Integration and pattern-matching of topological structures in a functional language

Antoine Spicher and Olivier Michel

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

Abstract. MGS is an experimental programming language dedicated to the manipulation of topological collections through rules of transformations. A topological collection is a set of elements organized by a neighborhood relation. This topological approach enables the extension of the case based definition of function to several non-algebraic data types. In this paper, we show how sophisticated data structures used to build geometric objects can be embedded smoothly in this framework. The expressivity of the language is then illustrated by some examples like the simulation of a diffusion limited aggregation process on complex geometries and the computation of the maximal flow on a graph.

1 Introduction

The MGS project aims at developing a programming language dedicated to the modelisation and the simulation of processes with a dynamical structure.

Dedicated languages provide programmer with abstractions and notations suitable for a particular applications field. They often are declarative and based on a little kernel. Their specialization makes them more attractive than a general language and enables a better productivity by making programmation and reusability easier.

The processes we are interested in are highly structured dynamical systems with a hierarchical organization. Such a system can be often characterized by a state specified by the association of some attributes (mass, temperature, charge, etc.) to some spatial structure.

The basic idea of MGS is to allow the definition of such states as a new kind of data field [Lis93] in a functional language. They are called *topological collections* to outline the underlying topological concepts. The evolution function of dynamical systems corresponds to a function that transforms a topological collection into another one. The specification of such transformations is greatly simplified using case-based definition through a powerful pattern-matching language. It appears that the notions developed for the simulation of dynamical system with a dynamical structure (also called $(DS)^2$) enable a new programming style and allow the very concise expression of various basic algorithms on set, sequence, graph, etc.

In this paper, after a brief presentation of the MGS experimental programming language, we extend a proposal made by J.-F. Dufourd and S. Luther [DL02] to parameterize geometrical objects via the λ-calculus. This enables us to expressively build and compose geometrical objects as a new kind of topological collections. Then, we illustrate through several examples the use of transformations to model several processes located on these complex objects.

2 A Brief Presentation of the MGS Language

MGS embeds a complete, impure, dynamically typed, strict, functional language. We focus on the notions required to understand the rest of the paper and we only describe here the major differences between the constructions available in MGS with respect to standard functional languages like OCAML [Ler04].

2.1 Atomic values.

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

2.2 Topological Collections.

The distinctive feature of the MGS language is its handling of entities structured by *abstract topologies* using *transformations* [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 (the topology depends on the properties of that relation) inducing a notion of *subcollection*. A subcollection S' of a collection S is a subset of connected elements of S and inheriting its organization from S.

Different predefined and user-defined collection types are available in MGS. In this paper we won't detail them. We will only introduce to the two collections required by the examples given below. More details will be specified later:

– Monoidal collections: the neighborhood relation is defined through a monoid. The main operation of the monoid is the insertion of an element and is denoted by a comma; if the comma is associative, sequences are built (each element has two neighbors except the extremities); if the comma is associative and commutative, bags are built (each element is neighbor of all the others); if the comma is idempotent, sets are built.

– Delaunay graphs: they are unoriented graphs. Values from an Euclidean space $\mathbb{R}^n$ are associated to their vertices. The neighborhood is computed from this embedding and the tessellation of Delaunay. This tessellation divides the space into cells surrounding each element of the graph. For a cell of a given vertex, the points of the cell are closer to the vertex than to the other vertices. This collection enables the representation of a neighborhood corresponding to the notion of "closer neighbor".

The current prototype of `MGS` provides, among other, topological collections like arbitrary graphs, Cayley's graphs (especially partition of plane and n dimensional grids, circular or not) and *quasi-manifolds* as we will see.

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

2.3 Transformations

Transformations are used to specify the evolution function of a $(\text{DS})^2$. The *global transformation* of a topological collection s consists in the *parallel application* of a set of *local transformations*. A local transformation is specified by a rule r that defines the replacement of a subcollection by another one. The application of a rewriting rule $\sigma \Rightarrow f(\sigma, ...)$ to a collection s:

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

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

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

Path Pattern. A pattern σ in the left hand side of a rule specifies a subcollection where an interaction occurs. A subcollection of interacting elements can have an arbitrary shape, making it very difficult to specify. Thus, it is more

convenient (and not so restrictive) to enumerate sequentially the elements of the subcollection. Such enumeration will be called a *path*.

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

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

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

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

empty element symbol `<undef>` matches an element with an undefined value, that is, an element whose position does not have an associated value.

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

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

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

iteration: $p*$ matches a subcollection composed of the repetition of subcollections matched by p.

alias: p *as* x allows to refer to the path matched by p through the name x. The elements filtered by the atomic patterns of a rule are different. Thus a path cannot have self-intersection.

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

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

For some collection types it is possible to replace a subcollection by a collection with a different shape. Such collections are termed as *leibnizian* and are opposed to *newtonian* collections. Examples of leibnizian collections include sets,

bags, sequences, Delaunay's graphs... A two-dimensional grid is an example of a newtonian collection: one cannot replace an arbitrary subset of a grid by a subset with another shape without destroying the 2D grid neighborhood.

2.4 Short Examples.

We give three examples that imply the transformation of a sequence of elements.

Bubble Sort in MGS. The bubble sort consists in

1. comparing two neighbors elements in a sequence and swapping them if they are not in order,
2. repeating the first step until a fixed point is reached.

This specification is immediate in `MGS` and can be written:

```
trans bubble_sort = {      x, y / (x > y) => y, x      }
```

The keyword `trans` introduces the definition of a transformation by a set of rules. Here there is only one rule. This transformation can be applied to a sequence `s := 4, 2, 3, 1` until a fixed point is reached: `bubble_sort['iter='fixpoint](s)`. The value of the predefined optional parameter `'iter` indicates that the application of the function `bubble_sort` must be iterated until a fixed point is reached. The results is `1, 2, 3, 4` as expected.

Hamiltonian Path. A graph is an `MGS` topological collection. It is very easy to find an Hamiltonian path in a graph using the following transformation:

```
exception Not_Found, Found;;
trans H = {
    x* / size(x) = size(self) => raise Found(x)
};;
fun hamilton(g) = try
                      H(g); raise Not_Found
                  with Found path -> path;;
```

Transformation H uses an iterated pattern x`*` that matches a path (a sequence of elements that are neighbors two by two). Keyword `self` refers to the collection which the transformation is applied onto, that is, the entire graph. The size of a graph returns the number of its vertices. So, if the length of path x is the same as the number of vertices in the graph, then path x is an Hamiltonian path because matched paths are simple (no repetition of an element). The right hand side raises an exception which is trapped in function *hamilton*. The normal return of H is followed by the raising of the `Not_found` exception in *hamilton*. Note in this example the complete integration between functions and transformations: a transformation is a first-citizen value.

School of fish. In this example, we will simulate the gathering and the displacement of a school of fish. This model is inspired by a simulation of flocking birds proposed by U. WILENSKY and by the development of steering behaviors of boids (generic simulated flocking creatures) invented by C. REYNOLDS [Rey87]. The algorithm we use here is very closed to the algorithm they use.

Each fish seems to follow the same direction. Nevertheless they are not led by any special leader. In fact each fish follows the same set of rules to choose the direction of its movement. The more simplistic model consists in three rules for a fish: (1) not to collide with neighbors, (2) to join the group when it is too far and (3) finally to head for the same global direction as its neighbors. They will be detailed later, we will begin by the description of a fish.

Representation of fishes and of the school. A fish is represented by two values: its position and its direction in space. To simplify the example, we only consider a two dimensional representation, knowing that the three dimensional representation is trivial. So, we use a record (a data structure equivalent to a `C struct`) composed by three fields: two for the position (`x` and `y`), and one for the direction as an angle based on the x-axis (`theta`). We will suppose that all fishes move with the same speed that is increased if the fish is too far from the group.

```
record Position = {x, y, theta}
```

The definition of type T specifies a predicate with the same name allowing the programmer to check if its argument is of type T (here, `Position` is a predicate that holds iff its argument has at least three fields with names `x`, `y`, `theta`).

One of the ways to generate a natural neighborhood between fishes is to use a Delaunay tessellation. This topological collection is used to represent the school of fishes. It is generated from a sequence of fishes and a function that returns the fish position in space. So, we define the new type `D2` that is a Delaunay graph of dimension 2 whose elements are values of type `Position`:

```
delaunay(2) D2 =
  \e.if Position(e)
  then (e.x, e.y)
  else error("bad element type for D2 delaunay type")
  fi
```

Fish Behavior. The basic flocking model consists in three simple steering behaviors which describe how an individual fish maneuvers based upon the positions and velocities of its nearby flockmates:

1. ***Separation*** : when a fish is too close to one of its neighbors, it changes its direction.
2. ***Cohesion*** : when a fish is too far from one of its neighbors, it tries to get closer by increasing its speed.
3. ***Alignment*** : when a fish is neither too close nor too far of its neighbors, it chooses a direction which is the average of the direction of its neighbors.

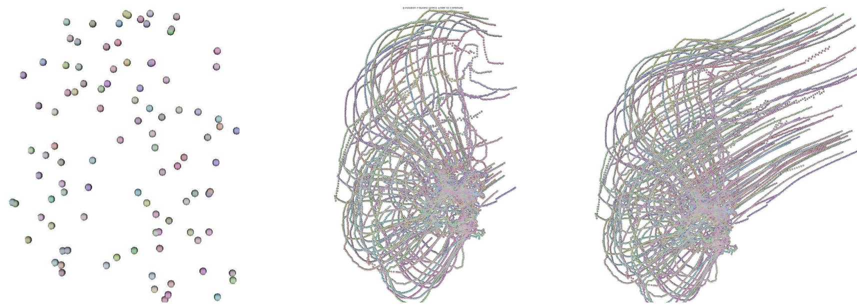

Fig. 1. Trajectory of a school of 50 fishes (the `MGS` program implementing the model is around 100 lines of code). Left plot: the initial state where each fish has a randomly chosen direction. Middle plot: the configuration after 300 iterations. Right plot: after 900 iterations of the evolution function `behavior`.

Only the rule that deals with the alignment of the fishes is presented here. The others have the same form. Thus the following transformation implements the behavior of the fishes:

```
trans behavior = {
  separation = ... => ... ;
  cohesion   = ... => ... ;
  alignment  =
    a => begin
           let phi = neighborsfold(add_theta, 0, a)
           and nb = neighborsfold(nb_neighbors, 0, a) in
           let dir = phi / nb in
             a + {x = a.x + speed*cos(dir) + random(noise),
                  y = a.y + speed*sin(dir) + random(noise),
                  theta = dir}
         end
}
```

For the rule `alignment`, first the sum of the directions of the neighbor fishes are computed to get the average direction `dir`. The function `neighborsfold` is applied on the sequence of the neighbors of fish `a`. Then, the number of neighbors is computed by the same way. The record `a` is updated: the fish moves and it now follows the new direction `dir`. Note that we introduce some noise to make the simulation more realistic. Figure 1 illustrates three steps of this process.

3 Building cellular complexes with functional operations

3.1 Beyond Graphs: Cellular Complex

The interaction structure of the two previous examples can be adequately described by a graph: two positions are connected by an edge if they are neighbors. Sequences correspond to linear graphs. Nevertheless graphs are too limited. As an example, let us consider the electrostatic laws. They depend on the geometry of the system and some values must be associated to a dimension: the distribution of electric charges corresponds to a *volumic* density while the electric flux through a surface is associated to a *surface*. Note that *balance equations* often link these values, such as the Gauss theorem for our electrostatic example. See also the work of E. Tonti [Ton74] for an elaboration.

In general, any component of any dimension and their interactions should appear in the description of the system, and we should be allowed to associate some values to them.

Arbitrary Topological Collection as Cellular Complex. The right framework to develop a topological collection type that allows the representation of arbitrary topologies is the *combinatorial algebraic topology* theory.

In this framework, the position and the neighborhood relationship of an arbitrary topological collection is a cellular complex. A cellular complex is a set of objects of various dimension called *k-cell*, where k is the dimension. To be more practical, 0-cells are vertices, 1-cells are edges, 2-cells are faces, 3-cells are volumes, etc. To build some arbitrary complex domain, the domain is divided into a cellular partition. Each cell represents a simple part of the domain and the cells are glued together: a k-cell c_1 is incident to a $(k-1)$-cell c_2 if $c_2 \subset \partial c_1$, where ∂c_1 denotes the border of c_1. This boundary relation ∂ can be used to specify the neighborhood relationships in a topological collection: *two k-cells are neighbors if they share an incident $(k-1)$-cell or if they are incident to a same $(k+1)$-cell.* A topological collection is then a function that associates some value to each cell of a cellular complex. This definition of a topological collection is consistent with the previous one.

3.2 G-Maps

There are several specializations of the notion of cellular complex. For instance abstract simplicials are special cases of cellular complexes where each cell is a simplex. A simplex of dimension n is the convex hull of $n+1$ vertices. A triangle is a simplex of dimension 2. In `MGS` one can use *generalized map* (or G-map) [Lie91] to build a cellular complex. The topological objects that can be described by G-maps are *quasi-manifolds.*

Here is a short description of G-maps. Consider the incidence graph of cells, *i.e.*, the graph whose vertices are cells and where two vertices a and b are linked with an oriented edge if a is incident to b. Let N denote the dimension of the domain, *i.e.*, the highest dimension that a cell can have. We will call a *cell-tuple*

the sequence $(c_N, c_{N-1}, \ldots, c_1, c_0)$ denoting a path in the incidence graph where c_i is a cell of dimension i. Finally two cell-tuples are i-*adjacent* if they share the same cells except for the dimension i where the cells can differ.

A G-map of dimension N is a couple whose first data is a set of *darts* and the second one is a set of $N+1$ involutions denoted by α_i $(0 \leq i \leq N)$. In fact a dart is an abstract view of a cell-tuple, and for any pair (d, d') of given darts, $d' = \alpha_i(d)$ if both cell-tuples are i-adjacent (Cf. figure 2). Finally, the last equation must be verified for G-map to be correct :

$$\forall i, j, \quad 0 \leq i < i+2 \leq j \leq N, \ \alpha_i \circ \alpha_j \text{ is an involution.}$$

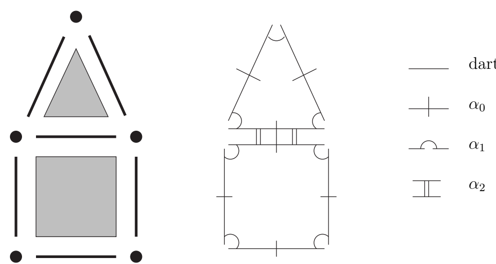

Fig. 2. Cellular complex in terms of darts and involutions.

3.3 G-maps and λ-terms

The integration of G-maps in MGS is somehow similar to the geometrical parameterization of objects via the λ-calculus suggested in [DL02]. J.-F. Dufourd and S. Luther use typed λ-calculus to describe and compose geometric objects and G-maps as using functional expressions. Using λ-calculus formalism is interesting because it allows a wide parameterization in the construction of topological objects. Indeed, they first propose a basic first-order description of G-maps, and build on this basis more complex operations for the construction and the manipulation of G-maps. Their prototype also provides an impure side that allows sharing, cloning and reusing of already defined objects.

The high expressivity of these functional expressions allows an easy specification of complex operations on G-maps by composing functions and using recursivity. Here is an example of a functional program that builds a line of n squares (viewed as 2-cells with 4 edges).

```
let rec line = function  0  ->  [edge: ○ — ●]
             |           n  ->  add_square(line(n-1))
```

Function `add_square` extends its argument by gluing a square on a distinguished edge (the empty and the gray circle denote the boundary of the distinguished edge). In the returned result, an edge is also distinguished. So, function `line` can be used recursively to build a sequence of squares by adding square from an original distinguished edge.

3.4 Using G-maps to implement cellular complexes.

To handle cellular complexes in `MGS`, we have extended the language to integrate the idea of G-maps in the same way as Dufourd *et al.* Then, we have implemented the notion of cells which is implicit in G-maps. Finally, we have adapted the pattern matching in the transformations to handle this new kind of topological collection.

The new data structure is called *QMF* referring to the term *Quasi-Manifold*. These collections are represented by a triplet (G, N, h) where G is a G-map, N is the dimension of G and h is a function that associates a value to each cell of G (in the current version of `MGS`, h is implemented as a hash-table). A QMF is a new type of topological collection parameterized by a G-map. A cell with no value will be associated to the special value `<undef>`.

`MGS` provides several operations to build G-maps: products, extrusion, contraction, fusion of two cells, etc. More generally all the operations available in the kernel of the `MOKA`[1] tool, an interactive CAD program, are also provided in `MGS`. `MGS` allows the loading and the saving of G-maps edited by `MOKA`. One may note the direct use of the G-maps manipulation library of `MOKA` in `MGS`.

The representation of topological cells from darts and involutions of the G-map is not trivial. Indeed a cell c is the set of darts corresponding to the tuples containing c. Nevertheless it is possible to enumerate the darts in this set as the orbit of a set of involutions denoted by $< \alpha_{i_1}, \alpha_{i_2}, ...\alpha_{i_k} >$. By applying any composition of these k involutions on a dart d of the G-map, we get the orbit. If d is a dart of a cell c of dimension i, the set of darts belonging to c can be computed as the orbit of $< \alpha_1, ..., \alpha_{(i-1)}, \alpha_{(i+1)}, .., \alpha_N >$ (that is, all involutions are present except the α_i). In fact involution α_i would make us change of cell during the enumeration. Thus to represent a cell, we just have to know one of its darts and its dimension. This couple can't be used directly in `MGS` to identify a cell in the G-map. Indeed this representation is not unique: there are as much representations of a cell as there are darts to compose it. It is a problem for managing the hash table of the QMF that requests unique keys. So we chose to use a special *normalized* dart representing the whole cell. This dart is the one whose memory address is the smallest in the set of darts of the cell. As a

[1] The `MOKA` home-page is available at:`http://www.sic.sp2mi.univ-poitiers.fr/moka/`

consequence we had to adapt our use of the G-map library of MOKA to deal with the normalization of a cell representation.

The last step of the G-map integration is the interfacing between the top-level system MGS written in OCAML, and the G-map library written in C++. We chose to use an abstract type [Ler04, pp234] to represent the darts and the G-maps in the OCAML side. These abstract types are blocks of the OCAML heap and these blocks are customized by *ad-hoc* functions of comparison, finalization and hashing. Thus one can use most of the basic functions on these values, and efficiently deal with the issues due to the garbage collector of OCAML.

4 Examples

In this section we present two examples of MGS programs using QMF and showing how transformations allow us to animate this new type of collection.

4.1 Modeling of the DLA process

We model in MGS the *diffusion limited aggregation* process, or DLA [WS81]. The principle is simple: a set of particles diffuse randomly on a given spatial domain. Initially one particle, the seed, is fixed. When a mobile particle collides a fixed one, they stick together and stay fixed. For the sake of simplicity, we suppose that they stick together forever and that there is no aggregate formation between two mobile particles.

This process leads to a simple cellular automata with an asynchronous update function or a lattice gas automata with a slightly more elaborate rule set [TM87]. It could be easily done in MGS [Gia03] using a QMF collection to model arbitrary spatial domains.

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

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

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

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

Initial state of the system. The most elegant way to associate values to the cells of a G-map is to define, when it is possible, a transformation that builds the initial state. For this transformation, we must consider that there is no value associated to the cells yet. So, we have to create the first fixed seed and then to randomly distribute mobile particles anywhere else on the domain:

```
trans init[flag = true] = {
    <undef> / flag              =>  (flag := false ; ‘fixed) ;
    <undef> / (random(4)==0)    =>  ‘mobile
}
```

The first rule is applied only one time because as the first occurrence is found the guard `flag` becomes false. The second rule is applied with a probability of 0.25.

Application on different topologies. Figure 3 shows applications of the DLA transformation on different kinds of objects built with G-maps. As a matter of fact, the change of topological collection doesn't affect the transformation and we still apply the same `dla` transformation. In these examples, the topological collections have dimension 2 and the values are associated only to 2-cells. The 2-cells are neighbors if they share a common edge on their boundary. At the top of the figure, only the 1-cells are figured, and at the bottom, only the 2-cells that hold a value, after that a fixed point is reached, are represented. Note that some cells have 4 neighbors whereas others have 3 (for instance at the poles of the sphere).

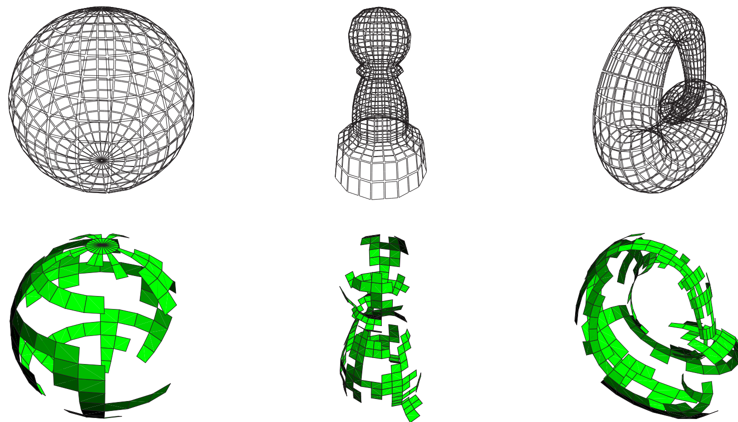

Fig. 3. DLA on complex objects. At the top, different objects are presented: from left to right, a sphere with 18 parallels and 24 meridians, a chess pawn and a Klein's bottle. The bottom line presents the final state of a DLA process on these objects.

4.2 Maximum flow problem

A graph is a cellular complex of dimension 1 where vertices are 0-cells and edges are 1-cells. We will take advantage of this property to handle graphs through the use of the QMF collection in `MGS`. This section aims at illustrating the high expressivity of transformations within the framework of a complex algorithm.

The algorithm of maximum flow problem requires to valuate the 0-cells and the 1-cells. It can be written is `MGS` as two transformations acting on cells of different dimensions.

Ford-Fulkerson algorithm. We want to represent communication network between different sites. The link between two sites are channels whose flow is limited by a maximal capacity. Let *source* and *target* be two sites of the network such as *source* is a source (no flow goes inside it) and *target* is a target (no flow goes outside it). The problem is the following: what is the maximal flow that *source* can release in the network and *target* can accept, without saturating the channels of the network?

The network is naturally represented by a graph whose vertices are the sites and whose oriented edges are channels between the sites. The Ford-Fulkerson algorithm [FF56] returns one of the solutions from a feasible initial state of the network. A flow is feasible if:

- for any site i (except *source* and *target*), the incoming flow of i is equal to the out-coming flow of i,
- for any channel c between two sites, the flow is positive and lower than the capacity of c.

A trivial feasible flow is the null flow: no data is circulating in the network. The algorithm is made of two steps:

1. Marking: we assume that we slightly increase the out-coming flow of *source*. It will be denoted by the `‘plus` mark associated to vertex *source*. Then we will try to spread this increase as following: let c be an edge linking sites i to j. If i is marked, j is unmarked and c isn't saturated: we can mark j by `‘plus`. On the opposite, if j is marked, i is not marked and j the flow of c isn't null: we mark i by `‘minus`. We iterate this marking until a fixpoint is reached.
2. Increasing the flow: if *target* is marked, we can find a path between *source* and *target* whose edges flow can be changed, creating an increase of the total flow of the network. This path is a *flow augmenting path.* We first compute this increase, we update the network and the process is iterated. If *target* isn't marked, the flow is maximal.

Note that the first step changes the vertex values, whereas the second one only acts on the edges.

Types definition. We will type values associated with cells. A site has a name and it can either be marked or unmarked. We will use a record `MGS` to represent them:

```
record site     = {name : string}
record marked   = site + {mark}
record unmarked = site + {~mark}
```

A record `marked` contains all the fields of a record of type `site` plus a field `mark`. A record `unmarked` contains all the fields of a `site` record and should not contain any `mark` field. The field `mark` will take its values in the set of symbols {`‘plus`, `‘minus`}.

The values of edges will also be records with two fields (for the flow and the capacity):

```
record channel = {flow : int, capacity : int}
```

Marking step. It is programmed by a first transformation which initializes the marking, and a second one which applies the previously described rules:

```
trans init = {
    x / source(x) => x + {mark = ‘plus} ;
    x             => x + {~mark}
}

trans marking = {
    i:marked, j:unmarked / (edge(i,j).flow < edge(i,j).capacity)
    => i, j+{mark = ‘plus} ;
    i:unmarked, j:marked / (edge(i,j).flow > 0)
    => i+{mark = ‘moins}, j
}
```

Function `source` is a predicate checking that its argument is *source* (we will also use the predicate `target` for *target*). Function `edge` takes two vertices as parameters and returns the value of the edge which links them. One also has to add an orientation test of the edge. We haven't introduced it to simplify the code. The filtered elements have type `site`. So the displacements are made by following cells of dimension 1. The addition between two records `r + r’` computes the asymmetric fusion: the result is a record that contains all the fields of `r` and `r’` with a priority for `r’` when a collision occurs.

Increasing the flow. The increasing string can be easily found by an iteration:

```
trans increasing_string = {
    (c:channel)* as C / source(vertex_i(hd(C)))
                      && cible(vertex_j(last(C)))
      => increase(C)
}
```

The functions `vertex_i` and `vertex_j` return the vertices of an edge. Function `increase` takes the edges path as an argument and increases the flow as much as possible on these edges. So the total flow is updated. If any path can be filtered by the rule, the flow is maximal. As a matter of fact applying the algorithm on a graph g is done by:

```
fun ff(x) = increasing_string(marking[‘iter=‘fixpoint](init(x)))
ff[‘iter=‘fixpoint](g)
```

5 Conclusion and Perspectives

This paper only focuses on a part of the features available in `MGS` that can be used to develop computer models of complex discrete dynamical systems that are concise and mathematically well-founded. The approach has been successfully applied to several biological processes (the growth of a tumor, colonies of ants foraging for food, the heterocysts differentiation during *Anabaena* growth, etc.) as well as more algorithmic problems (prime number generation, various sorting algorithms, Hamiltonian path, etc.). These examples, as well as the sources of the current `MGS` implementation, are available at `http://mgs.lami.univ-evry.fr`.

The modeling of $(\mathrm{DS})^2$ through their interaction structure is part of a long term research effort [Gia03]. The topological approach presented here provides a unified description of several computational models. Obviously, Lindenmayer systems [Lin68] correspond to transformations on sequences, the CHAM, Gamma and basic Paun systems [BB90,BFM01,Pau01] can be emulated using transformations on bags, and usual cellular automata [TM87] can be implemented on regular topologies with the GBF collection.

From a formal point of view, the concept of topological collection is connected with the concept of *topological chain* and the concept of transformation corresponds to some *cochains*, notions developed in homological algebra [GM02]. The integration of G-maps in `MGS` is very similar to the geometrical parameterization of objects via the λ-calculus suggested in [DL02]. However, a topological collection decorates each cell by a value handled through the concept of transformation. From the point of view of functional languages, a transformation extends the concept of rewriting of terms or function defined by case (filtering) in non-algebraic data structures.

The perspectives opened by this work are numerous. From the applications point of view, we are targeted to the simulation of developmental processes in biology [GM03,GMM04]. At the language level, the study of the topological collections concepts must continue with a finer study of transformation kinds. Several kinds of restrictions can be put on the transformations, leading to various kinds of pattern languages and rules. The complexity of matching such patterns has to be investigated. The efficient compilation of an `MGS` program is a long-term research.

Acknowledgments. The authors would like to thank the MOKA team at the Univ. of Poitier, J.-L. Giavitto and J. Cohen at LaMI, F. Jacquemard at INRIA/LSV-Cachan

and the members of the "Simulation and epigenesis" group at Genopole for technical support, stimulating discussions and biological motivations. This research is supported in part by the CNRS, GDR ALP, IMPG, University of Évry and Genopole/Évry.

References

[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.

[BFM01] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. Gamma and the chemical reaction model: Fifteen years after. *Lecture Notes in Computer Science*, 2235:17–44, 2001.

[DL02] Jean-Franois Dufourd and Sven Luther. Parametrizing geometric objects using λ-calculus. In *Proceedings of the 18th spring conference on Computer graphics*, pages 185–194. ACM Press, 2002.

[FF56] Lester R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[Gia03] J.-L. Giavitto. Invited talk: Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *Rewriting Technics and Applications (RTA'03)*, volume LNCS 2706 of *LNCS*, pages 208 – 233, Valencia, June 2003. Springer.

[GM02] Jean-Louis Giavitto and Olivier Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.

[GM03] J.-L. Giavitto and O. Michel. Modeling the topological organization of cellular processes. *BioSystems*, 70(2):149–163, 2003.

[GMM04] J.-L. Giavitto, G. Malcolm, and O. Michel. Rewriting systems and the modelling of biological systems. *Comparative and Functional Genomics*, 5:95–99, February 2004.

[Ler04] X. Leroy. The Objective Caml system, revision 3.07, 2004. Software and documentation available on the web at http://pauillac.inria.fr/ocaml/.

[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.

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

[Rey87] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model, July 1987.

[TM87] T. Toffoli and N. Margolus. *Cellular automata machines: a new environment for modeling*. MIT Press, Cambridge, 1987.

[Ton74] Enzo Tonti. The algebraic-topological structure of physical theories. In P. G. Glockner and M. C. sing, editors, *Symmetry, similarity and group theoretic methods in mechanics*, pages 441–467, Calgary, Canada, August 1974.

[WS81] T. A. Witten and L. M. Sander. Diffusion-limited aggregation, a kinetic critical phenomenon. *Phys. Rev. Lett.*, 47:1400–1403, 1981.