Interactive Proof for Diagrammatic Languages

Aleks Kissinger
SamsonFest 2013

June 3, 2013

So monoids...

So monoids...

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

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

So monoids...

- 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, an automated theorem prover would use these equations as rewrite rules, e.g.

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

So monoids...

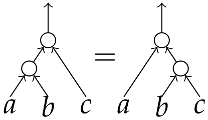
- Consider a monoid $(A, \cdot, e)$:

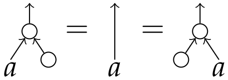
$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{and} \qquad a \cdot e = a = e \cdot a$$

- Normally, an automated theorem prover would use these equations as rewrite rules, e.g.

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

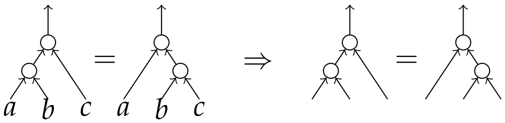
- It is also possible to write these equations as trees:





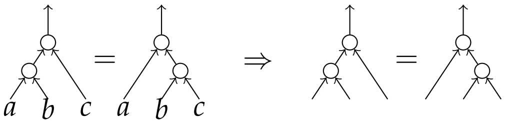
Monoids

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



Monoids

- 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*

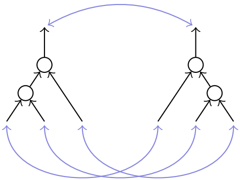


Diagram substitution

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

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

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:

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

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

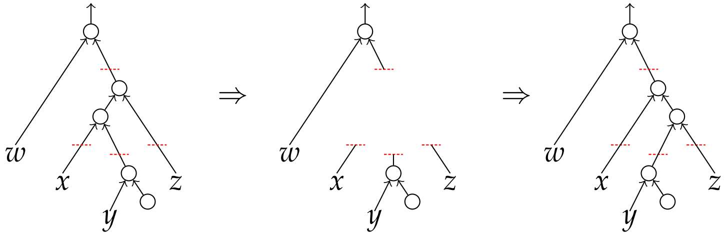
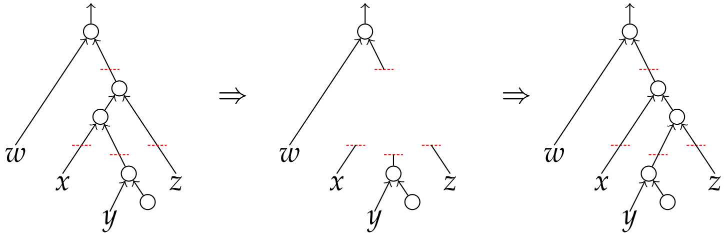


Diagram substitution

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

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

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



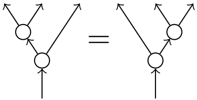
- This treats inputs and outputs symmetrically

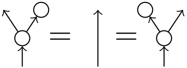
Algebra and coalgebra

- Coalgebra: algebraic structures “upside-down”

Algebra and coalgebra

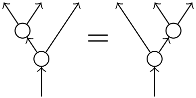
- Coalgebra: algebraic structures “upside-down”
- An example is a comonoid, which has a *comultiplication* operation  and a *counit*  satisfying:

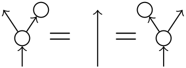




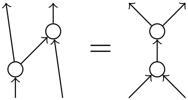
Algebra and coalgebra

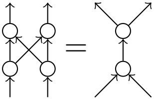
- Coalgebra: algebraic structures “upside-down”
- An example is a comonoid, which has a *comultiplication* operation  and a *counit*  satisfying:





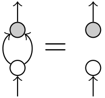
- Monoids and comonoids can interact in interesting ways, for instance:

Frobenius algebras: 

Bialgebras: 

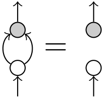
Equational reasoning with diagram substitution

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

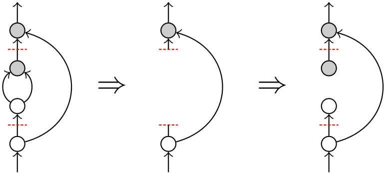


Equational reasoning with diagram substitution

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

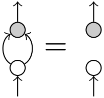


- For example:

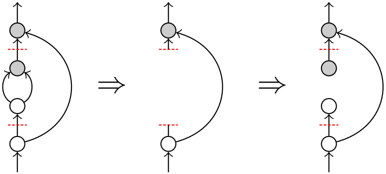


Equational reasoning with diagram substitution

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



- For example:



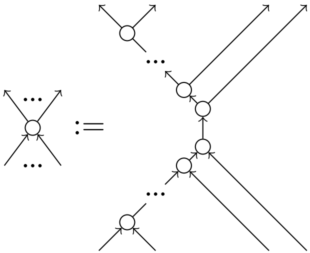
- This style of rewriting is sound and complete w.r.t. to traced symmetric monoidal categories

Diagrams with repetition

- In practice, many proofs concern infinite families of expressions

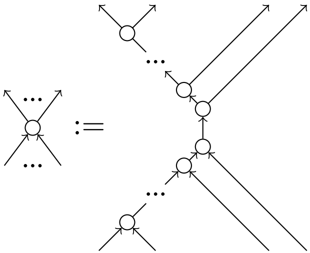
Diagrams with repetition

- In practice, many proofs concern infinite families of expressions
- As an example, define the (m, n)-fold multiplication/comultiplication as follows:

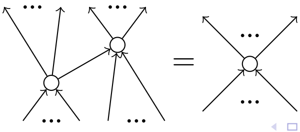


Diagrams with repetition

- In practice, many proofs concern infinite families of expressions
- As an example, define the (m, n)-fold multiplication/comultiplication as follows:

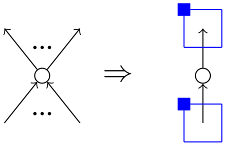


- An equivalent axiomitisation of (commutative) Frobenius algebras is:



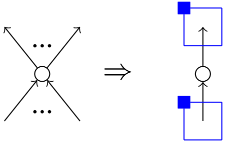
!-boxes

- We can formalise this “meta” diagram using some graphical syntax:

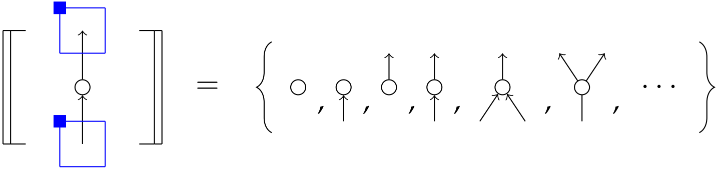


!-boxes

- We can formalise this “meta” diagram using some graphical syntax:

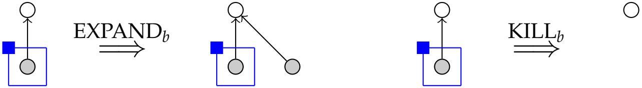


- The blue boxes are called !-boxes. A graph with !-boxes is called a !-graph. Can be interpreted as a set of concrete graphs:



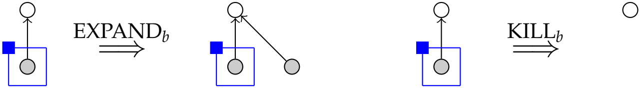
!-boxes

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

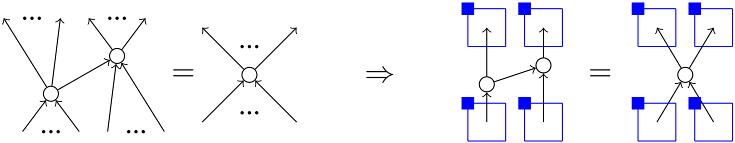


!-boxes

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

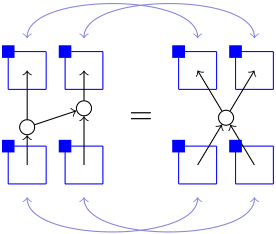


- We can also introduce equations involving !-boxes:



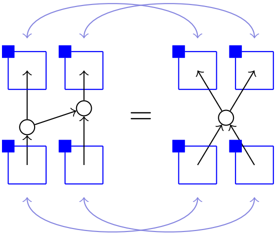
!-boxes: matching

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



!-boxes: matching

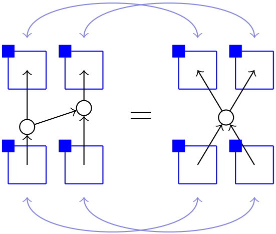
- !-boxes on the LHS are in 1-to-1 correspondence with RHS



- EXPAND and KILL operations applied to both sides simultaneously

!-boxes: matching

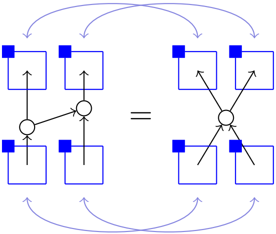
- !-boxes on the LHS are in 1-to-1 correspondence with RHS



- EXPAND and KILL operations applied to both sides simultaneously
- Rewriting concrete graphs: instantiate rule with EXPAND and KILL, then rewriting as usual

!-boxes: matching

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



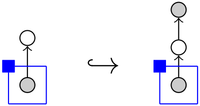
- EXPAND and KILL operations applied to both sides simultaneously
- Rewriting concrete graphs: instantiate rule with EXPAND and KILL, then rewriting as usual
- Sound and complete, in the absence of “wild” !-boxes

!-boxes: exact matching

- What about using !-graph equations to rewrite other !-graphs?

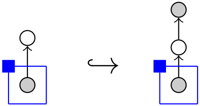
!-boxes: exact matching

- What about using !-graph equations to rewrite other !-graphs?
- Define an *exact matching* between !-graphs as an embedding that respects the !-boxes:



!-boxes: exact matching

- What about using !-graph equations to rewrite other !-graphs?
- Define an *exact matching* between !-graphs as an embedding that respects the !-boxes:



- However, there are other situations where one !-graph generalises another

!-boxes: inference rules

- Inference rules make new equations from old. Two obvious ones:

$$\frac{G = H}{\mathrm{EXPAND}_b(G = H)}\,exp \qquad\qquad \frac{G = H}{\mathrm{KILL}_b(G = H)}\,kill$$

!-boxes: inference rules

- Inference rules make new equations from old. Two obvious ones:

$$\frac{G = H}{\mathrm{EXPAND}_b(G = H)}\mathit{exp} \qquad \frac{G = H}{\mathrm{KILL}_b(G = H)}\mathit{kill}$$

- ...and some less obvious ones:

$$\frac{G = H}{\mathrm{COPY}_b(G = H)}\mathit{cp} \qquad \frac{G = H}{\mathrm{MERGE}_{b,b'}(G = H)}\mathit{mrg} \qquad \ldots$$

Induction Principle for !-Graphs

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

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

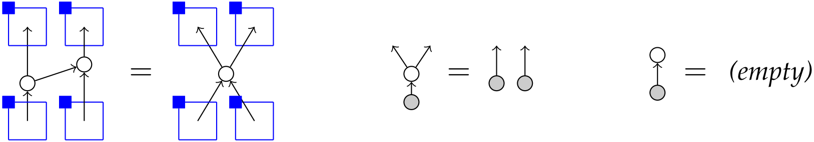
Induction Principle for !-Graphs

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

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

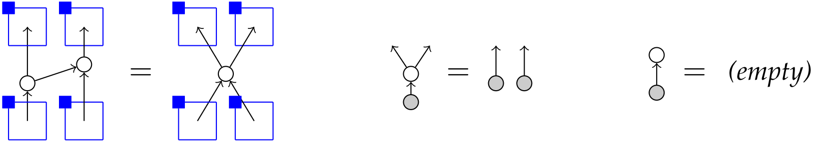
Induction example

- Suppose we have these three equations:

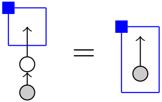


Induction example

- Suppose we have these three equations:

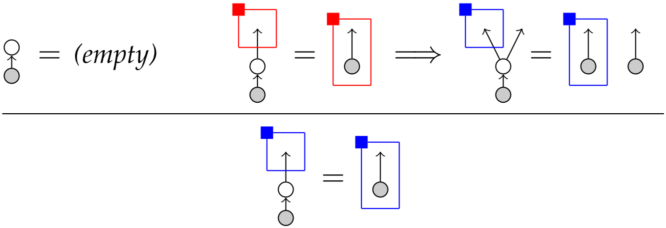


- ...then we can prove this using induction:



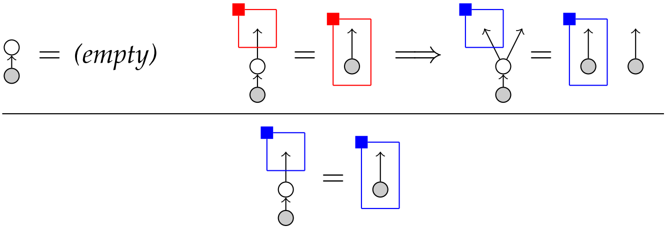
Induction example

- First (reverse) apply induction to get two sub-goals:

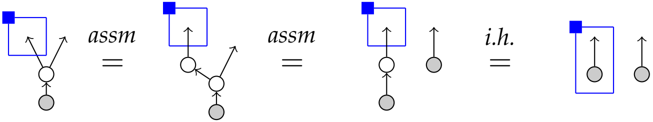


Induction example

- First (reverse) apply induction to get two sub-goals:



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



Constructing a diagrammatic proof assistant

- Why?

Constructing a diagrammatic proof assistant

- Why?
  - Diagrams are easier to understand, but easier to make mistakes

Constructing a diagrammatic proof assistant

- Why?
  - Diagrams are easier to understand, but easier to make mistakes
  - Want several layers of definition/abstraction (ex: quantum circuits and error-correcting encodings)

Constructing a diagrammatic proof assistant

- Why?
  - Diagrams are easier to understand, but easier to make mistakes
  - Want several layers of definition/abstraction (ex: quantum circuits and error-correcting encodings)
  - More expressive types of graphical languages $\Rightarrow$ new proof styles and techniques.

Constructing a diagrammatic proof assistant

- Why?
  - Diagrams are easier to understand, but easier to make mistakes
  - Want several layers of definition/abstraction (ex: quantum circuits and error-correcting encodings)
  - More expressive types of graphical languages $\Rightarrow$ new proof styles and techniques.
  - Unique from an HCI perspective. Possibly unexpected results.

Constructing a diagrammatic proof assistant

- Why?
  - Diagrams are easier to understand, but easier to make mistakes
  - Want several layers of definition/abstraction (ex: quantum circuits and error-correcting encodings)
  - More expressive types of graphical languages $\Rightarrow$ new proof styles and techniques.
  - Unique from an HCI perspective. Possibly unexpected results.
- Why not use terms?

Constructing a diagrammatic proof assistant

- Why?
  - Diagrams are easier to understand, but easier to make mistakes
  - Want several layers of definition/abstraction (ex: quantum circuits and error-correcting encodings)
  - More expressive types of graphical languages $\Rightarrow$ new proof styles and techniques.
  - Unique from an HCI perspective. Possibly unexpected results.
- Why not use terms?
  - There is a term language, using $\circ$, $\otimes$, swap maps, etc.

Constructing a diagrammatic proof assistant

- Why?
  - Diagrams are easier to understand, but easier to make mistakes
  - Want several layers of definition/abstraction (ex: quantum circuits and error-correcting encodings)
  - More expressive types of graphical languages $\Rightarrow$ new proof styles and techniques.
  - Unique from an HCI perspective. Possibly unexpected results.
- Why not use terms?
  - There is a term language, using $\circ$, $\otimes$, swap maps, etc.
  - Many congruences

Constructing a diagrammatic proof assistant

- Why?
  - Diagrams are easier to understand, but easier to make mistakes
  - Want several layers of definition/abstraction (ex: quantum circuits and error-correcting encodings)
  - More expressive types of graphical languages $\Rightarrow$ new proof styles and techniques.
  - Unique from an HCI perspective. Possibly unexpected results.
- Why not use terms?
  - There is a term language, using $\circ$, $\otimes$, swap maps, etc.
  - Many congruences
  - Simplest decision procedure: "draw the diagrams and compare"

Quantomatic: the good stuff

- Create, load, and save diagrams and rewrite rules

Quantomatic: the good stuff

- Create, load, and save diagrams and rewrite rules
- Apply rewrite rules manually, or normalise w.r.t. subsets of rewrite rules

Quantomatic: the good stuff

- Create, load, and save diagrams and rewrite rules
- Apply rewrite rules manually, or normalise w.r.t. subsets of rewrite rules
- Rewrites happen live, so proofs are easy to show off

Quantomatic: the good stuff

- Create, load, and save diagrams and rewrite rules
- Apply rewrite rules manually, or normalise w.r.t. subsets of rewrite rules
- Rewrites happen live, so proofs are easy to show off
- Education: Quantomatic-based labs for two years in conjunction with Categorical Quantum Mechanics course at Oxford

Quantomatic: limitations

- Once a proof is done, it’s gone. Only the result is left.

Quantomatic: limitations

- Once a proof is done, it's gone. Only the result is left.
- Only does rewriting, i.e. the purely equational part.

Quantomatic: limitations

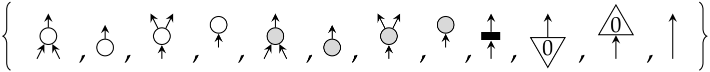
- Once a proof is done, it's gone. Only the result is left.
- Only does rewriting, i.e. the purely equational part.
- Rewrite rules are used naively. No search/normalisation strategies or Knuth-Bendix.

The Quanto2013 Projects

- Quantomatic is a (fairly) thin GUI built on QuantoCore, an ML based rewriting engine
- Starting this year, we are working on new projects based on QuantoCore:
  - **QuantoDerive** – graphical derivation editor, essentially the successor to Quantomatic GUI
  - **QuantoCosy** – conjecture synthesis for diagrams
  - **QuantoTactic** – Quantomatic/Isabelle integration

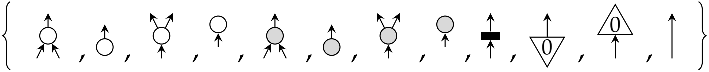
QuantoCosy

- Often, we have a concrete set of generators (e.g. a particular example of some algebraic structure), and we would like to derive the axioms
- Take a set of generators:

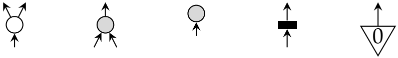


QuantoCosy

- Often, we have a concrete set of generators (e.g. a particular example of some algebraic structure), and we would like to derive the axioms
- Take a set of generators:

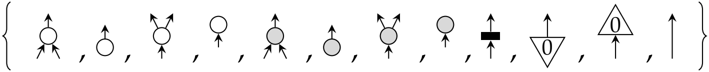


- For each disconnected graph, enumerate all of the ways it can be "plugged together":

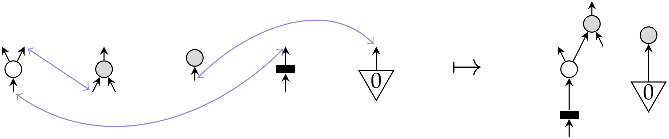


QuantoCosy

- Often, we have a concrete set of generators (e.g. a particular example of some algebraic structure), and we would like to derive the axioms
- Take a set of generators:

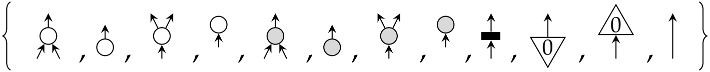


- For each disconnected graph, enumerate all of the ways it can be "plugged together":

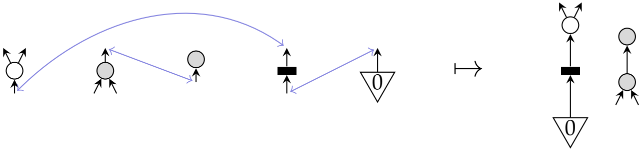


QuantoCosy

- Often, we have a concrete set of generators (e.g. a particular example of some algebraic structure), and we would like to derive the axioms
- Take a set of generators:



- For each disconnected graph, enumerate all of the ways it can be "plugged together":



QuantoCosy

- If we have concrete values for generators (e.g. as matrices), we can define an evaluation function $[\![-]\!]$ on diagrams

QuantoCosy

- If we have concrete values for generators (e.g. as matrices), we can define an evaluation function $[\![-]\!]$ on diagrams
- We can organise diagrams into equivalence classes $G \equiv H \Leftrightarrow [\![G]\!] = [\![H]\!]$

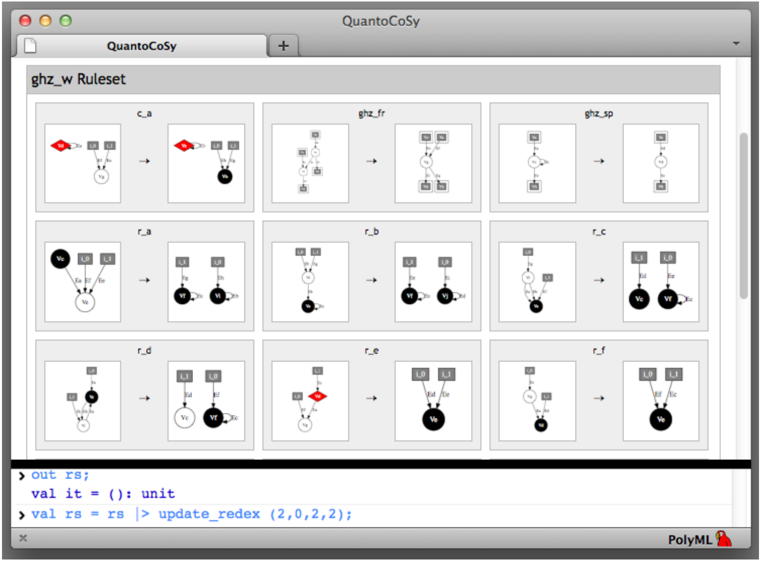
QuantoCosy

- If we have concrete values for generators (e.g. as matrices), we can define an evaluation function $[\![-]\!]$ on diagrams
- We can organise diagrams into equivalence classes $G \equiv H \Leftrightarrow [\![G]\!] = [\![H]\!]$
- If we define a metric on graphs, some equivalences $G \equiv H$ will become redexes $G \rightarrowtriangle H$

QuantoCosy

- If we have concrete values for generators (e.g. as matrices), we can define an evaluation function $[\![-]\!]$ on diagrams
- We can organise diagrams into equivalence classes $G \equiv H \Leftrightarrow [\![G]\!] = [\![H]\!]$
- If we define a metric on graphs, some equivalences $G \equiv H$ will become redexes $G \rightarrowtriangle H$
- In the 'Cosy style, we can use these redexes to cut down the search space by only enumerating *irreducible expressions*

QuantoCosy


QuantoCoSy
QuantoCoSy
ghz_w Ruleset
c_a
ghz_fr
ghz_sp
r_a
r_b
r_c
r_d
r_e
r_f
> out rs;
val it = (): unit
> val rs = rs |> update_redex (2,0,2,2);
PolyML

LCF-style Theorem Provers

- Theorem provers are large and complex. How can we be (fairly) confident they fit our mathematical models?

LCF-style Theorem Provers

- Theorem provers are large and complex. How can we be (fairly) confident they fit our mathematical models?
- In 1972, Milner came up with the LCF approach to automated theorem proving.

LCF-style Theorem Provers

- Theorem provers are large and complex. How can we be (fairly) confident they fit our mathematical models?
- In 1972, Milner came up with the LCF approach to automated theorem proving.
- The idea: write a kernel that is dumb (simple logic + a few inference rules) but sound

LCF-style Theorem Provers

- Theorem provers are large and complex. How can we be (fairly) confident they fit our mathematical models?
- In 1972, Milner came up with the LCF approach to automated theorem proving.
- The idea: write a kernel that is dumb (simple logic + a few inference rules) but sound
- Don't touch it! But tell it what to do with tactics, which are smart. The kernel is the "gatekeeper" of soundness.

QuantoTactic

- The idea: formalise equivalence up to diagrammatic equations in Isabelle:

$$\begin{aligned} \exists R, R' \quad & R \in \texttt{axioms} \wedge \\ & \texttt{instance-of}(R, R') \wedge \\ & \texttt{valid-rewrite}(R', G, H) \implies (G \equiv H) \end{aligned}$$

QuantoTactic

- The idea: formalise equivalence up to diagrammatic equations in Isabelle:

$$\exists R, R' \quad R \in \texttt{axioms} \wedge \texttt{instance-of}(R, R') \wedge \texttt{valid-rewrite}(R', G, H) \implies (G \equiv H)$$

- Wrap QuantoCore matching and rewriting capabilities in tactics, which do the hard stuff (e.g. finding witnesses R, R' for the implication above)

QuantoTactic

QuantoTactic is (or rather, will be...) three things:

1. A theory of diagrams and rewriting formalised in Isabelle

QuantoTactic

QuantoTactic is (or rather, will be...) three things:

1. A theory of diagrams and rewriting formalised in Isabelle
2. A tactic invoked by the prover, hooking the (powerful) Quantomatic core up to the (sound) Isabelle kernel

QuantoTactic

QuantoTactic is (or rather, will be...) three things:

1. A theory of diagrams and rewriting formalised in Isabelle
2. A tactic invoked by the prover, hooking the (powerful) Quantomatic core up to the (sound) Isabelle kernel
3. Language extensions and GUI support for inline graphical notation in proof documents

Thanks!

- Joint work with Lucas Dixon, Alex Merry, Ross Duncan, Vladimir Zamdzhiev, David Quick, and others
- See: `sites.google.com/site/quantomatic`