tensor expression $\mathrm{Op}_{\mathbf{fr}'_A(B),\mathbf{fr}''_B} \circ \mathrm{Op}_{B,\mathbf{fr}'_B} \circ \mathrm{Exp}_{A,\mathbf{fr}'_A}(G)$ may have names of the form x_1 corresponding to edges affected by the !-box operation $\mathrm{Exp}_{A,\mathbf{fr}_A}$, names of the form x_2 corresponding to edges affected by the operation $\mathrm{Op}_{B,\mathbf{fr}_B}$, and names of the form x_{12} corresponding to edges affected by both. We then fix a bijection $\mathbf{rn}$ such that:

$$\mathbf{rn} = \begin{cases} x_1 \mapsto \mathbf{fr}_A(x) \\ x_2 \mapsto \mathbf{fr}_B(x) \\ x_{12} \mapsto \mathbf{fr}_A(\mathbf{fr}_B(x)) \end{cases}$$

Such a bijection always exists since if the operation on B is killing, then this is simply $x \mapsto x_1$; if the operation is expanding then the images of $\mathbf{fr}_A$, $\mathbf{fr}_B$, and $\mathbf{fr}_A \circ \mathbf{fr}_B$ in $\mathrm{Exp}_{A,\mathbf{fr}_A} \circ \mathrm{Op}_{B,\mathbf{fr}_B}(G)$ are necessarily disjoint. It is now possible to show through straightforward but tedious case analysis that the lemma holds. □

A.2 Flip, Drop, and Copy

To prove soundness of the rules (Flip), (Drop), and (Copy) in theorem 4.6.3 we wish to rewrite sequences of !-box operations into a form only involving killing and expanding to show they are instantiations. The process for this is presented in theorem 4.1.12 and relies on pushing such unwanted operations left until they can be removed. In this section we prove the lemmas which allow this reordering.

Lemma A.2.1. *Suppose B is nested inside A in the !-tensor expression G and we are given the left hand side of the following equations. We claim there exist fresh renamings $\mathbf{fr}'_1, \ldots, \mathbf{fr}'_n$ such that these equalities hold:*

1. $\mathrm{Exp}_{A,\mathbf{fr}} \circ \mathrm{Flip}_B(G) = \mathrm{Flip}_{\mathbf{fr}(B)} \circ \mathrm{Flip}_B \circ \mathrm{Exp}_{A,\mathbf{fr}}(G)$

2. $\mathrm{Kill}_A \circ \mathrm{Flip}_B(G) = \mathrm{Kill}_A(G)$

3. $\mathrm{Kill}_B \circ \mathrm{Exp}_{B,\mathbf{fr}_n} \circ \ldots \circ \mathrm{Exp}_{B,\mathbf{fr}_1} \circ \mathrm{Flip}_B(G) = \mathrm{Kill}_B \circ \mathrm{Exp}_{B,\mathbf{fr}'_1} \circ \ldots \circ \mathrm{Exp}_{B,\mathbf{fr}'_n}(G)$

Proof.

1. Since flipping has no effect on edge or box names we know that $\mathbf{fr}$ is fresh for G and so the right hand side is well defined. It is then a straightforward structural induction proof to check the equality.

2. Any changes made by Flip_B happen inside the contents of A and hence are removed by Kill_A so that the equality is trivial.

3. Flipping has no effect on names and so the left hand side being well defined tells us that edges(G) and the images of each $\mathbf{fr}_i$ on $C(B)$ are disjoint. We define the operations $\mathbf{fr}'_i$ by:

$$\mathbf{fr}'_i(x) := \begin{cases} \mathbf{fr}_i(x) & x \in C(B) \\ \text{fresh} & \text{otherwise} \end{cases}$$

Each is fresh for its domain which contains the names in edges(G) and $\mathbf{fr}_j$ applied to the content of B for $j < i$.

It is then simple to check the equality as a structural induction proof.

□

Instances of Drop_B are easier to remove by a simple rewriting:

Lemma A.2.2. *Given Drop_B and G, there exist renamings $\mathbf{fr}$ and $\mathbf{rn}$ such that:*

$$\mathrm{Drop}_B(G) = \mathbf{rn} \circ \mathrm{Kill}_B \circ \mathrm{Exp}_{B,\mathbf{fr}}(G)$$

Proof. We let $\mathbf{fr}$ act on the contents of B by taking the name x to a fresh name x_1:

$$\mathbf{fr}(x) := \begin{cases} x_1 & x \in C(B) \\ \text{fresh} & \text{otherwise} \end{cases}$$

So the result of expanding and then killing has names x_1 where it should have names x. We correct this using the renaming $\mathbf{rn}$ defined by $x \leftrightarrow x_1$ for all $x \in C(B)$. The equation therefore follows. □

Lemma A.2.3. *Suppose B is nested inside A in G and we are given the left hand side of the following equations. We claim there exist renamings $\mathbf{fr}'_A, \mathbf{fr}'_B, \mathbf{fr}''_B$ such that the equalities hold:*

1. $\mathrm{Exp}_{A,\mathbf{fr}_A} \circ \mathrm{Copy}_{B,\mathbf{fr}_B}(G) = \mathrm{Copy}_{\mathbf{fr}'_A(B),\mathbf{fr}''_B} \circ \mathrm{Copy}_{B,\mathbf{fr}'_B} \circ \mathrm{Exp}_{A,\mathbf{fr}'_A}(G)$

2. $\mathrm{Kill}_A \circ \mathrm{Copy}_{B,\mathbf{fr}_B}(G) = \mathrm{Kill}_A(G)$

3. $\mathrm{Exp}_{\mathbf{fr}_B(B),\mathbf{fr}_A} \circ \mathrm{Copy}_{B,\mathbf{fr}_B}(G) = \mathrm{Copy}_{B,\mathbf{fr}'_B} \circ \mathrm{Exp}_{B,\mathbf{fr}'_A}(G)$

4. $\mathrm{Kill}_{\mathbf{fr}_B(B)} \circ \mathrm{Copy}_{B,\mathbf{fr}_B}(G) = G$

Proof. In each case it is straightforward but tedious to check the results on each side are equal !-tensors, up to renaming of some edges and !-boxes. This is sufficient to complete the proof of cases 2 and 4 as no new names are created. We now define appropriate renamings to ensure equality for cases 1 and 3.

1. To shift the copy operation after expansion on A, we need to copy both B and the fresh version created by expanding A. To ensure the names are correct, we need to be careful about the newly defined fresh renamings. We take $\mathbf{fr}'_A = \mathbf{fr}_A$, we also wish $\mathbf{fr}'_B$ to give the same results as $\mathbf{fr}_B$, though we need to be more careful and extend it to be fresh on new names created by $\mathrm{Exp}_{A,\mathbf{fr}'_A}$ on G:

$$\mathbf{fr}'_B(x) = \begin{cases} \mathbf{fr}_B(x) & x \in C(B) \\ \text{fresh} & \text{otherwise} \end{cases}$$

For our final renaming, note that the contents of $\mathbf{fr}'_A(B)$ have necessarily come from the operation $\mathrm{Exp}_{A,\mathbf{fr}'_A}$ and hence are of the form $\mathbf{fr}'_A(y)$ for some y. We can therefore define $\mathbf{fr}''_B$, which is only applied in $C(\mathbf{fr}'_A(B))$, in terms of such y:

$$\mathbf{fr}''_B(x) = \begin{cases} \mathbf{fr}_A(\mathbf{fr}_B(y)) & x = \mathbf{fr}'_A(y) \\ \text{fresh} & \text{otherwise} \end{cases}$$

Freshness of these functions comes from the fact that the images of $\mathbf{fr}_A$, $\mathbf{fr}_B$, and $\mathbf{fr}_A \circ \mathbf{fr}_B$ are disjoint on the contents of B.

3. We use $\mathbf{fr}'_A$ and $\mathbf{fr}'_B$ to assign fresh names to new edges and !-boxes by:

$$\mathbf{fr}'_A(x) = \mathbf{fr}_A(\mathbf{fr}_B(x)) \qquad \mathbf{fr}'_B(x) = \begin{cases} \mathbf{fr}_B(x) & x \in C(B) \\ \text{fresh} & \text{otherwise} \end{cases}$$

Freshness of these functions comes from the fact that the images of $\mathbf{fr}_A$ and $\mathbf{fr}_A \circ \mathbf{fr}_B$ are disjoint on the contents of B.

□

Appendix B

Induction on Arbitrary Input/Output

We now prove that the rule generalising induction for nodes with arbitrary arrangements of input and output edges can be derived from !L. The statement of the rule (as in theorem 6.1.1) is:

$$\frac{\Gamma \vdash \mathrm{Kill}_B(X) \qquad \Gamma, X \vdash \mathrm{Exp}^0_B(X) \qquad \Gamma, X \vdash \mathrm{Exp}^1_B(X)}{\Gamma \vdash X}\ (\text{Induct}^{\,0,1})$$

where B appears in the form $[[-]^{B_0}[-]^{B_1}]^B$, i.e. containing two !-boxes and their contents but nothing else directly inside B.

We will refer to the three premise sequents respectively as the **base** case, **step 0** case and **step 1** case.

Let us attempt induction on B appearing in the form $[[\ldots]^{B_0}[\ldots]^{B_1}]^B$ in a !-formula X. Induction on B will create new copies of B_0 and B_1 which will need separate inductions (we start with the copy of B_0). To save space and increase readability, we write X' for $\mathbf{fr}(X)$ and shorten Kill and Exp to K and E respectively. We also drop additional context (i.e. Γ in $\Gamma \vdash X$), this can trivially be added to the left of $\vdash$ in each of the following sequents, so long as the context does not contain the !-boxes we use ($\forall$I) on.

$$\dfrac{\dfrac{\textcircled{1}}{\vdash \mathrm{K}_B(X)} \qquad \dfrac{\dfrac{\dfrac{\dagger_1}{X \vdash \forall B_1'.\,\mathrm{K}_{B_0'}(\mathrm{E}_{B,\mathbf{fr}}(X))} \qquad \dfrac{\dagger_2}{X, \forall B_1'.\,\mathrm{E}_{B,\mathbf{fr}}(X) \vdash \forall B_1'.\,\mathrm{E}_{B_0'}(\mathrm{E}_{B,\mathbf{fr}}(X))}\ (\text{Induct})}{X \vdash \forall B_1'.\,\mathrm{E}_{B,\mathbf{fr}}(X)}}{X \vdash \forall B_0'.\forall B_1'.\,\mathrm{E}_{B,\mathbf{fr}}(X)}\ (\forall I)}{\vdash X}\ (\text{Induct})$$

Our first goal ① is precisely the **base** case. Then induction on B_1 reduces ($\dagger_1$) to another goal:

$$\dfrac{\dfrac{\dfrac{}{X \vdash X}\ (\text{Ident})}{X \vdash \mathrm{K}_{B_1'}\,\mathrm{K}_{B_0'}\,\mathrm{E}_{B,\mathbf{fr}}(X)}\ (\equiv) \qquad \dfrac{\textcircled{2}}{X, \mathrm{K}_{B_0'}\,\mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{E}_{B_1'}\,\mathrm{K}_{B_0'}\,\mathrm{E}_{B,\mathbf{fr}}(X)}}{X \vdash \mathrm{K}_{B_0'}\,\mathrm{E}_{B,\mathbf{fr}}(X)}\ (\text{Induct})}{\dagger_1}\ (\forall I)$$

where we have used the fact $\mathrm{K}_{B_1'}\,\mathrm{K}_{B_0'}\,\mathrm{E}_{B,\mathbf{fr}}(X) = X$ on the left hand side. Unfortunately our second goal (2) is not one of the step or base cases predicted above and similarly continuing ($\dagger_2$) leads to another two unexpected goals:

$$\dfrac{\dfrac{\dfrac{\textcircled{3}}{X, \forall B_1'.\, \mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{K}_{B_1'}\,\mathrm{E}_{B_0',\mathbf{fr}}(X)} \quad \dfrac{\textcircled{4}}{X, \forall B_1'.\, \mathrm{E}_{B,\mathbf{fr}}(X), \mathrm{E}_{B_0'}\,\mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{E}_{B_1'}\,\mathrm{E}_{B_0'}\,\mathrm{E}_{B,\mathbf{fr}}(X)}}{X, \forall B_1'.\, \mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{E}_{B_0'}\,\mathrm{E}_{B,\mathbf{fr}}(X)}\,(\forall I)}{\dagger_2}\,(\text{Induct})$$

We wish to show that (1) to (4) can be derived from the base and step cases above. This turns out to follow from a lemma which allows us to rewrite the above expressions in terms of Exp_B^0 and Exp_B^1.

Lemma B.0.1. *For a !-box B as above (made up of two further !-boxes B_0 and B_1 and nothing else) the following hold on a !-formula X:*

- *Given* $\mathrm{E}_{\mathbf{fr}(B_1),\mathbf{fr}_1}\,\mathrm{K}_{\mathbf{fr}(B_0)}\,\mathrm{E}_{B,\mathbf{fr}}$ *there exist fresh renamings* $\mathbf{fr}'$ *and* $\mathbf{fr}_1'$ *such that:*

$$\begin{aligned}\mathrm{E}_{\mathbf{fr}(B_1),\mathbf{fr}_1}\,\mathrm{K}_{\mathbf{fr}(B_0)}\,\mathrm{E}_{B,\mathbf{fr}} &= \mathrm{K}_{\mathbf{fr}(B_0)}\,\mathrm{E}_{B,\mathbf{fr}}\,\mathrm{K}_{\mathbf{fr}'(B_1)}\,\mathrm{E}_{\mathbf{fr}'(B_1),\mathbf{fr}_1'}\,\mathrm{K}_{\mathbf{fr}'(B_0)}\,\mathrm{E}_{B,\mathbf{fr}'}\\ &= \mathrm{K}_{\mathbf{fr}(B_0)}\,\mathrm{E}_{B,\mathbf{fr}}\,\mathrm{E}^1_{B,\mathbf{fr}_1\circ\mathbf{fr}}\end{aligned}$$

- *Given* $\mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{K}_{\mathbf{fr}(B_1)}\,\mathrm{E}_{B,\mathbf{fr}}$ *there exist fresh renamings* $\mathbf{fr}'$ *and* $\mathbf{fr}_0'$ *such that:*

$$\begin{aligned}\mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{K}_{\mathbf{fr}(B_1)}\,\mathrm{E}_{B,\mathbf{fr}} &= \mathrm{K}_{\mathbf{fr}(B_1)}\,\mathrm{E}_{B,\mathbf{fr}}\,\mathrm{K}_{\mathbf{fr}'(B_0)}\,\mathrm{E}_{\mathbf{fr}'(B_0),\mathbf{fr}_0'}\,\mathrm{K}_{\mathbf{fr}'(B_1)}\,\mathrm{E}_{B,\mathbf{fr}'}\\ &= \mathrm{K}_{\mathbf{fr}(B_1)}\,\mathrm{E}_{B,\mathbf{fr}}\,\mathrm{E}^0_{B,\mathbf{fr}_0\circ\mathbf{fr}}\end{aligned}$$

- *Given* $\mathrm{E}_{\mathbf{fr}(B_1),\mathbf{fr}_1}\,\mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{E}_{B,\mathbf{fr}}$ *there exist fresh renamings* $\mathbf{fr}'$ *and* $\mathbf{fr}_1'$ *such that:*

$$\begin{aligned}\mathrm{E}_{\mathbf{fr}(B_1),\mathbf{fr}_1}\,\mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{E}_{B,\mathbf{fr}} &= \mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{E}_{B,\mathbf{fr}}\,\mathrm{K}_{\mathbf{fr}'(B_1)}\,\mathrm{E}_{\mathbf{fr}'(B_1),\mathbf{fr}_1'}\,\mathrm{K}_{\mathbf{fr}'(B_0)}\,\mathrm{E}_{B,\mathbf{fr}'}\\ &= \mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{E}_{B,\mathbf{fr}}\,\mathrm{E}^1_{B,\mathbf{fr}_1\circ\mathbf{fr}}\end{aligned}$$

Proof. Each case follows by structural induction so long as we are careful about renaming. We take $\mathbf{fr}'$ to be a fresh renaming on the domain X which takes the name x to fresh name x_1. We then need to define $\mathbf{fr}_0'$ to be applied to $\mathrm{K}_{\mathbf{fr}'(B_1)}\,\mathrm{E}_{B,\mathbf{fr}'}(X)$ by an operation on the !-box $\mathbf{fr}'(B_0)$. Hence it is only important how it affects names in the contents of $\mathbf{fr}'(B_0)$ all of which are of the form x_1. Similarly, the definition of $\mathbf{fr}_1'$ is only important on names in the contents of $\mathbf{fr}'(B_1)$, also of the form x_1. We define:

$$\mathbf{fr}_0'(y) = \begin{cases}\mathbf{fr}_0(\mathbf{fr}(x)) & y = x_1\\ \text{fresh} & \text{otherwise}\end{cases} \qquad \mathbf{fr}_1'(y) = \begin{cases}\mathbf{fr}_1(\mathbf{fr}(x)) & y = x_1\\ \text{fresh} & \text{otherwise}\end{cases}$$

which guarantees that $\mathbf{fr}'_0 \circ \mathbf{fr}' = \mathbf{fr}_0 \circ \mathbf{fr}$ on the contents of B_0 and $\mathbf{fr}'_1 \circ \mathbf{fr}' = \mathbf{fr}_1 \circ \mathbf{fr}$ on the contents of B_1. The only thing left to note is that in each case above $\mathbf{fr}$ needs to be extended so that it is fresh for its new domain. This is not a problem as it will still only be applied to the contents of B and new names cannot clash with the extra names in the domain since they were fresh when added on the left hand side. □

Theorem B.0.2. *Let the !-formula X contain the subexpression $[[\ldots]^{B_0}[\ldots]^{B_1}]^B$ (so the only contents of B are B_0, B_1, and their contents). If B, B_0, and B_1 do not appear in Γ then we can apply the following induction rule:*

$$\frac{\Gamma \vdash \mathrm{Kill}_B(X) \qquad \Gamma, X \vdash \mathrm{Exp}^0_B(X) \qquad \Gamma, X \vdash \mathrm{Exp}^1_B(X)}{\Gamma \vdash X}\ (\mathit{Induct}^{0,1})$$

Proof. Again we now drop the context Γ, though it can trivially be added to the left of every $\vdash$. We need only show that ① to ④ follow from the premises. The first is trivial, we show the others:

$$\cfrac{\cfrac{\cfrac{X \vdash \mathrm{E}^0_{B,\mathbf{fr}_0 \circ \mathbf{fr}}(X)}{\mathrm{K}_{\mathbf{fr}(B_1)}\,\mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{K}_{\mathbf{fr}(B_1)}\,\mathrm{E}_{B,\mathbf{fr}}\,\mathrm{E}^0_{B,\mathbf{fr}_0 \circ \mathbf{fr}}(X)}\,(\equiv)}{\mathrm{K}_{\mathbf{fr}(B_1)}\,\mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{K}_{\mathbf{fr}(B_1)}\,\mathrm{E}_{B,\mathbf{fr}}(X)}\,(\text{Weaken})}{②}\,(\text{Partial})$$

$$\cfrac{\cfrac{\cfrac{X \vdash \mathrm{E}^1_{B,\mathbf{fr}_1 \circ \mathbf{fr}}(X)}{\mathrm{K}_{\mathbf{fr}(B_0)}\,\mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{K}_{\mathbf{fr}(B_0)}\,\mathrm{E}_{B,\mathbf{fr}}\,\mathrm{E}^1_{B,\mathbf{fr}_1 \circ \mathbf{fr}}(X)}\,(\equiv)}{\mathrm{K}_{\mathbf{fr}(B_0)}\,\mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{E}_{\mathbf{fr}(B_1),\mathbf{fr}_1}\,\mathrm{K}_{\mathbf{fr}(B_0)}\,\mathrm{E}_{B,\mathbf{fr}}(X)}\,(\text{Weaken})}{③}\,(\text{Partial})$$

$$\cfrac{\cfrac{\cfrac{X \vdash \mathrm{E}^1_{B,\mathbf{fr}_1 \circ \mathbf{fr}}(X)}{\mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{E}_{B,\mathbf{fr}}\,\mathrm{E}^1_{B,\mathbf{fr}_1 \circ \mathbf{fr}}(X)}\,(\equiv)}{\mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{E}_{B,\mathbf{fr}}(X) \vdash \mathrm{E}_{\mathbf{fr}(B_1),\mathbf{fr}_1}\,\mathrm{E}_{\mathbf{fr}(B_0),\mathbf{fr}_0}\,\mathrm{E}_{B,\mathbf{fr}}(X)}\,(\text{Weaken})}{④}\,(\text{Partial})$$

□

Bibliography

[1] S. Abramsky and B. Coecke. A categorical semantics of quantum protocols. In *Proceedings from LiCS*, arXiv:quant-ph/0402130v5, 2004.

[2] J. C. Baez and J. Erbele. Categories in control. Technical report, arXiv:1405.6881, 2014.

[3] F. Bonchi, P. Sobocinski, and F. Zanasi. A categorical semantics of signal flow graphs. In *CONCUR'14: Concurrency Theory.*, volume 8704 of *Lecture Notes in Computer Science*, pages 435–450. Springer, 2014.

[4] B. Coecke. Quantum picturalism. *Contemporary Physics*, 51:59–83, 2009. arXiv:0908.1787.

[5] B. Coecke and R. Duncan. Interacting quantum observables. In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, 2008.

[6] B. Coecke, R. Duncan, A. Kissinger, and Q. Wang. Strong complementarity and non-locality in categorical quantum mechanics. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society, 2012. arXiv:1203.4988.

[7] B. Coecke and D. Pavlovic. Quantum measurements without sums. *eprint arXiv:quant-ph/0608035*, August 2006.

[8] B. Coecke and R. Duncan. Interacting Quantum Observables: Categorical Algebra and Diagrammatics. arXiv:0906.4725v1 [quant-ph], 2009.

[9] B. Coecke and A. Kissinger. The Compositional Structure of Multipartite Quantum Entanglement. In *Automata, Languages and Programming*, volume 6199 of *Lecture Notes in Computer Science*, pages 297–308. Springer, 2010.

[10] B. Coecke, A. Kissinger, A. Merry, and S. Roy. The GHZ/W-calculus contains rational arithmetic. arXiv:1103.2812 [cs.LO], 2011.

[11] D. de Jongh and Z. Zhao. Positive formulas in intuitionistic and minimal logic. In *Logic, Language, and Computation*, pages 175–189. Springer, 2013.

[12] L. Dixon, R. Duncan, and A. Kissinger. Open Graphs and Computational Reasoning. In *Proceedings of DCM'10*, volume 26, pages 169–180. EPTCS, 2010.

[13] L. Dixon and R. Duncan. Extending Graphical Representations for Compact Closed Categories with Applications to Symbolic Quantum Computation. *AISC/MKM/Calculemus*, pages 77–92, 2008.

[14] L. Dixon and R. Duncan. Graphical Reasoning in Compact Closed Categories for Quantum Computation. *AMAI*, 56(1):20, 2009.

[15] L. Dixon and A. Kissinger. Open-graphs and monoidal theories. *Mathematical Structures in Computer Science*, 23:308–359, 4 2013. arXiv:1007.3794v1 [cs.LO].

[16] R. Duncan and S. Perdrix. Graph States and the necessity of Euler Decomposition. In K. Ambos-Spies, B. Löwe, and W. Merkle, editors, *Computability in Europe: Mathematical Theory and Computational Practice (CiE'09)*, volume 5635 of *Lecture Notes in Computer Science*, pages 167–177. Springer, 2009.

[17] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer Berlin Heidelberg, 1999.

[18] D. Hilbert, W. Ackermann, and R. E. Luce. *Principles of mathematical logic*, volume 69. American Mathematical Soc., 1950.

[19] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[20] A. Joyal and R. Street. The geometry of tensor calculus I. *Advances in Mathematics*, 88:55–113, 1991.

[21] A. Joyal, R. Street, and D. Verity. Traced Monoidal Categories. *Math. Proc. Camb. Phil. Soc.*, 119(3):447–468, 1996.

[22] D. Kaiser. Physics and Feynman's Diagrams. *American Scientist*, 93:156–165, 2005.

[23] D. Kartsaklis. *Compositional Distributional Semantics with Compact Closed Categories and Frobenius Algebras*. PhD thesis, University of Oxford, 2014.

[24] A. Kissinger. *Pictures of Processes: Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Computing*. PhD thesis, University of Oxford, 2011. arXiv:1203.0202 [math.CT].

[25] A. Kissinger. Abstract tensor systems as monoidal categories. In C. Casadio, B. Coecke, M. Moortgat, and P. Scott, editors, *Categories and Types in Logic, Language, and Physics: Festschrift on the occasion of Jim Lambek's 90th birthday*, volume 8222 of *Lecture Notes in Computer Science*. Springer, 2014. arXiv:1308.3586 [math.CT].

[26] A. Kissinger, A. Merry, and M. Soloviev. Pattern graph rewrite systems. In *Proceedings of DCM 2012*, volume 143 of *EPTCS*, 2012. arXiv:1204.6695 [math.CT].

[27] A. Kissinger, A. Merry, L. Dixon, R. Duncan, M. Soloviev, and B. Frot. Quantomatic. https://sites.google.com/site/quantomatic/, 2011.

[28] A. Kissinger and D. Quick. Tensors, !-graphs, and non-commutative quantum structures. In *Proceedings of the 11th workshop on Quantum Physics and Logic, QPL 2014, Kyoto, Japan, 4-6th June 2014.*, pages 56–67, 2014. arXiv:1412.8552 [cs.LO].

[29] A. Kissinger and D. Quick. A first-order logic for string diagrams. In *Proceedings of CALCO*, 2015. arXiv:1505.00343 [cs.LO].

[30] A. Kissinger and D. Quick. Tensors, !-graphs, and non-commutative quantum structures (extended version), 2015. arXiv:1503.01348.

[31] A. Kissinger and V. Zamdzhiev. !-graphs with trivial overlap are context-free. 2015. arXiv:1501.06059.

[32] A. Kissinger and V. Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning, 2015. arXiv:1503.01034.

[33] S. Lack. Composing PROPs. *Theory and Applications of Categories*, 13(9):147–163, 2004.

[34] S. Lack and P. Sobocinski. Adhesive Categories. *Basic Research in Computer Science*, pages 1–28, 2003.

[35] S. Lack and P. Sobocinski. Adhesive and quasiadhesive categories. *Theoretical Informatics and Applications*, 39(2):522–546, 2005.

[36] A. Lang and B. Coecke. Trichromatic Open Digraphs for Understanding Qubits. *ArXiv e-prints*, October 2011.

[37] A. D. Lauda and H. Pfeiffer. Open-closed strings: Two-dimensional extended tqfts and frobenius algebras. *Topology Appl.*, 155(7):623–666, 2008.

[38] S. Mac Lane. *Categories for the working mathematician.* Springer Verlag, 1998.

[39] A. Merry. *Reasoning with !-Graphs.* PhD thesis, University of Oxford, 2014.

[40] R. Penrose. Applications of negative dimensional tensors. In *Combinatorial Mathematics and its Applications*, pages 221–244. Academic Press, 1971.

[41] R. Penrose and W. Rindler. *Spinors and Space-Time*, volume 1. Cambridge University Press, 1984. Cambridge Books Online.

[42] U. Prange, H. Ehrig, and L. Lambers. Construction and properties of adhesive and weak adhesive high-level replacement categories. *Applied Categorical Structures*, 16(3):365–388, 2008.

[43] D. Quick. Encoding !-tensors as !-graphs with neighbourhood orders. In *Proceedings of the 12th workshop on Quantum Physics and Logic, QPL 2015, Oxford, England, 13-17th July 2015.*, 2015.

[44] P. Selinger. A survey of graphical languages for monoidal categories. In *New structures for physics*, pages 289–355. Springer, 2011.