MGS: Implementing a Unified View on Four Biologically Inspired Computational Models

Jean-Louis Giavitto & Olivier Michel

LaMI u.m.r. 8042 du CNRS
Université d'Evry Val d'Essone
91025 Evry Cedex, France.
Phone: +33 1.60.87.39.04
[giavitto,michel]@lami.univ-evry.fr

Abstract

We present the first results in the development of a new programming language called **MGS**. This language proposes a unified view on four computational mechanisms initially inspired by biological processes. Our approach is to formulate the basic computation step as the application of a morphism preserving the topological structure of a collection. By changing the topological structure of collection, the underlying computational model is changed.

The **MGS** language is devoted to the simulation of biological process, especially those whose state space must be computed jointly with the running state of the system (for instance, in morphogenesis). This requires a firm control of the morphisms applications. We present several features that help in this task.

Keywords: Gamma, CHAM, P system, L system, Cellular Automata, Rewriting, Topological Collection, Declarative Programming Language, Simulation of Biological Processes.

1 Introduction and Motivations

Several fundamental computational mechanisms have been inspired from nature, see for instance the "Parallel Problem Solving from Nature" (PPSN) conference series. Amongst them, some originate in a biological or biochemical metaphor. Therefore, it is not surprising to consider them as the basis of a programming language dedicated to the simulation of biological *dynamical systems* (DS). The hope is that the underlying concepts reveal as particularly well fitted to the specification and simulation of *DS with a dynamical structure.* Standard DS exhibit a static structure, that is, the exact phase space of the DS can be known statically before the simulation. This is not the case for several DS found in biology [Fon92, FB94, FB96] like plant growing, developmental biology (e.g. embryogenesis), integrative cell models, protein transport and compartment simulation, etc.

In this kind of situation, the dynamic of the system is often specified as several local competing transformations occurring in an organized set of simpler entities. The organization of this set is subject to possible drastic changes in the course of time.

This abstract view makes possible the unification in a same programming language of several biologically inspired computational models, namely : Γ and the CHAM, P systems, L systems and cellular automata. For the readers convenience, we give in the rest of this section a short overview of these models biased towards our goals. Then we present the basic idea for their integration into a single framework. This unification relies on the notion of *topological collection* discussed in section 2. Transformations of topological collections are introduced more concretely in section 3.2.

Nota Bene: we do not claim that we have achieved a useful theoretical framework encompassing the four cited formalisms. We advocate that few notions and a single syntax can be consistently used to allow the merging of these formalisms *for programming* purposes. It leads to the development of an experimental programming language called `MGS`. `MGS` is a vehicle used to investigate general notions of collections and transformations and to study their adequacy to the simulation of various biological processes. Simple examples of `MGS` programs are given in section 4. All examples are processed using the current version of the `MGS` interpreter. This presentation finishes by the review of some directions opened by this research.

1.1 The programming language Γ and the CHAM

The computational model underlying Γ [BM86, BCM87] is based on the chemical reaction metaphor; the data are considered as a multiset M of molecules and the computation is a succession of chemical reactions according particular rules. A rule (R, A) indicates which kind of molecules can react together (a subset m of M that satisfies predicates R) and the product of the reaction (the result of applying function A to m). Several reaction happen at the same time. No assumption is made on the order on which the reactions occurs. The only constraint is that if the reaction condition R holds for at least one subset of elements, at least one reaction occurs (the computation does not stop until the reaction condition does not hold for any subset of the multiset).

The CHemical Abstract Machine (CHAM) extends these ideas with a focus on the expression of semantic of non deterministic processes [BB89].

1.2 P systems

P systems [Pau98, Pau00] are a new distributed parallel computing model based on the notion of a membrane structure. A membrane structure is a nesting of cells represented, e.g, by a Venn diagram without intersection and with a unique superset: the skin. Objects are placed in the regions defined by the membranes and evolve following various transformations: an object can evolve into another object, can pass trough a membrane or dissolve its enclosing membrane. As for Γ, the computation is finished when no object can further evolve.

We do not pursue this description, assuming that the reader is familiar with the P systems formalism.

1.3 L systems

L systems are a formalism introduced by A. Lindenmayer in 1968 for simulating the development of multicellular organism. Related to abstract automata and formal language, this formalism has been widely used for the modeling of plants. A L system can be roughly described as a grammar. The productions are applied in parallel in a non deterministic manner. 0L systems are context-free grammars. D0L system are deterministic context-free grammars: given a letter A there is at most one production rule that can be applied. Parametric L systems deal with *module* instead of letters: a module is a letter associated with a list of parameters. The production rules are extended with side-conditions on the parameters. For example,

$$A(x, y) : x \leq 3 \quad \longrightarrow \quad A(2x, x + y)$$

is a rule that can be applied to the module $A(2, 5)$ to gives the module $A(4, 7)$. This rule cannot be applied on $A(7, 1)$ because the first parameter x does not match the condition.

1.4 Cellular Automata

Cellular automata (CA) have been invented many times under different names: tessalation automata, cell spaces, iterative arrays, etc. However, a fair fraction of the computer research on two-dimensional cellular automata has its ultimate origins in the work of J. Von Neumann to provide a more realistic model for the behavior of complex systems in biology [VN66].

In a simple case, a 2D cellular automaton consists in a grid of cells or sites, each with a value taken in a finite set $\mathcal{V}$. The values are updated in a sequence of discrete time steps, according to a definite, fixed, rule. Denoting the value of a site at position (i, j) by $a_{i,j}$, a simple rule gives its new value as $a'_{i,j} = \varphi(a_{i,j}; a_{k_1}, ..., a_{k_p})$, where φ is a function from $\mathcal{V}^{p+1}$ to $\mathcal{V}$, and where the a_{k_j} are the values of the p neighbors of site (i, j). For example, the Von Neumann neighbors of a cell (i, j) are the four cells $(i-1, j), (i+1, j), (i, j-1)$ and $(i, j+1)$.

Many variations are possible: organization of the cells in a regular lattice of any dimensions or even in a general graph, variable neighborhood, various finite set $\mathcal{V}$. However the main characteristics of CA are largely unaffected by such additional complications.

2 Topological Collections and their Transformations

2.1 Motivations

The previous formalisms have been developed with various goals in mind, e.g. parallel programming for Γ, semantic modeling of nondeterministic processes for the CHAM, calculability and complexity issues for P systems, formal language theory and biological modeling for L systems, parallel distributed model of computation for CA (this list is by no means exhaustive). This has motivated the development of various tools and concepts for their studies.

However, they share some important features, with respect to their possible use as a foundation for a programming language dedicated to DS with a dynamical structure. They can be themselves viewed abstractly as a formalism for specifying discrete dynamical systems. From this point of view, the following facts can be stressed.

Discrete space and time. What is playing the role of the state (the multiset in Γ, the membranes hierarchy in a P system, the word in a L system and the array in a CA) consists of a discrete *collection* of values. This discrete collection of values, the state, evolves in a sequence of discrete time steps.

Temporally local transformation. The computation of a new value in the new state depends only on previously computed values for a fixed number of preceding steps (usually just one step).

Spatially local transformation. The computation of a new collection is done by a structural combination of the results of more elementary computations involving only a small and static subset of the initial collection.

'Structural combination', means that the the elementary results are combined into a new collection, irrespectively of their precise value. 'Small and static subset' explicites that only a fixed subset of the initial elements are used to compute a new element value (this is measured for instance by the diameter of the evolution rule of a P systems, the local neighborhood of a CA, the number of variables in the right hand side of a Γ reaction or the context of a rule in a L system).

Considering these shared characteristics, the main difference between the formalisms appears to be the structure of the collection. Abstractly, the computational mechanism is always the same:

1. a subcollection A is selected in a collection C;

2. a new subcollection B is computed from A;

3. the collection B is substituted for A in C.

We call these three basic three a ***transformation***. In addition of transformation specification, there is a need to account for the various constraints in the selection of the subcollection A and the replacement B.

Constraining the Subcollections. There is a priori no constraint in the case of Γ: one element or many elements are replaced by zero, one or many elements. In the case of P systems, the evolution of a membrane may affect only the immediate enclosing membrane (by expelling some tokens or by dissolution): there is a *localization* of the changes. This is also the case for L systems: the new collection B is inserted at the place of A and not spread out over C. For CA, the changes are not only localized, but also A and B are constrained to have the same shape. This is more visible when considering a variant of the CA where it is allowed to specify rules updating simultaneously a set of cells. This extension does not add any computational power, because they can be emulated by a standard CA. However, such facility is very important in an application domain like lattice gas, see Fig. 1. When a pair A of contiguous cells is selected, then B must also be a pair of contiguous cells, because the grid must retain its topology in the transformation.

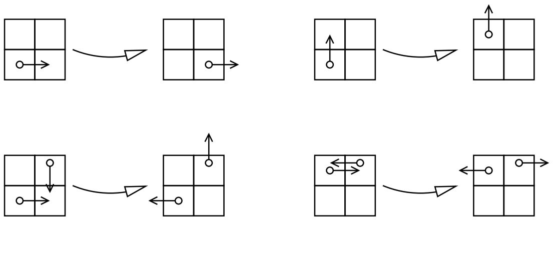


Figure 1: Some rules for a lattice gas automata.

Collections as Spaces. Considering these constraints and their expressions, it is very natural to see a collection as a set of *places* or *positions* organized by a *topology* defining the *neighborhood* of each element in the collection. To stress the importance of the topological organization of the collection's elements, we call them ***topological collection***.

This approach is part of a long term research effort [GMS96] developed for instance in [Gia00] where the focus is on the substructure and in [GM01a] where a general tool for uniform neighborhood definition is developed.

In the rest of this section, we propose a topological framework that unifies the previous formalisms as transformations of some topological collections.

2.2 Definitions

The reader not interested in the formal development may skip to section 3. We want to speak about a space made of places and the neighborhood of a place in this space. This sort of space will be the carrier of a topological collection. It is convenient to describe this space as build place by place. We use the operator $+$ to denote the gluing of places. This operator is commutative and associative because the order used to construct the space does not matter. An added place can also be retired. So, the set of places as the structure of an Abelian group. The element 0 represents no space at all (or a space with no place).

There is two ways for a place X to be connected with Y: because they share a common boundary or because they are both boundaries of a "bigger" object. This notion of bigger corresponds to the notion of the dimension of a space and enables the ranking of the set of places. These considerations motivates the following definitions[1]. They are well suited for their computer implementation because of their algebraic nature. The definition we introduce are illustrated by their application in paragraph 2.3.

[1]The definition of a chain complex is standard in algebraic topology [HY88, Hen94]. However, we give a special case, tailored with respect to our needs, of a more general definition.

Definition 1 (Cell, Chain, Boundary Maps and Involutive Chain Complex).

An *involutive chain complex* is a sequence $\mathcal{C} = (K_p, C_p, \partial_p)_{p \in \mathbb{N}}$ such that for all p:

- K_p is a set of elements called abstract p-dimensional cells or in short a p-cells. A 0-cell is also called a vertex. The element of K_p and K_q are distinct for $p \neq q$.
- C_p is the formal Abelian group with generators K_p and equations $c + c = 0$ for all $c \in K_p$ (these are the only proper equations between the group generators). An element of C_p is called a p-chain.
- The function $\partial_p : C_p \to C_{p-1}$ is a group homomorphism called a *boundary map*. The boundary maps must satisfy the condition $\partial_p \circ \partial_{p+1} = 0$.

If it exists q such that for all $p > q$, C_p is reduced to the trivial group $\{0\}$, then the dimension of the complex is q.

A chain complex will be the underlying organization of a topological collection. Intuitively, the element of K_p are the elementary places and the p-chains are all the spaces that we can build with them. There is essentially only one way to build a composite space with the cells.

Definition 2 (Proper Sum). Let $\mathcal{C} = (K_p, C_p, \partial_p)$ an involutive chain complex. A sum $c_1 + ... + c_k$ in C_p is called *proper* if for all $1 \leq i \leq k$, c_i belongs to K_p and $c_i \neq c_j$ for $i \neq j$.

Because of the way the groups C_p are defined, an element c of C_p can be written as a unique proper sum up to the permutations of the sum elements. If $c = \sum c_i$ and $\sum c_i$ is proper, then we write $c \equiv \sum c_i$. We also write $b \in c$ whenever $c \equiv b + c_1 + ... + c_n$.

Let $c = \sum c_i$ an element of C_p. Because ∂_p is an homomorphism, we have $\partial_p c = \sum \partial_p c_i$. In other word, the boundary of a sum of elements is the sum of the boundaries of the elements. The condition $\partial_p \circ \partial_{p+1} = 0$ in the chain complex definition reflects the intuitive property that "the border of something has no border itself". The specific construction of the groups C_p corresponds to the following technical trick[2] used to compute easily the boundary of a composite space; see Fig. 2. Suppose that $s \in C_p$ is composed of two subspaces s' and s''; this is denoted by $s = s' + s''$. Suppose than s' and s'' share only one element $a \in K_{p-1}$. Then a is not in the border of s because s' and s'' are glued along a. But a is in the boundary of s' and in the boundary of s''. Let $\partial_p s' \equiv a + \sum a'_j$ and $\partial_p s'' \equiv a + \sum a''_k$. Then we must have: $a + \sum a'_j + a + \sum a''_k = \sum a'_j + \sum a''_k$ which is automatically achieved when $a + a = 0$.

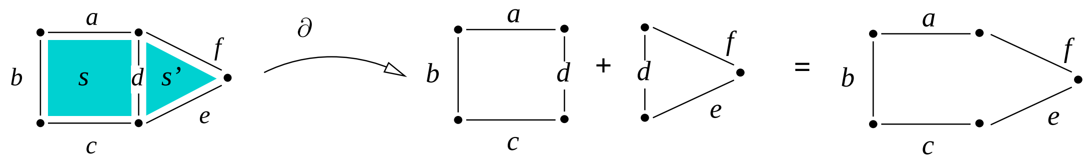


Figure 2: Example of the application of the boundary operator. Note that the complex represented here cannot be modeled as an abstract simplicial complex, nor a simplicial set.

Definition 3 (Summation and Setification). If S is a set $\{s_1, ..., s_n\}$ where all the s_i belongs to some C_p, we write $\sum S$ for the sum $s_1 + ... + s_n$. Let $s \in C_p$ such that $s \equiv \sum c_i$, then $set(s)$ denotes the subset $\{c_1, ...\}$ of K_p.

Definition 4 (Coboundary). Let $\mathcal{C} = (K_p, C_p, \partial_p)$ a chain complex. The family of homomorphisms $\delta_p : C_{p-1} \to C_p$ defined by

$$\delta_p b = \sum \{c \mid c \in C_{p+1} \text{ and } b \in \partial_{p+1} c\}$$

are called the coboundary maps of the chain complex $\mathcal{C}$. It is easy to check that $\delta_{n+1} \circ \delta_n = 0$.

[2]This approach avoids the problem of the orientation of each cell and is sufficient for our purposes here.

The coboundary of a p-cell s is the chain of the $(p+1)$-cell containing s as a boundary. The property $\delta_{n+1} \circ \delta_n = 0$ can be rephrased as: the coboundary of a coboundary is empty.

DEFINITION 5 (Subcomplex and Subcomplex Generated by a Set). An involutive chain complex $\mathcal{C} = (K_p, C_p, \partial_p)$ is a *subcomplex* of a complex $\mathcal{C}' = (K'_p, C'_p, \partial'_p)$ if for all p: K_p is a subset of K'_p, C_p is a subgroup of C'_p and $\partial_p x = \partial'_p x$ for all $x \in K_p$.

Let $\mathcal{C}' = (K'_p, C'_p, \partial'_p)$ a complex and $I_q \subset K'_q$. The *subcomplex generated by* I_q in $\mathcal{C}'$ is a subcomplex $\mathcal{C}'/I_q = (K_p, C_p, \partial_p)$ of $\mathcal{C}'$ such that:

- $K_q = I_q$,
- for $j < q$, $\quad K_j = set(\partial'_j \sum K_{j+1})$
- for $j > q$, $\quad K_j = \{c \mid c \in K'_J \text{ and } set(\partial'_j c) \subseteq K_{j-1}\}$

DEFINITION 6 (Connected Chain). Two elements s and s' in C_p are 0*-connected* if $set(s) \cap set(s') \neq \emptyset$ or $set(\partial s) \cap set(\partial s') \neq \emptyset$ or $set(\delta s) \cap set(\delta s') \neq \emptyset$. A chain $s \equiv c_1 + ... + c_n$ is *connected* if for all (i,j): c_i and c_j are 0-connected or it exists c_k such that c_i and c_k are 0-connected and c_k and c_j are connected.

DEFINITION 7 (Simple Topological Collection).

A *simple topological q-collection* F is a triple $(\mathcal{C}, f, q)$ where $\mathcal{C} = (K_p, C_p, \partial_p)$ is a chain complex and $f : K_q \to Val$ is a partial function from K_q to some set of values Val.

A simple q dimensional collection associates a value to some q-cells. We say that $\mathcal{C}$ is the *organization* of F and that q is the *dimension* of F. The set of the simple topological collections of organization $\mathcal{C}$ and dimension q is denoted by $\mathrm{STC}^q_{\mathcal{C}}$. When there are no ambiguities, we leave the mention of $\mathcal{C}$ and q implicit.

We need some notations to manipulate functions. The definition domain of a function f is written $\mathtt{dom}(f)$. The restriction of a function f to a set X, written f/X, is the function with domain $\mathtt{dom}(f) \cap X$ which coincide with f.

DEFINITION 8 (Extension and Shape of a Collection). Let $F = (\mathcal{C}, f, q)$ a collection of dimension q. The definition domain $\mathtt{dom}(F)$ of a collection is the definition domain of the associated function $\mathtt{dom}(f)$. The q-chain $\mathtt{ext}(F) = \sum \mathtt{dom}(f)$ is called the *extension* of F. If $\mathtt{ext}(F) \equiv c_1 + c_2 + ... + c_k$, then we write $F = f(c_1).c_1 + f(c_2).c_2 + ... + f(c_k).c_k$. The *shape* $\mathtt{sh}(F)$ of F is the element $\partial_q(\mathtt{ext}(F))$ of C_{q-1}.

DEFINITION 9 (Subcollection, Generated Subcollection and Neighborhood).

Let $F = (\mathcal{C}, f, q)$ and $G = (\mathcal{C}', g, q)$ two q-dimensional collections. F is a *subcollection* of G iff $\mathcal{C}$ is a subcomplex of $\mathcal{C}'$ and $f \sqsubseteq g$ where $\sqsubseteq$ is the Scott order defined on functions by: $f \sqsubseteq g \Leftrightarrow \mathtt{dom}(f) \subseteq \mathtt{dom}(g)$ and for all $x \in \mathtt{dom}(f)$, $f(x) = g(x)$. If F is a subcollection of G, we write $F \sqsubseteq G$.

Let $I \subset K'_q$. The subcollection $G/I = (\mathcal{C}'/I, g/I, q)$ is the *subcollection generated by* I.

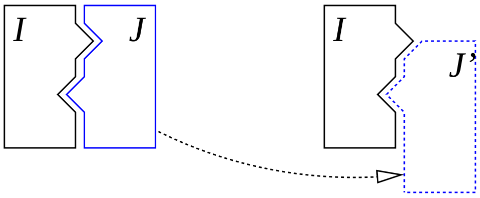
Let F a subcollection of a q-collection G. The *neighborhood* of F in G, denoted by $\widetilde{F}$, is the subcollection G/I of G where $I = set(\delta_{q-1}\,\mathtt{sh}(F)) \cup \mathtt{dom}(F)$.

An *elementary evolution* is a collection where a connected subcollection A has been substituted by a collection B in a collection C. The shapes of A and B must agree on the unchanged part of C. These notions are illustrated in Fig. 3. A *transformation* happens when an evolution can be computed as a function of the neighborhood of a subcollection.

DEFINITION 10 (Elementary Evolution). Let two topological collections $F = (\mathcal{C}, f, q)$ and $F' = (\mathcal{C}', f', q)$ with cells K_p and K'_p. Let $I = \{c \mid c \in K_q \cap K'_q, f(c) = f'(c)\}$, $J = K_q - I$ and $J' = K'_q - I$ where the *set difference* $A - B$ is defined by $A - B = \{a \mid a \in A \text{ and } a \notin B\}$. If $I \neq \emptyset$ and $F/I = F'/I$ then we say that F' is an *elementary evolution* of F. We write $F \rhd F'$ for the subcollection F/J and $F \mathrel{\widetilde{\rhd}} F'$ for $F/\widetilde{J}$.

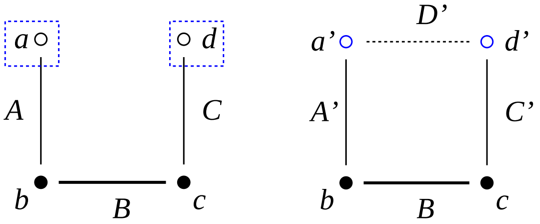
If J is connected we say that the evolution is *local.* If $\mathcal{C} = \mathcal{C}'$ the evolution is *organization-preserving.* The evolution is *shape preserving* if $\partial \sum J = \partial' \sum J'$.

Definition 11 (Elementary Functional Transformation). Let $T : \mathrm{STC}^q \to \mathrm{STC}^q$ a function that maps a q-collection to another one. If it exists a function φ such that for all $F \in \mathrm{STC}^q$, the q-collection $T(F) = (\mathcal{C}', f', q)$ is an elementary evolution of F such that $f' = \varphi(F \mathbin{\tilde{\triangleright}} T(F))$, then T is an *elementary functional transformation* of simple topological collections. *Local, shape-preserving* and *organization-preserving* functional transformations are defined in the same way. The transformation is *non-cooperative* if it exists a function φ' such that $\varphi(F \mathbin{\tilde{\triangleright}} T(F)) = \varphi'(F \triangleright T(F))$ for all F.



$q = 0$ (the values of the collections are associated with the vertices), $I = b + c$ and the p-cells of F/I are $K_0 = \{b, c\}$ and $K_1 = \{B\}$.

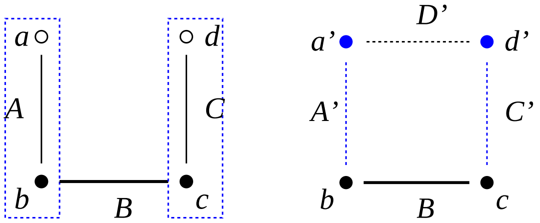
$$\begin{aligned} J &= a + d \\ \partial J &= \emptyset \\ \delta J &= A + C \end{aligned}$$



$$\begin{aligned} J' &= a' + d' \\ \partial J' &= \emptyset \\ \delta J' &= A' + C' \end{aligned}$$

$q = 1$ (the values of the collections are associated with the edges), $I = B$ and the p-cells of F/I are $K_0 = \{b, c\}$ and $K_1 = \{B\}$.

$$\begin{aligned} J &= A + C \\ \partial J &= a + b \\ &\quad +c + d \\ \delta J &= \emptyset \end{aligned}$$



$$\begin{aligned} J' &= A' + C' + D' \\ \partial J' &= b + c \\ \delta J' &= \emptyset \end{aligned}$$

Figure 3: Examples of an elementary evolution. We write J and J' as chains instead of sets (see the definition 10 in the text). This evolution involves the drastic change of a linear sequence into a circle. This last kind of organization cannot be easily represented by a sequence, an array or a multiset. This kind of changes, involving a change of the topological structure of the state of a DS, arises for instance during the gastrulation or the neurula stage in the early development of an embryo. Such situation has motivated the development of the topological collection framework.

2.3 Examples

We illustrates theses definitions on the previous computational models. We just sketch how these models can be roughly rephrased in the framework of topological collections. The representations given are only approximations of the exact computation processes, because we do not consider seriously the basic computation mechanisms (i.e. the exact nature of the function φ introduced above). These details are very relevant for the study of the expressive power of each formalism but are not considered here, as a programming language always embeds a lot of small extensions required to facilitate the programmer's life.

2.3.1 The Topology of Sets and Multisets

Gamma. A set V is represented by a topological 1-collection with vertices V and one edge $\top$. The function ∂_1 is defined by $\partial_1 \top = \sum V$. With this definition, all elements of V are connected together.

A multiset M of element $e \in E$ can be represented by a set $\hat{M} \subseteq \mathbb{N} \times E$. If $e \in M$ with multiplicity n, then the n elements $(1, e), (2, e), ..., (n, e)$ belong to $\hat{M}$. The multiset M is represented as the 1-collection associated to $\hat{M}$.

With this representation, the application of one Γ rule on a multiset M is a local non-cooperative elementary functional transformation of M into M'.

The connection between two multiset elements accounts the fact that any submultiset can be matched and replaced in a Γ rule.

P systems. The case of P systems is more interesting, because the topology can be used to take into account the locality of a computation step. In this approach, the region associated to a membrane would be a 2-cell and the membranes would be 1-cells; then a P system is viewed as a 2-collection with a 2-complex organization.

A cruder approach just associates a multiset M to the region associated with the skin of a P system. The difference with Γ is that the elements of M can be multiset themselves, associated to the inner membranes. In this approach, P systems are viewed as a theory of nested multiset rewriting. We can handle also this approach, constraining for example the function φ to be an elementary transformation.

2.3.2 The Topology of Sequences

A sequence $v_1, ..., v_n$ of n values can be represented by a 1-collection: $(\mathcal{C}, f, 1)$. $\mathcal{C}$ is a 1-complex (K_p, C_p, ∂_p) with: $K_0 = \{0, 1, .., n\}$ and $K_1 = \{(0, 1), (1, 2), ..., (n-1, n)\}$. We have $\partial_0 x = \emptyset$ and $\partial_1(i, j) = i + j$ (where the sum in the right is the group operation of C_1). And $f = v_1.(0, 1) + ... + v_n.(n-1, n)$.

The application of only one production of a D0L system is the application of a local, shape preserving, elementary functional transformation, each on an independent subcollection. In addition, for one rule application between F and F', the function φ in the transformation depends only of $\triangleright(F, F')$ and not of $\widetilde{\triangleright}(F, F')$ (because the productions are context-free).

2.3.3 The Topology of Arrays

An $n\times$ array is a 2-collection. The K_i of the underlying 2-organization are defined by:

$$
\begin{aligned}
K_0 &= \{(i, j) \mid 1 \leq i \leq n, 1 \leq j \leq m\} \\
K_1 &= \{(x, y) \mid x, y \in K_0 \text{ and } |x - y| \in \{(0, 1),\ (1, 0)\}\} \\
K_2 &= \{((x_1, x_2, x_3, x_4) \mid x_i \in K_0 \text{ and } |x_{\sigma i} - x_i| \in \{(0, 1),\ (1, 0)\}\}
\end{aligned}
$$

where σ is the circular permutation on $(1, 2, 3, 4)$. The boundary maps are defined by $\partial_1(x, y) = x + y$, $\partial_2(x_1, x_2, x_3, x_4) = (x_1, x_2) + (x_2, x_3) + (x_3, x_4) + (x_4, x_1)$.

A rule of a cellular automata is a local organization-preserving elementary functional transformation.

3 Getting Started with `MGS`

`MGS` embeds the idea of topological collections and their transformations into the framework of a simple dynamically typed functional language. Collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax. In our context, dynamically typed means that there is no static type checking and that type errors are detected at run-time. Although dynamically typed, the set of values has a rich type structure used in the definition of transformations.

We give here informally the main constructs concerning collections, transformations and their applications. Elements of the `MGS` syntax are given through examples.

3.1 Records and Collections

In addition to basic values like integers, floats, strings, lambda-expressions, etc., `MGS` handles records and several kind of collections. The operators acting on these values are functional: they combine values to give new values. They do not act by side-effect. Records and collections are (sub-)typed.

Collections. Several kind of topological collections are supported by `MGS`. We focus here on sets, multisets and sequences. There is a large amount of generic operations available for all collection kinds, based on the function algebra developed for polytypic programming: catamorphism (map, fold, ...), hylomorphism, ... Cf. [BDM97, BNTW95]. We do not enter in these features as they are not relevant for our purpose here.

Often there is a need to distinguish several collections of the same kind (e.g. several multisets nested in one other multiset). Various ways can be used to achieve the distinction. For instance, in the P system formalism, each multiset is labeled by a unique integer to reference them unambiguously. We chose to distinguish between collections of the same kind by types. The type of a collection must be thought as a label that does not change the structure of the collection. Types are organized by a subtyping relationship. The subtyping relation organizes types into a poset. The kind of a collection constitutes the maximal elements of this hierarchy. Collection type declarations look like:

$$\begin{aligned} \texttt{collection}\ \mathit{MySet} &= \texttt{set};\\ \texttt{collection}\ \mathit{AnotherSet} &= \texttt{set};\\ \texttt{collection}\ \mathit{AnotherMySet} &= \mathit{MySet}; \end{aligned}$$

These three declarations specifies a hierarchy of three types. Type *AnotherMySet* is a subtype of *MySet* which is a subtype of `set`. The type `set` is predefined and corresponds to a collection kind (other predefined types are `seq` for sequences and `bag` for multisets). The type *AnotherSet* is also a subtype of `set` but is not comparable with *MySet*.

As for record types, a type is also though as a predicate used to test the kind of a value or as a pattern in rule pattern-matching (Cf. section 3.2). It is also used in the building of a collection by enumeration of its elements:

$$1,\ 1+1,\ 2+1,\ 2*2,\ \mathit{MySet} : ()$$

is an expression evaluating in a collection of the four elements 1, 2, 3 and 4. The collection kind is a set, and its type is *MySet*. Actually, expression "$\mathit{Myset} : ()$" denotes the empty *MySet* and "," is an overloaded operator which creates a new collection by merging its arguments.

The type of a collection is taken into account for several collection operations. For instance, the *join* of two collections of type A and B gives a collection with type C corresponding to the common ancestor of A and C (with the previous example, `set` is the common ancestor of *MySet* and *AnotherSet*. Other example, *MySet* is the common ancestor of *AnotherMySet*) and itself.

Records. An `MGS` record is a special kind of collection. An `MGS` record is a map that associates a name, called *field*, to a value. The value can be of any type, including records or other collections. Accessing the value of a field in a record is achieved with the dot notation: the value of the expression $\{a = 1,\ b = \texttt{"red"}\}.b$ is the string `"red"`.

Records can be merged with the overloaded $+$ operator. Expression $r_1 + r_2$ computes a new record r having the fields of both r_1 and r_2. Then $r.a$ has the value of $r_1.a$ if the field a belongs to r, else the value of $r.b$ (asymmetric merge [Rém92]).

Type introduced by a type declarations can latter be used in pattern-matching (Cf. section 3.2) or to test if a value is of a given type. For records, type declarations look like

$$\begin{aligned}
\texttt{state}\ R &= \{a\};\\
\texttt{state}\ S &= \{b,\ \tilde{}c\} + R;\\
\texttt{state}\ T &= S + \{a = 1,\ d : \texttt{string}\};
\end{aligned}$$

(`state` is the keyword used to introduce the definition of a record type in `MGS`). The first declaration specifies a record type R which consists of the records with at least a field named a. A type name can be used as a predicate:

$$R(\{a = 2,\ x = 3\}) \qquad \text{or equivalently} \qquad \{a = 2,\ x = 3\} : R$$

evaluates to true because the record $\{a = 2,\ x = 3\})$ has a field named a. The second declaration defines S which has all the fields of R, a field b and *no* field c. The $+$ operator between record types emulates a kind of (multiple) inheritance. The definition T specializes type S by constraining the field a to the value 1 and saying that a field d must be present and valued by a string.

3.2 Pattern, Rule and Transformations

A transformation T is a set of rules:

$$\texttt{trans}\ T = \{\ \ldots\quad \textit{rule};\quad \ldots\ \}$$

When there is only one rule in the transformation, the enclosing braces can be dropped. A rule is a basic transformation taking the following form:

$$\textit{pattern}\ \ {=}{>}\ \ \textit{expression}$$

where *pattern* in the left hand side (lhs) of the rule matches a subcollection A of the collection C on which the transformation is applied. The subcollection A is substituted in C by the collection B computed by the *expression* in the right hand side (rhs) of the rule. However, the substitution takes places only if the shape of B agrees with the shape of A, as explained in section 2. There are also several kind of rules, as detailed below.

3.2.1 Patterns

We present the pattern expressions that have a generic meaning, that is, they can be interpreted against any collection kind. The grammar of such pattern expressions is the following

$$Pat ::= x \ \mid\ \{\ldots\}\ \mid\ p, p'\ \mid\ p+\ \mid\ p*\ \mid\ p : P\ \mid\ p/exp\ \mid\ p\ \texttt{as}\ x\ \mid\ (p)$$

where p, p' are patterns, x ranges over the pattern variables, P is a predicate and exp is an expression with a boolean value. The explanations below give an informal semantics for these patterns.

variable: a pattern variable x matches exactly one element. The name x can then occurs elsewhere in the rule.

state pattern: $\{\ldots\}$ are used to match one element which is a record. The content of the brackets can be used to match records with or without a specific field (eventually constrained to a given field type or field value). For instance, $\{a,\ b : \texttt{string},\ c = 3,\ \tilde{}d\}$ is a pattern that matches a record with fields a, b and c but no field d. In addition, the type of field b must be "string" and the value of the field c must be the integer 3.

neighbor: p, p' is a pattern that matches two connected collections p and p'. For example, x, y matches two connected elements. The connection relationship is introduced in section 2 and depends of the collection kind,

repetition: pattern $p+$ (resp. $p*$) matches a non empty suite of connected elements (resp. a possibly empty suite).

binding: a binding p *as* x gives the name x to the collection matched by p. This name can be used in the rest of the rule.

guard: p/exp matches the collections matched by p verifying *exp*. Pattern $p : P$ is an abbreviation of $(pasx)/P(x)$ where x is a fresh variable. For instance, $x : MySet$ filters an element which is of type *MySet*. Another example: $y \,/\, y > 3$ matches an element y such that $y > 3$ is true.

Here is a contrived example. Pattern

$$(x : int/x < 3) + \ as\ S \quad / \quad card(S) < 5 \ \ \& \ \ Fold[+](S) > 10$$

selects a connected collection S of integers less than 3, such that the cardinal of S is less than 5 and the sum of the elements in S is greater than 10. If this pattern is used against a sequence, S denotes a subsequence. If the pattern is used against a set or a multiset M, S is a subset or a submultiset of M as explained in section 2.

There are also some pattern constructs that are specific to a collection kind. For example, the construct "$\hat{}\,, x$" is used to select an element which has no left neighbor in a sequence. Such pattern has no meaning when the transformation is applied for instance to a set, and an error is raised. Another example of a specific construct is the operator *left* and *right*. They can be used in the guard of a pattern (or in the rhs of a rule) to refer to the element to the right or to the left of an element x matched in a sequence.

3.2.2 Rules

A transformation is a set of rules. When a transformation is applied to a collection, the strategy is to apply as many rules as possible in parallel. A rule can be applied if its pattern matches a subcollection. Several features are used to have a control over the choice of the rules applied within a transformation.

Exclusive and inclusive rules. *Exclusive rules* consume their argument: that is, a subcollection matched by an exclusive rule cannot intersect a subcollection matched by any other rule.

Inclusive rules don't have this kind of constraint. They are mainly used to transform independent parts of a complex object[3]. This is best explained by an example:

$$\{x \ \mathsf{as}\ v\} \quad += > \quad \{x = v + 1\}$$
$$\{y \ \mathsf{as}\ v\} \quad += > \quad \{y = 2 * v\}$$

are two inclusive rules (because the arrow is $+=>$) matching respectively a record with a field x and a record with a field y. So they can both apply to the record $\{x = 2,\ y = 3\}$. An inclusive rule of form $r += > r'$ where r is a record pattern and r' an expression evaluating to a record, replaces the matched record R by $R + r'$. So, the result of applying the two previous rule to $\{x = 2,\ y = 3\}$ is $\{x = 3,\ y = 6\}$. This result is computed as

$$\big(\{x = 2,\ y = 3\} + \{x = 2 + 1\}\big) + \{y = 2 * 3\}$$

or

$$\big(\{x = 2,\ y = 3\} + \{y = 2 * 3\}\big) + \{x = 2 + 1\}$$

and is independent of the order of application of the two rules. Indeed, the rules work on independent part of the record, both for accessing or updating the value of a field.

[3]Currently, only a rhs matching a record is allowed in an inclusive rule, but the idea must be extended to nested collections. The concept of inclusive rule may appear very specific. However, it is a very effective way to cut the combinatorial explosion of the behavior specifications.

Priority. Exclusive rules are applied before any inclusive rules. A priority can be associated to each rule, to specify a precedence order within each class (the priority of inclusive rules may be used to specify the relative order of their applications).

Local variables and conditional rules. Imperative local variables can be attached to a transformation and updated by side effects in the rhs of the rules. These variables can be used in a rule guard allowing the conditional use of a rule. For instance, the transformation

$$\texttt{trans}\ T[a=0] \quad = \quad x \ \ =\!\{\,\texttt{on}\ a<5\,\}\!=\!> \ \ (a := a+1;\ 2*x);$$

specifies a rule which is applied at most 5 times. The local imperative variable a, initialized to 0 when T is applied, counts the number of rule applications.

3.3 Managing the Applications of a Transformation

A transformation T can be used like a unary function. For instance, a transformation can be passed as an argument to another function. It makes able to sequence and compose transformations very easily.

The expression $T(c)$ denotes the application of one transformation step to the collection c. As said above, a transformation step consists in the parallel application of the rules (modulo the rule application's features). A transformation step can be easily iterated:

$T[n]\,(c)$	denotes the application of n transformation steps to c
$T[\texttt{fixpoint}]\,(c)$	application of the transformation T until a fixpoint is reached
$T[\texttt{fixrule}]\,(c)$	idem but the fixpoint is detected when no rule applies

In addition to the standard transformation step strategy, two other *application modes* exist. In the *stochastic mode*, the choice of the exclusive rule to apply is made randomly. The priorities of the exclusive rules are then considered as the relative probability of their effective application (when they can apply). In *asynchronous mode*, only one exclusive rule is applied in one transformation step.

4 Examples of MGS Programs

The following example are freely inspired by examples given for Γ, P systems and L systems.

Erastothene's Sieve on a Set. The idea is to generates a set with integers from 2 to N and to replace an x and an y such that x divides y by x. The results is the set of the prime integers

$$\begin{aligned}
\texttt{trans}\ G &= \{x, \mathit{true}\} \ => \ x, \{x+1, \mathit{true}\};\\
\texttt{trans}\ S &= \{x, \mathit{true}\} \ => \ x;\\
\texttt{trans}\ E &= (x, y \ /\ y \,\texttt{mod}\, x = 0) \ => \ x;
\end{aligned}$$

With this program, the expression $E[\texttt{fixrule}](S(G[N](\{2, \mathit{true}\})))$ computes the primes up to N.

Sorting a Sequence. Bubble sort is immediate:

$$\texttt{trans}\ \mathit{Sort} \ = \ (x, y \ /\ y < x) \ => \ y, x;$$

Erastothene's Sieve on a Sequence. The idea is to refine the previous algorithm using a sequence. Each element i in the sequence corresponds to the previously computed ith prime P_i and is represented by a record $\{x = P_i\}$. This element can receive a candidate number n, which is represented by a record $\{x = P_i, a = n\}$. If the candidate pas the test, then the element transforms itself to a record $r = \{x = P_i, b = n\}$. If the right neighbor of r is of form $x = P_{i+1}\}$, then the candidate n skip from r to the right neighbor. When there is no right neighbor to r, then n is prime and a new element is added at the end of the sequence. The first element of the sequence is distinguished and generates the candidates.

$$
\begin{array}{l}
\mathtt{trans}\ \mathit{Erasto} = \{ \\
\qquad \begin{array}{rcl}
\mathit{Genere}1 & = & n : \mathit{integer}\ /\ \tilde{}\mathtt{right}\ n\ \ \texttt{=>}\ \ n, x = n; \\
\mathit{Genere}2 & = & n : \mathit{integer}, \{x, \tilde{}a, \tilde{}b\}\ \ \texttt{=>}\ \ n+1, \{x = x, a = n\}; \\
\mathit{test}1 & = & \{x, a, \tilde{}b\}\ /\ a \,\mathtt{mod}\, x = 0\ \ \texttt{=>}\ \ \{x = x\}; \\
\mathit{test}2 & = & \{x, a, \tilde{}b\}\ /\ a \,\mathtt{mod}\, x <> 0\ \ \texttt{=>}\ \ \{x = x, b = a\}; \\
\mathit{Pas} & = & \{x\ \mathtt{as}\ x1, b\}, \{x\ \mathtt{as}\ x2, \tilde{}a, \tilde{}b\}\ \ \texttt{=>}\ \ \{x = x1\}, \{x = x2, a = b\}; \\
\mathit{Create} & = & \{x, b\}\ \mathtt{as}\ s\ /\ \tilde{}\mathtt{right}\ s\ \ \texttt{=>}\ \ \{x = x\}, \{x = b\};
\end{array} \\
\}
\end{array}
$$

In this example we have given an explicit name to each rule. The expression $\mathit{Erasto}[N]((2, \mathtt{seq} : ()))$ executes N steps of the Erastothene's sieve. For instance $\mathit{Erasto}[100]((2, \mathtt{seq} : ()))$ computes the sequence: 42, $\{a = 42,\ x = 2\}$, $\{b = 41,\ x = 3\}$, $\{x = 5\}$, $\{x = 7\}$, $\{x = 11\}$, $\{x = 13\}$, $\{b = 37,\ x = 17\}$, $\{x = 19\}$, $\{x = 23\}$, $\{x = 29\}$, $\{x = 31\}$.

5 Conclusion and Future Work

In the current implementation [GM01b], sets, multisets and sequences of elements are supported. Elements are of any types, allowing arbitrary nesting. Implementation of arrays is in progress and group-based data fields (GBF which generalizes functional arrays, cf. [GMS96, GM01a]) are planed in a short term. We also begin a collaboration to study a generic implementation of topological collections based on G-maps [Lie91]. The efficient compilation of a `MGS` program is a long-term research.

The perspectives opened by this preliminary work are numerous. From the applications point of view, we are targeted by the simulation of the topological changes at the early development of the embryo. Another motivating application is the case of the Golgi formation where some proteins stress the endoplasmic reticulum until a small vesicle is released. These vesicles accumulate to form the Golgi apparatus. This is an actual example of membrane formation and fusion.

At the language level, the study of the topological collections concepts must continue with a finer study of transformation kinds. We have restricted ourselves in several ways. For instance, simple collections are valuated only in one dimension and valuation of all cells must be worked out. Several kind of restrictions can be put on the transformations, leading to various kind of pattern languages and rules. The complexity of matching such patterns has to be investigated. We also want to develop a type system that can handle nested collections, along the lines developed in [Ble93]. At last but not least, we want to known if the topological spaces that can be built by transformations can be characterized through a non standard type system.

Acknowledgments

The authors would like to thanks the members of the "Simulation and Epigenesis" group at Genopole for stimulating discussions and biological motivations. They are also grateful to F. Delaplace and J. Cohen for numerous questions, encouragements and sweet cookies. This research is supported in part by the CNRS, the GDR ALP, IMPG and Genopole/Evry.

References

[BB89] Gerard Berry and G. Boudol. The chemical abstract machine. Technical Report RR-1133, Inria, Institut National de Recherche en Informatique et en Automatique, 1989.

[BCM87] J. P. Banatre, A. Coutant, and Daniel Le Metayer. Parallel machines for multiset transformation and their programming style. Technical Report RR-0759, Inria, 1987.

[BDM97] Richard Bird and Oege De Moor. *Algebra of Programming*, volume 100 of *Int. Series in Computer Science*. Prentice Hall, 1997.

[Ble93] Guy Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[BM86] J. P. Banatre and Daniel Le Metayer. A new computational model and its discipline of programming. Technical Report RR-0566, Inria, 1986.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 18 September 1995.

[FB94] W. Fontana and L. Buss. "the arrival of the fittest": Toward a theory of biological organization. *Bulletin of Mathematical Biology*, 1994.

[FB96] W. Fontana and L. Buss. *Boundaries and Barriers, Casti, J. and Karlqvist, A. edts,*, chapter The barrier of objects: from dynamical systems to bounded organizations, pages 56–116. Addison-Wesley, 1996.

[Fon92] Walter Fontana. Algorithmic chemistry. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Proceedings of the Workshop on Artificial Life (ALIFE '90)*, volume 5 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 159–210, Redwood City, CA, USA, February 1992. Addison-Wesley.

[Gia00] Jean-Louis Giavitto. A framework for the recursive definition of data structures. In *Proceedings of the 2nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, pages 45–55. ACM Press, September 20–23 2000.

[GM01a] J.-L. Giavitto and O. Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, September 2001.

[GM01b] J.-L. Giavitto and O. Michel. MGS: a programming language for the transformations of topological collections. Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001.

[GMS96] J.-L. Giavitto, O. Michel, and J. Sansonnet. Group-based fields. In *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068, pages 209–215, 1996.

[Hen94] M. Henle. *A combinatorial introduction to topology.* Dover publications, New-York, 1994.

[HY88] J. G. Hocking and G.S. Young. *Topology.* Dover publications, New-York, 1988.

[Lie91] P. Lienhardt. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1):59–82, 1991.

[Pau98] Gheorghe Paun. Computing with membranes. Technical Report TUCS-TR-208, TUCS - Turku Centre for Computer Science, November 11 1998.

[Pau00] G. Paun. From cells to computers: Computing with membranes (p systems). In *Workshop on Grammar Systems*, Bad Ischl, austria, July 2000.

[Rém92] Didier Rémy. Syntactic theories and the algebra of record terms. Technical Report 1869, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[VN66] J. Von Neumann. *Theory of Self-Reproducing Automata.* Univ. of Illinois Press, 1966.