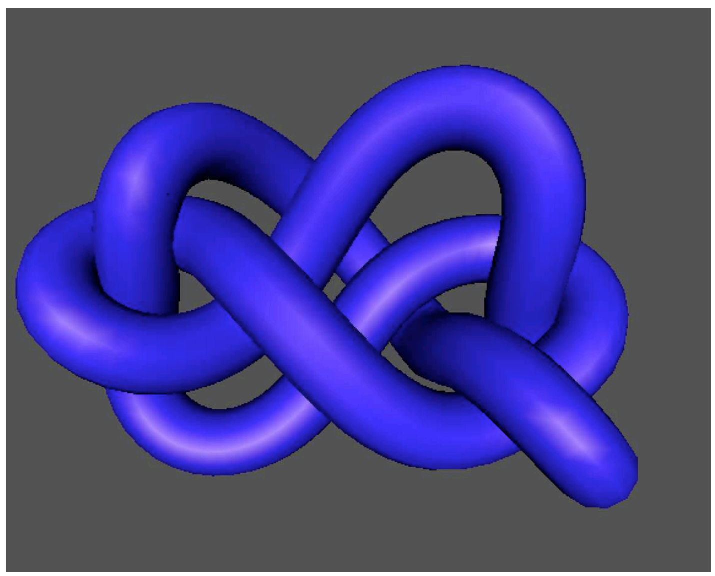
Chemlambda, universality and self-multiplication

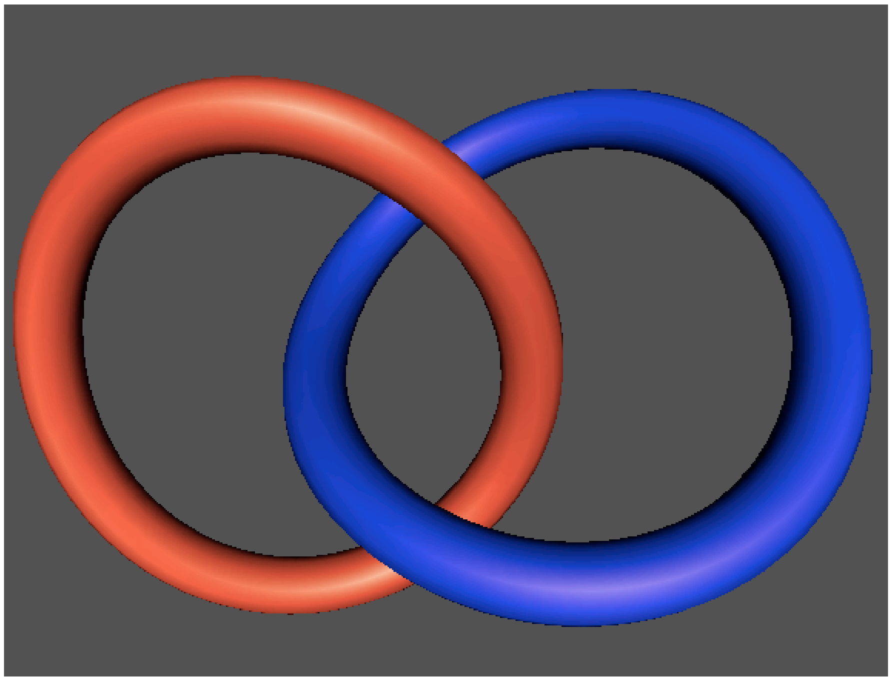
Marius Buliga [1] and Louis H. Kauffman [2]

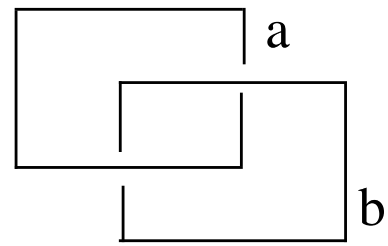
[1] Institute of Matematics of the Romanian Academy
P.O. BOX 1-764, RO 014700, Bucharest, Romania
`Marius.Buliga@gmail.com`
[2] Department of Mathematics, University of Illinois at Chicago
851 South Morgan Street, Chicago, Illinois, 60607-7045
`kauffman@uic.edu`

(Graphical Lambda Calculus and Knots)



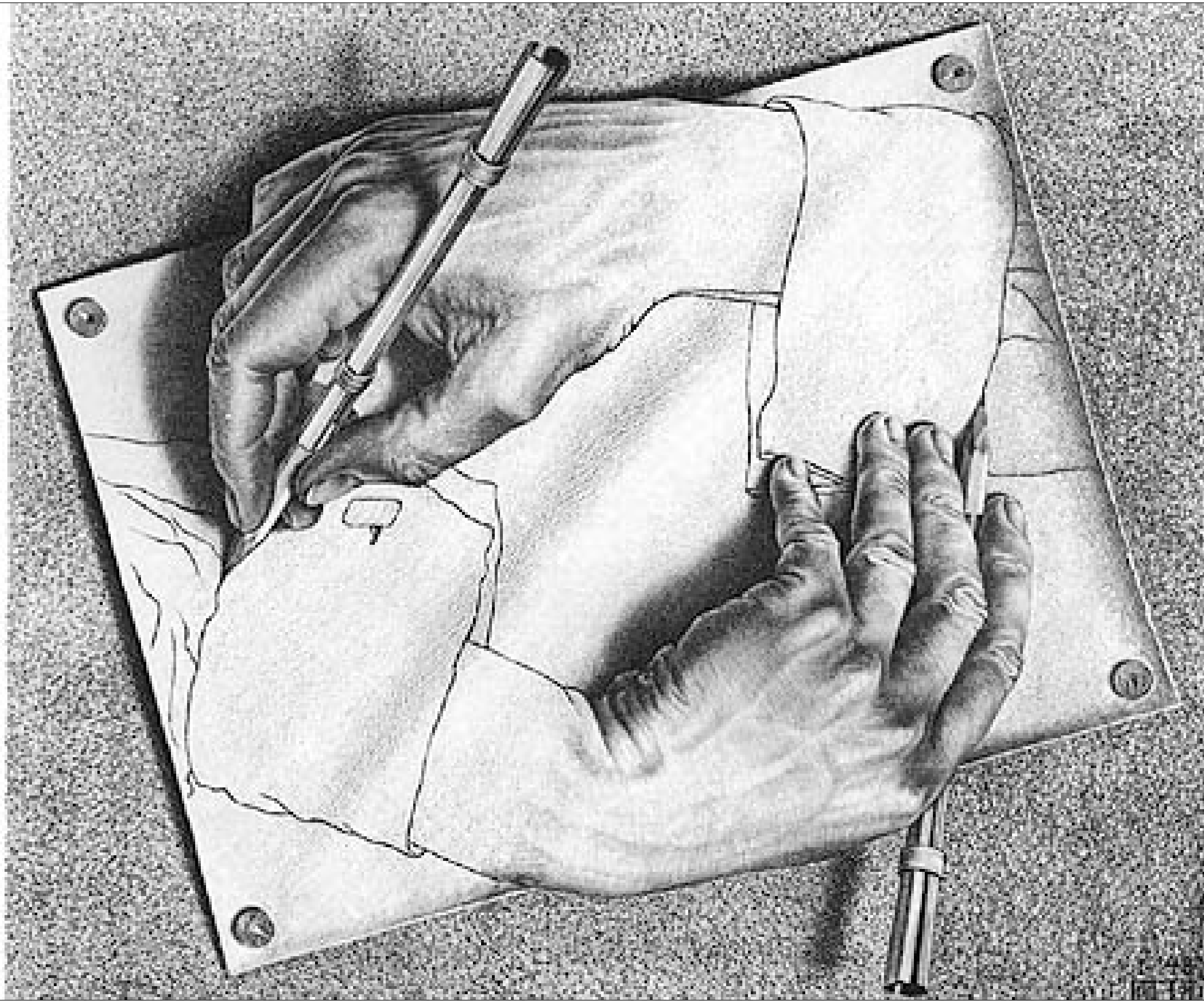
This talk is not quite about knots, but
this slide gives a hint that knots and
fixed points are linked with one another.

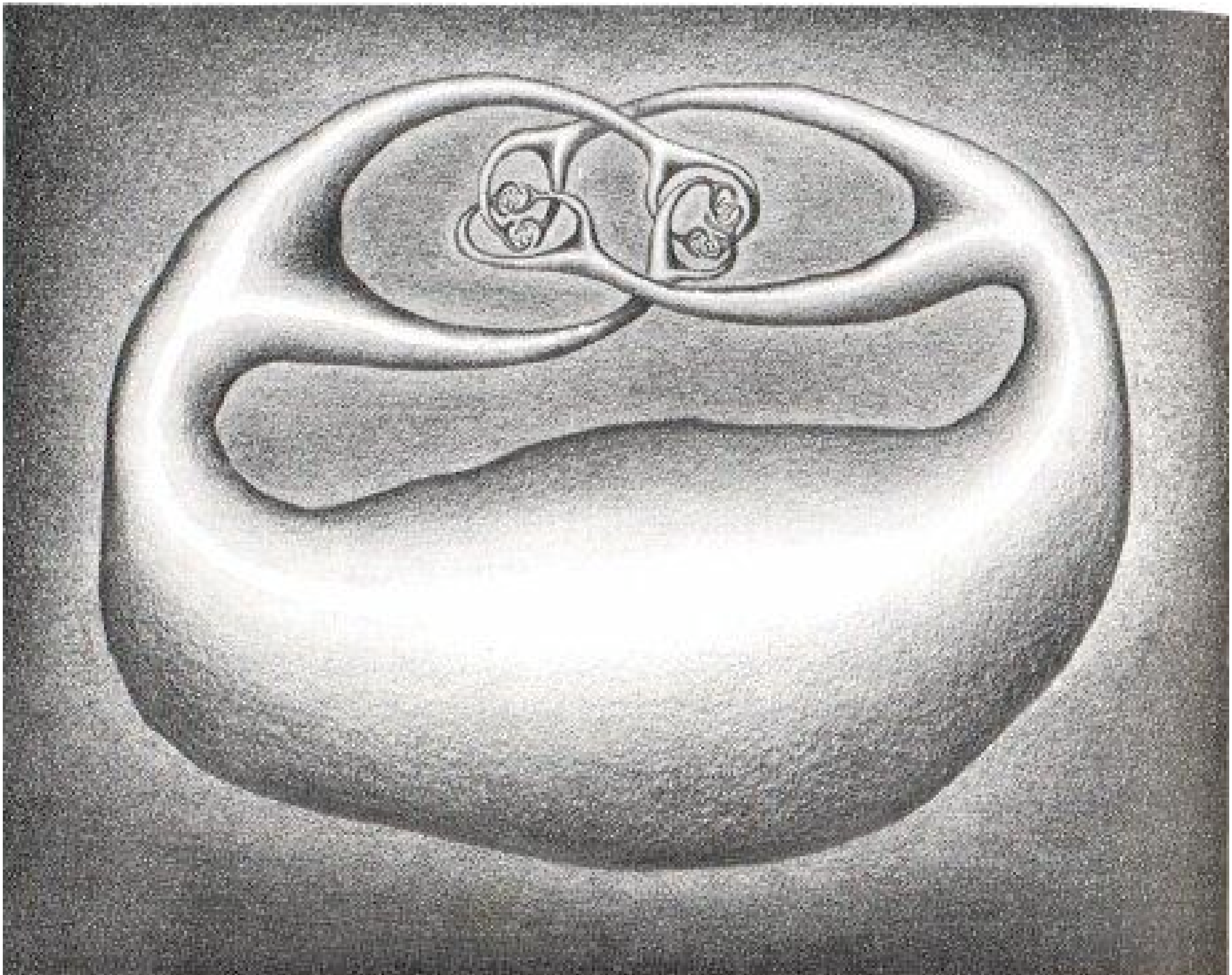


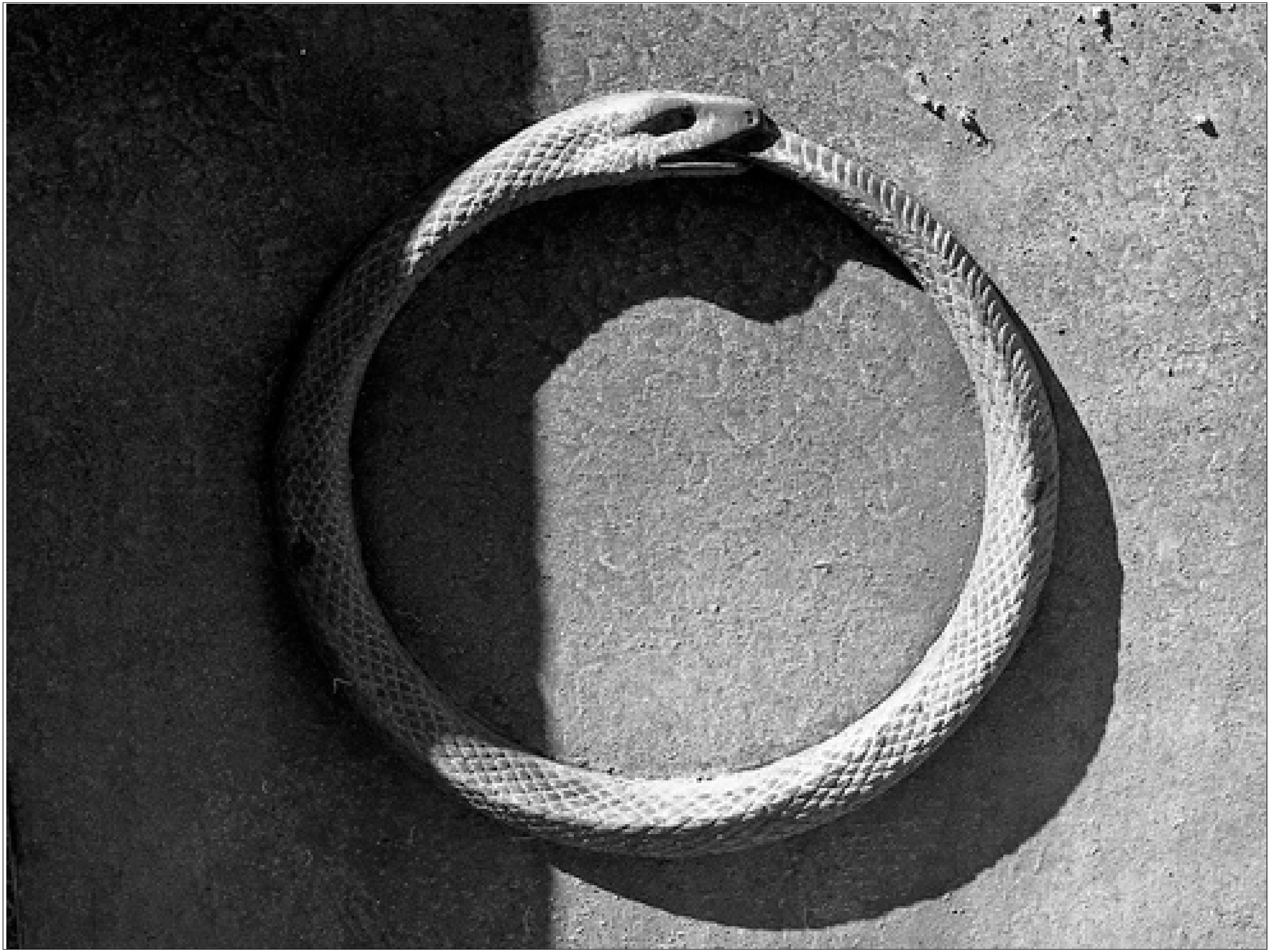


$a=\{b\}$
$b=\{a\}$

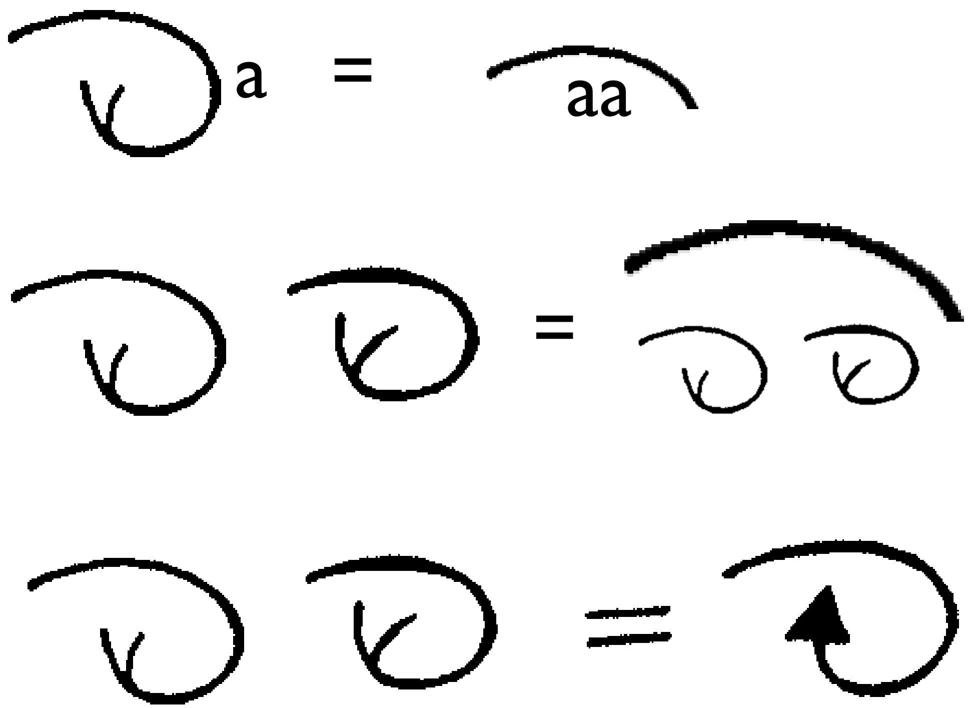
And why Topology and Recursion are Intertwined.







Duplicating Gremlin Meets Itself



A quick review of lambda calculus

Lambda Notation

$$F = \lambda xy.f(x,y)$$
$$(Fx)y = f(x,y).$$

(note the non-associativity)

For example, If

$$F = \lambda xy.y(yx),$$

then

$$(Fa)b = b(ba).$$

Church-Curry Fixed Point Theorem and Recursion

$$G = \lambda x.F(xx).$$

$$Gx = F(xx).$$

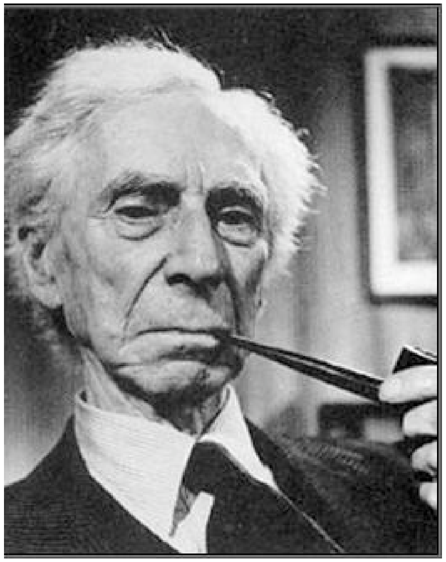
$$GG = F(GG).$$ Any F has a fixed point!

And Its Dangers

$$G = \lambda x. \sim (xx),$$

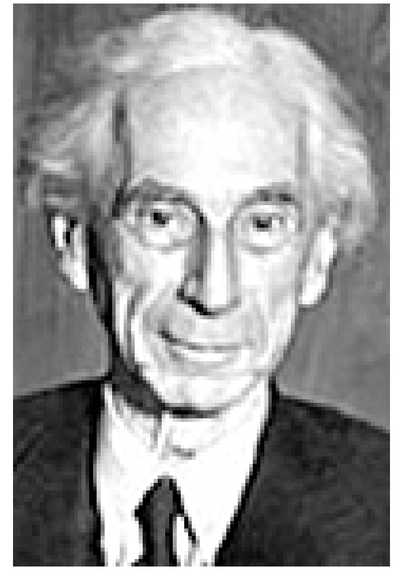
$$GG = \sim (GG).$$

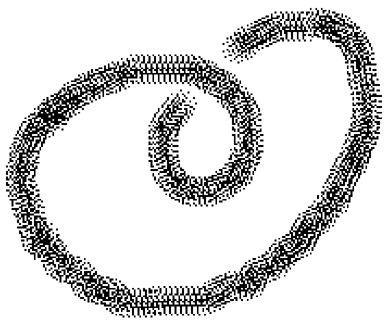
This is the Lambda version of the Russell Paradox.

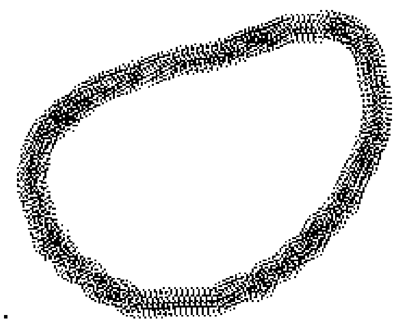


$$\boxed{\begin{array}{c} Rx = \sim xx \\ RR = \sim RR \end{array}}$$

Russell Paradox (K)not.







A belongs to A.

A does not belong to A.

For Lambda Calculus one resolves the paradox by replacing equality by a reductive move.

$$G = \lambda x.F(xx).$$

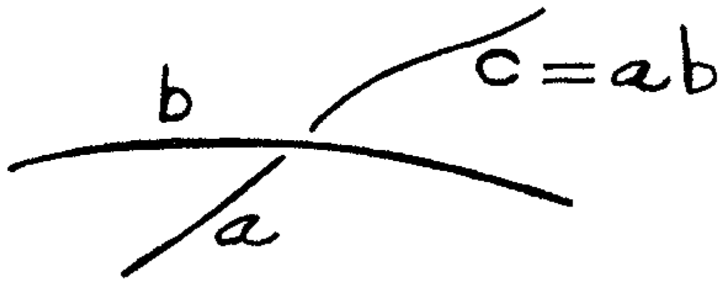
Ga -----Beta Reduction -----> F(aa)

GG -----> F(GG)
-----> F(F(GG))
-----> F(F(F(GG)))

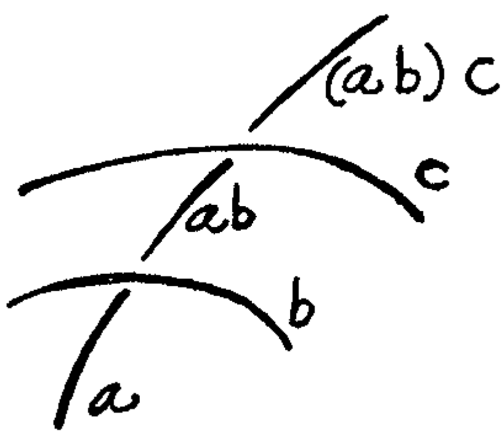
Whence Recursion.
And recursion must be controlled.

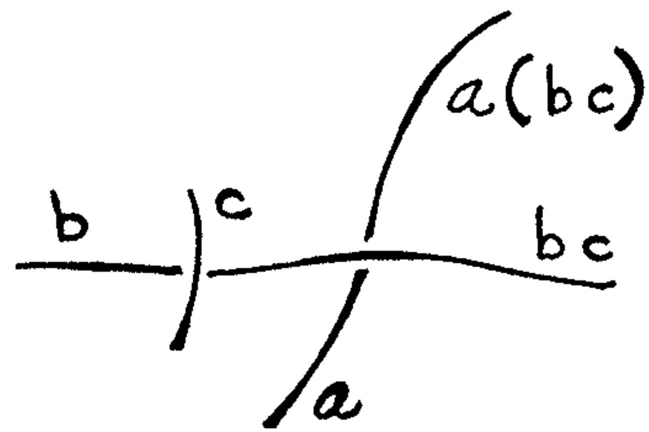
Non-Associative Formalism in Knot Diagrams

Label the arcs in a link diagram. Regard the label on the arc c obtained by underpassing b from a as a product of a and b : c= ab.



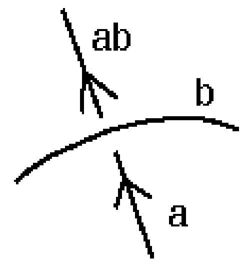
Here we abandon the notion of membership at a crossing and replace it with an algebraic product. Think of the overcrossing line as acting on the undercrossing line to produce the label for the continuation of the undercrossing. This is an inherently non- associative formalism, as the diagrams below demonstrate.



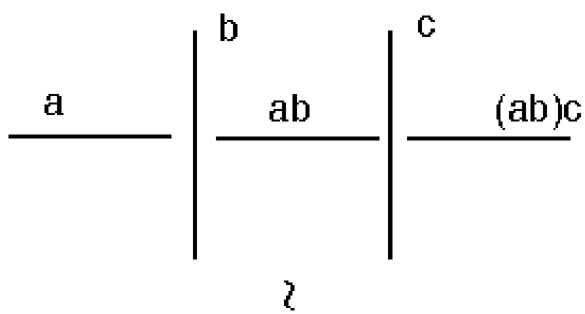


Knot-Logical Diagrammatic Lambda Calculus

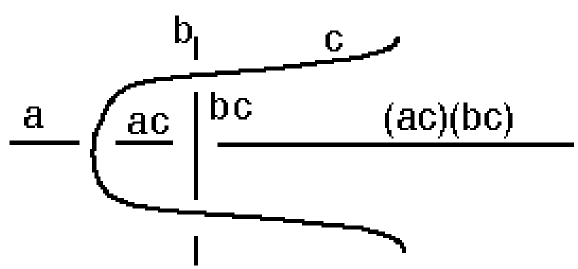
Knot diagrams as non-associative formalism



Multiplication at a Crossing



(basic non-associativity)



(topological moves have algebraic interpretations)

Figure 23: Knot Diagrammatic Multiplication

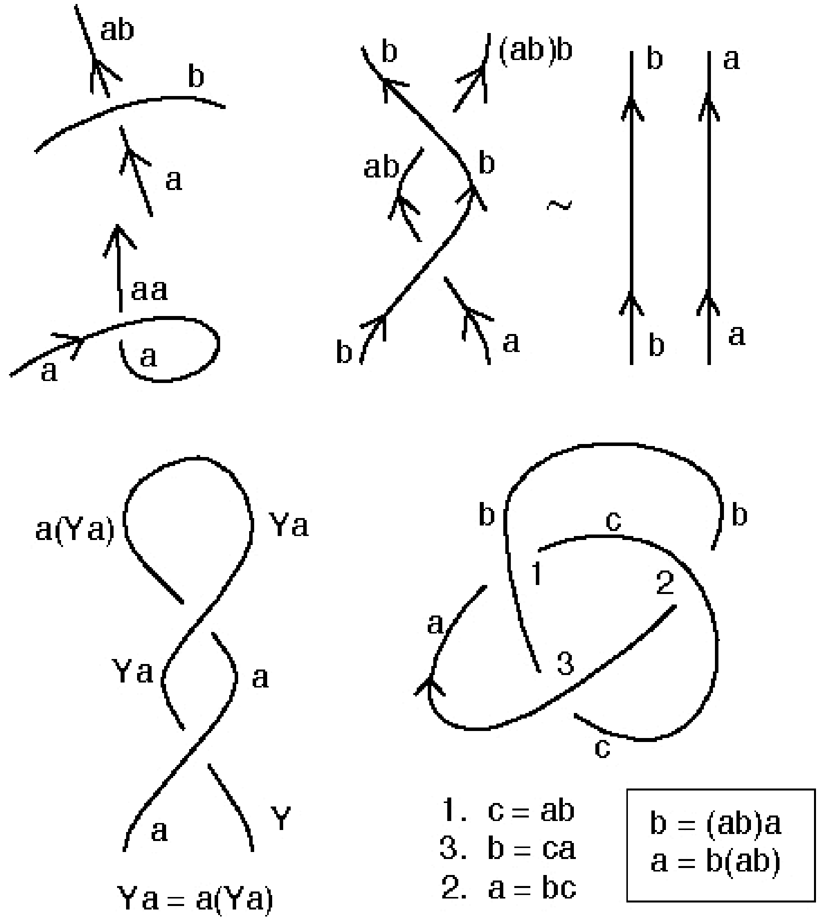
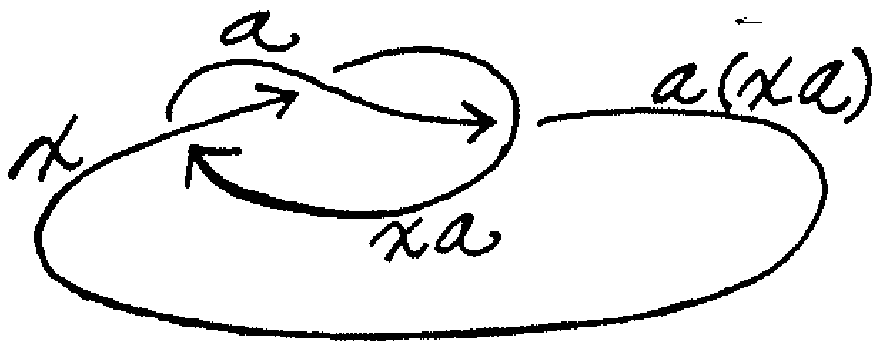


Figure 24: Relations and Diagrams with Loops



$$\boxed{x = a(xa)}$$

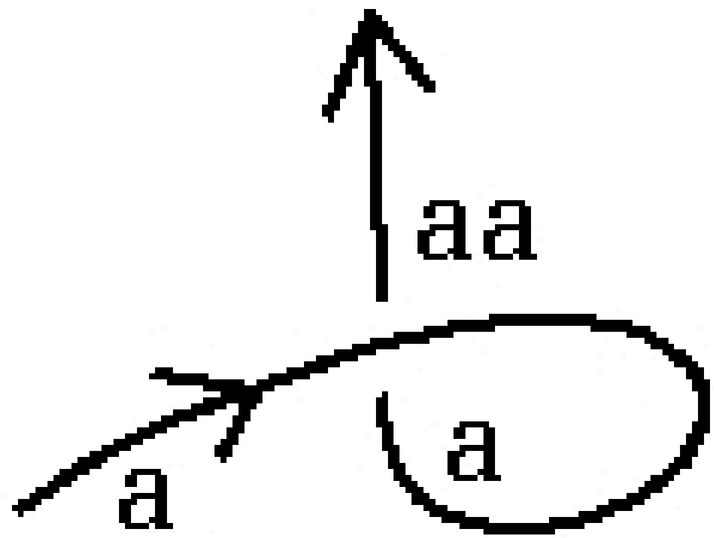
Fixed points occur naturally in knot theory but are handled not by lambda calculus, but by using an algebra with topological relations.

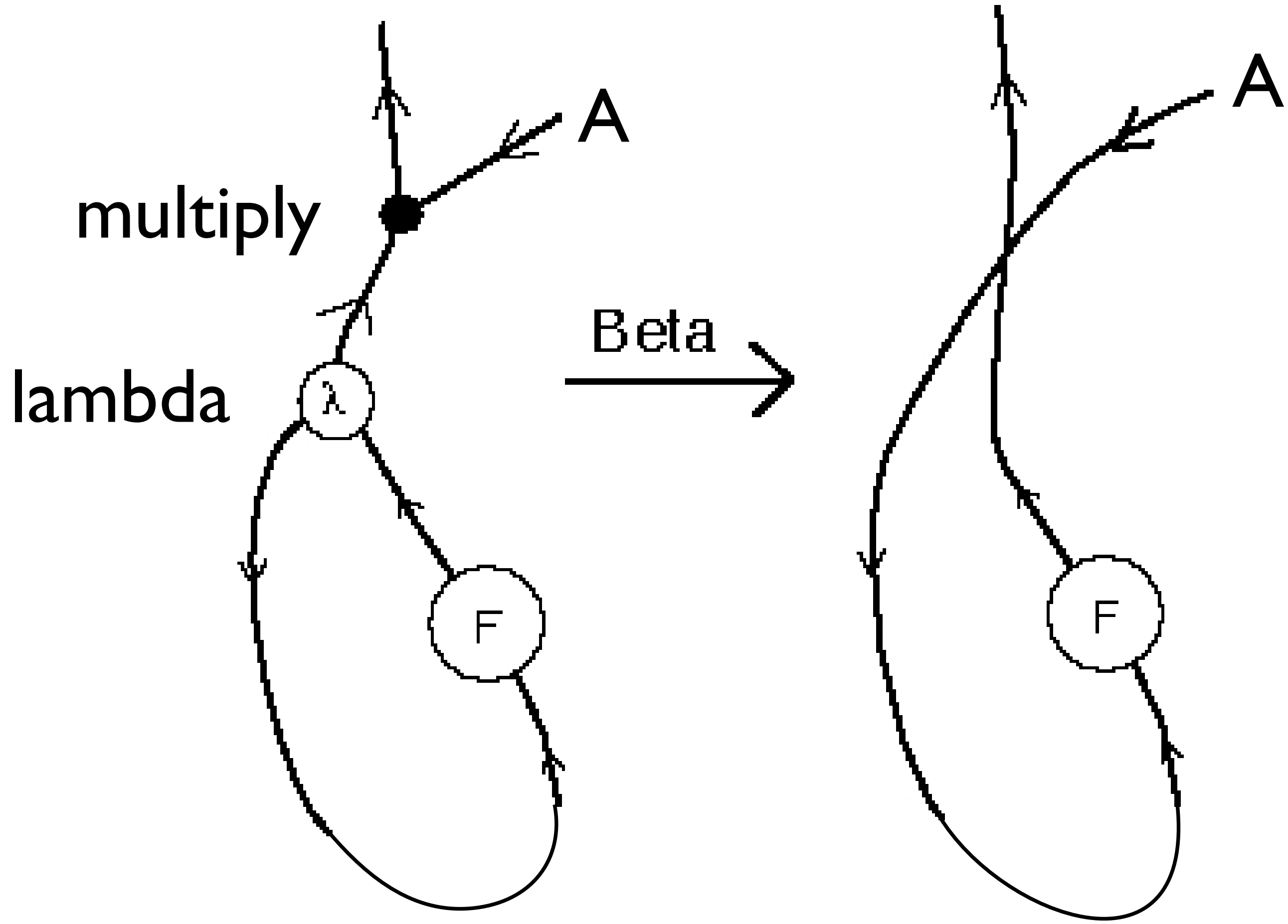
We are exploring extensions of knot theoretic topology by the addition of diagrammatic lambda calculus.

Lets use this glyph for

a

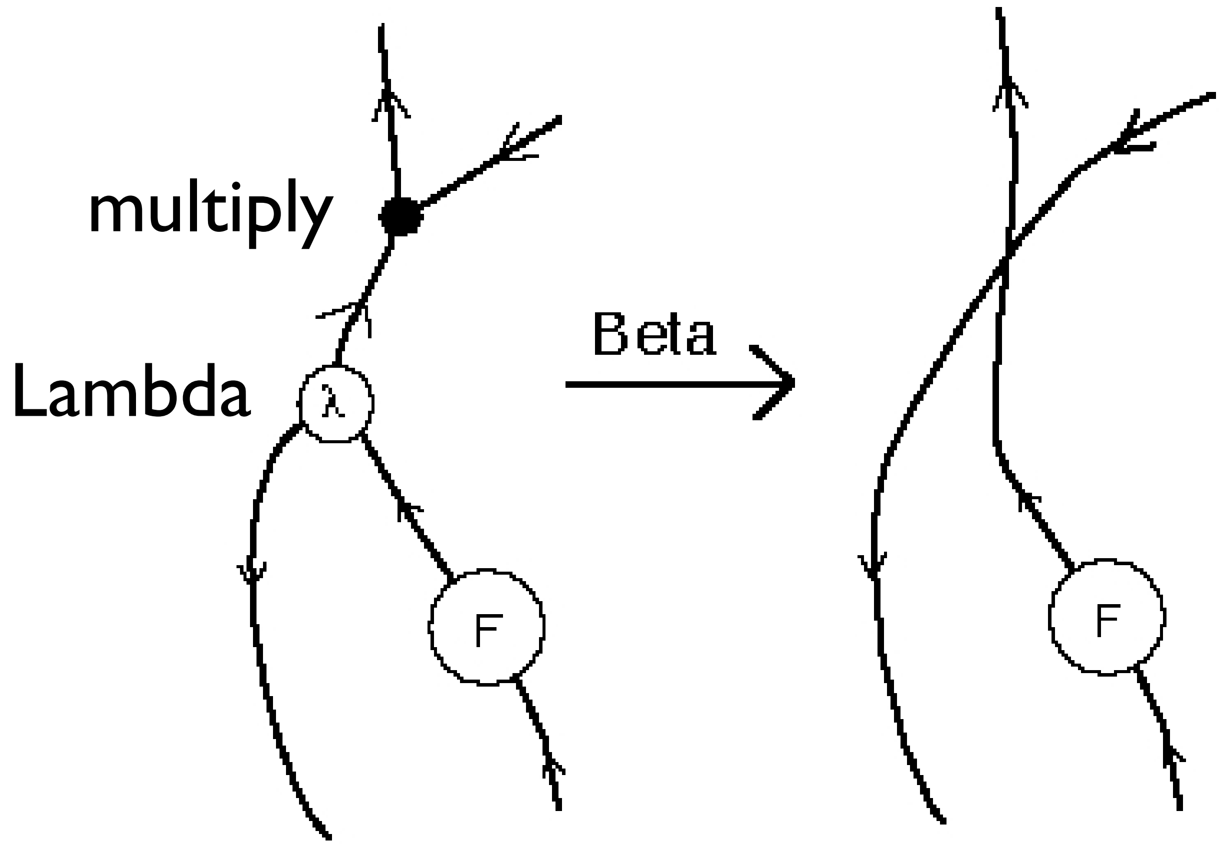
acting on itself.





[Lx.F(x)]A --------> F(A)

In the graphical representation, THERE IS NO VARIABLE X.



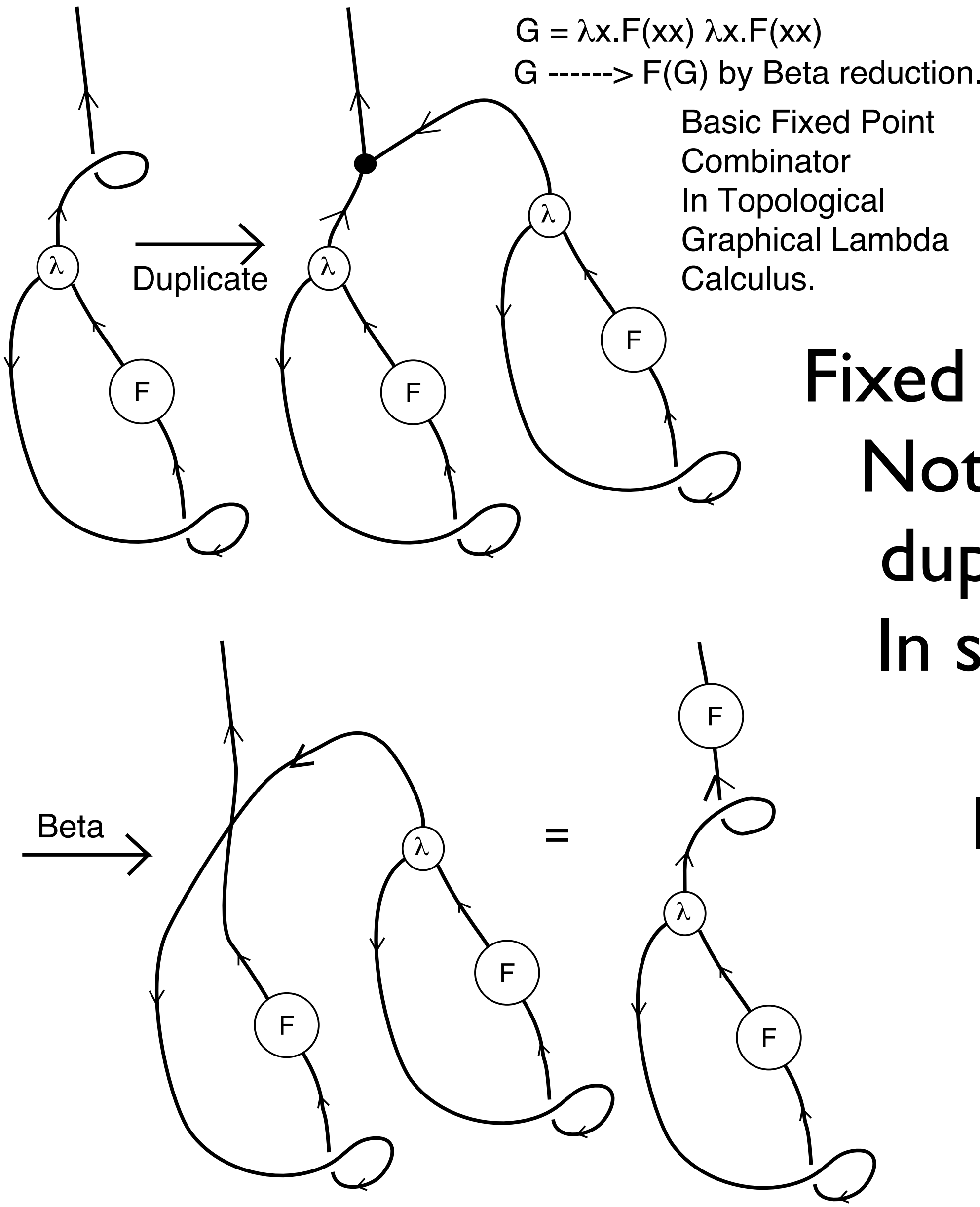
This is our general graphical representation with a multiplication node, a lambda node and an F.

We aim to do lambda calculus and computational generalizations of it by purely graphical, local moves on graphs.

The algebra disappears.

There are no inputs or outputs.

Everything is done by changing local graphical configurations. The actions can happen in a widely distributed network of nodes.



Fixed Point Combinator.
Note the adoption of a
duplication operation.
In some cases this can
be managed by
local operations
(as in DNA).

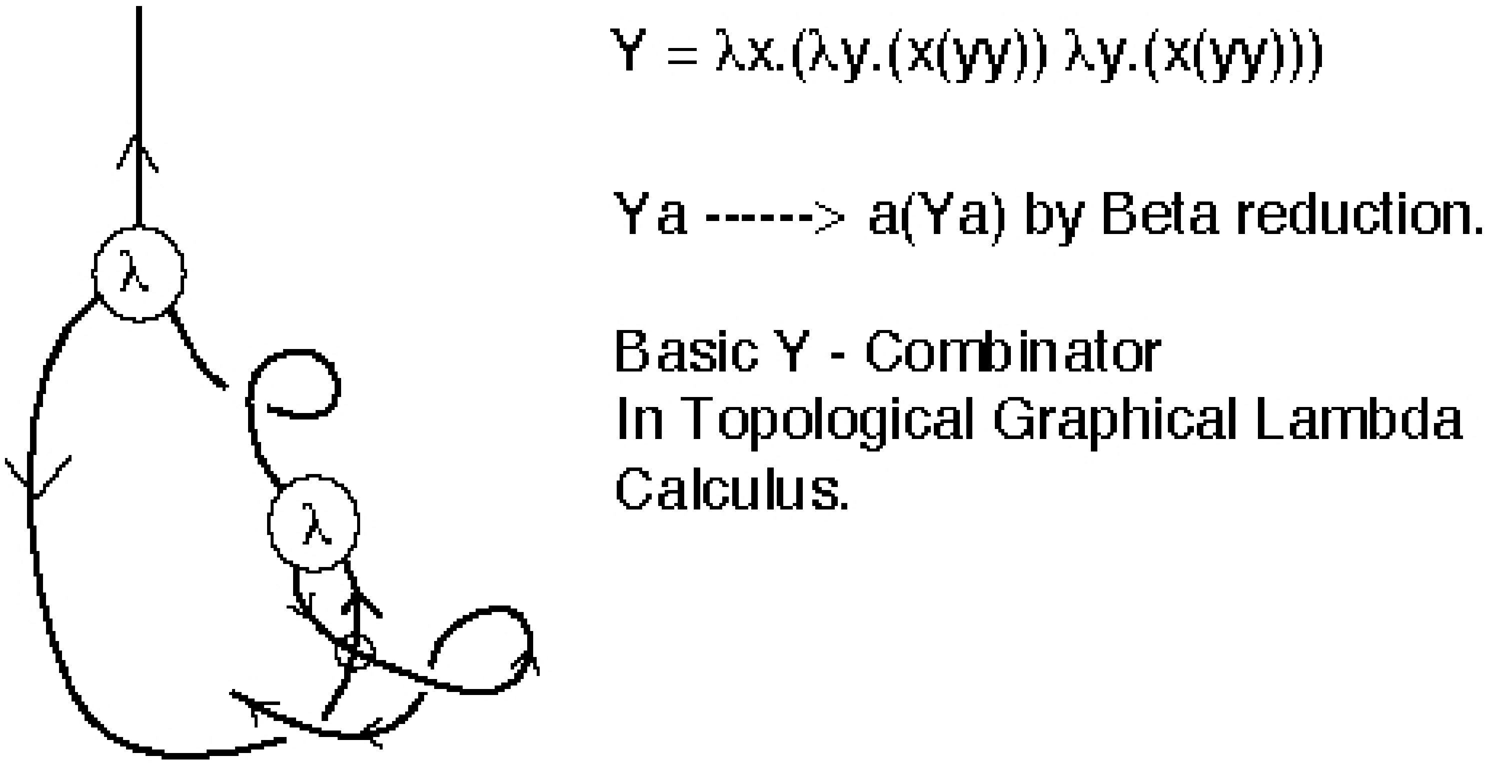


Figure 26: Topological Y - Combinator

Graphic Lambda Calculus

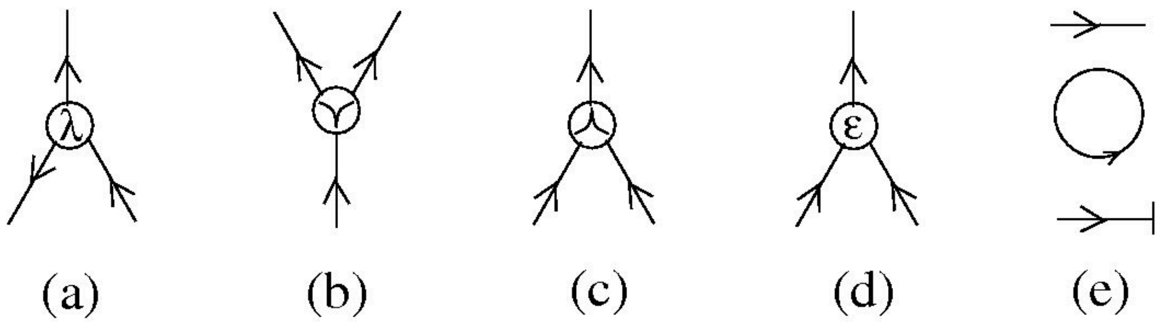


Figure 1: Basic pieces of GLC graphs

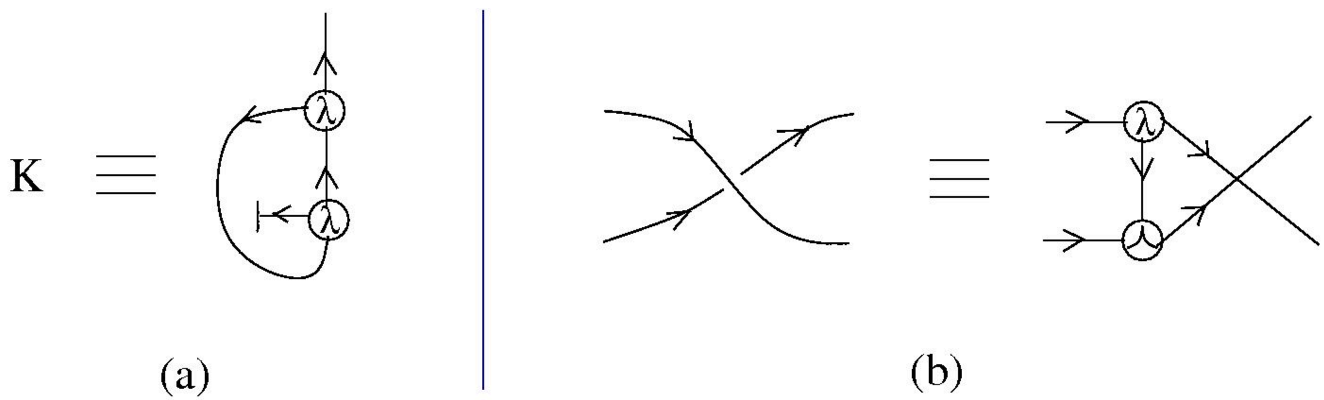


Figure 2: (a) the K combinator, (b) encoding of a crossing in GLC

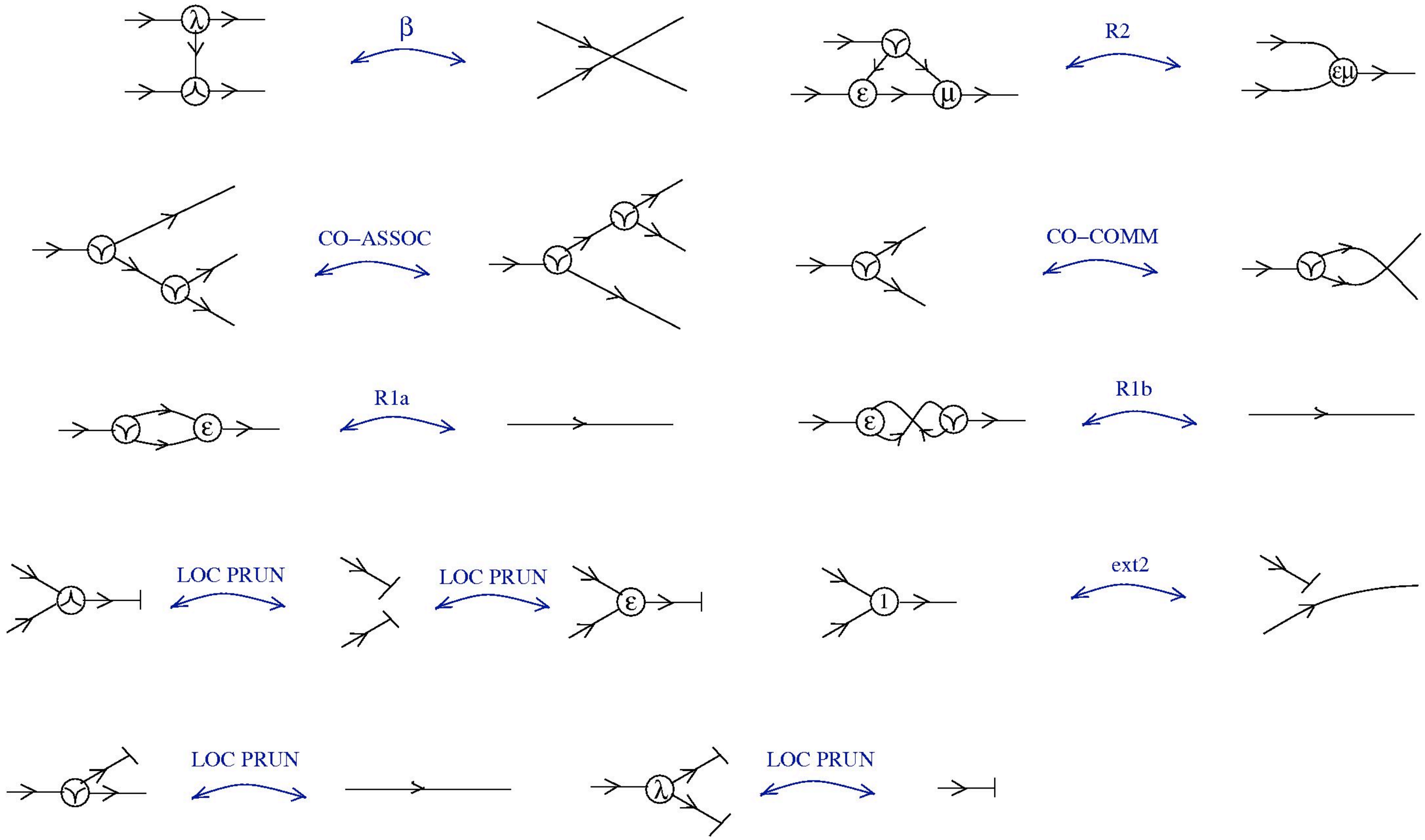


Figure 3: Local moves of GLC

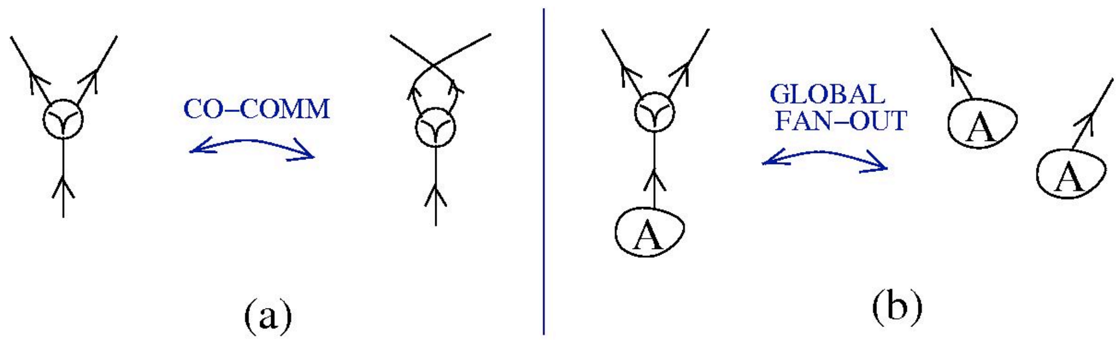
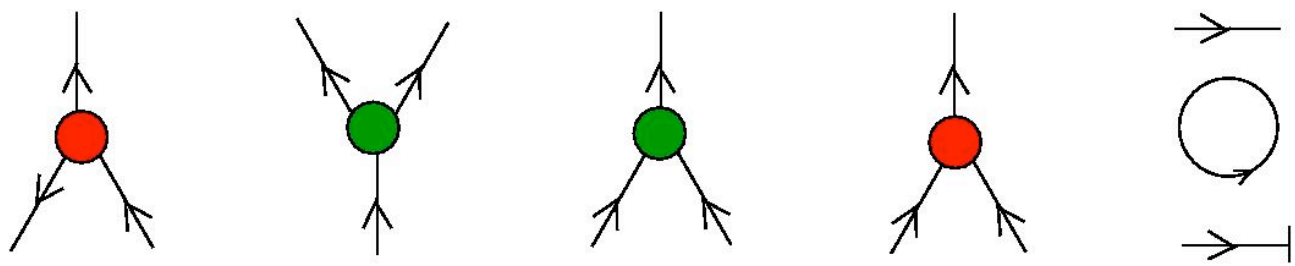
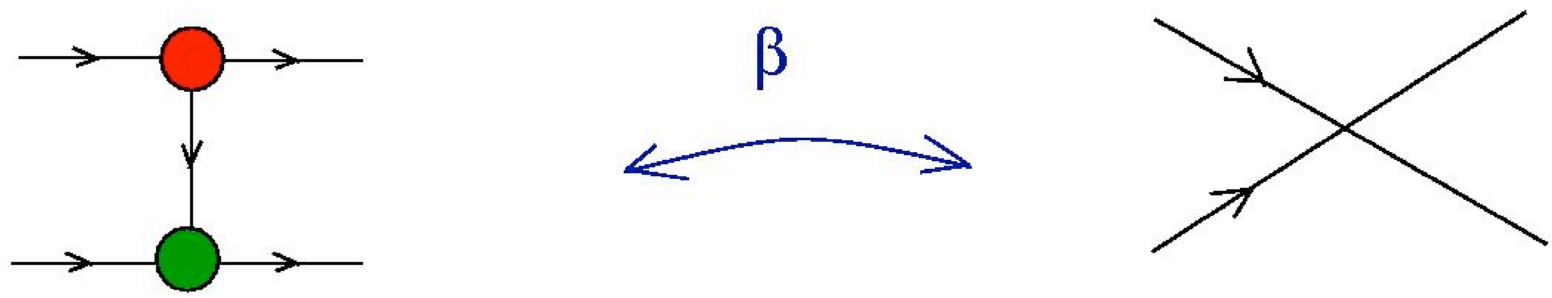


Figure 5: (a) the CO-COMM move is local, (b) the GLOBAL FAN-OUT move is global

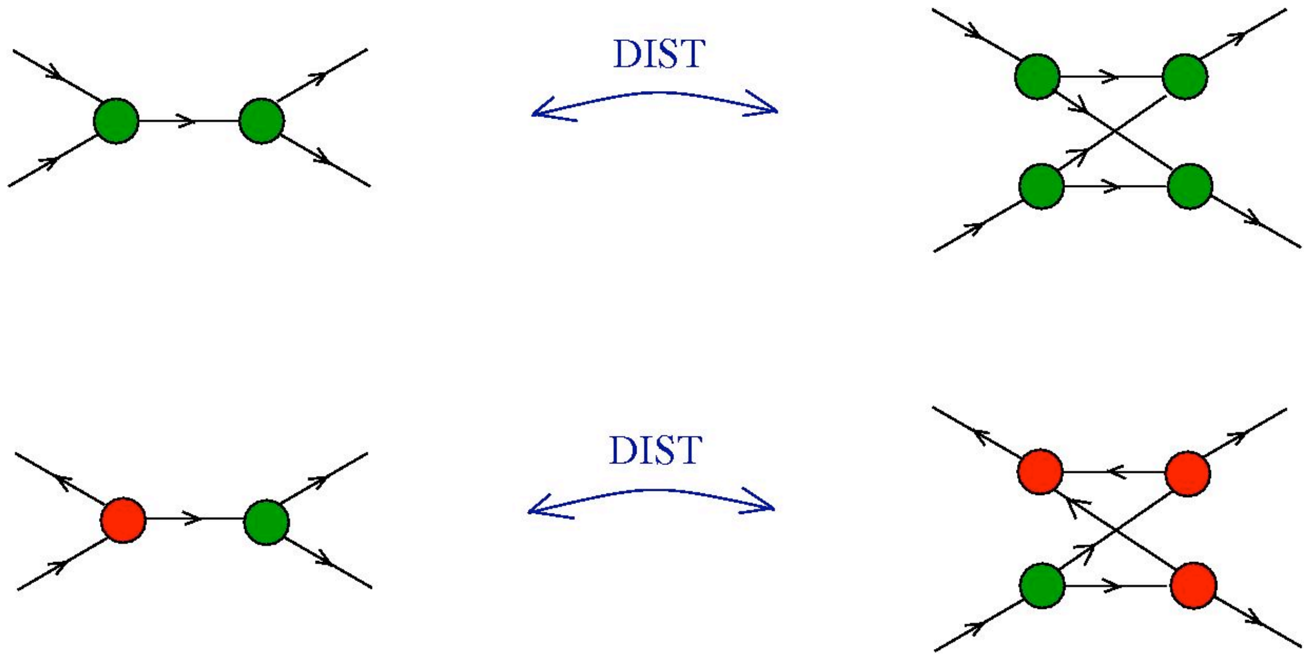
The Chemlambda formalism

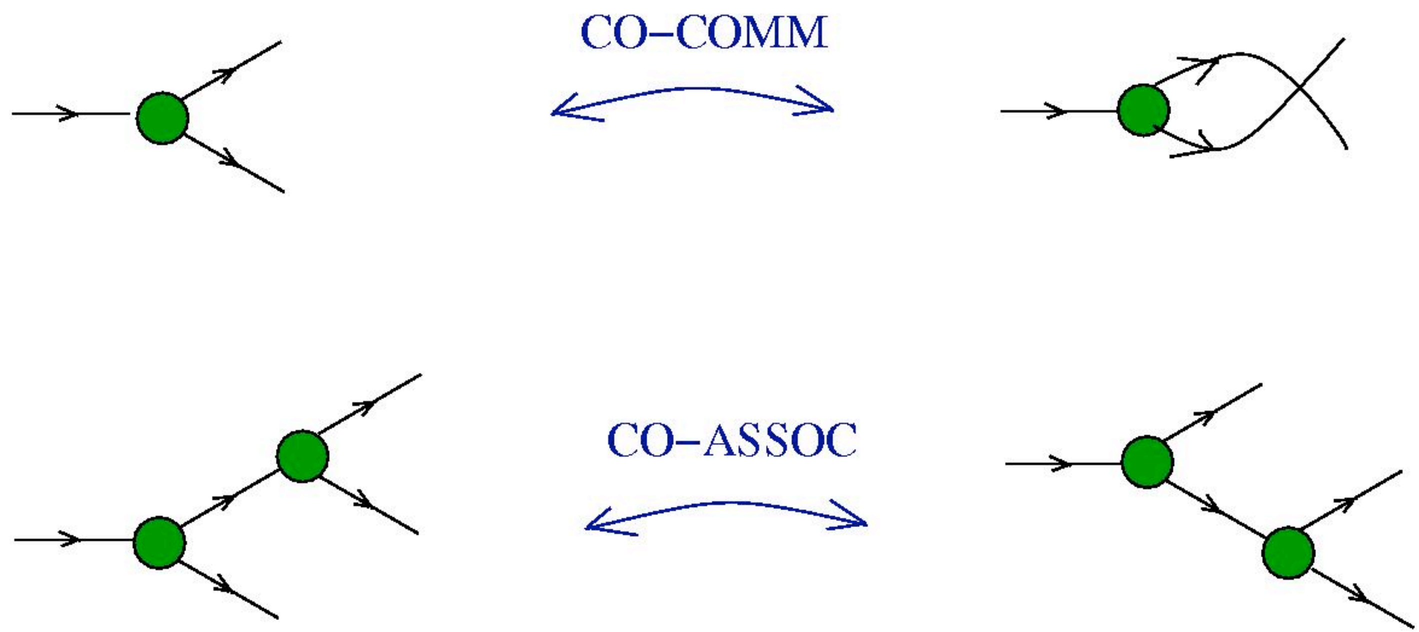


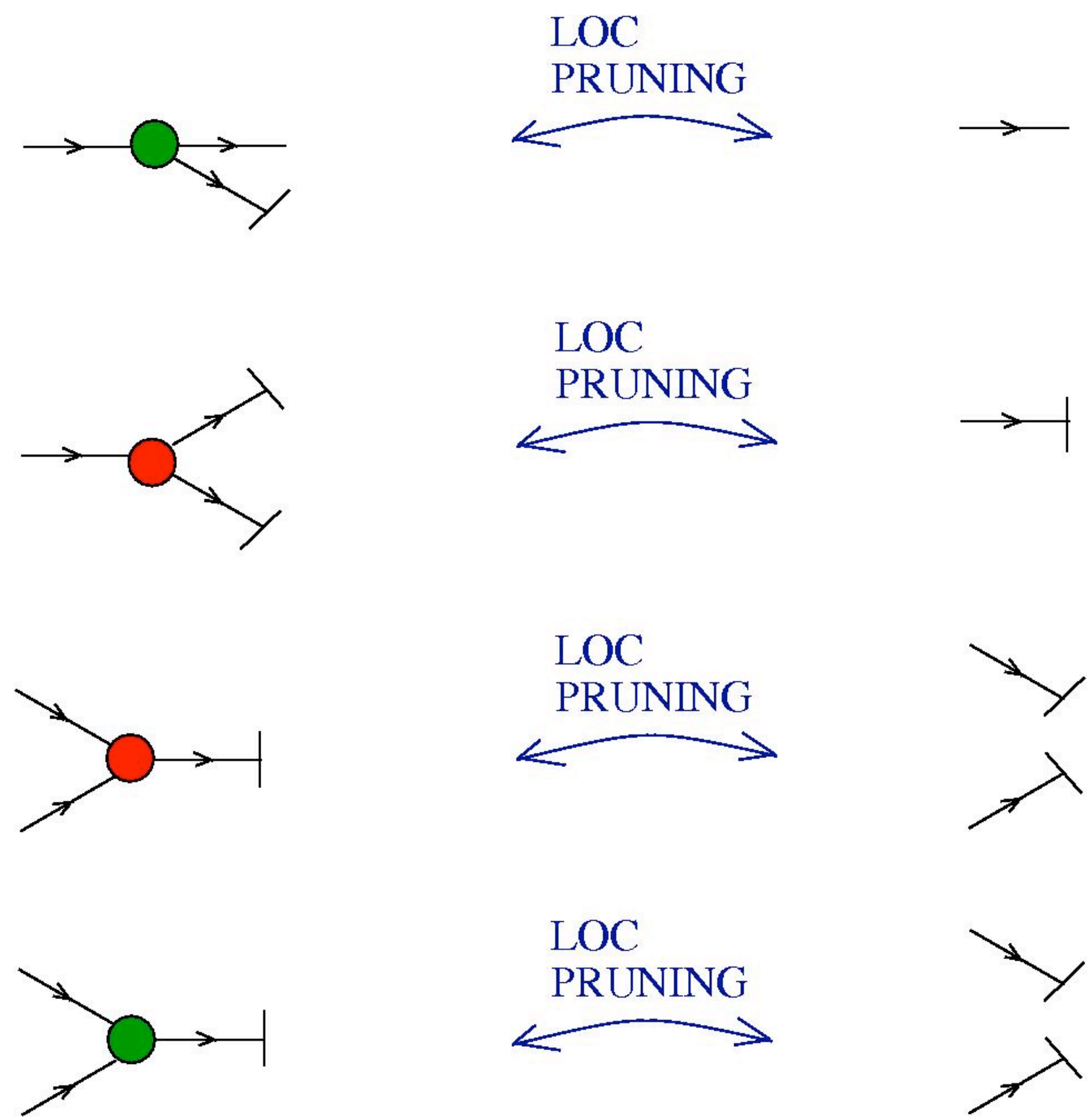
(a) (b) (c) (d) (e)


β


FAN–IN


DIST
DIST


CO−COMM
CO−ASSOC


LOC PRUNING
LOC PRUNING
LOC PRUNING
LOC PRUNING

$$\mathsf{B} \equiv \lambda xyz.x(yz), \quad \mathsf{B}' \equiv \lambda xyz.y(xz), \quad \mathsf{C} \equiv \lambda xyz.xzy,$$
$$\mathsf{I} \equiv \lambda x.x, \quad \mathsf{K} \equiv \lambda xy.x, \quad \mathsf{S} \equiv \lambda xyz.xz(yz),$$
$$\mathsf{W} \equiv \lambda xy.xyy.$$

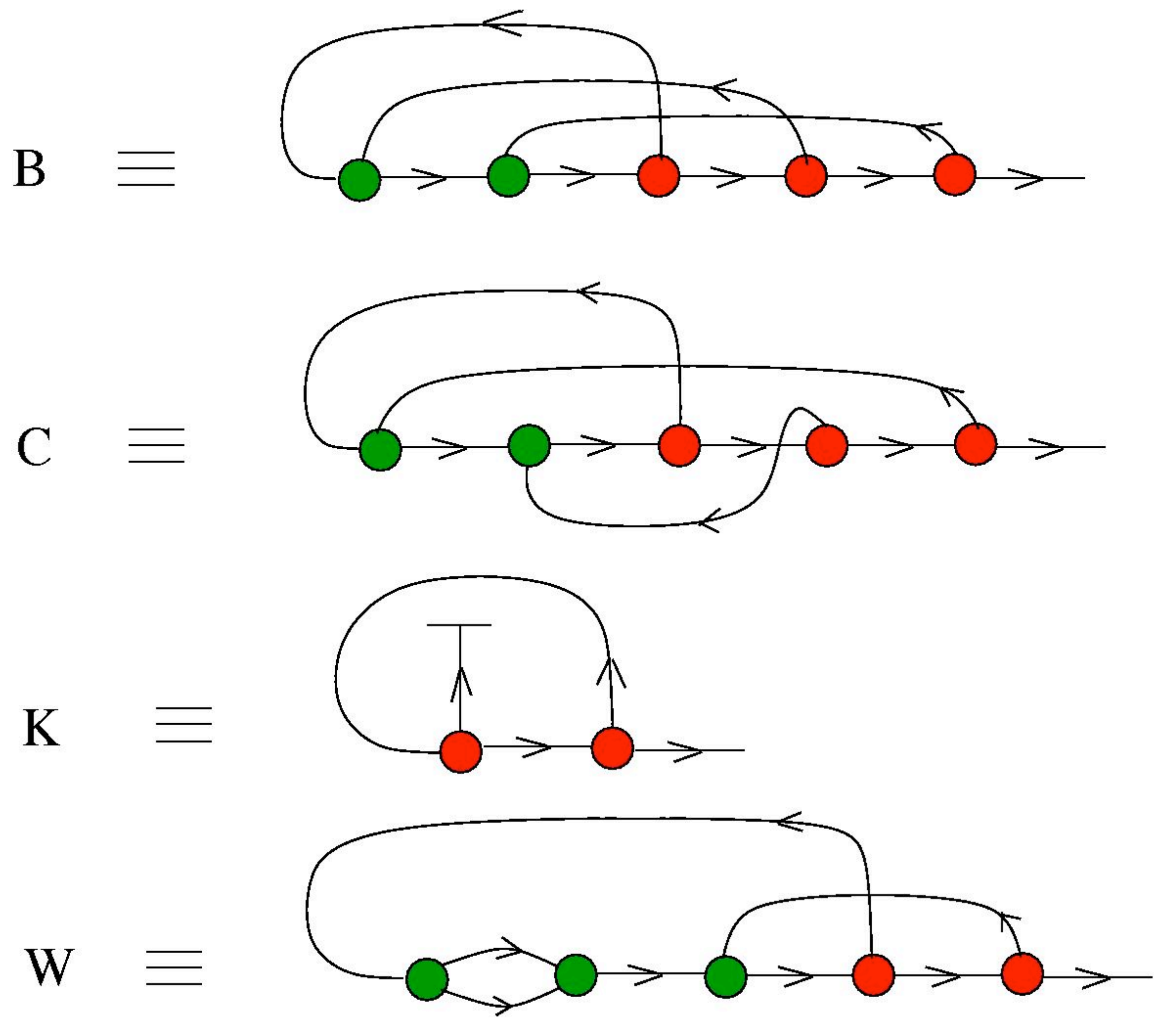
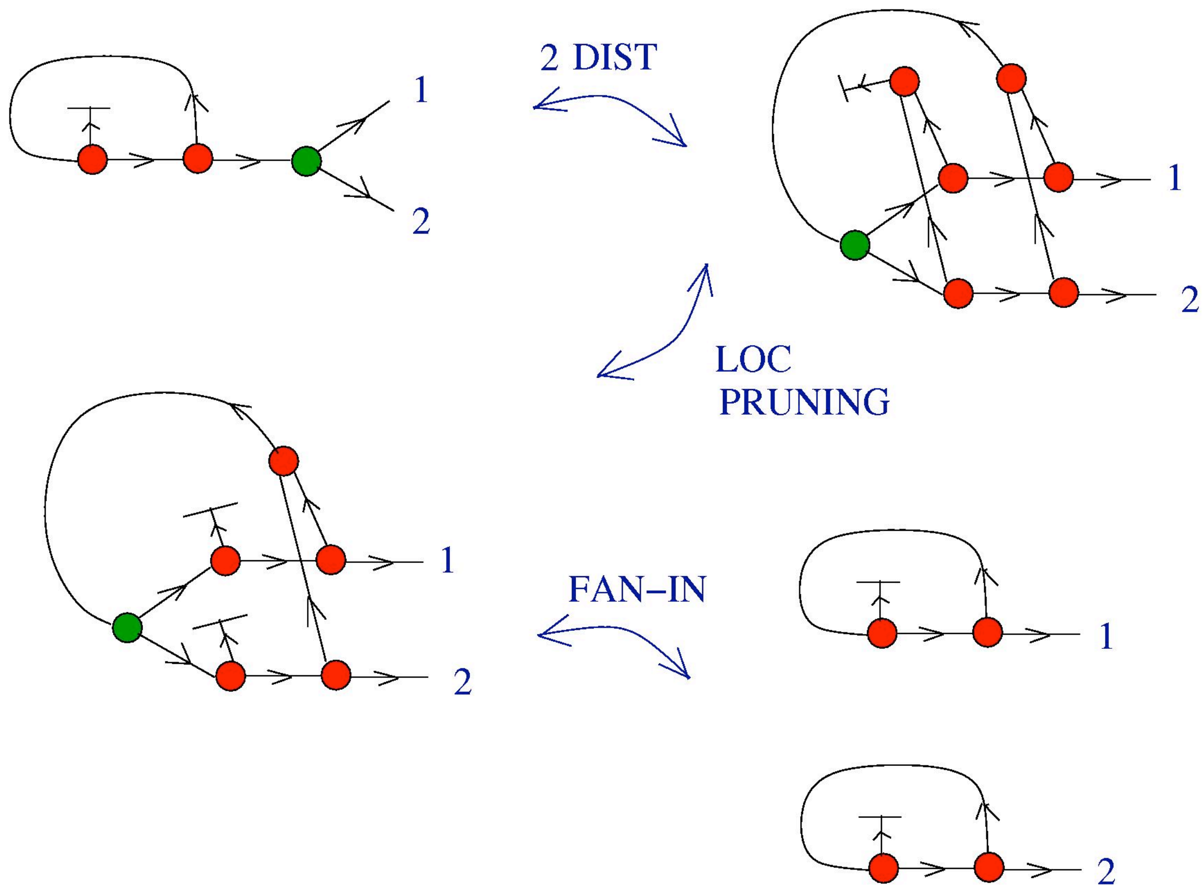


Figure 6: B,C,K,W combinators encoded in chemlambda



2 DIST
LOC
PRUNING
FAN−IN
1
2
1
2
1
2
1
2

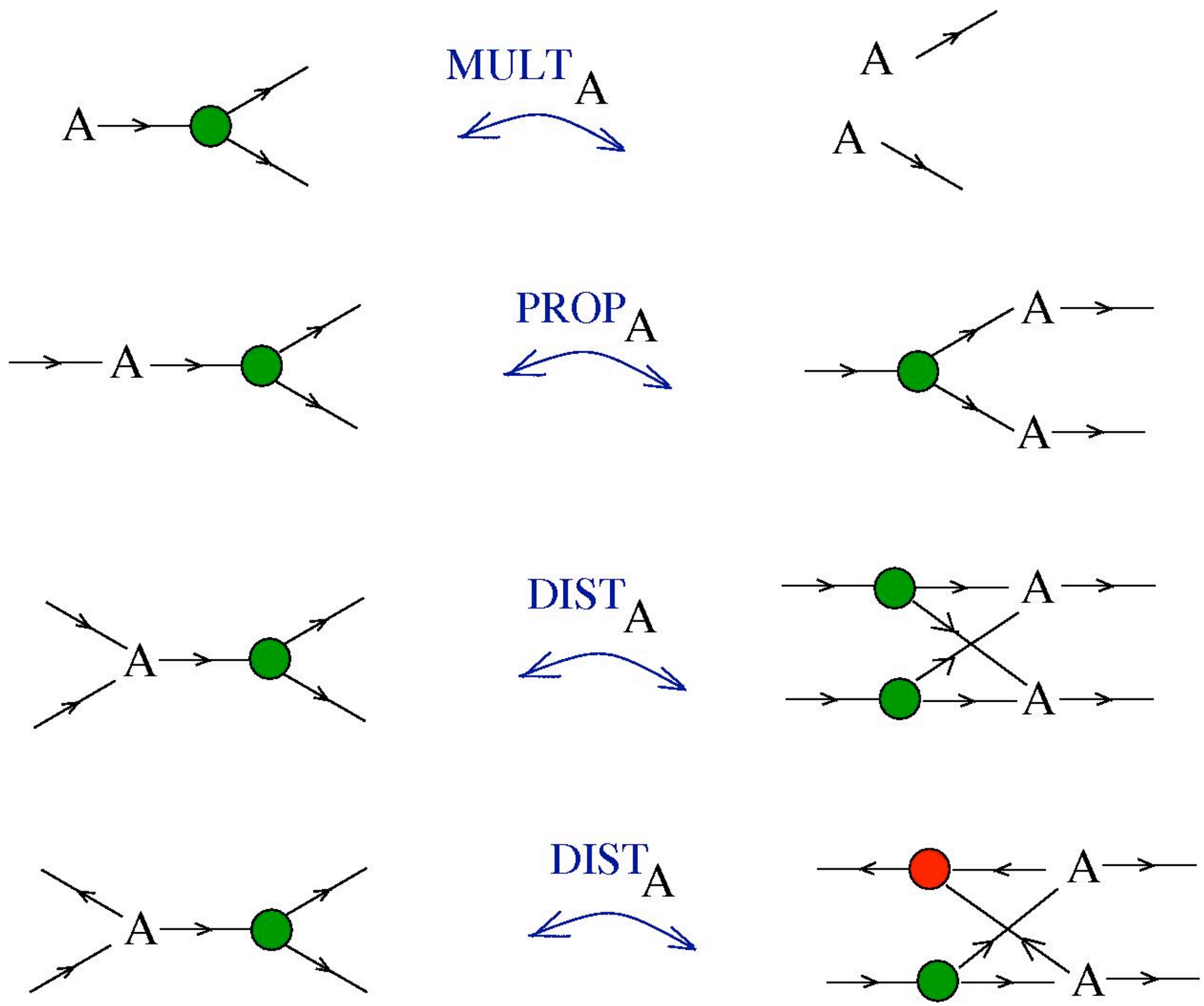


Figure 8: Definition of self-multipliers, propagators, distributors

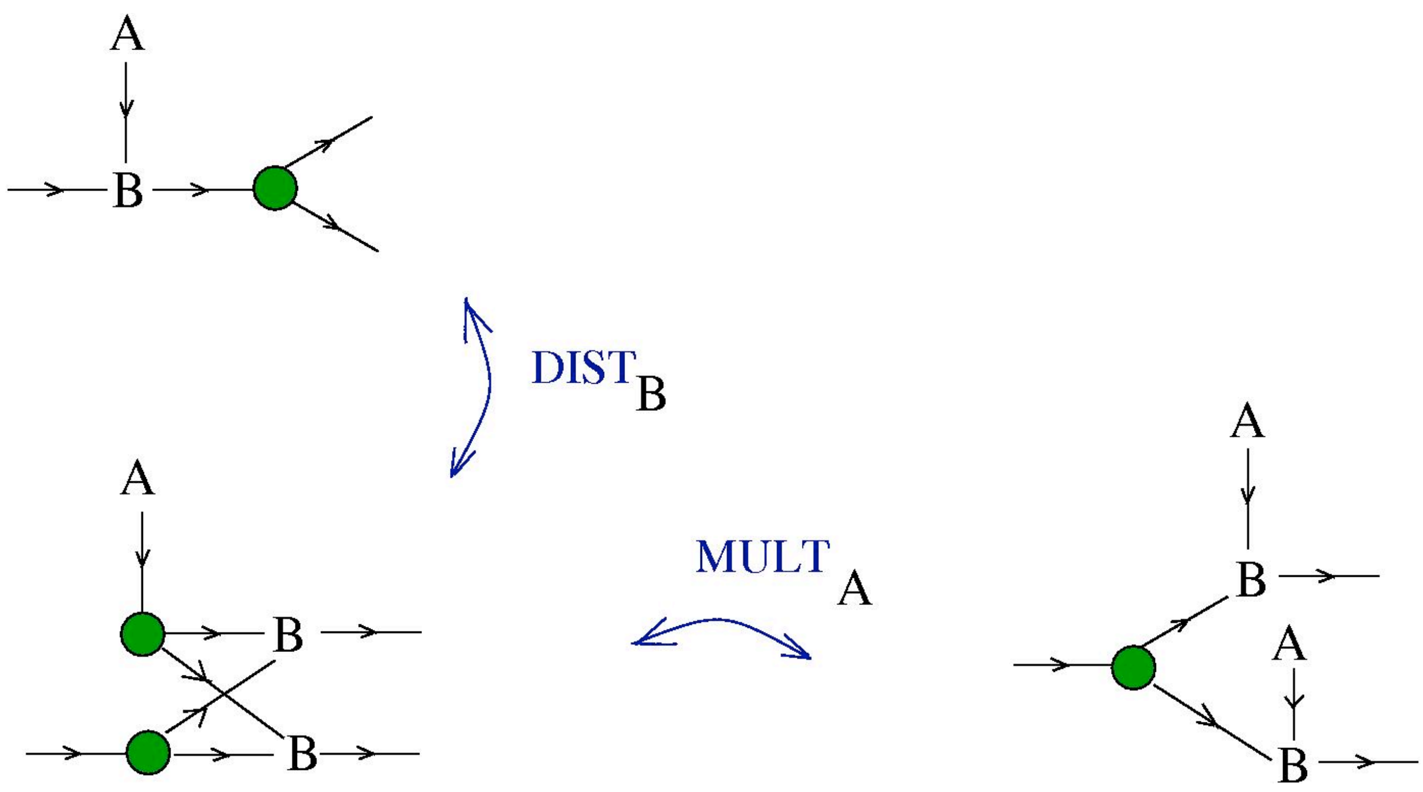


Figure 9: Propagator made from a multiplier and a distributor of the first kind

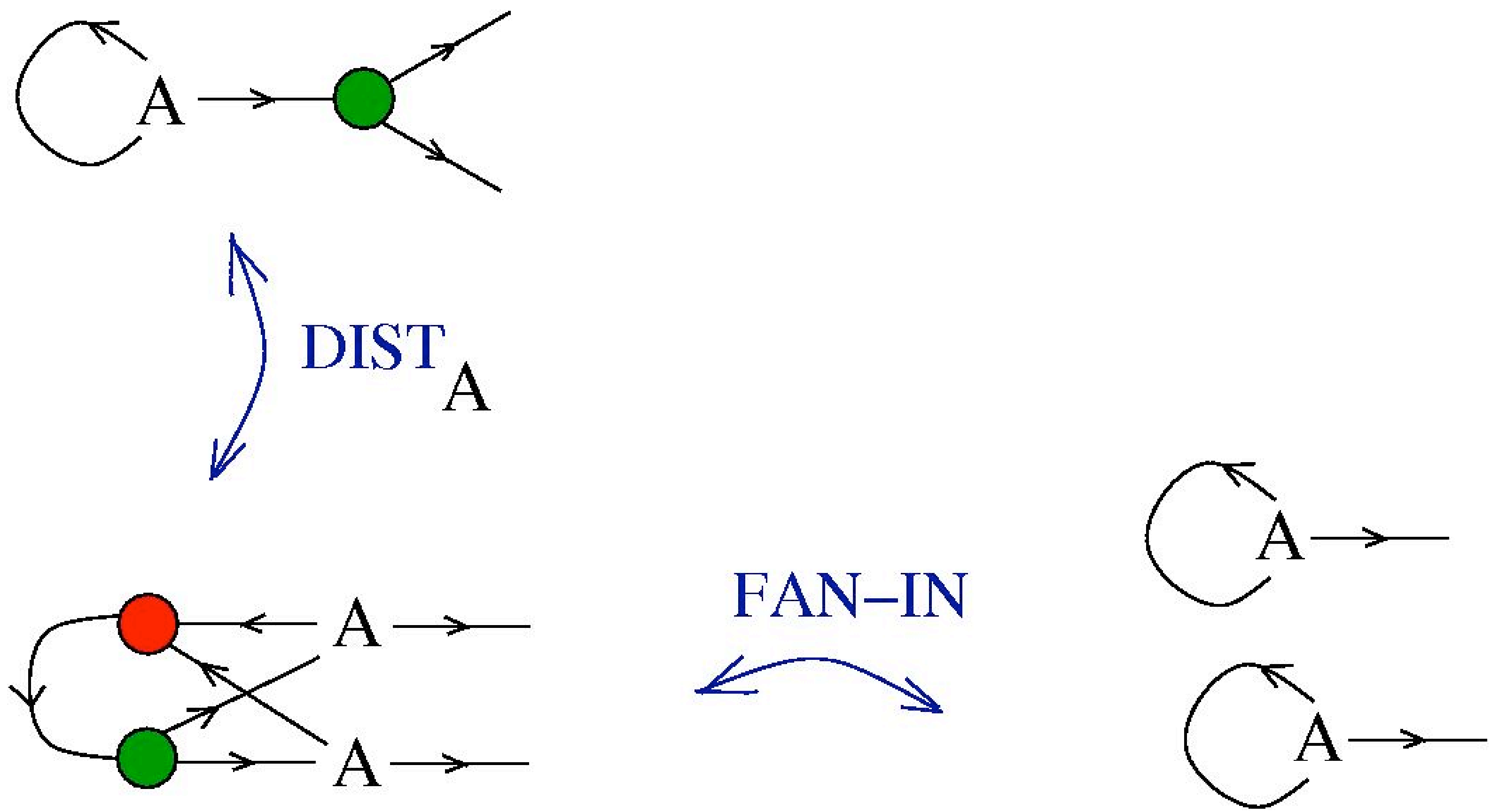


Figure 10: Multiplier made from a distributor of the second kind

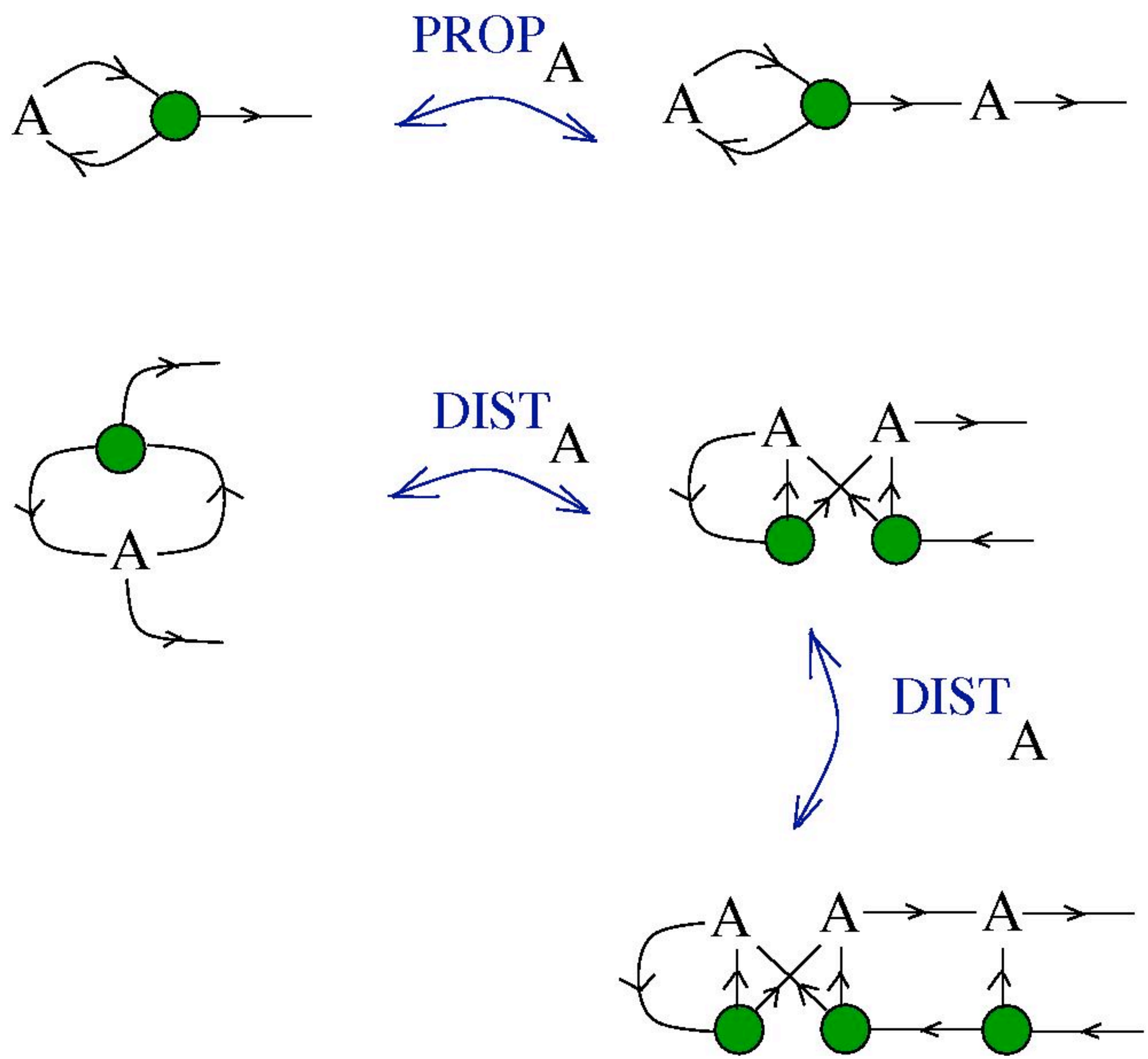


Figure 11: Examples of guns

The Y combinator has the expression

$$Y = \lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$$

and it has the following property: for any lambda term A the expression YA reduces to $A(YA)$. In particular, if A is another combinator, then YA is a fixed-point combinator for A.

In lambda calculus the string of reductions is the following sequence of beta moves:

$$YA \to (\lambda x.A(xx))(\lambda x.A(xx)) \to$$

$$\to A((\lambda x.A(xx))(\lambda x.A(xx))) = A(YA)$$

We see that the during the reduction process we needed a multiplication of the combinator A.

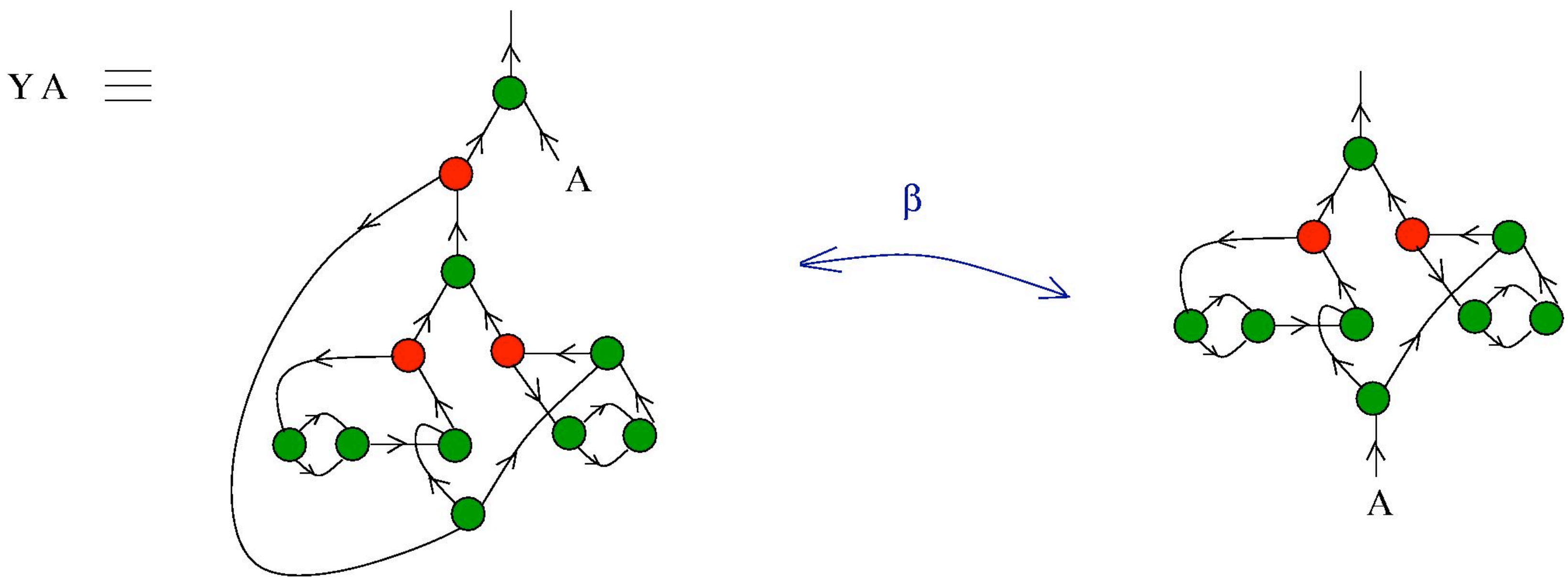


Figure 12: the YA combinator molecule and a first beta move

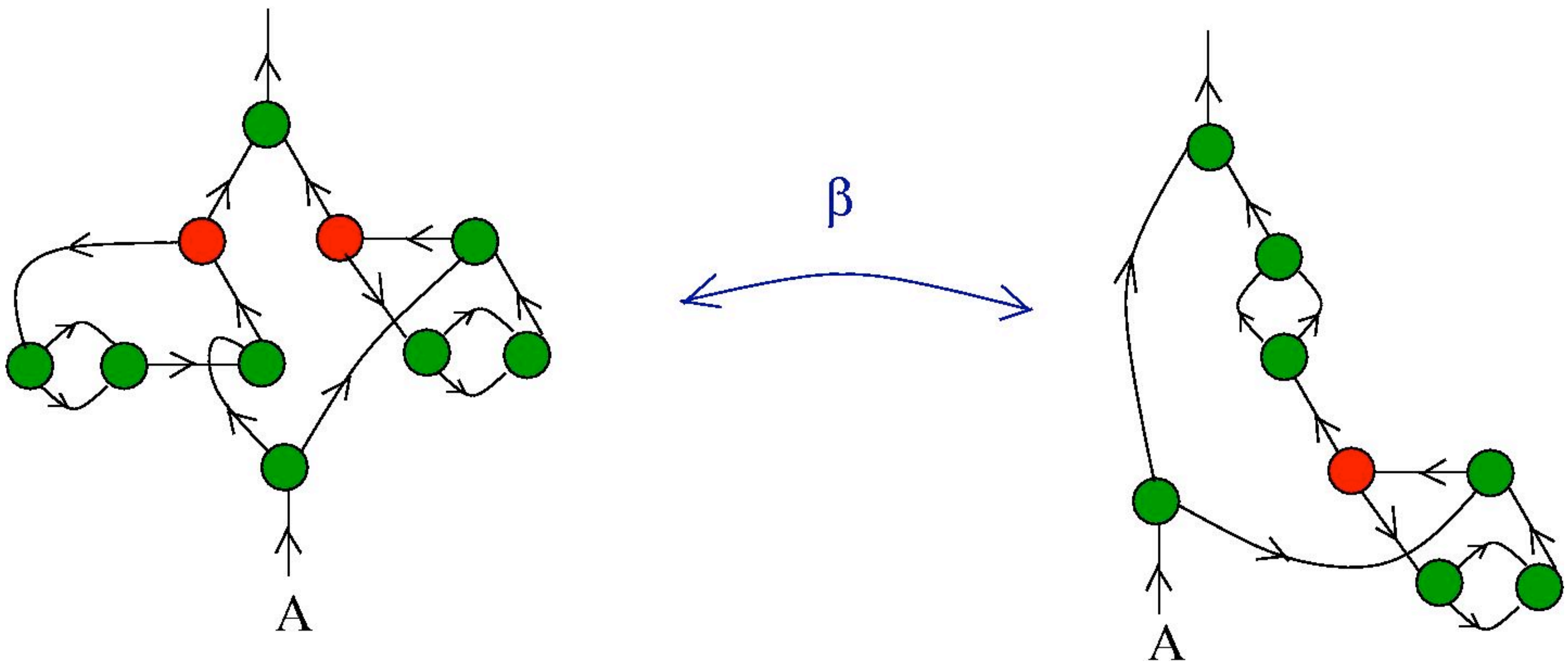


Figure 13: second beta move applied to the $Y A$ molecule

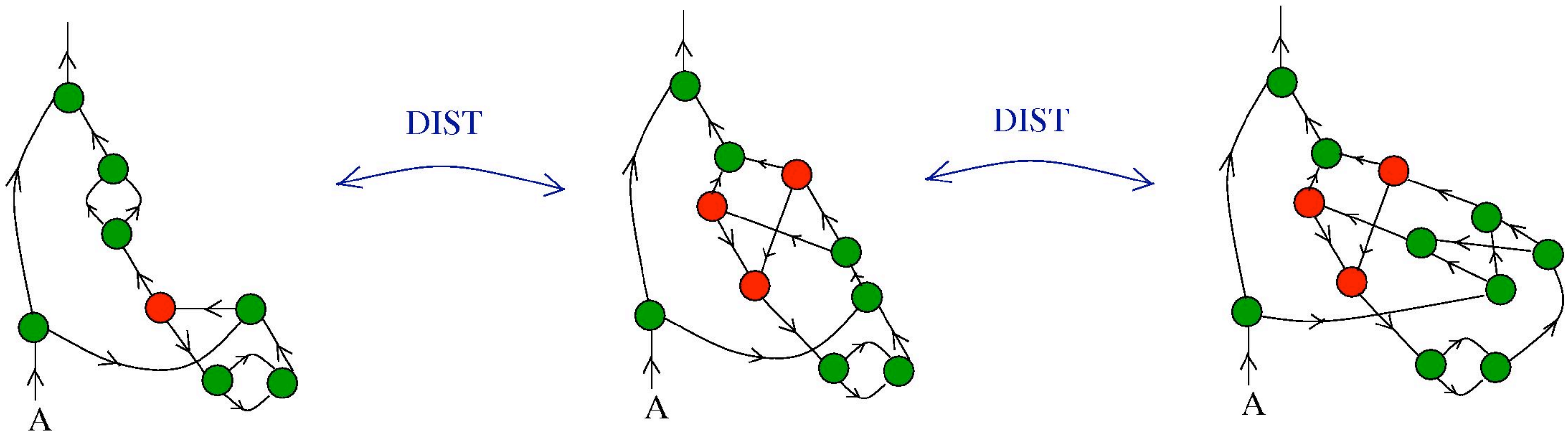


Figure 14: next step of reduction, two DIST moves

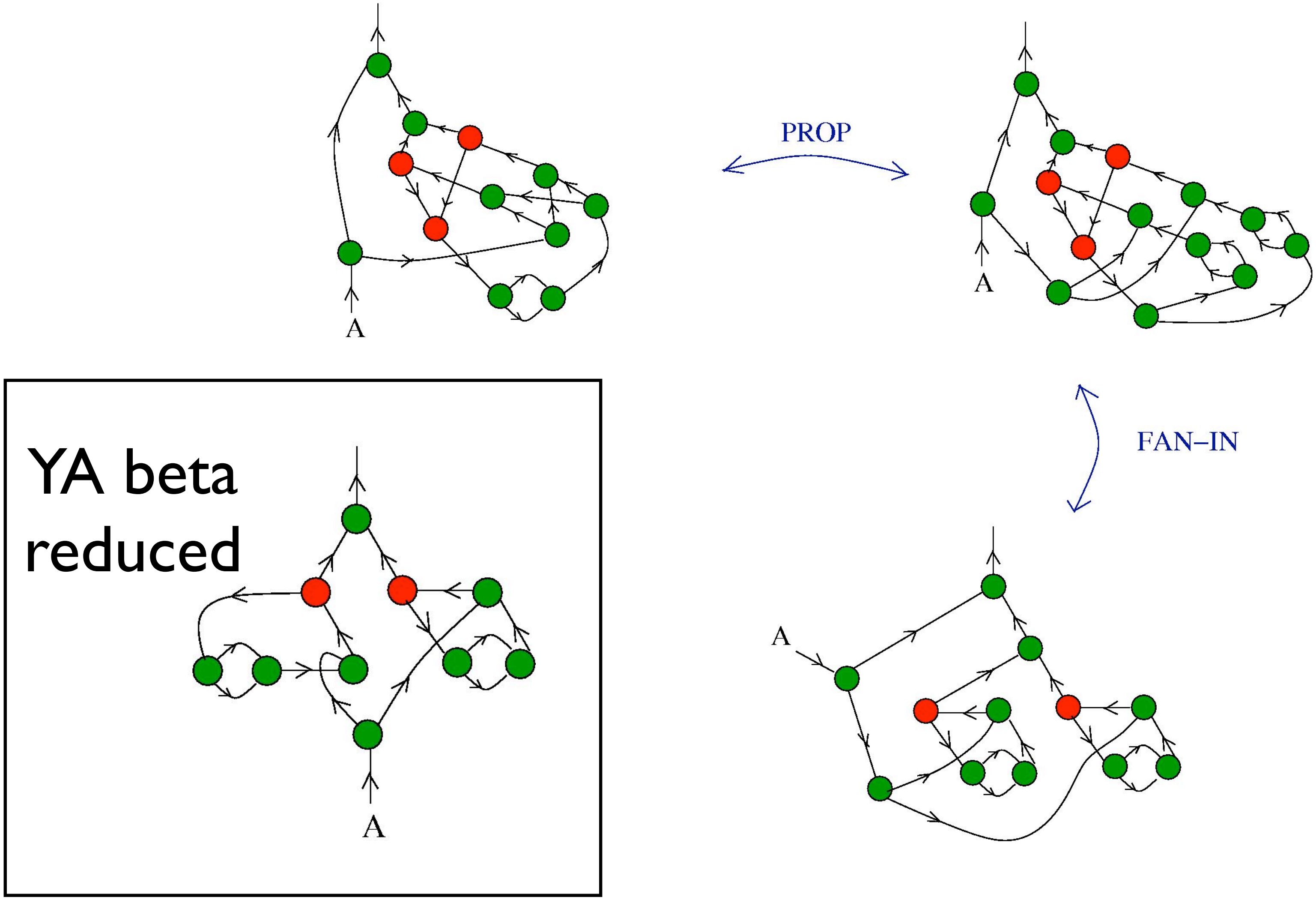


Figure 16: last two moves of the reduction of $Y\,A$ to $A(Y\,A)$

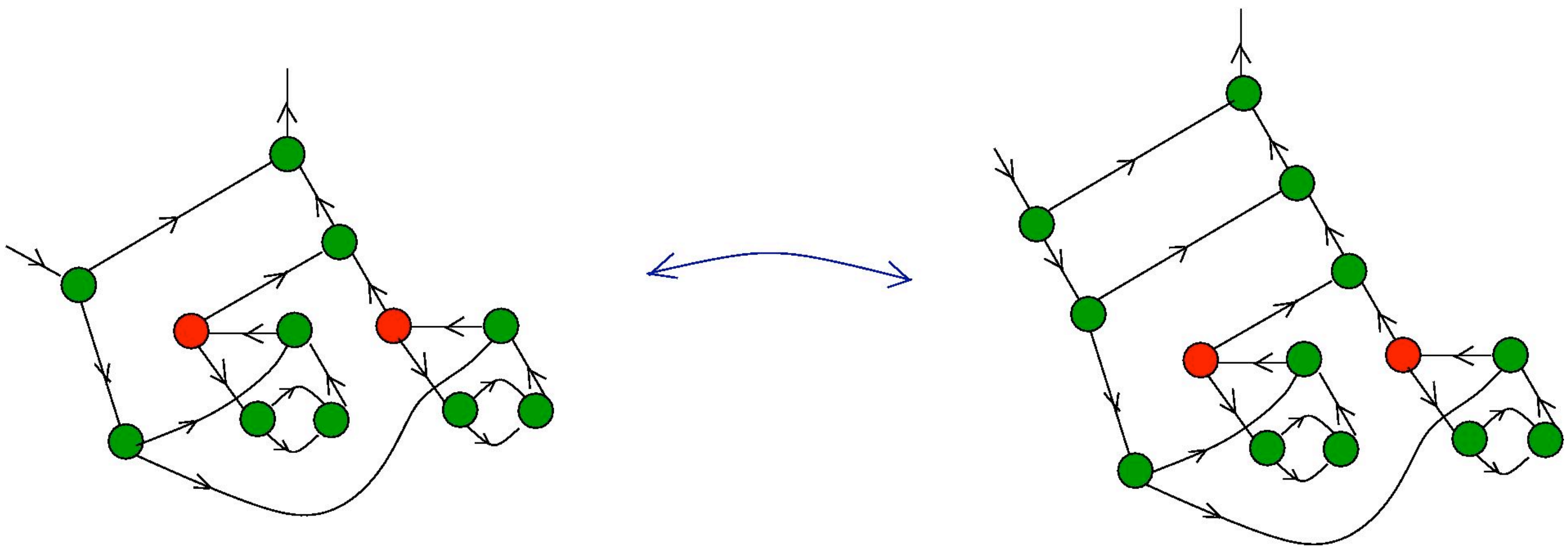


Figure 17: the Y molecule is a gun

Computing

Aim: To do widely distributed computing via chemlambda.

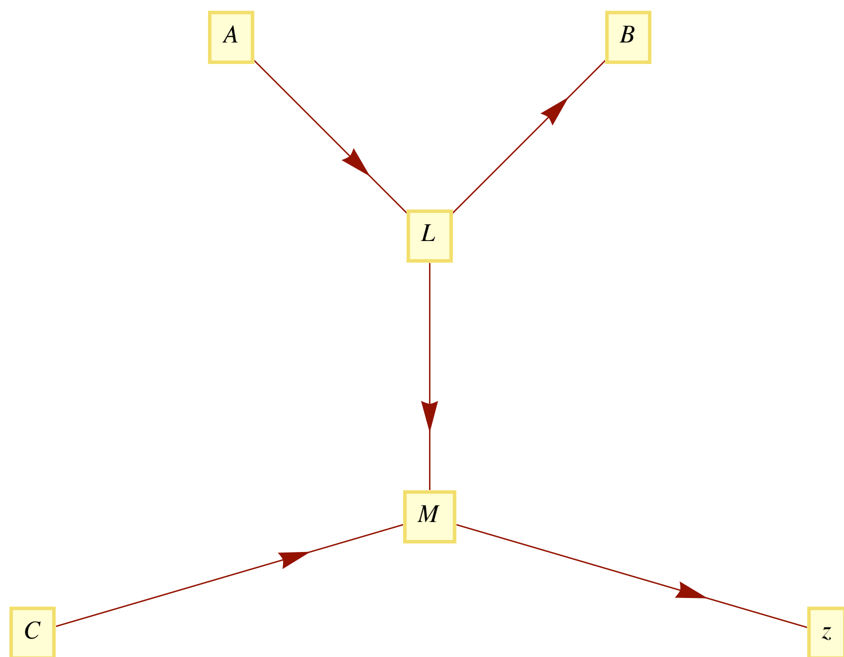
Present Status: Working with toy models.

```
(** Graphical Lambda Calculus **)
(** Here we use mathematica graph representation.
  H[a→b] represents an edge from node a to node b. This
 means that we can translate our rule driven formalism to graphical
 representation by just stripping the H[] from each edge rep. **)

(** Some improvements using Joshua Hermann'
 s suggestion that we input Defer[t]rather than t,
and we have implemented a version of his program that converts the H[a→b] notaton
 directly to a graphical picture. We use PGraf to illustrate an expression directly
 and Graf5 to illustrate the reult of applying the graphical lambda rules to it. **)

(** Apply rules to the graph formalism **)


rule101 = {H[a_ → L] H[L → b_] H[L → M] H[M → d_] H[c_ → M] :> H[c → b] H[a → d]};
rule102 = {H[a_ → x_] H[x_ → Fan] H[ Fan →  b_] H[ Fan →  c_] :>
    H[a  → Fan] H[Fan  → x ] H[x  → b] H[x  → c]};
rule103 = {H[x_ → Fan] H[ Fan →  b_] H[ Fan →  c_] :> H[Fan  → x ] H[x  → b] H[x  → c]};
rule104 = {H[a_ → x_] H[x_ → Fan] H[ Fan →  b_] H[ Fan →  b_] :>
    H[a  → Fan] H[Fan  → x ] H[x  → b] H[x  → b]};
rule105 = {H[x_ → Fan] H[ Fan →  b_] H[ Fan →  b_] :> H[Fan  → x ] H[x  → b] H[x  → b]};
rule106 = {H[x_ → Fan] H[ Fan →  b_] :> H[Fan  → x ] H[x  → b]};
rule107 = {H[a_ → L] H[L → b_] H[L → MM] H[MM → d_] H[c_ → MM] :> H[c → b] H[a → d]};
(**rule108={H[a_→x]H[x_ →Fan}H[Fan →M}:> H[x_ →M}H[x_ →M}H[a_→Fan]} **)

rule111 = {H[a_ → LL] H[LL → b_] H[LL → M] H[M → d_] H[c_ → M] :> H[c → b] H[a → d]};
rule112 = {H[a_ → x_] H[x_ → FFan] H[ FFan →  b_] H[ FFan →  c_] :>
    H[a  → FFan] H[FFan  → x ] H[x  → b] H[x  → c]};
rule113 = {H[x_ → FFan] H[ FFan →  b_] H[ FFan →  c_] :> H[FFan  → x ] H[x  → b] H[x  → c]};
rule114 = {H[a_ → x_] H[x_ → FFan] H[ FFan →  b_] H[ FFan →  b_] :>
    H[a  → FFan] H[FFan  → x ] H[x  → b] H[x  → b]};
rule115 = {H[x_ → FFan] H[ FFan →  b_] H[ FFan →  b_] :> H[FFan  → x ] H[x  → b] H[x  → b]};
rule116 = {H[x_ → FFan] H[ FFan →  b_] :> H[FFan  → x ] H[x  → b]};
rule117 = {H[a_ → LL] H[LL → b_] H[LL → MM] H[MM → d_] H[c_ → MM] :> H[c → b] H[a → d]};

PGraf[x_] :=
 Show[GraphPlot[Last[Last[Reap[Evaluate[x //. H → Sow][[1]]]]], DirectedEdges → True,
   VertexLabeling → True], ImageSize → Medium]

SGraf[x_] := Last[Last[Reap[Evaluate[x //. H → Sow][[1]]]]]

Graf5[x_] :=
 Show[GraphPlot[Last[Last[Reap[Evaluate[Graf[x] //. H → Sow][[1]]]]], DirectedEdges → True,
   VertexLabeling → True], ImageSize → Medium]

Graf[t_] :=
 Simplify[Defer[t] //. rule101 //. rule102 //. rule103 //. rule104 //. rule105 //.
           rule106 //. rule107 //. rule111 //. rule112 //.
       rule113 //. rule114 //. rule115 //. rule116 //. rule117 ]
```

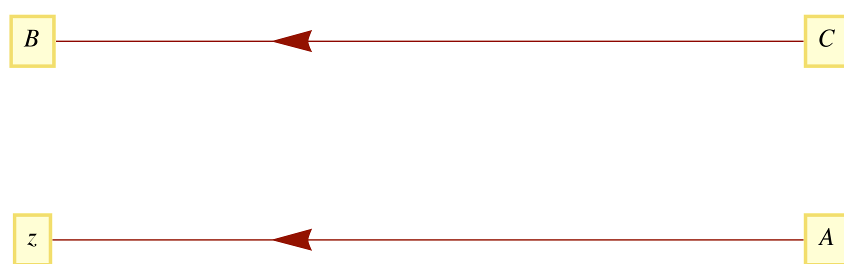
```
t = H[A → L] H[L → B] H[L → M] H[C → M] H[M → z];
Graf[t]
PGraf[t]
Graf5[t]
```

ut[32]= H[C → B] H[A → z]

ut[33]=



ut[34]=

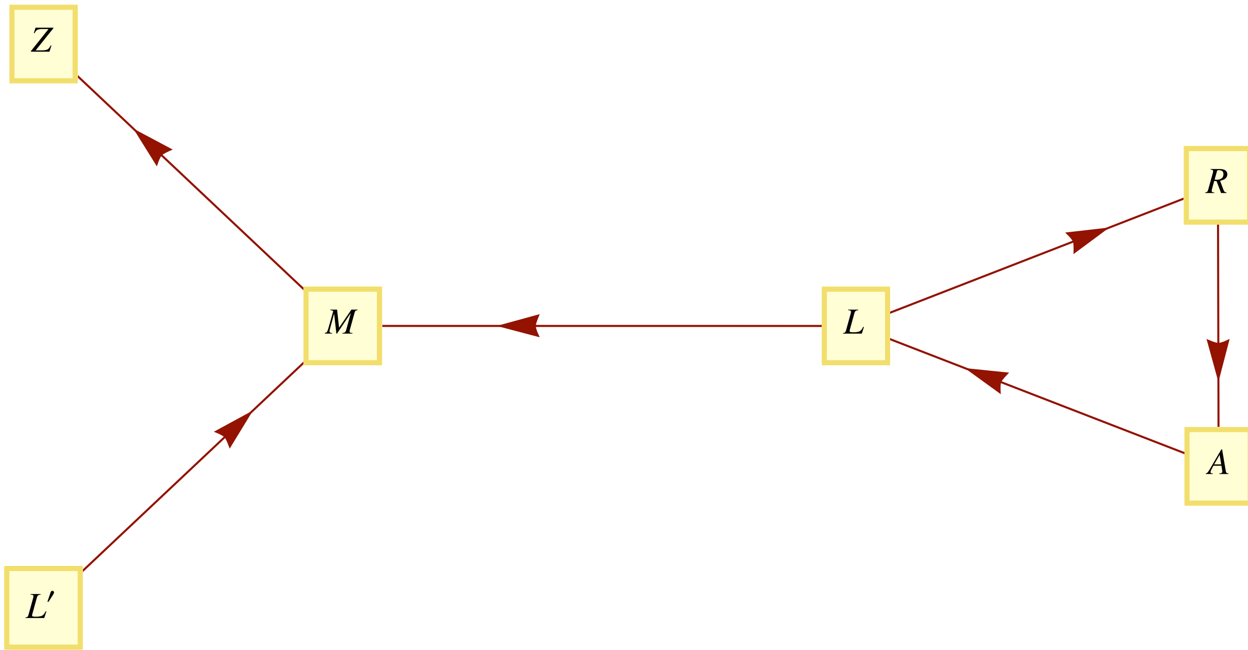


In[109]:=

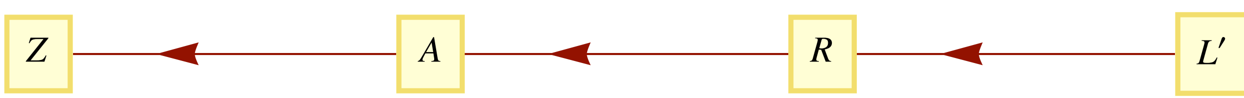
```
t =  H[L → R] H[R → A] H[A → L] H[L → M] H[L ' → M]  H[M → Z];
Graf[t]
PGraf[t]
Graf5[t]
```

Out[110]= H[R → A] (H[L′ → R] H[A → Z])

Out[111]=

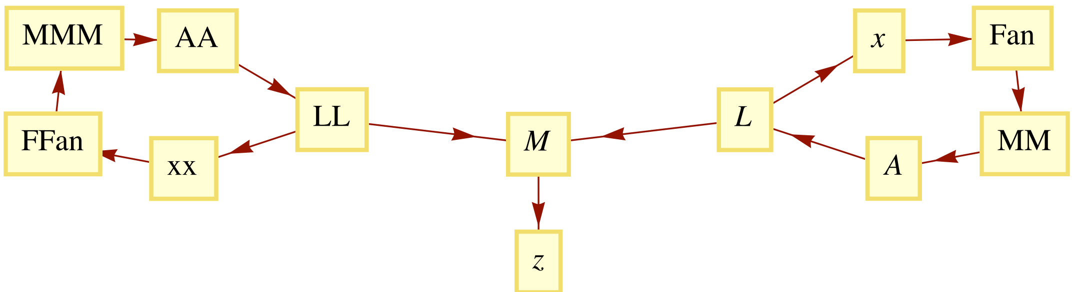


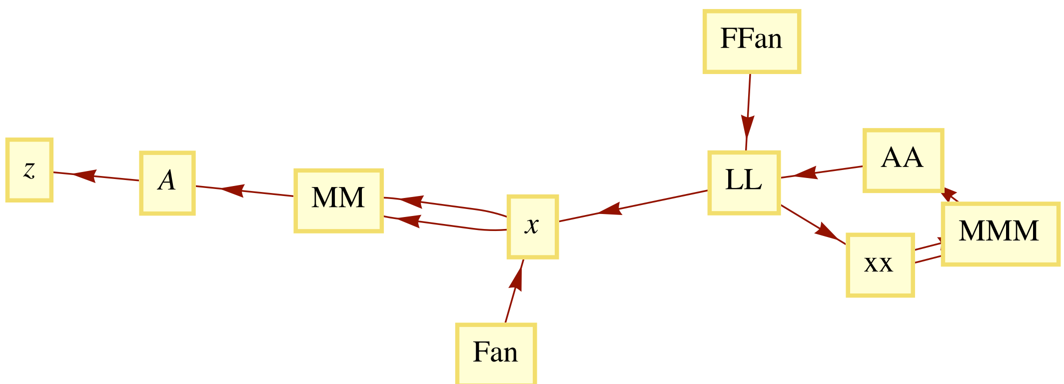
Out[112]=



```
t = H[L → x] H[x → Fan] H[Fan → MM] H[Fan → MM] H[MM → A] H[A → L] H[L → M] H[M → z]
    H[LL → xx] H[xx → FFan] H[FFan → MMM] H[FFan → MMM] H[MMM → AA] H[AA → LL] H[LL → M];
Graf[t]
PGraf[t]
Graf5[t]
```

```
H[AA → LL] H[MM → A] H[MMM → AA] (H[LL → x] H[A → z])
  (H[Fan → x] H[x → MM] H[x → MM]) (H[xx → MMM] H[xx → MMM] (H[FFan → LL] H[LL → xx]))
```





There is more to come.

The main point is that graphical lambda calculus and chemlambda can be done by
local
asynchronous operations
on widely distributed graphs.
Hence the possibility of
global and secure
computations
in this mode.

The connections with topology deserve deeper investigation.

Thank You!