Equational reasoning with context-free families of string diagrams

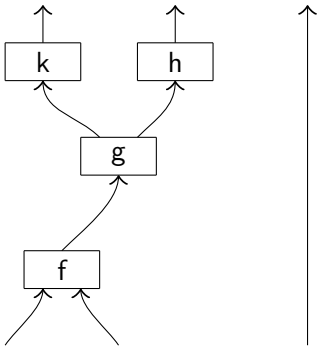
Aleks Kissinger Vladimir Zamdzhiev

Department of Computer Science
University of Oxford

21 July 2015

String Diagrams

Example



- First introduced by Roger Penrose in 1971 as alternative to the tensor-index notation used in theoretical physics.
- (Typed) nodes connected via (typed) wires
- Wires do not have to be connected to nodes at either end
- Open-ended wires serve as inputs/outputs
- Emphasis on compositionality

String diagram applications

Applications in:

- Monoidal category theory (sound and complete categorical reasoning)

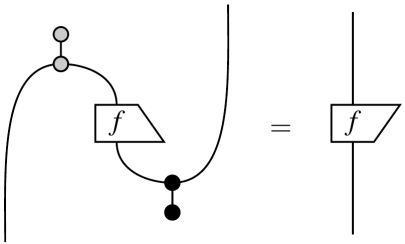


Figure: J. Vicary, W. Zeng (2014)

- Quantum computation and information (graphical calculi, e.g. ZX-calculus)

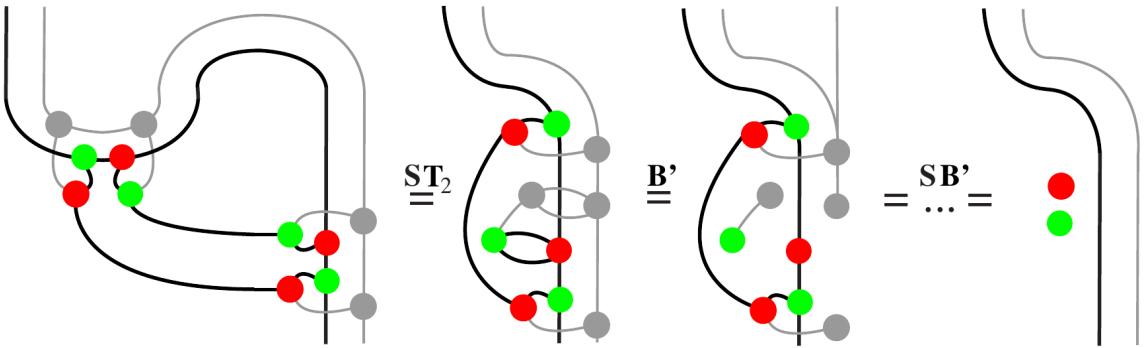


Figure: B. Coecke, R. Duncan (2011)

String diagram applications

- Concurrency (Petri nets)

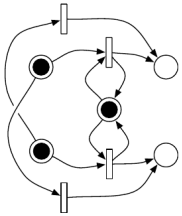


Figure: P. Sobocinski (2010)

- Computational linguistics (compositional semantics)

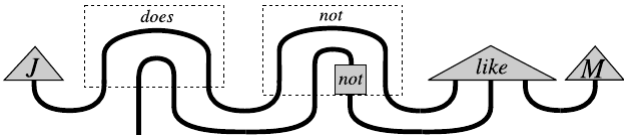


Figure: B. Coecke, E. Grefenstette, M. Sadrzadeh (2013)

String diagrams applications

- Control theory (signal-flow diagrams)

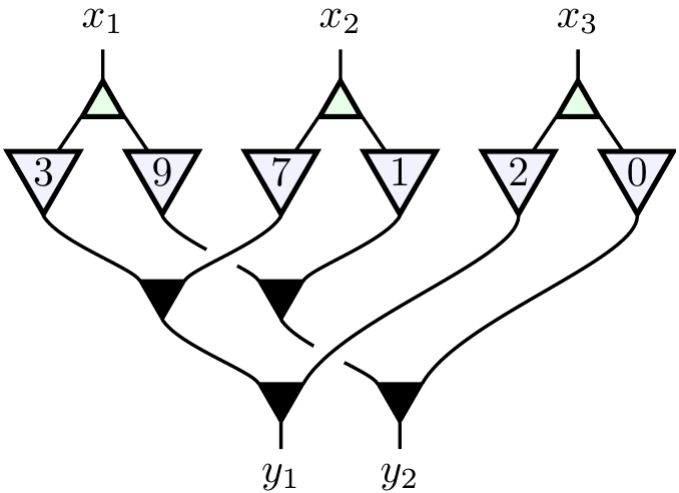


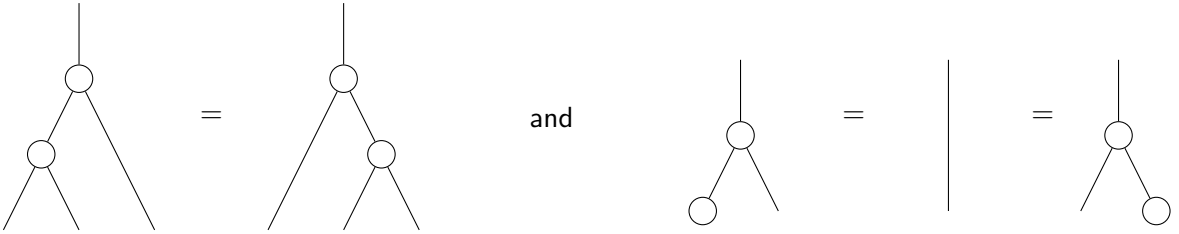
Figure: J. Baez, J. Erbele (2014)

String Diagram Example

A monoid is a triple $(A, \cdot, 1)$, such that:

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \quad \text{and} \quad 1 \cdot a = a = a \cdot 1$$

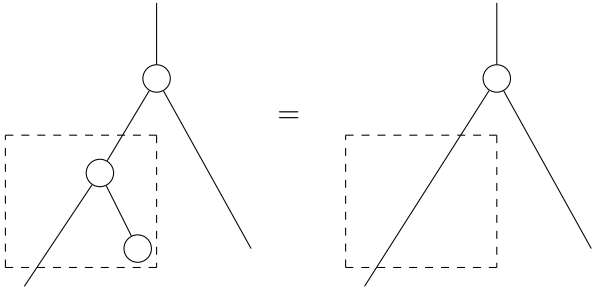
Setting $(\_ \cdot \_) :=$  and $1 :=$  ,we get:



String Diagram Example

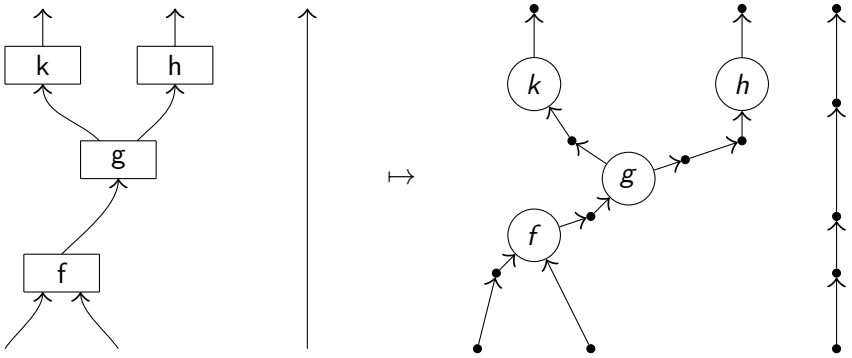
Equational reasoning is performed by replacing subdiagrams:

Example



String Graphs

Example

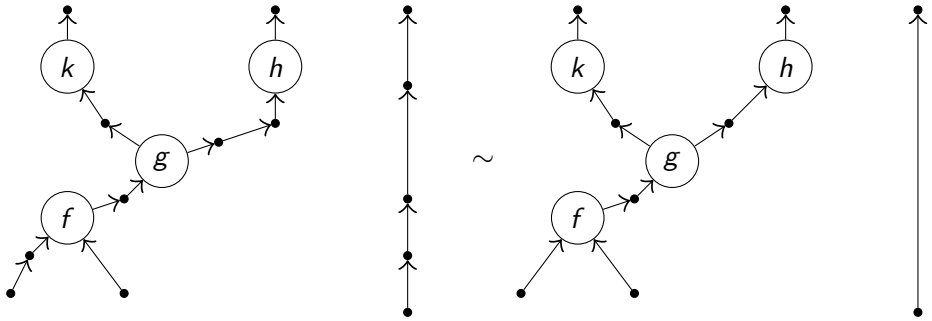


- String diagrams are formally described using (non-discrete) topological notions
- This is problematic for computer implementations
- Discrete representation exists in the form of *String Graphs*
- String graphs are typed (directed) graphs, such that:
  - Every vertex is either a *node-vertex* or a *wire-vertex*
  - No edges between node-vertices
  - In-degree of every wire-vertex is at most one
  - Out-degree of every wire-vertex is at most one

Wire-homeomorphism

We say two string graphs are equal if they are *wire-homeomorphic*, that is, we can obtain one from the other by increasing or decreasing the length of chains of wire-vertices.

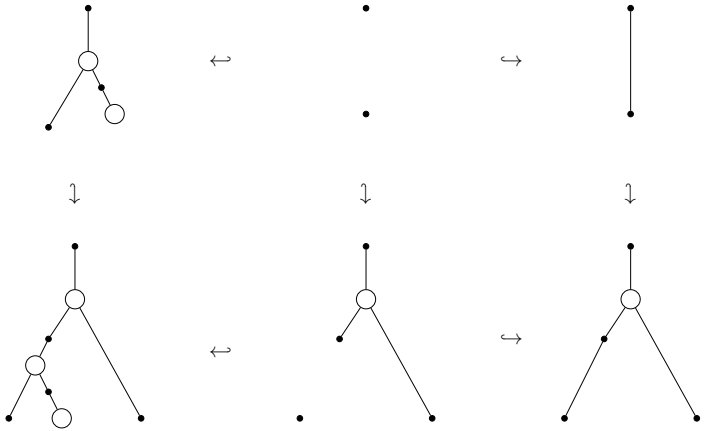
Example



By utilising wire-homeomorphism we can simulate string diagram matching and rewriting.

Reasoning with String Graphs

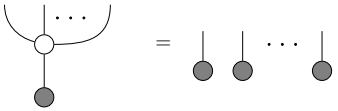
We use double-pushout (DPO) rewriting on string graphs to represent string diagram rewriting:



Families of string diagrams

- String diagrams (and string graphs) can be used to establish equalities between pairs of objects, one at a time.
- Proving infinitely many equalities simultaneously is only possible using metalogical arguments.

Example

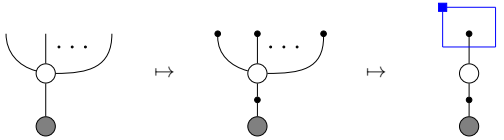


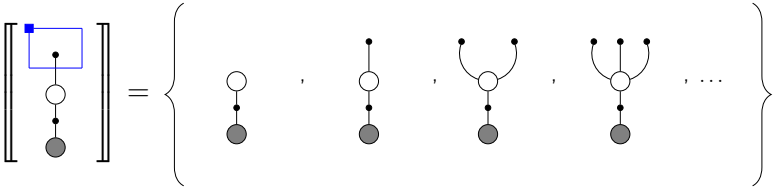
- However, this is imprecise and implementing software support for it would be very difficult.
- Handling this in a formal way is the primary motivation of this talk

Related Work

- A *!-graph* is a generalised string graph which allows us to represent an infinite family of string graphs in a formal way.
- Marked subgraphs called *!-boxes* can be repeated any number of times.

Example





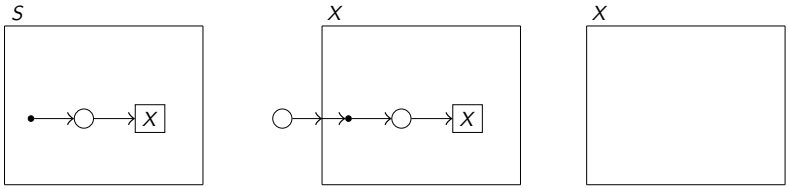
- However, !-graphs are not expressive enough to handle some languages of interest
- We propose an alternative which is more expressive than an important subclass of !-graphs

Context-free graph grammars

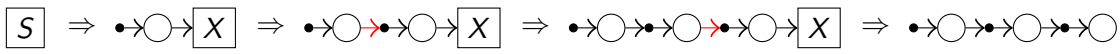
- We investigate context-free graph grammars first, as they have better structural, complexity and decidability properties compared to other more expressive graph grammars.
- Most studied context-free graph grammars are:
  - Hyperedge replacement grammars (HR)
  - Vertex replacement grammars (VR)
- Large body of literature available for both VR and HR grammars
- VR grammars (also known as C-edNCE grammars) are more expressive than HR grammars in general
- We will be working with VR grammars only, in particular boundary grammars (B-edNCE)

VR grammar example

The following grammar generates the set of all chains of node vertices with an input and no outputs:



A derivation in the above grammar of the string graph with three node vertices:



where we color the newly established edges in red.

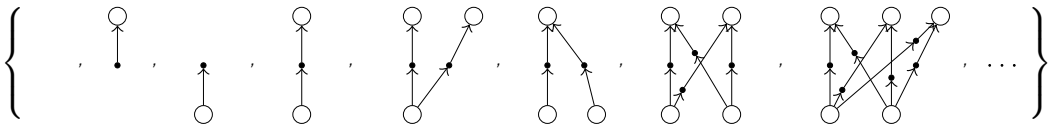
Expressivity of context-free grammars for string graphs

Proposition

A string graph language L can be generated by a VR grammar iff it can be generated by an HR grammar.

Corollary

The following language of string graphs:



cannot be directly generated by any VR grammar.

Proof.

Follows from a simple application of the pumping lemma for HR grammars. □

- The above language and other similar languages are of interest to us, so we propose a simple extension

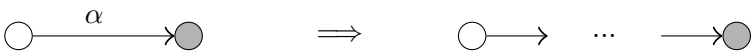
Encoded string graphs

Definition (Encoded string graph)

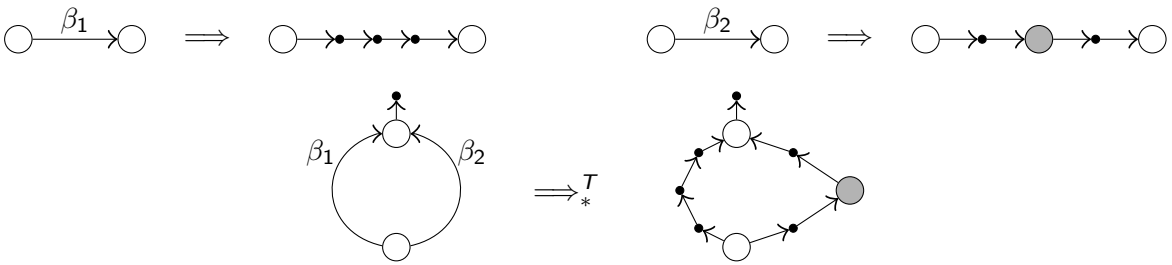
Let $\mathcal{E} = \{\alpha, \beta, \ldots\}$ be a finite set of *encoding symbols*. An *encoded string graph* is a string graph where we additionally allow edges labelled by encoding symbols $\alpha \in \mathcal{E}$ to connect pairs of node-vertices.

Definition (Decoding system)

A *decoding system* $\mathcal{T}$ is a set of DPO rewrite rules of the form:



Example



encoded string graph result of decoding

B-ESG grammars

Definition (B-ESG grammar)

A *B-ESG* grammar is a pair $B = (G, T)$, where T is a decoding system and G is a B-edNCE grammar, such that every production satisfies some static conditions, which we omit here.

- The conditions are static in the sense that they are easily decidable by a computer by simply inspecting the productions of the grammar.

B-ESG languages

- A derivation of a B-ESG grammar $B = (G, T)$ consists of generating an encoded string graph via G which is then decoded using T.
- The language of a B-ESG grammar B is the set of all graphs which can be derived from B.

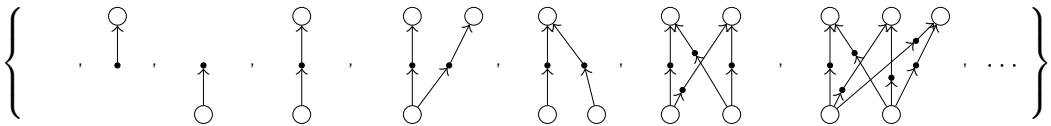
Theorem

Every graph in the language of a B-ESG grammar is a string graph.

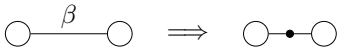
B-ESG grammar example

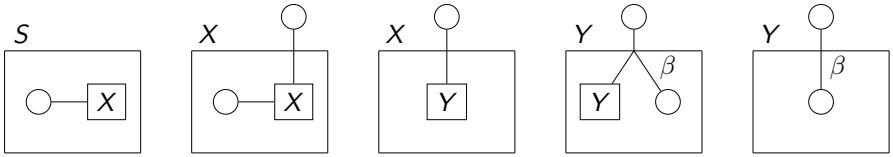
Example

The following language:

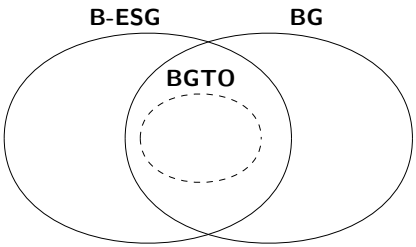


can be generated by the following B-ESG grammar:

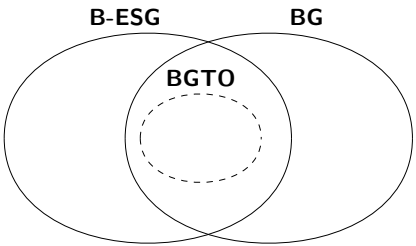




B-ESG grammar expressiveness

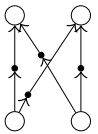


B-ESG grammar expressiveness



	!-graph	B-edNCE	B-ESG
$sK_{m,n}$	✓	✗	✓
sC_n	✗	✓	✓
sK_n	✗	✗	✓

sK_n ($sK_{m,n}$) - family of complete (bipartite) *string* graphs, sC_n - family of cycle string graphs, e.g.:

$sK_{2,2} :=$ 

B-ESG properties

Problem (Membership)

Given a string graph H and a B-ESG grammar B, does there exist a string graph $\widetilde{H} \sim H$, such that $\widetilde{H} \in L(B)$? In such a case, construct a derivation sequence $S \Longrightarrow_*^B \widetilde{H}$.

Theorem

The membership problem for B-ESG grammars is decidable.

Problem (Match-enumeration)

Given a string graph H and a B-ESG grammar B, enumerate all of the B-ESG concrete derivations $S \Longrightarrow_*^B K$, such that there exists a matching $m : K \to \widetilde{H}$ for some $\widetilde{H} \sim H$.

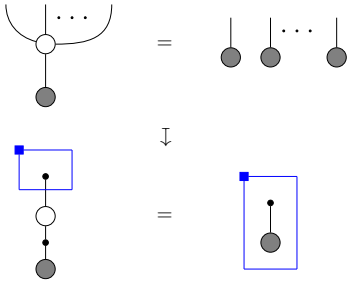
Theorem

The match-enumeration problem for a B-ESG grammar B is decidable if B is a match-exhaustive grammar.

- A B-ESG grammar is match-exhaustive if it satisfies certain other static conditions which we omit here.
- Decidability of the match-enumeration problem is of central importance for equational reasoning using a proof assistant

Quantification over equalities

- !-graphs can be used to formally establish infinitely many equalities via !-graph rewrite rules:



- Naturally, we wish to consider an alternative involving B-ESG grammars
- We can do this by using two B-ESG grammars whose productions are paired-up

B-ESG rewrite pattern

Definition (B-ESG rewrite pattern)

A *B-ESG rewrite pattern* is a pair of B-ESG grammars B_1 and B_2, such that there is a bijection between their productions which also:

- preserves non-terminals and their labels
- preserves inputs/outputs

Definition (B-ESG pattern instantiation)

Given a B-ESG rewrite pattern (B_1, B_2), a B-ESG pattern instantiation is given by a pair of concrete derivations:

$$S \Longrightarrow^{B_1}_{v_1,p_1} H_1 \Longrightarrow^{B_1}_{v_2,p_2} H_2 \Longrightarrow^{B_1}_{v3,p3} \cdots \Longrightarrow^{B_1}_{v_n,p_n} H_n \Longrightarrow^{T}_{*} F$$

and

$$S \Longrightarrow^{B_2}_{v_1,p_1} H'_1 \Longrightarrow^{B_2}_{v_2,p_2} H'_2 \Longrightarrow^{B_2}_{v3,p3} \cdots \Longrightarrow^{B_2}_{v_n,p_n} H'_n \Longrightarrow^{T}_{*} F'$$

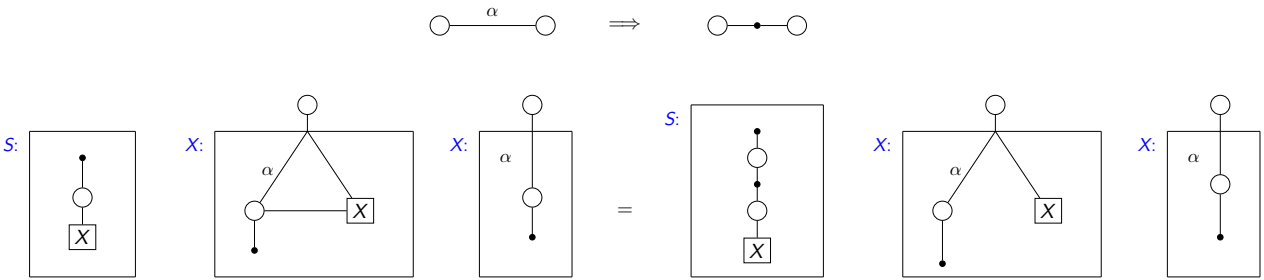
- That is, we always expand the same non-terminals in the two sentential forms in parallel

Theorem

Every B-ESG pattern instantiation is a string graph rewrite rule.

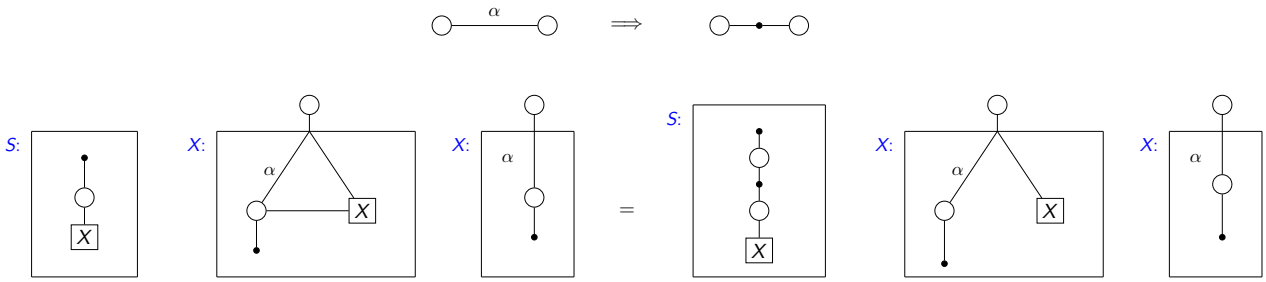
B-ESG rewrite pattern

Example



B-ESG rewrite pattern

Example



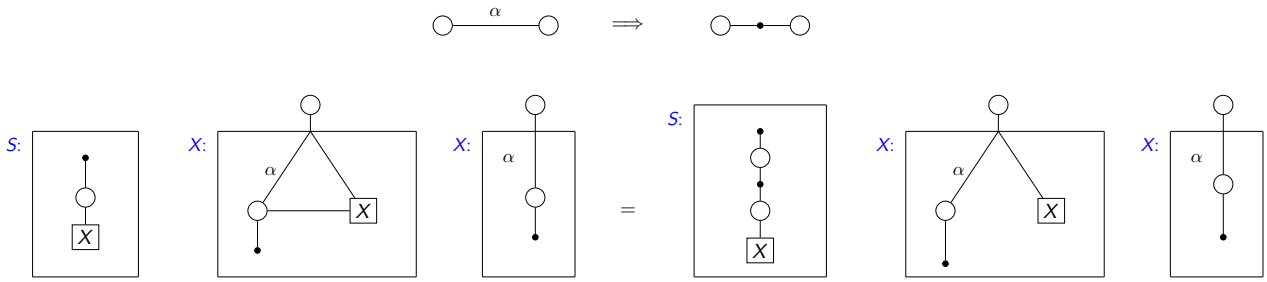
- Instantiation :

$\boxed{S}$

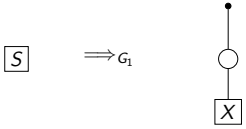
$\boxed{S}$

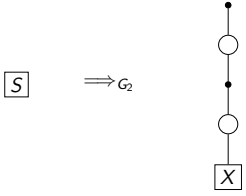
B-ESG rewrite pattern

Example



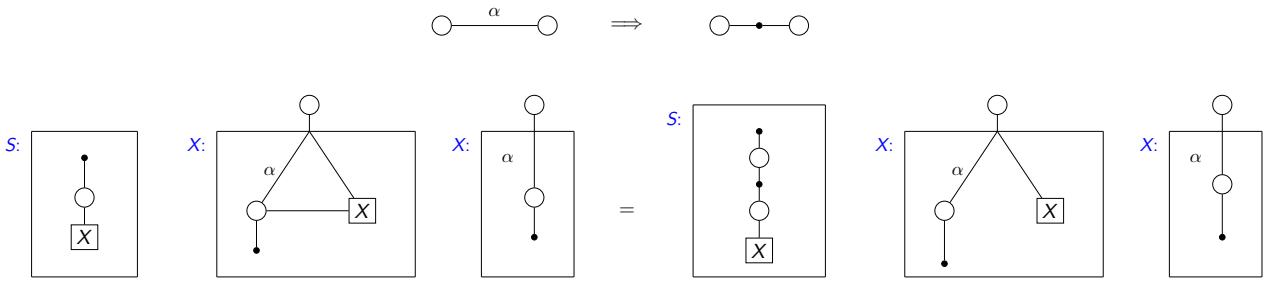
- Instantiation :



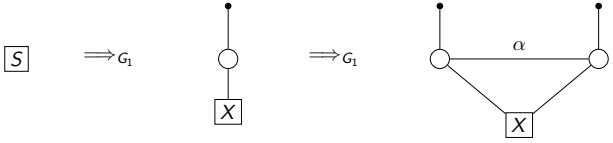


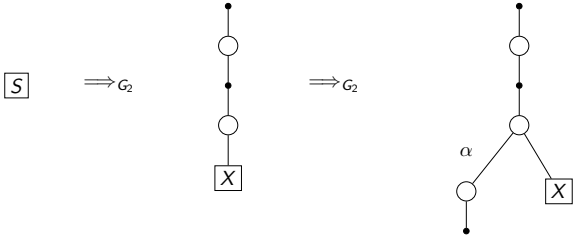
B-ESG rewrite pattern

Example



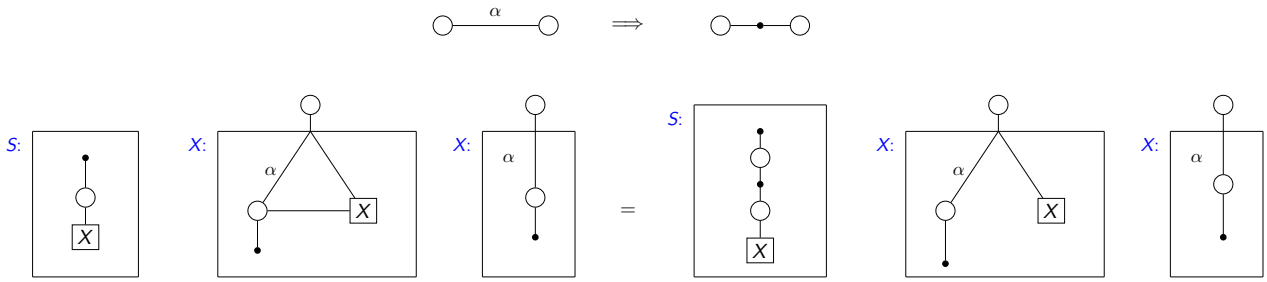
- Instantiation :



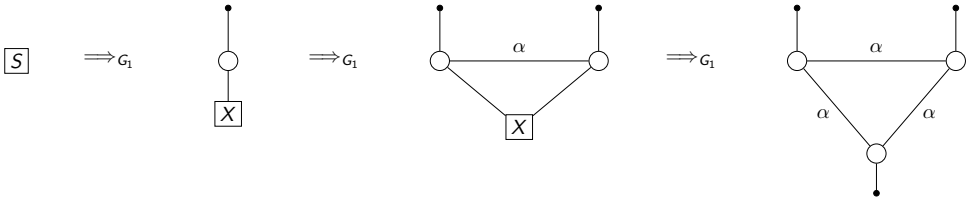


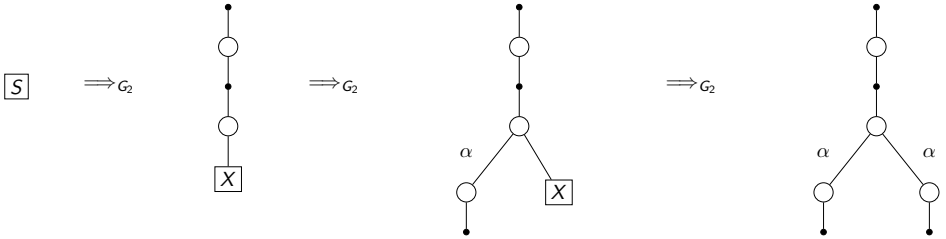
B-ESG rewrite pattern

Example



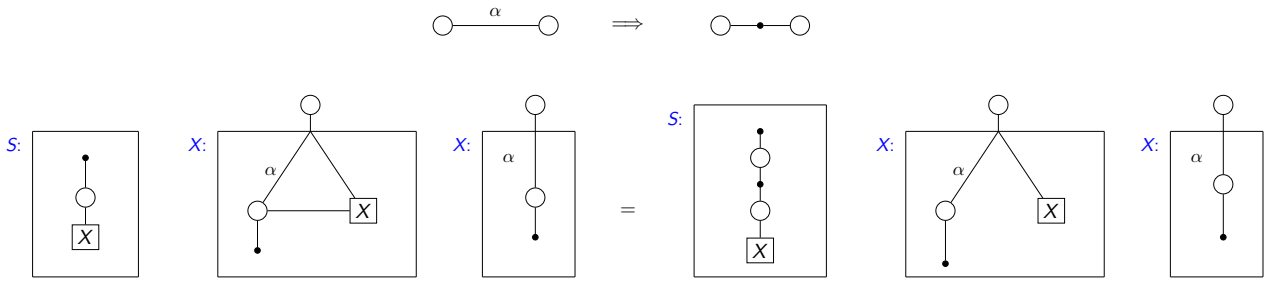
- Instantiation :



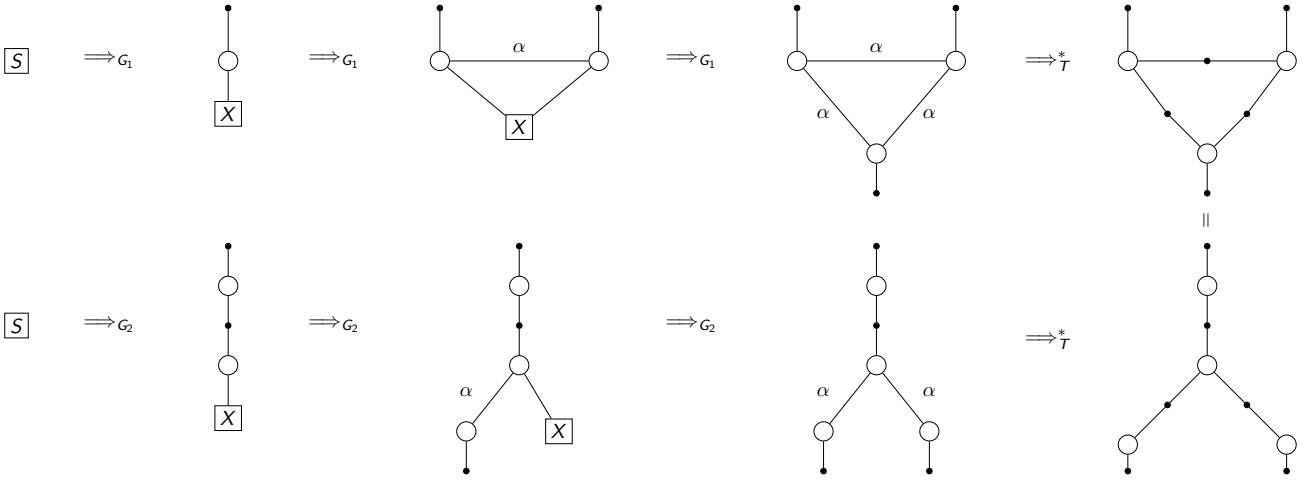


B-ESG rewrite pattern

Example

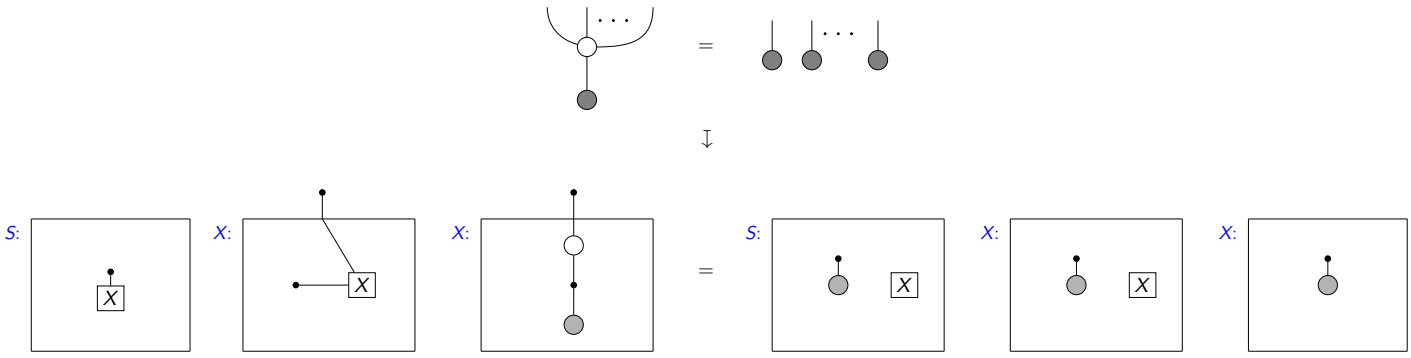


- Instantiation :



Obtaining new equalities

- We can encode infinitely many equalities between string graphs by using B-ESG rewrite patterns
- Thus B-ESG rewrite patterns can be useful for encoding axioms schemas of a string diagram theory

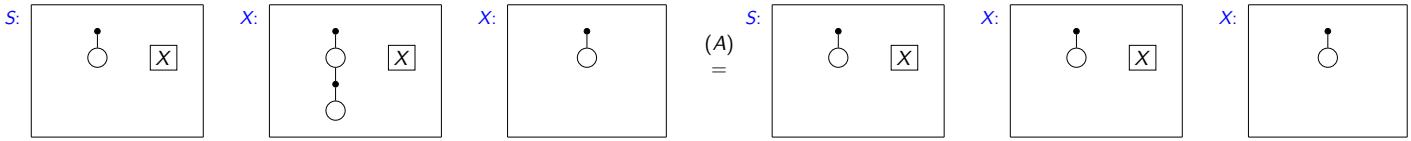


- Next, we show how to obtain new families of equalities from already existing ones in an admisible way with respect to the axioms of a theory using B-ESG grammars

Transforming B-ESG grammars

Example

Assume we have the axiom A :  = ◯ in our theory. We can then use this axiom to rewrite B-ESG grammars in an admisible way:



Theorem

Transforming B-ESG grammars in such a way induces a B-ESG rewrite pattern which is admisible with respect to the string graph rewrite rule used for rewriting.

Corollary

If B is a B-ESG grammar and (B_1, B_2) is a B-ESG rewrite pattern with B_1 match-exhaustive, then we can enumerate the pattern instantiations of (B_1, B_2) which induce an admisible rewrite pattern (B, B').

- Therefore, we can use B-ESG rewrite patterns to rewrite other B-ESG grammars in an admisible way.

Conclusion and Future Work

- Basis for formalized equational reasoning for context-free families of string diagrams.
- Identify more general conditions for B-ESG grammars such that the desired theorems and decidability properties hold
- Show that B-ESG rewrite patterns can be used to represent all !-graph rewrite rules (with trivial overlap)
- Introduce more powerful rewriting techniques for deriving B-ESG patterns
- Develop an induction principle for B-ESG grammars to allow for B-ESG pattern synthesis from basic string graph rewrite rules
- Consider Triple Graph Grammars as an alternative approach
- Implementation in software (e.g. Quantomatic proof assistant)

Thank you for your attention!