Topological Collections, Transformations and their Application to the Modeling and the Simulation of Dynamical Systems

Jean-Louis Giavitto

CNRS – LaMI, umr 8042, Université d'Évry – GENOPOLE
Tour Evry-2, 523 Place des Terrasses de l'Agora
91000 Évry, France

giavitto@lami.univ-evry.fr

1 Introduction

I take the opportunity given by this invited talk to promote two ideas: (1) a *topological point of view* can fertilize the notion of rewriting and (2) this topological approach of rewriting is at the core of the modeling and the simulation of an emerging class of dynamical systems (DS): the *DS that exhibit a dynamical structure* (or $(\mathrm{DS})^2$ in the rest of this paper).

This presentation is based upon the results of two research projects, 81/2 and MGS, that I have pursued hand in hand with Olivier Michel. The results and software tools presented here belong also to him and have been elaborated thanks to our long and fruitful collaboration.

I have voluntarily adopted in this presentation an informal style, including some non-technical considerations. Thus, the reader must take the opinions, subjective statements and positions expressed here with a grain of salt. For the technical details, he may refer to the papers published elsewhere. The MGS home page : http://mgs.lami.univ-evry.fr is a good starting point.

This presentation is organized as follows. Section 2 tries to develop an alternative understanding of the concept of a data structure: a data structure can be seen as a space where the computation moves following some path. This point of view is exemplified in section 3 on the design of a uniform data structure. The result, called a GBF, is funded on the group generated by the elementary moves (or displacements) in the data structure. The section 4 introduces the MGS 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). Simple examples of MGS programs are given in section 4.4. However, a privileged application domain for MGS is the modeling and simulation of dynamical systems that exhibit a dynamic structure. Section 5 sketches this point and gives a short presentation of several models. We review to conclude some related and future work.

2 Data Structures as Spaces

The fundamental concept of *data structure* is ubiquitous in computer science as well as in all branches of mathematics. Its characterization is then not easy. Some approaches emphasize on the construction of more sophisticated data structures from basic ones (e.g. domain theory); other approaches focus on the operations allowed on data structures (e.g. algebraic specification).

Species of structures. In [BLL97], a data structure s is presented as an *organization* or an *arrangement* o performed on a data set D. Quoting the introduction we can say that it is customary to consider the pair $s = (o, D)$ and to say that s is a structure o of D (for instance a *list* of *int*, an *array* of *float*, etc.). It is outlined that a customary approach consists in working with these pairs in the framework of axiomatic set theory. For example, the set $\mathcal{G}$ of simple directed graphs (directed graphs without multiple edges) can be defined by:

$$s = (o, D) \in \mathcal{G} \quad \Leftrightarrow \quad o \subseteq D \times D$$

This traditional approach consider equally the structure o and the set D and does not stress the structure o as a set of *places* or *positions*, independently of their occupation by elements of D. This last point of view is taken into account by the less traditional approach of *species of structures* [BLL97] motivated by the development of enumeration techniques and counting problems.

Space of a data structure. This point of view is also fruitful, even if one is not interested in counting the instances of a data structure. As a matter of fact, a lot of algorithms are structured following the structure of their data input or their data output and are largely insensitive to the precise values in their data set. This is obviously true for all polymorphic and polytypic functions, like `map`, `fold`, etc. [MFP91]. The notion of *shape* [Jay95] and *shape type* [FM97] also separates the set of places of a data structure from the values it contains.

Once we do not focus on the values manipulated in a program, we can analyze the previous notions as attempts to specify *classes of moves* or *paths* related to a given data structure. For example, there are two kinds of fold on lists: `fold_left` traverses the list from the head to the tail, and `fold_right` goes in the reverse direction. Another example: a shape type in [FM97] is defined as a grammar specifying the admissible paths resulting from following pointers in `C` data structures. So, in our context, the point of view is *topological* rather than combinatorial: *a data structure can be seen as a space*, the set of places or positions between which the programmers, the computation and the values, move.

At last, the notion of move or path relies on some notion of *neighborhood*: moving from one point to a neighbor point. Although speaking of *neighborhood* in a data structure is not usual, the relative accessibility from one element to another is a key point usually considered in a data structure. For example:

– 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, 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. For example $(i-1, j)$ is the index used to access the "north neighbor" of point (i, j) (we assume that the "north" direction is mapped to the first element of the index tuple). 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). More than 99% of array references are affine functions of array indexes in scientific programs [GG95].

This list of examples can be continued to convince ourselves that a notion of *logical neighborhood* is fundamental in the definition of a data structure. 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. The computation indeed complies with the logical neighborhood structure of the elements. For instance, recursive computations on a data structure respect so often the logical neighborhood, that standard high-order functions can be automatically defined from the data structure organization (think about catamorphisms and others polytypic functions on inductive types [FS96, NO94]).

Paths and Computations. In a sequential computation, elements of the data structure are visited one after the other. We assume that if element e' is visited just after element e in a data structure s, then e' must be a neighbor of e in some (concrete or abstract) way. We call the move from e to e' a *shift* and the succession of visited elements makes a path in s. The idea of sequential path can be extended to include parallel modes of computations: multi-dimensional paths must be used instead of one-dimensional paths [GJ92].

To summarize our presentation, we assume that a computation induces a path in a space defined by the neighborhood relationships between the elements of a data structure. At each shift, some elementary computation is done. Each topological operation used to build a path can then be turned into a new control structure that composes program fragments.

This schema is presented in an imperative setting but can be easily rephrased into the declarative programming paradigm by just specifying the linking of computational actions with path specifications. When a path specification matches an actual path in a data structure, then the corresponding action is triggered. It is very natural, especially in this topological framework, to require that the results of the computational action be *local* : the corresponding data structure

transformation is restricted to the value of the the elements involved in the path and eventually to the organization of the path elements and their neighborhood relationships. Such transformation is qualified as local.

This declarative schema induces a rule-oriented style of programming: a rule defines a local transformation by specifying the path to be matched and the corresponding action. A program run consists in the transformation of a whole data structure by the simultaneous application of local transformations to non-intersecting paths. Obviously, such *global* transformation can then be iterated. Figures 1, 2 and 3 present three examples of algorithms where this topological emphasis is particularly relevant.

Rewriting and the Topological Approach. This topological approach shares many features with the idea of rewriting. Indeed, we can suppose that the computational action linked to a path is to replace this path by another one: this is the case for the four previous examples. Then, we retrieve the idea of rewriting, see figure 4 and 5, except that usually rewriting is described as the substitution of some sub-structure by another one (e.g. a sub-term by another term). What we gain with the topological emphasis is to focus on paths in the data structure instead of sub-structures [1]. Is this a real gain ?

The purpose of the MGS research project is to answer this question. To have a positive answer we have to show that:

1. it is possible to define a data structure through the specification of the neighborhood of its elements,
2. it is possible to define the substitution of a path by another one and to control the substitution strategy,
3. and this is useful in some application area.

The next section sketches the notion of *Group Based Datafield* (GBF) and is an example of a positive answer to question 1. This example is important also because it integrates the array data structure, which opens the way to array rewriting and gives at least one answer to question 3. The section 4 introduces an experimental language, called also MGS, used to investigate the design space of question 2. The section 5 shows the use of the previous tools in the domain of dynamical systems modeling (especially in biology) and provide another answer to question 3.

3 The Example of Uniform Neighborhood Data Structures : GBF

From now on, we use the term *topological collection* to stress the topological organization of the data structure's elements. In this section, we will sketch a possible design for *uniform* topological collections. A topological collection is uniform if every element of the data structure has the same neighborhood

[1] see however [Gia00].

structure. More precisely, we assume in this study that: (1) the set of places filled by the elements of the data structure is predefined (i.e. preexists to any occurrence of the data structure), and (2) the shifts followed to go from some place to a neighbor place can be named and (3) that the set $\mathbf{G}$ of shift's names, called *directions*, is the same for all places (like for example a "next neighbor" and a "previous neighbor" that exist for each element in a circular list).

The Group Structure of Uniform Neighborhood. Let "a", "b", "c"... be the direction's names and let $P\langle a\rangle$ be the "a" neighbor of the element P. Displacement operations can be composed: using a multiplicative notation, we write $P\langle a.b\rangle$ for $\big(P\langle a\rangle\big)\langle b\rangle$. Displacement composition is associative. We note 1 the null displacement, i.e. $P\langle 1\rangle = P$. Furthermore we will define a unique inverse displacement a^{-1} for each displacement a such that $P\langle a.a^{-1}\rangle = P\langle a^{-1}.a\rangle = P$. In other words, the displacements constitute a *group* $\mathcal{G}$ for the displacement composition, and the application $\cdot\langle\cdot\rangle$ of the displacements to the places is the *action of the group over the places of the data structure*. The simplest choice for the set of places $\mathcal{P}$ and the corresponding action is to let $\mathcal{P} = \mathcal{G}$ and $P\langle a\rangle = P.a$ (the group acts transitively on itself).

We assume that the group $\mathcal{G}$ is specified through a finite presentation with generators $\mathbf{G}$ (and $\mathcal{G}$ denotes indifferently the group and its presentation). Then, the discrete space spawned by $\mathcal{G}$ acting on itself is conveniently described by the *Cayley graph* associated to the presentation. See figure 6 for a dictionary between graph theory and group related concepts.

Group Indexed Data Structure. A GBF is an extension of the notion of array, where the elements are indexed by the elements of a group [GMS95, GM01a]. A GBF value g of type $\mathcal{G}$ is a partial function with a finite definition domain[2] that associates a value to some group elements. The group elements are the places of the collection. Thus the empty GBF is the everywhere undefined function. The acronym GBF stands for Group Based Datafield. The formalization of a data structure as a function is not new; it constitutes for instance, the foundation of the theory of *data fields* [Lis93] and is heavily used in [Gia00]. In computer science, it is common to think about a function as a rule to be performed in order to obtain a result starting from an argument: this is the *intensional* notion of functions. Here, we better rely on the *extensional* notion: a function is a set of pairs relating the argument and the result. This is closer to the concept of a data structure: for instance, an array tabulates the relationship between the set of indices and the array elements and a GBF tabulates the relationship between the set of places $\mathcal{G}$ and their values (this is why GBFs are required to have a finite definition domain). We insist that the view of data structures as functions is only logical and appears only at the level of the data structure definition. It does not assume anything on the data structure implementation.

[2] The definition domain of g is the subset of $\mathcal{G}$ of the elements having a well defined image by g.

GBF in MGS. Here is an example. The finite presentation

```
gbf Grid2 = < north, east >
```

introduces in MGS (see section 4) a new collection type called *Grid2*, corresponding to the Von Neumann neighborhood in a classical array (a cell above, below, left or right – not diagonal) see figure 7. The two names `north` and `east` refer to the directions that can be followed to reach the neighbors of an element. These directions are the generators of the underlying group structure. The `<` and `>` brackets are used for the presentation of Abelian groups and to avoid the explicit writing of the commutation equations. In this presentation, there is no explicit equation (beside the implicit commutation of the generators): *Grid2* is a free Abelian group.

The following declaration defines a non-free Abelian group:

```
gbf Hexagon = <east, north, northwest; east+north=northwest>
```

The Cayley graph of *Hexagon* defines an hexagonal lattice that tiles the plane, see figure 7 and 8. Each cell has six neighbors (following the three generators and their inverses). The equation `east + north = northwest` specifies that a move following `northwest` is the same has a move following the `east` direction followed by a move following the `north` direction.

Uniform neighborhood and classical data structures. Free groups with n generators correspond to n-ary trees and Abelian GBF corresponds to twisted and circular grids (the free Abelian group with n generators generalizes n-dimensional arrays). Thus, GBF are able to describe in the same formalism both tree and array, a feature not available with regular inductive data types.

GBF Implementations. Accessing the value associated to a group element requires the comparison of generator words modulo the equation of the GBF: this is the word problem for groups and it is undecidable in general. However, for large and interesting families of groups (e.g. free groups, Abelian groups, automatic groups) the problem is solvable. Actually the MGS implementation is restricted to Abelian groups.

4 Topological Collections and their Transformations

In this section, we want to show how a declarative programming style, based on rules and a general notion of rewriting, can be developed on topological collections like the GBF presented in the previous section. The topological approach sketched in section 2 is investigated through an experimental declarative programming language called MGS [GM01c, GM02b]. MGS embeds the idea of topological collections and their transformations into the framework of a simple dynamically typed functional language. Collections are just new kinds of values and transformations are functions acting on collections and defined by a specific syntax using rules. Functions and transformations are first-class values and can

be passed as arguments or returned as the result of an application. MGS is an applicative programming language: operators acting on values combine values to give new values, they do not act by side-effect. In our context, dynamically typed means that there is no static type checking and that type errors are detected at run-time during evaluation. Although dynamically typed, the set of values has a rich type structure used in the definition of pattern-matching, rule and transformations.

Transformation of a Topological Collection. The *global transformation* of a topological collection C consists in the parallel application of a set of *local transformations*. A local transformation is specified by a rewriting rule r that specifies the change of a subcollection. The application of a rewrite rule $r = \beta \Rightarrow f(\beta, ...)$ to a collection C:

1. selects a path B of C whose elements match the *path pattern* β,
2. computes a new collection B' as a function f of B and its neighbors,
3. and specifies the insertion of B' in place of B into C.

In the rest of this section, we first describe the topological collection types available in MGS beside the GBF. We introduce the notion of "Newtonian" and "Leibnizian" collection, because this distinction is crucial for the behavior of rule application. Subsection 4.2 sketches the most common pattern that can be used in the left hand side (l.h.s.) of a rule. Then we discuss some of the application strategy available in MGS. Finally, subsection 4.4 gives some simple examples of real MGS program.

4.1 Newtonian and Leibnizian Collection Types

There are several predefined collection types in MGS, and also several means to construct new collection types. The collection types can range in MGS from totally unstructured with sets and multisets to more structured with sequences and Abelian GBFs, Delaunay neighborhood and graphs (other topologies are currently under development). For any collection type `T`, the corresponding empty collection is written `():T`. Elements in a collection `T` can be of any type, including collections, thus achieving *complex objects* in the sense of [BNTW95]. The name of a type is also a predicate used to test if a value has this type: `T(`v`)` returns true only if v has type `T`.

Monoidal Collections. Set, multiset (or bag) and sequences are members of the monoidal collection family. As a matter of fact, a sequence (resp. a multiset) (resp. a set) of values can be seen as an element of the free monoid (resp. the commutative monoid) (resp. the idempotent and commutative monoid). The join operation in V^* is written by a comma "`,`" and induces the neighborhood of each element: let E be a monoidal collection, then elements x and y in E are neighbors iff $E = u,x,y,v$ for some u and v. This definition induces the following topology:

- for sets (type `set`), each element in the set is neighbor of any other element (because of the commutativity, the term describing a set can be reordered following any order);
- for multiset (type `bag`), each element is also neighbor of any other (however, the elements are not required to be distinct as in a set);
- for sequence (type `seq`), the topology is the expected one: an element not at the end has a neighbor at its right.

The comma operator is overloaded in MGS and can be used to build any monoidal collection (the type of the arguments disambiguate the collection built).

Newtonian and Leibnizian Collection Types. Coming back to the idea of seeing a data structure as a space, we can note a great difference between the "kind of space" involved by the GBFs and the monoidal collections. The two concepts of space involved by these data structure may be contrasted as follows:

1. in a GBF, the underlying space preexists (as the Cayley graph of the finite presentation) and is thought as a *container* for the collection elements;
2. in a monoidal collection, e.g. a set, the underlying space exists only by the virtue of the elements present in the collection.

The first notion as been advocated by Newton in opposition with Leibniz and Huygens [Jam93]. The last attributes a positional quality to the elements. In this approach, there is no such things like an empty place[3].

This distinction has several impacts on the management of the data structures. Consider the rule:

x`,` $y \Rightarrow$ `x`

Intuitively it defines the erasure of an element y neighbor of an element x. This does not raise any difficulty in a Leibnizian collection: applied one time to a set with a cardinal greater than 2, this rule removes one randomly chosen element. However, in a Newtonian collection like an array, the erasure of y leave an empty cell, because the cell itself cannot disappear without breaking the neighborhood. The content of an empty cell is the special value `<undef>`.

Another distinction is that Newtonian collections correspond to an absolute space, where the place can be named, denoted and used in all the collections with the same type. There is no such thing for a Leibnizian collection: e.g. there is no notion of absolute place for the element of a multiset.

4.2 Path Patterns

A path is a sequence of elements and thus, a path pattern *Pat* is a sequence or a repetition *Rep* of *basic filters*. A basic filter*Bfilt* matches one element in a GBF.

[3] An empty `set/seq/bag` is a space of a certain kind without any place, and not an empty space.

The grammar of path patterns reflects this decomposition:

$$\begin{aligned} \mathit{Pat} &::= \mathit{Rep} \mid \mathit{Rep}\ \mathit{Dir}\ \mathit{Pat} \mid \mathit{Pat}\ \texttt{as}\ \mathrm{id} \mid \texttt{(}\mathit{Pat}\texttt{)} \\ \mathit{Rep} &::= \mathit{Bfilt} \mid \mathit{Bfilt}\texttt{/}\mathit{exp} \mid \mathit{Bfilt}\ \mathit{Dir}\texttt{+} \\ \mathit{Bfilt} &::= \texttt{cte} \mid \mathrm{id} \mid \texttt{\_} \mid \texttt{<undef>} \\ \mathit{Dir} &::= \texttt{,} \mid \texttt{|}\mathtt{u}_1\texttt{, ..., }\mathtt{u}_n\texttt{>} \end{aligned}$$

where `cte` is a literal value, id ranges over the pattern variables, *exp* is a boolean expression, and $\mathtt{u}_i$ is a word of generators of a GBF. 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 in a GBF with value 123.

empty element the symbol `<undef>` matches an element with an undefined value, that is, an element whose position does not belong to the support of the GBF. The use of this basic filter is subject to some restriction: it can occur only as the neighbor of a defined element.

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 the rule and denotes the value of the matched element.
If the pattern variable a is not used in the rest of the rule, one can spare the effort of giving a fresh name using the anonymous filter `_` that matches any element with a defined value. The position of a is accessible through the expression $pos(x)$.

neighbor: b *dir* p is a pattern that matches a path with first element matched by b and continuing as a path matched by p with the first element p_0 such that p_0 is neighbor of b following the *dir* direction. The specification *dir* of a direction is interpreted as follows:
— the comma "`,`" means that p_0 and b must be neighbors.
— the direction $\texttt{|}\mathtt{u}_1\texttt{, ..., }\mathtt{u}_n\texttt{>}$ means that p_0 must be a $\mathtt{u}_0$-neighbor *or* a $\mathtt{u}_1$-neighbor *or* ... *or* a $\mathtt{u}_n$-neighbor of b;
For example, x, y matches two connected elements (i.e., x must be a neighbor of y). The pattern

```
1 |east> _ |north,east> 2
```

matches three elements. The first must have the value `1` and the third the value `2`. The second is at the east of the first and the last is at the north *or* at the east of the second.

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

naming: a sub-pattern can be named using the `as` construct. For example, in the expression `(1,` x `|north>+ , 3) as P`, the variable `P` is binded to the path matched by `1,` x `|north>+, 3`.

repetition: pattern b *dir*+ matches a non-empty path b *dir* b *dir*...*dir* b. If the basic filter b is a variable, then its value refers the sequence of matched elements and not to one of the individual values. The pattern x`+` is an abbreviation for "`(_ ,+) as` 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 in the position of a filter. That is, the path pattern x,x is forbidden. However, this pattern can be rewritten for instance as: $x,y/y = x$.

4.3 Substitution and Application Strategies

Paths and Subcollections. A path pattern is used to find the occurrence of the specified path in a collection C. The matched path can be seen as a sequence of elements in C as well as a subcollection of C: the collection made of the elements in the path and inheriting its organization from C. Therefore, the right hand side (r.h.s.) can compute a sequence as well as a subcollection. If the r.h.s. is a sequence, then the nth element of the r.h.s. replaces the nth element of the matched path (this holds for Newtonian collections, Leibnizian collections are more flexible and allow the insertion or the deletion of elements). If the r.h.s is a collection, then this collection is pasted into C as a replacement for the matched path. The pasting operation depends of the collection kind and can be parameterized by giving explicit attributes to the arrow.

For example, suppose that we want to replace each `1` in a sequence by a series of three `1`. The corresponding rule is:

```
1 ⇒ 1, 1, 1
```

The behavior of the previous rule is the intended behavior. For example, applied to sequence `0, 1, 2, 1, 0` we obtain `0, 1, 1, 1, 2, 1, 1, 1, 0`. However, there is a possible ambiguity with a rule that replaces each `1` by only one element which is, unfortunately, a sequence. That is, the desired result is a sequence of five elements: `0, (1,1,1), 2, (1,1,1), 0` (this sequence has for elements 3 integers and 2 sequences). This behavior is achieved by overriding the default pasting strategy of ⇒:

```
1 =⇒{noflat}⇒ 1, 1, 1
```

The attribute `noflat` enables the desired behavior.

Priorities and Application Order. Others attributes enable the control of the rule application strategy. For instance, rules can have a priority used to chose the next paths to match. However, the only property ensured by the MGS rewriting engine is the following: if no rule at all applies during the application of a transformation T on a collection C, then there is no occurrence in C of the paths specified by the l.h.s. of the rules of T. Nevertheless, if the l.h.s. of the T's rules specify only paths of length one, an additional property is satisfied: these rule are applied in a maximal parallel manner. For example

x ⇒ `f(`x`)`

is a rule that implements a polytypic `map f`: this rule replaces each element x of a collection by `f(`x`)`, for any collection type.

Iterations and Fixpoints. A transformation T is a function like any other function and a *first-class* value. For instance, a transformation can be passed as an argument to another function or returned as a result. It allows to sequence and compose transformations very easily.

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

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

4.4 Simple Examples

The path pattern language introduced above is largely enough to code the examples of figures 1, 2, and 3. We present the first three algorithms.

The transformation:

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

is applied on a *Grid2*. The constant `empty` is used to give a value to an empty place and the constant `1` 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 exchanged. Indeed, the comma in the MGS expression at the r.h.s. of a rule[4] is used to build a sequence by listing its elements.

The transformation *BubbleSort* acts on a sequence. Although the sequence collection type can be specified as a GBF with only one generator, the sequence type is predefined in MGS and has specific properties (see section 4.1). The transformation is defined as:

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

This is not really a bubble sort because swapping of elements can take at arbitrary places; hence an out-of-order element does not necessarily bubble to the top in the characteristic way.

The two previous examples do not create new elements in the collection. The *Erastothene* transformation computes the ordered sequence of the prime integers. Each element i in the sequence corresponds to the previously computed ith prime P_i and is represented by a record $\{\texttt{p = } P_i\}$. This element can receive

[4] A comma is also used in a path expression to denote the neighborhood relationship between two elements in a collection in the l.h.s. The two usages agree, because in the sequence a`,` b the elements a and b are neighbors.

a candidate number n and is then represented by a record {p = P_i, a = n}. If the candidate passes the test, then the element transforms itself to a record r = {x = P_i, b = n}. If the right neighbor of r is of form {x = P_{i+1}}, then the candidate n skip from r to the right neighbor. When there is no right neighbor to r, then n is prime and a new element is added at the end of the sequence. The first element of the sequence is distinguished and generates the candidates. Accordingly, the *Erastothene* transformation consists in 6 rules named *genere1*, *genere2*, *test1*, *test2*, *pass* and *create*:

```
trans Erastothene = {
  genere1:  n/int(n), <undef>  ⇒  n, {x=n}
  genere2:  n/int(n), {p=x, ~a, ~b}  ⇒  n, {x=n}
  test1:    {p=x, a=y, ~b} / y mod x == 0  ⇒  {p=x}
  test2:    {p=x, a=y, ~b} / y mod x <> 0  ⇒  {p=x, b=y}
  pass:     {p=x1, b=y}, {p=x2, ~a, ~b}  ⇒  {p=x1}, {p=x2, a=y}
  create:   {p=x, b=y}, <undef>  ⇒  {p=x}, {p=y}
}
```

The pattern {p=x, a=y, ~b} matches a record with a field p (and the value of this field is binded to x), a field a and no field b. The pattern n/int(n), <undef> matches a path reduced to a single integer (there is nothing at the right of this integer). Consequently, it matches the end of a sequence (if this end is an integer). The rule genere1 is used only once, at the beginning, when the transformation is applied to the sequence singleton 2,():seq.

5 Application to the Modeling of Dynamical Systems with a Dynamic Structure

Our topological approach is motivated by some considerations internal to computer science, and also by the needs expressed by some application domains. A target application domain for MGS is the modeling and simulation of dynamical systems (DS) and especially DS that exhibit a dynamic structure ((DS)^2). This kind of dynamical systems is very challenging to model and simulate. New programming concepts must be developed to ease their modeling and simulation.

Dynamical Systems with a Dynamical Structure. Intuitively, a dynamical system is a formal way to describe how a point (the state of the system) moves in the *phase space* (the space of all possible states of the system). It gives a rule, the *evolution function*, telling us where the point should go next from its current location. There exist several formalisms used to describe a DS: ordinary differential equations (ODE), partial differential equations (PDE), iterated equations (finite set of coupled difference equations), cellular automata, etc., following the discrete or continuous nature of the time, the space and the value used in the modeling.

Many DS systems are structured, which means that they can be decomposed into parts and *sometimes* the whole state s of the system is simply the product

of the state of these parts. The evolution of the state of the whole system is then viewed as the result of the changes of the state of its parts. In this case, the evolution function h_i of a the state of a part o_i depends only on a subset $\{o_{i_j}\}$ of the state variables of the whole system. In this context, we say that the DS exhibits a *static structure* if:

1. the state of the system is statically described by the state of a fixed set of parts and this set does not change in time;
2. the relationships between the state of the parts, specified as the functions h_i between o_i and the arguments o_{i_j}, are also fixed and do not change in time.

Moreover, we say that the o_{i_j} are the *logical neighbors* of o_i (because very often, two parts of a system interact when they are physical neighbors). 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).

As pointed out by [GGMP02], many biological systems can be viewed as a dynamical system in which not only the values of state variables, but also the *set* of state variables *and/or* the evolution function, change over time. We call these systems *dynamical systems with a dynamic structure* following [GM01c], or $(\text{DS})^2$ in short. An obvious example is given by the development of an embryo. Initially, the state of the system is described solely by the chemical state o_0 of the egg (no matter how complex this chemical state can be). After several divisions, the state of the embryo is given not only by the chemical state o_i of the cells, but also by their spatial arrangement[5]. The number of cells, their spatial organization and their interactions evolve constantly in the course of the development and is not handled by one fixed structure $\mathcal{O}$. On the contrary, the phase space $\mathcal{O}(t)$ used to characterize the structure of the state of the system at time t must be computed jointly with the running state of the system. In this kind of situation, the dynamic of the whole system is often specified as several local competing transformations occurring in an organized set of simpler entities. The organization of this set is subject to possible drastic changes in the course of time and is a plain part of the state of the DS.

The MGS approach. The main idea to model $(\text{DS})^2$ is to follow an approach developed recently by several authors [FMP00, Man01, EKL+02b, EKL+02a]. The point is to use rewriting rules to model the parts of the system in interaction.

More specifically, we want to use an MGS topological collection S to represent the state of a dynamical system at a given time. The elements in the collection represent either entities (a subsystem or an atomic part of the dynamical system) or messages (signal, command, information, action, etc.) addressed to an entity.

[5] The neighborhood of each cell is of paramount importance to evolution of the system because of the interplay between the shape of the system and the state of the cells. The shape of the system has an impact on the diffusion of the chemical signals and hence on the cells state. Reciprocally, the state of each cell determines the evolution of the shape of the whole system.

A path or a subcollection in S represents a subset of interacting entities and messages in the system. The evolution of the system is achieved through transformations, where the l.h.s. of a rule typically matches an entity and a message addressed to it, and where the r.h.s. specifies the entity's updated state, and possibly other messages addressed to other entities.

If one uses a multiset organization for the collection, the entities interact in a rather unstructured way. More organized topological collections are used for more sophisticated spatial organizations and interactions (like GBFs or Delaunay).

More generally, many mathematical models of objects and processes are based on a notion of state that specifies the object or the process by assigning some data to each point of a physical or abstract space. The MGS programming language is designed to support this approach offering several mechanisms to build complex and evolving spaces and handling the maps between these spaces and the data.

In the rest of this section we present three examples involving various topology. The first one involves the use of sequences and multisets and is related to the cleavage of DNA strings floating in a chemical solution. The second example uses an hexagonal lattice to discretize the 2D formation of a snowflake. The last one sketch the trajectory of cells attracted by some neighbors. This example involves a dynamic topology computed as the result of the Delaunay triangulation of a set of points in Euclidean space.

5.1 Restriction Enzymes

This example shows the ability to nest different topologies to achieve the modeling of a biological structure. We want to represent the action of a set of restriction enzymes on the DNA. The DNA structure is simplified as a sequence of letters `A`, `C`, `T` and `G`. The DNA strings are collected in a multiset. Thus we have to manipulate a multiset of sequences. The following declarations:

```
collection DNA = seq;;
collection TUBE = bag;;
```

introduce a subtype called `DNA` of `seq` and a subtype of multisets called `TUBE`.

A restriction enzyme is represented as a rule that splits the DNA strings; for instance a rule like:

```
EcoRI = X+, ("G","A","A","T","T","C"), Y+
      ⇒ (X,"G") :: ("A","A","T","T","C",Y) :: ():TUBE ;
```

stands for the *EcoRI* restriction enzyme with recognition sequence `G^AATTC` (the point of cleavage is marked with `^`). The `X+` pattern filters the part of the `DNA` string before the recognition sequence. Identically, Y names the part of the string after the recognition sequence. The r.h.s. of the rule constructs a `TUBE` containing the two resulting `DNA` subsequences (the `::` operator indicates the "consing" of an element at the head of a sequence).

We need an additional rule `Void` for specifying that a `DNA` string without a recognition sequence must be inserted wrapped in a `TUBE`. The two rules are collected into one transformation:

```
trans Restriction = {
    EcoRI = ...;
    Void = X+ ⇒ X :: ():TUBE ;
}
```

The rule specification order in a transformation is taken into account, and so, the rule `Void` is used only if rule `EcoRI` cannot be applied. In this way, the result of applying the transformation *Restriction* on a `DNA` string is systematically a sequence with only one element which is a `TUBE`.

The transformation *Restriction* can then be applied to the `DNA` strings floating in a `TUBE` using the simple transformation:

```
trans React = { dna ⇒ hd(Restriction(dna)) }
```

The operator `hd` gives the head of the result of the transformation *Restriction*, i.e. a `TUBE` containing one or two `DNA` strings. These elements are then merged with the content of the enclosing `TUBE`. The transformation can be iterated until a fixpoint is reached :

```
React[fixpoint]((
    ("C","C","C","G","A","A","T","T","C","A","A",():DNA),
    ("T","T","G","A","A","T","T","C","G","G","G",():DNA),
    ():TUBE ));;
```

returns a tube with four DNA strings:

```
("T","T","G",():DNA),
("C","C","C","G",():DNA),
("A","A","T","T","C","A","A",():DNA),
("A","A","T","T","C","G","G","G",():DNA),
 ():TUBE
```

5.2 The Formation of a Snowflake

A crystal forms when a liquid is cooled below its freezing point. Crystals start from a seed and then grows by progressively adding more molecules to their surface. As an idealization, the molecules of a snowflake lie on an hexagonal grid and when a piece of ice is added, to the snowflake, the heat released by this process inhibits the addition of ice nearby.

This phenomenon leads to the following cellular automata rule [Wol02]: a black cell (value `1`) represents a place of the crystal filled with ice and a white cell (value `0`) is an empty place. A white cell becomes black if it has exactly one black neighbor, otherwise it remains white. The corresponding MGS transformation is:

```
trans SnowFlake = {
  0 as x / 1 == FoldNeighbor[\y.\acc.y+acc, 0](x)  ⇒  1
}
```

The construct `FoldNeighbor` is not a function but an operator available only within a rule: it enables to fold a function on the defined neighbors of an element matched in the l.h.s. Here, this operator is used to compute the number of neighbors (parameter `y` enumerates the neighbors and parameter `acc` acts as an accumulator). This transformation acts on a value of type *Hexagon* and a possible run is illustrated in figure 9.

5.3 System of Moving Cells Linked by Spring-Like Forces

We want to model the trajectory of a set of cells. A cell moves because it is attracted by its immediate neighbors (for example because the limited diffusion of a chemical that creates a gradient). The problem is that, due to the cell movements, the immediate neighbors of a cell can change. We use a Delaunay triangulation to compute the neighborhood of the cells. The Delaunay triangulation of a point set 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 value that represents a cell:

```
record Position = { x:float, y:float, z:float };;
record Cell = Position + {l};;
```

specify two record types, the first having the fields `x`, `y` and `z`, and the second having a field `l` in addition. The `l` field is used to associate an attractive force to each cell.

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

```
collection delaunay(3) D3 =
    \e.if Position(e)
       then (e.x, e.y, e.z)
       else ?("bad element type for D3 delaunay 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 check that the cell's value has a correct type and returns its coordinate (as a sequence of 3 floats).

We assume that the interaction between two cells is computed by a function of two arguments called `interaction`. Then, the following MGS program fragment:

```
epsilon = 0.05;;
fun add(u, v, e) =
  u + { x = u.x + e*v.x, y = u.y + e*v.y, z = u.z + e*v.z };;
fun sum(x, u, acc) = add(acc, interaction(x,u), epsilon);;
```

```
trans evol = {
  c ⇒
  add(c, FoldNeighbor[sum(x), {x=0,y=0,z=0,l=c.l}](c), epsilon)
};;
```

defines *evol*, the system's evolution function. The function `add` takes two records `u` and `v` and a float `e`. The result is a record containing all the fields of `u` but where the fields `x`, `y` and `z` have been updated by a linear combination of the corresponding fields in `u` and `v`. The function `sum` adds to its first argument, the interaction between two cells. This function is called in the `FoldNeighbor` construct appearing in the transformation *evol*. This transformation compute the sum of the interaction between a cell c and a neighbor cell; this sum is then used to change the state of the cell c.

The result of 250 iteration steps of this program, assuming an `interaction` function computing a force corresponding to a spring parameterized by the `l` observable, is showed at figure 10.

Delaunay collections are Leibnizian, so it is easy to extend the model to take into account cellular division and death.

6 Related Works, Current Work and Future Work

The topological approach we have sketched here is part of a long term research effort [GMS95] developed for instance in [Gia00] where the focus is on the substructure, or in [GM01a] where a general tool for uniform neighborhood definition is developed. In this research program, a data structure is viewed as a space where some computation occurs and moves in this space. The notion of neighborhood is then used to control the computations.

Related Works. Seeing a computation as a path in some abstract space is hardly new: the representation of the execution of a concurrent program as a trajectory in the Cartesian product of the sequential processes dates back to the 60's (in this representation, semaphore operations create topological obstructions and one can study the topology of theses obstructions to decide if a deadlock may occur). However, note that the considered space is based on the control structure, not on the involved data structure.

In the same line, the methods for building domains in denotational semantics have clearly topological roots, but they involve the *topology of the set of values*, not the *topology of a value*.

Transformation on multiset is reminiscent of multiset-rewriting (or rewriting of terms modulo AC). This is the main computational device of `Gamma` [BM86, BCM87], 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 [BB90]. 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 [Pau98, Pau01] are a new distributed parallel computing model based on the notion of a membrane structure. A membrane structure is a nesting of cells represented, e.g, by a Venn diagram without intersection and with a unique superset: the skin. Objects are placed in the regions defined by the membranes and evolve following various transformations: an object can evolve into another object, can pass trough a membrane or dissolve its enclosing membrane. As for `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[6].

Lindenmayer systems [Lin68] 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 [PLH$^+$90, PH92]). Obviously, L systems are dedicated to the handling of linear and tree-like structures.

There exists strong links between GBF and cellular automata (CA), especially considering the work of Z. Róka which has studied CA on Cayley graphs [Rók94]. However, our own works focus 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.

Formalizations and Implementations. A unifying theoretical framework can be developed [GM01b, GM02b], based on the notion of *chain complex* developed in algebraic combinatorial topology [Hen94]. The topology needed to describe the neighborhood in a set or a sequence, or more generally the topology of the usual data structures, are fairly poor. However, the topological framework unifies various situations (see the paragraph above). Nevertheless, 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 several formalisms (CA, L systems, P systems, etc.) *for programming* purposes. All the examples presented here are running with one or the other of the two existing MGS interpreters. A new version of the interpreter is currently developed, written in `OCAML` (a dialect of `ML`): please visit the MGS home page: `http://mgs.lami.univ-evry.fr`.

Perspectives. The perspectives opened by this preliminary work are numerous. We want to develop several complementary approaches to defines new topological collection types. One approach to extend the GBF applicability is to consider monoids instead of groups, especially automatic monoids which exhibit good algorithmic properties. Another direction is to handle general combinatorial spatial

[6] For example the rule "x`+ /n==length(`x`) ⇒ ?(`x`)`" can be used on a graph with `n` vertices to print an Hamiltonian path (function `?` print its argument and function `length` gives the length of a sequence).

structures like simplicial complexes or G-maps [Lie91]. At the language level, the study of the topological collections concepts must continue with a finer study of transformation kinds. Several kinds of restriction can be put on the transformations, leading to various kind of pattern languages and rules. The complexity of matching such patterns has to be investigated. The efficient compilation of a MGS program is a long-term research. We have considered in this paper only one-dimensional paths, but a general n-dimensional notion of path exists and can be used to generalize the substitution mechanisms of MGS. From the applications point of view, we are targeted by the simulation of more complex developmental processes in biology [GGMP02].

To conclude, I want to promote the use of topological notions in computer science. The work sketched here is a modest step in this direction. I use the qualifier modest because the notions used here rely on very elementary notions taken in the domain of *combinatorial algebraic topology*. We do not use deep theorems but rather fundamental definitions that structure the field and clarify the objects and mechanisms to manage. This is why I want to advocate the development of alternative topological approaches of computation, confident on their heuristic, technical and pedagogical virtues.

6.1 Acknowledgments

I would like to reiterate my gratitude for Olivier Michel: our disputes are always fertile, frustrating and insightful. I am also grateful to P. Prusinkiewicz at University of Calgary, C. Godin at CIRAD Montpellier and F. Delaplace, J. Cohen, P. Legall, G. Bernot at LaMI, and the members of the "Simulation and Epigenesis" group at GENOPOLE-Evry for fruitful discussions, biological motivations, warm encouragements and challenging questions. The delightful questions and encouragements of D. Blasco have also raised many issues and motivated many developments of this work.

Many parts of this presentation have been developed in the following papers [GM01b, GM01c, GM01a, GM02b, GGMP02, GMC02, GM02a]. This research is supported in part by the CNRS, the GDR ALP, IMPG, GENOPOLE and the University of Evry.

References

[ACD02] J.J. Arulanandham, C.S. Calude, and M.J. Dinneen. Bead-sort: A natural sorting algorithm. *Bulletin of the European Association for Theoretical Computer Science*, 76:153–162, February 2002. Technical Contributions.

[BB90] 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.

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

[BLL97] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*, volume 67 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 1997. isbn 0-521-57323-8.

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

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

[Ede58] M. Eden. In H. P. Yockey, editor, *Symposium on Information Theory in Biology*, page 359, New York, 1958. Pergamon Press.

[EKL+02a] S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and J. Sonmez. Pathway logic: Symbolic analysis of biological signaling. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 400–412, January 2002.

[EKL+02b] Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, , and Carolyn Talcott. Pathway logic: Executable models of biological networks. In *Fourth International Workshop on Rewriting Logic and Its Applications (WRLA'2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[FM97] P. Fradet and D. Le Métayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

[FMP00] Michael Fisher, Grant Malcolm, and Raymond Paton. Spatio-logical processes in intracellular signaling. *BioSystems*, 55:83–92, 2000.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, Florida, 21–24 January 1996.

[GG95] D. Gautier and C. Germain. A static approach for compiling communications in parallel scientific programs. *Scientific Programming*, 4:291–305, 1995.

[GGMP02] 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.

[Gia00] 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, September 2000. ACM-press.

[GJ92] E. Goubault and T. P. Jensen. Homology of higher-dimensional automata. In *Proc. of CONCUR'92*, Stonybrook, August 1992. Springer-Verlag.

[GM01a] 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, September 2001.

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

[GM01c] Jean-Louis Giavitto and Olivier Michel. Mgs: a rule-based programming language for complex objects and collections. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[GM02a] 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, October 2002. Lecture Notes in Computer Science.

[GM02b] J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.

[GMC02] J.-L. Giavitto, O. Michel, and J. Cohen. Pattern-matching and rewriting rules for group indexed data structures. Technical Report 76-2002, LaMI – Université d'Évry Val d'Essonne, June 2002.

[GMS95] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. Jr. 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 October 1995. Springer-Verlag.

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

[Jam93] Max Jammer. *Concepts of space – the history of theories of space in physics*. Dover, 1993. third enlarged edition (first edition 1954).

[Jay95] C. Barry Jay. A semantics of shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

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

[Lin68] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[Lis93] B. Lisper. On the relation between functional and data-parallel programming languages. In *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures*. ACM, ACM Press, June 1993.

[Man01] Vincenzo Manca. Logical string rewriting. *Theoretical Computer Science*, 264:25–51, 2001.

[MFP91] 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.

[NO94] Susumu Nishimura and Atsushi Ohori. A calculus for exploiting data parallelism on recursively defined data (Preliminary Report). In *International Workshop TPPP '94 Proceedings (LNCS 907)*, pages 413–432. Springer-Verlag, November 94.

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

[Pau01] G. Paun. From cells to computers: Computing with membranes (p systems). *Biosystems*, 59(3):139–158, March 2001.

[PH92] 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, February 1992.

[PLH+90] P. Prusinkiewicz, A. Lindenmayer, J. S. Hanan, et al. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[Rók94] Zsuzsanna Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 September 1994.

[Wol02] Stephen Wolfram. *A new kind of science.* Wolfram Media, 2002.

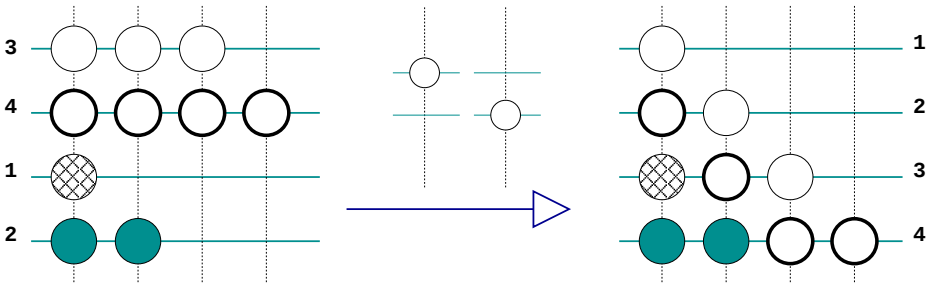

Fig. 1. Bead sort is a new sorting algorithm [ACD02]. The idea is to represent positive integers by a set of beads, like those used in an abacus. Beads are attached to vertical rods and appear to be suspended in the air just before sliding down (a number is read horizontally, as a row). After their falls, the rows of numbers have been rearranged such as the smaller numbers appears on top of greater numbers. The corresponding one-line MGS program is given in section 4.4.

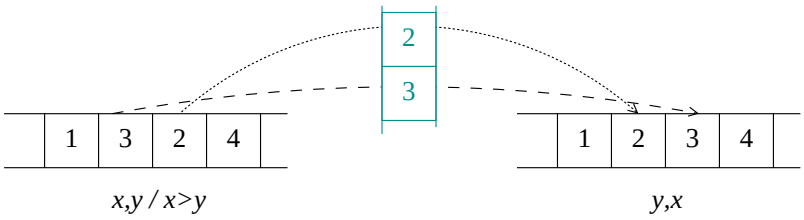

Fig. 2. A kind of bubble-sort is immediate in MGS; it is sufficient to specify the exchange of two non-ordered adjacent elements in a sequence. The corresponding one-line MGS program is given in section 4.4. This is not really bubble-sort because swapping of elements can take at arbitrary places; hence an out-of-order element does not necessarily bubble to the top in the characteristic way.)

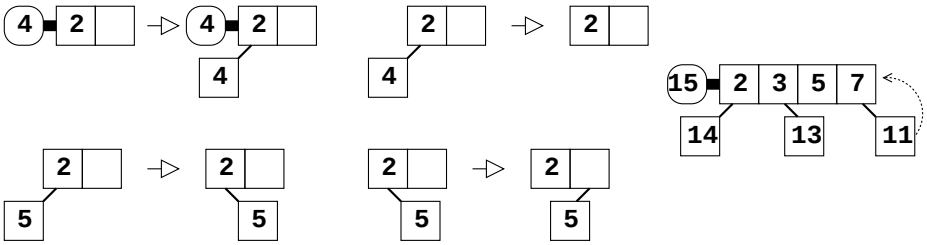

Fig. 3. Erastothene's sieve. The successive natural numbers are generated by the first cells (round box) and travel along a sequence of cells containing the previous prime number (square box). If a traveling number is divisible by the number in a cell, it is erased, else, it is passed to the right neighbor. When a number reach the end of the sequence, it becomes a new cell extending the sequence.

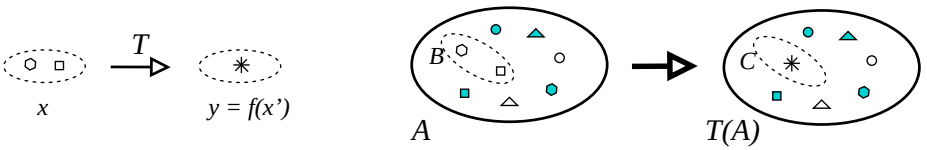

Fig. 4. *A local transformation of a topological collection.* Collection A is of some kind (set, sequence, array, cyclic grid, tree, term, etc). A rule T specifies that a subcollection B of A has to be substituted by a collection C computed from B. The right hand side of the rule is computed from the subcollection matched by the left hand side x and its possible neighbors x' in the collection A.

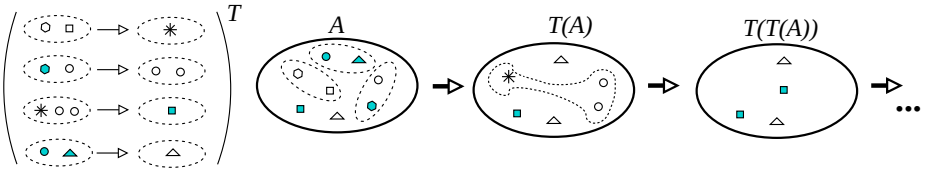

Fig. 5. *Transformation and iteration of a transformation.* A global transformation T is a set of local transformations applied in parallel and synchronously to make one evolution step. The local transformations do not interact together. A transformation can then be iterated.

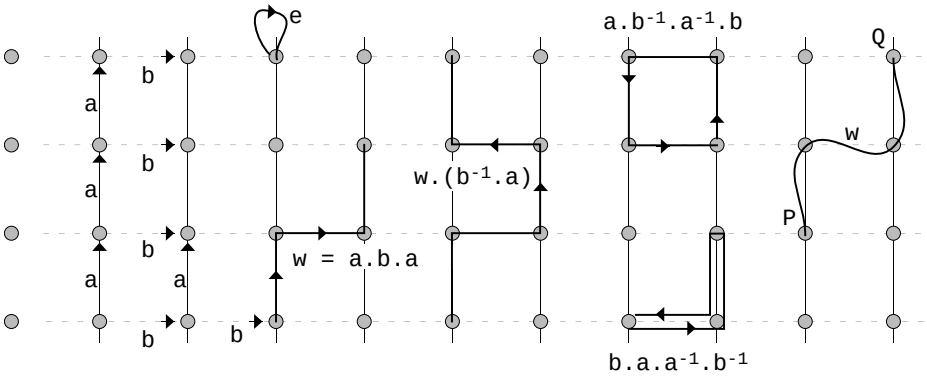

Fig. 6. Graphical representation of the relationships between Cayley graphs and group theory. A vertex is a group element. The label a of an edge corresponds to the generator a of the group. There is an edge between vertices P and Q labelled by a iff $P.a = Q$. A word (a product of generators) can be seen a path. Starting from vertex P, a path w ends in $P.w$. Path composition corresponds to word multiplication. A closed path (a cycle) is a word equal to e (the identity of the multiplication). An equation $v = w$ can be rewritten $v.w^{-1} = e$ and then corresponds to a cycle in the graph. There are two kinds of cycles in the graph: the cycles that are present in all Cayley graphs and corresponding to group laws (intuitively: a backtracking path like $b.a.a^{-1}.b^{-1}$) and closed paths specific to the own group equations (e.g.: $a.b^{-1}.a^{-1}.b$). The graph connexity (there is always a path going from P to Q) is equivalent to say that there is always a solution x to equation $P.x = Q$.

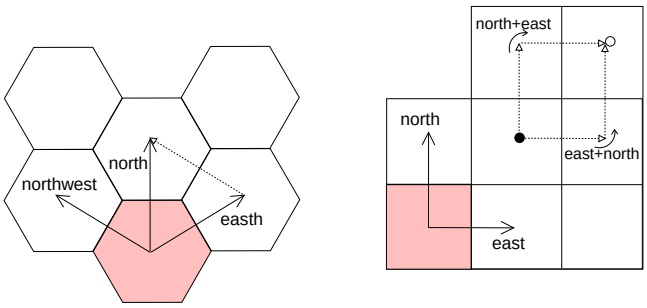

Fig. 7. These shapes correspond to a Cayley graph of *Hexagon* and *Grid2* with the following conventions: a vertex is represented as a face and two neighbors in the Cayley graphs share an edge in this representation. An empty cell has an undefined value. Only a part of the infinite domain is figured.

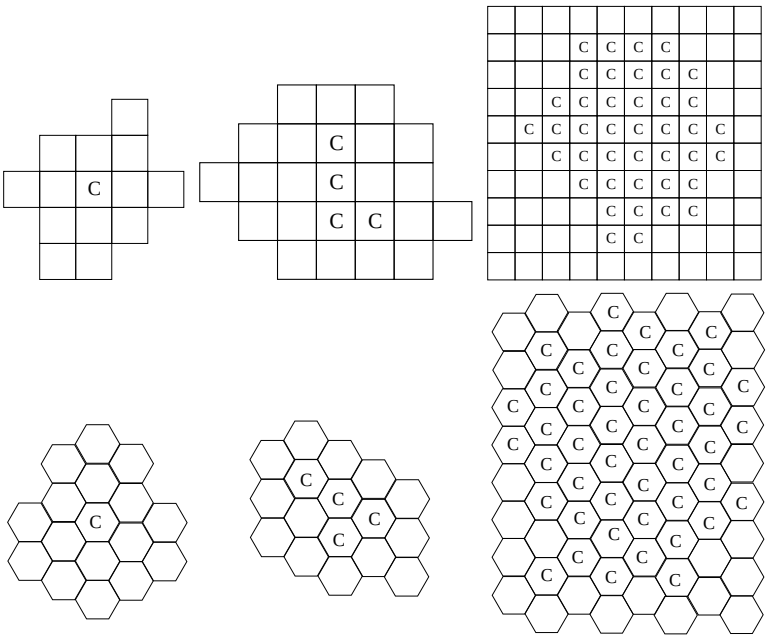

Fig. 8. Eden's model on a grid and on an hexagonal mesh (initial state, and states after the 3 and the 7 time steps). The Eden's aggregation process is a simple model of growth. The model has been used since the 1960's as a model for such things as tumor growth and growth of cities. In this model (specifically, a type B Eden model [Ede58]), 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.

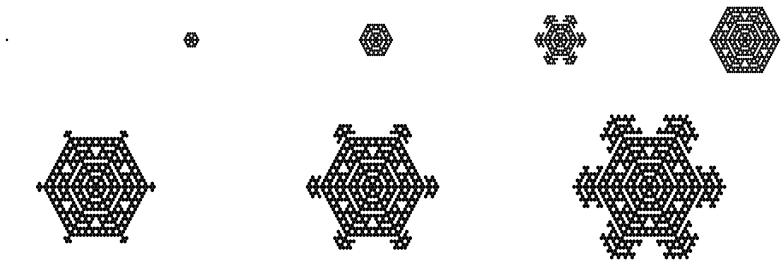

Fig. 9. Formation of a snowflake. See section 5.2 for the explanation. The transformation acts on a GBF *Hexagon* (cf. sec. 3). The pictured states are the step at time steps 1, 4, 8, 12, 16, 18, 20 and 23.

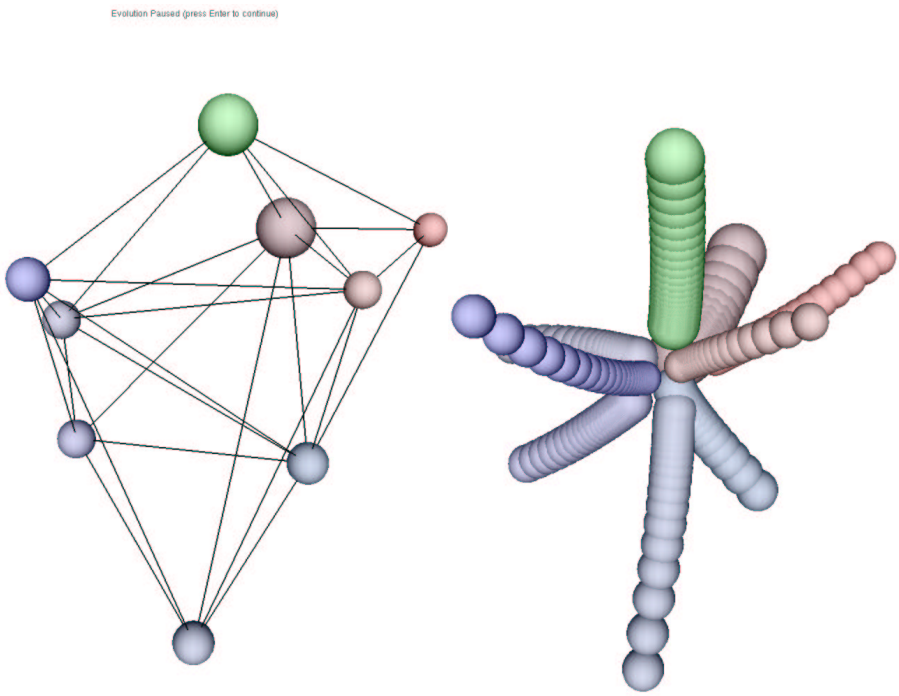

Fig. 10. Each sphere in the picture above corresponds to a cell attracted by its neighboring cells by a spring. The neighborhood of a cell is computed dynamically using a Delaunay triangulation built from the cells position. At each time step, this neighborhood can change. The first picture is the initial 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.