New reasoning techniques for monoidal algebra

Aleks Kissinger

November 4, 2015
Algebra and rewriting

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

\[(a \cdot b) \cdot c = a \cdot (b \cdot c)\]

and

\[a \cdot e = a = e \cdot a\]

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

\[\frac{(a \cdot b) \cdot c = a \cdot (b \cdot c)}{a \cdot e = a = e \cdot a}\]

It is also possible to write these equations as trees:
Algebra and rewriting

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

\[(a \cdot b) \cdot c = a \cdot (b \cdot c)\]  \[\text{and}\]  \[a \cdot e = a = e \cdot a\]
Algebra and rewriting

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

\[(a \cdot b) \cdot c = a \cdot (b \cdot c) \quad \text{and} \quad a \cdot e = a = e \cdot a\]

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

\[(a \cdot b) \cdot c \rightarrow a \cdot (b \cdot c) \quad a \cdot e \rightarrow a \quad e \cdot a \rightarrow a\]
Algebra and rewriting

- Consider a monoid \((A, \cdot, e)\):
  \[
  (a \cdot b) \cdot c = a \cdot (b \cdot c) \quad \text{and} \quad a \cdot e = a = e \cdot a
  \]

- Normally, mathematical tools, e.g. automated theorem provers would use these equations as rewrite rules:
  \[
  (a \cdot b) \cdot c \rightarrow a \cdot (b \cdot c) \quad a \cdot e \rightarrow a \quad e \cdot a \rightarrow a
  \]

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

\[
\begin{align*}
\text{Tree 1:} & \quad \begin{array}{c}
\text{Node} \quad \begin{array}{c}
\text{Node} \quad \begin{array}{c}
\text{Node} \quad \begin{array}{c}
\text{Node}
\end{array}
\end{array}
\end{array}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{Tree 2:} & \quad \begin{array}{c}
\text{Node} \quad \begin{array}{c}
\text{Node} \quad \begin{array}{c}
\text{Node}
\end{array}
\end{array}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{Tree 3:} & \quad \begin{array}{c}
\text{Node} \quad \begin{array}{c}
\text{Node}
\end{array}
\end{array}
\end{align*}
\]
Algebra and rewriting

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

\[
\begin{align*}
\begin{array}{c}
\text{Diagram 1} \\
\text{Diagram 2} \\
\text{Diagram 3}
\end{array}
\end{align*}
\]
Algebra and rewriting

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

\[
\begin{align*}
  a & \quad b & \quad c \\
  a & \quad b & \quad c \\
\end{align*}
\]

- 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 
\[ (a \cdot b) \cdot c \rightarrow a \cdot (b \cdot c) \]
using the usual “instantiate, match, replace” style:

\[ w \cdot ((x \cdot (y \cdot e)) \cdot z) \quad \rightarrow \quad w \cdot (x \cdot ((y \cdot e) \cdot z)) \]
Diagram substitution

- One could apply the rule 
  \[(a \cdot b) \cdot c \rightarrow a \cdot (b \cdot c)\]
  using the usual 
  “instantiate, match, replace” style:

  \[w \cdot ((x \cdot (y \cdot e)) \cdot z) \rightarrow w \cdot (x \cdot ((y \cdot e) \cdot z))\]

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

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

  \[
  w \cdot ((x \cdot (y \cdot e)) \cdot z) \quad \rightarrow \quad w \cdot (x \cdot ((y \cdot e) \cdot z))
  \]

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

  ![Diagram showing diagrammatic reasoning example]

  - 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 \( \bigtriangleup \) and a counit \( \Box \) satisfying:

\[
\begin{array}{ccc}
\text{comultiplication} & = & \text{comultiplication} \\
\text{counit} & = & \text{counit}
\end{array}
\]
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 \( \bigotimes \) and a *counit* \( \bigcirc \) satisfying:

\[
\begin{align*}
\bigotimes \bigcirc &= \bigcirc \\
\bigcirc &= \bigcirc
\end{align*}
\]

Algebra and coalgebra can interact in many interesting ways:

\[
\begin{align*}
\bigotimes &= \bigotimes \\
\bigotimes &= \bigotimes
\end{align*}
\]
Equational reasoning with diagram substitution

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

\[
\begin{array}{c}
\uparrow \\
\circ \\
\downarrow \\
\end{array}
\quad = 
\begin{array}{c}
\uparrow \\
\circ \\
\end{array}
\]

- This style of rewriting works for any (co)algebraic structure in a monoidal category, a.k.a. monoidal algebras.
Equational reasoning with diagram substitution

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

\[
\begin{align*}
\begin{array}{c}
\begin{tikzpicture}[baseline=0, thick, every node/.style={scale=0.8}, every label/.style={scale=0.8}]
  \node (2) at (-0.5,1) [circle, draw, fill=black!20] {}; 
  \node (1) at (0.5,1) [circle, draw, fill=black!20] {}; 
  \draw[thick, ->] (1) -- (2); 
  \draw[thick, ->] (2) -- (1); 
\end{tikzpicture}
\end{array} & = \\
\begin{tikzpicture}[baseline=0, thick, every node/.style={scale=0.8}, every label/.style={scale=0.8}]
  \node (2) at (-0.5,1) [circle, draw, fill=black!20] {}; 
  \node (1) at (0.5,1) [circle, draw, fill=black!20] {}; 
  \draw[thick, ->] (1) -- (2); 
  \draw[thick, ->] (2) -- (1); 
\end{tikzpicture}
\end{array}
\end{align*}
\]

- 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.
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 $A$:

\[
\begin{array}{c}
A \\
\textcircled{A} \\
A
\end{array}
\]

\[
\begin{array}{c}
A \\
\textcircled{A}
\end{array}
\]

\[
\begin{array}{c}
A \\
\textcircled{A}
\end{array}
\]

\[
\begin{array}{c}
A
\end{array}
\]

\[
\begin{array}{c}
\vdots
\end{array}
\]

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

• and some equations:

\[
\begin{array}{c}

\begin{array}{c}
\vdots
\end{array}
\end{array}
\]
Example: Frobenius algebras

- A commutative Frobenius algebra consists of a tuple \((A, \circlearrowleft, \circlearrowright, \circlearrowleft, \circlearrowright)\) such that:
  - \((A, \circlearrowleft, \circlearrowright)\) forms a commutative monoid:
    - \(\cdot\) is associative
    - There exists an identity element \(I\)
    - COMMUTATIVE
  - \((A, \circlearrowleft, \circlearrowright)\) forms a commutative comonoid:
    - \(\cdot\) is associative
    - There exists a co-identity element \(I\)
    - COMMUTATIVE
  - The Frobenius law is satisfied:
    - \(\cdot\) is associative
Example: Bialgebras

- A *(bi)commutative bialgebra* consists of a tuple \((A, \oplus, \odot, \odot, \odot)\) such that:
  - \((A, \oplus, \odot)\) forms a monoid:
    - Diagrams representing monoid properties:
  - \((A, \odot, \odot)\) forms a comonoid:
    - Diagrams representing comonoid properties:
  - The *bialgebra laws* are satisfied:
    - Diagrams representing bialgebra laws:
PROPs

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

1. Define a theory category $\mathcal{T}$ whose objects are natural numbers (i.e. arities) and:

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

   For SMCs, this is called a **PROduct** category with **Permutations** (PROP).
• Monoidal algebras can also be defined via *functorial semantics*:
  1. Define a theory category $\mathcal{T}$ whose objects are natural numbers (i.e. arities) and:

\[
m \otimes n := m + n
\]

For SMCs, this is called a **PROduct category with Permutations (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 **PROduct category with Permutations** (PROP).

2. Fix another SMC $C$ (e.g. functions, relations, linear maps, etc.).
3. $\mathbb{T}$-algebras in $C$ are then symmetric monoidal functors:

$$\llbracket - \rrbracket : \mathbb{T} \to C$$
Monoidal algebras can also be defined via *functorial semantics*:

1. Define a theory category $\mathcal{T}$ whose objects are natural numbers (i.e. arities) and:

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

   For SMCs, this is called a **PROduct category with Permutations (PROP)**.

2. Fix another SMC $\mathcal{C}$ (e.g. functions, relations, linear maps, etc.).

3. $\mathcal{T}$-algebras in $\mathcal{C}$ are then symmetric monoidal functors:

   $$\llbracket - \rrbracket : \mathcal{T} \to \mathcal{C}$$

**PROPs come in two flavours:**
**PROPs**

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

    \[ m \otimes n := m + n \]

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

    \[ \llbracket - \rrbracket : \mathcal{T} \to C \]

- PROPs come in two flavours:
  1. *Syntactic* PROPs have as morphisms diagrams of generators, modulo some set of diagram equations. Deciding equality $\iff$ solving a word problem.
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 **PROP**duct category with **Permutations** (PROP).

2. Fix another SMC $C$ (e.g. functions, relations, linear maps, etc.).

3. $\mathbb{T}$-algebras in $C$ are then symmetric monoidal functors:

   $$\llbracket - \rrbracket : \mathbb{T} \to C$$

**PROPs** come in two flavours:

1. *Syntactic* PROPs have as morphisms diagrams of generators, modulo some set of diagram equations. Deciding equality $\iff$ 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 $F$ 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 $F$ 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:

$$(f \otimes g)(i) = \begin{cases} f(i) & \text{if } i < m \\ g(i - m) + n & \text{if } i \geq m \end{cases}$$
Example: Commutative monoids are functions

- Let $F$ be the PROP whose morphisms $f : m \rightarrow n$ are functions between finite sets:

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

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

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

- 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 \( \circ \):
Example: Commutative monoids are functions

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

- Then, any diagram of $\circlearrowright$ and $\circlearrowleft$ 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 \( \bigcirc \):

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

- Similarly, \( \mathcal{F}^{\text{op}} \) is the PROP for cocommutative comonoids.
Distributive laws

- What happens when we combine two monoidal algebras, e.g. $(\mathcal{A}, \odot)$ and $(\mathcal{B}, \odot)$?
Distributive laws

• What happens when we combine two monoidal algebras, e.g. $(\uparrow, \bigcirc)$ and $(\nabla, \bullet)$?
• ...not much!
Distributive laws

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

- What happens when we combine two monoidal algebras, e.g. 
  \[(\bigtriangleup, \bigtriangledown)\text{ and } (\bigtriangleup, \bigtriangledown)\]?
- ...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. \((\odot, \circlearrowleft)\) and \((\odot, \circlearrowright)\)?
- ...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 \((\bullet, \circ\)) and a comonoid \((\circ, \bullet\)), and a distributive law:

\[
\begin{align*}
\text{Diagram 1} & = \text{Diagram 2} & \text{Diagram 3} & = \text{Diagram 4}
\end{align*}
\]
Example: Bialgebras are matrices

- Bialgebras consist of a monoid \((\bigcirc, \circlearrowright)\), a comonoid \((\bigcirc, \circlearrowleft)\), and a distributive law:

\[
\begin{align*}
\text{monoid} & \quad \text{comonoid} \\
\text{distributive law} & = \\
\text{normal forms} & = \\
\text{normal forms} & =
\end{align*}
\]

- So, normal forms look like this:
Example: Bialgebras are matrices

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

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

\[
\begin{pmatrix}
1 & 0 & 1 \\
1 & 2 & 0
\end{pmatrix}
\]
Example: Bialgebras are matrices

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

$$
\begin{pmatrix}
1 & 0 & 1 \\
1 & 2 & 0
\end{pmatrix}
$$
Many of these theorems have something in common: the deal with repeated structures, like trees and cotrees:

\[
\begin{align*}
\bullet & \quad \bullet \\
\quad & \quad =
\end{align*}
\]
Many of these theorems have something in common: the deal with repeated structures, like trees and cotrees:

...and tree/cotrees, a.k.a. spiders:
Diagrams with repetition

- Individual rules can be 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:

\[
\begin{array}{c}
\text{...}
\end{array}
\begin{array}{c}
\text{...}
\end{array}
= 
\begin{array}{c}
\text{...}
\end{array}
\begin{array}{c}
\text{...}
\end{array}
\]

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

\[
\begin{array}{c}
\text{...}
\end{array}
\begin{array}{c}
\text{...}
\end{array}
= 
\begin{array}{c}
\text{...}
\end{array}
\begin{array}{c}
\text{...}
\end{array}
\]
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.:

\[
\begin{align*}
\text{\textbullet} & \quad \text{\textbullet} \\
\circ & \quad \circ \\
\text{\textbullet} & \quad \text{\textbullet}
\end{align*}
\]

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.:

\[
\begin{array}{c}
\vdots \\
\end{array}

\begin{array}{c}
\vdots \\
\end{array}

= 

\begin{array}{c}
\vdots \\
\end{array}

\begin{array}{c}
\vdots \\
\end{array}

\begin{array}{c}
\vdots \\
\end{array}

\begin{array}{c}
\vdots \\
\end{array}

\begin{array}{c}
\vdots \\
\end{array}

\begin{array}{c}
\vdots \\
\end{array}

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 “\ldots”
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?
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 language
• We can formalise families of diagrams (with variable-arity generators) using some graphical syntax:

\[
\begin{array}{c}
\text{...} \\
\rightarrow
\end{array}
\Rightarrow
\begin{array}{c}
\text{!-boxes}
\end{array}
\]
!-boxes

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

\[
\begin{array}{c}
\longrightarrow \\
\end{array}
\]

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

\[
\left[ \begin{array}{c}
\text{...} \\
\end{array} \right] = \{ \begin{array}{c}
\text{...} \\
\end{array} \}
\]
The diagrams represented by a !-graph are all those obtained by performing EXPAND and KILL operations on !-boxes.
• The diagrams represented by a !-graph are all those obtained by performing EXPAND and KILL operations on !-boxes

![Diagrams](image)

• 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:

\[
\begin{array}{c}
\text{LHS} = \\
\text{RHS} =
\end{array}
\]

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

- COPY
  
  ![COPY diagram](image)

- DROP
  
  ![DROP diagram](image)
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:

- The operations are sound w.r.t. concrete instantiation, i.e. they don’t produce any new concrete instances.
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:

\[
\begin{align*}
\text{COPY}_b & \quad \rightarrow \\
\text{DROP}_{b'} & \quad \rightarrow
\end{align*}
\]

- 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:

\[
\begin{array}{c}
\text{[Diagram of graph structure]} \\
\text{= } \\
\Rightarrow \\
\text{= }
\end{array}
\]
!-graph to !-graph rewriting

- !-graph rewriting: first instantiate:

- Then apply:
Once we have !-boxes around, we can make recursive definitions:
Recursive definition

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

\[
\begin{cases}
\quad \vdash t := \vdash \emptyset \\
\quad \vdash t := \vdash t
\end{cases}
\]

• And, as usual, recursive definition goes hand-in-hand with inductive proof...
Induction principle for !-graphs

- Let $\text{FIX}_b(G = H)$ 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 $\text{FIX}$, we can define induction

$$
\frac{\text{KILL}_b(G = H) \quad \text{FIX}_b(G = H)}{\text{EXPAND}_b(G = H)} \quad \Rightarrow \quad G = H \quad \text{ind}
$$
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

\[
\text{KILL}_b(G = H) \quad \text{FIX}_b(G = H) \quad \implies \quad \text{EXPAND}_b(G = H) \quad \text{ind}
\]

\[
G = H \quad \text{ind}
\]

• 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:

\[
\begin{align*}
\circ & \quad = \quad (\text{empty}) \\
\end{align*}
\]
Induction example

• First apply induction to get two sub-goals:

\[ \begin{array}{c}
\vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image1.png}}} & = & \vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image2.png}}} \\
\vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image3.png}}} & \Rightarrow & \vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image4.png}}} \\
\vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image5.png}}} & = & \vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image6.png}}} \\
\ethod{empty} & \\
\end{array} \]

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

\[ \begin{array}{c}
\vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image7.png}}} & = & \vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image8.png}}} \\
\vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image9.png}}} & = & \vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image10.png}}} \\
\vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image11.png}}} & = & \vcenter{\hbox{\includegraphics[width=0.2\textwidth]{image12.png}}} \\
i.h. & \\
\end{array} \]
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, let’s 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, let’s 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:

  ![Diagram of interacting bialgebras](image)

- This theory is known as 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...
The simplest example also assumes:

\[ := \quad = \quad = \quad = \]
Interacting bialgebras

- The simplest example also assumes:

\[ := \quad = \quad = \quad = \]

- 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:

  ![Diagram](attachment:image.png)

- 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 $\text{VecRel}_{\mathbb{Z}_2}$, 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 $\text{VecRel}_{\mathbb{Z}_2}$, maps $f : m \rightarrow n$ 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 $\text{VecRel}_{\mathbb{Z}_2}$, 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:
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
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
Let's see how this works...

- Subspaces can be represented as:

\[
\begin{pmatrix}
0 \\
1 \\
0
\end{pmatrix},
\begin{pmatrix}
1 \\
0 \\
1
\end{pmatrix}
\]

- The 1's indicate where edges appear for each vector.
Let's see how this works...

- Subspaces can be represented as:

\[ \langle \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rangle \]

- The 1's indicate where edges appear for each vector.
Lets see how this works...

- Subspaces can be represented as:

\[ \langle \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \rangle \]

- The 1’s indicate where edges appear for each vector.
• 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:

\[
\langle \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \rangle
\]
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:

\[
\begin{align*}
\begin{array}{c}
\text{Original} \\
\begin{array}{c}
\text{Diagram 1} \\
\quad = \\
\quad \text{Diagram 2}
\end{array}
\end{array}
\end{align*}
\]

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

\[
\begin{align*}
\begin{array}{c}
\text{Original} \\
\begin{array}{c}
\text{Diagram 3} \\
\quad = \\
\quad \text{Diagram 4}
\end{array}
\end{array}
\end{align*}
\]
Addition is a \( \oplus \)-box rule

- This ‘addition’ operation can be written as a \( \oplus \)-box rule:

  \[
  \begin{align*}
  \begin{tikzpicture}
  \node (A) at (0,0) [circle,draw] {};
  \node (B) at (1,0) [circle,draw] {};
  \node (C) at (0,1) [circle,draw] {};
  \node (D) at (1,1) [circle,draw] {};
  \draw (A) -- (B);
  \draw (C) -- (B);
  \draw (A) -- (D);
  \draw (C) -- (D);
  \end{tikzpicture}
  \quad = \quad \begin{tikzpicture}
  \node (A) at (0,0) [circle,draw] {};
  \node (B) at (1,0) [circle,draw] {};
  \node (C) at (0,1) [circle,draw] {};
  \node (D) at (1,1) [circle,draw] {};
  \draw (A) -- (B);
  \draw (C) -- (B);
  \draw (A) -- (D);
  \draw (C) -- (D);
  \end{tikzpicture}
  \end{align*}
  \]

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

  \[
  \begin{align*}
  \begin{tikzpicture}
  \node (A) at (0,0) [circle,draw] {};
  \node (B) at (1,0) [circle,draw] {};
  \node (C) at (0,1) [circle,draw] {};
  \node (D) at (1,1) [circle,draw] {};
  \draw (A) -- (B);
  \draw (C) -- (B);
  \draw (A) -- (D);
  \draw (C) -- (D);
  \end{tikzpicture}
  \quad = \quad \begin{tikzpicture}
  \node (A) at (0,0) [circle,draw] {};
  \node (B) at (1,0) [circle,draw] {};
  \node (C) at (0,1) [circle,draw] {};
  \node (D) at (1,1) [circle,draw] {};
  \draw (A) -- (B);
  \draw (C) -- (B);
  \draw (A) -- (D);
  \draw (C) -- (D);
  \end{tikzpicture}
  \end{align*}
  \]

- 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:

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

⇒

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

⇒

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
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:

\[ \ldots \Rightarrow \ldots \Rightarrow \ldots \]

- 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