Specification of a diagrammatic reasoning tool

Ambroise LAFONT

July 26, 2024

This is a description of a tool that allows to reason graphically. The user interaction process consists of two stages:

1. an algebraic specification, that is, specification of an algebraic structure;
2. a construction mode to build elements in any model, based on some input data.

Example 0.1 (Proof that monomorphisms compose) *The algebraic structure is the structure of categories. The graphical reasoning mode would allow to construct the composite of two monomorphisms, and mark it as a monomorphism.*

Contents

1 Algebraic specification

The algebraic specification consists in two parts:

- A sort specification, that specifies what are the different sorts of elements available in the algebraic structure.
- A rewriting system specification, that is, a list of rewriting rules.

Example 1.1 *Continuing Example 0.1, the sort specification would be a list of fours sorts: objects, morphisms, commutative triangles, and monomorphisms. One rewriting rule would consist in completing any composable pair of morphism can be completed to a commutative triangle.*

From the user perspective, the algebraic specification mainly consists in drawing diagrams, where certain elements are visually "distinguished" (in this document, we will use the red color, but it could be something else).

1.1 Sort specification

The sorts are specified one after the other. Each sort is specified with a diagram, where the undistinguished elements represent the *dependencies* of the sort, while the distinguished elements determine how this new sort should be represented.

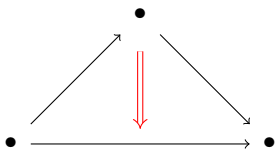
Example 1.2 *Let us detail the sort specification of Example 1.1.*

Objects *The sort has no dependencies, and we want to represent them as vertices, so the diagram would be:*

Morphisms *The sort has two dependencies: the source and target object. We want to represent a morphism as an arrow, so the diagram is:*

Monomorphisms *The sort depends on a morphism, and we want to represent it as an additional "monic" mark on the arrow.*

Commutative triangles *The sort has dependencies triangles. We can represent a commutative diagram as a 2-cell.*



1.2 Rewriting rules

1.2.1 First-order rules

A rewriting rule is specified as a *rewriting diagram*, where the non distinguished part is the pattern. The idea is that when this pattern is found in the workspace (we call this a *match*), the workspace can be extended with the distinguished part.

Remark 1.1 *Intuitively, the rewriting diagram is a rule that says: "if you see these (non distinguished) elements, you can add those (distinguished) elements". Logically, it is a formula* $\forall..\exists...$

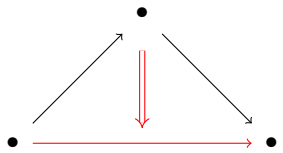
A rewriting diagram may also include equality tags between elements with the same dependencies (e.g., between two morphisms with the same source and target): those elements will be merged in the extended workspace.

Remark 1.2 *Logically, it is a formula* $\forall..\exists..P$ *where* P *is a cunjunction of equalities between variables introduced by* $\forall$*.*

Remark 1.3 *The unique existence property* $\forall\vec{x}\exists!\vec{y}..P(\vec{x})$ *is equivalent to the conjunction of* $\forall\vec{x}\exists\vec{y}P(\vec{x})$ *and* $\forall\vec{x}/P\forall\vec{y}\forall\vec{y}'\vec{y}=\vec{y'}$*, where* $\vec{x}/P$ *is a smaller vector of variables (taking into account the equations in* P*), and thus can be represented by two rewriting diagrams. The software should be able to automatically turn a unique existence rewriting diagram into two rewriting diagrams.*

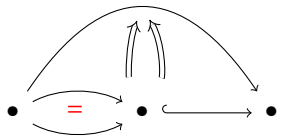
Example 1.3 *Let us detail the first-order rewriting rules of Example 1.1.*

Composition



(Comp)

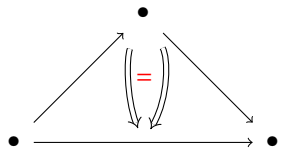
Monomorphic property



(Mono)

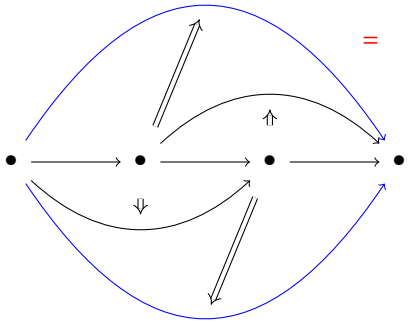
Note that there is an ambiguity here about the dependencies of the 2-cells. The interface should give a way to make them visually explicit, e.g., using a color convention as follows:

Note that we could have added for convenience[1] *an equality tag between 2-cells, but it would be more principled to have a seperate rewrite rule to make commutative triangles irrelevant:*



Alternatively, we could have a built-in mechanism to handle proof-irrelevant sorts.

Associativity of composition *We use the blue color for the specifying which elements are related by the equality tag.*



(Assoc)

Question 1.1 *What about identities? custom labelling of nodes?*

The user has two ways of creating new admissible rewriting rules, that is, rules that are valid in any model of the original set of rewrting rules.

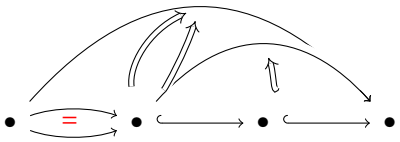
Chaining Chaining rewriting rules;

Pruning Removing distinguished elements from a rewriting diagram.

Example: the composition of two monomorphisms is monomorphic In this section, we show that the previous set of rewriting rules is enough to build the following rewriting rule, that shows that the composition of two monomor-

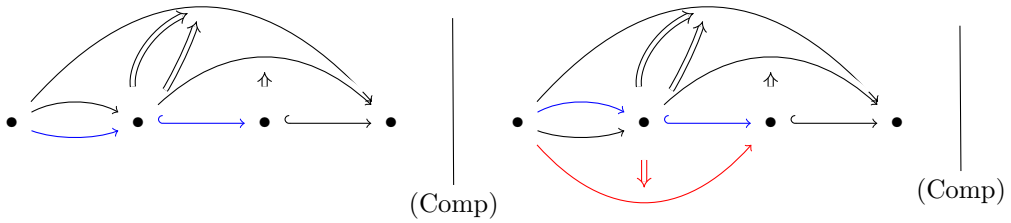
[1] This is to make the output diagram smaller.

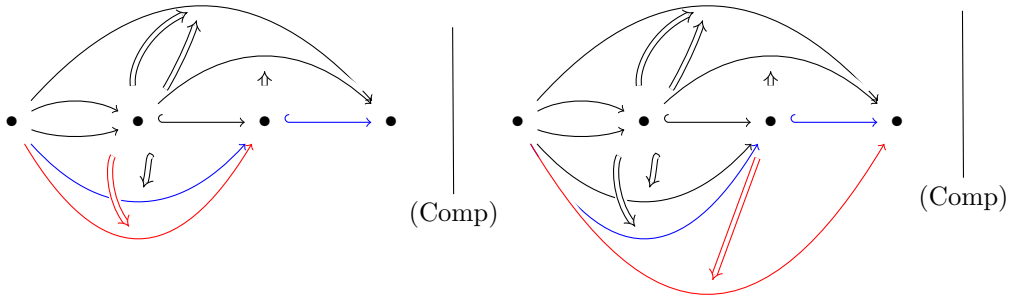
phisms is monomorphic.

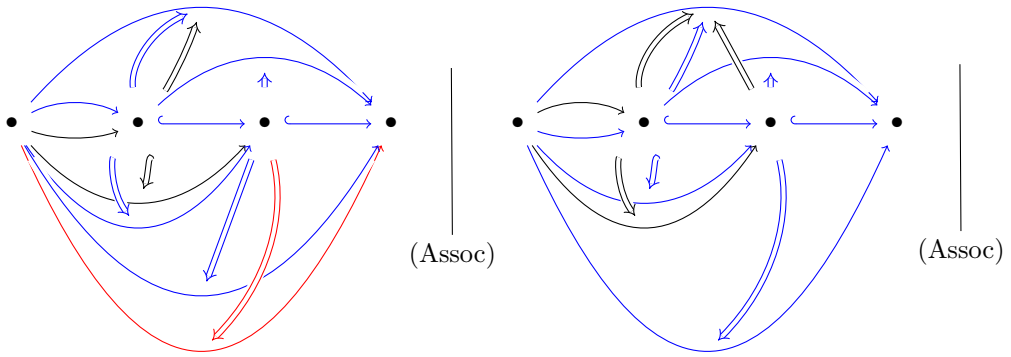


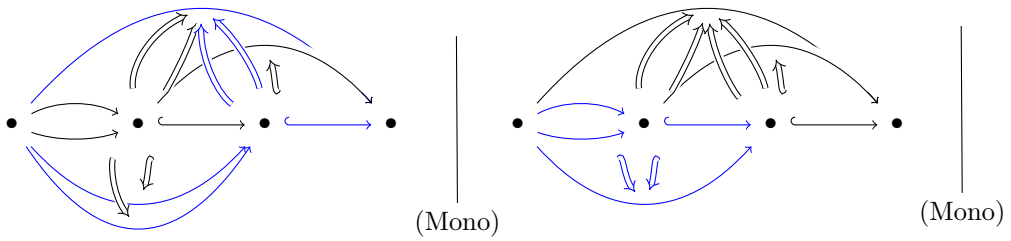
(1)

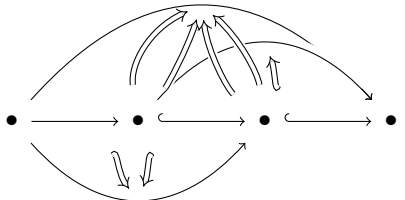
We start by chaining rewriting rules as follows. We use blue to denote a match, and the red color to show the added part, and violet (or orange) for the match and added part.


(Comp)
(Comp)


(Comp)
(Comp)

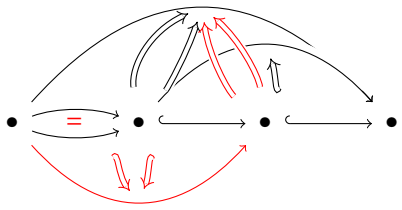

(Assoc)
(Assoc)


(Mono)
(Mono)



Question 1.2 *Clearly, this is too complicated... How can we make it simpler? See Appendix A for some attempts. We should offer a way to focus on some subpart of the diagram (for example, elements that depends on some selected parts), hide some stuff.*

This sequence is compactified by the software into a single rewriting rule:

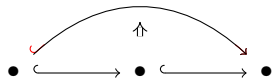


(2)

By removing distinguished elements, we recover the above rewriting rule (1).

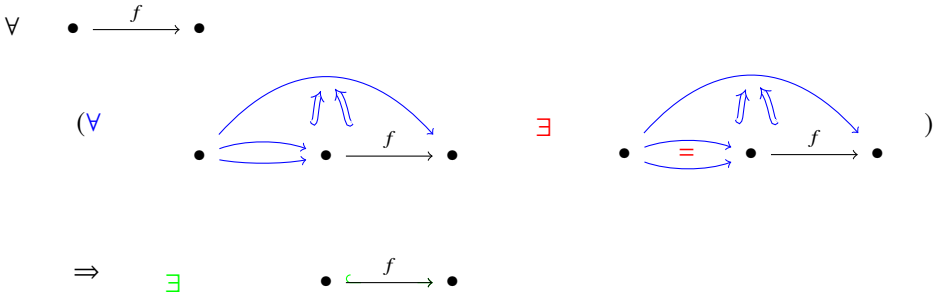
1.2.2 Second-order rewriting rules

The admissible rewriting rule (1) shows that the composition of two monomorphisms is monomorphic. This makes following rewriting rule legitimate:



Unfortunately, there is no way to deduce this rule using the chaining and pruning mechanisms.

The formula that we would like to account is of the following shape:



It does not fit the format $\forall..\exists..P$ of our previous rewriting rules (see Remark 1.2).

This motivates what we call *second-order rewriting rules*, which handles formulae of the shape above, where the second line can occur many time. That is,we handle formulae of the shape $\forall(\forall\exists..)^*\exists...$

1.2.3 Third-order order rules and beyond

One may worry that second-order rules are not strong enough to capture all the possible rewriting rules that we want. What if we want additional nesting of quantifiers, like the third-order rewriting rule below?

$$\forall x.\, (\forall y.(\forall z \exists w.P) \Rightarrow \exists..Q) \Rightarrow \exists..R \tag{3}$$

In fact, we can "emulate" them with second-order rewriting rules with a trick: we can introduce a new sort, and use it to represent the inner quantifiers[2].

Let us illustrate this trick with Equation (3). We introduce a new sort S depending on x, y such that

$$\forall xy\, (\exists s : S(x,y) \;\Leftrightarrow\; \forall z \exists w P) \tag{4}$$

so that Equation (3) is equivalent to the following second-order rule:

$$\forall x.\, (\forall y (s : S(x,y)) \exists..Q) \Rightarrow \exists..R$$

Now, let us explain how we can enforce the equivalence of Equation (4). For the left-to-right implication, a first-order rewriting rule is enough:

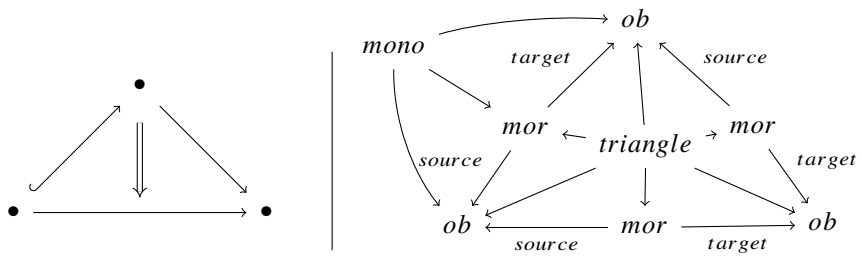
$$\forall xy (s : S(x,y)) z\ \exists w.P$$

For the reverse implication, we can write a second order rewriting rule:

$$\forall xy (\forall z \exists w.P) \;\Rightarrow \exists s : S(x,y)$$

2 Internal representation

A diagram is stored as a graph: each element in the graph is a node, and each dependency is an edge. For example, the following diagram below left is represented as the graph below right:



Looking for a match can be done at the level of this graph representation.

[2]Intriguingly enough, github copilot correctly guessed the end of this sentence, after I wrote "emulate them".

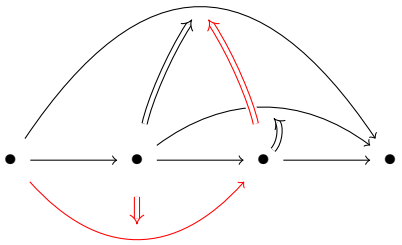
3 Semantics

A sort specification corresponds to a finite direct category. A diagram with distinguished elements corresponds to a natural transformation between two functors from this category to the category of sets.

A Attempts to simplify the proof that monomorphisms compose

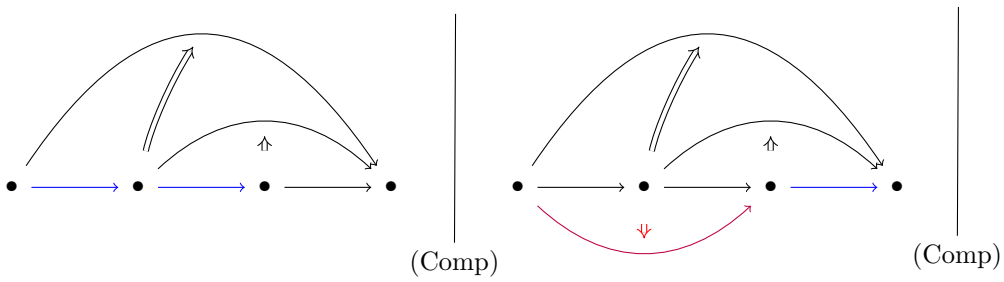
The last attempt is pretty convincing; it we require tactics.

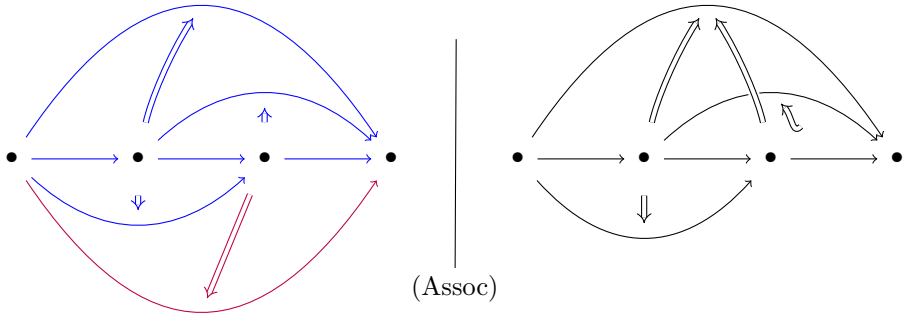
Simplifying the above example First we can construct the following associativity rule.



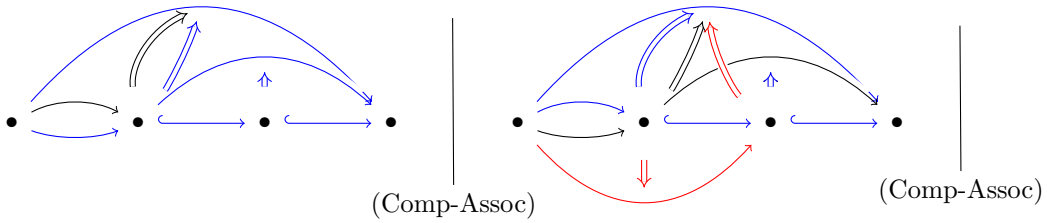
(Comp-Assoc)

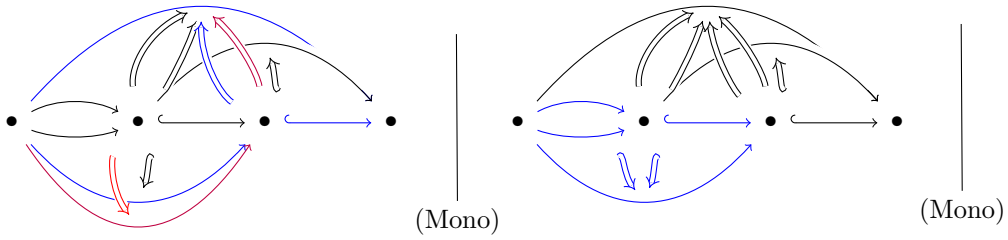
Let us derive it more carefully.

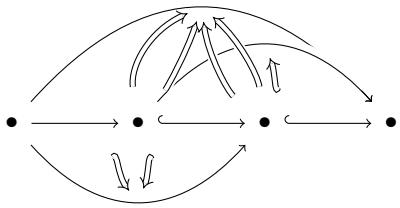




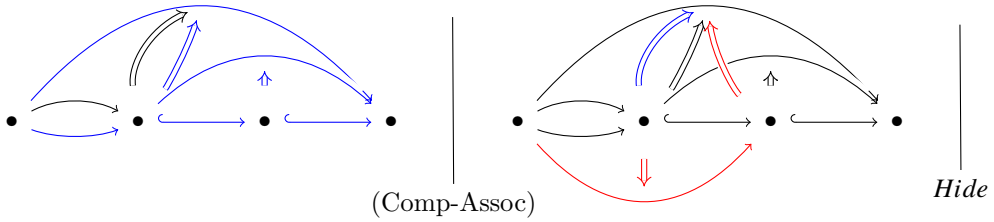
We present a second version of the long proof of (2).

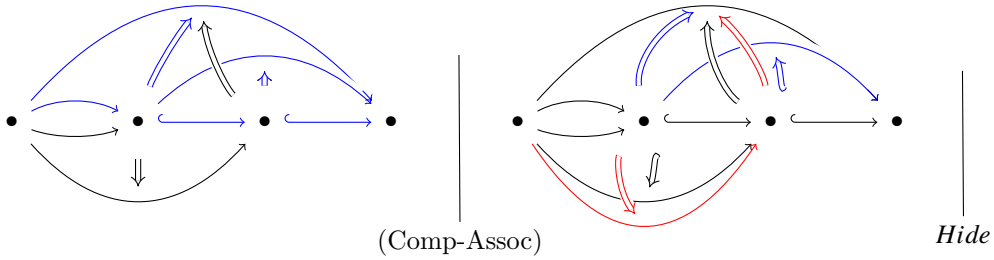

(Comp-Assoc)
(Comp-Assoc)

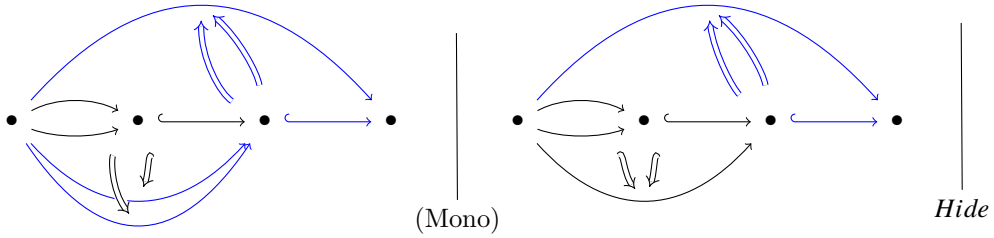

(Mono)
(Mono)

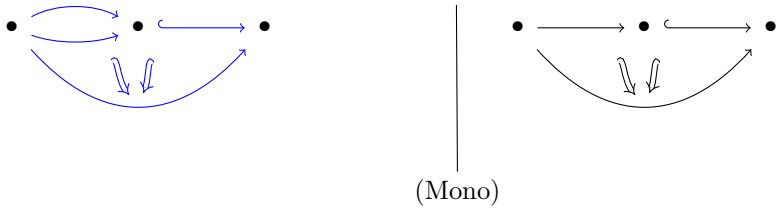


A version with a hide primitive

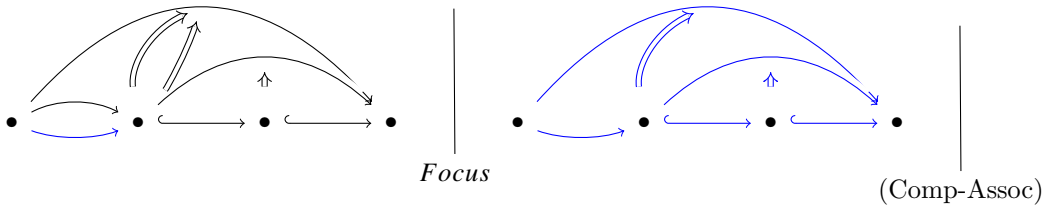


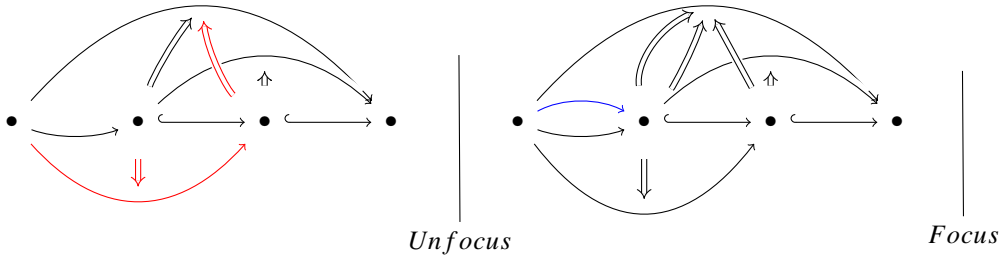


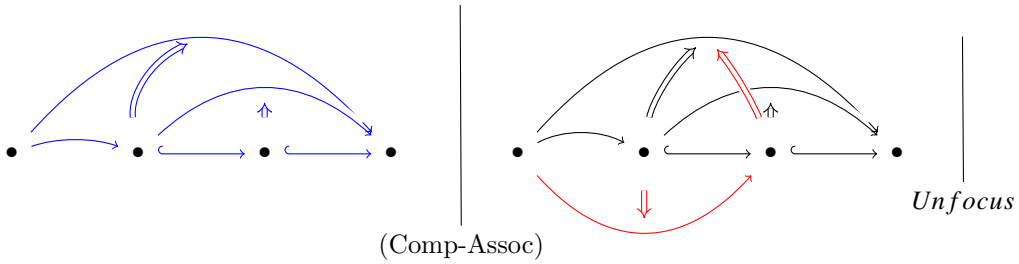


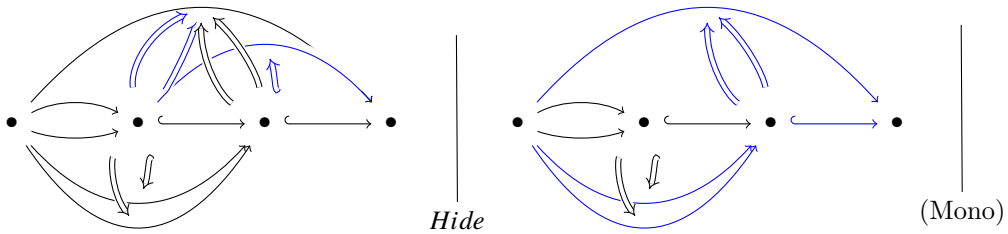


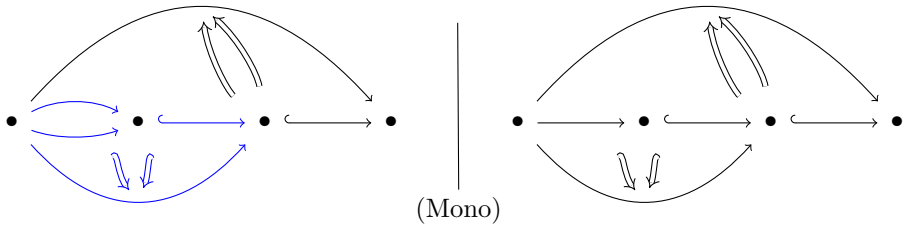
A version with hide and focus primitives The idea is that we keep all the elements that depend on the selected elements, and hide the rest.


Focus
(Comp-Assoc)


Unfocus
Focus


(Comp-Assoc)
Unfocus


Hide
(Mono)


(Mono)

Maybe names could help, to remember stuff...

A version with hide tactical We assume that we have a tactic Apply-Hide that takes as input a rewriting rule, and a set of elements to hide. It will apply the rewriting rules greedily on the selected elements, and then hide the objects.