Artificial chemistry experiments with chemlambda, lambda calculus, interaction combinators

Marius Buliga

Institute of Mathematics, Romanian Academy
P.O. BOX 1-764, RO 014700
Bucureşti, Romania

Marius.Buliga@imar.ro , mbuliga@protonmail.ch

31.03.2020

Abstract

Given a graph rewrite system, a graph G is a quine graph if it has a non-void maximal collection of non-conflicting matches of left patterns of graphs rewrites, such that after the parallel application of the rewrites we obtain a graph isomorphic with G. Such graphs exhibit a metabolism, they can multiply or they can die, when reduced by a random rewriting algorithm.

These are introductory notes to the pages of artificial chemistry experiments with chemlambda, lambda calculus or interaction combinators, available from the entry page chemlambda.github.io [13]. The experiments are bundled into pages, all of them based on a library of programs, on a database which contains hundreds of graphs and on a database of about 150 pages of text comments and a collection of more than 200 animations, most of them which can be re-done live, via the programs. There are links to public repositories of other contributors to this experiments, with versions of these programs in python, haskell, awk or javascript.

1 Introduction

The experiments are bundled into pages, all of them based on a library of programs, on a database which contains hundreds of graphs and on a database of about 150 pages of text comments and a collection of more than 200 animations, most of them which can be re-done live, via the programs.

The entry page of the library [13] contains links to the following pages:

- Interaction combinators and chemlambda quine graphs, to explore various quine graphs from chemlambda and interaction combinators [23] and to generate new ones. Given a graph rewrite system, a graph G is a quine graph if it has a non-void maximal collection of non-conflicting matches of left patterns of graphs rewrites, such that after the parallel application of the rewrites we obtain a graph isomorphic with G. Such graphs exhibit a metabolism, they can multiply or they can die, when reduced by a random rewriting algorithm.

- How to test a quine, which allows you to check or to prove that a graph is indeed a quine graph or not

- The quine lab allows you to input or output an interesting graph or it's reductions, with examples to play with. The mol format used to encode the graphs is a variant of linear graphs [2] [3], or port graphs [27] familiar to those interested in interaction nets [24] and linear logic [20].

For those interested in lambda calculus [15] there are two pages:

- Lambda calculus to chemlambda parser, to transform untyped lambda terms into chemlambda molecules, to reduce them graphically and then to output the result,

- The Ouroboros explains how the first chemlambda quine graph, the ouroboros, was discovered from the graphical reduction of a lambda term.

Finally, for those interested into the possibilities of these artificial chemistry, or into artificial life, there is a big collection of animations, with a more artistical nature. These animation are produced from simulations performed with the programs from the library, with a minimal editing concerning only the speed and the framing of the visual output. They come with abundant comments.

- Chemlambda collection of animations

- Same collection with bigger animations

was produced initially as a Google+ collection, after an effort of public research communication. It is now enhanced by the possibility to re-done the animations in javascript, with the programs from the library.

The entry page contains links to more informations and, more importantly, to public repositories of other contributors to this experiment. You find there versions of these programs in python, haskell or my original programs in awk. I want to express my thanks for all contributions. This version of the programs, in javascript, is actually based on the first javascript-only port of my original programs. All the visualizations are done via the excellent d3.js.

2 How not to read these notes

These notes serve to clarify the basics and as a reference for the concepts used in the programs and pages of the library of experiments [13]. You only need a browser with the javascript turned on to start exploring these artificial chemistry experiments.

But if you don't understand what are you looking at, then come back to these notes.

3 About this project

The source of these experiments is the discovery of a graphical formalism for differential calculus in metric spaces known as sub-riemannian spaces. See [4] [5] [6], if you are more interested in geometry or analysis in metric spaces, or [7] for the most recent mix of lambda calculus with emergent algebras.

This formalism of dilation structures, or emergent algebras, composes graph rewriting with a passage to the limit to prove differential or algebraic statements. Trying to understand the computational power of this formalism I started from an algebraic point. Emergent algebras are families of idempotent right quasigroups, indexed by a topological commutative groups, along with a topological structure which allows uniform limits. This algebraic framework did not allow to understand the full generality of the graphical formalism. Therefore I changed to an interaction nets like proposal called graphic lambda calculus [8]. As my intuition was that these graphs and graph rewrites represent a sort of programs with space itself, it follows that the graphical reduction algorithm should be the most simple one. This is how I concentrated on a purely local version of graphic lambda calculus, an artificial chemistry called chemlambda (from "chemistry" and "lambda calculus") which sees graphical rewrites as chemical reactions mediated by enzymes (local machines). Here artificial chemistry is used not in the usual sense [1], [17] but in the sense of asynchronous graph rewrite automata [28].

Chemlambda artificial chemistry was initially proposed with the name chemical concrete machine [9] (an allusion to Berry and Boudol chemical abstract machine [14]). A version of the graph rewrites, but without the algorithm of applications, appears with the name chemlambda in a collaboration with Kauffman [10], where we also explore interactions of knot theory with computation. Knot theory is related to this subject via the fact that idempotent right quasigroups which are also distributive are quandles, or quandles are algebraic structures which encode the Reidemeister moves (graph rewrites) of knot diagrams. Kauffman [21] [22] [25] has a long time interest in the relation between knot theory and computation. Previously with Kauffman, we proposed a model of computation based on graphic lambda calculus and actors in [11], which can be recognized as a mathematicians rediscovery of a version of Bawden [2] [3] linear graphs applications in distributed computing.

The actual version of chemlambda as an artificial chemistry is built into the first collection of programs and the associated demo pages, referenced in this library. These programs appeared from the realization that, besides the mathematician specialized interest, there may be applications of these ideas in chemistry. Lambda calculus has been used with a chemical theme, by Fontana and Buss [18] [19], in their Alchemy (algorithmic chemistry). They consider populations of lambda terms, whose members interact by lambda calculus application. Graph rewriting is of obvious interest for chemistry applications and frameworks like the Graph Grammar Library [26], which allow creation of graph rewriting as chemical reactions rules, existed. The new idea brought by chemlambda was that interaction nets like graph rewriting could be the base of interesting artificial chemistries, or why not, real chemistry. Hence the idea of a molecular computer [12] which is simply a (chemical) molecule (in this toy chemistry) which computes via random chemical rewrites, that is independently, not in a controlled medium or laboratory. The main question was if interesting computations can be done, taking as inspiration graphical reductions of lambda calculus terms, in a interaction nets asynchronous automaton. (Chemlambda however, differently from interaction nets, does have conflicting rewrites.)

The chemlambda project took a completely open turn, based on programs and demonstrations which could be validated by reproduction or by porting of the programs into other

languages (the first programs used awk for the processing and d3.js for visualization). And indeed the project was validated by the other contributors which are listed in the library entry page.

4 Informal description of patterns, mols, graph rewrites

This is an informal description of a variant of linear graphs [2] [3], or port graphs [27] and their double push-out [16] graph rewriting, as they appear in the artificial chemistry experiments described in these notes.

We are concerned with graphs which have a finite number of nodes. Nodes are connected with edges via node ports. The valence of a node is the number of its node ports. Each edge connects two different node ports. Every node port is connected to an edge. Each node has a type from the list NT of node types. The type $A \in NT$ of a node gives the valence $ar(A)$ of the node, as well as a numbering of the node ports (from 0 to $ar(A) - 1$ in the programs, or from 1 to ar(A) in the description).

Any graph made of a finite number of nodes, each node of a type from the list of node types, admits a mol notation. We need an infinite alphabet *Edge* of symbols which we use to decorate the edges of the graph, such that every two different edges have different decorations. We arbitrarily order the nodes of the graph. Then we produce the mol notation of the graph which is a list of node items, one for each node. Each item is a list which starts with the node type, then with the edge decorations of each edge incident to a node port, in the order of the node ports.

For example the list:

$$((A, a, b, c), (A, b, d, e), (B, c, a, d), (A, e, f, f))$$

describes a graph with 4 nodes, 3 of them of type A and one of type B. All nodes have valence 3. There are 6 edges, decorated with the symbols a,b,c,d,e,f. (By using the order of the nodes given by the list) we know that the edge "a" connects the 1st port of the first node with the 2nd port of the second node, and so on until the last edge, "f", which connects the 2nd and 3rd ports of the 4th node.

Further we shall write node items and node words in the following format: one node item per line, without parantheses or commas, like this

$$\begin{array}{cccc} A & a & b & c \\ A & b & d & e \\ B & c & a & d \\ A & e & f & f \end{array}$$

Notice that each edge decoration appears exactly twice.

The mol notation of a graph is unique up to the renaming of the edges (decorations) and up to permutation of the node items. Similarly, two graphs are isomorphic if they admit the same mol notation, again up to renaming of the edges and up to permutation of the node items.

A pattern is a collection of nodes of the graph, together with the internal edges (joining the nodes of the pattern) and with their external half-edges, but seen in the same mol notation. For the example of a graph taken previously, any collection of lines in the mol notation is a pattern, like this one:

$$\begin{array}{cccc} A & a & b & c \\ B & c & a & d \end{array}$$

In the mol notation of a pattern, each edge decoration appears at most twice. Those edge decorations which appear once are the external half-edges.

A match of a pattern into a graph (described by its mol notation) is a pair of functions (f, g), where g is an injective function from the nodes (lines in the list) of the pattern to the nodes of the graph, and f is a renaming of the edge decorations, such that on those those

decorations which appear twice in the pattern the renaming is injective, and on the external half-edges the renaming is at most 2-to-1.

For example if we take the pattern:

$$\begin{array}{cccc} A & 1 & 2 & 3 \\ A & 3 & 4 & 5 \end{array}$$

then there is a match with the graph taken as an example:

$$\begin{array}{cccc} A & 1 & 2 & 3 \\ A & 3 & 4 & 5 \end{array} \quad \rightarrow \quad \begin{array}{cccc} A & a & b & c \\ \mathbf{A} & \mathbf{b} & \mathbf{d} & \mathbf{e} \\ B & c & a & d \\ \mathbf{A} & \mathbf{e} & \mathbf{f} & \mathbf{f} \end{array}$$

Indeed, here the function g matches the 1st line of the pattern with the 2nd line of the mol notation of the graph and the 2nd line of the pattern with the 4th line of the mol notation of the graph. The function f is defined as:

$$f(1) = b\,.\,f(2) = d\,f(3) = e\,f(4) = f(5) = f$$

A graph rewrite $LHS \rightarrow RHS$ is defined by two patterns with the same external half-edges, called the LHS and the RHS patterns of the rewrite. For our example suppose that we have the following graph rewrite:

$$\begin{array}{cccc} A & 1 & 2 & 3 \\ A & 3 & 4 & 5 \end{array} \quad \rightarrow \quad \begin{array}{ccc} C & 1 & 2 \\ C & 4 & 5 \end{array}$$

In the usual style of DPO rewrites [16] (adapted to the mol notation) the application of the graph rewrite is a 3 parts process, where:

- we start from a match of the LHS pattern into the graph, for example the one considered previously,
- we produce an intermediary mol notation by deleting the lines in the mol notation of the graph which are in the scope of the match thus obtaining

$$\begin{array}{cccc} A & a & b & c \\ B & c & a & d \end{array}$$

- we concatenate the intermediary mol notation with a copy of the RHS pattern, such that all the internal edges have fresh decorations (not the case in our example, where the RHS does not have internal edges) and such that the external edges inherit the decoration from the match of the LHS pattern. In our example we get:

$$\begin{array}{cccc} A & a & b & c \\ B & c & a & d \\ \mathbf{C} & \mathbf{b} & \mathbf{d} & \\ \mathbf{C} & \mathbf{f} & \mathbf{f} & \end{array}$$

In order to allow graphs with some free edges we suppose that we always have in NT 1-valent nodes named "FR*" where "*" is a word of our choice. For example a pattern which has external half-edges, like

$$\begin{array}{cccc} A & a & b & c \\ A & c & d & e \end{array}$$

could be seen as the mol notation of a graph with free half-edges a, b, d, e, by adding to it 4 FR* nodes, like this:

$$\begin{array}{cccc} A & a & b & c \\ A & c & d & e \\ FRIN & a & & \\ FRIN & b & & \\ FROUT & c & & \\ FROUT & d & & \end{array}$$

so that now each edge decoration appears exactly twice. This completion of a pattern to a mol notation does not erase the difference between mols and patterns though.

There are two kinds of graph rewrites which we want to allow. We want to allow parallel graph rewrites which don't share nodes, but they do share half-edges. Also we want to allow graph rewrites which delete nodes. That is why we suppose that we always have in NT 2-valent nodes named "Arrow*". Moreover we shall always add to the graph rewrite systems considered a family of graph rewrites called "COMB", which consist into the elimination of the Arrow* elements. For example suppose we have a graph rewrite which deletes a pair of neighboring nodes and joins the dangling half-edges. This can be done by a graph rewrite of the form

$$\begin{array}{cccc} A & 1 & 2 & 3 \\ B & 3 & 4 & 5 \end{array} \quad \rightarrow \quad \begin{array}{ccc} Arrow & 1 & 5 \\ Arrow & 4 & 2 \end{array}$$

This allows to apply several such rewrites in parallel, without confusion concerning the wiring of the edges. After the rewrites are done (one or more in parallel), we end up with the following two situations. A node connected to an Arrow*, or an Arrow* with both ports connected to the same edge. The COMB rewrites are then used to erase the Arrow* elements. For any $A \in NT$

$$\begin{array}{cccc} A & ... & e_i & ... \\ Arrow* & e_i & b & \end{array} \quad \rightarrow \quad A \quad ... \quad b \quad ...$$

and

$$Arrow* \quad a \quad a \quad \rightarrow \emptyset$$

5 Mathematical description of patterns, mols, graph rewrite systems

Linear graphs [2] [3], or port graphs [27] and their double push-out [16] graph rewriting, appear mostly in relation to interaction nets [24] and linear logic [20]. Here we present a variant of this formalism, with a chemical blend, as it used in these experiments.

5.1 Notations

A vector $v = [a_1, ..., a_p]$ is a function $v : \{1, ..., p\} \rightarrow M$, $v(k) = a_k$. The support $supp(v)$ of the vector v is the image of the function v, that is the set of values a_k. The arity of the vector is $ar(v) = p$.

5.2 Molecules

Let NT be a finite, non-empty set of node types or colors and node type arity a function:

$$nar : NT \longrightarrow \mathbb{N}^*$$

A NT-molecule pattern is a triple of sets $M = (H, N, E)$, where N is a NT-colored partition of H into ordered subsets, called nodes, and E is a set of unordered pairs of half-edges, called edges.

Definition 5.1 *A NT-molecule pattern is a triple of sets $M = (H, N, E)$, where:*
(a) H is a finite, non-empty set of half-edges,
(b) N is a set of nodes. Each node $n \in N$

$$n = \left[c, h_1, ..., h_{nar(c)}\right]$$

is colored with a $color(n) = c \in NT$ and has a vector $half(n) = \left[h_1, ..., h_{nar(c)}\right]$ of half-edges incident to the node, of arity $nar(c)$,
(c) E is a set of edges. Each edge $e \in E$ is a set of two different half-edges incident to the edge,

which satisfy the conditions:

(d) for any half-edge $h \in H$ there exists and are unique a node $n = center(h) \in N$ and an index $k = port(h) \in \{1, ..., nar(color(n))\}$ such that $h = half(n)_k$,
(e) for any half-edge $h \in H$ there is at most one edge $e = edge(h) \in E$ such that $h \in e$.

The half-edge $h \in H$ is bound in M if there exists an edge $e = edge(h) \in E$ as in condition (e). We write $h \in Bound(M)$. Otherwise we say that h is free in M, $h \in Free(M)$, and we define $edge(h) = h \in Free(M)$.
A molecule is a molecule pattern without free half-edges.
For completeness we accept as a molecule the empty one $\emptyset$, with no half-edges.

We may supplement the node type arity with a node valence function, which associates to each node type $c \in NT$ a vector

$$nV(c) = [v_1, ..., v_{nar(c)}]$$

with support in $\{0, 1\}$. We say that the port k of the color c is "in" if $nV(c)_k = 0$, otherwise is "out".

An oriented molecule pattern $M = (H, N, E)$ is a molecule as in definition 5.1, with the following modifications: edges are ordered, they have a source and a target and sources of edges are connected to "out" ports of nodes and targets of edges are connected to "in" ports of nodes. More precisely, we have the following modifications of conditions (c) and (e):

(c) E is a set of edges. Each edge $e \in E$ is a pair $(source(e), target(e)) \in H^2$ with $source(e) \neq target(e)$,

(e) for any half-edge $h \in H$ there is at most one edge $e = edge(h) \in E$ such that $h = source(e)$ or $h = target(e)$. Let $n = center(h) \in N$ be the node incident with h and $k = port(h)$ be the index of h in n. Let $c = color(n)$ be the color of the node. If $h = source(e)$ then $nV(c)_k = 1$. If $h = target(e)$ then $nV(c)_k = 0$.

From definition 5.1 we see that a molecule pattern is equally described by the associated functions $color$, $half$, $port$, $center$ and $edge$. A molecule pattern morphism is one which commutes with all these functions.

Definition 5.2 *A morphism from a NT-molecule pattern $M = (H, N, E)$, with associated $color$, $half$, $port$, $center$ and $edge$, to another NT-molecule pattern $M' = (H', N', E')$, with associated $color'$, $half'$, $port$, $center'$ and $edge'$, is a function*

$$F : H \to H'$$

with the properties:
(a) for any $h_1, h_2 \in H$, if $center(h_1) = center(h_2)$ then $center'(F(h_1)) = center'(F(h_2))$
(b) for any $h \in H$ $port'(F(h)) = port(h)$,
(c) for any $h \in H$ $color'(center'((F(h))) = color(center(port(h)))$,
(d) for any $h \in Bound(M)$ we have $F(h) \in Bound(M')$ and

$$edge'(F(h)) = \{F(g) | g \in edge(h)\}$$

Let (NT, nar) and (NT', nar') be two sets of node types with associated arity functions. A connection morphism from a NT-molecule pattern $M = (H, N, E)$, with associated $color$, $half$, $port$, $center$ and $edge$, to another NT'-molecule pattern $M' = (H', N', E')$ is a pair (K, F)

$$K : NT \to NT' \quad , \quad F : H \to H'$$

such that F satisfies conditions (a), (b), (d) and F, K satisfy the modified condition:
(c') for any $c \in NT$ $nar'(K(c)) = nar(c)$ and for any $h \in H$

$$color'(center'((F(h))) = K(color(center(port(h))))$$

For oriented molecule patterns there are similar definitions of morphisms and connection morphisms.

Notice that a morphism does not preserve free half-edges: it is not true, in general, that if $h \in Free(M)$ then $F(h) \in Free(M')$. Indeed, it is possible for the images of two free half-edges in M to form an edge in M'.

If we accept among the node types 1-valent node types called FREE, or FRIN (free in) or FROUT (free out) in the case of oriented molecule patterns, then we can transform any molecule pattern M with $Free(M) \neq \emptyset$ into a molecule $Cap(M)$ simply by adding for any half-edge $h \in Free(M)$ a new half-edge h', a node $[FREE, h']$, (or $[FRIN, h']$ if h is "in" or $[FROUT, h']$ h is "out"), and an edge $\{h, h'\}$ (or $[h', h]$ if h is "in" or $[h, h']$ if h is "out").

Remark that $M \mapsto Cap(M)$ does not induce a functor in the category of mol patterns, because morphisms do not preserve free half-edges. However, if M is isomorphic with M' then $Cap(M)$ is isomorphic with $Cap(M')$.

5.3 The graph associated to a molecule

Let C be a finite, non-empty set of colors and $nar : C \to \mathbb{N}^*$ be a color arity function.

Definition 5.3 *A C-colored graph is a triple $(N, E, color)$ where N is a finite nonempty set of nodes, $color : N \to C$ and E is a set of edges. Each edge is a set of two different nodes.*

If a node belongs to an edge then we say that the edge is incident to the node. If two different nodes belong to an edge then we say that the edge links the nodes.

The arity $ar(n)$ of a node n is the number of edges which are incident to that node. We demand that:

$$ar(n) = nar(color(n))$$

for any node of the graph.

To any molecule M is associated a colored graph $Graph(M) = (Nodes, Edges)$.

Denote by m the maximal arity of nodes in the molecule M. Then we add to the set of colors the set $\{1, ..., m\}$ (we suppose of course that C is disjoint from this set) and we obtain a new set of colors C'. We exetnd the node type arity

$$nar : C \to \mathbb{N}^*$$

to a color arity (denoted the same) defined over C', with

$$nar(k) = 2$$

for each $k \in \{1, ..., m\}$.

To each node $n = [c, h_1, ..., h_p]$ in M, of arity p, we add p + 1 nodes in $Nodes$, the k-th node colored with k, for $k = 1, ..., p$, and the last node colored with $c = color(n)$. This last node is named a center node and the other nodes are named port nodes. We add edges in $Edges$ from the center node to each of its port nodes, we call these internal edges. Finally, for each pair of half-edges which form an edge in M, we add an external edge linking the corresponding node ports.

If M is a molecule pattern with $Free(M) \neq \emptyset$, then by definition

$$Graph(M) = Graph(Cap(M))$$

$Graph(M)$ determines the molecule pattern M up to isomorphism. In general a $M \mapsto Graph(M)$ is a functor only from the category of molecules to the category of graphs, because, again, morphisms of pattern molecules do not preserve free half-edges.

5.4 Mol notation

The mol notation corresponds to Bawden [3] linear graph notation.

Definition 5.4 *An NT-mol node with edge tags in the set Tag is a vector*

$$\bar{n} = \left[c, tag_1, ..., tag_{nar(c)}\right]$$

where $c = color\bar{n}) \in NT$ is the node type of the mol node and $half(\bar{n}) = \left[tag_1, ..., tag_{nar(c)}\right]$ is of tags, elements of Tag, such that any $tag \in Tag$ occurs at most twice.

An NT-mol pattern with edge tags in the set Tag is a vector of NT-mol nodes, such that any $tag \in Tag$ occurs at most twice over all mol nodes vectors.

In the oriented version, when NT comes with a node valence function nV, an NT-mol pattern with edge tags in the set Tag is mol pattern such that every tag $tag \in Tag$ which occurs twice, it does appear two positions (ports) which are "in" and "out".

In general, the tags which occur twice in the mol pattern M are said to be bounded, their set is denoted $Bound(M)$. The tags which occur once we said to be free, their set is denoted $Free(M)$.

A mol pattern (or just mol) is a record of a molecule pattern.

Definition 5.5 *Let $M = (H, N, E)$ be NT-molecule pattern, Tag a set of tags in bijection $tag : E \to Tag$ with E, $Id : \{1, ..., m\} \to N$ a bijective numbering of nodes.*

The mol (notation) of M is the mol pattern $[\bar{n}_1, ..., \bar{n}_m]$, where for each $k \in \{1, ..., m\}$,

$$n = Id(k) = n = \left[c, h_1, ..., h_{nar(c)}\right]$$

$$\bar{n}_k = \left[c, tag(h_1), ..., tag(h_{nar(c)})\right]$$

5.5 Local machines

Given a set of node types NT, with node type arity or node valence function in the oriented case, a set Tag of edge tags, a finite set of special symbols and a natural number K, we define a K-local machine. This is a Turing machine which has two tapes: an IO (input-output tape and a work tape, and a set of internal states, such that the size of the work tape plus the number of possible internal states is at most K.

The tapes are made by cells, each cell can contain a node type, a tag, or a special symbol.

The IO tape has to contain at each moment only a mol pattern and special symbols, like field separators of a node vector or line separator for the mol vector. The IO tape head, after reading a cell, if it reads a tag it can then jump to the beginning of the mol node containing it, or to the cell which contains the other occurence of the tag (if it exists), or to one of the other tags from the same mol node, or to the end of mol pattern.

The machine can read from on the IO tape a cell, write a whole mol node at the end of the mol pattern, or delete a whole mol node. For each read/write or delete from/to the IO tape the machine writes the same on the work tape. The machine can read from the work tape and it can write only on the blank cells of the work tape.

When the machine halts (by writing a halt symbol on the work tape) it either deletes the whole work tape and the IO head jumps to a random mol node beginning, or the machine halts.

The machine may have a source of new tags (not already present on the IO tape) or not. The machine may have access to a random coin.

Section 7.3 explains why we may not need a source for new tags.

6 Molecules for programmers

If you are familiar with linear graphs [2] [3], or port graphs [27] and their double push-out [16] graph rewriting, then you can use this section for faster understanding of the programs in the library.

Such graphs are familiar to programmers interested in interaction nets [24] and linear logic [20]. However, in these experiments, the accent is on the use of the simplest, purely local algorithms, and not on the semantics or the high level point of view. Here we want to understand if these simplest algorithms can do something interesting in the artificial chemistries we consider.

In order to define mols (molecules) and mol patterns, we introduce mol node types and their nodeValence vectors.

6.1 Mol nodes

The mol nodes are defined in the script nodes.js.

Mol nodes types. Let NT be a finite set of mol node types. In the script NT is in the vector autoFilter.

```
autoFilter = [
"L","A","FI","D","FOE","FOX","FO","T","Arrow","GAMMA","DELTA"
];
```

nodeValence associates to any mol node type a valence vector, with elements 0 or 1, whose length is the valence of the mol node.

```
nodeValence = {
  "L":  [0,1,1], // (12) , 1-z
  "A":  [0,0,1], // (231), (z-1)/z
  "FI": [0,0,1], // (312), 1/(1-z)
  "D": [0,0,1],  // ()   , z
  "FOE":[0,1,1], // (23) , 1/z
  "FOX": [0,1,1], // (13) , z/(z-1)
  "FO": [0,1,1], //
  "T":  [0],
  "FRIN":[1],
  "FROUT":[0],
  "Arrow":[0,1],
// interaction combinators
  "GAMMA":[0,0,0],
  "DELTA":[0,0,0],
}
```

Mathematically, given the set NT of mol node types, the nodeValence is a function which associates to a mol node type "t" the word w = nodeValence(t). Here the word w is made of letters "0" and "1". The valence of the mol node type is the length of the word w.

In the definition of nodeValence we see some mol node types with comments. These mol node types can be decorated with permutations of 3 elements, or with elements of the anharmonic group. These decorations are a bridge from the present work to emergent algebras (see for example arXiv:1807.02058) which are "commutative" in the sense that they satisfy a rewrite called "the shuffle trick".

Here we are going to use this correspondence only heuristically, for example to choose the form of the rewrites "DIST", which increase the number of nodes. See more about this correspondence at:

- anharmonic lambda calculus

- the tool to choose DIST rewrites
- the commented js script rhs.js

A detailed explanation of the relation with commutative emergent algebras will appear in the future.

Mol nodes. Given a nonempty, finite set of edge tags E, a mol node whose ports are E-decorated is a vector (t, η) where:

- t is a mol node type, called the type of the mol node
- η is a word over the alphabet E, such that:
  - any letter from E appears at most twice,
  - the length of η equals the valence of t.

For a mol node we shall use a notation like

```
L a b c
```

where "L" is a mol node type and "abc" is the word η with letters from an alphabet E which contains a, b, c. You can see that the mol node type L has nodeValence vector [0,1,1], whose length is 3, equal to the length of the word abc.

6.2 Mol patterns

A mol pattern (which is E-decorated) is a vector of mol nodes, such that any letter from E appears at most twice over all mol nodes.

Any letter which appears exactly once in a mol pattern P, is called a free edge. The set of free edges of the mol pattern P is denoted by Free(P). The set of edges of the mol pattern P which are not free is denoted by Bound(P).

A mol (which is E-decorated), is a mol pattern without free edges.

For mol patterns, in particular for mols, we use a notation like

```
L a b c
A c d e
```

which is amenable to a record with a line separator (here we use newline) which separates the record into lines. Each line is a mol node, itself a record with a field separator (here we use space, therefore we can't admit space or newline in the alphabet E).

Such records are available in iceMol.js, in the form of the function molLibrary(). In those records the line separator is "$\hat{}$".

Note that any mol pattern can be turned into a mol by adding some new mol nodes "FRIN" (i.e. "free in") and "FROUT" (i.e. "free out").

6.3 Molecules

A molecule is an equivalence class of mol patterns, up to the reordering of mol nodes and up to renaming of the edge decorations. This equivalence is defined in terms of morphisms of mol patterns.

Consider two mol patterns:

$P = [(t_i, \eta_i)]_{i=1...n}$, which is E-decorated,

$Q = \left[(t'_j, \eta'_j)\right]_{j=1...m}$, which is F-decorated.

A morphism from P to Q is a pair of functions $[f, d]$, where $f : \{1...n\} \to \{1...m\}$ and $d : E \to F$. The function d induces a function $d^* : E^* \to F^*$ which transforms a word $\eta \in E^*$ into the word $d^*(\eta)$ which is obtained by replacing each letter a of η with the letter $d(a)$. The pair of functions $[f, d]$ satisfies the property: for every $i \in \{1...n\}$ we have

(a) $t'_{f(i)} = t_i$

(b) $\eta'_{f(i)} = d^*(\eta_i)$

(c) f is injective

(d) d restricted to $Bound(P)$ takes values in $Bound(Q)$ and is injective

(e) d restricted to $Free(P)$ is at most 2-to-1.

In this category of mol patterns, two of them are equivalent if they are isomorphic.

Part of the condition (d) (that d takes bound edge decorations to bound edge decorations) and the condition (e) are needed, even if they seem overly complex. In fact we shall identify molecules with decorated graphs as made of half-edges. The mentioned part of the condition (d) means just that the morphism preserves edges of these graphs. The condition (e) allows pairs of half-edges which are not part of edges in (the graph associated to) P to be transformed in pairs of half-edges which form an edge (in the graph associated to) Q.

At the level of the library of programs, we have to be sure that our programs respect this equivalence relation.

A mol pattern contains more information than a molecule, but this information is not geometrical and has to be destroyed. One of the means to keep only the geometrical information is to use random permutations (or morphisms) where needed in the algorithms.

6.4 Molecules as graphs

A molecule can be turned into a graph. This means we need a conversion from a mol pattern to a graph such that two isomorphic mol patterns are converted into the same graph. In order to understand the conversion (done by functions in ioprep.js), we have to understand the relation between mol patterns and graphs.

A graph is pair (G,E) where G is a finite nonempty set of nodes and E is a set of edges. Each edge is a set of two nodes (therefore the nodes have to be different). We say that an edge links the two nodes. There are no edges which connect a node with itself.

An oriented graph is a different mathematical object. It is a pair of functions (source, target) defined on a set E of edges, with values in a set G of nodes. An oriented graph admits edges with the source and target being the same node. If an oriented graph does not have such edges, then it can be turned into a graph (as defined previously) by associating to each edge e the set formed by the nodes source(e) and target(e).

A graph can also be seen as a collection of half-edges, according to the following definition. A graph is a finite set of H of half-edges, with a set N of disjoint sets of half-edges, called nodes, and a set E of disjoint unordered pairs of half-edges, called edges.

We shall use for our graphs the following modification of a graph as a collection of half-edges. Let C be a set of colors. A C-decorated graph is:

- a finite collection of half-edges H,
- with a set N of nodes, where each node is a vector $[c, h_1, ..., h_k]$ where k is the arity of the node and $h_1, ..., h_k$ are half-edges, such that a half-edge $h \in H$ appears exactly once in one of the nodes,
- and with a set of unordered pairs of half-edges, called edges.

We transform a C-decorated graph G into a unoriented usual graph G' with colored nodes. Denote by m the maximal arity of nodes in G. Then we add to the set of colors the set $\{1, ..., m\}$ (we suppose of course that C is disjoint from this set) and we obtain a new set of colors C'. To each node $n = [c, h_1, ..., h_p]$ in G, of arity p, we add p + 1 nodes in G', the k-th node colored with "k", for $k = 1, ..., p$, and the last node colored with "c". This last node is named a center node and the other nodes are named port nodes. We add edges in G' from the center node to each of its port nodes, we call these internal edges. Finally, for each pair of half-edges which form an edge in G, we add an external edge linking the corresponding node ports.

d3 graphs are oriented graphs, in the format used in the js library d3.js. See more about d3 graphs in myD3Graph.js. A d3 graph is given as a pair of vectors, called nodes and links.

The nodes vector has elements, each one (node) is an object

```
{"id": id,
"type": type,
x: x, y: y,
vx:0, vy:0,
links:[],
"age":age}
```

where "id" is the identity (index of the node in the nodes vector), "type" is the graph node type, x, y, vx, vy are the coordinates and velocities (used in the force graph simulation done by d3), links is a vector of links (i.e. edges) connected with the present node. "age" is the age of the node, used in some of the experiments. Possibly other fields may be added.

The links vector describes the edges of the oriented graph. Each links element is an object

```
{"source": nsource,
"target": ntarget,
"value": value,
"age":age}
```

where "source" is the node (index) of the source node, "target" is the node (index) of the target node and "value" is a number which is used for the width of the link (edge) as drawn by d3.js. Internal edges (those from the center node to its port nodes) and the external edges will have different width. The paramater "age" is the age of the link, used in some experiments. Possibly other fields may be added.

An oriented graph structure. like these d3 graphs, do contain non-geometrical information, like the one we mentioned previously concerning mol patterns. We have to be careful to not use this information, or to destroy it somehow, in the algorithms.

Here we use d3 graphs as usual graphs, in the sense that we don't treat edges as oriented. (At the level of the programs, for example starting with a node, we may search for the other nodes which are linked to this one, irrespective to the fact that they are sources or targets). This is done in the following way.

6.4.1 Graph nodes

Graph nodes types. We introduce a set GT of graph node types. Further we exploit the fact that here we use mol node types with valence at most 3 (but the extension to any valence should be obvious).

If NT is the set of mol node types, then GT is obtained from NT by adding 3 new types:

- "in", "middle", "out".

We also have a predicate isCenter (defined in myD3Graph.js), which is true for any graph node type which is not "in", "middle" or "out". We say that a graph node is a center if it has a graph node type for which isCenter is true.

A node which is not a center is called a port node.

GT is kept in the vector graphNodes, in the script nodes.js.

```
graphNodes = [
"in","out","middle",
"L","A","FI","D","FOE","FOX","FO",
"T","FRIN","FROUT","Arrow",
"GAMMA","DELTA"
];
```

Because we want to represent graphically the nodes, we associate colors to each graph node type. Remark that we don't use a bijection from the graph node types to the colors.

Each graph node (which we are going to define from a mol pattern) is graphically represented by a colored circle. The radius of the circle is a function of the predicate isCenter, that is center nodes have a radius and the port nodes have a different radius.

From mol nodes to graph nodes. To each mol node we associate a GT-decorated graph. The association is the following.

There is a center node according to the mol node type. There are port nodes, one for each letter of the mol node word. Here we exploit the fact that we use mol nodes of valence at most 3.

The function which associates a vector of graph node types (among "in", "out" or "middle") to the valence of a mol node is nodePortTypes, in the script nodes.js.

```
nodePortTypes = [
["in"],
["in","out"],
["in","middle","out"]
];
```

So the first port node is always "in", the last one (for valence at least 2) is always "out", the remaining port is "middle".

The graph associated to a mol node is formed by the center node and the port nodes, with edges from the center nodes to the port nodes. Specifically, we draw a center graph node as an "o" and a port node as a dash "—" with a number (starting from 1).

Then a mol node of valence 1, of mol node type "t" and edge "a"

```
t a
```

becomes a graph as in figure 1.

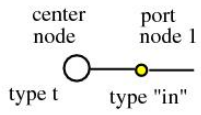

Figure 1: The graph associated to a 1-valent mol node

A mol node of valence 2, of mol node type "t" and edges "ab"

```
t a b
```

becomes a graph as in figure 2. A mol node of valence 3, of mol type "t" and edges "abc"

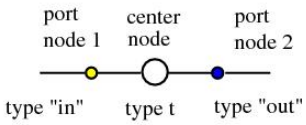

Figure 2: The graph associated to a 2-valent mol node

```
t a b c
```

becomes a graph as in figure 3.

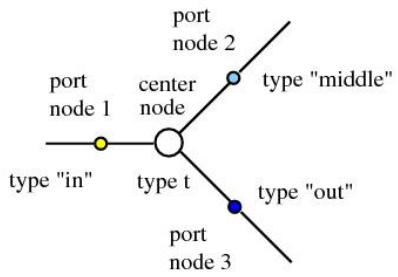

Figure 3: The graph associated to a 3-valent mol node

6.4.2 Mol patterns as graphs

To a mol pattern we associate the graph formed by (the graphs of) all mol nodes, with edges connecting port nodes which have the same edge tag. The definition of a mol pattern assures us that this is possible, because each edge tag cannot occur more than twice.

Let us come back to the d3 graphs, which are oriented. The orientation (i.e. which node is the source and which one is the target) will be neglected. This is possible because the graphs which we obtain do not have edges which connect a node with itself. Indeed, even if an edge tag appears twice in the same mol node, the corresponding edge connects different node ports.

7 Rewrites

A rewrite is an object, like

```
{
left:"A",right:"FO",              \\ the nodes types of the LHS pattern
action:"DIST1",                   \\ the action name
named:"A-FO",                     \\ the name of the rewrite
t1:"FOE",t2:"FOE",t3:"A",t4:"A", \\ the node types of the RHS pattern
blocks:["FOE-A"],                 \\ patterns which have LHS in this RHS
kind:"DIST"                       \\ the general type of rewrite
}
```

7.1 Rewrites of mol patterns

A mol pattern rewrite is a double push-out [16] rewrite. In this case of mol patterns, a general rewrite is defined by a pair of mol patterns $[LHS, RHS]$, with the property that

$$Free(LHS) = Free(RHS)$$

LHS is called the left hand side pattern and RHS is called the right hand side pattern. $Free(LHS) = Free(RHS)$ is the interface.

Given a mol pattern M and a rewrite $[LHS, RHS]$, an application of the rewrite consists of:

- a morphism from LHS to M, with image $match^L(LHS)$
- a morphism from RHS to M, with image $match^R(RHS)$, such that

(var1) $match^R(Free(RHS)) = match^L(Free(LHS))$

(var2) $match^R(Bound(RHS)) \cap Bound(M) = \emptyset$

- from the mol pattern M are eliminated (spliced) the mol nodes of $match^L(LHS)$

- the mol modes of $match^R(RHS)$ are added to the result.

This definition of a rewrite application has to be backed by algorithms which can be implemented by local machines (section 5.5):

(a) from M and LHS output $match^L(LHS)$

(b) from M, LHS, RHS and $match^L(LHS)$ output $match^R(RHS)$

(c) from M, $match^L(LHS)$ and $match^R(RHS)$ output M', the mol pattern after the rewrite.

Among these algorithm, (c) is clear, (a) is a matching algorithm and (b) is more problematic because of the condition (var2), which asks for a way to produce new bound edges, i.e. new names which are not in $Bound(M)$. As (c) is trivial, we may think about it as the final part of the algorithm (b), thus we have:

(a) match algorithm: from M and LHS output $match^L(LHS)$

(b) rewrite algorithm: from M, LHS, RHS and $match^L(LHS)$ output $match^R(RHS)$,then replace $match^L(LHS)$ by $match^R(RHS)$ in M. Output M', the mol pattern after the rewrite.

It is also interesting to consider a finite collection of general rewrites

$$[LHS_1, RHS_1], ..., [LHS_N, RHS_N]$$

Algorithms $(a)_k$ and $(b)_k$ for all $k = 1...N$ can be merged into one (a) algorithm and (b) algorithm. This can be done by abstracting over the node types. Looking back at the definition of molecules as equivalence classes of mol patterns, we could consider a larger equivalence relation, by admitting mol pattern morphisms which do not preserve mol node types. Instead we might treat mol node types as we treated edges tags, perhaps retaining only nodeValence of the node type (or only the nodes arities).

The most general formalism may turn out to be much more verbose than what we need. That is why let's make only the following definition: a connection morphism between two mol patterns

$P = [(t_i, \eta_i)]_{i=1...n}$, which is E-decorated, with mol node types in NT,

$Q = \left[(t'_j, \eta'_j)\right]_{j=1...m}$, which is F-decorated, with mol node types in NT',

is a triple of functions $[f, d, g]$, where $f : \{1...n\} \to \{1...m\}$, $d : E \to F$ and $g : NT \to NT'$ with the properties: for every $i \in \{1...n\}$ we have

(a) $t'_{f(i)} = g(t_i)$

(b) $\eta'_{f(i)} = d^* (\eta_i)$

(c) f and g are injective

(d) d restricted to $Bound(P)$ takes values in $Bound(Q)$ and is injective

(e) d restricted to $Free(P)$ is at most 2-to-1.

A connection pattern is then an equivalence class of mol patterns up to connection isomorphisms.

Connection patterns rewrites are defined like mol patterns rewrites. The advantage is that we may compress a finite collection of general mol pattern rewrites into a much smaller collection of connection patterns rewrites. For a general connection pattern rewrite we shall use the name "action" (which explains the field "action" in the description of a rewrite).

At the action level, all LHS patterns we shall consider are of the form.

```
n1type e ...
n2type ...e...
```

where n1type and n2type are the types of the nodes. To identify such a LHS pattern we need to specify

```
left:n2type, right:n1type,    \\ the two nodes types of the LHS pattern
named:n2type + "-" + n1type,  \\ the name of the rewrite
```

Here the "left" and "right" node type come from the image that the edge "e" is oriented from left to right, with the mol node of type n1type at right.

This notation for a LHS pattern supposes that there is only one connection pattern for a pair of node types n1type, n2type. The connection pattern will appear as a predicate in the algorithm (a).

Each action (i.e. connection pattern rewrite) is identified by a name, for our example:

```
action:"DIST1",                    \\ the action name
```

Finally, the RHS patterns are also given via their connection patterns, for our example:

```
t1:"FOE",t2:"FOE",t3:"A",t4:"A", \\ the node types of the RHS pattern
```

The algorithm (b) takes as input:

- the LHS mol pattern $match^L(LHS)$, given by the algorithm (a)
- the action name (which gives the connection pattern rewrite) and the node types needed to build the $match^R(LHS)$ pattern.

The problem of how to generate new names which are not in $Bound(M)$ remains.

7.2 Graph rewrites

At the graph level, the connection pattern

```
n1type e ...
n2type ...e...
```

translates into a center node of type $n1type$, whose first node port is connected to a node port of a center node of type $n2type$. The match algorithm (a) is the function findTransform(n1) from chemistry.js, where n1 is the center node of type $n1type$. The function findAllTransforms() uses the previous function for all center nodes to make a vector of all possible graph rewrites.

The algorithm (b) is embodied by the function doTransform(n1, trans)from chemistry.js, which takes as input the center node n1 and a matching graph rewrite trans. Instead of generating new names (of nodes or edges), the algorithm delegates this to d3.js part, by using functions like addNodeAndEdges, or removeNodeAndEdges, which are defined in myD3Graph.js.

7.3 A solution of the problem of new names

Project Hapax gives such a solution, described here.

The idea is simple: we can exclude the need of new names if we reformulate all rewrites as permutations of edges. As an example, let's consider the rewrite "L-A", as it appears in chemlambda (and which is a well known graphical version of the β rewrite). In terms of LHS, RHS mol patterns, this rewrite transforms the LHS

```
L c b a
A a d e
```

into the RHS

```
Arrow c e
Arrow d b
```

(Here we don't need new edge names, but the example is easy to follow.)

This rewrite can be reformulated as: transform the new LHS' pattern

```
L c b a
A a d e
Arrow b' a'
Arrow a' b'
```

into the new RHS' pattern

```
L b' a b'
A a' a a'
Arrow c e
Arrow d b
```

This can be further restated as a chemical reaction:

```
L c b a       Arrow b' a'        Arrow c e       L b' a b'
A a d e   +   Arrow a' b'   -->  Arrow d b   +   A a' a a'
```

which has the form

```
LHS   +   Token1   -->  RHS   +   Token2
```

Here $Token1$ and $Token2$ are 2-mol nodes patterns which have to be added to the rewrite in order to make it conservative.

The solution thus transfers the problem of generating new names to the problem of generating new tokens. Or, we know how to reliably generate new tokens. We may imagine that $Token1$, $Token2$ are money tokens and that each rewrite involves a cost (but mind that all it really matters is that there exist well known ways to generate new tokens in a decentralized way).

References

[1] W. Banzhaf, L. Yamamoto, Artificial Chemistries, MIT Press, Cambridge, (2015)

[2] A. Bawden, Connection graphs, In: Proceedings of the 1986 ACM conference on LISP and functional programming, 258-265, ACM Press, (1986)

[3] A. Bawden, Implementing Distributed Systems Using Linear Naming, A.I. Technical Report No. 1627, MIT, (1993)

[4] M. Buliga, Dilatation structures I. Fundamentals, *J. Gen. Lie Theory Appl.*, **1**, 2, 65-95, (2007)

[5] M. Buliga, Infinitesimal affine geometry of metric spaces endowed with a dilatation structure, *Houston Journal of Mathematics*, **36** 1, 91-136, (2010)

[6] M. Buliga, A characterization of sub-riemannian spaces as length dilation structures constructed via coherent projections, *Commun. Math. Anal.* **11**, No. 2, 70-111, (2011)

[7] M. Buliga, The em-convex rewrite system, arXiv: 1807.02058, (2018)

[8] M. Buliga, Graphic lambda calculus, *Complex Systems* **22**, 4, 311-360, (2013)

[9] M. Buliga, Chemical concrete machine, Figshare, (2013)

[10] M. Buliga, L.H. Kauffman, Chemlambda, universality and self-multiplication, ALIFE 2014: The Fourteenth International Conference on the Synthesis and Simulation of Living Systems July 2014, MIT Press, 490-497, (2014)

[11] M. Buliga, L.H. Kauffman, GLC actors, artificial chemical connectomes, topological issues and knots, arXiv:1312.4333, (2013)

[12] M. Buliga, Molecular computers, arXiv:1811.04960, (2015-2018)

[13] M. Buliga, Chemlambda page, https://chemlambda.github.io/index.html (2020)

[14] G. Berry, G. Boudol, The chemical abstract machine, *Theoretical Computer Science* **96**, 1, 217-248, (1992)

[15] A. Church, An unsolvable problem of elementary number theory, *Amer. J. of Math.*, **58**, 2, (1936), 345-363

[16] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Loewe, Algebraic approaches to graph transformation, part I: Basic concepts and double push-out approach, Handbook of Graph Grammars and Computing by Graph Transformation, 163-245 (1997)

[17] Dittrich, P., Ziegler, J., & Banzhaf, W. (2001). Artificial ChemistriesA Review. *Artificial Life*, **7** (3), 225-275

[18] W. Fontana and L. W. Buss, What would be conserved if 'the tape were played twice'?, *Proc. Natl. Acad. Sci. USA* 91, 757-761, (1994)

[19] W. Fontana and L. W. Buss, 'The Arrival of the Fittest': Toward a Theory of Biological Organization, Bull. Math. Biol., 56, 1-64 (1994)

[20] J.-Y. Girard (1987), Linear Logic, *Theoretical computer Science* **50** 1, 1-101

[21] L.H. Kauffman, Knot automata. In Proceedings of TheTwenty-Fourth International Symposium on Multiple-Valued Logic, Boston, Massachusetts, 328-333, (1994)

[22] L.H. Kauffman, Knot logic. In Kauffman, L., editor, Knot sand Applications, 1110. World Scientific Pub. (1994)

[23] Y. Lafont, Interaction Combinators, *Information and Computation* **137**, 1, 69-101, (1997)

[24] Y. Lafont, From proof nets to interaction nets, Advances in Linear Logic, eds. J.-Y. Girard, Y.Lafont, LMS Lecture Notes Series 222, (2010), 225-248

[25] S.J. Lomonaco, L.H. Kauffman, , Proc. SPIE 8057, Quantum Information and Computation IX, 805702 (3 June 2011)

[26] M. Mann, H. Ekker, C. Flamm, The Graph Grammar Library - a generic framework for chemical graph rewrite systems, In: Proceedings of the International Conference on Model Transformation (ICMT 2013), D. Varro and K. Duddy and G. Kappel (eds), Springer-Verlag Berlin Heidelberg, LNCS 7909, pp. 52-53, (2013)

[27] C. Stewart (2002), Reducibility between classes of port graph grammar, *Journal of Computer and System Sciences* **65**,169-223, (2002)

[28] K. Tomita, H. Kurokawa, S. Murata, Graph-rewriting automata as a natural extension of cellular automata. Chapter 14 in: Adaptive Networks, Understanding Complex Systems, T. Gross, H. Sayama (eds.), NECSI Cambridge/Massachusetts (2009)