Computations in Space and Space in Computations

Jean-Louis GIAVITTO, Olivier MICHEL, Julien COHEN, Antoine SPICHER

LaMI, umr 8042 du CNRS, Université d'Évry – GENOPOLE
Tour Évry-2, 523 Place des Terrasses de l'Agora
91000 Évry, France
[giavitto,michel]@lami.univ-evry.fr

The Analytical Engine weaves algebraic patterns just as the Jacquard loom weaves flowers and leaves.
Ada Lovelace

1 Goals and Motivations

The emergence of terms like *natural computing*, *mimetic computing*, *parallel problem solving from nature*, *bio-inspired computing*, *neurocomputing*, *evolutionary computing*, etc., shows the never ending interest of the computer scientists for the use of "natural phenomena" as "problem solving devices" or more generally, as a fruitful source of inspiration to develop new programming paradigms. It is the latter topic which interests us here. The idea of *numerical experiment* can be reversed and, instead of using computers to simulate a fragment of the real world, the idea is to use (a digital simulation of) the real world to compute. In this perspective, the processes that take place in the real world are the objects of a new calculus:

$$\begin{aligned}\text{description of the world's laws} &= \text{program}\\ \text{state of the world} &= \text{data of the program}\\ \text{parameters of the description} &= \text{inputs of the program}\\ \text{simulation} &= \text{the computation}\end{aligned}$$

This approach can be summarized by the following slogan: "programming *in* the language of nature" and was present since the very beginning of computer science with names like W. Pitts and W. S. McCulloch (formal neurons, 1943), S. C. Kleene (inspired by the previous for the notion of finite state automata, 1951), J. H. Holland (connectionist model, 1956), J. Von Neumann (cellular automata, 1958), F. Rosenblatt (the perceptron, 1958), etc.

This approach offers many advantages from the *teaching*, *heuristic* and *technical* points of view: it is easier to explain concepts referring to real world processes that are actual examples; the analogy with the nature acts as a powerful source of inspirations; and the studies of natural phenomena by the various scientific disciplines (physics, biology, chemistry...) have elaborated a large body of concepts and tools that can be used to study computations (some concrete examples of this cross fertilization relying on the concept of dynamical system are given in references [6, 5, 34, 12]).

There is a *possible fallacy* in this perspective: the description of the nature is not unique and diverse concurent approaches have been developed to account for the same objects. Therefore, there is not a unique "language of nature" prescribing a unique and definitive programming paradigm. *However*, there is a common concern shared by the various descriptions of nature provided by the scientific disciplines: *natural phenomena take place in time and space.*

In this paper, we propose the use of spatial notions as structuring relationships *in* a programming language. Considering space in a computation is hardly new: the use of spatial (and temporal) notions is at the basis of computational complexity *of* a program; spatial and temporal relationships are also used in the implementation of parallel languages (if two computations occur at the same time, then the two computations must be located at two different places, which is the basic constraint that drives the scheduling and the data distribution problems in parallel programming); the methods for building domains in denotational semantics have also clearly topological roots, but they involve the *topology of the set of values*, not the *topology of a value.* In summary, spatial notions have been so far mainly used to describe the running of a program and not as *means to design new programs.*

We want to stress this last point of view: we are not concerned by the organization of the resources used by a program run. What we want is to develop a spatial point of view on the entities built by the programmer when he designs his programs. From this perspective, a program must be seen as a space where computation occurs and a computation can be structured by spatial relationships. We hope to provide some evidences in the rest of this paper that the concept of space can be as fertile as mathematical logic for the development of programming languages. More specifically, we advocate that the concepts and tools developed for the algebraic construction and characterization of shapes[1] provide interesting teaching, heuristic and technical alternatives to develop new data structures and new control structures for programming.

The rest of this paper is organized as follows. Section 2 and section 3 provide an informal discussion to convince the reader of the interest of introducing a topological point of view in programming. This approach is illustrated through the experimental programming language MGS used as a vehicle to investigate and validate the topological approach.

[1] G. Gaston-Granger in [23] considers three avenues in the formalization of the concept of space: *shape* (the algebraic construction and the transformation of space and spatial configurations), *texture* (the continuum) and *measure* (the process of counting and coordinatization [39]). In this work, we rely on elementary concepts developed in the field of combinatorial algebraic topology for the construction of spaces [24].

Section 2 introduces the idea of seeing a data structure as a space where the computation and the values move. Section 3 follows the spatial metaphor and presents control structures as path specifications. The previous ideas underlie MGS. Section 4 sketches this language. The presentation is restricted to the notions needed to follow the examples in the next section. Section 5 gives some examples and introduces the $(\mathrm{DS})^2$ class of dynamical systems which exhibit a dynamical structure. Such kind of systems are hard to model and simulate because the state space must be computed jointly with the running state of the system. To conclude in section 6 we indicate some of the related work and we mention briefly some perspectives on the use of spatial notions.

2 Data Structures as Spaces[2]

The relative accessibility from one element to another is a key point considered in a data structure definition:

– In a simply linked list, the elements are accessed linearly (the second after the first, the third after the second, etc.).
– In a circular buffer, or in a double-linked list, the computation goes from one element to the following *or* to the previous one.
– From a node in a tree, we can access the sons.
– The neighbors of a vertex V in a graph are visited after V when traveling through the graph.
– In a record, the various fields are locally related and this localization can be named by an identifier.
– Neighborhood relationships between array elements are left implicit in the array data-structure. Implementing neighborhood on arrays relies on an index algebra: index computations are used to code the access to a neighbor. The standard example of index algebra is integer tuples with linear mappings $\lambda x.x \pm 1$ along each dimension (called "Von Neumann" or "Moore" neighborhoods).

This accessibility relation defines a logical neighborhood. The concept of logical neighborhood in a data structure is not only an abstraction perceived by the programmer and vanishing at the execution, but it does have an actual meaning for the computation. Very often the computation indeed complies with the logical neighborhood of the data elements and it is folk's knowledge that most of the algorithms are structured either following the structure of the input data or the structure of the output data. Let us give some examples.

The recursive definition of the `fold` function on lists propagates an action to be performed along the traversal of a list. More generally, recursive computations on data structures respect so often the logical neighborhood, that standard high-order functions (e.g. *primitive recursion*) can be automatically defined from the data structure organization (think about catamorphisms and other polytypic functions on inductive types [29, 26]).

[2] The ideas exposed in this section are developed in [19, 14].

The list of examples can be continued to convince ourselves that a notion of logical neighborhood is fundamental in the definition of a data structure. So to define a data organization, we adopt a *topological* point of view: *a data structure can be seen as a space*, the set of positions between which *the computation moves. Each position possibly holds a value*[3]. The set of positions is called the *container* and the values labeling the positions constitute the *content*.

This topological approach is constructive: one can define a data type by the set of moves allowed in the data structure. An example is given by the notion of "Group Based Fields" or GBF in short [21, 16]. In a uniform data structure, i.e. in a data structure where any elementary move can be used against any position, the set of moves possesses the structure of a mathematical group $\mathcal{G}$. The neighborhood relationship of the container corresponds to the Cayley graph of $\mathcal{G}$. In this paper, we will use only two very simple groups $\mathcal{G}$ corresponding to the moves `|north>` and `|east>` allowed in the usual two-dimensional grid and to the moves allowed in the hexagonal lattice figured at the right of Fig. 3.

3 Control Structures as Paths

In the previous section, we suggested looking at data structure as spaces in which computation moves. Then, when the computation proceeds, a path in the data structure is traversed. This path is driven by the control structures of the program. So, a control structure can be seen as a path specification in the space of a data structure. We elaborate on this idea into two directions: concurrent processes and multi-agent systems.

3.1 Homotopy of a Program Run

Consider two sequential processes `A` and `B` that share a semaphore s. The current state of the parallel execution `P = A` $||$ `B` can be figured as a point in the plane $A \times B$ where A (resp. B) is the sequence of instructions of `A` (resp. `B`). Thus, the running of `P` corresponds to a path in the plane $A \times B$. However, there are two constraints on paths that represent the execution of `P`. Such a path must be "increasing" because we suppose that at least one of the two subprocesses `A` or `B` must progress. The second constraint is that the two subprocesses cannot be simultaneously in the region protected by the semaphore s. This constraint has a clear geometrical interpretation: the increasing paths must avoid an "obstruction region", see Fig. 1. Such representation is known at least from the 1970's as "progress graph" [7] and is used to study the possible deadlocks of a set of concurrent processes.

Homotopy (the continuous deformation of a path) can be adapted to take into account the constraint of increasing paths and provides effective tools to detect deadlocks or to classify the behavior of a parallel program (for instance

[3] *A point in space is a placeholder awaiting for an argument*, L. Wittgenstein, (Tractatus Logico Philosophicus, 2.0131).

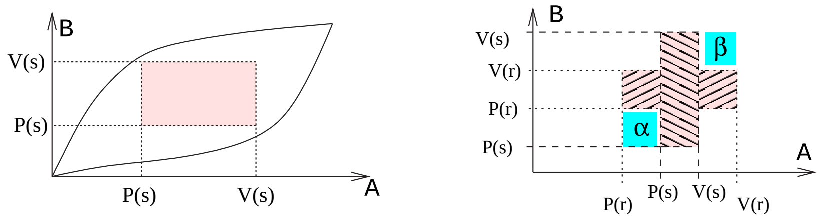

Fig. 1. *Left:* The possible path taken by the process A || B is constrained by the obstruction resulting of a semaphore shared between the processes A and B. *Right:* The sharing of two semaphores between two processes may lead to deadlock (corresponding to the domain α) or to the existence of a "garden of Eden" (the domain β cannot be accessed from outside β and can only be leaved.)

in the previous example, there are two classes of paths corresponding to executions where the process A or B enters the semaphore first). Refer to [22] for an introduction to this domain.

3.2 The Topological Structure of Interactions[4]

In a multi-agent system (or an object based or an actor system), the control structures are less explicit and the emphasis is put on the local interaction between two (sometimes more) agents. In this section, we want to show that the interactions between the elements of a system exhibit a natural topology.

The starting point is the decomposition of a system into subsystems defined by the requirement that the elements into the subsystems interact together and are truly independent from all other subsystems parallel evolution.

In this view, the decomposition of a system S into subsystems $S_1, S_2, \ldots, S_n$ is *functional*: state $s_i(t+1)$ of the subsystem S_i depends solely of the previous state $s_i(t)$. However, the decomposition of S into the S_i can depend on the time steps. So we write $S^t = \{S_1^t, S_2^t, \ldots, S_{n_t}^t\}$ for the decomposition of the system S at time t and we have: $s_i(t+1) = h_i^t(s_i(t))$ where the h_i^t are the "local" evolution functions of the S_i^t. The "global" state $s(t)$ of the system S can be recovered from the "local" states of the subsystems: there is a function φ^t such that $s(t) = \varphi^t(s_1(t), \ldots, s_{n_t}(t))$ which induces a relation between the "global" evolution function h and the local evolution functions: $s(t+1) = h(s(t)) = \varphi^t(h_1^t(s_1(t)), \ldots, h_{n_t}^t(s_{n_t}(t)))$.

The successive decomposition $S_1^t, S_2^t, \ldots, S_{n_t}^t$ can be used to capture the *elementary parts* and the *interaction structure* between these elementary parts of S. Cf. Figure 2. Two subsystems S' and S'' of S interact if there are some t such that $S', S'' \in S^t$. Two subsystems S' and S'' are *separable* if there are some t such that $S' \in S^t$ and $S'' \notin S^t$ or vice-versa. This leads to consider the set $\mathcal{S}$, called the *interaction structure* of S, defined by the smaller set closed by intersection that contains the S_j^t.

[4] This section is adapted from [36].

Set $\mathcal{S}$ has a *topological structure*: $\mathcal{S}$ corresponds to an *abstract simplicial complex*. An abstract simplicial complex [24] is a collection $\mathcal{S}$ of finite nonempty set such that if A is an element of $\mathcal{S}$, so is every nonempty subset of A. The element A of $\mathcal{S}$ is called a *simplex* of $\mathcal{S}$; its *dimension* is one less that the number of its elements. The dimension of $\mathcal{S}$ is the largest dimension of one of its simplices. Each nonempty subset of A is called a *face* and the *vertex set* $V(\mathcal{S})$, defined by the union of the one point elements of $\mathcal{S}$, corresponds to the *elementary functional parts* of the system S. The abstract simplicial complex notion generalizes the idea of *graph*: a simplex of dimension 1 is an edge that links two vertices, a simplex f of dimension 2 can be thought of as a surface whose boundaries are the simplices of dimension 1 included in f, etc.

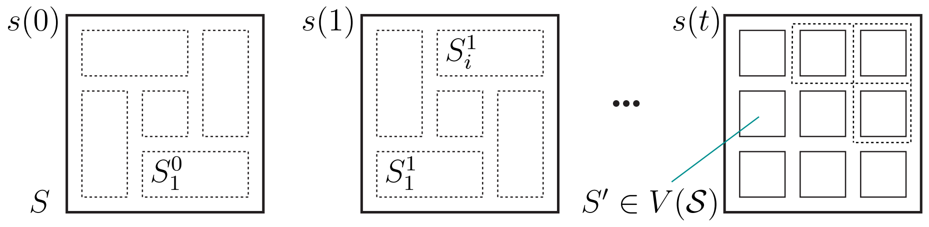

Fig. 2. The interaction structure of a system S resulting from the subsystems of elements in interaction at a given time step.

4 MGS Principles

The two previous sections give several examples to convince the reader that a topological approach of the data and control structures of a program present some interesting perspectives for language design: a data structure can be defined as a space (and there are many ways to build spaces) and a control structure is a path specification (and there are many ways to specify a path).

Such a topological approach is at the core of the MGS project. Starting from the analysis of the interaction structure in the previous section, our idea is to define directly the set $\mathcal{S}$ with its topological structure and to specify the evolution function h by specifying the set S_i^t and the functions h_i^t:

– the interaction structure $\mathcal{S}$ is defined as a new kind of data structures called *topological collections*;
– a set of functions h_i^t together with the specification of the S_i^t for a given t are called a *transformation*.

We will show that this abstract approach enables an homogeneous and uniform handling of several computational models including cellular automata (CA), lattice gas automata, abstract chemistry, Lindenmayer systems, Paun systems and several other abstract reduction systems.

These ideas are validated by the development of a language also called MGS. This language embeds a complete, strict, impure, dynamically or statically typed functional language.

4.1 Topological Collections

The distinctive feature of the MGS language is its handling of entities structured by *abstract topologies* using *transformations* [20]. A set of entities organized by an abstract topology is called a *topological collection*. Here, topological means that each collection type defines a neighborhood relation inducing a notion of *subcollection*. A subcollection S' of a collection S is a subset of connected elements of S and inheriting its organization from S. Beware that by "neighborhood relation" we simply mean a relationship that specify if two elements are neighbors. From this relation, a cellular complex can be built and the classical "neighborhood structure" in terms of open and closed sets can be recovered [35].

A topological collection can be thought as a function with a finite support from a set of positions (the elements of $V(\mathcal{S})$) to a set of values (the support of a function is the set of elements on which the function takes a well defined value). Such a data structure is called a *data field* [13]. This point of view is only an abstraction: the data structure is not really implemented as a function. This approach makes a distinction between the content and the container. The notions of *shape* [25] and *shape type* [11] also separate the set of positions of a data structure from the values it contains. Often there is no need to distinguish between the positions and their associated values. In this case, we use the term "element of the collection".

Collection Types. Different predefined and user-defined collection types are available in MGS, including sets, bags (or multisets), sequences, Cayley graphs of Abelian groups (which include several unbounded, circular and twisted grids), Delaunay triangulations, arbitrary graphs, quasi-manifolds [36] and some other arbitrary topologies specified by the programmer.

Building Topological Collections. For any collection type `T`, the corresponding empty collection is written `():T`. The join of two collections C_1 and C_2 (written by a comma: C_1`,`C_2) is the main operation on collections. The comma operator is overloaded in MGS and can be used to build any collection (the type of the arguments disambiguates the collection built). So, the expression `1, 1+2, 2+1, ():set` builds the set with the two elements `1` and `3`, while the expression `1, 1+2, 2+1, ():bag` computes a bag (a set that allows multiple occurrences of the same value) with the three elements `1`, `3` and `3`. A set or a bag is provided with the following topology: in a set or a bag, any two elements are neighbors. To spare the notations, the empty sequence can be omitted in the definition of a sequence: `1, 2, 3` is equivalent to `1, 2, 3,():seq`.

4.2 Transformations

The MGS experimental programming language implements the idea of transformations of topological collections into the framework of a functional language: collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules. Transformations (like functions) are first-class values and can be passed as arguments or returned as the result of an application.

The *global transformation* of a topological collection s consists in the *parallel application* of a set of *local transformations*. A local transformation is specified by a rule r that specifies the replacement of a subcollection by another one. The application of a rewriting rule $\sigma \Rightarrow f(\sigma, ...)$ to a collection s:

1. selects a subcollection s_i of s whose elements match the *pattern* σ,
2. computes a new collection s'_i as a function f of s_i and its neighbors,
3. and specifies the insertion of s'_i in place of s_i into s.

One should pay attention to the fact that, due to the parallel application strategy of rules, *all distinct instances s_i of the subcollections matched by the σ pattern are "simultaneously replaced" by the $f(s_i)$.*

Path Pattern. A pattern σ in the left hand side of a rule specifies a subcollection where an interaction occurs. A subcollection of interacting elements can have an arbitrary shape, making it very difficult to specify. Thus, it is more convenient (and not so restrictive) to enumerate sequentially the elements of the subcollection. Such enumeration will be called a *path*.

A path pattern *Pat* is a sequence or a repetition *Rep* of *basic filters*. A basic filter *BF* matches one element. The following fragment of the grammar of path patterns reflects this decomposition:

$$\mathit{Pat} ::= \mathit{Rep} \mid \mathit{Rep}\ ,\ \mathit{Pat} \qquad \mathit{Rep} ::= \mathit{BF} \mid \mathit{BF}/\mathit{exp} \qquad \mathit{BF} ::= \texttt{cte} \mid \mathrm{id} \mid \texttt{<undef>}$$

where `cte` is a literal value, id ranges over the pattern variables and *exp* is a boolean expression. The following explanations give an interpretation for these patterns:

literal: a literal value `cte` matches an element with the same value.

empty element the symbol `<undef>` matches an element whose position does not have an associated value.

variable: a pattern variable a matches exactly one element with a well defined value. The variable a can then occur elsewhere in the rest of pattern or in the r.h.s. of the rule and denotes the value of the matched element.

neighbor: b, p is a pattern that matches a path which begins by an element matched by b and continues by a path matched by p, the first element of p being a neighbor of b.

guard: p/exp matches a path matched by p when the boolean expression exp evaluates to `true`.

Elements matched by basic filters in a rule are distinct. So a matched path is without self-intersection. The identifier of a pattern variable can be used only once as a basic filter. That is, the path pattern x,x is forbidden. However, this pattern can be rewritten for instance as: x,y / y=x.

Right Hand Side of a Rule. The right hand side of a rule specifies a collection that replaces the subcollection matched by the pattern in the left hand side. There is an alternative point of view: because the pattern defines a sequence of elements, the right hand side may be an expression that evaluates to a sequence of elements. Then, the substitution is done element-wise: element i in the matched path is replaced by the element i in the r.h.s. This point of view enables a very concise writing of the rules.

A Very Simple Transformation. The *map* function which applies a function to each element of a collection is an example of a simple transformation:

```
    trans map[f=\z.z] = {   x => f(x)   }
```

This transformation is made of only one rule. The syntax must be obvious (the default value of the optional parameter `f` is the identity written using a lambda-notation). This transformation implements a *map* since each element e of the collection is matched by the pattern x and will be replaced by f(e) in a parallel application strategy of the rule.

5 Examples

5.1 The modeling of Dynamical Systems

In this section, we show through one example the ability of MGS to concisely and easily express the state of a dynamical system and its evolution function. More examples can be found on the MGS web page and include: cellular automata-like examples (game of life, snowflake formation, lattice gas automata...), various resolutions of partial differential equations (like the diffusion-reaction *à la* Turing), Lindenmayer systems (e.g. the modeling of the heterocysts differentiation during Anabaena growth), the modeling of a spatially distributed signaling pathway, the flocking of birds, the modeling of a tumor growth, the growth of a meristem, the simulation of colonies of ants foraging for food, etc.

The example given below is an example of a discrete “classical dynamical system”. We term it “classical” because it exhibits a *static structure*: the state of the system is statically described and does not change with the time. This situation is simple and arises often in elementary physics. For example, a falling stone is statically described by a position and a velocity and this set of variables does not change (even if the value of the position and the value of the velocity change in the course of time). However, in some systems, it is not only the values of state variables, but also the *set* of state variables *and/or* the evolution function, that changes over time. We call these systems *dynamical systems with a dynamic structure* following [17], or (DS)^2 in short. As pointed out by [15], many biological systems are of this kind. The rationale and the use of MGS in the simulation of (DS)^2 is presented in [14, 15].

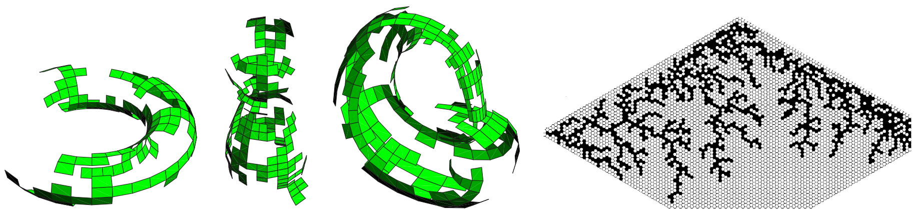

Fig. 3. From left to right: the final state of a DLA process on a torus, a chess pawn, a Klein's bottle and an hexagonal meshes. The chess pawn is homeomorphic to a sphere and the Klein's bottle does not admit a concretization in Euclidean space. These two topological collections are values of the *quasi-manifold* type. Such collection are build using *G-map*, a data-structure widely used in geometric modeling [27]. The torus and the hexagonal mesh are GBFs.

Diffusion Limited Aggregation (DLA). DLA, is a fractal growth model studied by T.A. Witten and L.M. Sander, in the eighties. The principle of the model is simple: a set of particles diffuses randomly on a given spatial domain. Initially one particle, the seed, is fixed. When a mobile particle collides a fixed one, they stick together and stay fixed. For the sake of simplicity, we suppose that they stick together forever and that there is no aggregate formation between two mobile particles. This process leads to a simple CA with an asynchronous update function or a lattice gas automata with a slightly more elaborate rule set. This section shows that the MGS approach enables the specification of a simple generic transformation that can act on arbitrary complex topologies.

The transformation describing the DLA behavior is really simple. We use two symbolic values `‘free` and `‘fixed` to represent respectively a mobile and a fixed particle. There are two rules in the transformation:

1. the first rule specifies that if a diffusing particle is the neighbor of a fixed seed, then it becomes fixed (at the current position);
2. the second one specifies the random diffusion process: if a mobile particle is neighbor of an empty place (position), then it may leave its current position to occupy the empty neighbor (and its current position is made empty).

Note that the order of the rules is important because the first has priority over the second one. Thus, we have :

```
trans dla = {
    ‘free, ‘fixed   =>  ‘fixed, ‘fixed
    ‘free, <undef>  =>  <undef>, ‘free
}
```

This transformation is polytypic and can be applied to any kind of collection, see Fig. 3 for a few results.

5.2 Programming in the Small: Algorithmic Examples

The previous section advocates the adequation of the MGS programming style to model and simulate various dynamical systems. However, it appears that the MGS programming style is also well fitted for the implementation of algorithmic tasks. In this section, we show some examples that support this assertion. More examples can be found on the MGS web page and include: the analysis of the Needham-Schroeder public-key protocol [30], the Eratosthene's sieve, the normalization of boolean formulas, the computation of various algorithms on graphs like the computation of the shortest distance between two nodes or the maximal flow, etc.

Gamma and the Chemical Computing Metaphor. In MGS, the topology of a multiset is the topology of a complete connected graph: each element is the neighbor of any other element. With this topology, transformations can be used to easily emulate a Gamma transformations [2, 3]. The Gamma transformation:

```
M = do
    rp $x_1, \dots, x_n$
    if $P(x_1, \dots, x_n)$
    by $f_1(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n)$
```

is simply translated into the following MGS transformation:

```
trans $M$ = {
    $x_1, \dots, x_n$
    / $P(x_1, \dots, x_n)$
    => $f_1(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n)$ }
```

and the application `M(b)` of a Gamma transformation `M` to a multiset `b` is replaced in MGS by the computation of the fixpoint iteration `M[iter='fixpoint](b)`. The optional parameter `iter` is a system parameter that allows the programmer to choose amongst several predefined application strategies:

f`[iter='fixpoint]`(x_0)

computes $x_1 = f(x_0), x_2 = f(x_1), ..., x_n = f(x_{n-1})$ and returns x_n such that $x_n = x_{n-1}$.

As a consequence, the concise and elegant programming style of Gamma is enabled in MGS: refer to the Gamma literature for numerous examples of algorithms, from knapsack to the maximal convex hull of a set of points, through the computation of prime numbers. See also the numerous applications of multiset rewriting developped in the projects Elan [38] and Maude [37].

One can see MGS as "Gamma with more structure". However, one can note that the topology of a multiset is "universal" in the following sense: it embeds any other neighborhood relationship. So, it is always possible to code (at the price of explicit coding the topological relation into some value inspected at run-time) any specific topology on top of the multiset topology. We interpret the development of "structured Gamma" [10] from this perspective. In addition, transformations are functions and functions are first citizen values in MGS. So the higher-order features of the higher-order chemical programming style (see the article by Banâtre et *al.* in this volume) can be easely achieved in MGS.

Two Sorting Algorithms. A kind of bubble-sort is straightforward in MGS; it is sufficient to specify the exchange of two non-ordered adjacent elements in a sequence, see Fig. 4. The corresponding transformation is defined as:

```
trans BubbleSort = {   x,y / x>y  ⇒  y,x   }
```

The transformation *BubbleSort* must be iterated until a fixpoint is reached. This is not a real a bubble sort algorithm because swapping of elements happen at arbitrary places; hence an out-of-order element does not necessarily bubble to the top in the characteristic way.

Bead sort is a new sorting algorithm [1]. The idea is to represent positive integers by a set of beads, like those used in an abacus. Beads are attached to vertical rods and appear to be suspended in the air just before sliding down (a number is read horizontally, as a row). After their falls, the rows of numbers have been rearranged such as the smaller numbers appears on top of greater numbers, see Fig. 4. The corresponding one-line MGS program is given by the transformation:

```
trans BeadSort = { 'empty |north> 'bead  ⇒  'bead, 'empty }
```

This transformation is applied on the usual grid. The constant `'empty` is used to give a value to an empty place and the constant `'bead` is used to represent an occupied cell. The l.h.s. of the only rule of the transformation *BeadSort* selects the paths of length two, composed by an occupied cell at north of an empty cell. Such a path is replaced by a path computed in the r.h.s. of the rule. The r.h.s. in this example computes a path of length two with the occupied and the empty cell swapped.

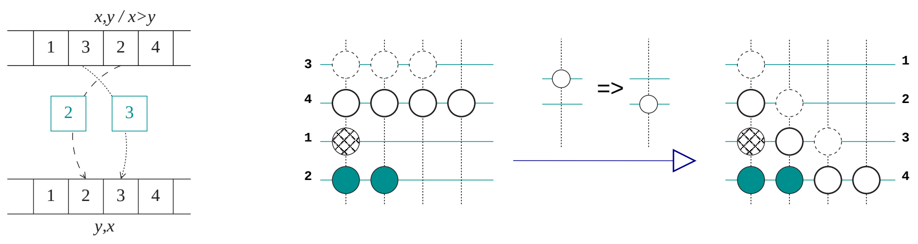

Fig. 4. *Left:* Bubble sort. *Right:* Bead sort [1].

Hamiltonian Path. A graph is a MGS topological collection. It is very easy to list all the Hamiltonian paths in a graph using the transformation:

```
trans H = {
    x* / size(x) = size(self) / Print(x) / false => !(false)
}
```

This transformation uses an iterated pattern x`*` that matches a path (a sequence of elements neighbor two by two). The keyword `self` refers to the collection on which the transformation is applied, that is, the entire graph. The size of a graph returns the number of its vertices. So, if the length of the path x is the same as the number of vertices in the graph, then the path x is an Hamiltonian path because matched paths are simple (no repetition of an element). The second guard prints the Hamiltonian path as a side effect and returns its argument which is not a false value. Then the third guard is checked and returns false, thus, the r.h.s. of the rule is never triggered (the `!` operator introduces an assertion and `!(false)` raises an exception that stops the evaluation process if it is evaluated). The matching strategy ensures a maximal rule application. In other words, if a rule is not triggered, then there is no instance of a possible path that fulfills the pattern. This property implies that the previous rule must be checked on all possible Hamiltonian paths and H`(g)` prints all the Hamiltonian path in `g` before returning `g` unchanged.

6 Current Status and Related Work

The topological approach we have sketched here is part of a long term research effort [21] developed for instance in [13] where the focus is on the substructure, or in [16] where a general tool for uniform neighborhood definition is developed. Within this long term research project, MGS is an experimental language used to investigate the idea of associating computations to paths through rules. The application of such rules can be seen as a kind of rewriting process on a collection of objects organized by a topological relationship (the neighborhood). A privileged application domain for MGS is the modeling and simulation of dynamical systems that exhibit a dynamic structure.

Multiset transformation is reminiscent of multiset-rewriting (or rewriting of terms modulo AC). This is the main computational device of Gamma [2], a language based on a chemical metaphor; the data are considered as a multiset M of molecules and the computation is a succession of chemical reactions according to a particular rule. The CHemical Abstract Machine (CHAM) extends these ideas with a focus on the expression of semantic of non deterministic processes [4]. The CHAM introduces a mechanism to isolate some parts of the chemical solution. This idea has been seriously taken into account in the notion of P systems. P systems [31] are a recent 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 Gamma, the computation is finished when no object can further evolve. By using nested multisets, MGS is able to emulate more or less the notion of P systems. In addition, patterns like the iteration `+` go beyond what is possible to specify in the l.h.s. of a Gamma rule.

Lindenmayer systems [28] have long been used in the modeling of $(\mathrm{DS})^2$ (especially in the modeling of plant growing). They loosely correspond to transformations on sequences or string rewriting (they also correspond to tree rewriting, because some standard features make particularly simple to code arbitrary trees, Cf. the work of P. Prusinkiewicz [32]). Obviously, L systems are dedicated to the handling of linear and tree-like structures.

There are strong links between GBF and cellular automata (CA), especially considering the work of Z. Róka which has studied CA on Cayley graphs [33]. However, our own work focuses on the construction of Cayley graphs as the shape of a data structure and we develop an operator algebra and rewriting notions on this new data type. This is not in the line of Z. Róka which focuses on synchronization problems and establishes complexity results in the framework of CA.

A unifying theoretical framework can be developed [18, 20], based on the notion of *chain complex* developed in algebraic combinatorial topology. However, we do not claim that we have achieved a useful theoretical framework encompassing the previous paradigms. We advocate that few topological notions and a single syntax can be consistently used to allow the merging of these formalisms *for programming* purposes.

The current MGS interpreter is freely available at the MGS home page: `mgs.lami.univ-evry.fr`. A compiler is under development where a static type discipline can be enforced [8, 9]). There are two versions of the type inference systems for MGS: the first one is a classical extension of the Hindley-Milner type inference system that handles homogeneous collections. The second one is a soft type system able to handle heterogeneous collection (e.g. a sequence containing both integers and booleans is heterogeneous).

Acknowledgments

The authors would like to thanks Franck Delaplace at LaMI, Frédéric Gruau at University of Paris-Sud, Florent Jacquemard at INRIA/LSV-Cachan, C. Godin and P. Barbier de Reuille at CIRAD-Montpellier, Pierre-Etienne Moreau at Loria-Nancy, Éric Goubault at CEA-Saclay, P. Prusinkiewicz at the University of Calgary (who coined the term "computation in space") and the members of the Epigenomic group at GENOPOLE-Évry, for stimulating discussions, thoughtful remarks and warm support. We gratefully acknowledge the financial support of the CNRS, the GDR ALP, IMPBIO, the University of Évry and GENOPOLE.

References

1. J. Arulanandham, C. Calude, and M. Dinneen. Bead-sort: A natural sorting algorithm. *Bulletin of the European Association for Theoretical Computer Science*, 76:153–162, Feb. 2002. Technical Contributions.
2. J.-P. Banatre, A. Coutant, and D. L. Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.

3. J.-P. Banâtre, P. Fradet, and D. L. Métayer. Gamma and the chemical reaction model: Fifteen years after. *Lecture Notes in Computer Science*, 2235:17–44, 2001.
4. G. Berry and G. Boudol. The chemical abstract machine. In *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*, pages 81–94. ACM Press, New York, 1990.
5. R. W. Brockett. Dynamical systems that sort lists, diagonalize matrices, and solve linear programming problems. *Linear Algebra and its Applications*, 146:79–91, 1991.
6. K. M. Chandy. Reasoning about continuous systems. *Science of Computer Programming*, 14(2–3):117–132, Oct. 1990.
7. E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, 1971.
8. J. Cohen. Typing rule-based transformations over topological collections. In J.-L. Giavitto and P.-E. Moreau, editors, *4th International Workshop on Rule-Based Programming (RULE'03)*, pages 50–66, 2003.
9. J. Cohen. Typage fort et typage souple des collections topologiques et des transformations. In V. Ménissier-Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA 2004)*, pages 37–54. INRIA, 2004.
10. P. Fradet and D. L. Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, July 1998.
11. P. Fradet and D. L. Mtayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.
12. F. Geurts. Hierarchy of discrete-time dynamical systems, a survey. *Bulletin of the European Association for Theoretical Computer Science*, 57:230–251, Oct. 1995. Surveys and Tutorials.
13. J.-L. Giavitto. A framework for the recursive definition of data structures. In *ACM-Sigplan 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 45–55, Montral, Sept. 2000. ACM-press.
14. J.-L. Giavitto. Invited talk: Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *Rewriting Technics and Applications (RTA'03)*, volume LNCS 2706 of *LNCS*, pages 208 – 233, Valencia, June 2003. Springer.
15. J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz. *Modelling and Simulation of biological processes in the context of genomics*, chapter "Computational Models for Integrative and Developmental Biology". Hermes, July 2002. Also republished as an high-level course in the proceedings of the Dieppe spring school on "Modelling and simulation of biological processes in the context of genomics", 12-17 may 2003, Dieppes, France.
16. J.-L. Giavitto and O. Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, Sept. 2001.
17. J.-L. Giavitto and O. Michel. Mgs: a rule-based programming language for complex objects and collections. In M. van den Brand and R. Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science, 2001.
18. 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.
19. J.-L. Giavitto and O. Michel. Data structure as topological spaces. In *Proceedings of the 3nd International Conference on Unconventional Models of Computation*

UMC02, volume 2509, pages 137–150, Himeji, Japan, Oct. 2002. Lecture Notes in Computer Science.

20. J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.
21. J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. J. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *LNCS*, pages 209–215, Beaune (France), 2–4 Oct. 1995. Springer-Verlag.
22. E. Goubault. Geometry and concurrency: A user's guide. *Mathematical Structures in Computer Science*, 10:411–425, 2000.
23. G.-G. Granger. *La pense de l'espace.* Odile Jacob, 1999.
24. M. Henle. *A combinatorial introduction to topology.* Dover publications, 1994.
25. C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.
26. J. Jeuring and P. Jansson. Polytypic programming. *Lecture Notes in Computer Science*, 1129:68–114, 1996.
27. P. Lienhardt. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1):59–82, 1991.
28. A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.
29. E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Cambridge, MA, August 26–30, 1991. Springer.
30. O. Michel and F. Jacquemard. *An Analysis of a Public-Key Protocol with Membranes*, pages 281–300. Natural Computing Series. Springer Verlag, 2005.
31. G. Paun. From cells to computers: Computing with membranes (P systems). *Biosystems*, 59(3):139–158, March 2001.
32. P. Prusinkiewicz and J. Hanan. L systems: from formalism to programming languages. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 193–211. Springer Verlag, Feb. 1992.
33. Z. Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 Sept. 1994.
34. M. Sintzoff. Invariance and contraction by infinite iterations of relations. In *Research directions in high-level programming languages, LNCS*, volume 574, pages 349–373, Mont Saint-Michel, France, june 1991. Springer-Verlag.
35. R. D. Sorkin. A finitary substitute for continuous topology. *Int. J. Theor. Phys.*, 30:923–948, 1991.
36. A. Spicher, O. Michel, and J.-L. Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)*, volume 3305 of *LNCS*, Amsterdam, October 2004. Springer.
37. The MAUDE project. Maude home page, 2002. `http://maude.csl.sri.com/`.
38. The PROTHEO project. Elan home page, 2002. `http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/`.
39. H. Weyl. *The Classical Groups (their invariants and representations).* Princeton University Press, 1939. Reprint edition (October 13, 1997). ISBN 0691057567.