New reasoning techniques for monoidal algebra

Aleks Kissinger

November 4, 2015

Quantum Group
Department of Computer Science
University of Oxford

Algebra and rewriting

Algebra and rewriting

- Consider a monoid $(A, \cdot, e)$:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{and} \qquad a \cdot e = a = e \cdot a$$

Algebra and rewriting

- Consider a monoid $(A, \cdot, e)$:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{and} \qquad a \cdot e = a = e \cdot a$$

- Normally, mathematical tools, e.g. automated theorem provers would use these equations as rewrite rules:

$$(a \cdot b) \cdot c \longrightarrow\!\!\triangleright\; a \cdot (b \cdot c) \qquad a \cdot e \longrightarrow\!\!\triangleright\; a \qquad e \cdot a \longrightarrow\!\!\triangleright\; a$$

Algebra and rewriting

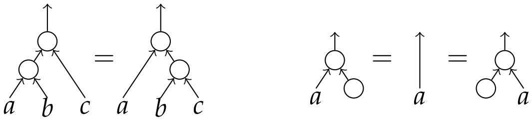
- Consider a monoid $(A, \cdot, e)$:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{and} \qquad a \cdot e = a = e \cdot a$$

- Normally, mathematical tools, e.g. automated theorem provers would use these equations as rewrite rules:

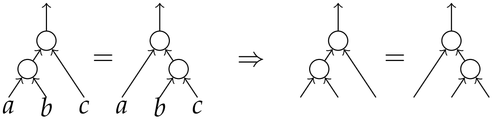
$$(a \cdot b) \cdot c \rhd a \cdot (b \cdot c) \qquad a \cdot e \rhd a \qquad e \cdot a \rhd a$$

- It is also possible to write these equations as trees:



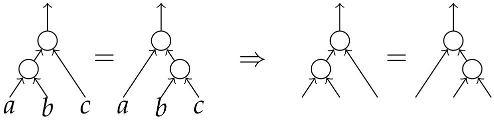
Algebra and rewriting

- Since these equations are (left- and right-) linear in the free variables, we can drop them:



Algebra and rewriting

- Since these equations are (left- and right-) linear in the free variables, we can drop them:



- The role of variables is replaced by the notion that the LHS and RHS have a *shared boundary*

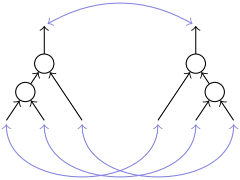


Diagram substitution

- One could apply the rule "$(a \cdot b) \cdot c \to a \cdot (b \cdot c)$" using the usual "instantiate, match, replace" style:

$$w \cdot ((x \cdot (y \cdot e)) \cdot z) \quad \longrightarrow \quad w \cdot (x \cdot ((y \cdot e) \cdot z))$$

Diagram substitution

- One could apply the rule "$(a \cdot b) \cdot c \to a \cdot (b \cdot c)$" using the usual "instantiate, match, replace" style:

$$w \cdot ((x \cdot (y \cdot e)) \cdot z) \quad \longrightarrow \quad w \cdot (x \cdot ((y \cdot e) \cdot z))$$

- ...or by cutting the LHS directly out of the tree and gluing in the RHS:

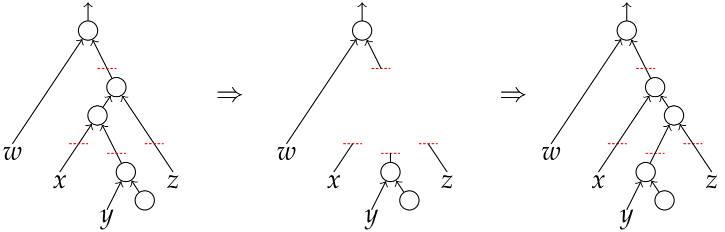
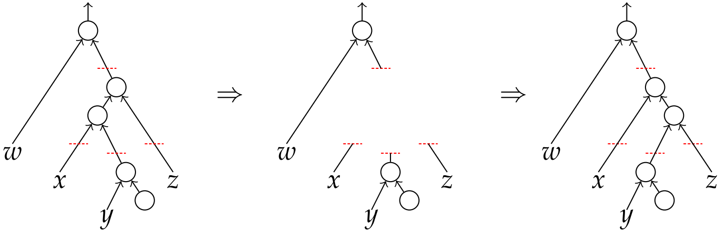


Diagram substitution

- One could apply the rule "$(a \cdot b) \cdot c \to a \cdot (b \cdot c)$" using the usual "instantiate, match, replace" style:

$$w \cdot ((x \cdot (y \cdot e)) \cdot z) \quad \longrightarrow \quad w \cdot (x \cdot ((y \cdot e) \cdot z))$$

- ...or by cutting the LHS directly out of the tree and gluing in the RHS:



- This treats inputs and outputs symmetrically

Algebra and coalgebra

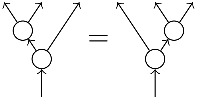
- We can consider structures with many *outputs* as well as inputs.

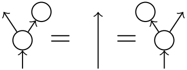
Algebra and coalgebra

- We can consider structures with many *outputs* as well as inputs.
- *Coalgebraic structures*: algebraic structures “upside-down”

Algebra and coalgebra

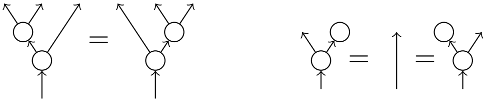
- We can consider structures with many *outputs* as well as inputs.
- *Coalgebraic structures*: algebraic structures “upside-down”
- E.g. *comonoids*, which consist of a *comultiplication* operation  and a *counit*  satisfying:



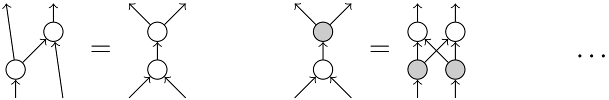


Algebra and coalgebra

- We can consider structures with many *outputs* as well as inputs.
- *Coalgebraic structures*: algebraic structures "upside-down"
- E.g. *comonoids*, which consist of a *comultiplication* operation  and a *counit*  satisfying:

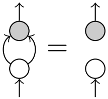


- Algebra and coalgebra can interact in many interesting ways:



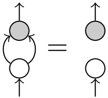
Equational reasoning with diagram substitution

- As before, we can use graphical identities to perform substitutions, but on graphs, rather than trees

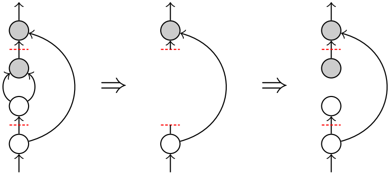


Equational reasoning with diagram substitution

- As before, we can use graphical identities to perform substitutions, but on graphs, rather than trees

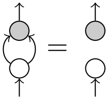


- For example:

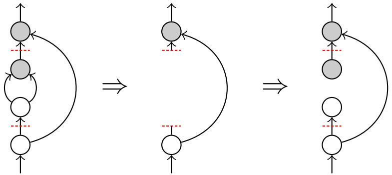


Equational reasoning with diagram substitution

- As before, we can use graphical identities to perform substitutions, but on graphs, rather than trees



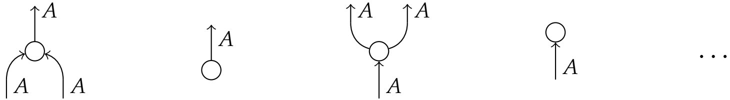
- For example:



- This style of rewriting works for any (co)algebraic structure in a *monoidal category*, a.k.a. *monoidal algebras*.

Algebraic structures in SMCs

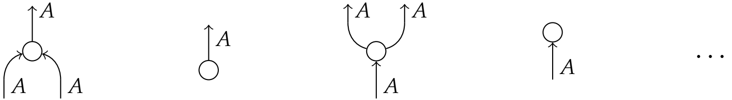
- A (single-sorted) monoidal algebra $\mathcal{A}$ consists of an object A and a set of morphisms whose inputs/outputs have type A:



called the *generators* of $\mathcal{A}$,

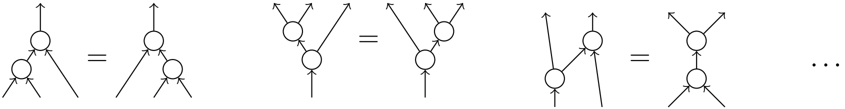
Algebraic structures in SMCs

- A (single-sorted) monoidal algebra $\mathcal{A}$ consists of an object A and a set of morphisms whose inputs/outputs have type A:

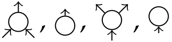
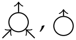


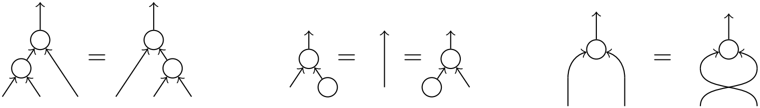
called the *generators* of $\mathcal{A}$,

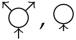
- and some equations:

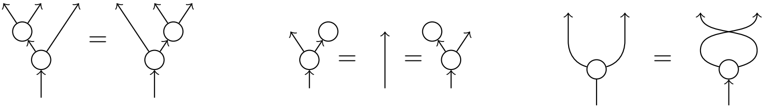


Example: Frobenius algebras

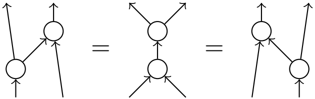
- A *commutative Frobenius algebra* consists of a tuple (A, ) such that:
  - (A, ) forms a commutative monoid:



  - (A, ) forms a commutative comonoid:

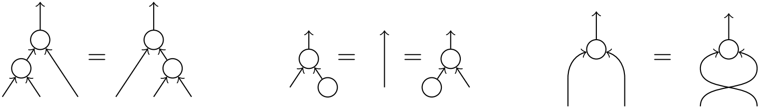


  - The *Frobenius law* is satisfied:

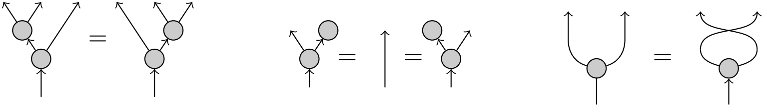


Example: Bialgebras

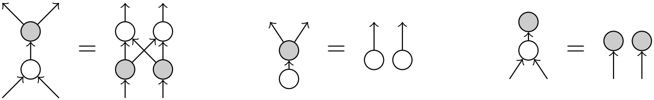
- A *(bi)commutative bialgebra* consists of a tuple $(A,$ , , , $)$ such that:
  - $(A,$ , $)$ forms a monoid:



  - $(A,$ , $)$ forms a comonoid:



  - The *bialgebra laws* are satisfied:



PROPs

- Monoidal algebras can also be defined via *functorial semantics*:

PROPs

- Monoidal algebras can also be defined via *functorial semantics*:
  1. Define a theory category $\mathbb{T}$ whose objects are natural numbers (i.e. arities) and:
  $$m \otimes n := m + n$$
  For SMCs, this is called a **PRO**duct category with **P**ermutations (PROP).

PROPs

- Monoidal algebras can also be defined via *functorial semantics*:
  1. Define a theory category $\mathbb{T}$ whose objects are natural numbers (i.e. arities) and:
  $$m \otimes n := m + n$$
  For SMCs, this is called a **PRO**duct category with **P**ermutations (PROP).
  2. Fix another SMC $\mathcal{C}$ (e.g. functions, relations, linear maps, etc.).

PROPs

- Monoidal algebras can also be defined via *functorial semantics*:
  1. Define a theory category $\mathbb{T}$ whose objects are natural numbers (i.e. arities) and:
  $$m \otimes n := m + n$$
  For SMCs, this is called a **PRO**duct category with **P**ermutations (PROP).
  2. Fix another SMC $\mathcal{C}$ (e.g. functions, relations, linear maps, etc.).
  3. $\mathbb{T}$-algebras in $\mathcal{C}$ are then symmetric monoidal functors:
  $$[\![-]\!] : \mathbb{T} \to \mathcal{C}$$

PROPs

- Monoidal algebras can also be defined via *functorial semantics*:
  1. Define a theory category $\mathbb{T}$ whose objects are natural numbers (i.e. arities) and:
  $$m \otimes n := m + n$$
  For SMCs, this is called a **PRO**duct category with **P**ermutations (PROP).
  2. Fix another SMC $\mathcal{C}$ (e.g. functions, relations, linear maps, etc.).
  3. $\mathbb{T}$-algebras in $\mathcal{C}$ are then symmetric monoidal functors:
  $$[\![-]\!] : \mathbb{T} \to \mathcal{C}$$
- PROPs come in two flavours:

PROPs

- Monoidal algebras can also be defined via *functorial semantics*:
  1. Define a theory category $\mathbb{T}$ whose objects are natural numbers (i.e. arities) and:

     $$m \otimes n := m + n$$

     For SMCs, this is called a **PRO**duct category with **P**ermutations (PROP).
  2. Fix another SMC $\mathcal{C}$ (e.g. functions, relations, linear maps, etc.).
  3. $\mathbb{T}$-algebras in $\mathcal{C}$ are then symmetric monoidal functors:

     $$[\![-]\!] : \mathbb{T} \to \mathcal{C}$$

- PROPs come in two flavours:
  1. *Syntactic* PROPs have as morphisms diagrams of generators, modulo some set of diagram equations. Deciding equality $\Leftrightarrow$ solving a word problem.

PROPs

- Monoidal algebras can also be defined via *functorial semantics*:
  1. Define a theory category $\mathbb{T}$ whose objects are natural numbers (i.e. arities) and:
  $$m \otimes n := m + n$$
  For SMCs, this is called a **PRO**duct category with **P**ermutations (PROP).
  2. Fix another SMC $\mathcal{C}$ (e.g. functions, relations, linear maps, etc.).
  3. $\mathbb{T}$-algebras in $\mathcal{C}$ are then symmetric monoidal functors:
  $$[\![-]\!] : \mathbb{T} \to \mathcal{C}$$

- PROPs come in two flavours:
  1. *Syntactic* PROPs have as morphisms diagrams of generators, modulo some set of diagram equations. Deciding equality $\Leftrightarrow$ solving a word problem.
  2. *Semantic* PROPs have morphisms with a concrete description (functions, relations, finite matrices, etc.). Equality is usually (easily) decidable.

Example: Commutative monoids are functions

- Let $\mathbb{F}$ be the PROP whose morphisms $f : m \to n$ are functions between finite sets:

$$f : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\}$$

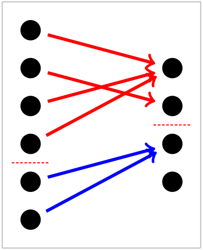
Example: Commutative monoids are functions

- Let $\mathbb{F}$ be the PROP whose morphisms $f : m \to n$ are functions between finite sets:

$$f : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\}$$

- $f \otimes g : m + m' \to n + n'$ is given by disjoint union of functions:

$$(f \otimes g)(i) = \begin{cases} f(i) & \text{if } i < m \\ g(i-m)+n & \text{if } i \geq m \end{cases} \quad \rightsquigarrow$$



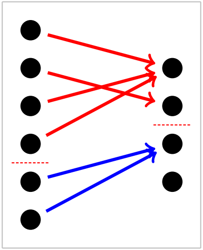
Example: Commutative monoids are functions

- Let $\mathbb{F}$ be the PROP whose morphisms $f : m \to n$ are functions between finite sets:

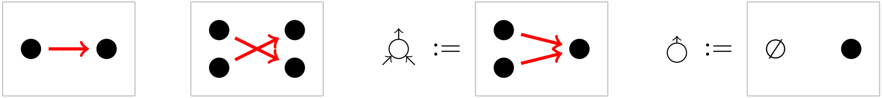
$$f : \{0, \ldots, m-1\} \to \{0, \ldots, n-1\}$$

- $f \otimes g : m + m' \to n + n'$ is given by disjoint union of functions:

$$(f \otimes g)(i) = \begin{cases} f(i) & \text{if } i < m \\ g(i-m) + n & \text{if } i \geq m \end{cases} \quad \rightsquigarrow$$

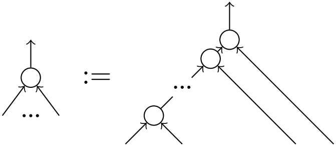


- This whole category is generated by identities, swaps, and a single commutative monoid:



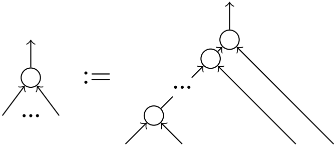
Example: Commutative monoids are functions

- Pretty easy to see, just consider n-ary trees of  :

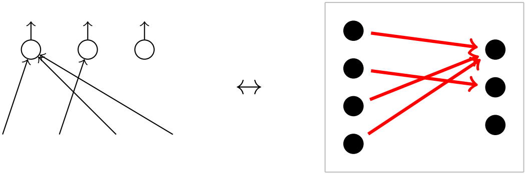


Example: Commutative monoids are functions

- Pretty easy to see, just consider n-ary trees of  :

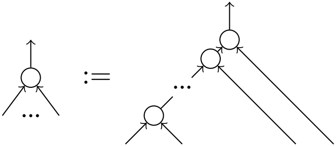


- Then, any diagram of  and  can be put in normal form, and those normal forms are classified by functions:

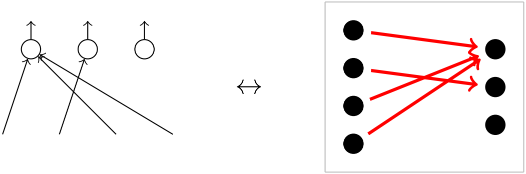


Example: Commutative monoids are functions

- Pretty easy to see, just consider n-ary trees of  :



- Then, any diagram of  and  can be put in normal form, and those normal forms are classified by functions:



- Similarly, $\mathbb{F}^{\mathrm{op}}$ is the PROP for cocommutative comonoids.

Distributive laws

- What happens when we combine two monoidal algebras, e.g. ( , ) and ( , )?

Distributive laws

- What happens when we combine two monoidal algebras, e.g. ( , ) and ( , )?
- ...not much!

Distributive laws

- What happens when we combine two monoidal algebras, e.g. ( , ) and ( , )?
- ...not much! Until we add a distributive law.

Distributive laws

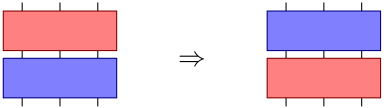
- What happens when we combine two monoidal algebras, e.g. ( , ) and ( , )?
- ...not much! Until we add a distributive law.
- This is a distributive law of monads in the bicategory of monoids in spans of categories

Distributive laws

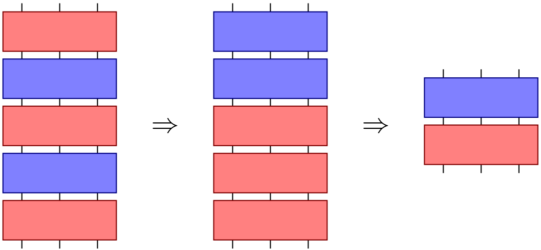
- What happens when we combine two monoidal algebras, e.g. ( , ) and ( , )?
- ...not much! Until we add a distributive law.
- This is a distributive law of monads in the bicategory of monoids in spans of categories ...or something like that...

Distributive laws

- More concretely, give us the means to move two pieces of structure past each other:

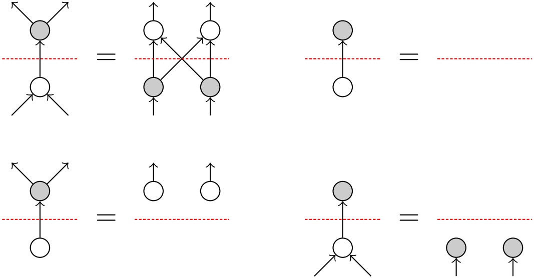


- So, normal forms for each of the individual theories become normal forms for the composed theory:



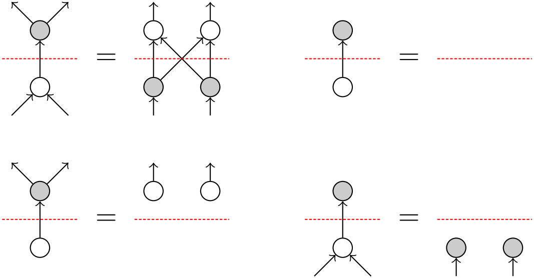
Example: Bialgebras are matrices

- Bialgebras consist of a monoid ( , ), a comonoid ( , ), and a distributive law:

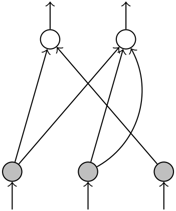


Example: Bialgebras are matrices

- Bialgebras consist of a monoid ( , ), a comonoid ( , ), and a distributive law:

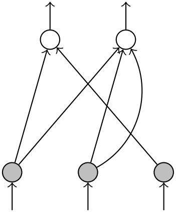


- So, normal forms look like this:



Example: Bialgebras are matrices

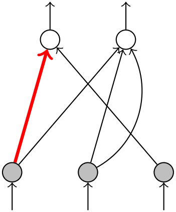
- These are classified by matrices over $\mathbb{N}$:



$$\leftrightarrow \quad \begin{pmatrix} 1 & 0 & 1 \\ 1 & 2 & 0 \end{pmatrix}$$

Example: Bialgebras are matrices

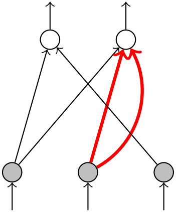
- These are classified by matrices over $\mathbb{N}$:



$$\leftrightarrow \begin{pmatrix} \color{red}{1} & 0 & 1 \\ 1 & 2 & 0 \end{pmatrix}$$

Example: Bialgebras are matrices

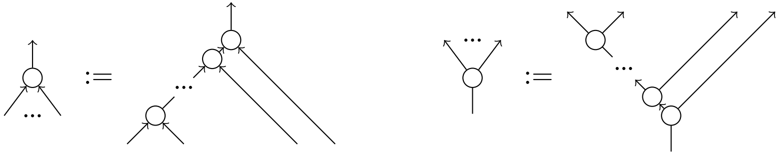
- These are classified by matrices over $\mathbb{N}$:



$$\leftrightarrow \quad \begin{pmatrix} 1 & 0 & 1 \\ 1 & \mathbf{2} & 0 \end{pmatrix}$$

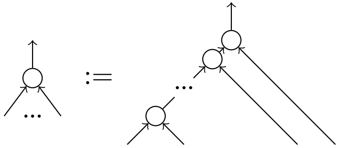
Diagrams with repetition

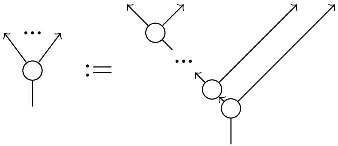
- Many of these theorems have something in common: the deal with repreated structures, like **trees** and **cotrees**:



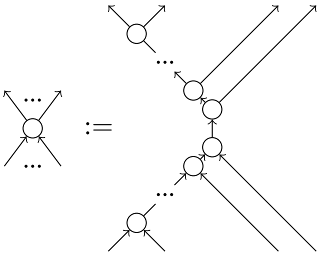
Diagrams with repetition

- Many of these theorems have something in common: the deal with repreated structures, like **trees** and **cotrees**:





- ...and tree/cotrees, a.k.a. **spiders**:

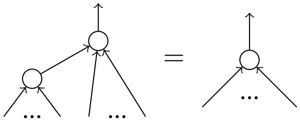


Diagrams with repetition

- Individual rules can by *meta-rules*

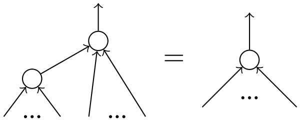
Diagrams with repetition

- Individual rules can by *meta-rules*
- For example, the rules of commutative monoids can be all be expressed by letting trees fuse:

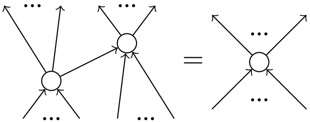


Diagrams with repetition

- Individual rules can by *meta-rules*
- For example, the rules of commutative monoids can be all be expressed by letting trees fuse:



- Similarly, the rules of commutative Frobenius algebras are expressed by letting spiders fuse:

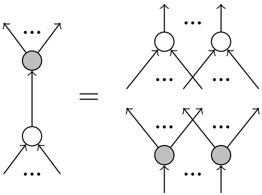


Diagrams with repetition

- Others are harder to say. For instance, bialgebras have several meta-rules.

Diagrams with repetition

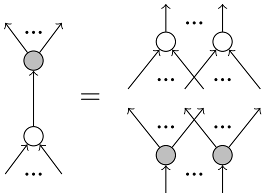
- Others are harder to say. For instance, bialgebras have several meta-rules.
- The most general is the path counting rule, but this has some intriguing consequences, e.g.:



where the RHS is a connected bipartite graph.

Diagrams with repetition

- Others are harder to say. For instance, bialgebras have several meta-rules.
- The most general is the path counting rule, but this has some intriguing consequences, e.g.:



where the RHS is a connected bipartite graph.

- These three examples have something in common: they rely on your brain, and some “blah blah” to fill in the “$\cdots$”

Diagrammatic meta-language

- Can we develop a meta-language for diagrams which is...

Diagrammatic meta-language

- Can we develop a meta-language for diagrams which is...
  - **easy** enough to use by hand,

Diagrammatic meta-language

- Can we develop a meta-language for diagrams which is...
    - **easy** enough to use by hand,
    - **expressive** enough to talk about lots of different kinds of families of diagrams,

Diagrammatic meta-language

- Can we develop a meta-language for diagrams which is...
  - **easy** enough to use by hand,
  - **expressive** enough to talk about lots of different kinds of families of diagrams,
  - **formal** enough to produce machine-checkable proofs,

Diagrammatic meta-language

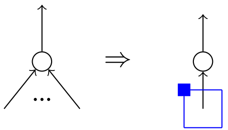
- Can we develop a meta-language for diagrams which is...
    - **easy** enough to use by hand,
    - **expressive** enough to talk about lots of different kinds of families of diagrams,
    - **formal** enough to produce machine-checkable proofs,
    - and comes with a **bag of tricks** for building those proofs?

Diagrammatic meta-language

- Can we develop a meta-language for diagrams which is...
  - **easy** enough to use by hand,
  - **expressive** enough to talk about lots of different kinds of families of diagrams,
  - **formal** enough to produce machine-checkable proofs,
  - and comes with a **bag of tricks** for building those proofs?
- One answer is the *!-box langauge*

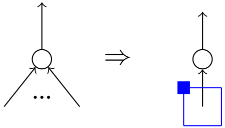
!-boxes

- We can formalise families of diagrams (with variable-arity generators) using some graphical syntax:

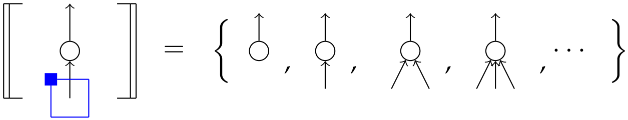


!-boxes

- We can formalise families of diagrams (with variable-arity generators) using some graphical syntax:

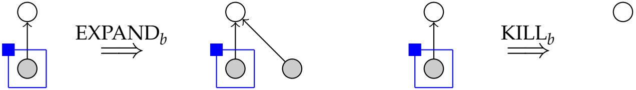


- The blue boxes are called !-boxes. A graph with !-boxes is called a !-graph. Can be interpreted as a set of concrete graphs:



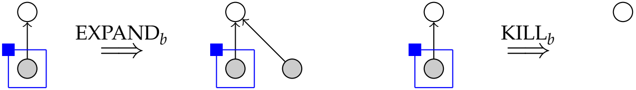
!-boxes

- The diagrams represented by a !-graph are all those obtained by performing EXPAND and KILL operations on !-boxes

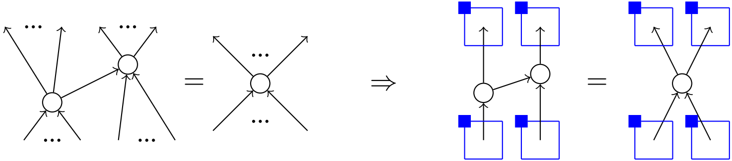


!-boxes

- The diagrams represented by a !-graph are all those obtained by performing EXPAND and KILL operations on !-boxes

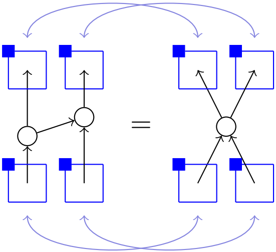


- We can also introduce equations involving !-boxes:



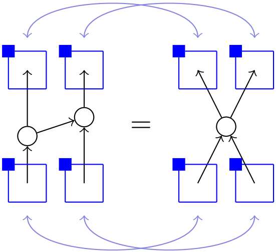
!-boxes: matching

- !-boxes on the LHS are in 1-to-1 correspondence with RHS



!-boxes: matching

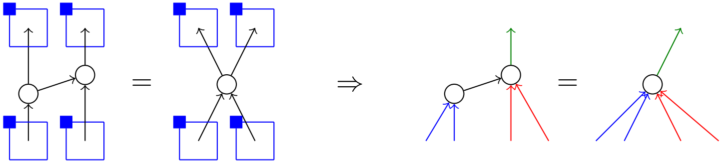
- !-boxes on the LHS are in 1-to-1 correspondence with RHS



- EXPAND and KILL operations applied to both sides simultaneously to instantiate a rule.

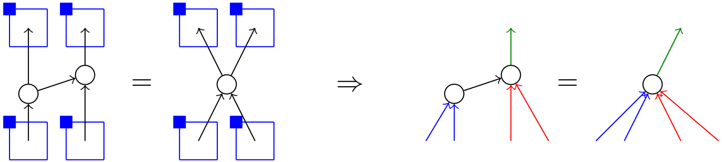
!-graph to concrete graph rewriting

- Rewriting concrete diagrams: find an instantiation of the rule such that the LHS matches the diagram:

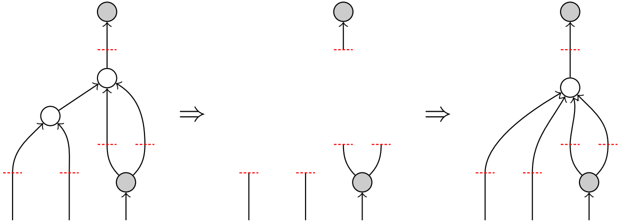


!-graph to concrete graph rewriting

- Rewriting concrete diagrams: find an instantiation of the rule such that the LHS matches the diagram:

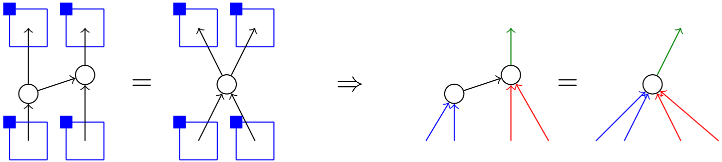


- Then apply it as usual:

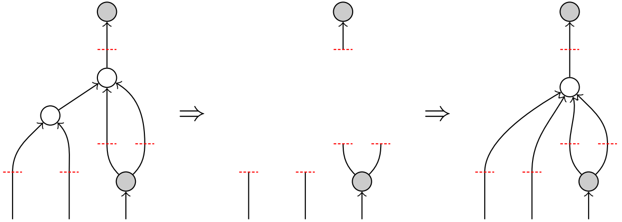


!-graph to concrete graph rewriting

- Rewriting concrete diagrams: find an instantiation of the rule such that the LHS matches the diagram:



- Then apply it as usual:



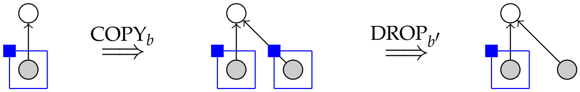
- Sound and complete, in the absence of “wild” !-boxes

!-graph to !-graph rewriting

- The real power comes from applying !-box rewrite rules on !-graphs themselves.

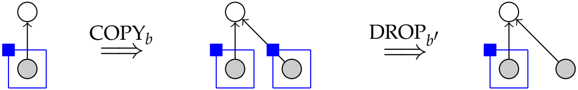
!-graph to !-graph rewriting

- The real power comes from applying !-box rewrite rules on !-graphs themselves.
- To define a more powerful notion of instantiation, we decompose EXPAND as two new operations:



!-graph to !-graph rewriting

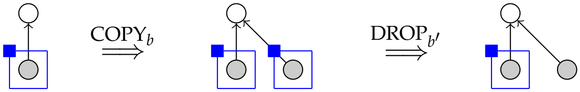
- The real power comes from applying !-box rewrite rules on !-graphs themselves.
- To define a more powerful notion of instantiation, we decompose EXPAND as two new operations:



- These operations are sound w.r.t. concrete instantiation, i.e. they don't produce any new concrete instances.

!-graph to !-graph rewriting

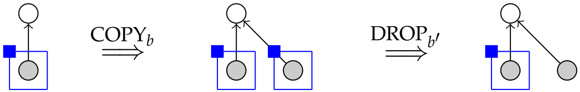
- The real power comes from applying !-box rewrite rules on !-graphs themselves.
- To define a more powerful notion of instantiation, we decompose EXPAND as two new operations:



- These operations are sound w.r.t. concrete instantiation, i.e. they don't produce any new concrete instances.
- Now, rewriting !-graphs is just the same as rewriting concrete graphs, with one extra restriction:

!-graph to !-graph rewriting

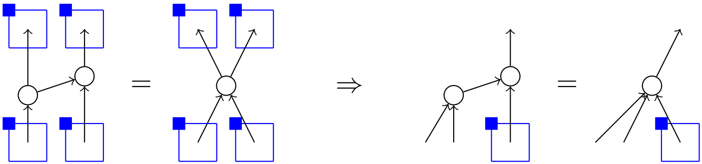
- The real power comes from applying !-box rewrite rules on !-graphs themselves.
- To define a more powerful notion of instantiation, we decompose EXPAND as two new operations:



- These operations are sound w.r.t. concrete instantiation, i.e. they don't produce any new concrete instances.
- Now, rewriting !-graphs is just the same as rewriting concrete graphs, with one extra restriction:
- If any part of an edge is in a !-box, **we must cut through it.**

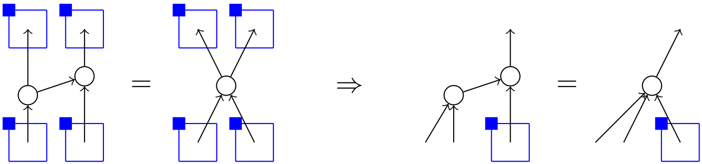
!-graph to !-graph rewriting

- !-graph rewriting: first instantiate:

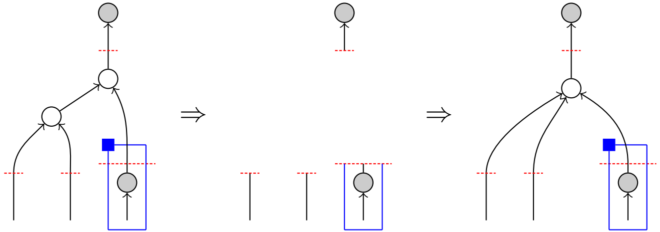


!-graph to !-graph rewriting

- !-graph rewriting: first instantiate:

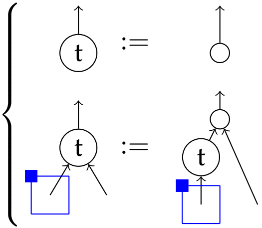


- Then apply:



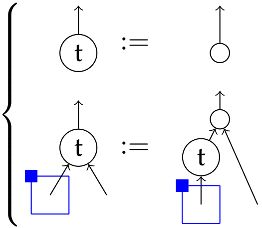
Recursive definition

- Once we have !-boxes around, we can make recursive definitions:



Recursive definition

- Once we have !-boxes around, we can make recursive definitions:



- And, as usual, recursive definition goes hand-in-hand with inductive proof...

Induction principle for !-graphs

- Let $\mathrm{FIX}_b(G = H)$ be the same as $G = H$, but !-box b cannot be expanded

Induction principle for !-graphs

- Let $\text{FIX}_b(G = H)$ be the same as $G = H$, but !-box b cannot be expanded
- Using FIX, we can define induction

$$\frac{\text{KILL}_b(G = H) \qquad \text{FIX}_b(G = H) \implies \text{EXPAND}_b(G = H)}{G = H}\,ind$$

Induction principle for !-graphs

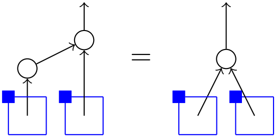
- Let $\text{FIX}_b(G = H)$ be the same as $G = H$, but !-box b cannot be expanded
- Using FIX, we can define induction

$$\frac{\text{KILL}_b(G = H) \qquad \text{FIX}_b(G = H) \implies \text{EXPAND}_b(G = H)}{G = H}\, ind$$

- By (normal) induction over proofs involving concrete graphs, we can prove admissibility.

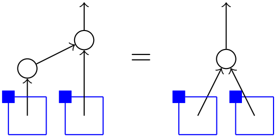
Induction principle for !-graphs

- Using !-box induction, we can now prove standard things like:

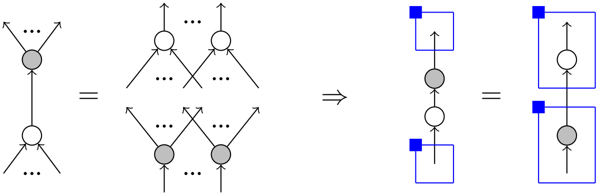


Induction principle for !-graphs

- Using !-box induction, we can now prove standard things like:

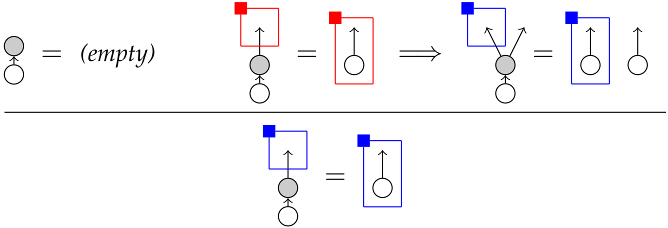


- But this just looks like something in term-land. We can actually prove much more interesting things like:



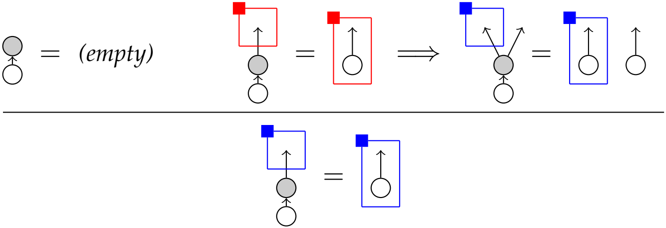
Induction example

- First apply induction to get two sub-goals:

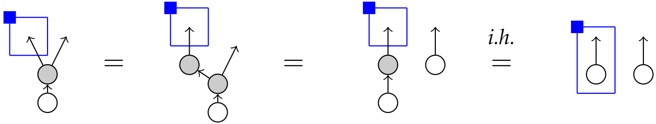


Induction example

- First apply induction to get two sub-goals:

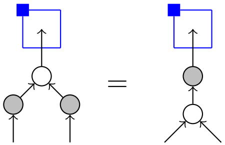


- The base case is an assumption, step case by rewriting:



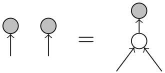
Induction Example

Lemma

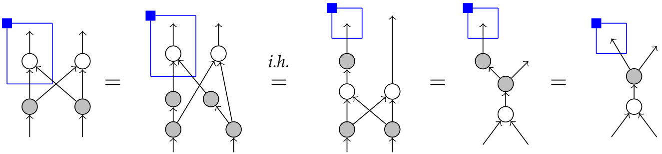


Proof.

Base:

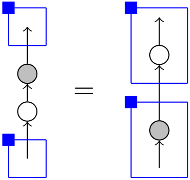


Step:



Induction Example

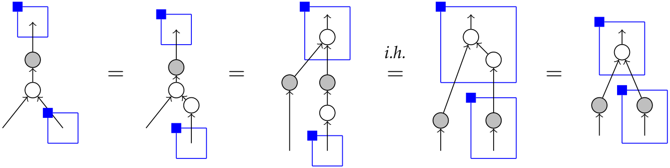
Theorem



Proof.

Base: (by lemma)

Step:



□

Interacting bialgebras

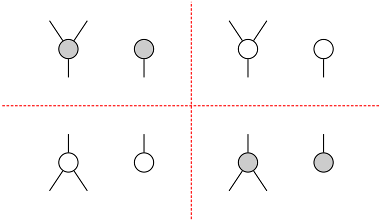
- Before, we considered algebras with nice, well-understood n.f.'s.

Interacting bialgebras

- Before, we considered algebras with nice, well-understood n.f.'s.
- Now, lets kick things up a notch, and study something whose algebraic behaviour is less well-understood.

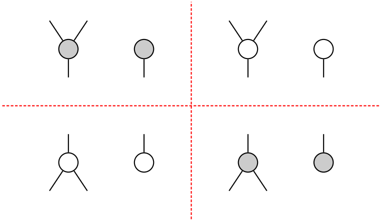
Interacting bialgebras

- Before, we considered algebras with nice, well-understood n.f.'s.
- Now, lets kick things up a notch, and study something whose algebraic behaviour is less well-understood.
- Consider two bi-algebras which interact with each other as Frobenius algebras:



Interacting bialgebras

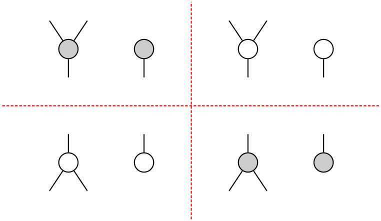
- Before, we considered algebras with nice, well-understood n.f.'s.
- Now, lets kick things up a notch, and study something whose algebraic behaviour is less well-understood.
- Consider two bi-algebras which interact with each other as Frobenius algebras:



- This theory is known as $\mathbb{IB}$, or the phase-free fragment of the ZX-calculus.

Interacting bialgebras

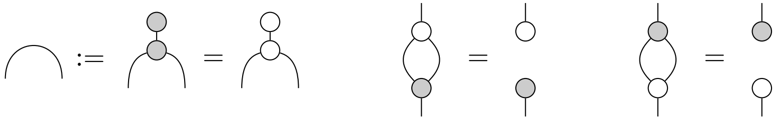
- Before, we considered algebras with nice, well-understood n.f.'s.
- Now, lets kick things up a notch, and study something whose algebraic behaviour is less well-understood.
- Consider two bi-algebras which interact with each other as Frobenius algebras:



- This theory is known as $\mathbb{IB}$, or the phase-free fragment of the ZX-calculus.
- Its pops up all over the place: signal-flow networks, Petri nets with boundaries, quantum circuits...

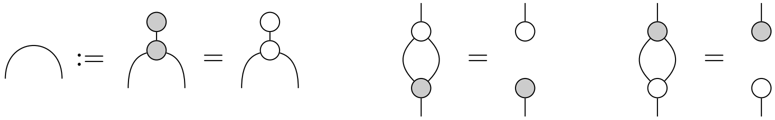
Interacting bialgebras

- The simplest example also assumes:



Interacting bialgebras

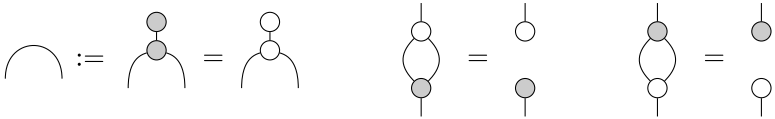
- The simplest example also assumes:



- The first essentially means we can ignore directions in diagrams, and the second means these *bialgebras* are actually *Hopf algebras*, with trivial antipode.

Interacting bialgebras

- The simplest example also assumes:



- The first essentially means we can ignore directions in diagrams, and the second means these *bialgebras* are actually *Hopf algebras*, with trivial antipode.
- Last year, Sobocinski and Bonchi showed (using non-rewriting techniques) that the PROP for this thing is $\mathrm{VecRel}_{\mathbb{Z}_2}$, the category of *linear relations*.

Interacting bialgebras are linear relations

- A linear relation from V to W is just a subspace of $V \times W$. They are composed relation-style.

Interacting bialgebras are linear relations

- A linear relation from V to W is just a subspace of $V \times W$. They are composed relation-style.
- In $\mathrm{VecRel}_{\mathbb{Z}_2}$, maps $f : m \to n$ are subspaces of $\mathbb{Z}_2^m \times \mathbb{Z}_2^n$.

Interacting bialgebras are linear relations

- A linear relation from V to W is just a subspace of $V \times W$. They are composed relation-style.
- In $\mathrm{VecRel}_{\mathbb{Z}_2}$, maps $f : m \to n$ are subspaces of $\mathbb{Z}_2^m \times \mathbb{Z}_2^n$.
- This gives us a natural notion of pseudo-normal form for diagrams:

Interacting bialgebras are linear relations

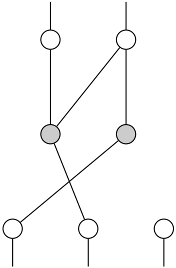
- A linear relation from V to W is just a subspace of $V \times W$. They are composed relation-style.
- In $\mathrm{VecRel}_{\mathbb{Z}_2}$, maps $f : m \to n$ are subspaces of $\mathbb{Z}_2^m \times \mathbb{Z}_2^n$.
- This gives us a natural notion of pseudo-normal form for diagrams:
    - **white dots** are place-holders

Interacting bialgebras are linear relations

- A linear relation from V to W is just a subspace of $V \times W$. They are composed relation-style.
- In $\text{VecRel}_{\mathbb{Z}_2}$, maps $f : m \to n$ are subspaces of $\mathbb{Z}_2^m \times \mathbb{Z}_2^n$.
- This gives us a natural notion of pseudo-normal form for diagrams:
  - **white dots** are place-holders
  - **grey dots** are vectors spanning the subspace

Lets see how this works...

- Subspaces can be represented as:

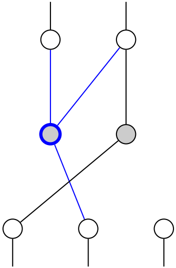


$$\leftrightarrow \quad \left\langle \begin{pmatrix} 0 \\ 1 \\ 0 \\ \hline 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ \hline 0 \\ 1 \end{pmatrix} \right\rangle$$

- The 1′s indicate where edges appear for each vector.

Lets see how this works...

- Subspaces can be represented as:

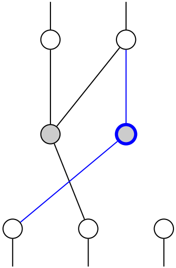


$$\leftrightarrow \quad \left\langle \begin{pmatrix} 0 \\ 1 \\ 0 \\ \hline 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ \hline 0 \\ 1 \end{pmatrix} \right\rangle$$

- The 1's indicate where edges appear for each vector.

Lets see how this works...

- Subspaces can be represented as:

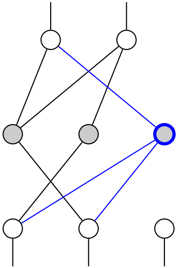


$$\leftrightarrow \quad \left\langle \begin{pmatrix} 0 \\ 1 \\ 0 \\ \hline 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ \hline 0 \\ 1 \end{pmatrix} \right\rangle$$

- The 1′s indicate where edges appear for each vector.

Lets see how this works...

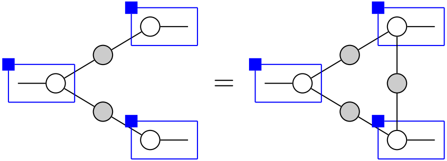
- However, this is not unique. We can always add or remove a vector that is the sum of two other spanning vectors and get the same space:



$$\leftrightarrow \quad \left\langle \begin{pmatrix} 0 \\ 1 \\ 0 \\ \hline 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ \hline 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \\ \hline 1 \\ 0 \end{pmatrix} \right\rangle$$

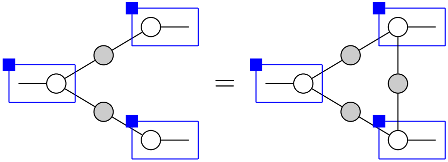
Addition is a !-box rule

- This 'addition' operation can be written as a !-box rule:

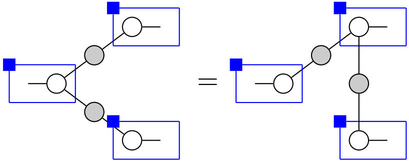


Addition is a !-box rule

- This ‘addition’ operation can be written as a !-box rule:

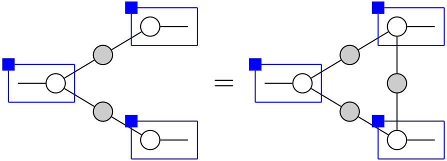


- We can also apply this forward then backward to get a ‘rotation’ rule:

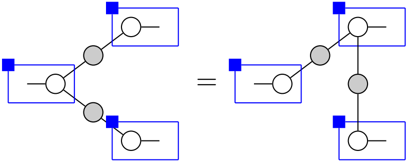


Addition is a !-box rule

- This 'addition' operation can be written as a !-box rule:



- We can also apply this forward then backward to get a 'rotation' rule:



- Note this rule decreases the arity of the white dot on the left by 1.

A reduction strategy...

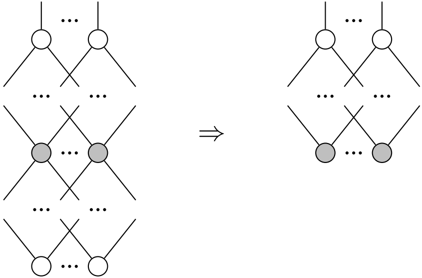
- This gives a reduction strategy for **IB**-diagrams.

A reduction strategy...

- This gives a reduction strategy for **IB**-diagrams.
- First, write diagram as a layer of **interior white** dots, then **interior grey** dots, then **boundary white** dots.

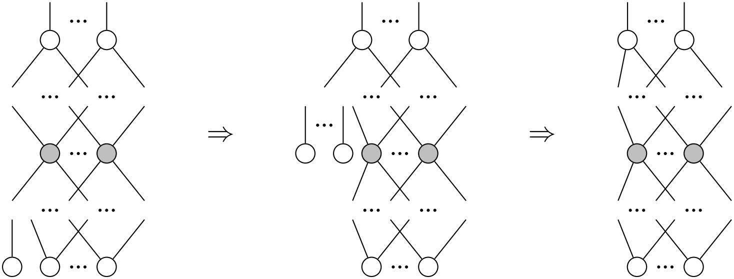
A reduction strategy...

- This gives a reduction strategy for **IB**-diagrams.
- First, write diagram as a layer of **interior white** dots, then **interior grey** dots, then **boundary white** dots.
- To get to pseudo-normal form, we just need to get rid of the interior white dots:



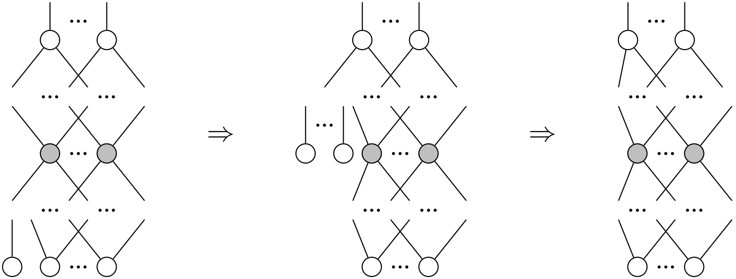
A reduction strategy...

- We do this by applying a rule to reduce the arity of a single white dot, until the arity is 1, then copy through:



A reduction strategy...

- We do this by applying a rule to reduce the arity of a single white dot, until the arity is 1, then copy through:



- **Time to fire up Quantomatic!**

Thanks!

- Joint work with Lucas Dixon, Alex Merry, Ross Duncan, Vladimir Zamdzhiev, David Quick, and others
- See: `quantomatic.github.io`