LaMI

Laboratoire de Méthodes Informatiques

Computation in Space & Space in Computation

Jean-Louis Giavitto, Oliviert Michel,
Julien Cohen, Antoine Spicher

email(s) : `[giavitto, michel, acohen, aspicher] @lami.univ-evry.fr`

Rapport de Recherche n^o **103-2004**

Août 2004

CNRS – Université d'Evry Val d'Essonne
523, Place des Terrasses de l'agora
F–91000 Evry France

Computation in Space and **Space in Computation**

Jean-Louis Giavitto, Olivier Michel, Julien Cohen, Antoine Spicher

LaMI[*]– CNRS – Université d'Évry – Genopole

(*draft paper*)

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:

description of the world's laws	=	program
state of the world	=	data of the program
parameters of the description	=	inputs of the program
simulation	=	the computation

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 based on the concept of dynamical system are given in references [6, 8, 7, 49, 17]).

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 (e.g. the two examples given in Figure 3). 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*[1].

[*]LaMI umr 8042 CNRS – Université d'Évry, Tour Évry-2, 523 place des terrasses de l'agora, 91000 Évry, France. Emails: `[michel, giavitto, jcohen, aspicher]@lami.univ-evry.fr`

[1]We cannot refrain us to cite Kant's famous sentences on the ontological preeminence of the intuition of space over all other object perceptions [34, Transcendental Aesthetic, sect. 1, A.24, B.39]: *Space is a necessary a priori representation, which underlies all outer intuitions. We can never represent to ourselves the absence of space, though we can quite well think it as empty of objects. It must therefore be regarded as the condition of the possibility of appearances, and not as a*

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 develop 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[2] 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. The other sections are more technical and focus on the experimental programming language MGS used as a vehicle to investigate and validate the topological approach.

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 sections. Section 5.1 and section 5.2 give some examples in the field of dynamical system simulation. We introduce the (DS)^2 class of dynamical systems which exhibit a dynamical structure. Such a kind of systems are hard to model and simulate because the state space must be computed jointly with the running state of the system. It appears that the notions developed for the simulation of dynamical systems with a dynamical structure enable a new programming style and section 6 gives some examples of the expressive use of MGS to specify, in a very concise manner, some fundamental algorithms in various areas of computer science. The previous examples illustrate the use of spatial notions in the "programming in the small" task [13]. To conclude in section 7 we indicate some of the related work and we mention briefly some perspectives on the use of spatial notions to support "programming in the large".

2 Data Structures as Spaces[3]

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.

determination dependent upon them. It is an a priori representation, which necessarily underlies outer appearances. This statement justifies in some way the current trend towards the geometrization of physics since the end of the nineteenth century [35].

[2]G. Gaston-Granger in [29] 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 [56]). In this work, we rely on elementary concepts developed in the field of combinatorial algebraic topology for the construction of spaces [30].

[3]The ideas exposed in this section are developed in [24, 19].

- 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.

In section 6.2 we present two sorting algorithms. The first one is a variation of the well known bubble-sort. In this sorting procedure, two adjacent elements (neighbors!) are exchanged if some condition is met. In the second one, tokens move vertically in the columns of an array.

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 [40, 33]).

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*[4]. 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 [26, 21]. 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 a two-dimensional grid *Grid2* and to the moves `|a>`, `|b>`, and `|c>` allowed in the hexagonal lattice *Hexa* figured at the right of Fig. 4.

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,

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

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" [9] 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 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 [28] for an introduction to this domain.

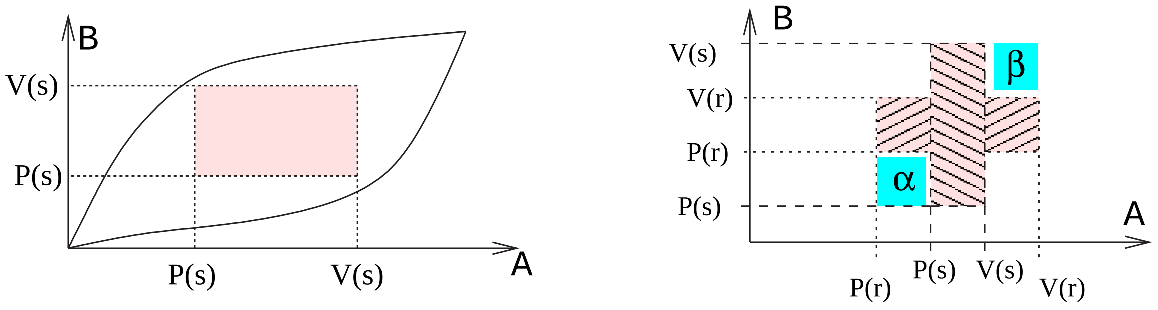


Figure 1: *Left:* The possible path taken by the process `A` || `B` is constrained by the obstruction resulting of a semaphore shared between the processes `A` and `B`. *Right:* The sharing of two semaphores between two processes may lead to deadlock (corresponding to the domain α) or to the existence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

3.2 The Topological Structure of Interactions[5]

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 the subsystems $S_1, S_2, \ldots, S_n$ is *functional*: the 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 $S_1^t, S_2^t, \ldots, S_{n_t}^t$ 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 S_j^t such that $S', S'' \in S_j^t$. Two subsystems S' and S'' are *separable* if there are some S_j^t such that $S' \in S_j^t$ and $S'' \notin S_j^t$ 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.

The set $\mathcal{S}$ has a *topological structure*: $\mathcal{S}$ corresponds to an *abstract simplicial complex*. An abstract simplicial complex [30] is a collection $\mathcal{S}$ of finite non-empty 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

[5]This section is adapted from [51].

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.

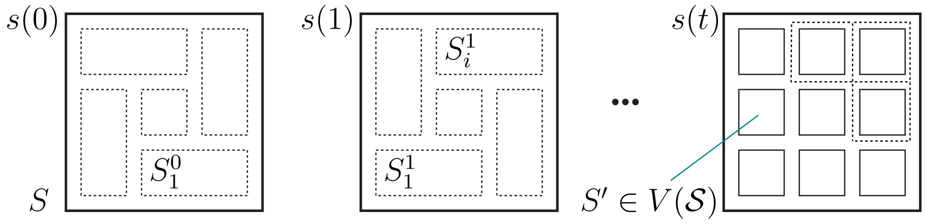


Figure 2: The interaction structure of a system S resulting from the subsystems of elements in interaction at a given time step.

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. We focus on the notions required to understand the rest of the paper.

4.1 Atomic values.

Atomic values (like integers, floats, booleans, strings, symbols...) with their usual functions, are available. Since MGS is a functional language, it has functions as first-class values. Functions are defined using the construction `fun` like in `fun max(x, y) = if (x > y) then x else y fi`. Optional parameters can be specified between brackets: `fun succ[inc=1](x) = x + inc`. In the application of the function, these parameters can be omitted like in `succ(0)` which returns `1` or explicitly set: `succ[inc=3](0)` returns `3`.

4.2 Topological Collections

The distinctive feature of the MGS language is its handling of entities structured by *abstract topologies* using *transformations* [25]. 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 [50].

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* [18]. 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* [32] and *shape type* [15] 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". Let us give an example. A *sequence* can be seen as a partial function $\mathbb{N} \to \mathit{Value}$. The support of a sequence of length n is the set $\{0, ..., n-1\}$. The set of positions $\mathbb{N}$ must be given with the neighborhood relation: i is neighbor of j iff $j = i + 1$.

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 [51] and some other arbitrary topologies specified by the programmer. We introduce some specific collection types along with the examples.

The MGS interpreter provides a dynamically typed version of the language. However, a compiler is under development and it is possible to enforce a static type discipline [10, 11]. 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).

The type system contains some particular types of the form $[\tau]\rho$ for the collections where τ is the type of the elements inside the collection, called the *content type* and ρ is its *container type* or *topology*. A topology can be a symbol of the set $\{set,\ bag,\ seq, \dots\}$ that contains the possible topologies of a collection, or a topology variable that we will denote by the Greek letter θ. Let us remark that a topology is not a type and that a topology variable θ cannot be used instead of a type variable α and vice-versa. A function f of type $[int]seq \to bool$ is a predicate on sequence of integers, $[\alpha]seq \to bool$ is a polymorphic predicate that acts on any sequence of elements and a predicate of type $[\alpha]\theta$ is a *polytypic* predicate that acts on any collection. The primitive `empty` that returns true if its argument does not contain any element is an example of the latter.

In order to deal with heterogeneous collections in our soft type system we use some *union types*, previously used by other authors [1, 12, 16]. A value with the type $\tau_1 \cup \tau_2$ has either the type τ_1 or the type τ_2. Knowing that a value has the type $\tau_1 \cup \tau_2$ you cannot conclude that it has the type τ_1 nor that it has the type τ_2. The integer `1` has the type *int* and has also the type $int \cup bool$.

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 d`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`.

4.3 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:

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

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

literal: a literal value `cte` matches an element with the same value. For example, 123 matches an element with value 123.

empty element the symbol `<undef>` matches an element with an undefined value, that is, 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. For some collection types, the neighborhood relation can be made more precise. For example, in a two-dimensional grid, one may look only for a neighbor along the `|north>` direction. This is simply written b `|north>` p.

guard: p/exp matches a path matched by p when the boolean expression exp evaluates to `true`. For instance, x`,`y `/` y`>`x matches two neighbor elements x and y such that the value associated to y is greater than the value associated to x.

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.

For some collection types it is possible to replace a subcollection by a collection with a different shape. Such collections are termed as *leibnizian* and are opposed to *newtonian* collections[6]. Examples

[6]This term comes from the different visions of space Newton and Leibniz had [19].

of leibnizian collections include sets, bags, sequences, graphs... A two-dimensional grid is an example of a newtonian collection: one cannot replace an arbitrary subset of a grid by a subset with another shape without destroying the 2D grid structure.

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). This transformation has type $(\alpha \to \beta) \to [\alpha]\theta \to [\beta]\theta$ and can be applied to any collection irrespectively to its topology. Such functions are said to be *polytypic* [33]. Polytypism comes "for free" when considering data structures in a uniform framework.

5 Simulation

In this section, we show through various examples 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[7] 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 meristeme, the simulation of colonies of ants foraging for food, etc.

5.1 The modeling of Dynamical Systems

5.1.1 Heat Diffusion in a Rod.

We will simulate the diffusion of heat H through a thin rod. The parabolic PDE $\frac{\partial H}{\partial t} = \alpha \frac{\partial^2 H}{\partial x^2}$ that describes the diffusion process can be discretized:

$$H(x, t + \Delta t) = (1 - 2c)H(x, t) + c(H(x - 1, t) + H(x + 1, t))$$

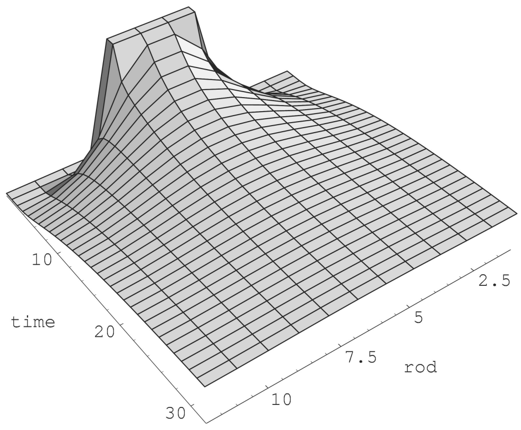
where $H(x, t)$ is the temperature of the element x of the sequence at time t and the parameter c depends on diffusion characteristics α of the rod (c is less than 0.5). Some boundary conditions can be added to the model. We can easily write the corresponding MGS program that computes the evolution of the temperature in the rod:

```
trans diffuse[c=0.25, Hleft=0, Hright=0] = {
    x / (left(x) == <undef>)  => Hleft;
    x / (right(x) == <undef>) => Hright;
    x => (1-2*c)*x + c*(right(x) + left(x))
}
```

The two first rules deal with the boundary conditions given by parameters `Hleft` and `Hright`. The operators `right` and `left` give access to the neighbors in a sequence. The special value `<undef>` is returned as the value associated with a position that does not have an associated value. The plot at the left of Fig. 3 gives an illustration of the output.

In the previous model, a sequence is used to describe the discretized rod. Bags can also be used to represent the same physical phenomena by using a different point of view. The rod is still discretized as

[7] `http://mgs.lami.univ-evry.fr`



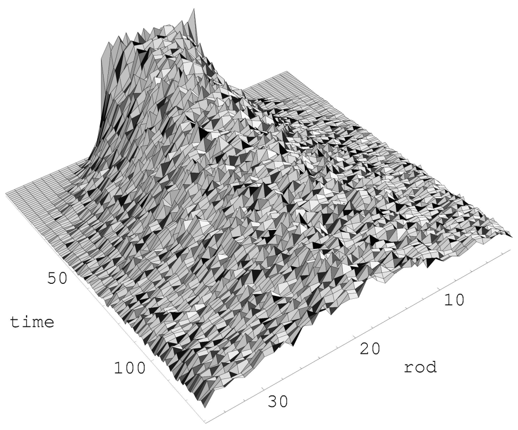


Figure 3: Result of the heat diffusion with two different models. The boundaries of the rod are kept at $0°C$. In the initial condition, only the middle third has a non zero temperature. In the two diagrams, the time goes from back to front.

a sequence of small boxes but this time we adopt a "Individual-based Modelling" approach. Each box contains some particles that represent a quantum of heat. We can represent each quantum of heat in the box x by an occurrence of the integer x in the bag. In this way, the state of the rod is a bag of integers. The cardinal of the bag represents the total amount of heat in the rod.

The corresponding evolution function is simple to state: a quantum of heat in a box x is allowed to jump in one of the neighbor boxes or to stay in its current box. The corresponding transformation is trivial in MGS:

```
trans diffuse = {     n => n + random(-1, 0, 1)     }
```

Two others rules can be added to deal with the boundary conditions. The function `random` chooses randomly one of its argument. The probability of choosing an argument instead of another is related to the parameter c in the continuous formulation. Indeed c is the probability for a particle to change its state. As a consequence and because each box has two neighbors, the probability to stay in the same state is $1 - 2c$ as it appears in the continuous equation. The right plot in Fig. 3 presents the stochastic version of the heat diffusion. In this simulation, they are 2500 quanta of heat in 30 boxes.

5.1.2 The DLA Evolution Function in MGS

Diffusion Limited Aggregation, or 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);

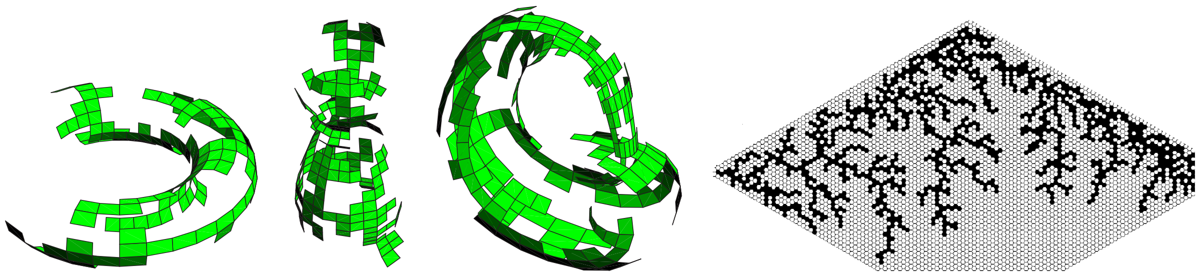


Figure 4: From left to right: the final state of a DLA process on a torus, a chess pawn, a Klein's bottle and an hexagonal meshes. The chess pawn is homeomorphic to a sphere and the Klein's bottle does not admit a concretization in Euclidean space. These two topological collections are values of the *quasi-manifold* type. Such collection are build using *G-map*, a data-structure widely used in geometric modeling [38]. The torus and the hexagonal mesh are GBFs.

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. 4 for a few results.

5.2 The modeling of Dynamical Systems with a Dynamical Structure (DS)^2

The previous examples are examples of continuous or discrete "classical dynamical systems". We term them "classical" because they exhibit 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 [22], or (DS)^2 in short. As pointed out by [20], many biological systems are of this kind.

To illustrate the use of MGS in the simulation of (DS)^2, we want to model a growing sheet of interacting cells. We take into account three phenomena: the tension and pressure between cells, the diffusion and reaction of two chemicals and the cell division that is driven by the concentration of one of the chemicals. The problem is that, due to the cell movements and division, the immediate neighbors of a cell evolve with the time. An illustration of the result is given in Fig. 5.

We use a Delaunay triangulation to compute the neighborhood of the cells. The Delaunay triangulation of a set of points is a collection of edges satisfying an "empty circle" property: for each edge we can find a circle containing the edge's endpoints but not containing any other points. In MGS, we start by defining the type of the collection that represents a cell:

```
record MecaCell = {x, y, z, vx, vy, vz, fx, fy, fz};;
record BioCell = {a, b, da, db, c};;
record Cell = MecaCell + BioCell;;
```

specify two record types, the first having the fields `x`, `y`, `z`, etc., representing the position of the cell and its velocity and acceleration; the second having the fields `a`, `b`, etc., representing the two diffusing chemical and their first derivative. The record *Cell* contains the fields of both *MecaCell* and *BioCell*.

We then define a *Delaunay collection type*. The specification:

```
collection delaunay(3) D3 =
  \e.if Cell(e) then (e.x, e.y, e.z) else ?("bad element for D3 type") fi ;;
```

defines a new Delaunay collection type in 3 dimensions. The type, called *D3*, is parameterized by a user function that extracts from each element in the collection, an abstract coordinate. In this example, the coordinate are simply stored in the value that represents a cell and the function simply checks that the cell's value has a correct type and returns its coordinate (as a sequence of 3 floats).

The movement of the cells results from the elastic and viscous forces in the Newton's equation of dynamics:

$$m.\frac{d^2F}{dt^2} = F_{\text{elastic}} + F_{\text{viscous}} = -k(L - L_0) - \mu\frac{dF}{dt}$$

The spring term has a constant k, the rest length is L_0 and μ is a viscous parameter. This equation is integrated for each cell by a simple method implemented by the transformation:

```
trans Meca = {
  e => let f = neighborsfold(sum(e), {fx=0,fy=0,fz=0}, e)
       in  e+{ x = e.x + dt*e.vx,
               ...
               vx = e.vx + dt*f.fx,
               ...
               fx = f.fx,
             }
}
```

The primitive `neighborsfold` iterates over the neighbors of a cell `e`. This primitive uses the function parameterizing the type *D3* to compute the neighbors of a given element. The expression in the `let` assignment computes the sum of the interaction force between a cell c and the neighbor cells; this sum is then used to update the state of the cell c. The operator `+` on records is an asymmetric merge: the fields of $\texttt{r} = \texttt{r}_1 + \texttt{r}_2$ are the fields of both $\texttt{r}_1$ and $\texttt{r}_2$ and $\texttt{r}.a$ has the value of $\texttt{r}_2.a$ if a is a field of $\texttt{r}_2$ else the value of $\texttt{r}_1.a$.

The diffusion and reaction of the two chemicals is modeled as a simplified version [55] of the Turing's diffusion-reaction model [54]. The evolution of the two chemicals is also implemented with only one transformation, using also a `neighborsfold` to compute the Laplacian of the concentrations.

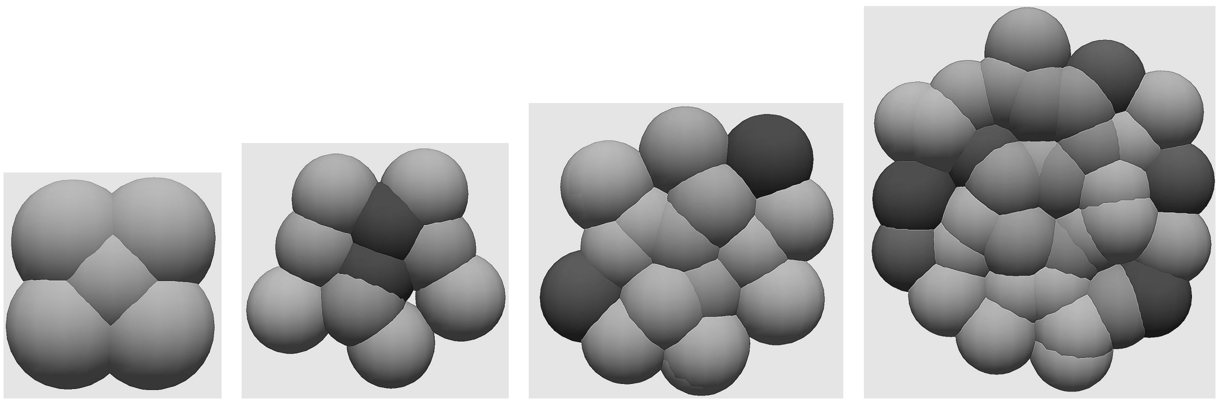


Figure 5: Four steps in the growth of a sheet of cells. The color of a cell is correlated with the concentration of the chemical that triggers the cell division (black cell will divide).

At last, when the concentration of `b` increases above a given level, the cell divides, which is implemented by the transformation:

```
trans Division = {
  e / e.b > lsplit => (e + {a=e.a/3, b=e.b/3}),
                      (e + {a=e.a/3, b=e.b/3, x=noise(e.x), y=noise(e.y), ...})
}
```

The coefficient 3 used to compute the repartition of the concentration in the daughter cells is arbitrary; the function `noise` disturbs a little its argument by adding a very small random quantity.

6 Programming in the Small: Algorithmic Examples

The two previous sections show 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 [42], 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.

6.1 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 [3, 4]. The Gamma transformation on the left is simply translated into the MGS transformation on the right:

```
M = do                                            trans M = {
    rp $x_1, \ldots, x_n$                             $x_1, \ldots, x_n$
    if $P(x_1, \ldots, x_n)$                $\Longrightarrow$       / $P(x_1, \ldots, x_n)$
    by $f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)$         => $f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)$ }
```

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 [53] and Maude [52].

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" [14] in this perspective.

6.2 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. 6. 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 really a bubble sort 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 [2]. 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. 6. The corresponding one-line MGS program is given by the transformation:

```
trans BeadSort = {   `empty |north> `bead  ⇒  1, `empty   }
```

This transformation is applied on a *Grid2*. The symbol `‘empty` is used to give a value to an empty place and the symbol `‘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.

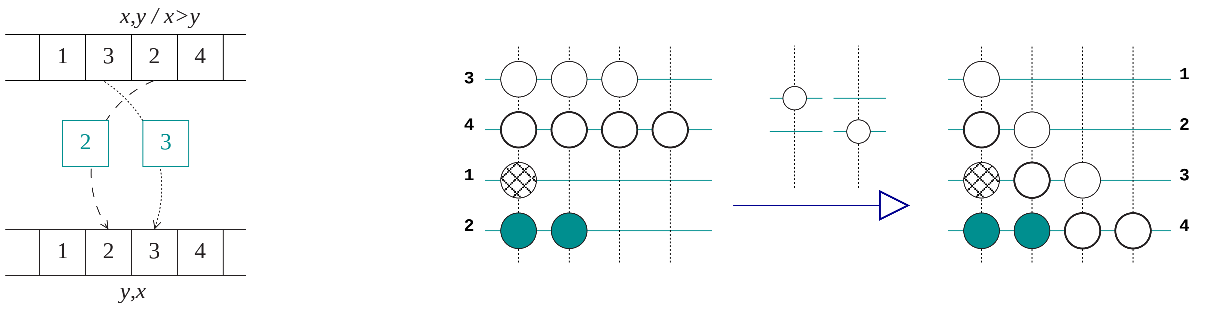


Figure 6: *Left:* Bubble sort. *Right:* Bead sort [2].

6.3 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 => assert(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 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 rules must be checked on all possible Hamiltonian paths and *H*`(g)` prints all the Hamiltonian path in `g` before returning `g`.

7 Conclusion: Programming in the Large ?

7.1 Current Status and Related Work

The topological approach we have sketched here is part of a long term research effort [26] developed for instance in [18] where the focus is on the substructure, or in [21] 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 [3], 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 [5]. 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 [44, 45] 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 [39] have long been used in the modeling of (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 [47, 46]). 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 [48]. 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 [23, 25], 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 paradigm. 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 URL `http://mgs.lami.univ-evry.fr`. All the examples in this paper have been processed with this interpreter. The results have been stored into a file using a variety of formats and visualized using either Mathematica or Imoview (an OpenGL viewer developed within the MGS project).

7.2 Programming in the Large ?

The previous presentation shows the ability of MGS to handle complex models of DS and to concisely express several algorithms. However, these examples illustrate only the "programming in the small" task and do not address the problem of the "programming in the large", that is the problems raised by the support of large software architecture, the interconnection of modules, the hiding of information, the capitalization and the reuse of existing code, etc. Programming in the large is certainly one of the major challenge a programming language must face [13]. We mention very briefly some of the possible supports provided by a topological approach to the "programming in the large" activity.

First, we advocate that the polytypic approach enables a better code reuse because it widens the applicability of the code. The topological approach, providing a unified framework for all data structure, promotes the development of polytypic and reusable functions.

Then, the underlying ideas of the MGS topological approach are smoothly embedded in a functional language: a transformation is a function, a topological collection type acts as an opaque type, etc. So, all the notions developed to help the programming in the large within functional languages, are also available: modules or packages and functors [37], mixin [41, 31], preprocessing tools, etc.

More directly, topological notions can be used to formalize *inheritance restructuring* [43]. The aim is to infer or restructure the inheritance hierarchy of classes in an object oriented language to achieve smaller, consistent data structure and better code reuse. The corresponding tools apply equally well to modules refactoring. The problem of inferring a class hierarchy from a set of concrete objects can be rephrased as the inclusion relation of a simplex in which the criteria used by Moore [43] and others to constrain the class hierarchy are *topological constraints* that have a simple and intuitive meaning [27].

This last work opens the way to a topological basis for a "module interconnection language" for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programmers and that can be checked for consistency by a compiler. The idea is to represent the services required or provided by a class, a module, a mixin or any code fragment, as the boundaries of a cellular space [30]. The boundary can then be merged to assemble code fragments into an integrated whole. This kind of geometric representation is a promising perspective, see for instance the preliminary work of F. Lamarche [36], where the free variables in a term correspond to the borders of a complex and the substitution is represented by the gluing of complexes along their simplices.

Acknowledgments

The authors would like to thanks Franck Delaplace at LaMI, Frédéric Gruau at University of Paris-Sud, Florent Jacquemard at 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 discussions. We gratefully acknowledge the financial support of the CNRS, the GDR ALP, IMPBIO, the ACI NANO-prog, the University of Évry and GENOPOLE-Évry.

References

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM Press, June 1993.

[2] 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.

[3] 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.

[4] 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–??, 2001.

[5] 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.

[6] R. W. Brockett. Smooth dynamical systems which realize arithmetical and logical operations. In H. Nijmeijer and J. M. Schumacher, editors, *Three Decades of Mathematical Systems Theory: A Collection of Surveys at the Occasion of the 50th Birthday of J. C. Willems*, volume 135 of *Lecture Notes in Control and Information Sciences*, pages 19–30. Springer-Verlag, 1989.

[7] R. W. Brockett. Dynamical systems that sort lists, diagonalize matrices, and solve linear programming problems. *Linear Algebra and its Applications*, 146:79–91, 1991.

[8] K. M. Chandy. Reasoning about continuous systems. *Science of Computer Programming*, 14(2–3):117–132, Oct. 1990.

[9] E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, 1971.

[10] 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.

[11] 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.

[12] F. Damm. Subtyping with union types, intersection types and recursive types. In *Symposium on Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 687–706. Springer-Verlag, 1994.

[13] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, 1975. also published in ACM SIGPLAN Notices Vol. 10, Issue 6 (June 1975).

[14] P. Fradet and D. L. Métayer. Structured Gamma. *Science of Computer Programming*, 31(2–3):263–289, July 1998.

[15] P. Fradet and D. L. Métayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

[16] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[17] 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.

[18] 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, Montréal, Sept. 2000. ACM-press.

[19] 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.

[20] 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 simumation of biological processes in the context of genomics", 12-17 may 2003, Dieppes, France.

[21] 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.

[22] 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 Publishers, 2001.

[23] 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.

[24] 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.

[25] J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.

[26] 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.

[27] J.-L. Giavitto and E. Valencia. *Diagrammatic Representation and Reasoning*, chapter A Topological Framework for Modeling Diagrammatic Reasoning Tasks. Springer-Verlag, 2001.

[28] E. Goubault. Geometry and concurrency: A user's guide. *Mathematical Structures in Computer Science*, 10:411–425, 2000.

[29] G.-G. Granger. *La pensée de l'espace.* Odile Jacob, 1999.

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

[31] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, editor, *Programming Languages and Systems, ESOP'2002*, volume 2305 of *LNCS*, pages 6–20, 2002.

[32] C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[33] J. Jeuring and P. Jansson. Polytypic programming. *Lecture Notes in Computer Science*, 1129:68–114, 1996.

[34] I. Kant. *Critique of Pure Reason.* Palgrave Macmillan, 2003. 1th edition: 1929. 2nd Rev. edition: 2003. Translation by Norman Kemp Smith of the 1787 2nd edition.

[35] H. Kragh. *Quantum Generations: A History of Physics in the Twentieth Century.* Princeton University Press, 2002. ISBN: 0-691-09552-3.

[36] F. Lamarche. Une théorie géométrique des objets symboliques. First meeting of the project GEOCAL (géometrie du calcul), Institut Mathématique de Luminy (T. Ehrhard), Marseille, Jan. 2004.

[37] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

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

[39] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[40] 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, Berlin.

[41] O. Michel and J.-L. Giavitto. Amalgams: Names and name capture in a declarative framework. Technical Report 32, LaMI – Université d'Évry Val d'Essonne, Jan. 1998. also avalaible as LRI Research-Report RR-1159.

[42] O. Michel and F. Jacquemard. An analysis of the Needham-Schroeder public-key protocol with MGS. In G. Mauri, G. Paun, and C. Zandron, editors, *Preproceedings of the Fifth workshop on Membrane Computing (WMC5)*, pages 295–315. EC MolConNet - Universita di Milano-Bicocca, June 2004.

[43] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 235–250. ACM Press, 1996.

[44] G. Paun. Computing with membranes. Technical Report TUCS-TR-208, TUCS - Turku Centre for Computer Science, Nov. 11 1998.

[45] G. Paun. From cells to computers: Computing with membranes (P systems). *Biosystems*, 59(3):139–158, March 2001.

[46] 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.

[47] P. Prusinkiewicz, A. Lindenmayer, J. S. Hanan, et al. *The Algorithmic Beauty of Plants.* Springer-Verlag, 1990.

[48] Z. Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 Sept. 1994.

[49] 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.

[50] R. D. Sorkin. A finitary substitute for continuous topology. *Int. J. Theor. Phys.*, 30:923–948, 1991.

[51] 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)*, LNCS, Amsterdam, October 2004. Springer. to be published.

[52] The MAUDE project. Maude home page, 2002. `http://maude.csl.sri.com/`.

[53] The PROTHEO project. Elan home page, 2002. `http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/`.

[54] A. M. Turing. The chemical basis of morphogenesis. *Phil. Trans. Roy. Soc. of London*, Series B: Biological Sciences(237):37–72, 1952.

[55] G. Turk. Generating textures for arbitrary surfaces using reaction-diffusion. In T. W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 289–298, July 1991.

[56] H. Weyl. *The Classical Groups (their invariants and representations).* Princeton University Press, 1939. Reprint edition (October 13, 1997). ISBN 0691057567.

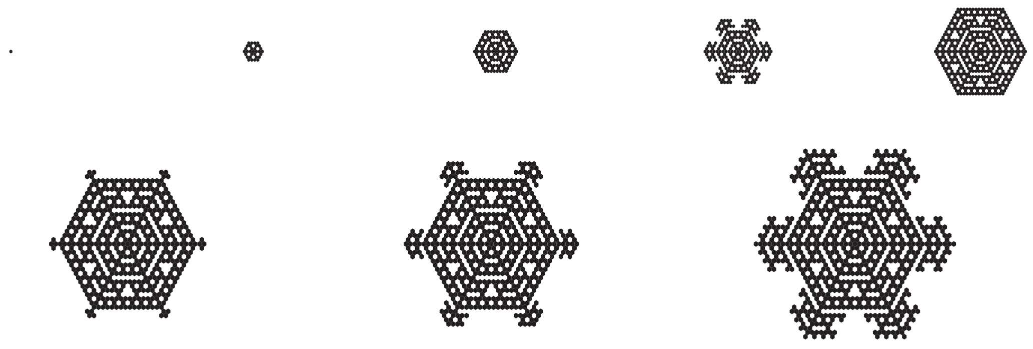


Figure 7: Formation of a snowflake modeled as a cellular automata on an hexagonal mesh in MGS. The pictured states are the step at time steps 1, 4, 8, 12, 16, 18, 20 and 23.

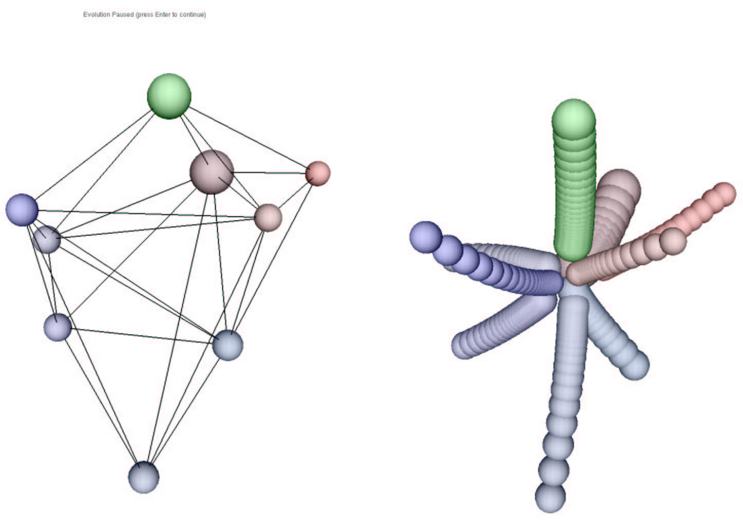


Figure 8: Each sphere in the picture above corresponds to a cell attracted by its neighboring cells by a spring. The neighborhood of a cell is computed dynamically using a Delaunay triangulation built from the cells position. At each time step, this neighborhood can change. The first picture is the initial state and shows the neighborhood using links between the cells. The second picture shows the final state, when the system has reached an equilibrium (each "tube" in this picture represents the successive positions of a cell). In MGS, the Delaunay collection type is a type constructor corresponding to the building of collections with a neighborhood computed from the positions of the elements in a d-dimensional Euclidean space.

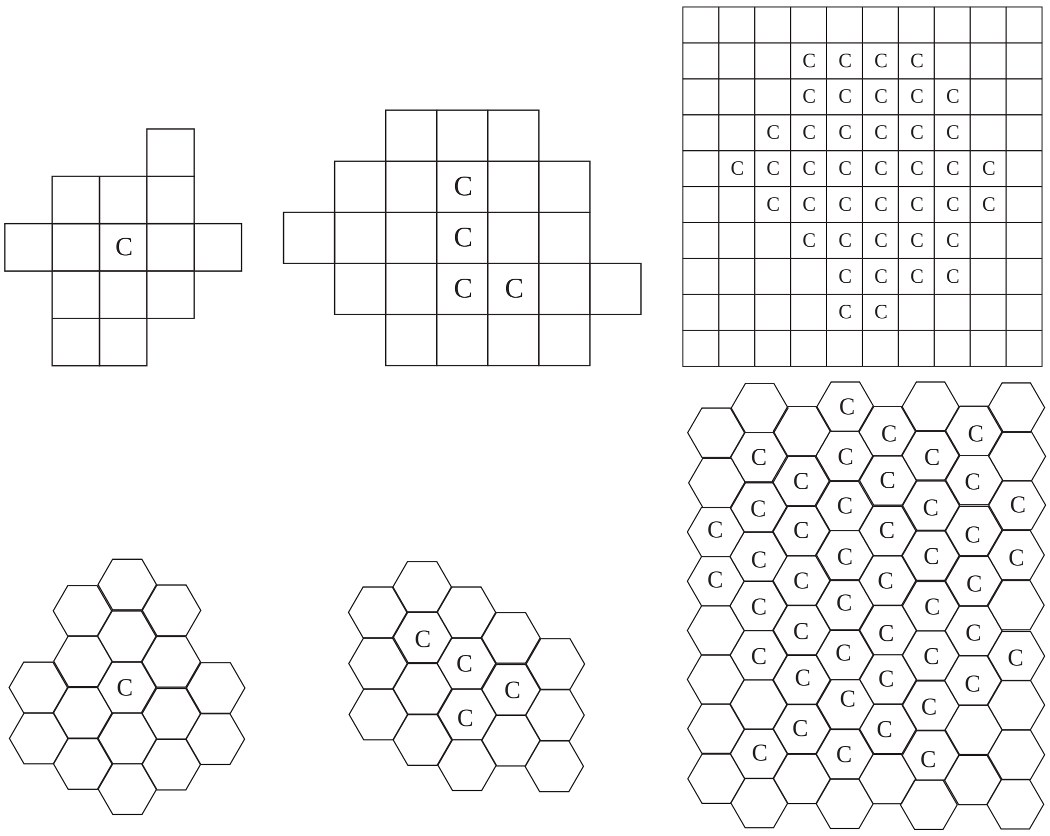


Figure 9: Eden's model on a grid and on an hexagonal mesh (initial state, and states after the 3 and the 7 time steps). The Eden's aggregation process is a simple model of growth. The model has been used since the 1960's as a model for such things as tumor growth and growth of cities. In this model (specifically, a type B Eden model), a 2D space is partitioned in empty or occupied cells. We start with only one occupied cell. At each step, occupied cells with an empty neighbor are selected, and the corresponding empty cell is made occupied. This process simply described as *exactly the same* transformation for both cases:

```
trans Eden = {   x,<undef> / x  ⇒  x,true   }
```

We assume that the boolean value `true` is used to represent an occupied cell, other cells are simply left undefined. Then the previous rule can be read: an occupied element x and an undefined neighbor are transformed into two occupied elements. This model cannot be coded by only one simple rule on a two-state cellular automata if one wants to avoid that two distinct occupied cells preempt the same unoccupied cell.