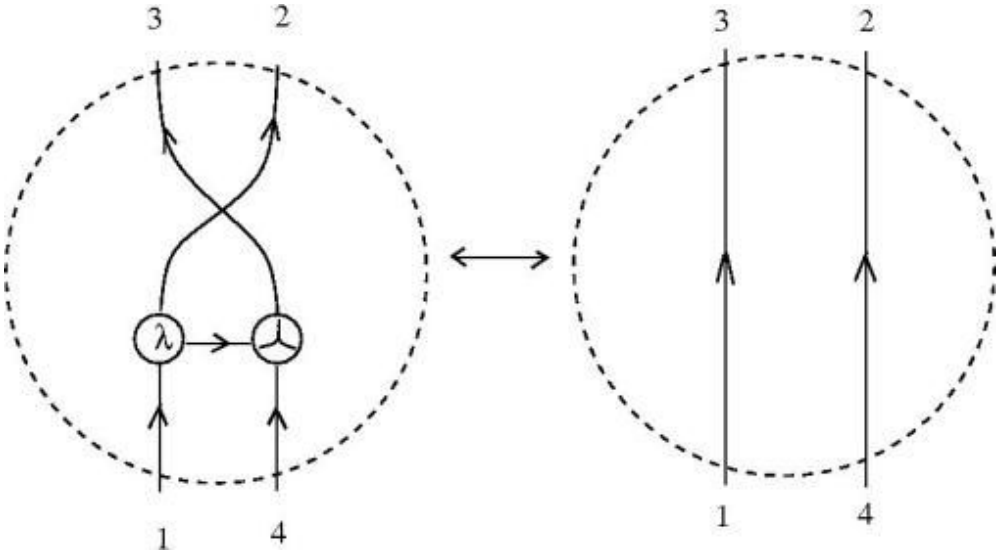
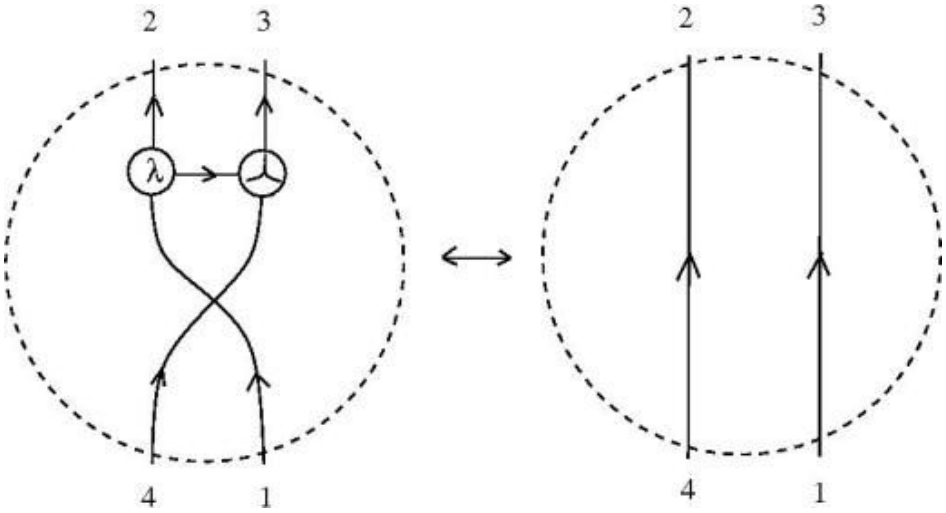
GLC macros and some computations

[source: http://chorasimilarity.wordpress.com/2012/09/06/graphic-beta-rule-as-braiding/]

Here are two equivalent depictions of the graphic beta move, coming from different choices of 1-3, 4-2 decorations. We may see the graphic beta move as

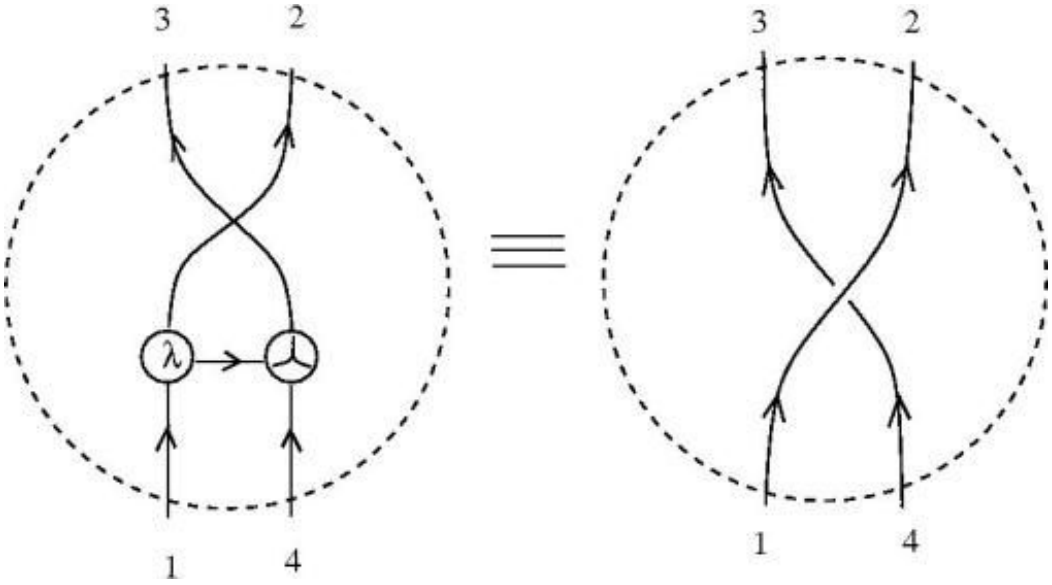


but also as

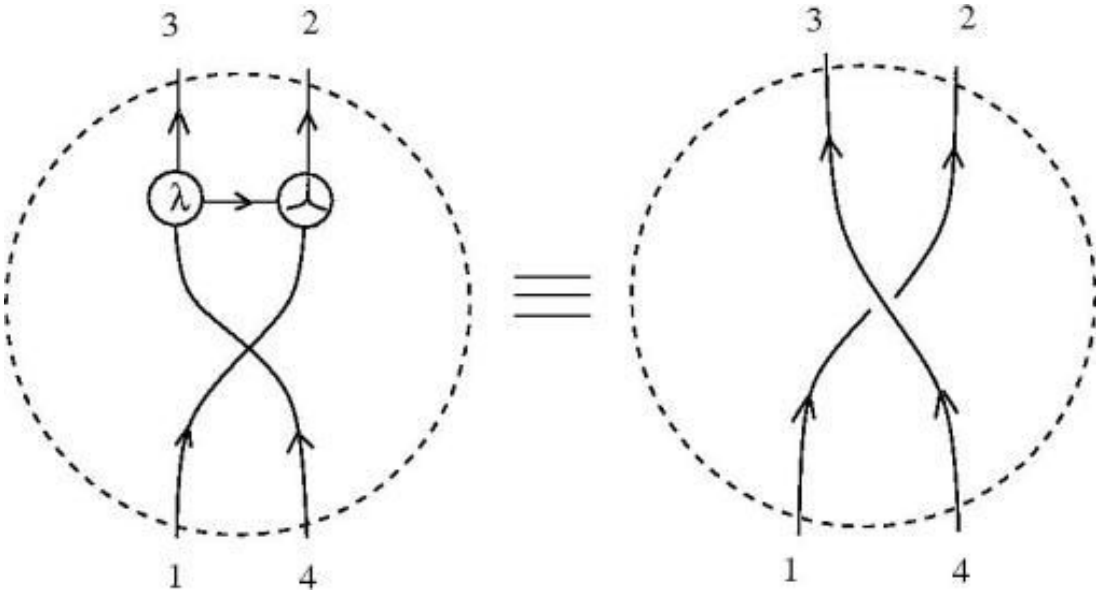


Let me then make some ***notations*** of the figures from the left hand sides of the previous diagrams. Here they are: for the first figure we introduce the

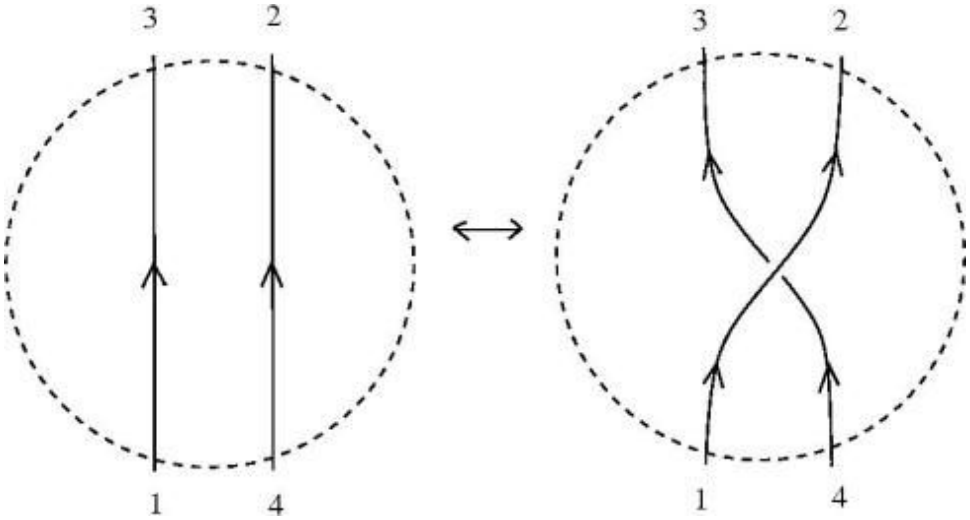
“crossing notation”



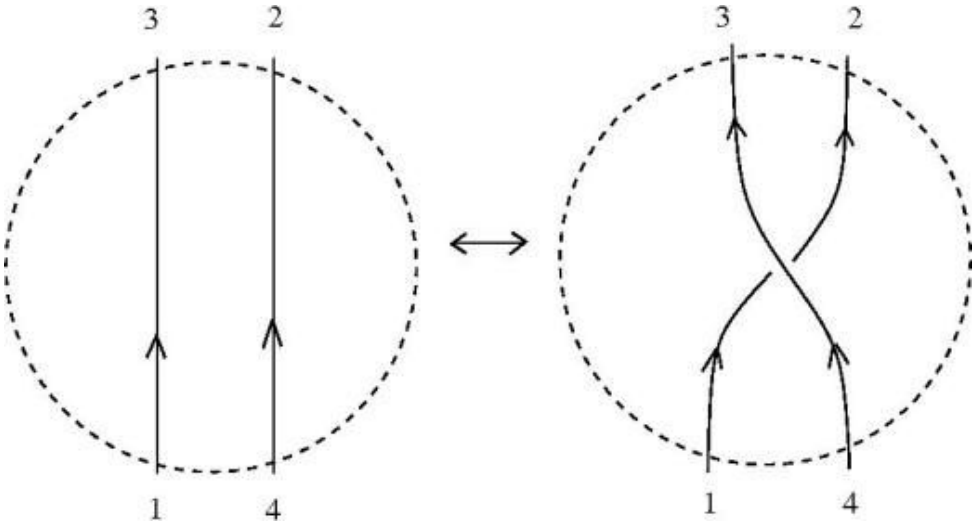
and for the second figure the “opposite crossing notation”:



With these notations the graphic beta move may be see as this ***braiding operation***

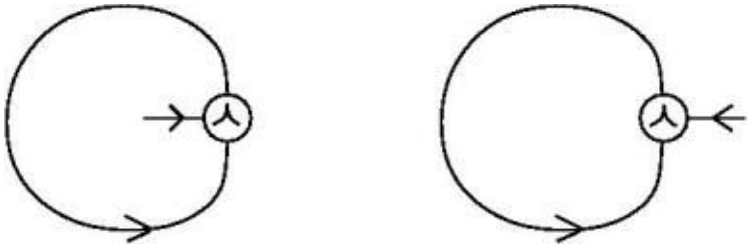


or as this other braiding operation



[source: http://chorasimilarity.wordpress.com/2012/11/14/combinators-and-stickers/]

There are two stickers, they are graphs in GRAPH with the following form:

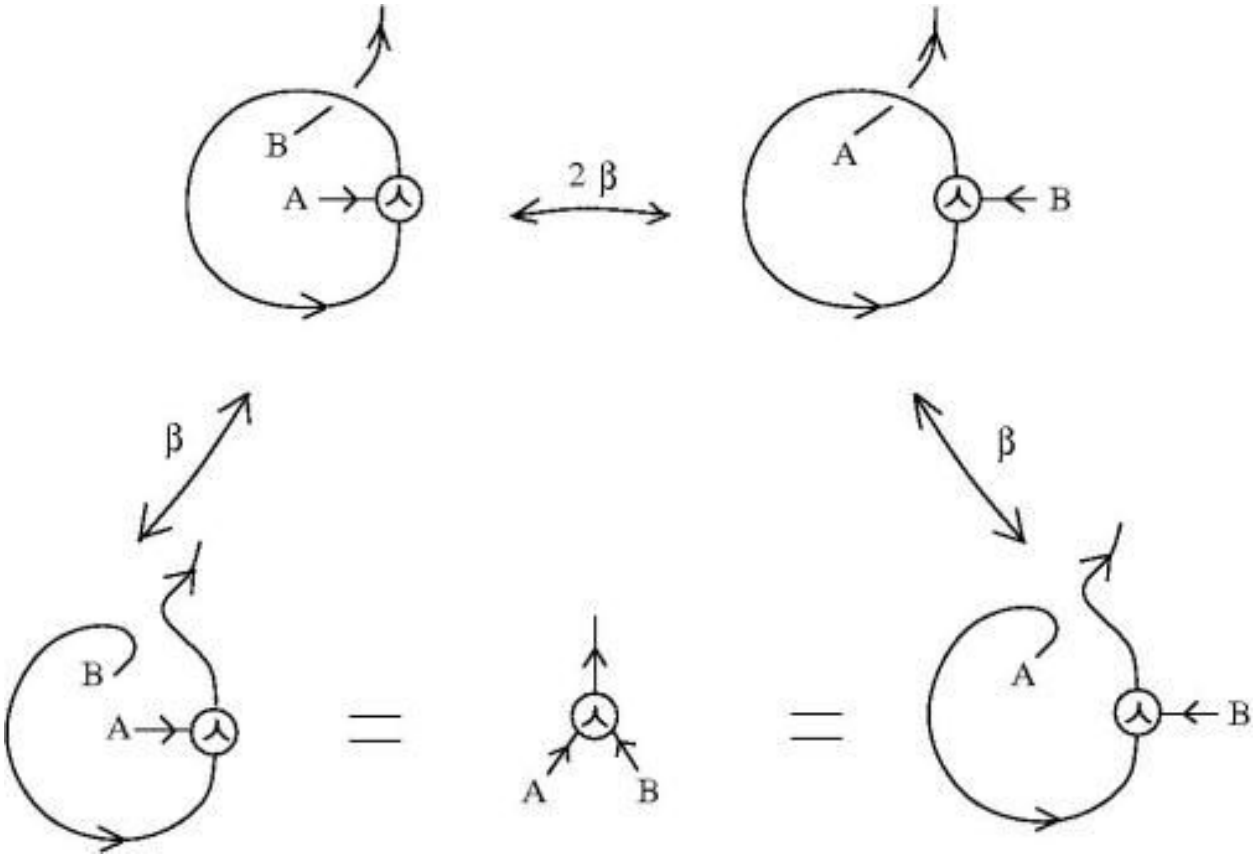


I call them “stickers” because they behave like devices which concentrate the attention at the region encircled by the closed loop.

Think about a sticker with nothing written on it. This is a thing which you can stick in some place and you can write something on it.

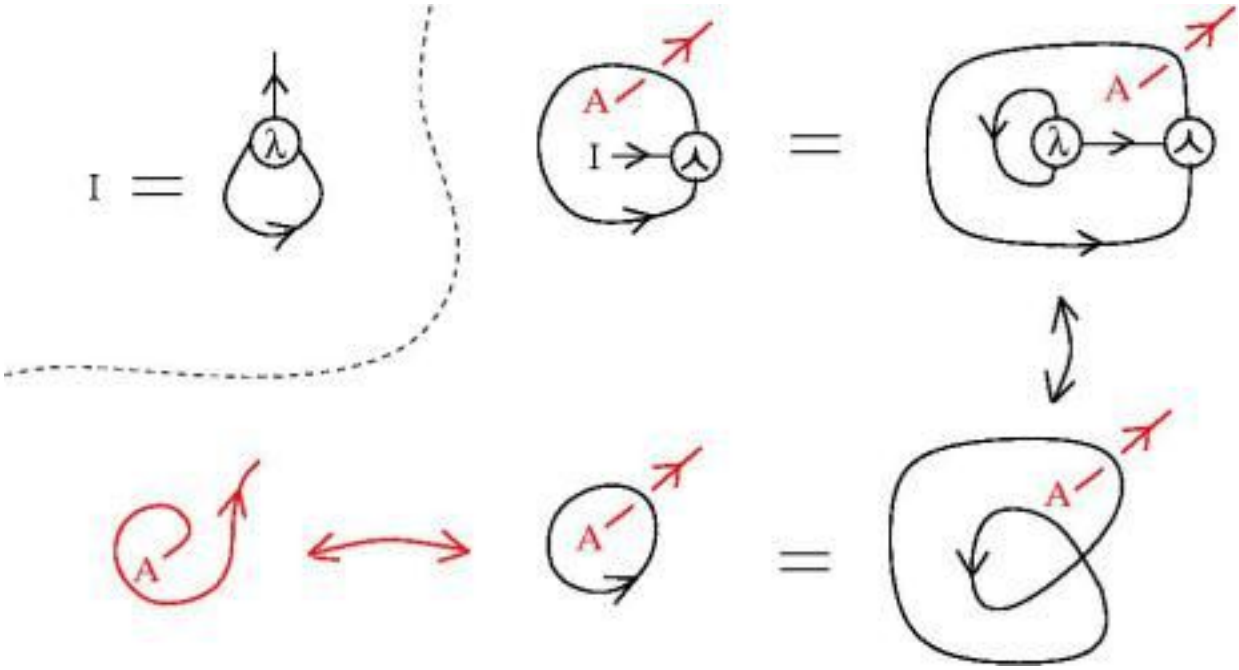
Likewise, take any graph in GRAPH with a marked OUT leaf (for any example take any combinator in graphic lambda calculus form, which always has an unique OUT leaf) and connect it with one of the stickers (i.e. “write it on the sticker”). The operation analoguous to sticking the written sticker on something is to cross the sticker connected with a combinator with another combinator, where “crossing” refers to the knot diagrams sector, with its graphic lambda crossings.

Look how the stickers behave:



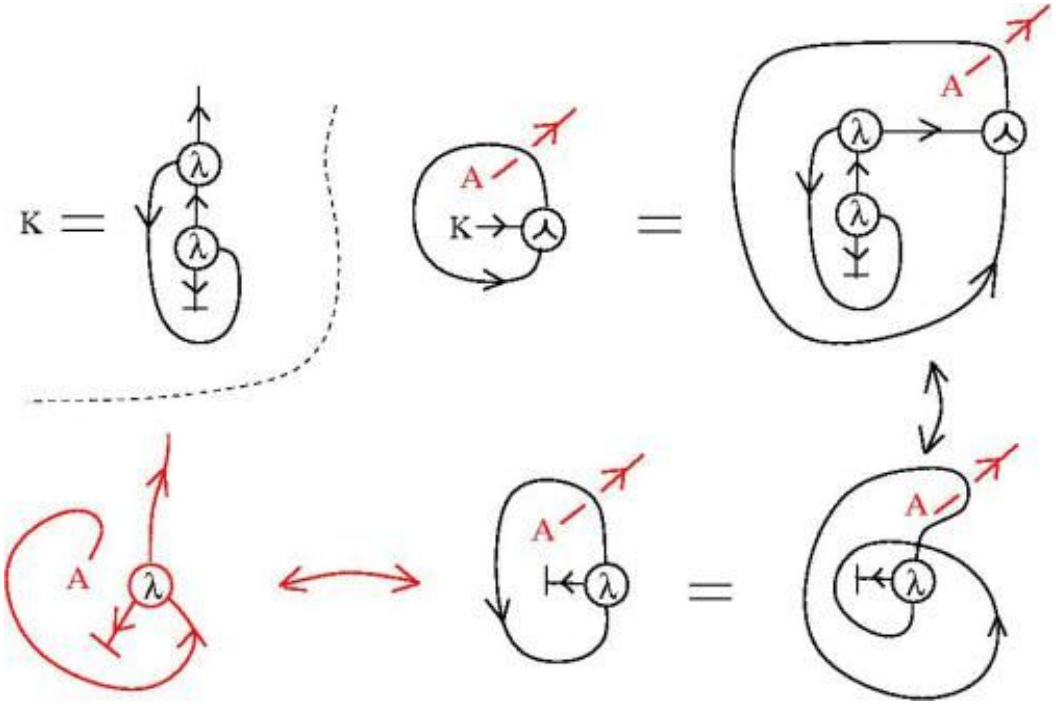
They may be converted one into another (by way of graphic beta moves). Moreover, they transform the application operation gate into a crossing!

Let us see the behaviour of some combinators connected with stickers. Take the combinator $I = \lambda x.x$. In the next figure, we see it in the upper left corner, in graphic lambda formalism. (The figure has two layers. One may read only what is written in black, considering after what happens if the red drawings are added.)



In black, we see that the graph of I connected with a sticker transforms by a graphic beta move into a loop. If we add the red part, we see that the graph transforms into a loop with a crossing with a graph A. If we perform a second graphic beta move (in red) then we get A. This corresponds to the combinatory logic relation $IA \to A$.

Likewise, let's consider the combinator $K = \lambda xy.x$, with associated graph described in the left upper corner of the next figure.

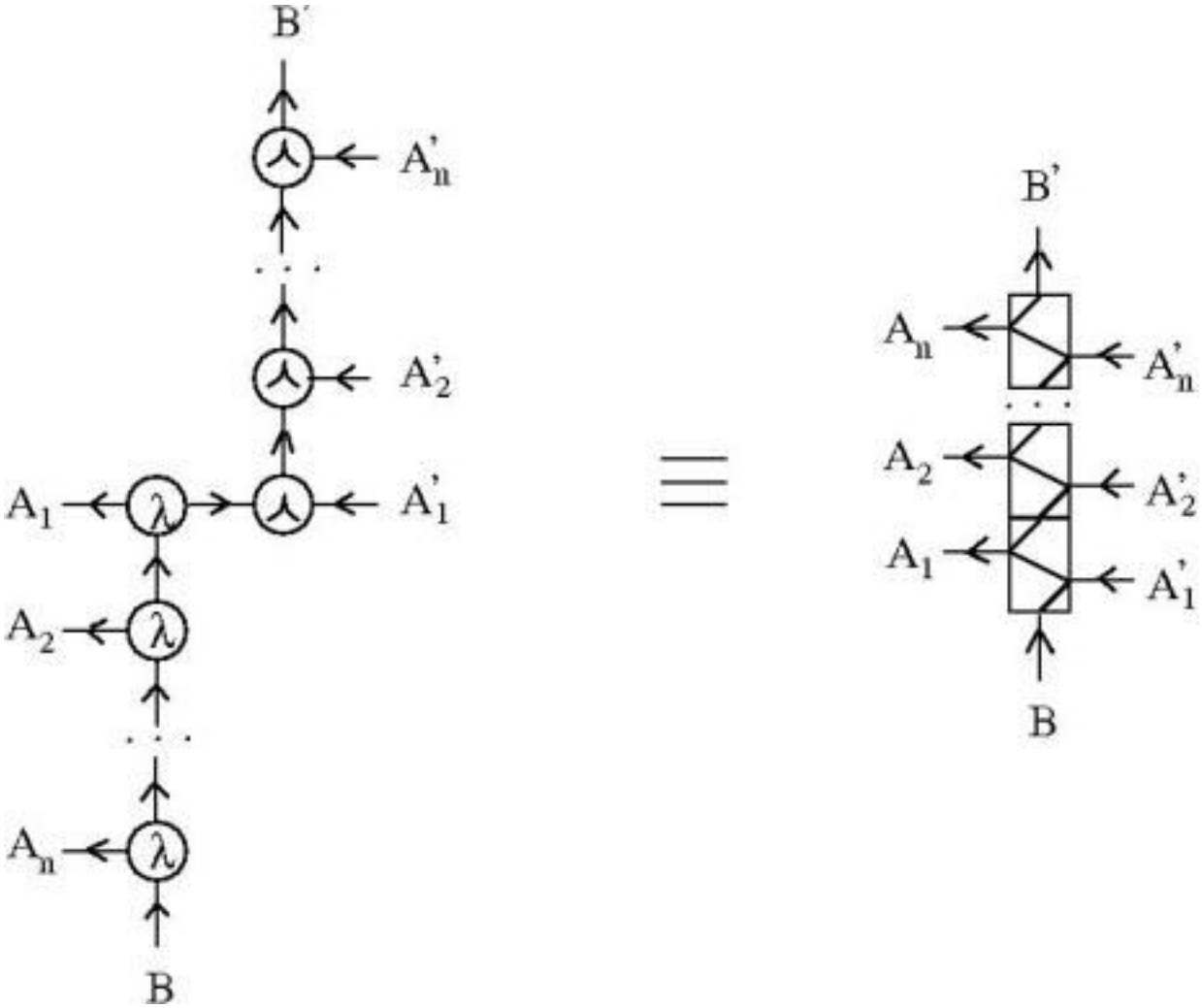


With the added red part, it corresponds to the relation $KA \to \lambda y.A$ with y a fresh variable for A.

[source: http://chorasimilarity.wordpress.com/2012/11/12/the-zipper-macro-and-zipper-moves/]

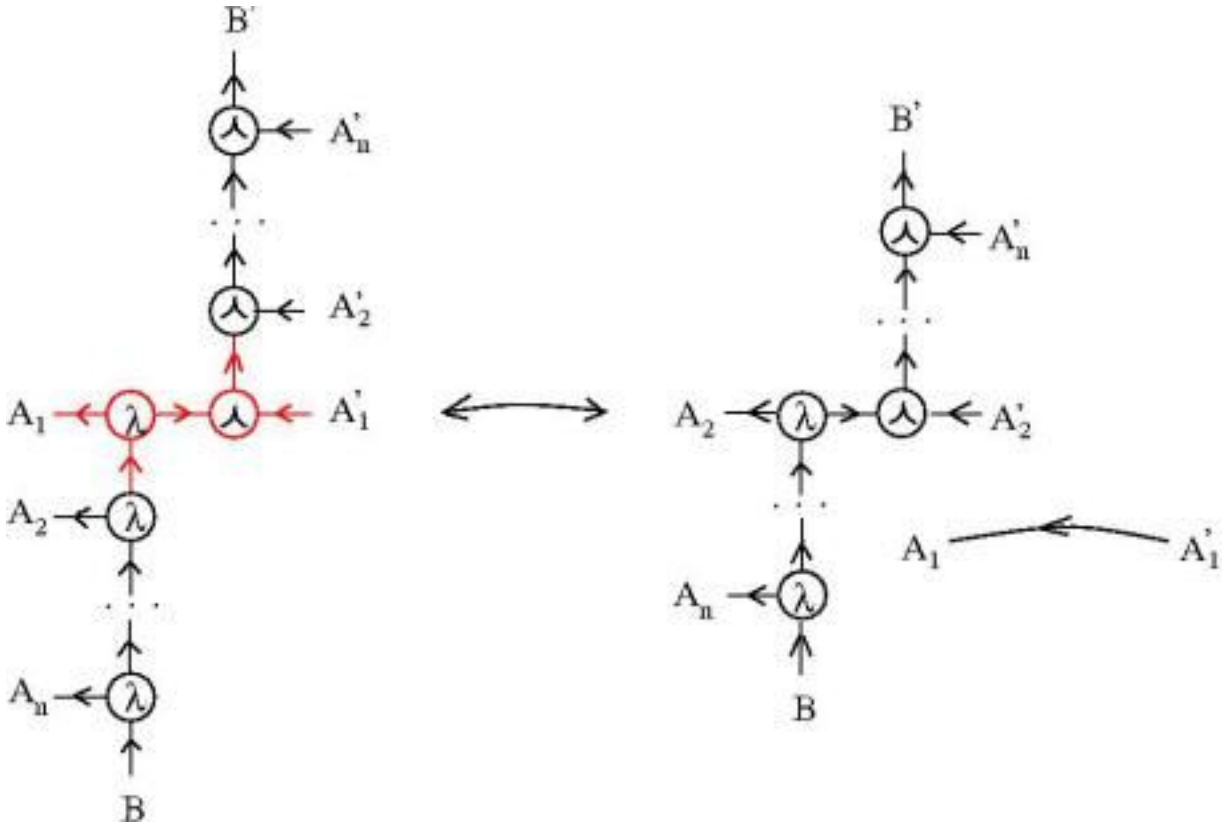
Another interesting macro (over graphic lambda calculus) is the zipper, together with its associated zipper moves.

Let's take $n \geq 2$ a natural number and let's consider the following graph in $GRAPH$, called the ***n-zipper***:

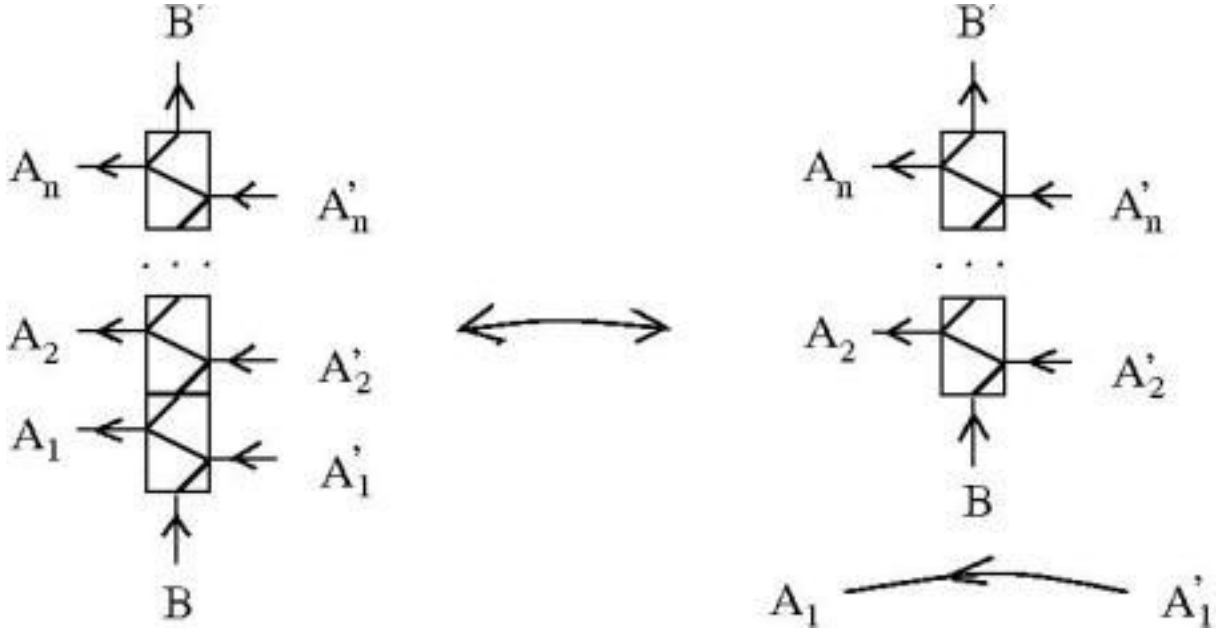


At the left is the n-zipper graph; at the right is a NOTATION for it, or a macro. We could as well take $n = 2$, with obvious modifications of the figure, so the 2-zipper exists. Even $n = 1$ makes sense, but the 1-zipper is kind of degenerate, see later.

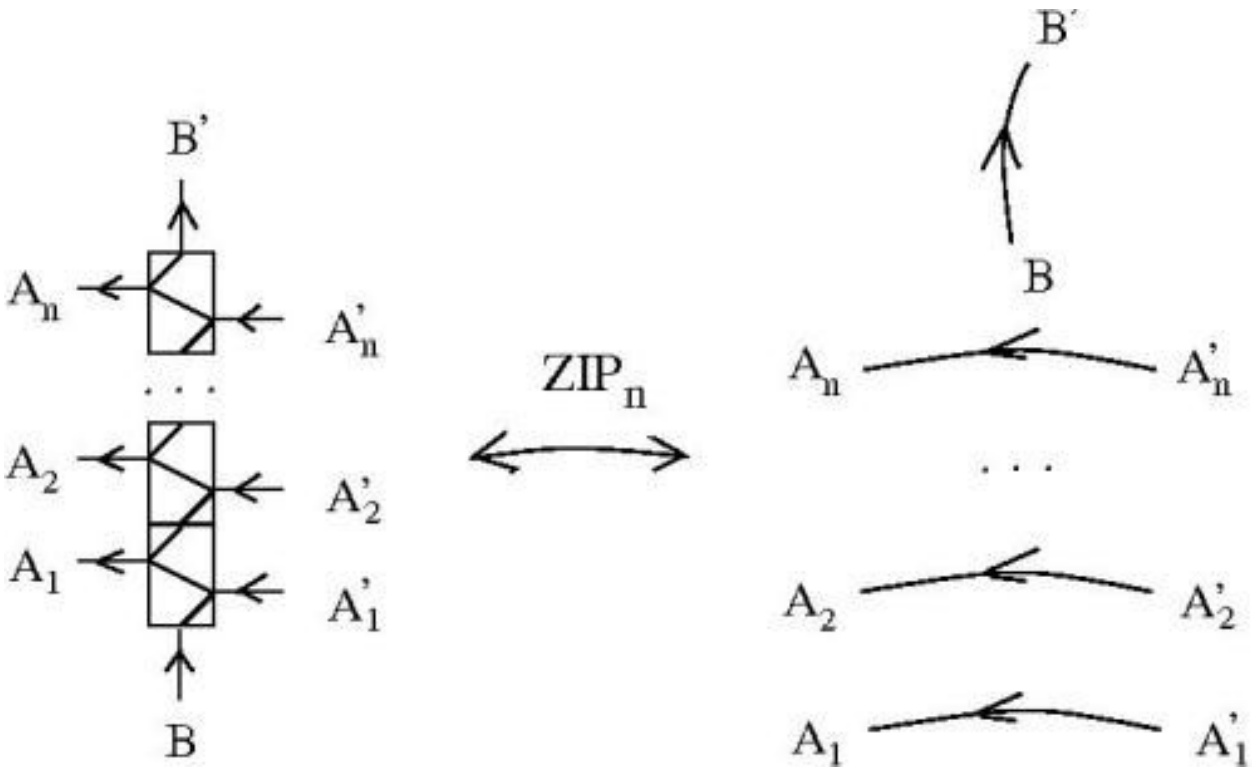
There is a graphic beta move which we can perform on the n-zipper graph. In the following picture I figured in red the place where the graphic beta move is applied.



In terms of zipper notation this graphic beta move has the following appearance:



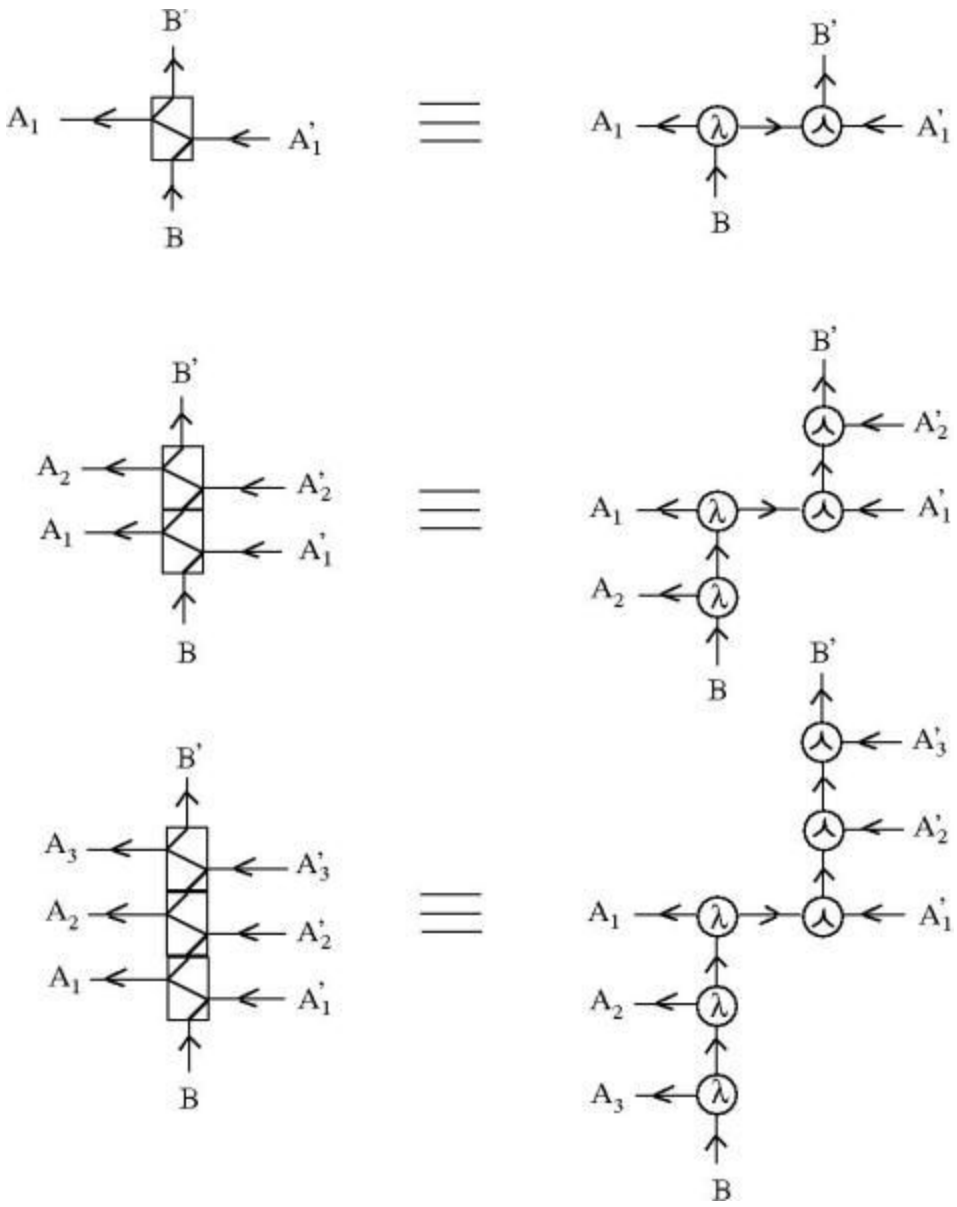
We see that a n-zipper transforms into a (n-1)-zipper plus an arrow. We may repeat this move, as long as we can. What is the result? A ***n-zipper move***:



The 1-zipper move, called ZIP_1 is just the graphic beta move, which transforms the 1-zipper into two arrows.

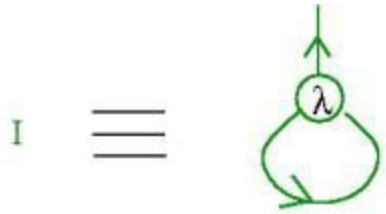
[source: http://chorasimilarity.wordpress.com/2013/02/09/combinators-and-zippers/]

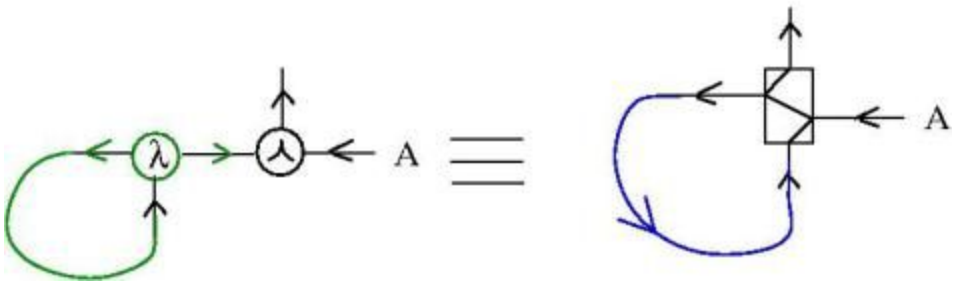
Zippers have been introduced here. In particular, the first three zippers are depicted in the following figure.



The combinator I has the expression $I = \lambda x.x$ and it satisfies the relation $IA \to A$, where $\to$ means any combination of beta reduction and alpha renaming (in this case is just one beta reduction: $IA = (\lambda x.x)\, A \to A$).

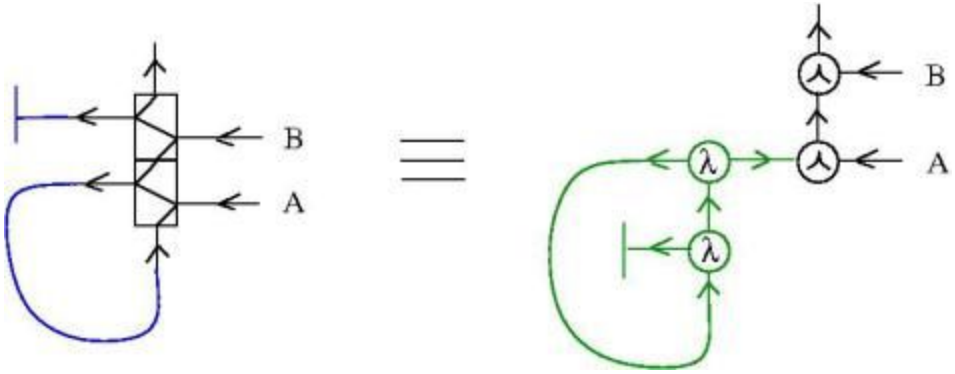
In the next figure it is shown that the combinator I (figured in green) is just a half of the zipper_1, with an arrow added (figured in blue).





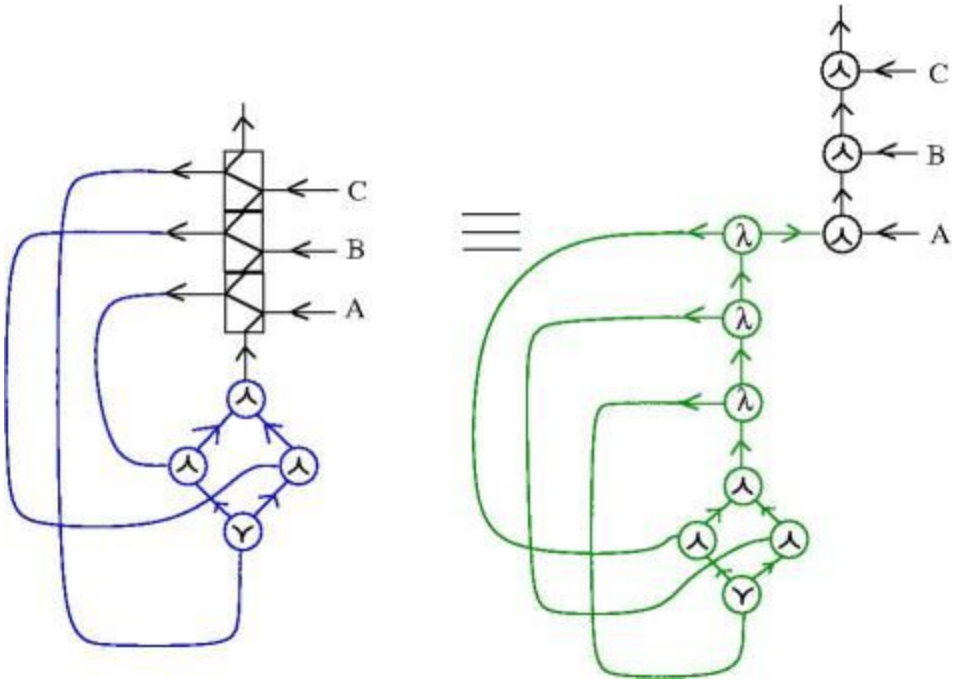
When you open the zipper you get A, as it should.

The combinator $K = \lambda xy.x$ satisfies $KAB = (KA)B \to A$. In the next figure the combinator K (in green) appears as half of the zipper_2, with one arrow and one termination gate added (in blue).

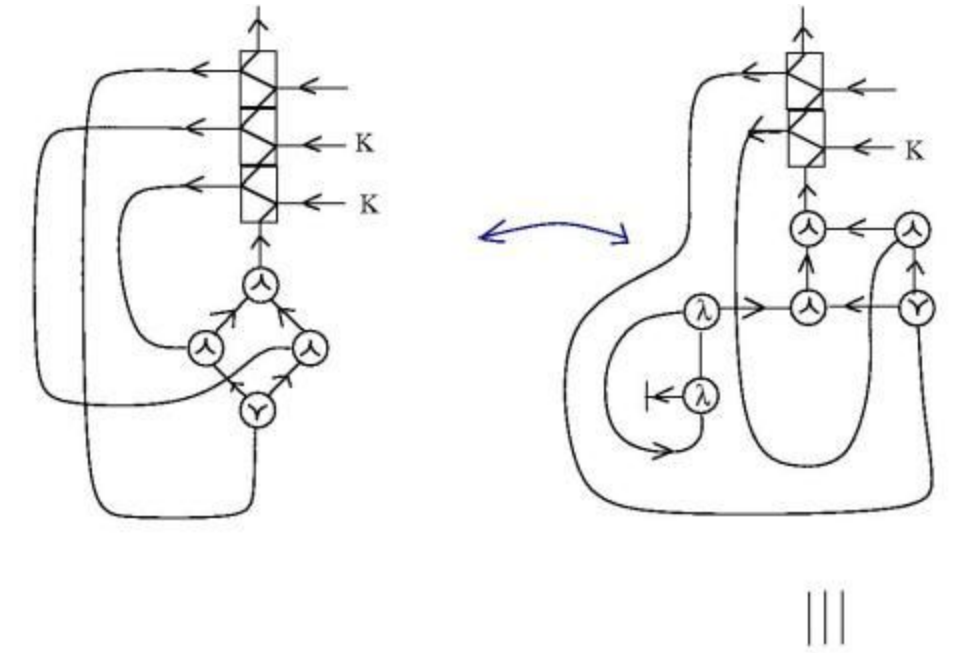


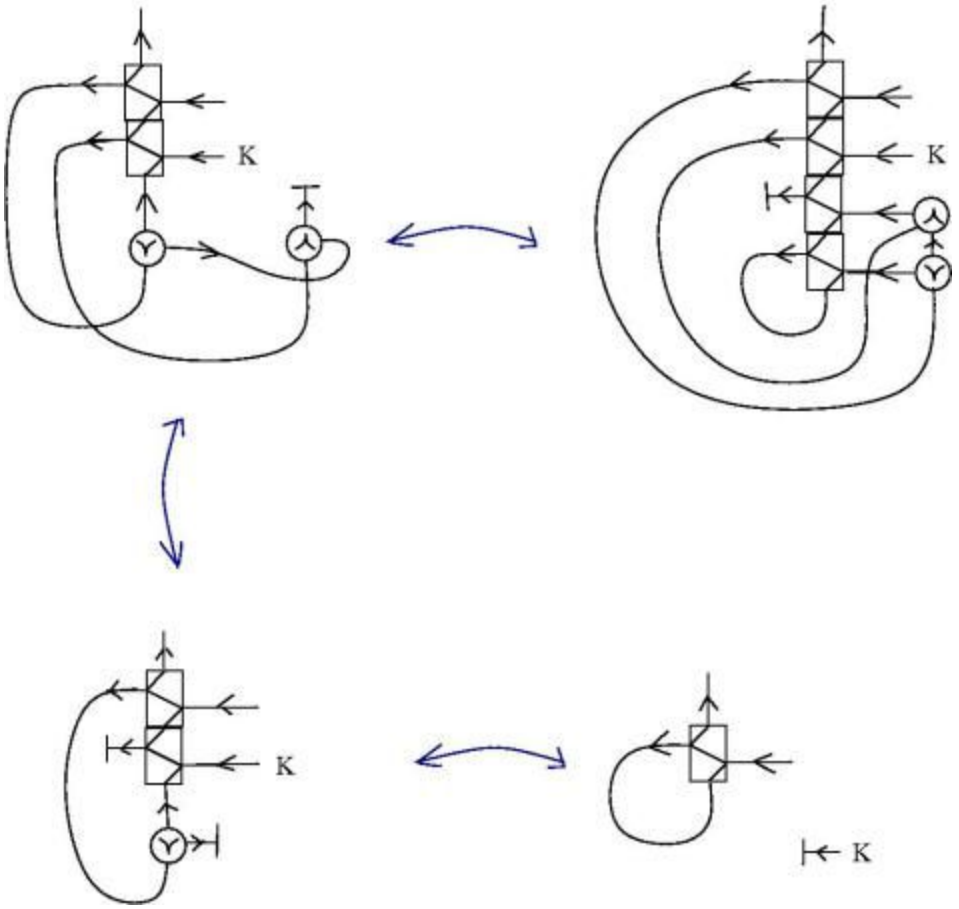
When you open the zipper you obtain a pair made by A and B which gets the termination gate on top of it. GLOBAL PRUNING sends B to the trash bin.

Finally, the combinator $S = \lambda xyz.((xz)(yz))$ satisfies $SABC = ((SA)B)C \to (AC)(BC)$. The combinator S (in green) appears to be made by half of the zipper_3, with some arrows added and also with a “diamond” added (all in blue). Look well at the “diamond”, it is very much alike the emergent sum gate from this post.



Here is for example the relation $SKK \to I$, as deduced in graphic lambda calculus, by using zippers:


K
K
K
λ
λ

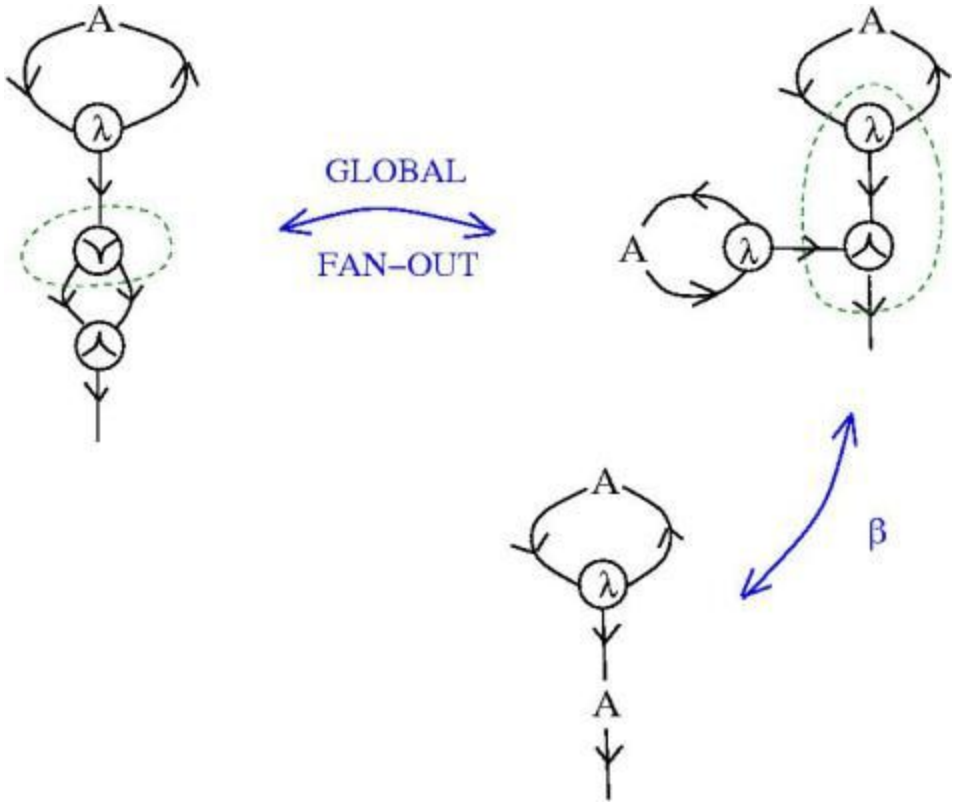

K
K
K
K

[source:
http://chorasimilarity.wordpress.com/2013/02/11/fixed-points-in-graphic-lambda-calculus/]

Let A be a graph in $GRAPH$ with one input and one output. For any graph B with one output, we denote by $A(B)$ the graph obtained by grafting the output of B to the input of A.

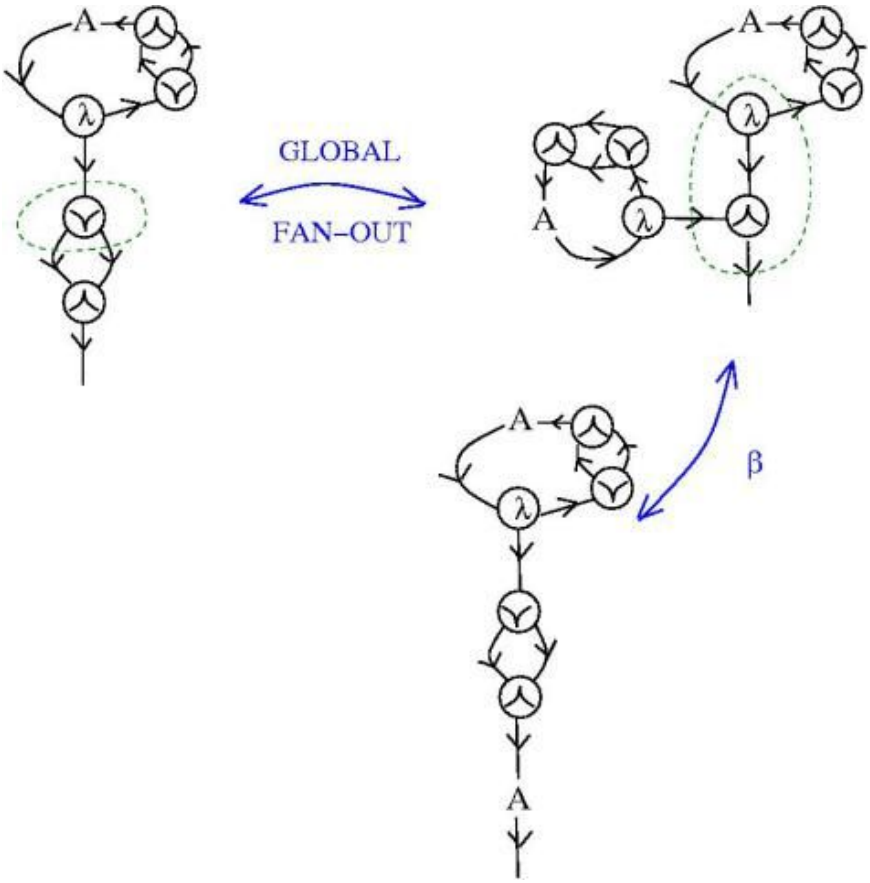
Problem: Given A, find B such that $A(B) \leftrightarrow B$, where $\leftrightarrow$ means any finite sequence of moves in graphic lambda calculus.

The solution is in principle the same as in usual lambda calculus, can you recognize it? Here is it. We start from the following:



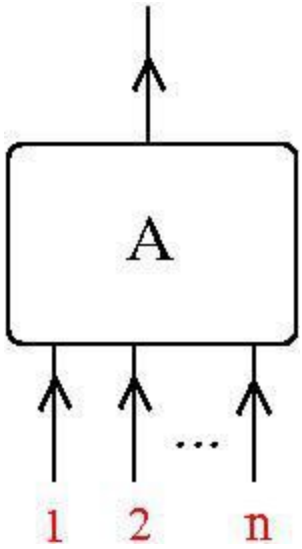
That's almost done. It suffices to do this:

to get a good graph B:

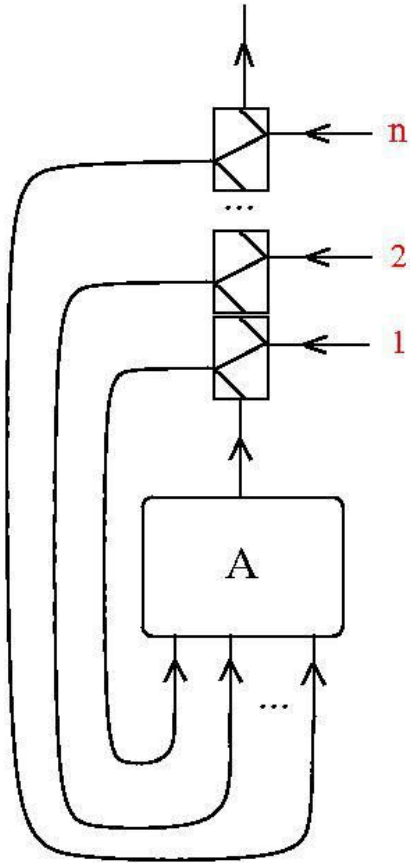


[source: http://chorasimilarity.wordpress.com/2013/03/13/currying-by-using-zippers-and-an-allusion-to-the-cartesian-theater/]

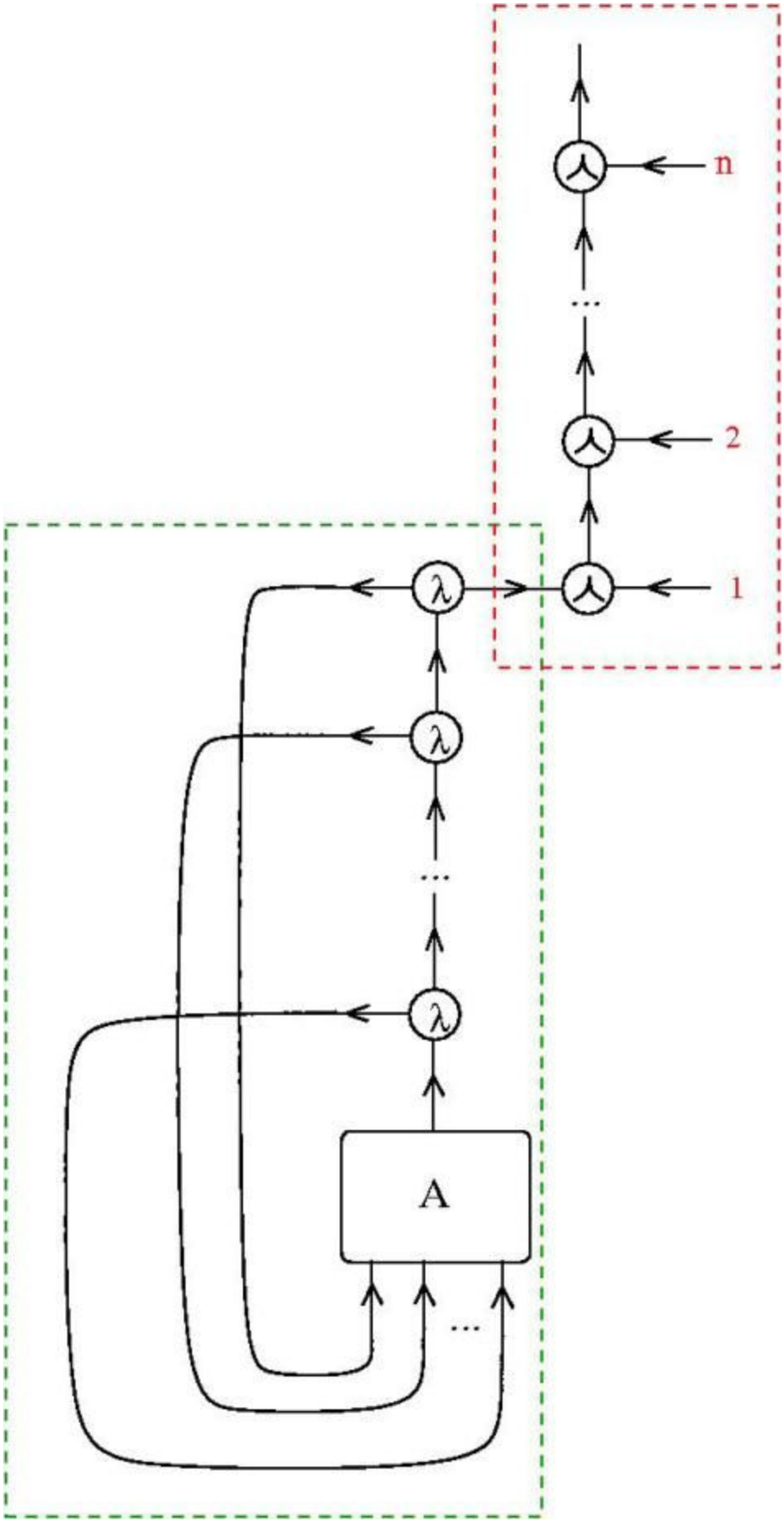
We have a graph $A \in GRAPH$ which has one output and several inputs. We want to curry it. For this we have to artificially give names to the inputs, i.e. to number them (notice that such a thing is not needed in graphic lambda).



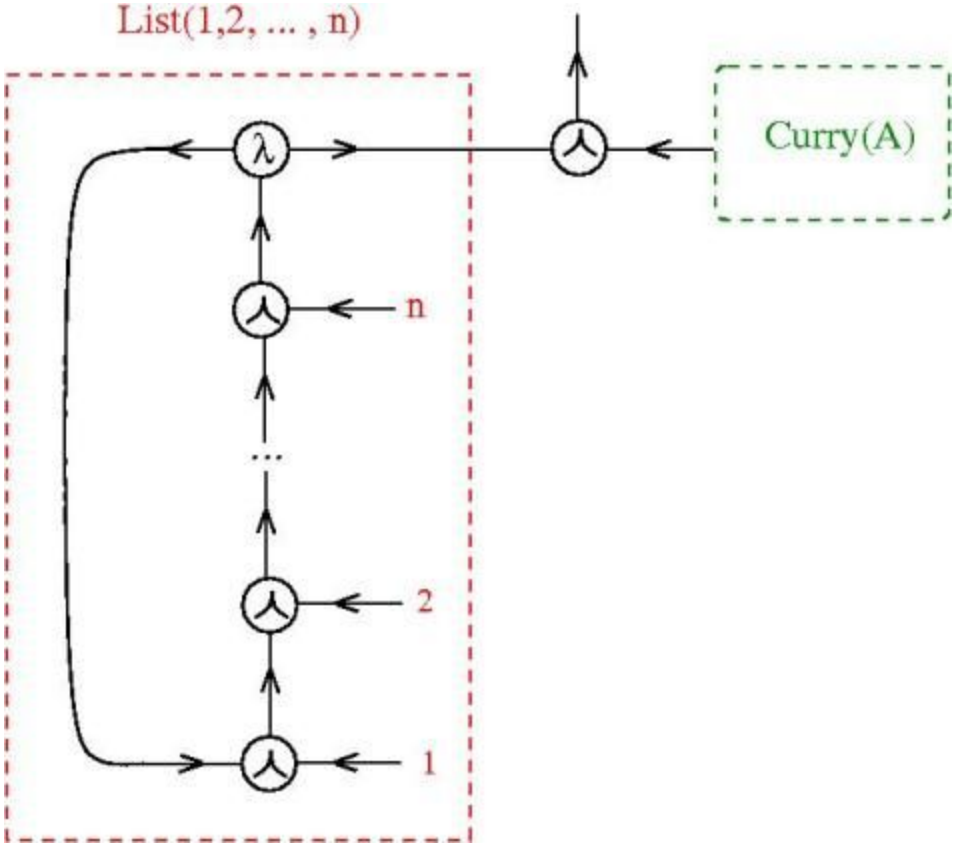
The next step is to use a n-zipper in order to clip the inputs, by using n graphic beta moves, until we get this:



This graph is, in fact, the following one (we replace the n-zipper, which is just a notation, or a macro, with its expression).


n
2
1
λ
λ
λ
A

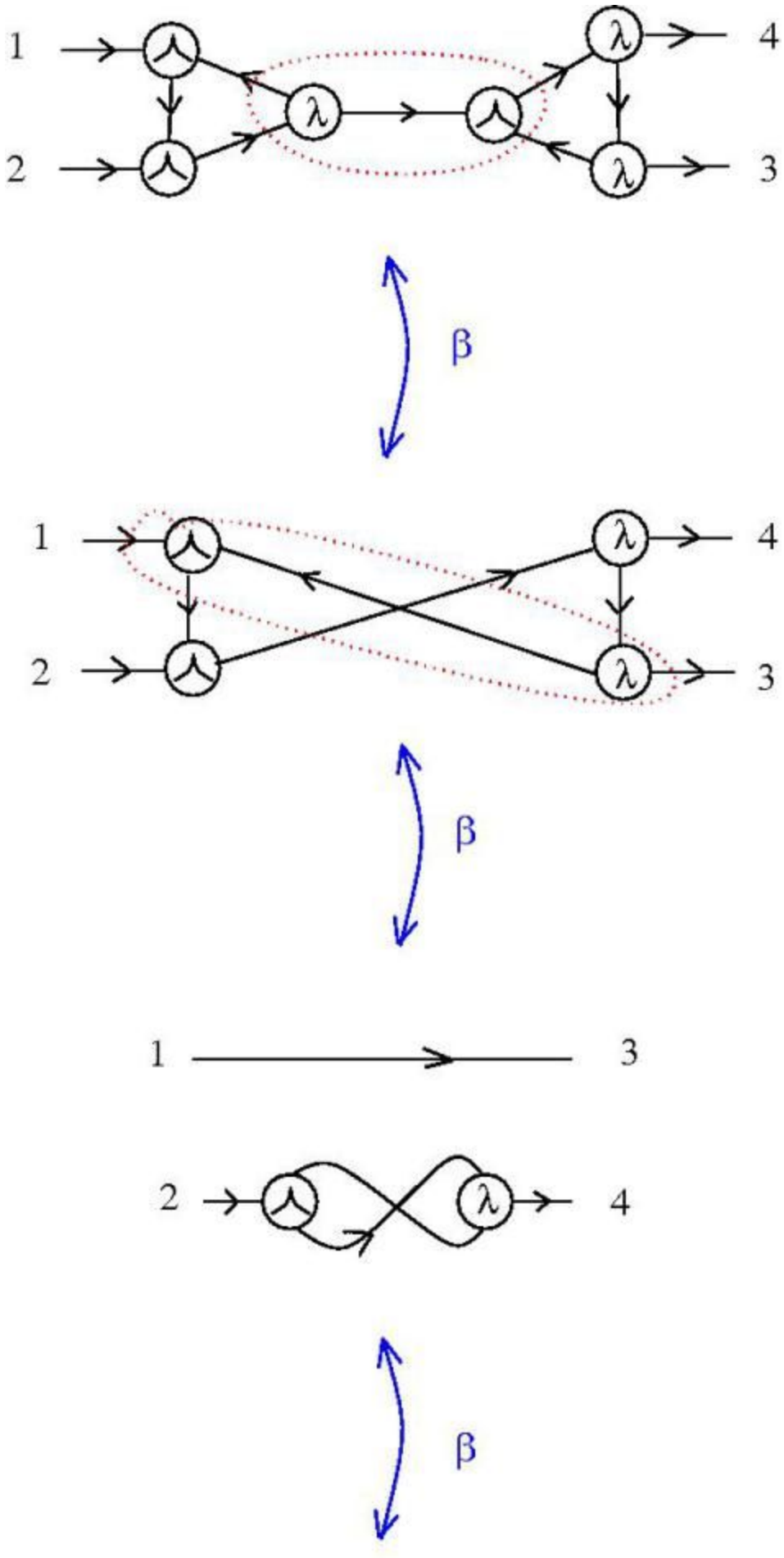
The graph inside the green dotted rectangle is the currying of A, let's call him $Curry(A)$. This graph has only one output and no inputs. (The procedure of currying can be made itself into a graph which is applied to the output of A, but we stop at this level for this post.) The graph inside the red dotted rectangle is almost a list. We shall transform it into a list by using again a zipper and one graphic beta move.



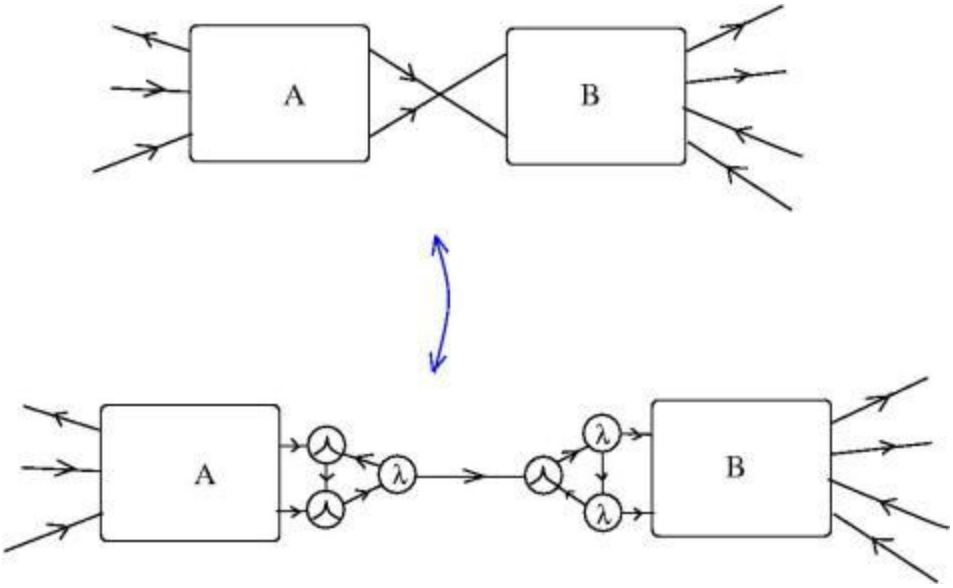
Now we're done!

[source: http://chorasimilarity.wordpress.com/2013/03/28/packing-and-unpacking-arrows-in-graphic-lambda-calculus/]

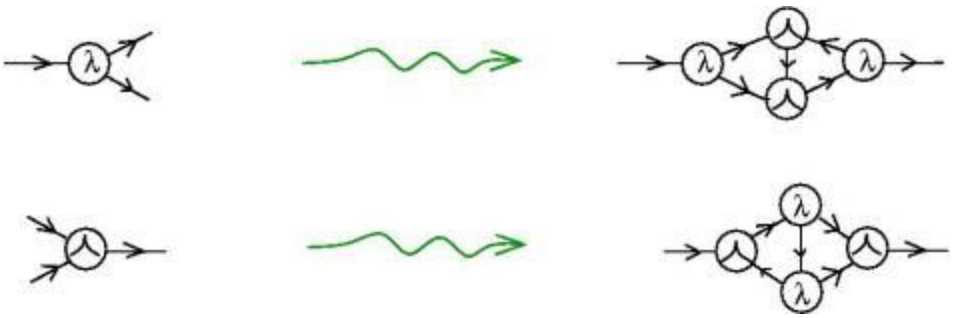
We start from the following sequence of three graphic beta moves.


1
2
λ
λ
λ
4
3
β
1
2
λ
λ
4
3
β
1
3
2
λ
4
β

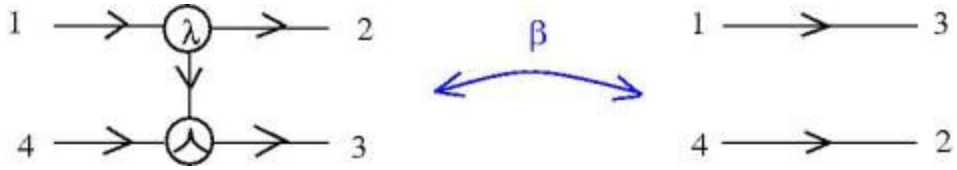
With words, this figure means: we pack the 1, 2, entries into a list, we pass it trough one arrow then we unpack the list into the outputs 3, 4. This packing-unpacking trick may be used of course for more than a pair of arrows, in obvious ways, therefore it is not a restriction of generality to write only about two arrows.
We may apply the trick to a pair of graphs in $GRAPH$, call them A and B, which are connected by a pair of arrows, like in the following figure.



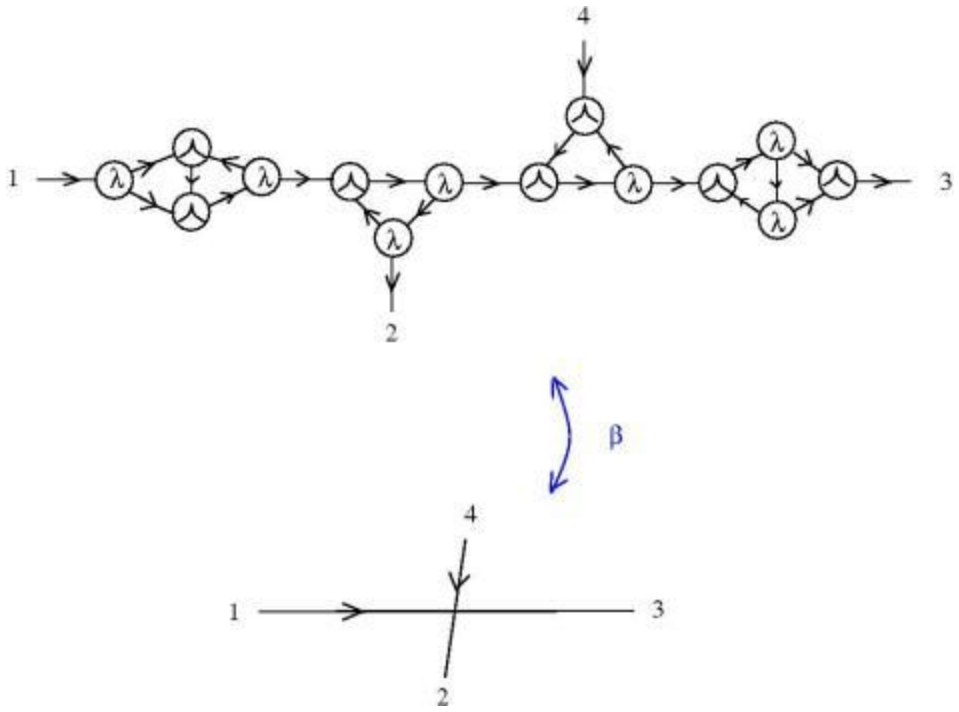
With the added packing and unpacking triples of gates, the graphs A, B are interacting only by the intermediary of one arrow.
In particular, we may use this trick for the elementary gates of abstraction and application, transforming them into graphs with one input and one output, like this:



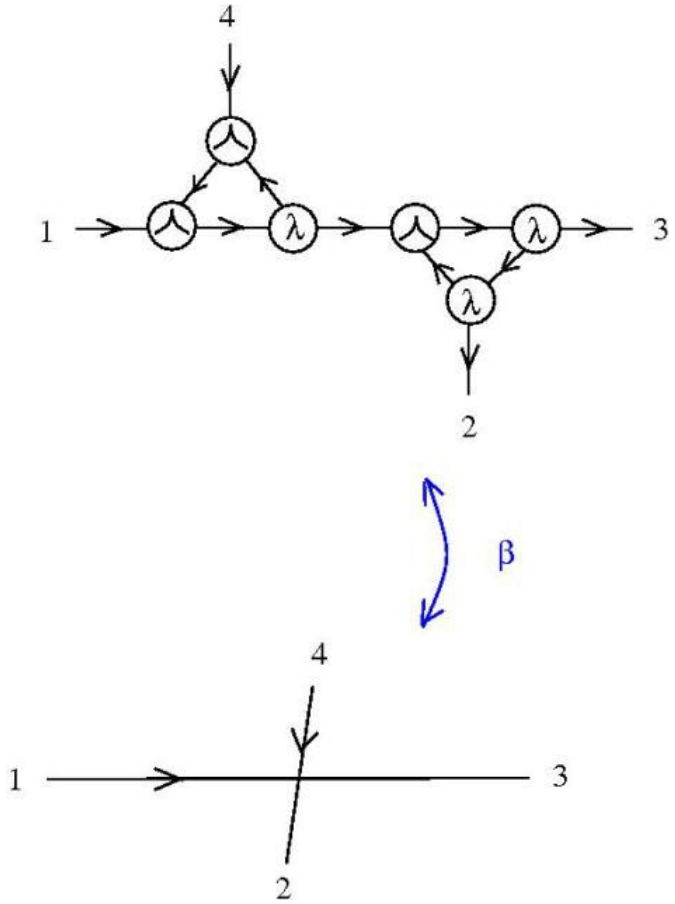
Let's look now at the graphic beta move:



If we use the elementary gates transformed into graphs with one input and one output, the move becomes this almost algebraic, 1D rule:

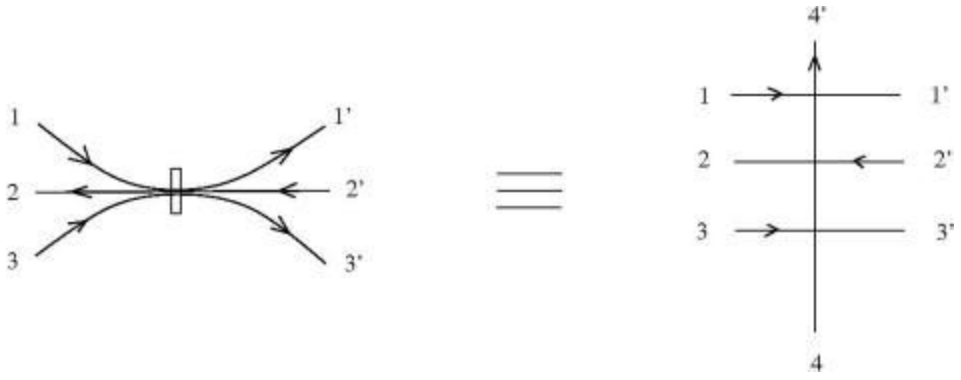


Finally, the packing-unpacking trick described in the first figure becomes this:

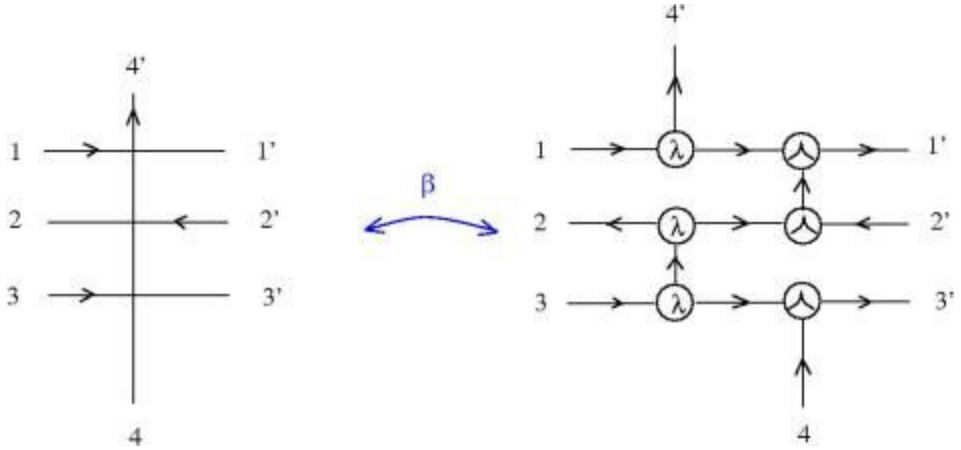


[source: http://chorasimilarity.wordpress.com/2013/04/10/packing-arrows-ii-and-the-strand-networks-sector/]

Strand networks are a close relative to spin networks (in fact they may be used as primitives, together with some algebraic tricks in order to define spin networks). See the beautiful article by Roger Penrose, Angular momentum: an approach to combinatorial space-time, in *Quantum Theory and Beyond*, ed. T. Bastin, Cambridge University Press, Cambridge, 1971. where strand networks appear in and around figure 17.
However the strand networks which I shall write about here are oriented. Moreover, a more apt name for those would be "states of strand networks". The main thing to retain is that the algebraic glue can be put over them (for example, as concerns free algebras generated by those, counting states, grouping them into equivalence classes which are the real strand networks and so on).
In graphic lambda calculus we may consider graphs in $GRAPH$ which are obtained by the following procedure. We start with a finite collection of loops and we clip together, as we please, strands from these loops. How can we do this? Is simple, we choose our strands we want to clip together and the we cross them with an arrow. Here is an example for three strands:

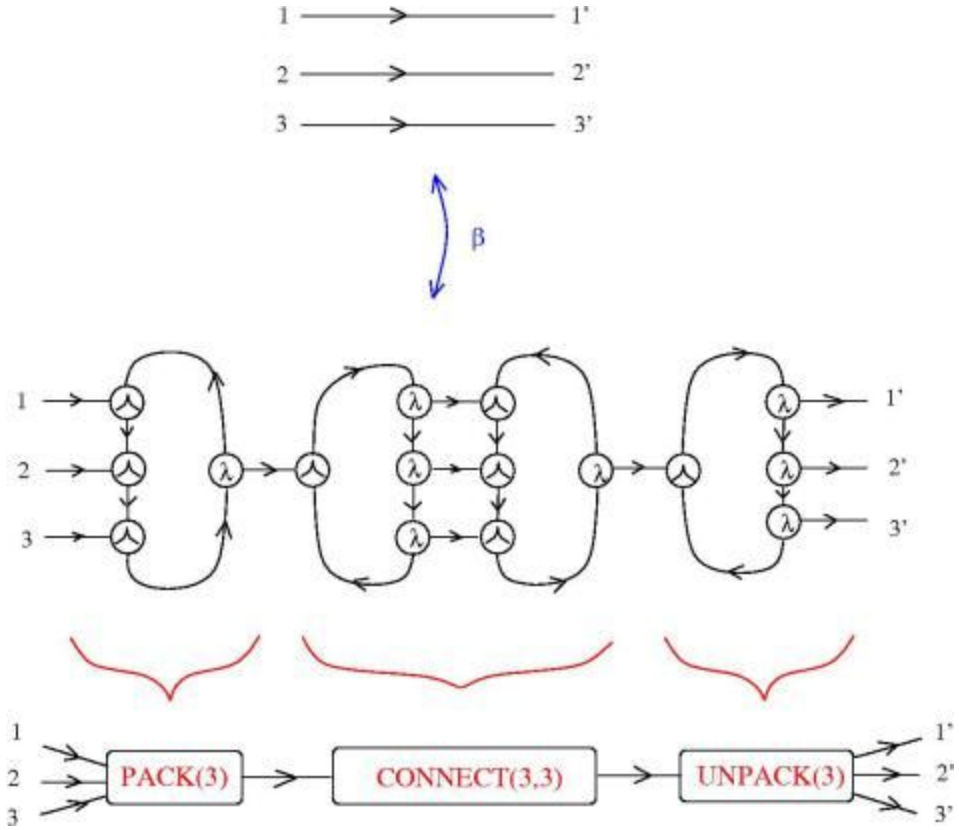


We want to make this clipping permanent, so to say, therefore we apply as many graphic beta moves as are needed, three in this example:



The beautiful thing about this procedure is that in the middle of the graph from the right hand side we have now three arrows with the same orientation. So, let's pack them into one arrow.

The packing procedure for three arrows, with the same orientation, is the following:

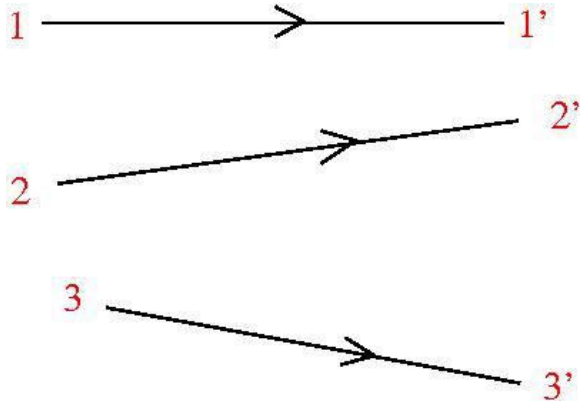


So, we pack the arrows, we connect them and we unpack afterwards.
With the previous trick, we may define now decorated arrows of strand networks, but mind that we have bundles, packs of oriented strands, so the decoration has to be more precise than just indicating the number of strands.
That is about all, only we have to be careful to discern among clipped strands which become decorated arrows and clipped strands which become nodes of the strand network. But this is rather obvious, if you think about.
Conclusion. Let's think about the purpose of the algebraic glue which is needed in order to define a spin network and then an evolving spin network. As you can see all comes down to another way of deciding which is the sequence of applications of (graphic) beta moves, which is this time probabilistic. It is another type of computation than the one from the lambda calculus sector, where we use the combinators and zippers (as well as some strategy of evaluation) in order to make a classical computation. So at the level of graphic lambda calculus, it becomes transparent that both are computations, but with slightly different strategies.

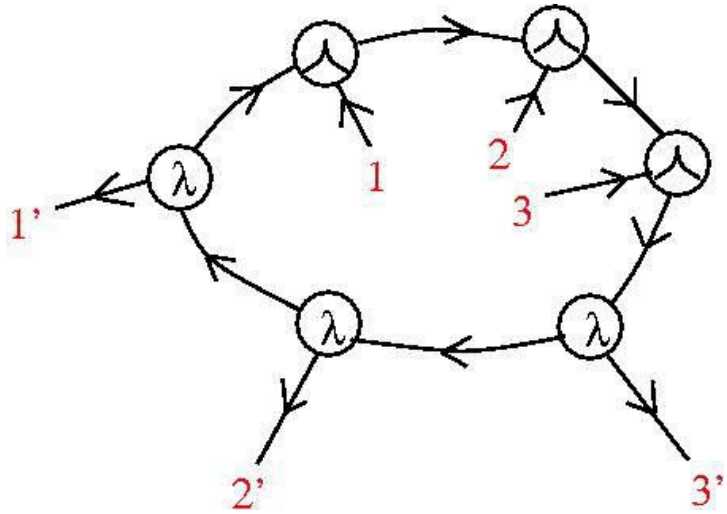
[source:
http://chorasimilarity.wordpress.com/2013/05/16/sets-lists-and-order-of-moves-in-graphic-lambd

a-calculus/]

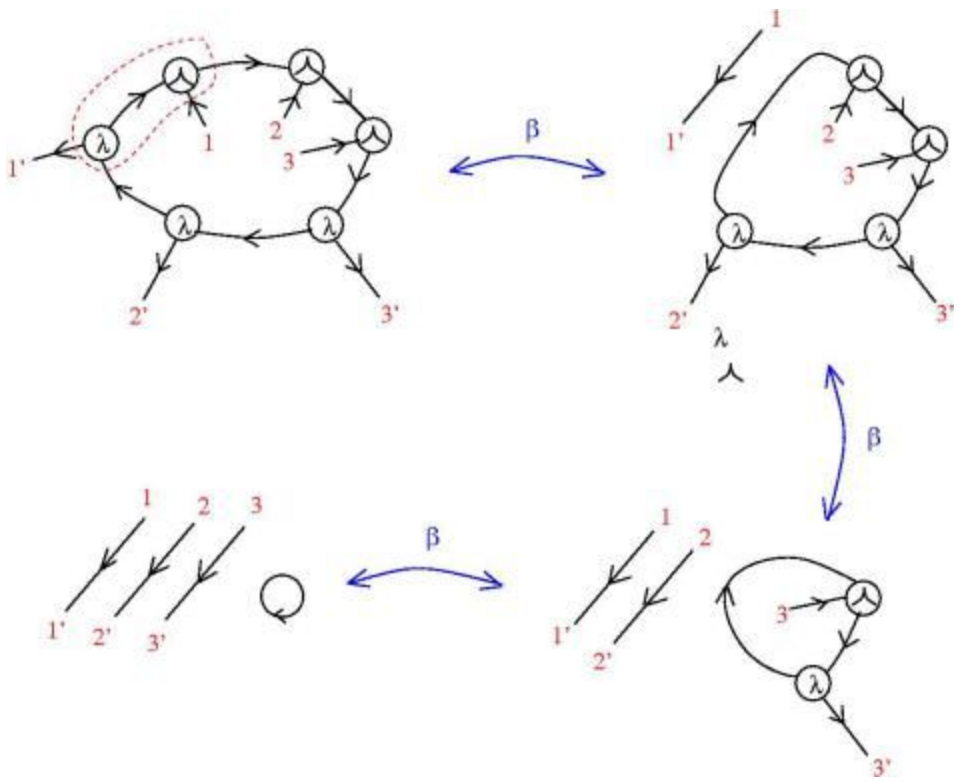
Suppose that we want to group together three arrows in graphic lambda calculus. We have this:



We want to group them together such that later, by performing graphic beta moves, the first arrow available to be 11′, then 22′, then 33′. Moreover, we want to group the arrows such that we don't have to make choices concerning the order of the graphic beta moves, i.e. such that there is only one way to unpack the arrows. The solution is to "pack" the arrows into a variant of a list. Lists have been defined here, in relation to currying.



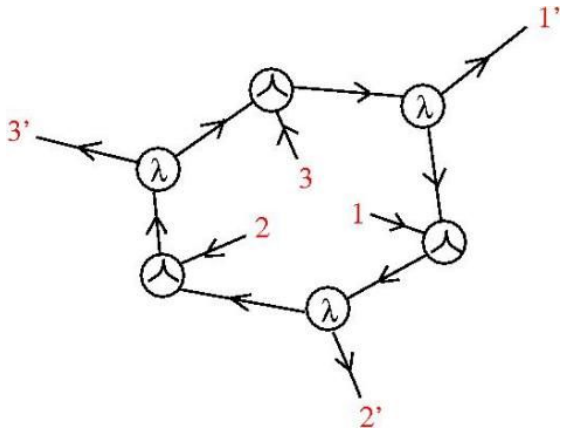
Basically we take a zipper and we close it. Further we see how to unpack this list.



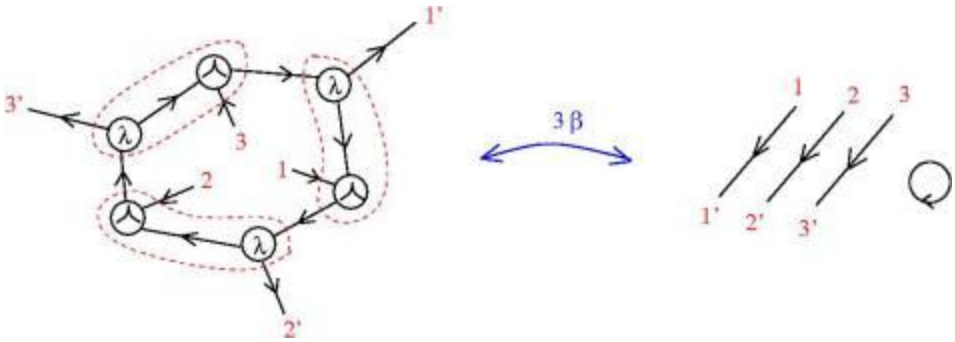
The dashed red curve encircles the only place where we can use a graphic beta move. The first move frees the 11′ arrow and then there is only one place where we can do a graphic beta move, which frees the 22′ arrow and finally a last move frees the 33′ arrow and produces a loop which can be eliminated.

The uniqueness of the order of moves is true, in fact, if we accept as valid beta moves only those from left to right (i.e. those which eliminate gates). Otherwise we can go back and forth with a beta move as long as we want.

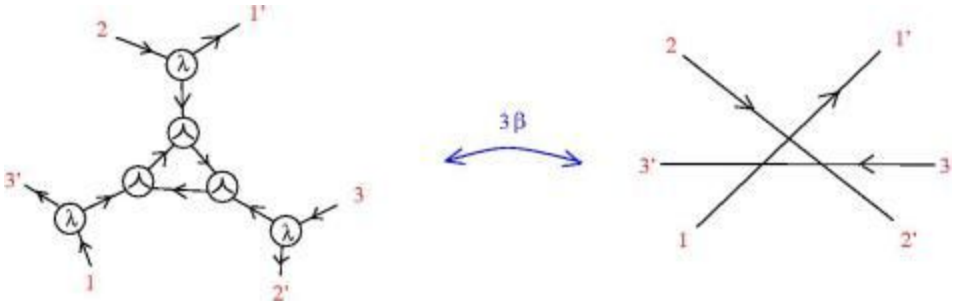
There is another way to pack the three arrows, under the form of another graph, which could aptly be called a “set”. This time we need a graph with the property that we can extract any arrow we want from it, by one graphic beta move. Here is the solution:



Indeed, in the next figure we see that we have three places, one for each arrow, which can be independently used for extraction of the arrow of choice.



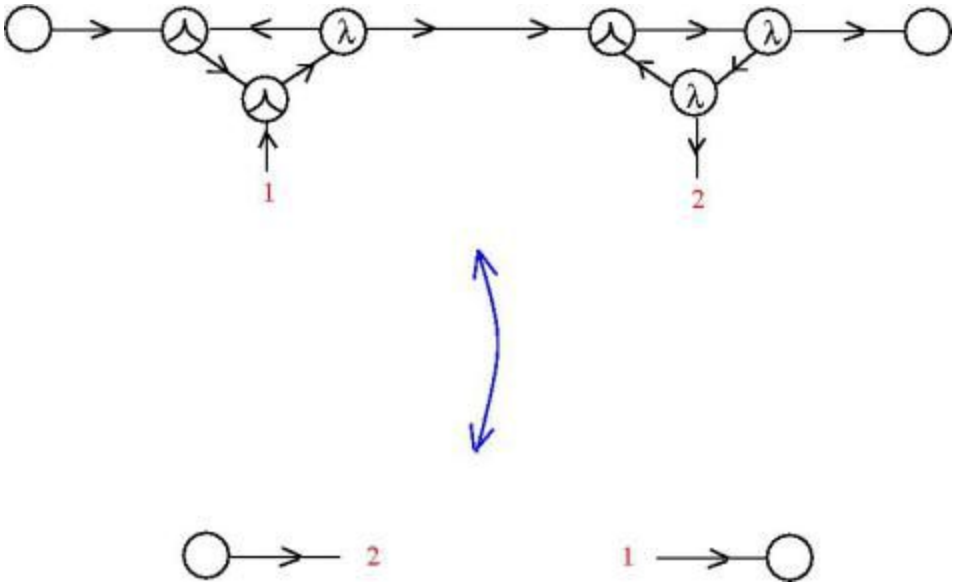
In between these extremes, there are other possibilities. In the next figure is a graph which packs the three arrows such that: there are three places where a graphic beta move can be performed, as in the case of the set graph, but once a beta move is performed, the symmetry is broken. The performed beta move does not free any arrow, but now we have the choice between the other two possible beta moves. Any such choice frees only one arrow, and the last possible beta move frees the remaining two arrows simultaneously.
Here is the figure:



The graph from the left hand side is not a list, nor a set, although it is as symmetric as a set graph. There are $3 \cdot 2 = 6$ possible ways to unpack the graph. So this graph encodes all lists of two arrows out of the three arrows.

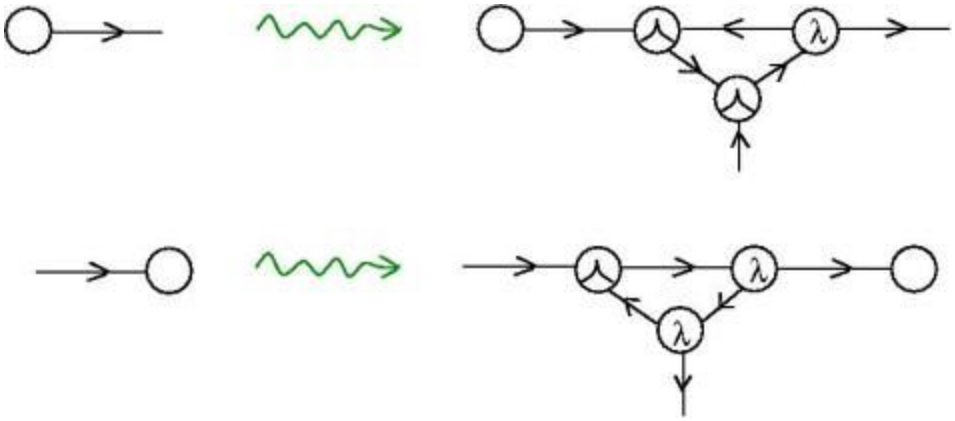
[source:
http://chorasimilarity.wordpress.com/2013/05/18/freedom-sector-of-graphic-lambda-calculus/]

, graphic lambda calculus has a freedom sector. Which means in that sector you can do anything you like (modulo some garbage, though). It's yet not clear to me if this means a kind of universality property of graphic lambda calculus.
The starting point is the procedure of packing arrows explained in this post. This procedure can be seen in the following way:



Here, the left and right void circles with the respective arrows represent: the one from the left is a generic out arrow which exits from a gate and the one from the right is a generic in arrow which enters in a gate.

This gives the following idea: replace the inputs and the outputs of the gates from graphic lambda calculus by the following graphs (the green wiggly arrow means “replace”):



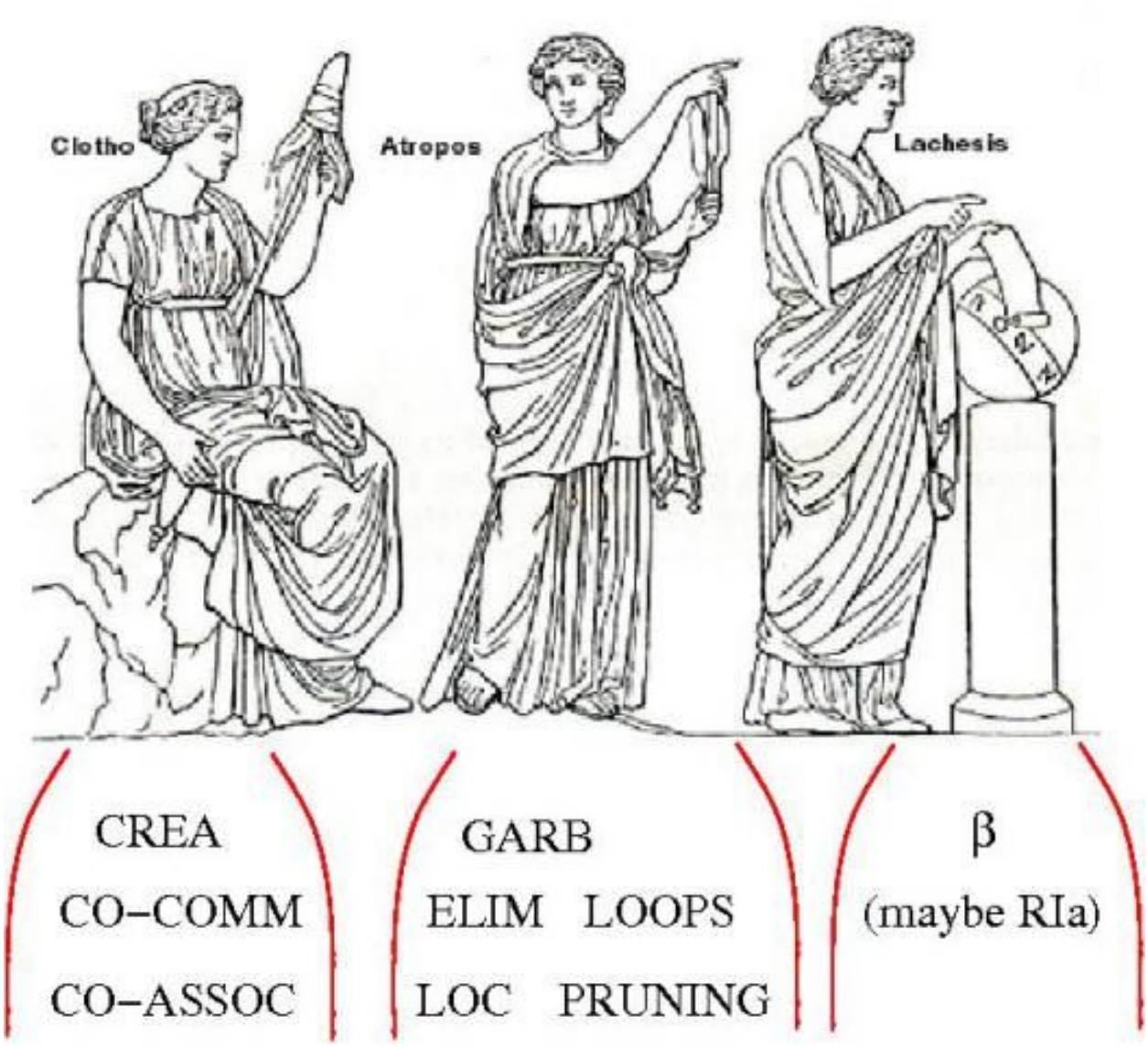
For example, look how it’s done for the 人 graph. Technically we define new macros, one for each elementary gate. Let’s call these macros “the free gates”.

These free gates define the free sector of the graphic lambda calculus, which consists all graphs made by free gates, along with the move of cutting or gluing arrows.
The free sector has inside a copy of the whole graphic lambda calculus, with the condition of adding a local move of elimination of garbage, which is the local move of elimination (goes only one way, not both) of any graph which is not made by free gates with at most, say, 100 arrows + gates. This move is needed, for example, for the case of emulating the graphic beta move with free gates, where we are left with some garbage consisting of one λ gate and one 人 gate, seen as disconnected graphs.

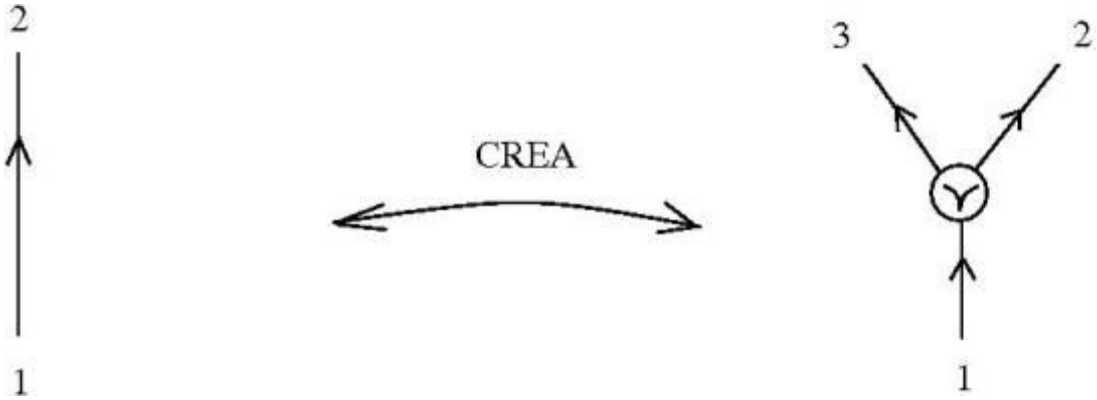
[source: http://chorasimilarity.wordpress.com/2012/11/21/ancient-turing-machines-i-the-three-moirai/]

The question is how to create any graph in GLC from simple ones. We may do it by adding some new moves.

By analogy with the work of Fates (or Moirai):



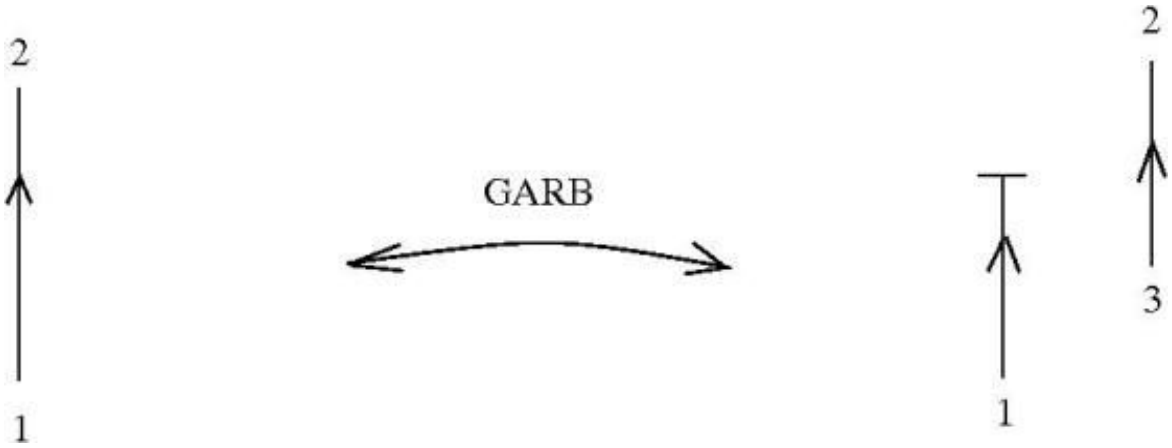
CLOTHO is creating the thread, namely the new move called “CREA” (from “creation”):



Basically she introduces a FAN-OUT gate into the thread. In order to make

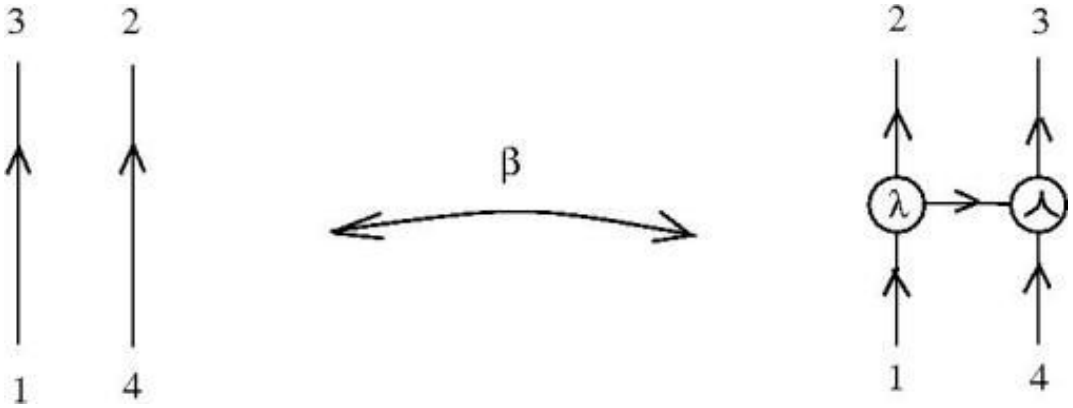
this gate to function as FAN-OUT, she also needs from the graphic lambda calculus the moves CO-COMM (which allows her to permute the outputs) and CO-ASSOC (which allows her to not care about the order of application of a cascade of FAN_OUT gates).

ATROPOS cuts the thread, namely she is performing a move which I shall call “GARB” (from “garbage”), which is a new move introduced in graphic lambda calculus:



She picks from the moves of graphic lambda calculus LOCAL PRUNING and ELIMINATION OF LOOPS, which are kind of her style.

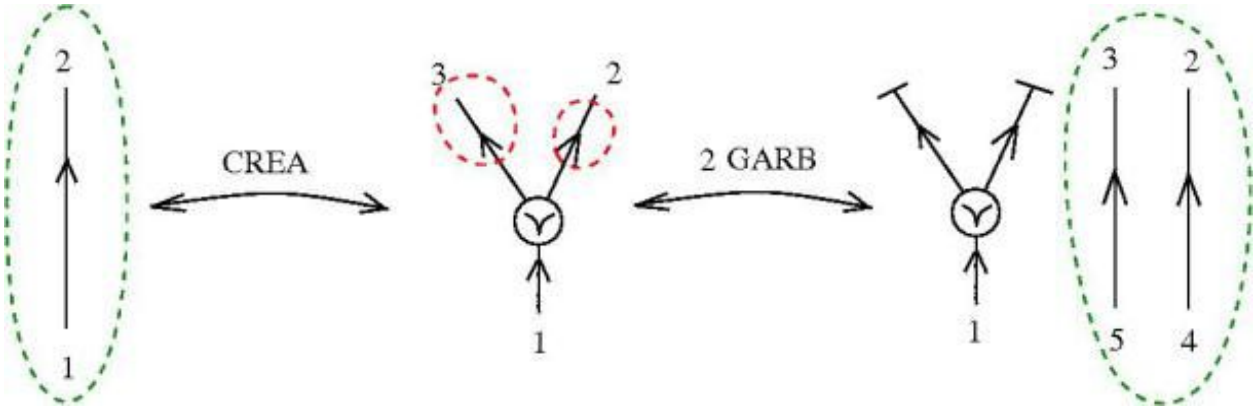
LACHESIS is doing only one move, the graphic beta, described here (and see the paper) as a braiding move, when seen in knot diagrams macro. (She might actually be able to do also the oriented Reidemeister 1a move, see further.)



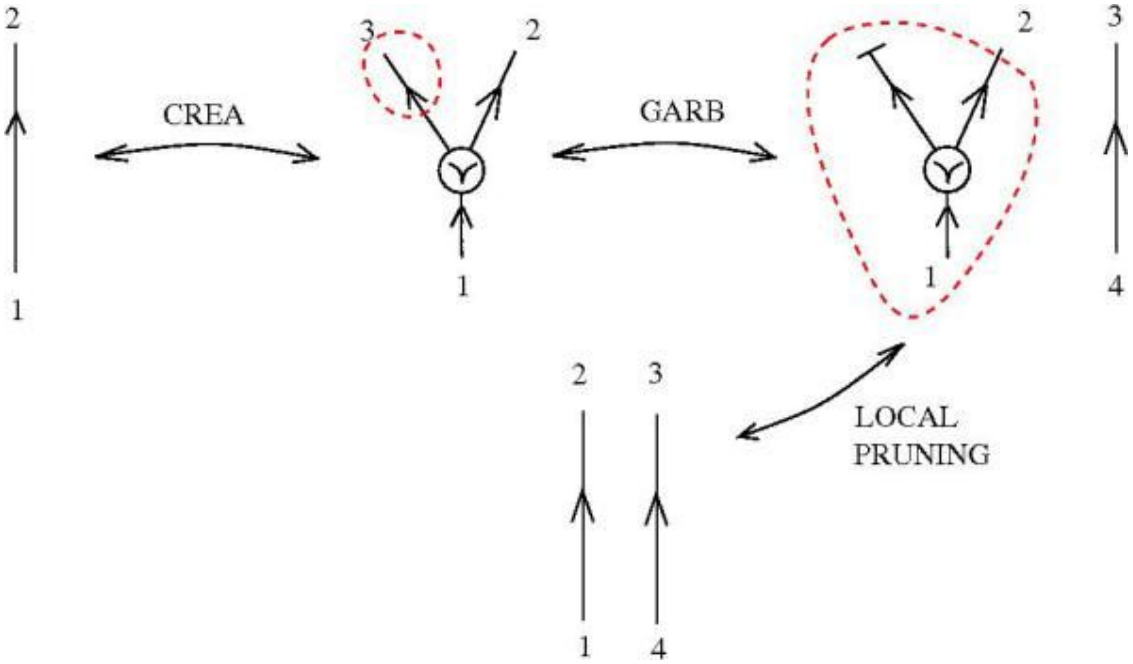
This is a graphic form of β reduction, so you may say that LACHESIS is performing something akin to β reduction.

3. How does it work? The Moirai have a thread to start from. Their first goal

is to produce the gates. They can easily have two gates, one appearing after GARB, the other appearing after CREA. They still need the application gate (corresponding to the application operation in lambda calculus) and the lambda abstraction gate.
They also need to have enough threads to play with. Here are two ways of getting them. The first one is using only GARB and CREA moves. The dashed green curves represent the input and the output of their activities. The dashed red curves indicate where the moves are applied.



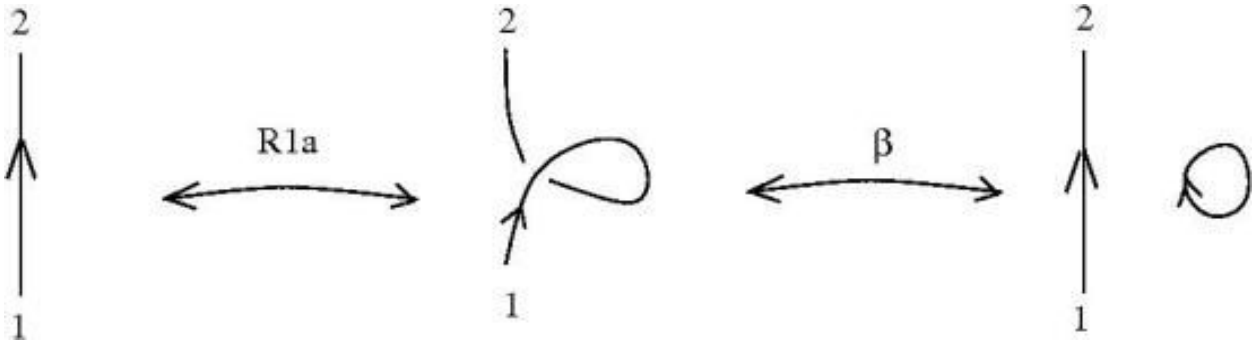
Another way of producing two threads from one, more specifically producing a new thread and also keeping the old one, uses also LOCAL PRUNING:



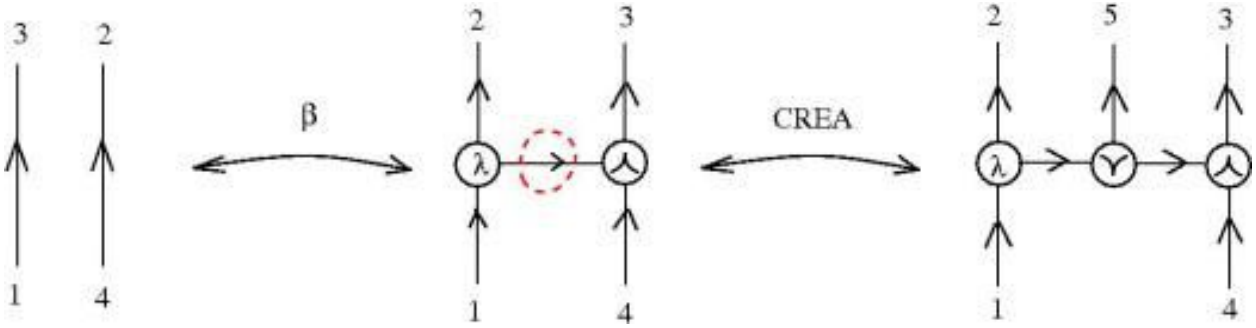
If the Moirai have only one thread and no loop, then we have to add to

LACHESIS's competences the three Reidemeister moves, or at least the Reidemeister 1a move:

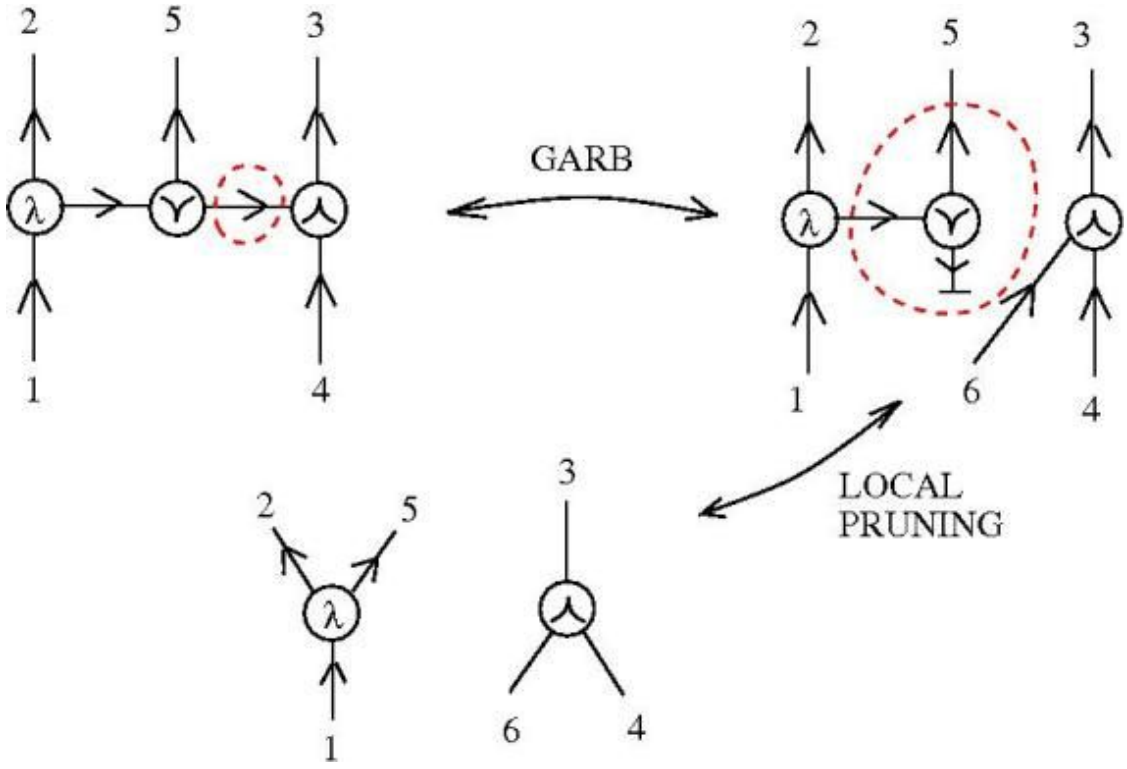
Then LACHESIS may use her graphic beta move in order to get a thread and a loop. ATROPOS has to refrain to use her ELIMINATION OF LOOPS for later!



Now the three Moirai are ready to produce the application and lambda abstraction gates. CLOTHO and LACHESIS start with two threads (which they already have), in order to get to an intermediary step.



From here, with some help from ATROPOS, they get a lambda gate and an application gate.



Author: Marius Buliga (chorasimilarity)