Data Structure as Topological Spaces

Jean-Louis Giavitto and Olivier Michel

LaMI, umr 8042 du CNRS, Université d'Evry – GENOPOLE
523 Place des terasses de l'agora, Tour Evry-2
91000 Evry, France
`{giavitto,michel}@lami.univ-evry.fr`

Abstract. In this paper, we propose a topological metaphor for computations: computing consists in moving through a path in a data space and making some elementary computations along this path. This idea underlies an experimental declarative programming language called `MGS`. `MGS` introduces the notion of *topological collection*: a set of values organized by a neighborhood relationship. The basic computation step in `MGS` relies on the notion of *path* : a path C is substituted for a path B in a topological collection A. This step is called a *transformation* and several features are proposed to control the transformation applications. By changing the topological structure of the collection, the underlying computational model is changed. Thus, `MGS` enables a unified view on several computational mechanisms. Some of them are initially inspired by biological or chemical processes (Gamma and the CHAM, Lindenmayer systems, Paun systems and cellular automata).

Keywords. Topological collection, declarative and rule-based programming language, rewriting, Paun system, Lindenmayer system, cellular automata, Cayley graphs, combinatorial algebraic topology.

1 Introduction

Our starting point is the following intuitive meaning of a data structure: a data structure s is an *organization* o performed on a data set D. 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.) and to use set theoretic constructions to specify o. However, here, we want to stress the structure o as a set of *places* or *positions*, independently of their occupation by elements of D. Following this perspective, a data structure in [Gia00] is a function from a set of positions to a set of values: this is the point of view promoted by the *data fields* approach. Data fields have been mainly focussed on arrays and therefore on $\mathbb{Z}^n$ as the set of positions [Lis93]. One of our motivations is to define in the same framework the set of positions representing a tree, an array or a multiset independently of the set of values.

Data Structure and Neighborhood. 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*. This topological approach relies on the notion of *neighborhood* to specify a move from one position to one of its neighbor. Although speaking of neighborhood in a data structure is not usual, the relative accessibility from one element to another is a key point considered in a data structure definition:

1. In a simply linked list, the elements are accessed linearly (the second after the first, the third after the second, etc.).
2. In a circular buffer, or in a double-linked list, computation goes from one element to the following *or* to the previous one.
3. From a node in a tree, we can access the sons.
4. The neighbors of a vertex V in a graph are visited after V when traveling through the graph.
5. In a record, the various fields are locally related and this localization can be named by an identifier.
6. 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. And 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.

Elementary Shifts and Paths. 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. For example, the recursive definition of the `fold` function on lists propagates an action to be performed from the the tail to the head of the 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 others polytypic functions on inductive types [MFP91]).

These considerations lead to the idea of *path*: 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. The move from e to e' is called 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].

Paths and Computations. At this point we can summarize our presentation: we assume that a computation induces a path in a space defined by the neighborhood relationship 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 our 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.

Organization of the paper. In section 2 we introduce the `MGS` programming language. `MGS` is used as a vehicle to experiment our topological ideas. We start by the definition of several types of topological collections. The notions underlying the selection of a path and path substitution are then sketched. Section 3 illustrates the previous constructions with two examples taken from the domain of molecular computing and cellular automata. All examples given are real `MGS` programs running on top of one or the other of the two available interpreters. In the last section, we review some related works and some perspectives opened by this research.

2 The MGS Programming Language

The topological approach sketched in section 1 is investigated through an experimental declarative programming language called `MGS`. `MGS` is aimed at the representation and manipulation of local transformations of entities structured by *abstract topologies* [GM01c, GM02]. A set of entities organized by an abstract topology is called a *topological collection*. Topological means here that each collection type defines a neighborhood relation specifying both the notion of *locality* and the notion of *sub-collection*. A sub-collection B of a collection A is a subset of elements of A defined by some path and inheriting its organization from A. 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 sub-collection. The application of a a rewrite rule $r = \beta \Rightarrow f(\beta, ...)$ to a collection A:

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

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.

2.1 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 GBFs [GMS95, Mic96, GM01a] (other topologies are currently under development and include Voronoï partitions and abstract simplicial complexes). This paper focuses on two families of collection types: *monoidal collection* and *GBF*.

For any collection type T, the corresponding empty collection is written ():T. 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. Each collection type can be subtyped:

```
collection U = T;;
```

introduces a new collection type U, which is a subtype of T. These two types share the same topology but a value of type U can be distinguished from a value of type T by the predicate U. Elements in a collection T can be of any type, including collections, thus achieving *complex objects* in the sense of [BNTW95].

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 taken in V can be seen as an element of the free monoid V^* (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 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 one 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). So, the expression `1, 1+1, 2+1, ():set` builds the set with the three elements $1, 2$ and 3, while the expression `1, 1+1, 2+1, ():seq` makes a sequence s with the same three elements. The comma operator is overloaded such that if x and y are not monoidal collections, then x`,`y builds a sequence of two elements. So, the expression `1, 1+1, 2+1` evaluates to the sequence s too.

Group-Based Data Field. Group-based data fields (GBF in short) are used to define organizations with *uniform* neighborhood. A GBF is an extension of the notion of array, where the elements are indexed by the elements of a group, called the *shape* of the GBF [GMS95, GM01a]. For example:

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

defines a gbf collection type called `Grid2`, corresponding to the Von Neuman neighborhood in a classical array (a cell above, below, left or right – not diagonal). 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 right hand side (r.h.s.) of the GBF definition gives a finite presentation of the group structure. The list of the generators can be completed by giving equations that constraint the displacements in the shape:

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

defines an hexagonal lattice that tiles the plane, see. figure 1. Each cell has six neighbors (following the three generators and their inverses). The equation `east + north = northeast` specifies that a move following `northeast` is the same has a move following the `east` direction followed by a move following the `north` direction.

A GBF value of type `T` is a partial function that associates a value to some group elements (the group elements are the positions of collection and the the empty GBF is the everywhere undefined function). The topology of `T` is easily visualized as the Cayley graph of the presentation of `T`: each vertex in the Cayley graph is an element of the group and vertices x and y are linked if there is a generator `g` in the presentation such that $x + \mathtt{g} = y$.

A presentation starting with `<` and ending with `>` introduces an *Abelian* organization: they are implicitly completed with the equations specifying the commutation of the generators `g+g' = g'+g`. Currently only free and Abelian groups are allowed: 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).

2.2 Matching a Path

Path patterns are used in the left hand side (l.h.s) of a rule to match a sub-collection to be substituted. We give only a fragment of the grammar of the patterns:

$$Pat ::= \texttt{x} \;\mid\; \texttt{<undef>} \;\mid\; p\,\texttt{,}\,p' \;\mid\; p\,\texttt{|g>}\,p' \;\mid\; p\,\texttt{*} \;\mid\; p/exp \;\mid\; p\;\texttt{as}\;\texttt{x}$$

where p, p' are patterns, `g` is a GBF generator, `x` ranges over the pattern variables and *exp* is an expression evaluating to a boolean value.

Informally, a path pattern can be flattened into a sequence of basic filters and repetition specifying a sequence of positions with their associated values. The order of the matched elements can be forgotten to see the result of the matching as a sub-collection. A pattern variable `x` matches exactly one element (somewhere in the collection) and the identifier `x` can be used in the rest of the rule to denote the value of the matched element. More generally, the naming of the value of a sub-path is achieved using the construction `as`. The constant `<undef>` is used to match an element with an undefined value (i.e., a position with no value). The pattern p`,`p' stands for a path beginning like p and ending like p' (i.e., the last element in path p must be a neighbor of the first element in path p'). For example, `x,y` matches two connected elements (i.e., `y` must be a neighbor of `x`). The neighborhood relationship depends of the collection kind and is decomposed in several sub-relations in the case of a GBF. The comma operator is then refined in the construction p `|g>` p': the first element of p' is the `g`-neighbor of the last element in path p. The pattern p`*` matches a (possibly empty) repetition $p, \ldots, p$ of path p. Finally, p/exp matches the path p only if *exp* evaluates to true. For example

```
(s/seq(s))+ as S / size(S) == 5
```

selects a sub-collection `S` of size 5, each element of `S` being a sequence. If this pattern is used against a set, S is a subset, if this pattern is used against a sequence, S is a sub-sequence (that is, an interval of contiguous elements), etc.

2.3 Path Substitution and Transformations

There are several features to control the application of a rule: rules may have priority or a probability of application, they may be guarded and depend on the value of local variables, they "consume" their arguments or not, . . . , see [GM01b] for more details.

Substitutions of Sub-collections. A rule $\beta \Rightarrow c$ can be seen as a rule for substituting a path or a sub-collection (recall that a path can be seen as a sub-collection by simply forgetting the order of the elements in the path). For example the rule

```
(x / x<3)+ as S ⇒ 3,4,5,():set
```

applied to the set `1,2,3,4,():set` returns the set `3,4,5,():set` because `S` matches the subset `1,2,():set` and is replaced by the set `3,4,5,():set`. The final result is computed as (`3,4,():set`) ∪ (`3,4,5,():set`).

Substitutions of Paths. Because the matched sub-collection is also a path, that is a sequence of elements, the `seq` type has a special role when appearing in the r.h.s. of a rule. If the r.h.s. evaluates to a sequence, and if this sequence has the same length as the matched path, then the first element of the sequence is used to replace the first element of the matched path, and so on. This convention is coherent with the sub-collection substitution point of view and simplifies the building of the r.h.s.

For example, suppose that in a GBF of type `Grid2`, we want to model the random walk of a particle `x`. Then, two neighboring elements, one being `x` the other undefined, must exchange their values. This is achieved with only one simple rule

```
x, <undef>  ⇒  <undef>, x
```

without the need to mention the precise neighborhood relationships between the two elements.

Newtonian and Leibnizian Collections. We have mentioned above that the result of replacing a sub-set by a set is computed using set union. More generally, the insertion of a collection C in place of a sub-collection B depends on the "borders" of the involved collections. For example, in a sequence, the sub-collection B defines in general two borders which are used to glue the ends of collection C. The gluing strategy may admit several variations. The programmer can select the appropriate behavior using the rule's attributes.

We discuss here only the *flattening/nesting behavior* linked with the *Leibnizian/Newtonian kind* of the involved collection. Consider the rule:

```
x ⇒ x, x
```

Intuitively, it defines the substitution of one element by two copies of it. However the evaluation of the r.h.s. gives a couple and then, there are two possibilities to replace `x`: one may replace the element matched by `x` by one element which is a couple, *or*, one may "merge" the couple in place of `x` preserving the neighborhood of `x`. For example, if this rule is used on the sequence `1,2,3`, the first interpretation gives the result `(1,1), (2,2), (3,3)` (a sequence of sequences of integers) and the second interpretation returns `1,1,2,2,3,3` (a flat sequence of integers).

The two possibilities, exemplified here for a sequence, hold for any monoidal collection. For a GBF, e.g. `Grid2`, this rule has no meaning, because we cannot insert arbitrary positions between two others without changing the topology of `Grid2`. The set of positions of a GBF exists independently of the values involved in the collection. GBF are *Newtonian* space: the positions exist *a priori* and can be occupied or left empty by the values. In the opposite, monoidal collections have a *Leibnizian* character in the sense that their topology exist only as a relation between the actual values. A consequence is that there is no position with an undefined value in a Leibnizian collection.

Transformations. A transformation `R` is a set of rules:

```
trans R = {  ...   rule;    ... }
```

For example, the transformation `trans Mf = { x ⇒ f(x); }` defines a function `Mf` similar to the `map(f)` operator. The expression `Mf(c)` denotes the application of one transformation step to the collection c and a transformation step consists in the parallel application of the rules (modulo the rule application's features). Thus `Mf(c)` computes a new collection where each element e of collection `c` is replaced with `f(`e`)`. Transformations may have parameters, which enables, e.g., the writing of a generic `map`: the transformation `trans M[fct] = { x ⇒ fct(x); }` requires an additional argument when applied. The arguments between brackets are passed to the transformation using a name as in [GAK94]. So, expression `M[fct=\x.x+1](c)` returns a collection where each element of`c` is increased by one. This transformation is polytypic in the sense that it can be applied to any collection type. A transformation step can be easily iterated:

`T[iter=n](c)`	denotes the application of n transformation steps
`T[iter=fixpoint](c)`	application of `T` until a fixpoint is reached
`T[iter=fixrule](c)`	the fixpoint is detected when no rule applies

3 Examples

Because the lack of space, we present here only two simple examples. However, more examples can be found in [GM01b, GM01c, GGMP02] including the tokenization of a sequence of letters, the Eratosthene's sieve, primitive recursion operators on sequences and sets, the computation of the convex hull of a set of points, the maximal segment sum and some other optimization problems, the computation of the disjunctive normal form of a logical formula, direct coding of Lindenmayer systems and Paun systems, Turing-like diffusion-reaction processes, the simulation of a spatially distributed biochemical interaction networks, examples in population dynamics, paradigmatic examples in the field of artificial chemistry and cellular automata, etc.

3.1 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+ as X,
         (cut+ as CUT / CUT = "G","A","A","T","T","C",():DNA),
         y+ as Y
       ⇒ (X,"G") :: ("A","A","T","T","C",Y) :: ():TUBE ;
```

corresponds to 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 and the result is named X (the `+` operator denotes repetition of neighbors). 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 with 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+ as X ⇒ X :: ():TUBE ;
}
```

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`. Note that the rule `Void` is applied only when the rule `EcoRI` cannot be applied.

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

3.2 The Eden Model

We start with a simple model of growth sometimes called the Eden model (specifically, a type B Eden model [YPQ58]). 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, a 2D space is partitioned in empty or occupied cells (we use the value `true` for

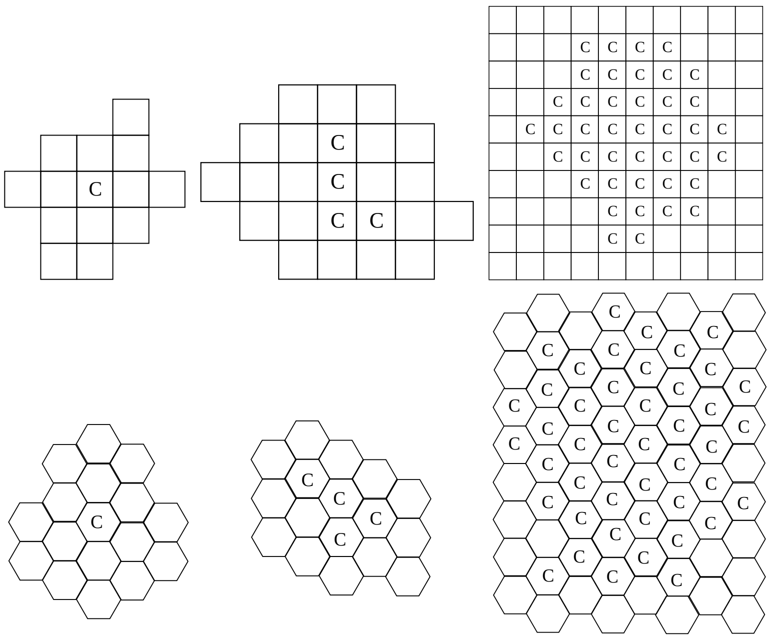

Fig. 1. Eden's model on a grid and on an hexagonal mesh (initial state, and states after the 3 and the 7 time steps). *Exactly the same* transformation is used for both cases. These shapes correspond to a Cayley graph of `Grid2` and `Hexagon` whit the following conventions: a vertex is represented as 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.

an occupied cell and left undefined the unoccupied 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.

The Eden's aggregation process is simply described as the following transformation:

```
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. The transformation `Eden` defines a function that can then be applied to compute the evolution of some initial state. One of the advantages of the `MGS` approach, is that this transformation can apply indifferently on grid or hexagonal lattices, or *any* other collection kind.

It is interesting to compare transformations on GBFs with the genuine cellular automata (CA) formalism. There are several differences. The notion of GBF extends the usual square grid of CA to more general Cayley graphs. The value of a cell can be arbitrary complex (even another GBF) and is not restricted to take a value in a finite set. Moreover, the pattern in a rule may match an arbitrary domain and not only one cell as it is usually the case for CA. For example the transformation:

```
gbf G2 = <X, Y >;;
trans Turn = { a |X> b |Y-X> c |-X-Y> d |X-Y> e ⇒ a,e,b,c,d; }
```

specify the 90°-turn of a cross in GBF `G2` (see illustration 2). The pattern fragment `b |Y-X> c` specifies that `c` is at the north-west of element `b` if we take the `X` dimension as the *east* direction and the `Y` dimension as the *north* direction.

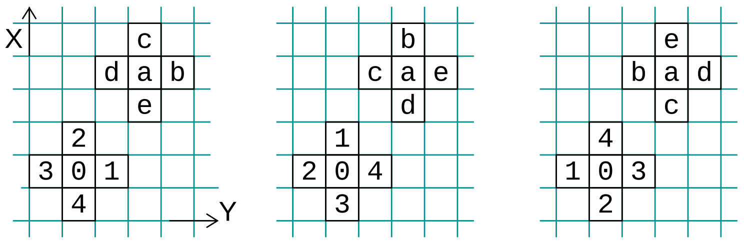

Fig. 2. First and second iteration of transformation `Turn` on the GBF to the left (only defined values are pictured). In contrast with cellular automata, the evolution of a multi-cell domain can be easily specified by a single rule.

4 Related and Future Work

This topological approach formalizing the notion of collection is part of a long term research effort [GMS95] developed for instance in [Gia00] where the focus is on the substructure and in [GM01a] where a general tool for uniform neighborhood definition is developed.

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 sixties(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 elementary computations, 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*.

Another example of topological inspiration is the approach taken in [FM97], that rephrased in our perspective, uses a *regular language* to model the displacements resulting from following pointers in `C` data structures.

There exists strong links between GBF and cellular automata, 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.

Obviously, Lindenmayer systems [Lin68] correspond to transformations on sequences, and basic Paun systems [Pau00] can be emulated using transformations on multisets.

Formalizations and Implementations. A unifying theoretical framework can be developed [GM01b, GM02], 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 cited 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.

Currently, two versions of an `MGS` interpreter exist: one written in `OCAML` (a dialect of `ML`) and one written in `C++`. There are some slight differences between the two versions. For instance, the `OCAML` version is more complete with respect to the functional part of the language. These interpreters are freely available (see url `http://www.lami.univ-evry.fr/mgs`).

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 exhibits good algorithmic properties. Another direction is to handle general combinatorial spatial 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 developmental processes in biology [GGMP02]. Another motivating application is the case of a spatially distributed biochemical interaction networks, for which some extension of rewriting as been advocated, see [FMP00].

Acknowledgments. The authors would like to thanks the members of the "Simulation and Epigenesis" group at Genopole for fruitful discussions and biological motivations. They are also grateful to C. Godin, P. Prusinkiewicz, F. Delaplace and J. Cohen for numerous challenging questions and useful comments. This research is supported in part by the CNRS, the GDR ALP, IMPG and Genopole/Evry.

References

[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.

[FM97] P. Fradet and D. Le Mtayer. 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 signalling. *BioSystems*, 55:83–92, 2000.

[GAK94] Jacques Garrigue and H. At-Kaci. The typed polymorphic label-selective lambda-calculus. In *Principles of Programming Languages*, Portland, 1994.

[GGMP02] J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz. *Biological Modeling in the Genomic Context*, chapter Computational Models for Integrative and Developmental Biology. Hermes, July 2002.

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

[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. 85p.

[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.

[GM02] J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 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 *Lecture Notes in Computer Sciences*, pages 209–215, Beaune (France), 2–4 October 1995. Springer.

[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.

[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.

[Mic96] O. Michel. *Reprsentations dynamiques de l'espace dans un langage dclaratif de simulation.* PhD thesis, Universit de Paris-Sud, centre d'Orsay, December 1996. N°4596, (in french).

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

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

[YPQ58] Hubert P. Yockey, Robert P. Platzman, and Henry Quastler, editors. *Symposium on Information Theory in Biology*. Pergamon Press, New York, London, 1958.