Graphic Lambda Calculus

Marius Buliga

Institute of Mathematics of the Romanian Academy
P.O. Box 1-764, RO 014700
Bucharest, Romania
Marius.Buliga@imar.ro

Graphic lambda calculus, a visual language that can be used for representing untyped lambda calculus, is introduced and studied. It can also be used for computations in emergent algebras or for representing Reidemeister moves of locally planar tangle diagrams.

1. Introduction

Graphic lambda calculus consists of a class of graphs endowed with moves between them. It might be considered a visual language in the sense of Erwig [1]. The name comes from the fact that it can be used for representing terms and reductions from untyped lambda calculus. Its main move is called the graphic beta move for its relation to the beta reduction in lambda calculus. However, the graphic beta move can be applied outside the "sector" of untyped lambda calculus, and the graphic lambda calculus can be used for other purposes than that of visually representing lambda calculus.

For other visual, diagrammatic representations of lambda calculus see the VEX language [2], or Keenan's website [3].

The motivation for introducing graphic lambda calculus comes from the study of emergent algebras. In fact, my goal is to eventually build a logic system that can be used for the formalization of certain "computations" in emergent algebras. The system can then be applied for a discrete differential calculus that exists for metric spaces with dilations, comprising Riemannian manifolds and sub-Riemannian spaces with very low regularity.

Emergent algebras are a generalization of quandles; namely, an emergent algebra is a family of idempotent right quasigroups indexed by the elements of an Abelian group, while quandles are self-distributive idempotent right quasigroups. Tangle diagrams decorated by quandles or racks are a well-known tool in knot theory [4, 5].

In Kauffman [6] knot diagrams are used for representing combinatory logic, thus forming a graphical notation for untyped lambda cal-

Complex Systems, **22** © 2013 Complex Systems Publications, Inc.
https://doi.org/10.25088/ComplexSystems.22.4.311

culus terms. Also, Meredith and Snyder [7] associate to any knot diagram a process in pi-calculus.

Is there any common ground between these three apparently separate fields, namely, differential calculus, logic, and tangle diagrams? As a first attempt for understanding this, I proposed λ-scale calculus [8], which is a formalism containing both untyped lambda calculus and emergent algebras. Also, in [9] I proposed a formalism of decorated tangle diagrams for emergent algebras and I called "computing with space" the various manipulations of these diagrams with geometric content. Nevertheless, in that paper I was not able to give a precise sense of the use of the word "computing." I speculated, by using analogies from studies of the visual system, especially the "brain as a geometry engine" paradigm of Koenderink [10], that, in order for the visual front end of the brain to reconstruct the visual space in the brain, there should be a kind of "geometrical computation" in the neural network of the brain akin to the manipulation of decorated tangle diagrams described in this paper.

I hope to convince the reader that graphic lambda calculus gives a rigorous answer to this question, being a formalism that contains, in a sense, lambda calculus, emergent algebras, and tangle diagrams.

2. Graphs and Moves

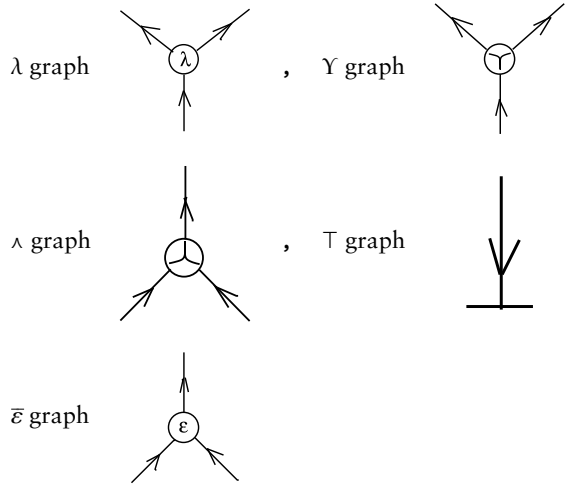
An oriented graph is a pair (V, E), with V the set of nodes and $E \subset V \times V$ the set of edges. Let us denote by $\alpha : V \to 2^E$ the map that associates to any node $N \in V$ the set of adjacent edges $\alpha(N)$. In this paper we work with locally planar graphs with decorated nodes; that is, we shall attach supplementary information to a graph (V, E).

- A function $f : V \to A$ that associates to any node $N \in V$ an element of the "graphical alphabet" A (see Definition 1).
- A cyclic order of $\alpha(N)$ for any $N \in V$, which is equivalent to giving a local embedding of the node N and edges adjacent to it into the plane.

We shall construct a set of locally planar graphs with decorated nodes, starting from a graphical alphabet of elementary graphs. We shall define local transformations, or moves, on the set of graphs. Global moves or conditions will then be introduced.

Definition 1. The graphical alphabet contains the elementary graphs, or gates, denoted by λ, Υ, $\curlywedge$, $\top$, and for any element ε of the commutative group Γ, a graph denoted by $\overline{\varepsilon}$. Here are the elements of the graphical alphabet:

https://doi.org/10.25088/ComplexSystems.22.4.311



With the exception of the $\top$ graph, all other elementary graphs have three edges. The $\top$ graph has only one edge.

There are two types of "fork" graphs: λ and Υ, and two types of "join" graphs: $\wedge$ and $\overline{\varepsilon}$. I now briefly explain what they are supposed to represent and why they are needed in this graphic formalism.

The λ gate corresponds to the lambda abstraction operation from untyped lambda calculus. This gate has one input (the entry arrow) and two outputs (the exit arrows); therefore, at first view, it cannot be a graphical representation of an operation. In untyped lambda calculus the λ abstraction operation has two inputs, namely a variable name x and a term A, and one output, the term $\lambda x.A$. An algorithm is presented in Section 3 to transform a lambda calculus term into a graph made by elementary gates, such that to any lambda abstraction that appears in the term there is a corresponding λ gate.

The Υ gate corresponds to a fan-out gate. It is needed because the graphic lambda calculus described in this paper does not have variable names. Υ gates appear in the process of eliminating variable names from lambda terms, as described in Section 3.

Another justification for the existence of two fork graphs is that they are subjected to different moves: the λ gate appears in the graphic beta move, together with the $\wedge$ gate, while the Υ gate appears in the fan-out moves. Thus, even if the λ and Υ gates have the same topology, they are subjected to different moves, which in fact characterizes their lambda abstraction and fan-out qualities. The alternative, consisting of only one generic fork gate, leads to identifying, in a sense, lambda abstraction with fan-out, which would be confusing.

The $\wedge$ gate corresponds to the application operation from lambda calculus. The algorithm from Section 3 associates a $\wedge$ gate to any application operation used in a lambda calculus term.

https://doi.org/10.25088/ComplexSystems.22.4.311

The $\overline{\varepsilon}$ gate corresponds to an idempotent right quasigroup operation, which appears in emergent algebras as an abstraction of the geometrical operation of taking a dilation (of coefficient ε), based at a point and applied to another point.

The existence of two join gates, with the same topology, is justified by the fact that they appear in different moves.

2.1 The Set of Graphs

We now construct the set of graphs *graph* over the graphical alphabet.

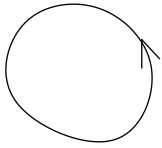
Definition 2. The set of graphs *graph* is obtained by grafting edges from a finite number of copies of the elements of the graphical alphabet. During the grafting procedure, we start from a set of gates and add a finite number of gates one at a time, such that, at any step, any edge of any elementary graph is grafted on any other free edge (i.e., not already grafted to another edge) of the graph, with the condition that they have the same orientation.

For any node of the graph, the local embedding into the plane is given by the element of the graphical alphabet that decorates it.

The set of free edges of a graph $G \in graph$ is named the set of leaves $L(G)$. Technically, imagine that we complete the graph $G \in graph$ by adding to the free extremity of any free edge a decorated node, called a "leaf," with decoration "in" or "out," depending on the orientation of the respective free edge. The set of leaves $L(G)$ thus decomposes into a disjoint union $L(G) = \text{in}(G) \bigcup \text{out}(G)$ of in or out leaves.

Moreover, we admit into *graph* arrows without nodes,

called wires or lines, and loops (without either nodes from the elementary graphs or leaves):



Graphs in *graph* can be disconnected. Any graph that is a finite reunion of lines, loops, and assemblies of the elementary graphs is in *graph*.

2.2 Local Moves

These are transformations of graphs in *graph* that are local, in the sense that any of the moves apply to a limited part of a graph, keeping the rest of the graph unchanged.

https://doi.org/10.25088/ComplexSystems.22.4.311

We may define a local move as a rule transforming a graph into another of the following form.

First, a subgraph of graph G in *graph* is any collection of nodes and/or edges of G. It is not supposed that the mentioned subgraph must be in *graph*. Also, a collection of some edges of G, without any node, counts as a subgraph of G. Thus, a subgraph of G might be imagined as a subset of the reunion of nodes and edges of G.

For any natural number N and any graph G in *graph*, let $\mathcal{P}(G, N)$ be the collection of subgraphs P of the graph G with the sum of the number of their edges and nodes less than or equal to N.

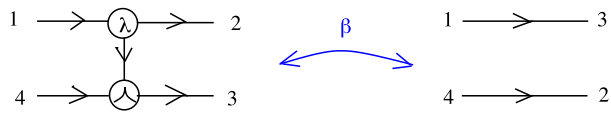
Definition 3. A local move has the following form: there is a number N and a condition C that is formulated in terms of graphs with the sum of the number of their edges and nodes less than or equal to N, such that for any graph G in *graph* and for any $P \in \mathcal{P}(G, N)$, if C is true for P then transform P into P', where P' is also a graph with the sum of the number of its edges and nodes less than or equal to N.

We may graphically group the elements of the subgraph, subjected to the application of the local rule, into a region encircled with a dashed, closed, simple curve. The edges that cross the curve (thus connecting the subgraph P with the rest of the graph) will be numbered clockwise. The transformation will affect only the part of the graph that is inside the dashed curve (inside meaning the bounded connected part of the plane that is bounded by the dashed curve) and, after the transformation is performed, the edges of the transformed graph will connect to the graph outside the dashed curve by respecting the numbering of the edges that cross the dashed line.

However, the grouping of the elements of the subgraph has no intrinsic meaning in graphic lambda calculus. It is just a visual aid and is not a part of the formalism. Sometimes colors are used in the figures as a visual aid. The colors, as well, do not have any intrinsic meaning in the graphic lambda calculus.

2.2.1 Graphic β Move

This is the most important move, inspired by the β-reduction from lambda calculus; see Theorem 1, part (d):



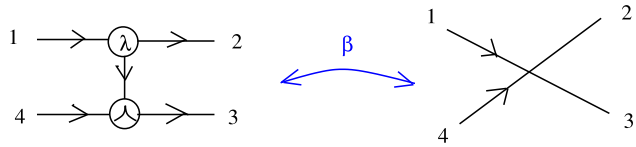
The labels "1, 2, 3, 4" are used only as guides for correctly gluing the new pattern, after removing the old one. As with the encircling

https://doi.org/10.25088/ComplexSystems.22.4.311

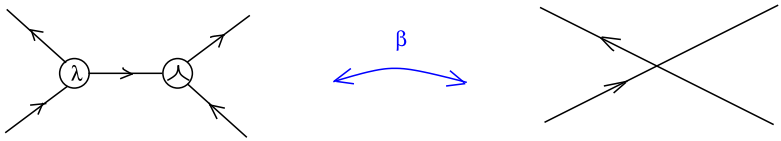
dashed curve, they have no intrinsic meaning in graphic lambda calculus.

This "sewing braids" move will also be used in contexts outside of lambda calculus! It is the most powerful move in this graphic calculus. A primitive form of this move appears as the rewiring move (W1) [9, pp. 20, 21].

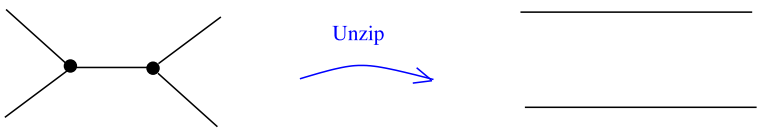
Here is an alternative notation for the sewing braids move:



A move that looks very much like the graphic beta move is the unzip operation from the formalism of knotted trivalent graphs; see, for example, [11, Section 3]. In order to see this, we draw the graphic beta move again, this time without labeling the arrows:



The unzip operation acts only from left to right in the following figure. Remarkably, it acts on trivalent graphs (but not oriented):

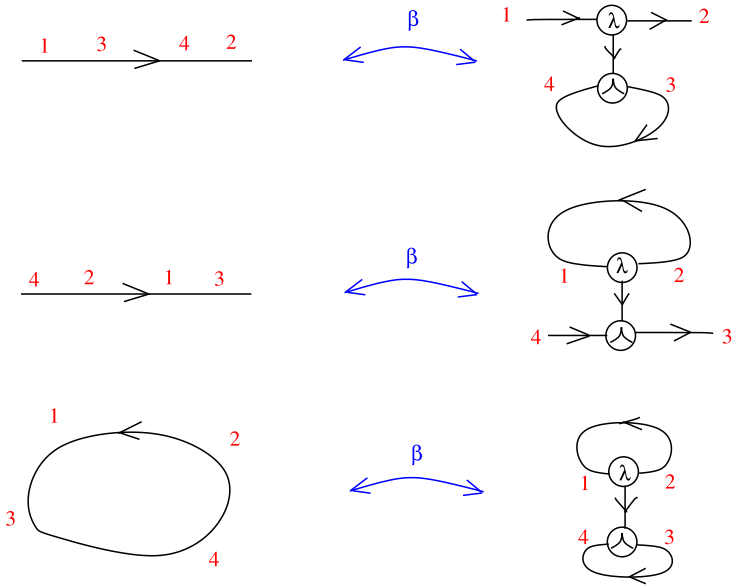


Let us go back to the graphic beta move and remark that it does not depend on the particular embedding in the plane. For example, the intersection of the 1,3 arrow with the 4,2 arrow is an artifact of the embedding; there is no node there. Intersections of arrows have no meaning, since we are working with graphs that are locally planar, not globally planar.

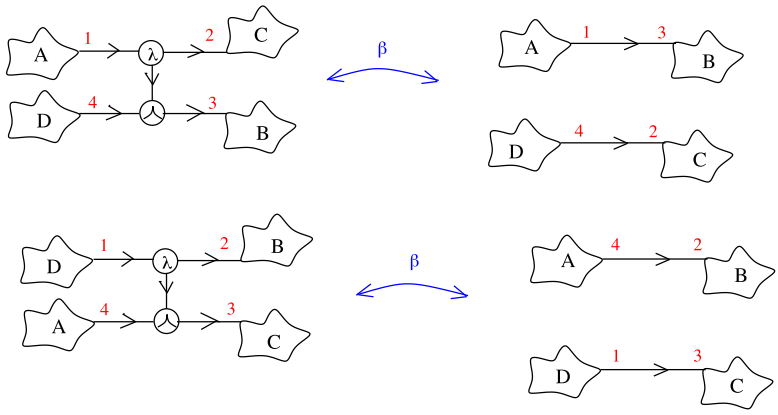
The graphic beta move goes in both directions. In order to apply the move, pick a pair of arrows and label them with 1,2,3,4, such that, according to the orientation of the arrows, 1 points to 3 and 4 points to 2. There are no nodes or labels between 1 and 3 or between 4 and 2. Then, a graphic beta move will replace the portions of the two arrows that are between 1 and 3 and between 4 and 2 with the pattern from the left-hand side of the figure, which describes the graphic beta move.

https://doi.org/10.25088/ComplexSystems.22.4.311

As an illustration of this, the graphic beta move may be applied even to a single arrow or to a loop. The next figure shows three applications of the graphic beta move that illustrate the need to consider loops and wires as members of *graph*:

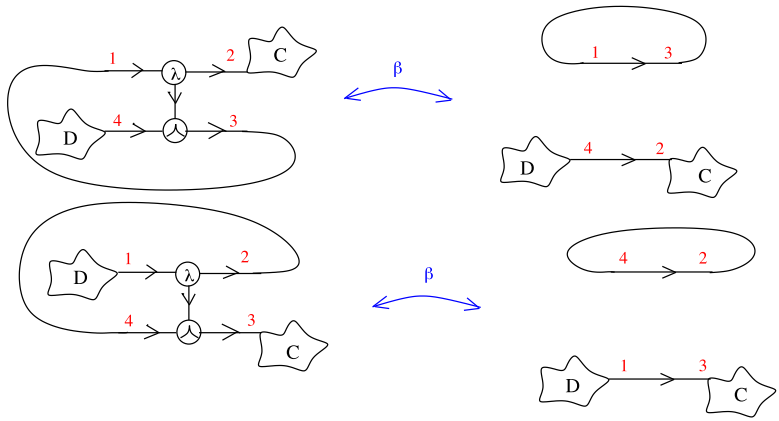


We can apply a graphic beta move in different ways to the same graph and in the same place, simply by using different labels 1, ... 4 (here *A*, *B*, *C*, *D* are graphs in *graph*):

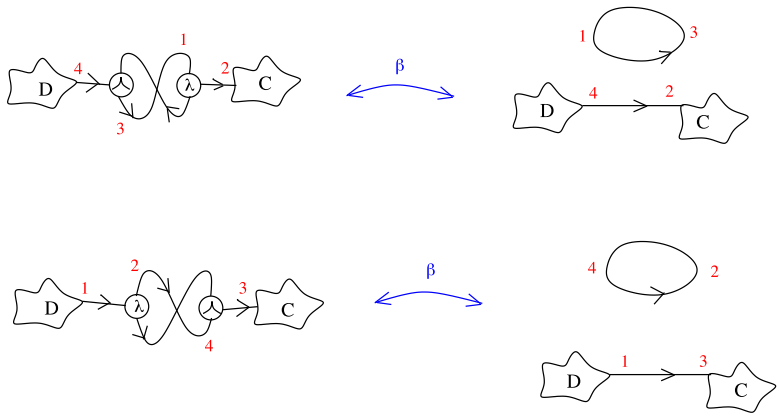


A particular case of the previous figure is yet another justification for having loops as elements in *graph*:

https://doi.org/10.25088/ComplexSystems.22.4.311



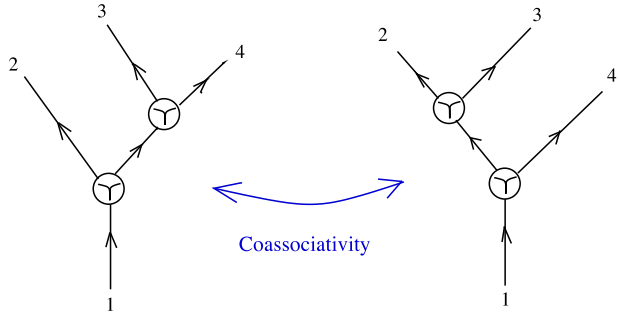
The two previous applications of the graphic beta move may be represented alternatively like this:



2.2.2 Coassociativity Move

This is the coassociativity move involving the Υ graphs. Consider the Υ graph as corresponding to a fan-out gate:

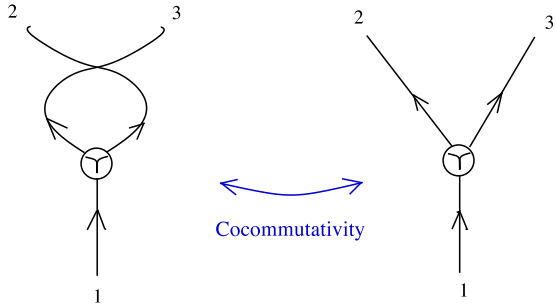
https://doi.org/10.25088/ComplexSystems.22.4.311



By using coassociativity moves, we can move between any two binary trees formed only with Υ gates, with the same number of output leaves.

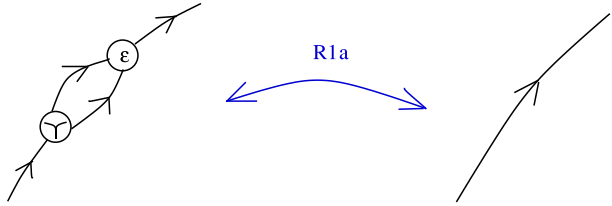
2.2.3 Cocommutativity Move

This is the cocommutativity move involving the Υ gate. It will be not used until Section 6 concerning knot diagrams:



2.2.3a R1a Move

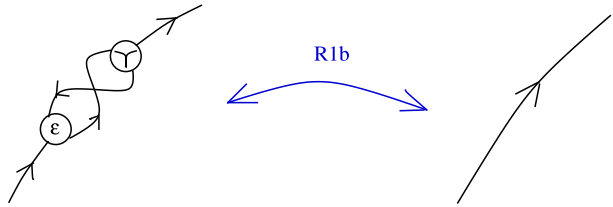
This move is imported from emergent algebras. Explanations are given in Section 5. It involves an Υ graph and a $\overline{\varepsilon}$ graph, with $\varepsilon \in \Gamma$:



https://doi.org/10.25088/ComplexSystems.22.4.311

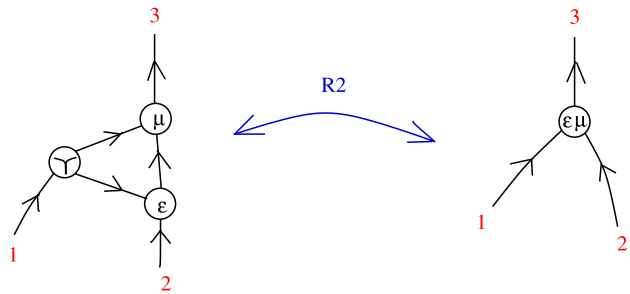
2.2.3b R1b Move

Here is the R1b move (also related to emergent algebras):



2.2.4 R2 Move

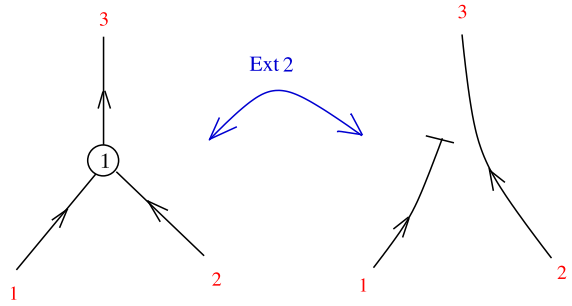
This corresponds to the Reidemeister II move for emergent algebras. It involves an Υ graph and two others: a $\overline{\varepsilon}$ and a $\overline{\mu}$ graph, with $\varepsilon, \mu \in \Gamma$:



The R2 move appears in [9, p. 21], with the supplementary name "triangle move."

2.2.5 Ext2 Move

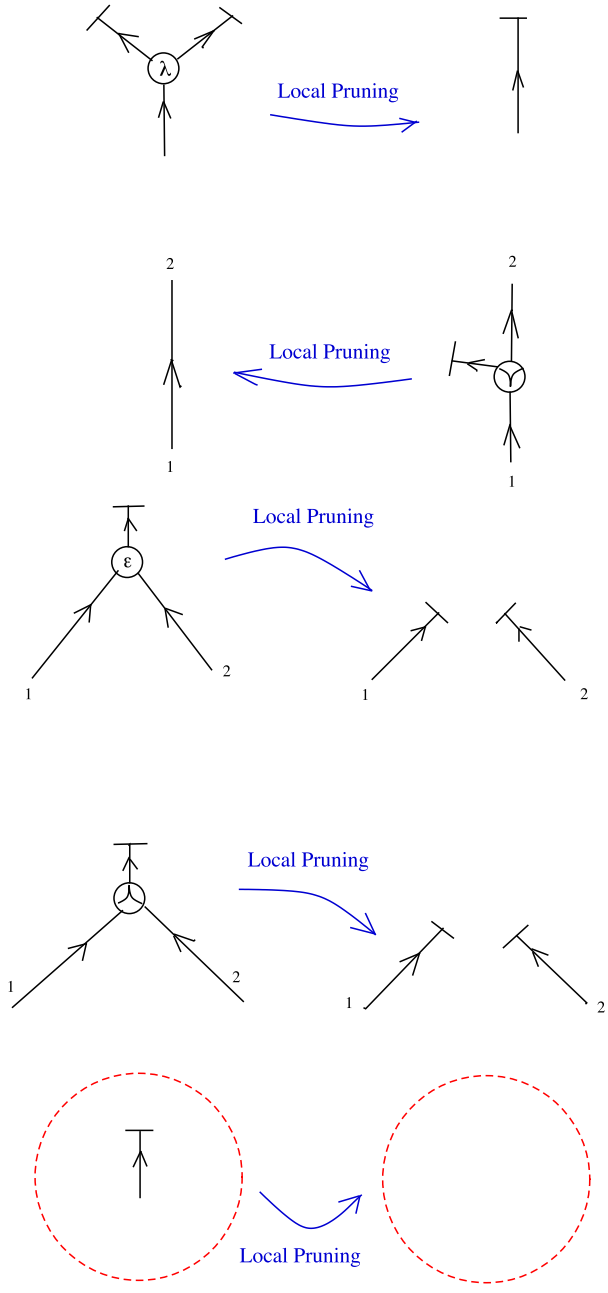
This corresponds to the rule (ext2) from λ-scale calculus. It expresses the fact that in emergent algebras the operation indexed with the neutral element 1 of the group Γ has the property $x \circ_1 y = y$:



https://doi.org/10.25088/ComplexSystems.22.4.311

2.2.6 Local Pruning

Local pruning moves are local moves that eliminate dead edges. Note that, unlike the previous moves, these are one way (you can eliminate dead edges, but not add them to graphs):



https://doi.org/10.25088/ComplexSystems.22.4.311

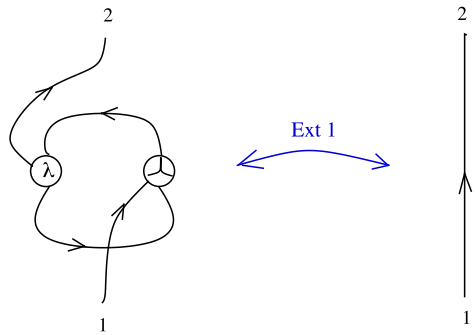
2.3 Global Moves or Conditions

Global moves are not local either because the condition C applies to parts of the graph that may have an arbitrarily large sum of edges plus nodes or because after the move the graph P' that replaces the graph P has an arbitrarily large sum of edges plus nodes.

2.3.1 Ext1 Move

This corresponds to the rule (ext1) from λ-scale calculus, or to η-reduction in lambda calculus (see Theorem 1, part (e) for details). It involves a λ graph (similar to the λ abstraction operation in lambda calculus) and a $\curlywedge$ graph (similar to the application operation in lambda calculus).

The rule is: if there is no oriented path from 2 to 1, then the ext1 move can be performed:

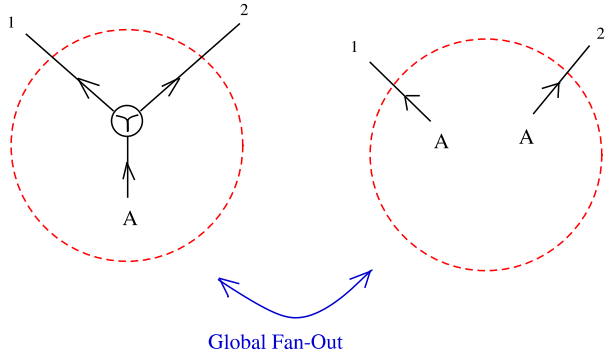


2.3.2 Global Fan-Out Move

This is a global move that consists of replacing (under certain circumstances) a graph by two copies of that graph.

The rule is: if a graph in $G \in graph$ has an Υ bottleneck, that is, if we can find a subgraph $A \in graph$ connected to the rest of the graph G only through an Υ gate, then we can perform the move depicted in the next figure, from the left to the right:

https://doi.org/10.25088/ComplexSystems.22.4.311



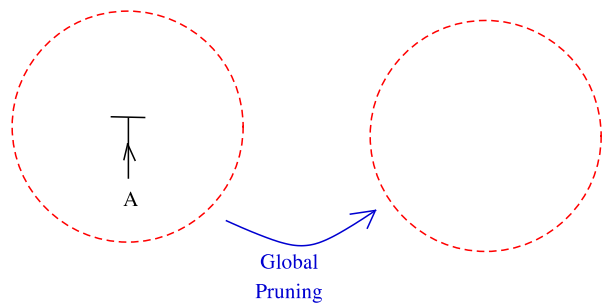
Conversely, if in the graph G we can find two identical subgraphs (denoted by A) that are in *graph* and have no edge connecting one with another and that are connected to the rest of G only through one edge, as in the right-hand side of the figure, then we can perform the move from the right to the left.

Note that global fan-out trivially implies cocommutativity. As a local rule alternative to the global fan-out, we might consider the following. Fix a number N and consider only graphs A that have at most N (nodes + arrows). The N local fan-out move is the same as the global fan-out move, except it only applies to such graphs A. This local fan-out move does not imply cocommutativity.

2.3.3 Global Pruning

This a global move to eliminate dead edges.

The rule is: if a graph in $G \in \mathit{graph}$ has a $\top$ ending, that is, if we can find a subgraph $A \in \mathit{graph}$ connected only to a $\top$ gate, with no edges connecting to the rest of G, then we can erase this graph and the respective $\top$ gate:



The global pruning may be needed because of the λ gates that cannot be removed only by local pruning.

https://doi.org/10.25088/ComplexSystems.22.4.311

2.3.4 Elimination of Loops

It is possible that after using a local or global move, we obtain a graph with an arrow that closes itself, without being connected to any node. For example, if a loop appears after the application of a graphic beta move, then it can be erased by the elimination of loops move.

2.4 λ-graphs

The edges of an elementary graph λ can be numbered unambiguously, clockwise, by 1, 2, 3, such that 1 is the number of the entrant edge.

Definition 4. A graph $G \in graph$ is a λ-graph, denoted as $G \in \lambda\, graph$, if:

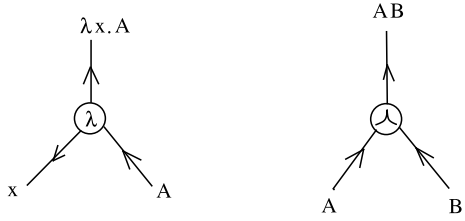
- it does not have any $\overline{\varepsilon}$ gates,
- for any node λ any oriented path in G starting at edge 2 of this node can be completed to a path that either terminates in a graph $\top$, or else terminates at edge 1 of this node.

The condition $G \in \lambda\, graph$ is global, in the sense that in order to decide if $G \in \lambda\, graph$, we have to examine parts of the graph that may have an arbitrarily large sum of edges plus nodes.

3. Conversion of Lambda Terms

This section describes how to associate a lambda term to a graph in *graph* that is then used to show that β-reduction in lambda calculus transforms into the β rule for *graph*.

Indeed, to any term $A \in T(X)$ (where $T(X)$ is the set of lambda terms over the variable set X) we associate its syntactic tree. The syntactic tree of any lambda term is constructed by using two gates, one corresponding to the λ abstraction and the other corresponding to the application. We draw syntactic trees with the leaves (elements of X) at the bottom and the root at the top. We shall use the following notation for the two gates: at the left is the gate for the λ abstraction and at the right is the gate for the application:



Notice that these two gates are from the graphical alphabet of *graph*, but the syntactic tree is decorated: at the bottom we have

https://doi.org/10.25088/ComplexSystems.22.4.311

leaves from X. Also, notice the peculiar orientation of the edge from the left (in tree notation convention) of the λ gate. For the moment, this orientation is in contradiction with the implicit orientation (from down-up) of edges of the syntactic tree, but soon this matter will become clear.

We shall remove all leaf decorations, with the price of introducing new gates, namely $\curlyvee$ and $\top$. This will be done in a sequence of steps, detailed later. Take the syntactic tree of $A \in T(X)$, drawn with the mentioned conventions (concerning gates and the positioning of leaves and root, respectively).

We take as examples the following five lambda terms:

$$I = \lambda x.x$$
$$K = \lambda x.(\lambda y.x)$$
$$S = \lambda x.(\lambda y.(\lambda z.((xz)\,(yz))))$$
$$\Omega = (\lambda x.(xx))\,(\lambda x.(xx))$$
$$T = (\lambda x.(xy))\,(\lambda x.(xy))$$

3.1 Step 1: List Bound Variables

Any leaf of the tree is connected to the root by a unique path. Start from the leftmost leaf, perform the algorithm explained next, then go to the right and repeat until all leaves are exhausted. We also initialize a list $B = \emptyset$ of bound variables.

Take a leaf, say decorated with $x \in X$. To this leaf is associated a word (a list) that is formed by the symbols of gates on the path connecting (from the bottom up) the leaf with the root, together with information about which way, left (L) or right (R), the path passes through the gates. Such a word is formed by the letters λ^{L}, λ^{R}, ${}_{\curlywedge}{}^{\mathrm{L}}$, ${}_{\curlywedge}{}^{\mathrm{R}}$.

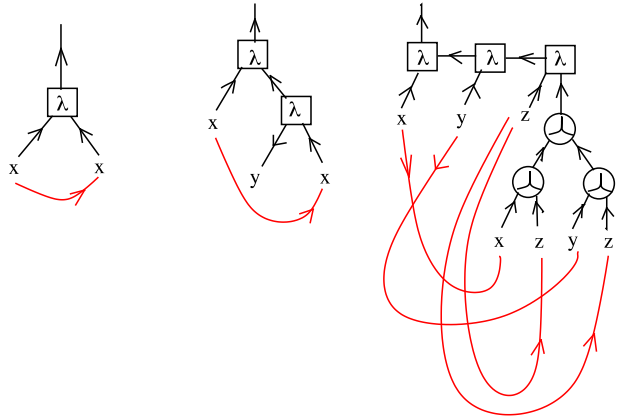
If the first letter is λ^{L}, then add to the list B the pair $(x, w(x))$ formed by the variable name x and the associated word (describing the path to follow from the respective leaf to the root). Then pass to a new leaf.

Otherwise, continue along the path to the root. If we arrive at a λ gate, which can happen only coming from the right leg of the λ gate, we can find only the letter λ^{R}. In such a case look at the variable y that decorates the left leg of the same λ gate. If $x = y$, then add a new edge to the syntactic tree from y to x and proceed farther along the path; otherwise proceed farther. If the root is attained, then pass to the next leaf.

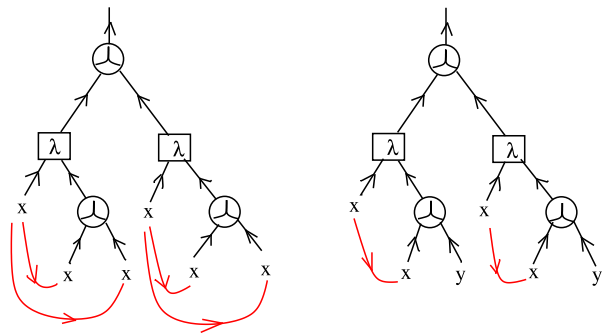
Here are some examples of graphs associated to the mentioned lambda terms, together with the list of bound variables.

https://doi.org/10.25088/ComplexSystems.22.4.311

- $I = \lambda\, x.x$ has $B = \{(x, \lambda^{\mathrm{L}})\}$, $K = \lambda x.(\lambda y.x)$ has $B = \{(x, \lambda^{\mathrm{L}}), (y, \lambda^{\mathrm{L}}\lambda^{\mathrm{R}})\}$, and $S = \lambda\, x.(\lambda\, y.(\lambda\, z.((x\, z)\,(y\, z))))$ has $B = \{(x, \lambda^{\mathrm{L}}), (y, \lambda^{\mathrm{L}}\lambda^{\mathrm{R}}), (z, \lambda^{\mathrm{L}}\lambda^{\mathrm{R}}\lambda^{\mathrm{R}})\}$:



- $\Omega = (\lambda\, x.(x\, x))\,(\lambda\, x.(x\, x))$ has $B = \{(x, \lambda^{\mathrm{L}}\curlywedge^{\mathrm{L}}), (x, \lambda^{\mathrm{L}}\curlywedge^{\mathrm{R}})\}$ and $T = (\lambda\, x.(x\, y))\,(\lambda\, x.(x\, y))$ has $B = \{(x, \lambda^{\mathrm{L}}\curlywedge^{\mathrm{L}}), (x, \lambda^{\mathrm{L}}\curlywedge^{\mathrm{R}})\}$:



3.2 Step 2: Eliminate Bound Variables

We now have a list B of bound variables. If the list is empty, then go to the next step. Otherwise, do the following, starting from the first element of the list, until the list is finished.

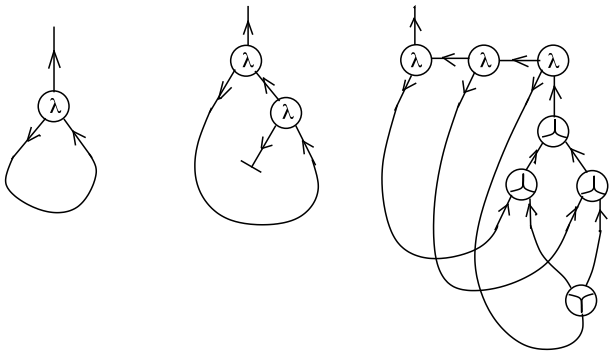
An element, say $(x, w(x))$, of the list is either connected to other leaves by one or more edges added at step 1 or not. If it is not connected, then erase the variable name with the associated path $w(x)$ and replace it with a $\top$ gate. If it is connected, then erase it and replace it with a tree formed by $\curlyvee$ gates, starting at the place where the elements of the list were before the erasure and stopping at the leaves

https://doi.org/10.25088/ComplexSystems.22.4.311

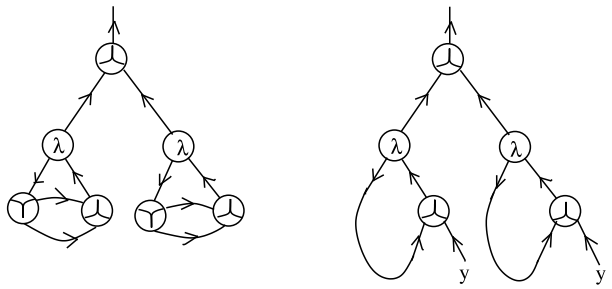
that were connected to x. Erase all decorations that were joined to x and also erase all edges added at step 1 to leaf x from the list.

Here are examples showing the graphs associated to the mentioned lambda terms after step 2.

- The graphs of $I = \lambda\, x.x$, $K = \lambda x.(\lambda y.x)$, and $S = \lambda\, x.(\lambda\, y.(\lambda\, z.((x\, z)\, (y\, z))))$:



- The graphs of $\Omega = (\lambda\, x.(x\, x))\, (\lambda\, x.(x\, x))$ and $T = (\lambda\, x.(x\, y))\, (\lambda\, x.(x\, y))$:



Notice that at this step the necessity of having the peculiar orientation of the left leg of the λ gate becomes clear.

Note also that there may be more than one possible tree of gates Υ, for example, at each elimination of a bound variable (in cases when a bound variable has at least three occurrences). One may use any tree of Υ that is fit. The problem of multiple possibilities is the reason for introducing the coassociativity move.

3.3 Step 3: Remove Leaf Decorations

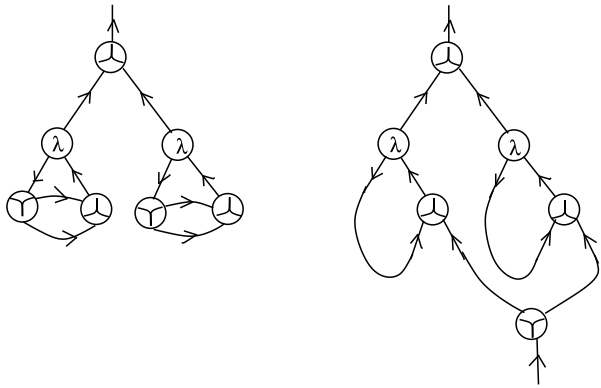
There may still be leaves decorated with free variables. Starting from the left to the right, group them together in case some of them occur in multiple places, then replace the multiple occurrences of a free variable by a tree of Υ gates with a free root ending exactly at the occurrences of the respective variables. Again, there are multiple ways of

https://doi.org/10.25088/ComplexSystems.22.4.311

doing this, but we may pass from one to another by a sequence of coassociativity moves.

Here are examples after step 3. All the graphs associated to the mentioned lambda terms, except the last one, are left unchanged. The graph of the last term changes.

- The graph of $\Omega = (\lambda\, x.(x\, x))\,(\lambda\, x.(x\, x))$, left unchanged by step 3, and the graph of $T = (\lambda\, x.(x\, y))\,(\lambda\, x.(x\, y))$:



Theorem 1. Let $A \mapsto [A]$ be a transformation of a lambda term A into a graph $[A]$ as described previously (multiple transformations are possible because of the choice of Υ trees). Then:

(a) For any term A the graph $[A]$ is in λ *graph*.

(b) If $[A]'$ and $[A]''$ are transformations of the term A, then we may pass from $[A]'$ to $[A]''$ by using a finite number (exponential in the number of leaves of the syntactic tree of A) of coassociativity moves.

(c) If B is obtained from A by α-conversion, then we may pass from $[A]$ to $[B]$ by a finite sequence of coassociativity moves.

(d) Let $A, B \in T(X)$ be two terms and $x \in X$ be a variable. Consider the terms $\lambda\, x.A$ and $A[x := B]$, where $A[x := B]$ is the term obtained by substituting in A the free occurrences of x by B. We know that β reduction in lambda calculus consists of passing from $(\lambda\, x.A)\, B$ to $A[x := B]$. Then, by one β move in *graph* applied to $[(\lambda\, x.A)\, B]$ we pass to a graph that can be further transformed into one of $A[x := B]$, via global fan-out, coassociativity, and pruning moves.

(e) With the notations from (d), consider the terms A and $\lambda\, x.A\, x$ with $x \notin FV(A)$; then the η reduction, consisting of passing from $\lambda\, x.A\, x$ to A, corresponds to the ext1 move applied to the graphs $[\lambda\, x.A\, x]$ and $[A]$.

Proof. (a) We have to prove that, for any node λ, any oriented path in $[A]$ starting at the left exiting edge of the λ node can be completed as

https://doi.org/10.25088/ComplexSystems.22.4.311

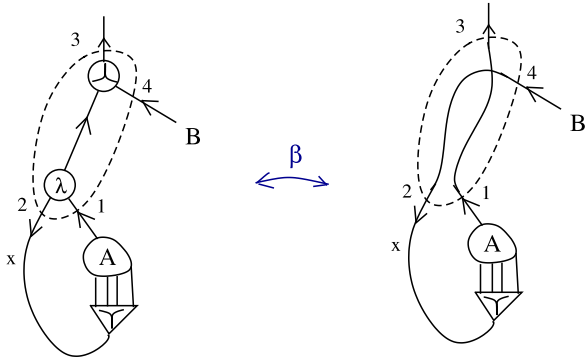
a path that either terminates in a $\top$ graph, or else terminates at the entry peg of the λ node, but this is clear. Indeed, either the bound variable (of this λ node in the syntactic tree of A) is fresh and gets replaced by a $\top$ gate, or else the bound variable is replaced by a tree of Υ gates. No matter which path we choose, we may complete it as a cycle passing by the said λ node.

(b) Clear also, because the coassociativity move is designed for passing from a tree of Υ gates to another tree with the same number of leaves.

(c) Indeed, the names of bound variables of A do not affect the construction of $[A]$, therefore if B is obtained by α-conversion of A, then $[B]$ differs from $[A]$ only by the particular choice of trees of Υ gates. But this is solved by coassociativity moves.

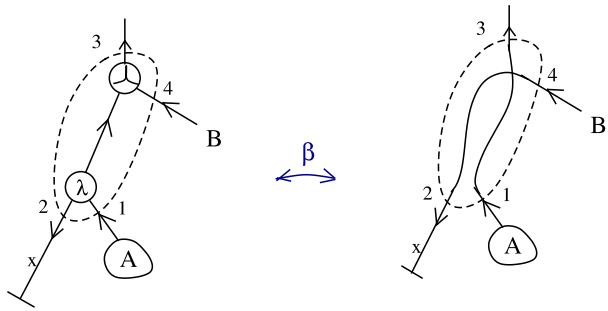
(d) This may be the surprising part of the theorem. There are two cases: x is fresh for A or not. If x is fresh for A, then in the graph $[(\lambda x.A)\,B]$ the named variable x is replaced by a $\top$ gate. If not, then all the occurrences of x in A are connected by an Υ tree with its root at the left peg of the λ gate where x appears as a bound variable.

In the case when x is not fresh for A, the left-hand side of the figure has the graph $[(\lambda x.A)\,B]$ (with a remaining decoration of "x"). We perform a graphic β move and obtain the graph on the right:



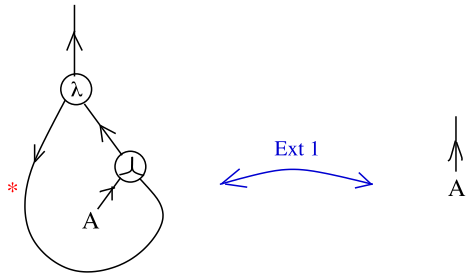
This graph can be transformed into a graph of $A[x := B]$ via global fan-out and coassociativity moves. Here is the case when x is fresh for A:

https://doi.org/10.25088/ComplexSystems.22.4.311



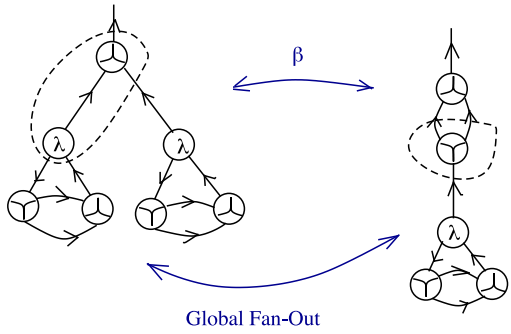
We see that the graph obtained by performing the graphic β move is the union of the graph of A and the graph of B with a $\top$ gate added at the root. After pruning we are left with the graph of A, consistent with the fact that when x is fresh for A then $(\lambda\, x.A)\, B$ transforms by β reduction into A.

(e) In the next figure the left-hand side is the graph $[\lambda\, x.A\, x]$ and the right-hand side is the graph $[A]$:



The red asterisk marks the arrow that appears in the construction $[\lambda\, x.A\, x]$ from the variable x, taking into account the hypothesis $x \notin FV(A)$. We have a pattern where we can apply the ext1 move and obtain $[A]$, as claimed. □

As an example, let us manipulate the graph of $\Omega = (\lambda\, x.(x\, x))\, (\lambda x.(x\, x))$:

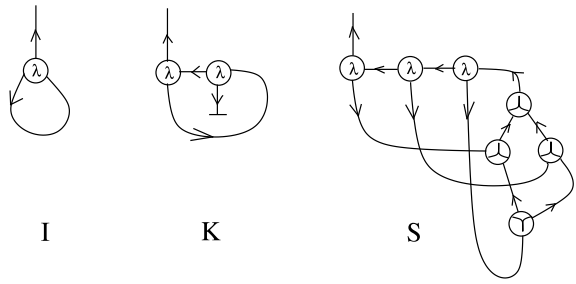


https://doi.org/10.25088/ComplexSystems.22.4.311

We can pass from the left-hand side of the figure to the right-hand side by using a graphic β move. Conversely, we can pass from the right-hand side of the figure to the left-hand side by using a global fan-out move. These manipulations correspond to the well-known fact that Ω remains unchanged after β reduction: let $U = \lambda x.(x\,x)$, then $\Omega = U\,U = (\lambda x.(x\,x))\,U \leftrightarrow U\,U = \Omega$.

3.4 Example: Combinatory Logic

The combinators $I = \lambda x.x$, $K = \lambda x.(\lambda y.x)$, and $S = \lambda x.(\lambda y.(\lambda z.((x\,z)\,(y\,z))))$ have the following correspondents in *graph*, denoted by the same letters:



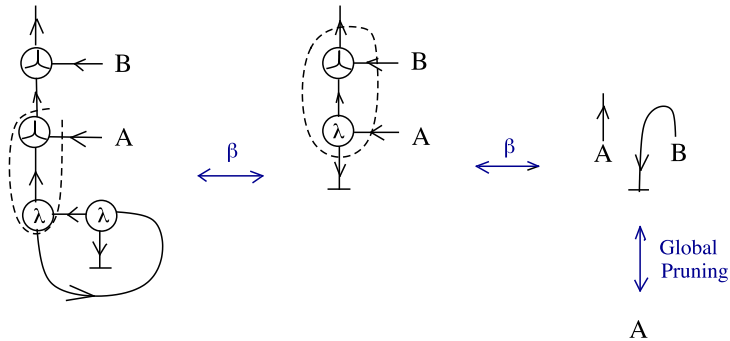
Proposition 1. (a) By one graphic β move, $I \wedge A$ transforms into A, for any $A \in$ *graph* with one output.

(b) By two graphic β moves followed by a global pruning, for any $A, B \in$ *graph* with one output, the graph $(K \wedge A) \wedge B$ transforms into A.

(c) By five graphic β moves followed by one local pruning move, the graph $(S \wedge K) \wedge K$ transforms into I.

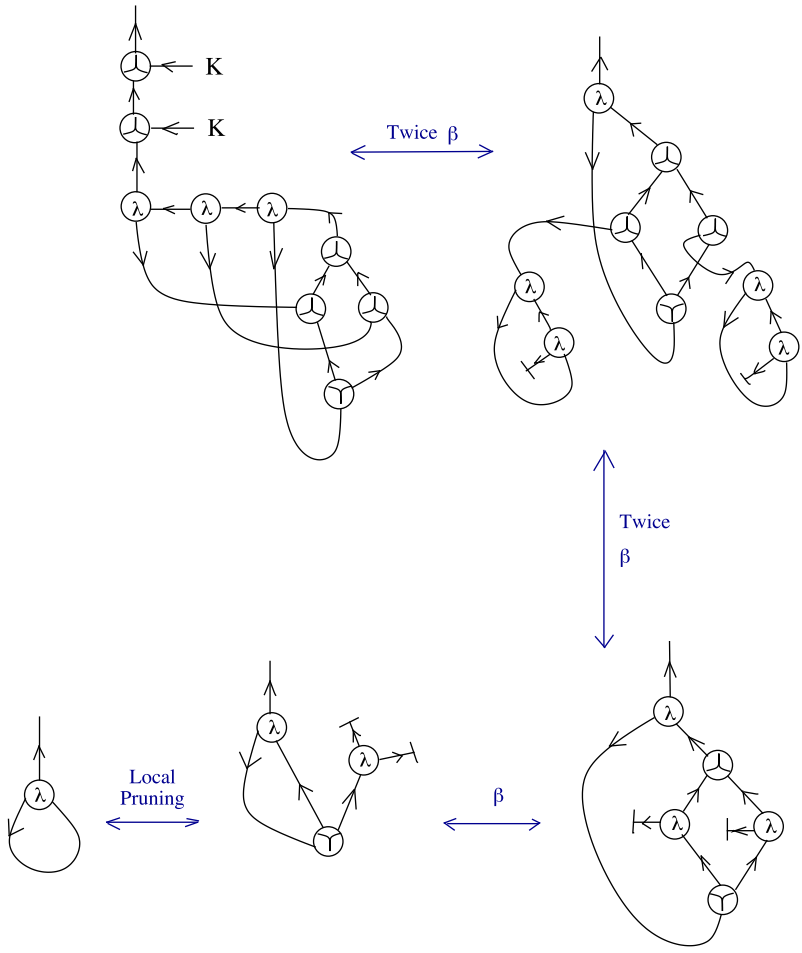
(d) By three graphic β moves followed by a global fan-out move, for any $A, B, C \in$ *graph* with one output, the graph $((S \wedge A) \wedge B) \wedge C$ transforms into the graph $(A \wedge C) \wedge (B \wedge C)$.

Proof. The proof of (b) is given as this figure:



https://doi.org/10.25088/ComplexSystems.22.4.311

The proof of (c) is given as this figure:



(a) and (d) are left to the interested reader. □

4. Using Graphic Lambda Calculus

The graph manipulations presented in this section can be applied for graphs that represent lambda terms. However, they can also be applied for graphs that do not represent lambda terms.

4.1 Fixed Points

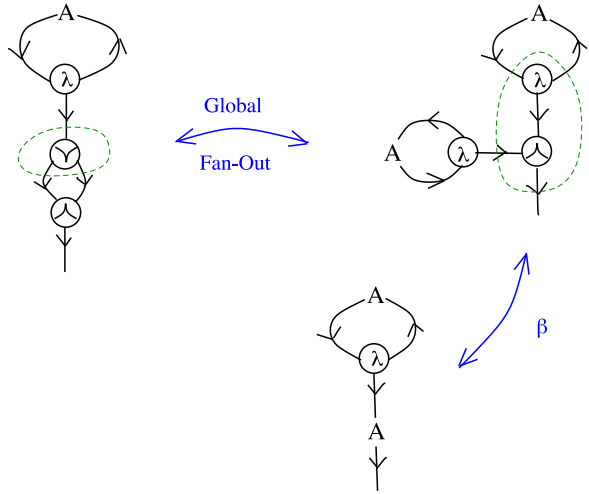
Let us start with a graph $A \in graph$ that has one distinguished input and one distinguished output represented as follows:

https://doi.org/10.25088/ComplexSystems.22.4.311

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.

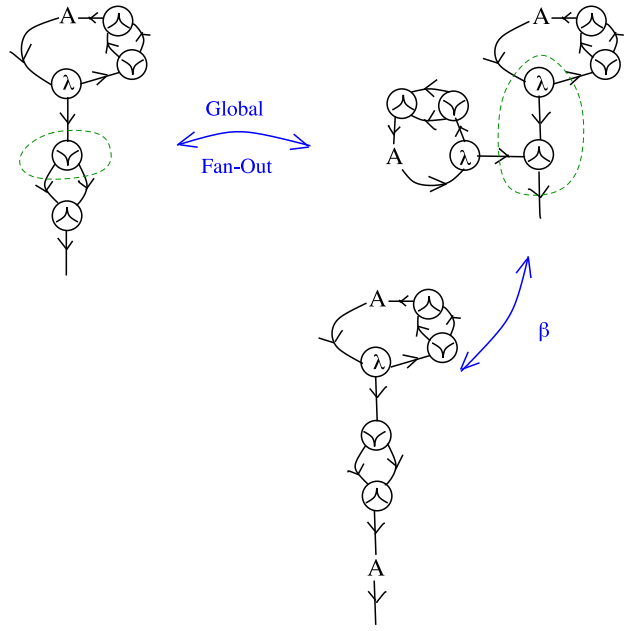
I want to find B such that $A(B) \leftrightarrow B$, where $\leftrightarrow$ means any finite sequence of moves in graphic lambda calculus. Such a graph B is called a fixed point of A.

The solution of this problem is the same as in usual lambda calculus. We start from the following succession of moves:



This is very close to the solution; we only need a small modification:

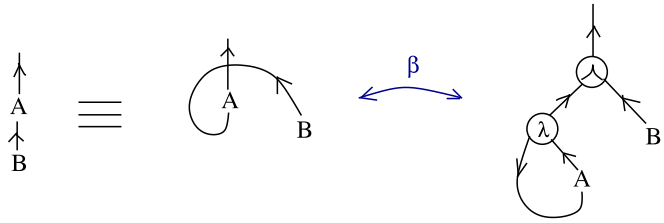
https://doi.org/10.25088/ComplexSystems.22.4.311



4.2 Grafting: Application or Abstraction?

If the A, B from the previous paragraph represent lambda terms, then the natural operation between them is not grafting, but the application. Or, in graphic lambda calculus the application is represented by an elementary graph, therefore $A\,B$ (seen as the term in lambda calculus obtained as the application of A to B) is not represented as a grafting of the output of B to the input of A.

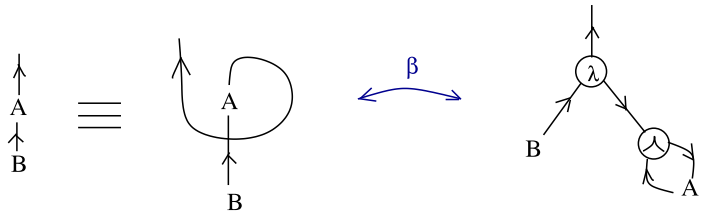
We can easily transform grafting into the application operation:



Suppose that A and B are graphs representing lambda terms. More precisely, suppose that A is representing a term (also denoted by A) and its input represents a free variable x of the term A. Then, the grafting of B to A is the term $A[x := B]$ and the graph from the right is representing $(\lambda\, x.A)\, B$; therefore, both graphs are representing terms from lambda calculus.

https://doi.org/10.25088/ComplexSystems.22.4.311

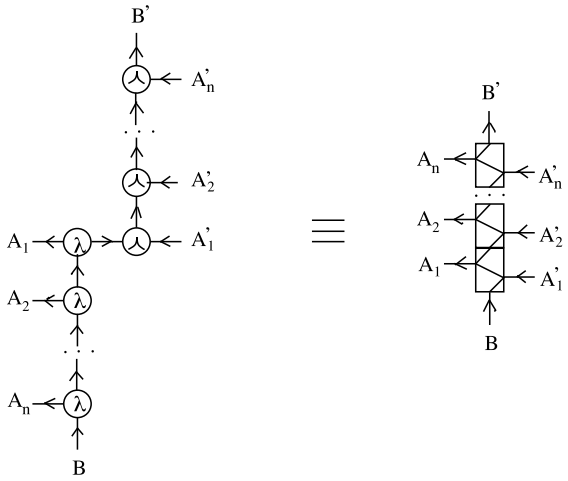
We can transform grafting into something else:



This has no meaning in lambda calculus, but it looks as if the abstraction gate (the λ gate) plays the role of an application operation, except for the orientation of one of the arrows of the graph from the right.

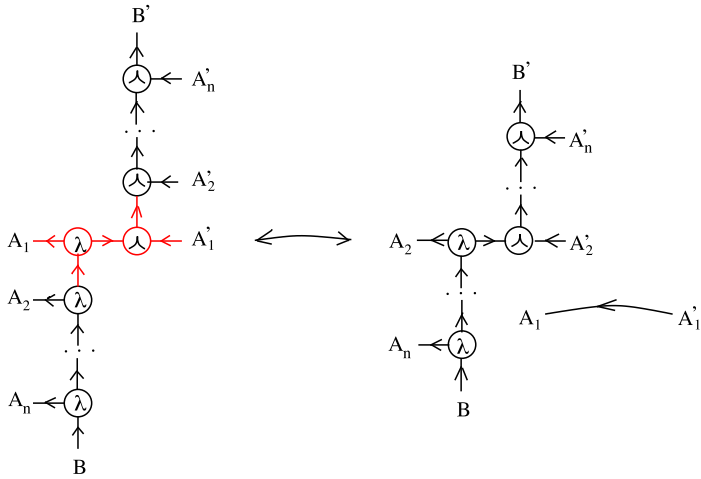
4.3 Zippers and Combinators as Half-Zippers

Let us take $n \geq 1$ to be a natural number and let us consider the following graph in *graph*, called the *n*-zipper:

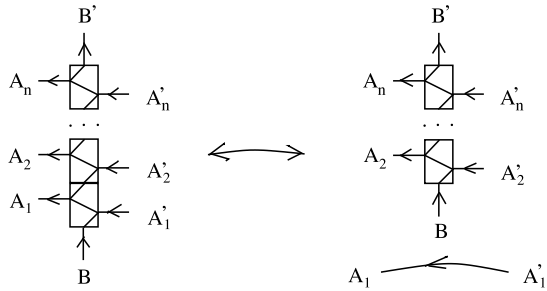


At the left is the *n*-zipper graph; at the right is a notation for it, or a "macro." The zipper graph is interesting because it allows performing (nontrivial) graphic beta moves in a fixed order. In the following figure, red depicts the place where the first graphic beta move is applied:

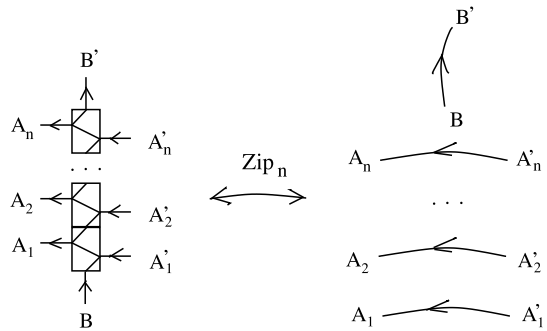
https://doi.org/10.25088/ComplexSystems.22.4.311



This graphic beta move has the following appearance in zipper notation:



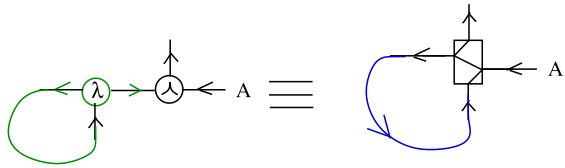
We see that an n-zipper transforms into an $(n\text{-}1)$-zipper plus an arrow. This move may be repeated as long as possible. This procedure defines a zipper move:



We may consider the 1-zipper move the same as the graphic beta move, transforming the 1-zipper into two arrows.

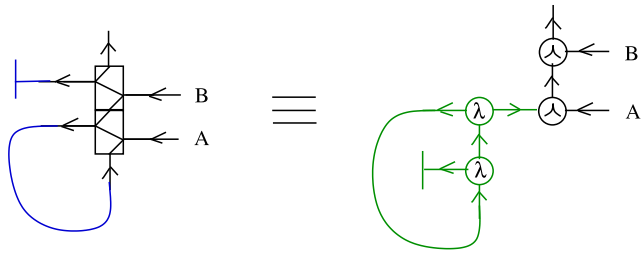
https://doi.org/10.25088/ComplexSystems.22.4.311

The combinator $I = \lambda x.x$ satisfies the relation $I\,A = A$. The next figure shows that I (in green), when applied to A, is just a half of the 1-zipper, with an arrow added (in blue):



By opening the zipper we obtain A, as we should.

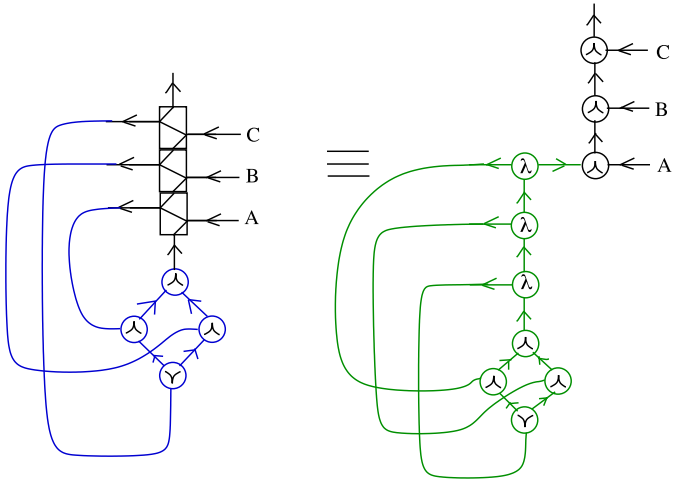
The combinator $K = \lambda x\,y.x$ satisfies $K\,A\,B = (K\,A)\,B = A$. In the next figure the combinator K (in green) appears as half of the 2-zipper, with one arrow and one termination gate added (in blue):



After opening the zipper we obtain a pair made by A and B that gets the termination gate on top of it. A global pruning move sends B to the trash bin.

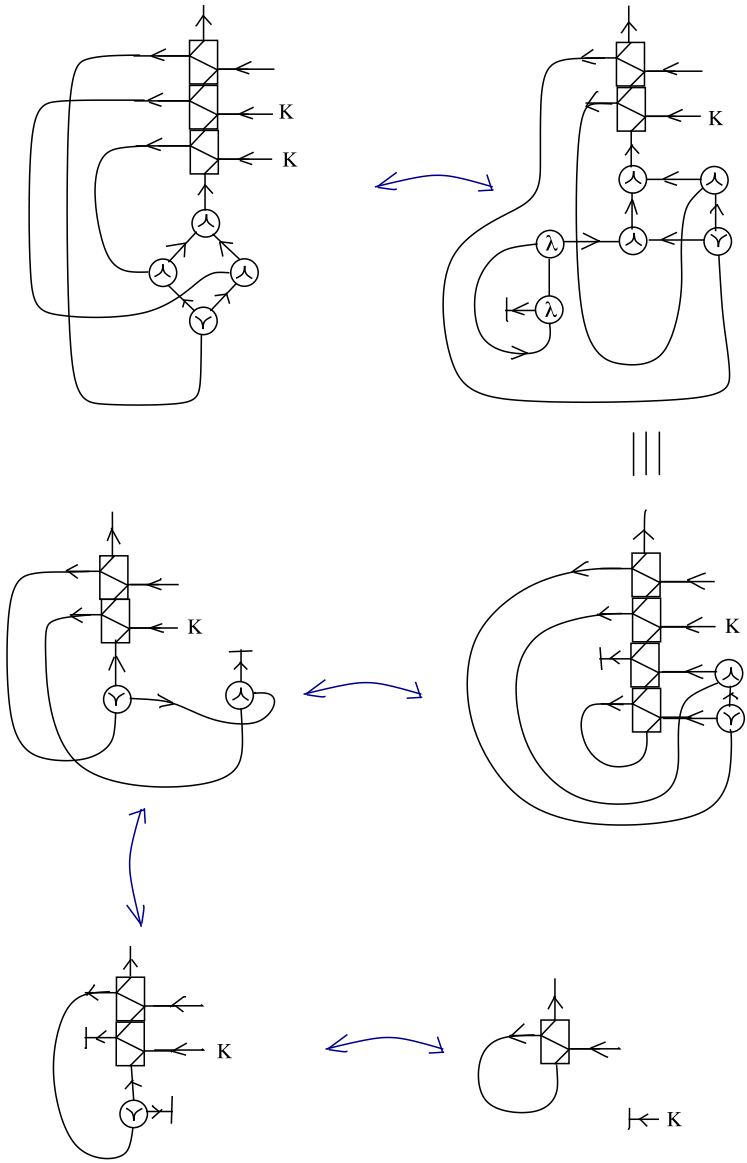
Finally, the combinator $S = \lambda x\,y\,z.((x\,z)\,(y\,z))$ satisfies $S\,A\,B\,C = ((S\,A)\,B)\,C = (A\,C)\,(B\,C)$. The combinator S (in green) appears to be made by half of the 3-zipper, with some arrows and also with a "diamond" added (all in blue). Interestingly, the diamond looks similar to the ones from Definition 8 in Section 5.

https://doi.org/10.25088/ComplexSystems.22.4.311



Expressed with the help of zippers, the relation $SKK = I$ appears like this:

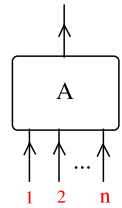
https://doi.org/10.25088/ComplexSystems.22.4.311



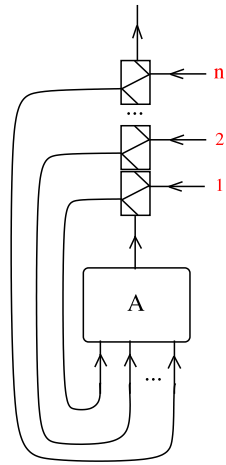
4.4 Lists and Currying

With the help of zippers, we may enhance the procedure of turning grafting into the application operation. We have a graph $A \in \mathit{graph}$ that has one output and several inputs:

https://doi.org/10.25088/ComplexSystems.22.4.311

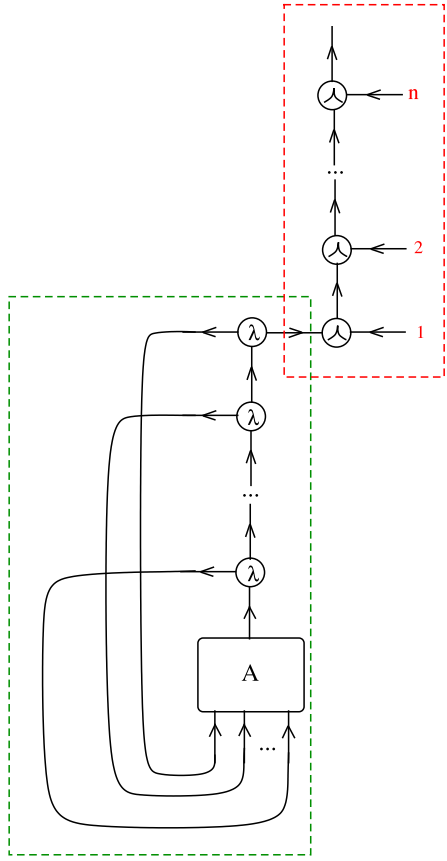


We use an n-zipper in order to clip the inputs with the output:



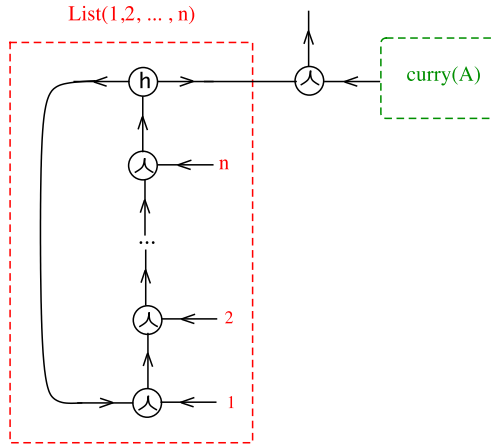
This graph is, in fact, the following one:

https://doi.org/10.25088/ComplexSystems.22.4.311



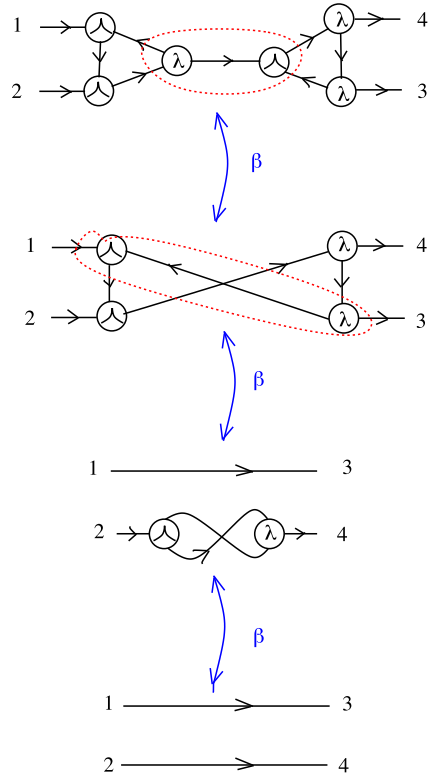
We may interpret the graph inside the green dotted rectangle as the currying of A, called *curry*(A). This graph has only one output and no inputs. The graph inside the red dotted rectangle is almost a list. We shall transform it into a list by again using a zipper and one graphic beta move:

https://doi.org/10.25088/ComplexSystems.22.4.311



4.5 Packing Arrows

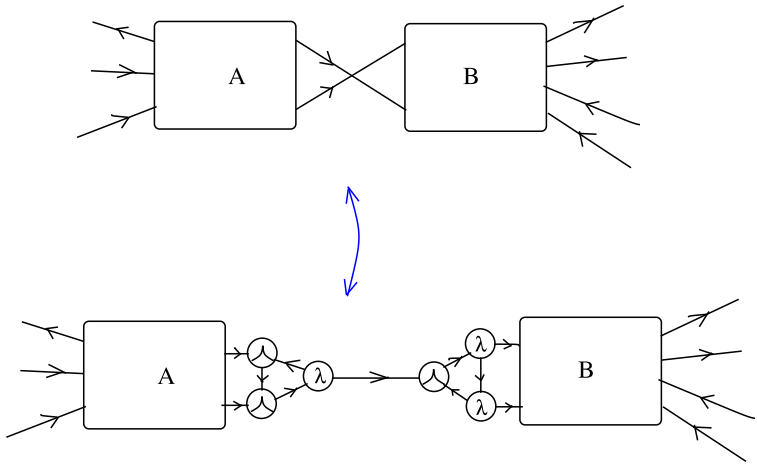
We may pack several arrows into one. The case of two arrows is described first. We start from the following sequence of three graphic beta moves:



https://doi.org/10.25088/ComplexSystems.22.4.311

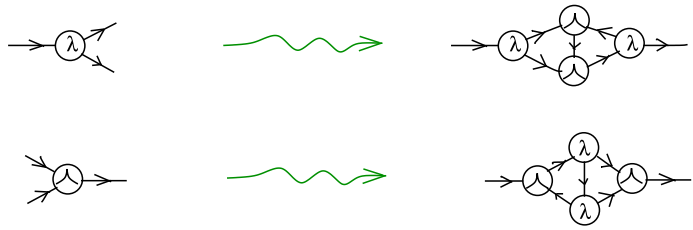
This figure means: we pack the 1,2 entries into a list, pass it through one arrow, then unpack the list into the outputs 3,4. Of course, this packing/unpacking trick may be used for more than a pair of arrows, in obvious ways; therefore, it is not a restriction of generality to write only about two arrows.

We may apply the trick to a pair of graphs A and B that are connected by a pair of arrows, as in the following figure:



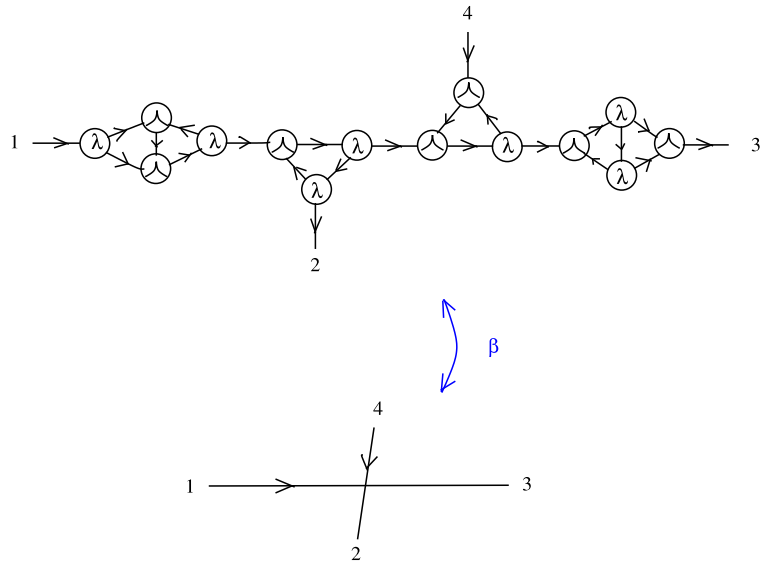
With the added packing/unpacking triples of gates, the graphs A, B are interacting only by the intermediary of one arrow.

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:



If we use the elementary gates transformed into graphs with one input and one output, the graphic beta move becomes this almost-algebraic, one-dimensional rule:

https://doi.org/10.25088/ComplexSystems.22.4.311



With such procedures, we may transform any graph in *graph* into a one-dimensional string of graphs, consisting of transformed elementary graphs and packers/unpackers of arrows, which could be used, in principle, for transforming graphic lambda calculus into a text programming language.

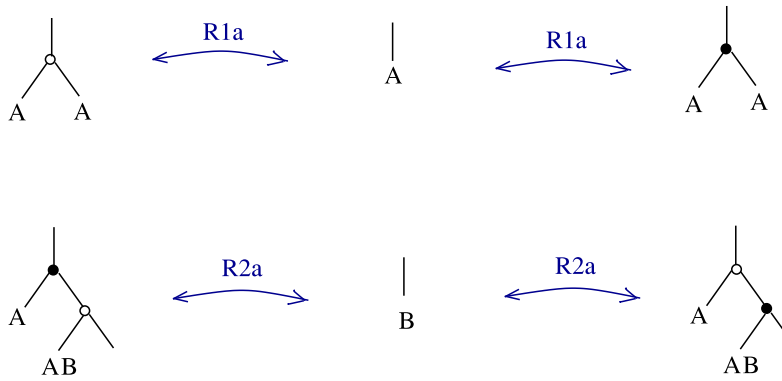
5. Emergent Algebras

Emergent algebras [12, 13] are a distillation of differential calculus in metric spaces with dilations [14]. This class of metric spaces contains the "classical" Riemannian manifolds, as well as fractal-like spaces such as Carnot groups or, more generally, sub-Riemannian or Carnot–Carathéodory spaces (see Bellaïche [15] or Gromov [16]), endowed with an intrinsic differential calculus based on some variant of the Pansu derivative [17].

In [14, Section 4] I proposed a formalism for making various calculations easier with dilation structures. This formalism works with moves acting on binary decorated trees, with the leaves decorated with elements of a metric space.

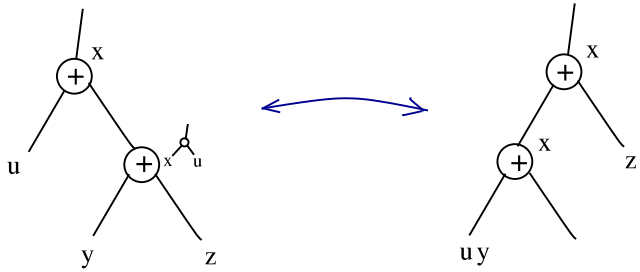
Here is an example of the formalism. The moves are (with the same names as those used in graphic lambda calculus; see the explanation below):

https://doi.org/10.25088/ComplexSystems.22.4.311



Define the following graph (consider it as the graphical representation of an operation $u + v$ with respect to the basepoint x):

Then, in the binary trees formalism the following "approximate" associativity relation can be proved, by using the moves R1a, R2a (it is approximate because a basepoint appears that is different from x, but is close to x in the geometric context of spaces with dilations):



It was puzzling that the formalism worked without needing to know which metric space is used. Moreover, reasoning with moves acting on binary trees gave proofs of generalizations of results from sub-Riemannian geometry, while classical proofs involve elaborate calculations with pseudo-differential operators. At a close inspection it looked like somewhere in the background there is an abstract nonsense machine that is applied just to the particular case of sub-Riemannian spaces.

https://doi.org/10.25088/ComplexSystems.22.4.311

In this paper I shall take the following pure algebraic definition of an emergent algebra (compare with [12, Definition 5.1]), which is a stronger version of [13, Definition 4.2] of a Γ idempotent right quasigroup, in the sense that I define a Γ idempotent quasigroup here.

Definition 5. Let Γ be a commutative group with neutral elements denoted by 1 and operation denoted multiplicatively. A Γ idempotent quasigroup is a set X endowed with a family of operations $\circ_\varepsilon : X \times X \to X$, indexed by $\varepsilon \in \Gamma$, such that:

1. For any $\varepsilon \in \Gamma \backslash \{1\}$ the pair $(X, \circ_\varepsilon)$ is an idempotent quasigroup; that is, for any $a, b \in X$ the equations $x \circ_\varepsilon a = b$ and $a \circ_\varepsilon x = b$ have unique solutions, and moreover $x \circ_\varepsilon x = x$ for any $x \in X$.
2. The operation $\circ_1$ is trivial: for any $x, y \in X$ we have $x \circ_1 y = y$.
3. For any $x, y \in X$ and any $\varepsilon, \mu \in \Gamma$ we have: $x \circ_\varepsilon \left(x \circ_\mu y \right) = x \circ_{\varepsilon\mu} y$.

Here are some examples of Γ idempotent quasigroups.

Example 1. Real (or complex) vector spaces: let X be a real (complex) vector space, $\Gamma = (0, +\infty)$ (or $\Gamma = \mathbb{C}^*$), with multiplication as the operation. We define for any $\varepsilon \in \Gamma$ the following quasigroup operation: $x \circ_\varepsilon y = (1 - \varepsilon) x + \varepsilon y$. These operations give to X the structure of a Γ idempotent quasigroup. Note that $x \circ_\varepsilon y$ is the dilation, based at x, of coefficient ε, applied to y.

Example 2. Contractible groups: let G be a group endowed with a group morphism $\phi : G \to G$. Let $\Gamma = \mathbb{Z}$ with the operation of integer addition (thus we may adapt Definition 5 to this example by using "$\varepsilon + \mu$" instead of "$\varepsilon\mu$" and "0" instead of "1" as the name of the neutral element of Γ). For any $\varepsilon \in \mathbb{Z}$ let $x \circ_\varepsilon y = x \, \phi^\varepsilon \left(x^{-1} y \right)$. This is a $\mathbb{Z}$ idempotent quasigroup. The most interesting case is the one when ϕ is a uniformly contractive automorphism of a topological group G. The structure of these groups is an active exploration area; see, for example, [18] and the bibliography therein. A fundamental result here is Siebert [19], which gives a characterization of topologically connected, contractive, locally compact groups as being nilpotent Lie groups endowed with a one-parameter family of dilations, that is, almost Carnot groups.

Example 3. A group with an invertible self-mapping $\phi : G \to G$ such that $\phi(e) = e$, where e is the identity of the group G. It looks like Example 2 but shows that there is no need for ϕ to be a group morphism.

5.1 Local Versions

We may accept that there is a way (definitely needing a careful formulation, but intuitively clear) to define a local version of the notion of a

https://doi.org/10.25088/ComplexSystems.22.4.311

Γ idempotent quasigroup. With such a definition, for example, a convex subset of a real vector space gives a local $(0, +\infty)$ idempotent quasigroup (as in Example 1) and a neighborhood of the identity of a topological group G. An identity-preserving, locally-defined, invertible self map (as in Example 3) gives a $\mathbb{Z}$ local idempotent quasigroup.

Example 4. A particular case of Example 3 is a Lie group G with the operations defined for any $\varepsilon \in (0, +\infty)$ by $x \circ_\varepsilon y = x \exp\left(\varepsilon \log\left(x^{-1} y\right)\right)$.

Example 5. A less-symmetric example is the one of X's being a Riemannian manifold, with associated operations defined for any $\varepsilon \in (0, +\infty)$ by $x \circ_\varepsilon y = \exp_x\left(\varepsilon \log_x(y)\right)$, where exp is the metric exponential.

Example 6. More generally, any metric space with dilations is a local idempotent (right) quasigroup.

Example 7. One-parameter deformations of quandles. A quandle is a self-distributive quasigroup. Now, take a one-parameter family of quandles (indexed by $\varepsilon \in \Gamma$) that also satisfies points 2 and 3 from Definition 5. What is interesting about this example is that quandles appear as decorations of knot diagrams [4, 5], which are preserved by the Reidemeister moves (more on this in Section 6). At closer examination, Examples 1 and 2 are particular cases of one-parameter quandle deformations!

The operations of approximate sum and approximate difference associated to a Γ idempotent quasigroup are defined next.

Definition 6. For any $\varepsilon \in \Gamma$ we give the following names to several combinations of operations of emergent algebras:

- The approximate sum operation is $\Sigma_\varepsilon^x(u, v) = x \bullet_\varepsilon ((x \circ_\varepsilon u) \circ_\varepsilon v)$.
- The approximate difference operation is $\Delta_\varepsilon^x(u, v) = (x \circ_\varepsilon u) \bullet_\varepsilon (x \circ_\varepsilon v)$.
- The approximate inverse operation is $\mathrm{inv}_\varepsilon^x u = (x \circ_\varepsilon u) \bullet_\varepsilon x$.

Here is the approximate sum operation for Example 1:

$$\sum_\varepsilon^x (u, v) = u(1 - \varepsilon) - x + v.$$

It is clear that, as ε converges to 0, this becomes the operation of addition in the vector space with x as a neutral element. It might be said that it is the operation of addition of vectors in the tangent space at x, where x is seen as an element of the affine space constructed over the vector space from Example 1.

This is a general phenomenon that becomes really interesting in noncommutative situations, as in the examples from the end of the provided list.

https://doi.org/10.25088/ComplexSystems.22.4.311

These approximate operations have many algebraic properties that can be found by the abstract nonsense of manipulating binary trees.

Another construction that can be done in emergent algebras is taking finite differences (at a high level of generality, not bound to vector spaces).

Definition 7. Let $A : X \to X$ be a function (from X to itself, for simplicity). Here is the finite difference function associated to A, with respect to the emergent algebra over X, at a point $x \in X$:

$$T^x_\varepsilon A : X \to X\,,\ T^x_\varepsilon A(u) = A(x) \bullet_\varepsilon (A(x) \circ_\varepsilon u)).$$

For Example 1, the finite difference has the expression:

$$T^x_\varepsilon A(u - x) = A(x) + \frac{1}{\varepsilon}(A(x + \varepsilon u) - A(x)),$$

which is a finite difference indeed. In more generality, for Example 2 this definition leads to the Pansu derivative [17].

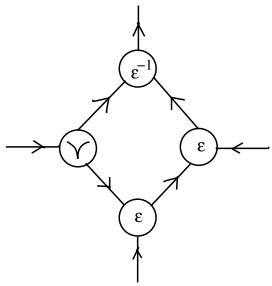
Finite differences as defined here behave like discrete versions of derivatives. Again, the proofs consist of manipulating well-chosen binary trees.

All this can be formalized in graphic lambda calculus, thus transforming the proofs into computations inside graphic lambda calculus.

I shall not stress this further, with the exception of describing the emergent algebra sector of graphic lambda calculus.

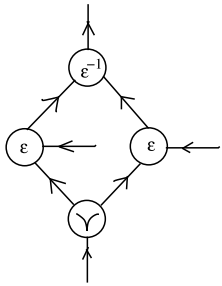
Definition 8. For any $\varepsilon \in \Gamma$, the following graphs in *graph* are introduced.

- The approximate sum graph Σ_ε:

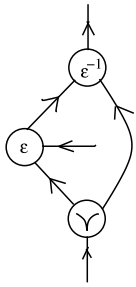


https://doi.org/10.25088/ComplexSystems.22.4.311

- The approximate difference graph Δ_ε:



- The approximate inverse graph inv_ε:



Let A be a set of symbols $a, b, c, \ldots$ (these symbols will play the role of scale parameters going to 0). With A and with the Abelian group Γ we construct a larger Abelian group $\overline{\Gamma}$ that is generated by A and by Γ.

Now we introduce the emergent algebra sector (over the set A).

Definition 9. The subset of *graph emer*(A) (over the group $\overline{\Gamma}$) is generated by the following list of gates.

- Arrows and loops.
- The Υ and $\overline{\varepsilon}$ gates for any $\varepsilon \in \Gamma$.
- The approximate sum gate Σ_a and the approximate difference gate Δ_a, for any $a \in A$.

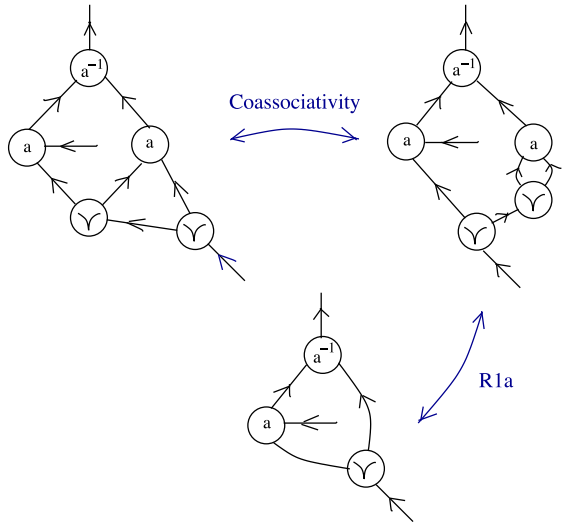
The operations of linking output to input arrows need the following list of moves.

- Fan-out moves.
- Emergent algebra moves for the group $\overline{\Gamma}$.
- Pruning moves.

The set *emer*(A) with the given list of moves is called the emergent algebra sector over the set A.

https://doi.org/10.25088/ComplexSystems.22.4.311

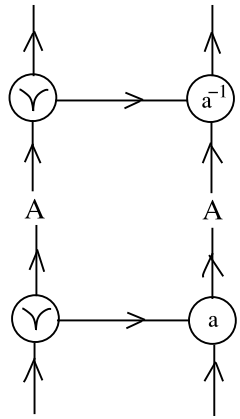
The approximate inverse is not included in the list of generating gates because we can easily prove that for any $a \in A$ we have $\mathrm{inv}_a \in emer(A)$. (If $\varepsilon \in \Gamma$ then we trivially have $\mathrm{inv}_\varepsilon \in emer(A)$ because it is constructed from emergent algebra gates decorated by elements in Γ that are in the list of generating gates.) Here is the proof: we start with the approximate difference Δ_a and an Υ gate. We arrive at the approximate inverse inv_a by the following sequence of moves:



We proved the following relation for emergent algebras: $\Delta_a^x(u, x) = \mathrm{inv}_a^x u$. This relation appears as a computation in graphic lambda calculus.

For the finite differences, we proceed as in Definition 10.

Definition 10. A graph $A \in graph$, with one input and one output distinguished, is computable with respect to the group $\overline{\Gamma}$ if the following graph



https://doi.org/10.25088/ComplexSystems.22.4.311

can be transformed by the moves from graphic lambda calculus into a graph made by assembling:

- graphs from *emer*(*A*)
- gates λ, $\wedge$, and $\top$

It would be interesting to mix the emergent algebra sector with the lambda calculus sector (in a sense this is already suggested in Definition 10). At first view, it seems that the emergent algebra $\overline{\varepsilon}$ gates are operations that are added to the lambda calculus operations, the latter being more basic than the former. I think this is not the case. In the formalism of lambda-scale calculus [8, Theorem 3.4] (of which graphic lambda calculus is a visual variant), I show to the contrary that emergent algebra gates could be applied to lambda terms and the result would be a collection, or hierarchy, of lambda calculi, organized into an emergent algebra structure. This is surprising, at least for the author, because the initial goal of introducing lambda-scale calculus was to mimic lambda calculus with emergent algebra operations.

6. Crossings

In this section we discuss tangle diagrams and graphic lambda calculus.

An oriented tangle is a collection of wires in three-dimensional space; more precisely it is an embedding of an oriented one-dimensional manifold in three-dimensional space. Two tangles are the same up to topological deformation of the three-dimensional space. An oriented tangle diagram is, visually, a projection of a tangle, in general position, on a plane. More specifically, an oriented tangle diagram is a globally planar oriented graph with 4-valent nodes representing crossings of wires (as seen in the projection), along with supplementary information about which wire passes over the respective crossing. A locally planar tangle diagram is an oriented graph that satisfies the previous description, with the exception that it is only locally planar. Visually, a locally planar tangle diagram looks like an ordinary one, except that there may be crossings of edges of the graph that are not tangle crossings (i.e., nodes of the graph).

The purpose of this section is to show that we can "simulate" tangle diagrams with graphic lambda calculus. This can be expressed more precisely in two ways. The first way is to define "crossing macros," which are certain graphs that play the role of crossings in a tangle diagram (i.e., we can express the Reidemeister moves, described later, as compositions of moves from graphic lambda calculus

https://doi.org/10.25088/ComplexSystems.22.4.311

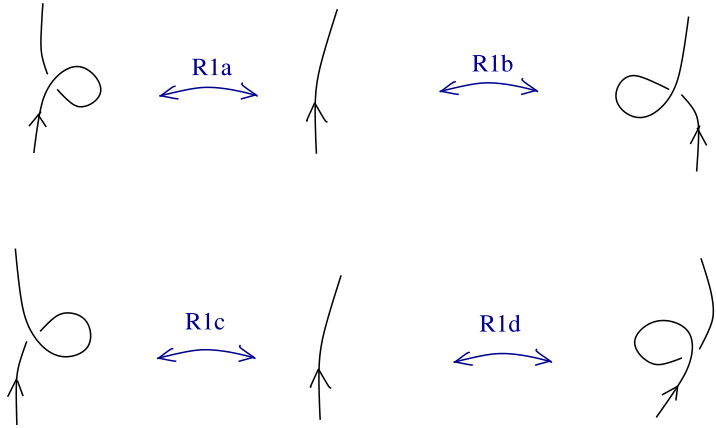
between such graphs). The second way is to associate a graph in *graph* to any tangle diagram such that a certain composition of moves from graphic lambda calculus is associated to any Reidemeister move.

Meredith and Snyder [7] achieve this goal using pi-calculus instead of graphic lambda calculus. Kauffman, in the second part of [6], associates tangle diagrams to combinators and writes about "knotlogic."

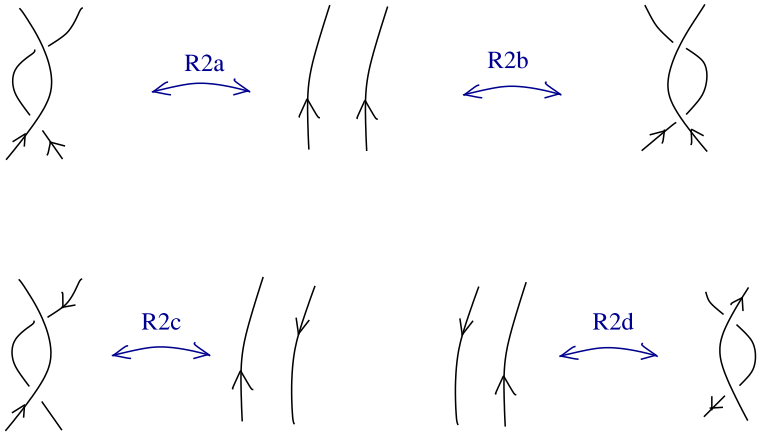
6.1 Oriented Reidemeister Moves

Two tangles are the same, up to topological equivalence, if and only if any tangle diagram of one tangle can be transformed by a finite sequence of Reidemeister moves into a tangle diagram of the second tangle. Here are the oriented Reidemeister moves (using the same names as Polyak [20], but with the letter Ω replaced by the letter R).

- Four oriented Reidemeister moves of type 1:

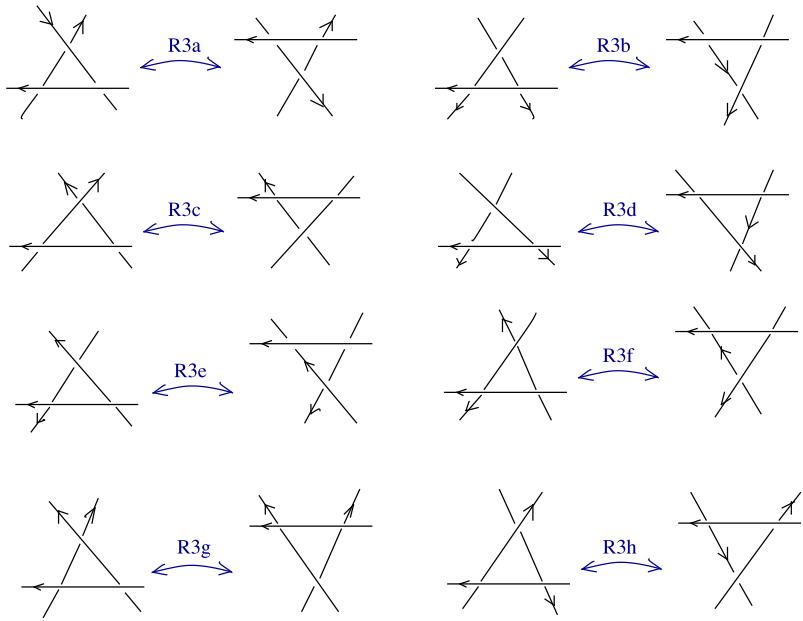


- Four oriented Reidemeister moves of type 2:



https://doi.org/10.25088/ComplexSystems.22.4.311

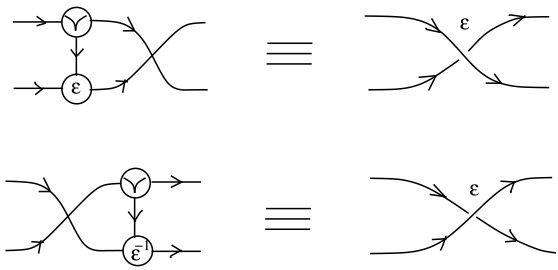
- Eight oriented Reidemeister moves of type 3:



6.2 Crossings from Emergent Algebras

Section 5, Example 7 mentions that there is a connection between tangle diagrams and emergent algebras, via the notion of a quandle. Quandles are self-distributive idempotent quasigroups that were invented as decorations for the arrows of a tangle diagram, which are invariant with respect to the Reidemeister moves.

Here are the emergent algebra crossing macros:



We can choose to neglect the ε decorations of the crossings, or, on the contrary, we can do as in Definition 9 and add a set A to the group Γ and use even more nuanced decorations for the crossings.

https://doi.org/10.25088/ComplexSystems.22.4.311

In [9, Sections 3 through 6] the use of these crossings for exploring emergent algebras and spaces with dilations is presented. All constructions and reasonings from there can be put into the graphic lambda calculus formalism. Here I shall explain only some introductory facts.

Let us associate to any locally planar tangle diagram T a graph in $[T] \in graph$, called the translation of T, which is obtained by replacing the crossings with the emergent crossing macros (for a fixed ε). Also, to any Reidemeister move we associate its translation in graphic lambda calculus, consisting of a local move between the translations of the left- and right-hand side tangles that appear in the respective move. (Note: these translations are not added to the moves that define graphic lambda calculus.)

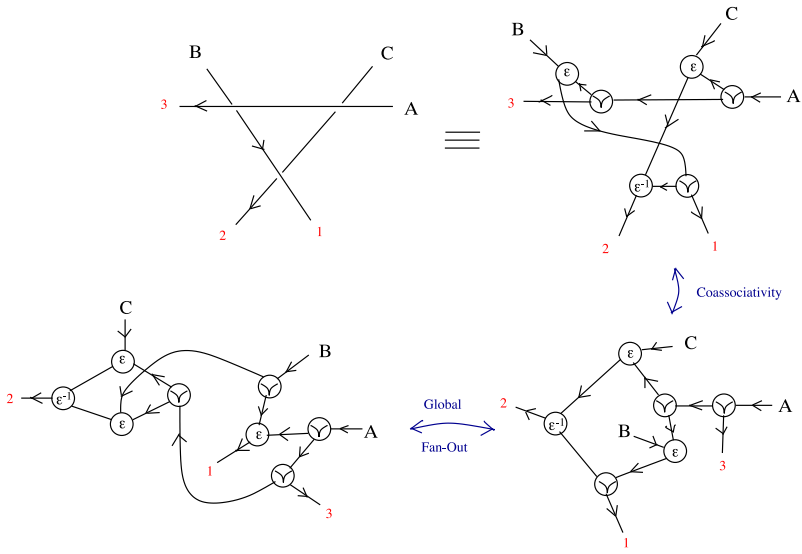
Theorem 2. The translations of all oriented Reidemeister moves of types 1 and 2 can be realized as sequences of the following moves from graphic lambda calculus: emergent algebra (i.e., R1a, R1b, R2, ext2), fan-out (i.e., cocommutativity, coassociativity, global fan-out), and pruning. More precisely, the translations of the Reidemeister moves R1a, R1b are, respectively, the graphic lambda calculus moves R1a, R1b, modulo fan-out moves. Moreover, all translations of Reidemeister moves of type 2 can be expressed in graphic lambda calculus with the R2, fan-out, and pruning moves.

The proof is left to the interested reader; see however [9, Section 3.4].

The fact that Reidemeister moves of type 3 are not true for (the algebraic version of) the emergent algebras, that is, that the translations of the type 3 moves cannot be expressed as a sequence of moves from graphic lambda calculus, is a feature of the formalism and not a weakness. This is explained in detail in [9, Sections 5 and 6], but unfortunately at the moment of writing that paper, the graphic lambda calculus was not available. It would be interesting to express the constructions from the mentioned sections in [9] as statements about computability in the sense of Definition 10 with translations of certain tangle diagrams.

As a justification for this point of view, let us remark that all tangle diagrams that appear in the Reidemeister moves of type 3 have translations that are related to the approximate difference or approximate sum graphs from Definition 8. For example, let us take the translation of the graph from the right-hand side of the move R3d and call it D. This graph has three inputs and three outputs. Let us then consider a graph formed by grafting three graphs A, B, and C at the inputs of D, such that A, B, and C are not otherwise connected. Then we can perform the following sequence of moves:

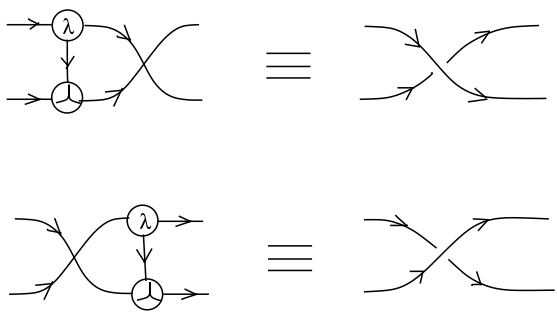
https://doi.org/10.25088/ComplexSystems.22.4.311



The lower-left graph is formed by an approximate difference, a $\overline{\varepsilon}$ gate, and several Y gates. Therefore, if A, B, and C are computable in the sense of Definition 8, then the initial graph (the translation of the left-hand side of R3d with A, B, and C grafted at the inputs) is also computable.

6.3 Graphic Beta Move as Braiding

We now construct crossings, in the sense previously explained, using gates from lambda calculus:



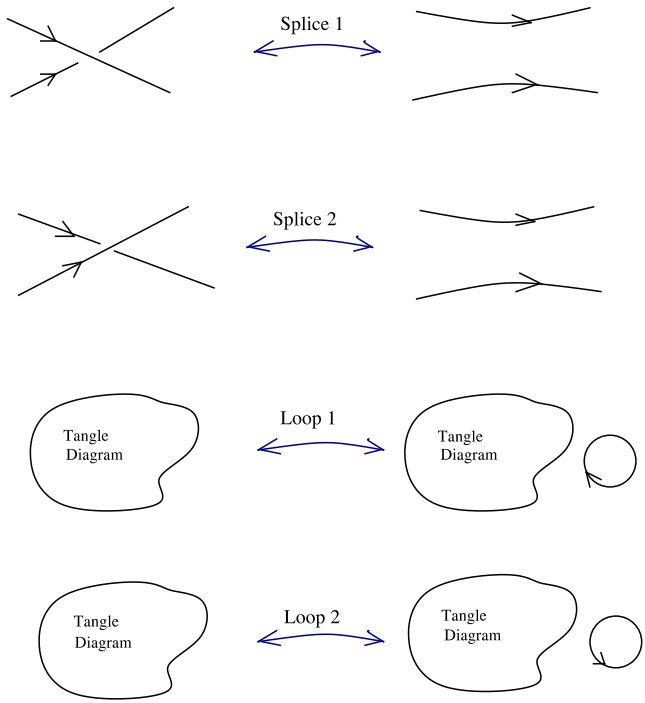
As previously, we define translations of (locally planar) tangle diagrams into graphs in *graph*. There is a one-to-one correspondence between the class of locally planar tangle diagrams and a class of graphs in *graph*. We call this the λ-tangle class.

We could proceed in the inverse direction; namely, consider the class of λ-tangle graphs, along with graphic beta moves and elimina-

https://doi.org/10.25088/ComplexSystems.22.4.311

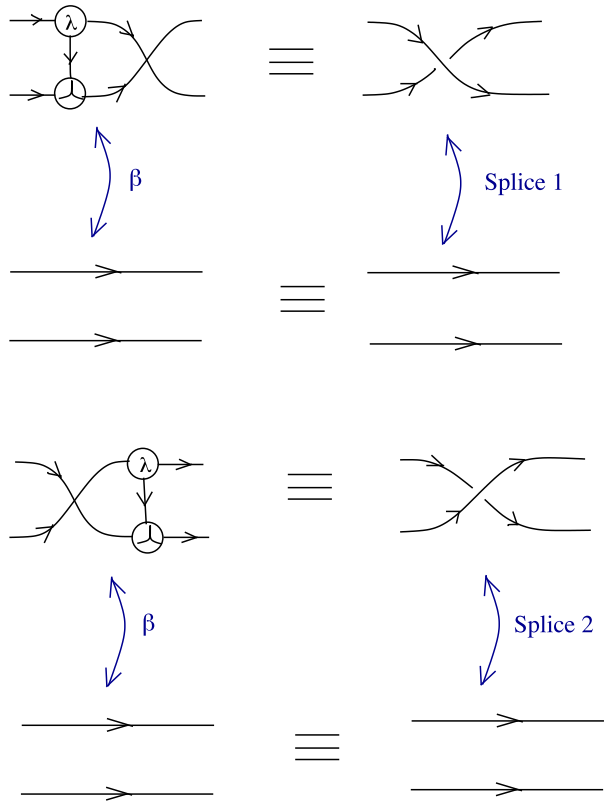
tion of loops. Then, we make the (inverse) translation of graphs in λ-tangle into locally planar tangle diagrams and the (inverse) translation of the graphic beta move and the elimination of loops. Proposition 2 explains what we obtain.

Proposition 2. The class of λ-tangle graphs is closed with respect to the application of the graphic beta move and elimination of loops. The translations of the graphic beta and elimination of loops moves are the Splice 1, 2 (translation of the graphic beta move) and Loop 1, 2 (translation of the elimination of loops) moves:



Proof. The proposition becomes obvious if we find the translation of the graphic beta move. There is one translation for each crossing:

https://doi.org/10.25088/ComplexSystems.22.4.311



Likewise, there are two translations for the elimination of loops, depending on the orientation of the loop that is added/erased. □

Theorem 3 clarifies which oriented Reidemeister moves can be expressed as sequences of graphic lambda calculus moves applied to graphs in a λ-tangle. Among these moves, some are more powerful than others, as witnessed by the following.

Theorem 3. All the translations of an oriented Reidemeister move into moves between graphs in a λ-tangle, except R2c, R2d, R3a, and R3h, can be realized as sequences of graphic beta and elimination of loops moves. Moreover, the translations of moves R2c, R2d, R3a, and R3h are equivalent up to graphic beta and elimination of loops moves (i.e., any of these moves, together with graphic beta and elimination of loops, generate the other moves in this list).

Proof. It is easy, but tedious, to verify that all the mentioned moves can be realized as sequences of splice and loop moves. It is easy to verify that the moves R2c, R2d, R3a, and R3h are equivalent up to splice and loop moves. It is not obvious that the moves R2c, R2d, R3a, and R3h cannot be realized as a sequence of splice and loop moves. In or-

https://doi.org/10.25088/ComplexSystems.22.4.311

der to do this, we prove that R2d cannot be generated by splice and loop. Thanks are due to Peter Kravchuk for the idea of the proof, given in an answer to a question I asked on mathoverflow [21], where I described the moves splice and loop.

To any locally planar tangle diagram A associate its reduced diagram $R(A)$, which is obtained by the following procedure: first use splice 1,2 from left to right for all crossings, then use loop 1,2 from right to left in order to eliminate all loops present at this stage. Note the following.

- The order of applying splice moves does not matter, because they are applied only once per crossing. There is a finite number of splices, equal to the number of crossings. Define the bag of splices $splice(A)$ to be the set of splice moves applied.
- The same is true for the order of loop elimination by loop 1,2. There is a finite number of loop eliminations, because the number of loops (at this stage) cannot be bigger than the number of edges of the initial diagram. Define the bag of loops $loop(A)$ to be the set of all loops present after all splices are done.

Let us now check that the reduced diagram does not change if one of the four moves is applied to the initial diagram.

Apply a splice 1,2 move to the initial diagram A, from left to right, and get B. Then $splice(B)$ is what is left in the bag $splice(A)$ after taking out the respective splice. Also, $loop(B) = loop(A)$ because of the definition of bags of loops. Therefore, $R(A) = R(B)$.

Apply a splice 1,2 from right to left to A and get B. Then, $R(A) = R(B)$ by the same proof, with A, B switching places.

Apply a loop 1,2 move from left to right to A and get B. The new loop introduced in the diagram does not participate in any crossing (therefore, $splice(A) = splice(B)$), so we find it in the bag of loops of B made by all the elements of $loop(A)$ and this new loop. Therefore, $R(A) = R(B)$. The same goes for loop 1,2 applied from right to left.

Finally, the reduced diagram of the left-hand side of the move R2d is different from the reduced diagram of the right-hand side of the move R2d; therefore, the move R2d cannot be achieved with a sequence of splices and loop addition/elimination. □

Acknowledgments

This work was supported by a grant of the Romanian National Authority for Scientific Research, CNCS UEFISCDI, project number PN-II-ID-PCE-2011-3-0383.

https://doi.org/10.25088/ComplexSystems.22.4.311

References

[1] M. Erwig, "Abstract Syntax and Semantics of Visual Languages," *Journal of Visual Languages and Computing*, **9**(5), 1998 pp. 461–483. doi:10.1006/jvlc.1998.0098.

[2] W. Citrin, R. Hall, and B. Zorn, "Programming with Visual Expressions," *Proceedings of the 11th IEEE Symposium on Visual Languages*, Darmstadt, 1995 pp. 294–301. doi:10.1109/VL.1995.520822.

[3] D. C. Keenan. "To Dissect a Mockingbird: A Graphical Notation for the Lambda Calculus with Animated Reduction." (Oct 2, 2013) http://dkeenan.com/Lambda.

[4] R. Fenn and C. Rourke, "Racks and Links in Codimension Two," *Journal of Knot Theory and its Ramifications*, **1**(4), 1992 pp. 343–406.

[5] D. Joyce, "A Classifying Invariant of Knots; the Knot Quandle," *Journal of Pure and Applied Algebra*, **23**(1), 1982 pp. 37–65. doi:10.1016/0022-4049(82)90077-9.

[6] L. Kauffman (ed.), *Knot Logic: Knots and Applications* (*Series on Knots and Everything, Vol. 6)*, River Edge, NJ: World Scientific, 1995.

[7] L. G. Meredith and D. F. Snyder, "Knots as Processes: A New Kind of Invariant." http://arxiv.org/abs/1009.2107.

[8] M. Buliga, "λ-Scale, a Lambda Calculus for Spaces with Dilations." http://arxiv.org/abs/1205.0139.

[9] M. Buliga, "Computing with Space: A Tangle Formalism for Chora and Difference." http://arxiv.org/abs/1103.6007.

[10] J. J. Koenderink, "The Brain a Geometry Engine," *Psychological Research*, **52**(2–3), 1990 pp. 122–127.

[11] D. P. Thurston, "The Algebra of Knotted Trivalent Graphs and Turaev's Shadow World." http://arxiv.org/abs/math/0311458.

[12] M. Buliga, "Emergent Algebras." http://arxiv.org/abs/0907.1520.

[13] M. Buliga, "Braided Spaces with Dilations and sub-Riemannian Symmetric Spaces," in *Geometry. Exploratory Workshop on Differential Geometry and its Applications* (D. Andrica and S. Moroianu, eds.), Cluj-Napoca, Romania: Cluj University Press, 2011 pp. 21–35. http://arxiv.org/abs/1005.5031.

[14] M. Buliga, "Dilatation Structures I. Fundamentals," *Journal of Generalized Lie Theory and Applications*, **1**(2), 2007 pp. 65–95. http://arxiv.org/abs/math/0608536.

[15] A. Bellaïche, "The Tangent Space in sub-Riemannian Geometry," in *Sub-Riemannian Geometry* (A. Bellaïche and J.-J. Risler, eds.), *Progress in Mathematics*, **144**, Boston: Birkhäuser, 1996 pp. 4–78.

[16] M. Gromov, "Carnot-Carathéodory Spaces Seen from Within," in *Sub-Riemannian Geometry* (A. Bellaïche and J.-J. Risler, eds.), *Progress in Mathematics*, **144**, Boston: Birkhäuser, 1996 pp. 79–323.

https://doi.org/10.25088/ComplexSystems.22.4.311

[17] P. Pansu, "Métriques de Carnot-Carathéodory et quasi-isométries des espaces symétriques de rang un," *Annals of Mathematics*, **129**(2), 1989 pp. 1–60.

[18] H. Glöckner, "Contractible Lie Groups over Local Fields," *Mathematische Zeitschrift*, **260**(4), 2008 pp. 889–904. doi:10.1007/s00209-008-0305-x.

[19] E. Siebert, "Contractive Automorphisms on Locally Compact Groups," *Mathematische Zeitschrift*, **191**(1), 1986 pp. 73–90. doi:10.1007/BF01163611.

[20] M. Polyak, "Minimal Generating Sets of Reidemeister Moves." http://arxiv.org/abs/0908.3127.

[21] M. Buliga. "Oriented Reidemeister Move R2d by Splices and Loop Adding/Erasing?" MathOverflow. (Oct 2, 2013) mathoverflow.net/questions/128511.

https://doi.org/10.25088/ComplexSystems.22.4.311