LaMI

Laboratoire de Méthodes Informatiques

MGS: a Programming Language for the Transformations of Topological Collections

Jean-Louis Giavitto & Olivier Michel

email(s) : giavitto *ou* michel @lami.univ-evry.fr

Rapport de Recherche n^o 61-2001

Mai 2001

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

MGS: a Programming Language for the Transformations of Topological Collections

Jean-Louis Giavitto & Olivier Michel

LaMI u.m.r. 8042 du CNRS
Université d'Evry Val d'Essone
91025 Evry Cedex, France.
[giavitto,michel]@lami.univ-evry.fr

LaMI technical report N° 61-2001, May 2001

Abstract

We present the first results in the development of a new declarative programming language called MGS. This language is devoted to the simulation of biological processes, especially those whose state space must be computed jointly with the running state of the system (for instance, in morphogenesis).

MGS proposes 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). They share the property of specifying local transformations in space and time.

The basic computation step in MGS replaces in a collection A of elements, some subcollection B, by another collection C. The collection C is function solely of B and its adjacent elements in A. The pasting of C into $A-B$ depends on the shape of the involved collections. This step is called a *transformation*.

The organisation of the elements into the collections is viewed from a topological viewpoint. A formal framework to specify this notion of *topological collections* is proposed. By changing the topological structure of the collection, the underlying computational model is changed.

The specification of the collection to be substituted can be done in many ways. We propose here a pattern language based on the neighborhood relationship induced by the topology of the collection. Several features to control the transformation applications are then presented.

Keywords

Topological collection, transformation, declarative programming language, simulation of biological processes, dynamical systems, dynamical structure, Gamma, CHAM, P system, L system, cellular automata, rewriting, rule based programming, combinatorial algebraic topology, chain complex, chain group.

The authors of this research report can be contacted at:

La.M.I., CNRS UMR 8042
Université d'Évry Val d'Essonne
Tour Évry 2 / 4eme etage
523 Place des terrasses de l'agora
91000 Évry Cedex France

Tel: +33 (0)1 60 87 39 04
Fax: +33 (0)1 60 87 37 89

The MGS interpreters are freely available, by sending a demand to giavitto *or* michel @lami.univ-evry.fr . The MGS home page is located at url http://www.lami.univ-evry.fr/~mgs .

Versions of this report:

- Revision september 2001: Added example 4.10 and appendix B, some additional references and acknowledgements, corrections of typos.
- Initial Version: may 2001.

Table of Contents

List of Figures

List of Definitions

1 Motivations

We want to developp a framework dedicated to the simulation of dynamical systems with a dynamical structure. The application area we have in mind is the simulation of some biological processes, especially those whose state space must be computed jointly with the running state of the system. This technical report is organized as follow:

Section 1 gives our motivations. After a very brief presentation of the notions related to the dynamical systems, we introduce the notion of dynamical structure through some examples.

Section 2 sketches an unified framework to describe dynamical system with a dynamic structure. The notions of collection, subcollection and transformation are described.

Section 3 contains a brief description of the MGS programming language. This language implements a subset of the previous ideas.

Section 4 illustrates the MGS language through paradigmatic examples.

Section 5 presents the first development of a formal description of the MGS constructions, using mathematical notion developped in the field of algebraic topology. Our main goal in this section is to introduce some of the topological notions upon which a theory of transformations can be build.

Section 6 makes a comparison with other approaches: the Gamma language and the CHAM, Lindenmayer systems, Paun systems and cellular automata. A comparaison with the approach of multi-agent systems, often advocated in the simulation of complex dynamical systems, is also developped.

Section 7 concludes this report by giving some directions opened by this work.

1.1 Dynamical Systems and their State Structures.

A *dynamical system* (or DS in short) corresponds to a phenomenon that evolves in time. The phenomenon is located on a *system* characterized by " observables ". The observables are called the *variables* of the system, and are linked by some relations. The value of the variables evolves with the time. The set of the values of the variables that describes the system constitutes its *state*. The state of a system is its observation at a given instant. The state has often a spatial extent (the speed of a fluid in every point of a pipe for example). The temporal sequence of state changes is called the *trajectory* of the system.

Intuitively, a DS 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 telling us where the point should go next from its current location (the *evolution function*). These notions are illustrated in Fig.1.

We are interested in the simulation of such systems. This requires the specification of the system state and the evolution function. This specification can be very difficult to achieve

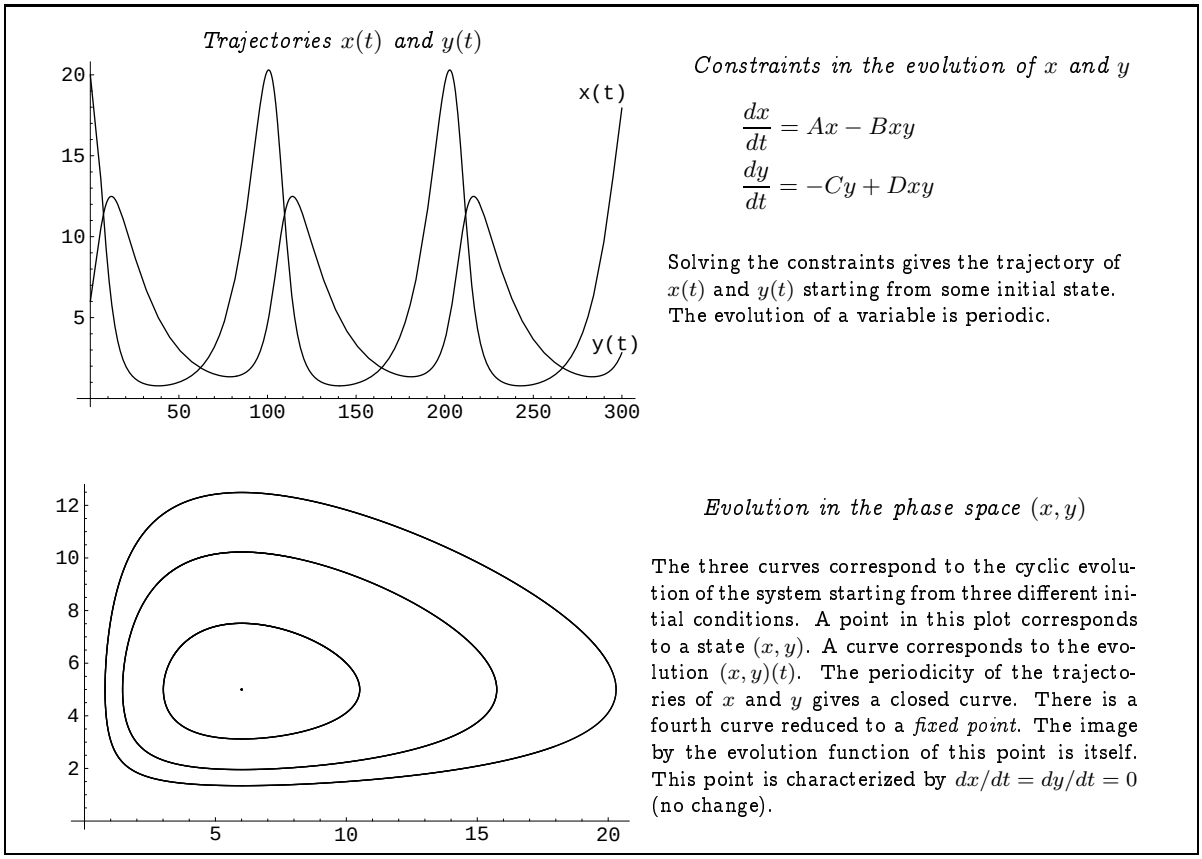


Figure 1: Evolution of a predator-prey system (a DS with a static structure).

The system is characterized by two variables: x corresponds to the number of predators and y to the number of preys in some ecological system. The number of preys changes because of the growth of the population and because the preys are eaten by the predators. The number of births is proportional to the number of preys and the decrease is proportional to the number of prey-predator encounters, which is itself proportional to the product xy. The number of predators decreases because the competition between predators and the increase is proportional to the chance of prey-predator encounters. The resulting differential equations specify the evolution function. They can be integrated to plot the trajectory of x and y (top picture) and the state evolution (bottom picture). The structure of the system is static in the sense that the state of the system is always described as an element of $\mathbb{R}^2$.

because of the complexity of the description of the phase space and of the evolution function. However, more we know about the phase space, more we know about the DS behavior. For example, if the phase space is finite, every trajectory is finally cyclic.

Very often the phase space has some structure and this structure can be used to simplify the description of the state and its evolution and to gain some knowledge about the system. For example, one may specify the evolution function h_i for each observable o_i and recover the global evolution function h as a product of the " local " h_i.

Standard DS exhibit a static structure, that is, *the exact phase space* of the DS can be known statically before the simulation. For instance, in the example of a fluid flowing through a pipe, since the geometry of the pipe is not subject to change, the structure of the state is not a function of time (and the phase space corresponds to the vector fields on the static volume of the pipe).

1.2 DS with a Dynamical Structure

The *a priori* determination of the phase space cannot always be done. This situation is usual in biology [Fon92, FB94, FB96] for instance in the modeling of plant growing, in developmental biology, integrative cell models, protein transport and compartment simulation, etc. This account for the fact that the structure of the phase space must be computed jointly with the running state of the system. In this case, we say that the DS has a *dynamical structure*. Often, the description of DS with a dynamical structure is especially hard.

> *In this kind of situation, the dynamic of the 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.*

A simple example is given in figure 2. This example is simple beacuse the structure of the state at time$t$ does not depend of the previous trajectory (the states at time $t' < t$). However, the usual cases are much more intricated and difficult to specify.

This is best shown on some examples. The following examples play a central role in the motivations of the framework presented in section 2 and were very influential on the current work. They exhibit some key properties we want to emphasize and precise the kind of entities, organizations and transformations we have in mind. They outline some important features needed for a language devoted to the simulation of DS with a dynamical structure.

The reader not interested by our motivations, may omit the rest of this section and go directly to section 2 page 11.

1.3 Chemical Reactions

Suppose that we have a system consisting in two molecules of type a and one molecule of type b floating in a test tube at time t. The state of the system can be represented by the multiset

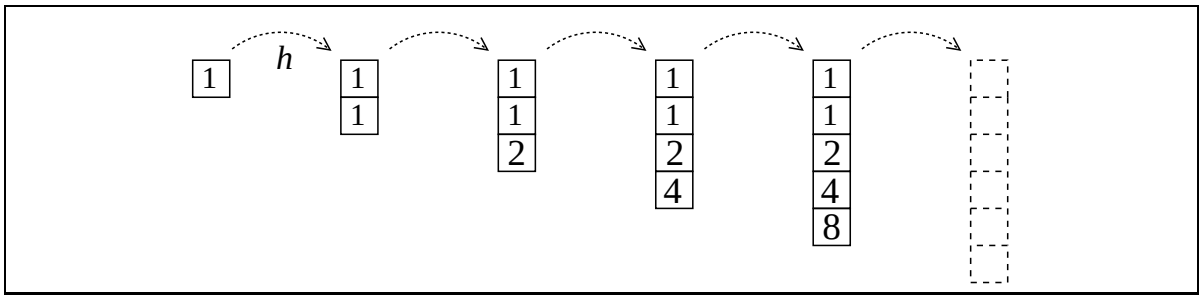


Figure 2: DS with a simple dynamical structure. The state of the system is a vector of integers. If at time t the state s_t is an element of $\mathbb{Z}^q$, then the state s_{t+1} at time $(t+1)$ is an element of $\mathbb{Z}^{q+1}$ computed as follows. The ith element of s_{t+1} is equal to the ith element of s_t and the $(q+1)$th element of s_{t+1} is equal to the sum of all previous elements. It is easy to see that the set S_t of the possible states at time t is $\mathbb{Z}^t$ if $S_1 = \mathbb{Z}$. Then, the set S of all possible states (at any time) is $S = \mathbb{Z}^* = \mathbb{Z}^1 \cup \mathbb{Z}^2 \cup ... \cup \mathbb{Z}^n \cup ....$ This DS has a dynamical structure because $S_t \neq S$. It is always possible to consider S instead of the sets S_t but a lot of informations on the system structure is lost. In this example, the evolution function h is simple to specify and the function H that gives the set S_{t+1} from $s_t \in S_t$ is very simple: $H(s_t)$ does not depend on the value s_t but only on S_t. Then it is easy to infer the set S as $\bigcup H^t(\mathbb{Z})$. Sections 1.3 to 1.7 give some examples where the function H really depends of the precise value of the state.

$\{\!|\, a, a, b \,|\!\}$. We suppose that one a and one b can react together to give two c (reaction α) and that two c react together to give one d (reaction β) only in the presence of a (a is a catalyst).

Starting from $\{\!|\, a, a, b \,|\!\}$ a reaction α may occur. So, at time $t+1$ the state of the system is represented by $\{\!|\, a, c, c \,|\!\}$. At time $t+1$ there is no more subpart $\{\!|\, a, b \,|\!\}$ in the system. Then the evolution rule α cannot be applied anymore. However, the reaction β may be applied to give a new state $\{\!|\, a, d \,|\!\}$.

At this point no more reaction can occur and we can say that the system has reached a fixpoint (or an equilibrium). We can resume the trajectory of the system by:

$$\{\!|\, a, a, b \,|\!\}_t \xrightarrow{\alpha} \{\!|\, a, c, c \,|\!\}_{t+1} \xrightarrow{\beta} \{\!|\, a, d \,|\!\}_{t+2}$$

We have decided to model the state of the system crudely by the multiset of molecules present in the test tube. Then, the point is that the evolution of the system cannot be described by evolution rules linked solely to one basic system element a, b, c or d. In this case, it is natural to link the evolution rule, a chemical reaction $a + b \rightarrow c + c$, to a *subpart* $\{\!|\, a, b \,|\!\}$ of the whole system. Note that reaction β can be modeled as consuming one a and producing one a: $c + c + a \rightarrow d + a$ and then this rule can be linked to the subpart$\{\!|\, c, c, a \,|\!\}$. However, a is left unchanged and it is perhaps more natural to say that the rule $c + c \rightarrow d$ is linked with subpart $\{\!|\, c, c \,|\!\}$ but holds only if there is some a.

With this particular choice of representation, the subparts of the system are changing with the application of a rule. At time $t+1$ there is no more subpart $\{\!|\, a, b \,|\!\}$ in the system and the evolution rule α cannot be applied anymore. At time $t+2$ there is no more subpart $\{\!|\, a, b \,|\!\}$ nor subpart $\{\!|\, a, c, c \,|\!\}$ and no rule at all can apply. Although the phase space can be characterized uniformly as a multiset, the number of elements of this multiset is changing, as well as the evolution rules that can be used to make the system evolves. This is why we say

that this DS exhibits a dynamic structure.

Obviously another choice of state representation avoids this burden. For instance, the system can be modeled as four numbers quantifying the number of molecules a, b, c and d present in the test tube. With this representation, the phase space is uniformly $\mathbb{N}^4$. However, we insist here on this particular model where each individual molecule is explicitly represented. The reader accustomed to the usual chemical models can be disturbed by this point of view, but one may imagine a situation where a molecule cannot be abstracted by an integer and requires its explicit appearance in the model. More generally, it does not always exist a representation of the system state that avoids the change of the phase space, or this representation is not always desirable.

The points we want to emphase, are:

- The evolution of the system is specified as a set of evolution rules.
- An evolution rule gives the evolution of a subpart of the system.
- We insist that the subparts subject of these rules are in general not reduced to only one element.

1.4 Game of Life

The *game of life* is a particular type of cellular automata (cf section 6.4 for a more general description of cellular automata). It can be described in the following way. Each element of an array represents a cell in two possible states: dead or alive. A dead cell surrounded by 3 alive neighbors becomes alive. An alive cell surrounded by less than three alive neighbors becomes dead from isolation. An alive cell surrounded by more than three living neighbors dies by starvation.

In this example, one sees that the global state of the system is described by the state of all the cells. The evolution function for one cell depends on both the current state of the cell and the state of the neighbors. However, in the contrary of the preceding example, the subpart of the system which evolves is the cell x alone, and not the cell x together with the other arguments of the evolution function (the neighbors). Indeed, the neighbors evolve for themselves, even if their current state interact in the evolution of other cells. Put in other words, the states of the neighbors are not consumed by the evolution of cell x.

This is perhaps better explained by contrast with an alternative of cellular automata: *lattice gas automata*. In this formalism, a set of molecules moves in a grid. So, a cell in a grid has a state indicating if the cell is empty or if it contains one (or several) molecule moving in a given direction. Molecules interact when they meet in a cell. Figure 3 gives some examples of rules. In opposition with the game of life approach, a rule specifies the evolution of simultaneously several cells. However, the evolution function is *local* in the sense that the interacting cells are connected.

The two examples stress the space organization of the elements of the system. Being present at the same time (in a chemical solution as in the preceding example) is not a sufficient condition to interact. They must moreover be neighbors. However, this concept of

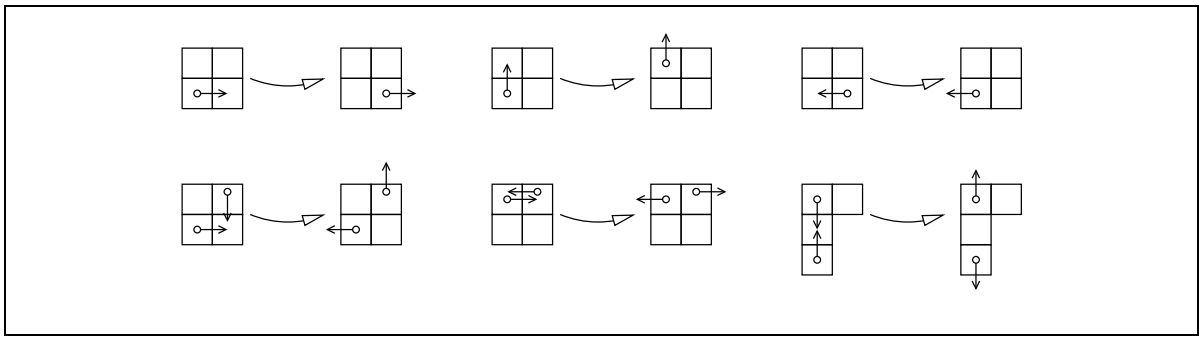


Figure 3: Some rules for a lattice gas automata.

neighborhood is fixed here once and for all, because the evolution of a cell gives again a cell in the same place (or in lattice gas automata, the evolution of a subpart gives a subpart with exactly the same shape).

The points which we want to underline are the followings:

- The elements of the system have a strong spatial organization, which defines a concept of neighborhood.
- Two elements can only interact if they share a neighbor relationship.
- The interaction specifying the evolution of an element does not necessarily describe the evolution of the participating neighbor elements.

1.5 Embryogenesis

The preceding example shows a strong spatial organization of the entities which compose the dynamical system (at least compared with the chemical solution). This spatial organization is however static. Here an example, drawn from biology, which needs intrinsically a dynamic spatial organization.

During the development of an embryo, several domains of cells change their shapes. For instance, the neural tube is formed dorsally in the embryonic development of Vertebrates by the joining of the 2 upturned neural folds formed by the edges of the ectodermal neural plate, giving rise to the brain and spinal nerve cord; see figure 4.

In general, the morphogenesis of biological systems is a consequence of the local evolution of cells like growing and proliferation, mobility, differentiation and apoptosis (programmated death of cells). There is no centralized control, only the diffusion of chemical signals from a cell to its neighbors and the own internal evolution of the cell.

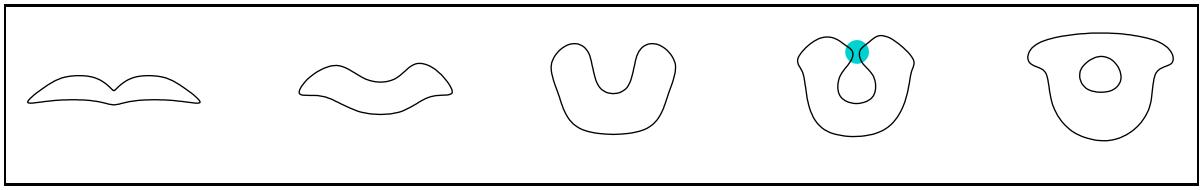


Figure 4: During the neurula stage, the neural plate is folded to shape a tube.

The state of the entire system cannot be reduced to the set of the state of the cells because such representation misses the information related to the neighborhood of each cell. And the neighborhood of a cell changes in time. For example, some cells on the boundary of the neural plate are glued together and become internal cells at the end of the neurula stage.

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. This example if further developped in section 6.6.

The changes in the boundary of a cell are due to cell mobility, apoptosis and proliferation. These causes are " internal " to the DS. The changes in the neigborhood of a cell can also be caused by a change in the environment (the conditions outside the system), e.g. a change in the geometry of the embedding space, the encounter with an obstacle or an obstruction, etc. See for example the change of growth in a plant encountering an obstacle, in figure 5.

In the neurula stage and in the plant growth, the interactions are still done between neighboors elements. The cells at the boundary of the neural plate become neighbors when the plate is folded, and the growth unit of a plant becomes a neighbor of a wall or another growth unit. The evolution is then still specified through *local* rule, even if the structure of the DS is changing. The global change in the DS structure, is the " sum " of the local changes.

The points we want to emphasize are:

- The structure of the DS, that is, the organization of its elements, may depend of external or internal factors.
- However, the evolution of the system is always specified through local rules. It is the result of all the local changes that gives the global change of the system.

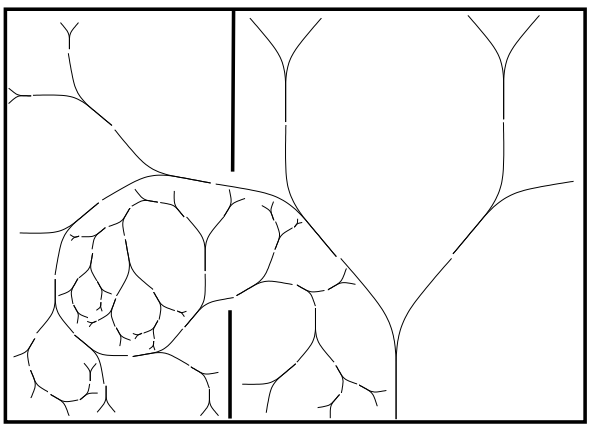


Figure 5: Plant growth in presence of obstruction. This figure represents the growth of a plant, as it can be modeled by a L system (cf. section 6.3), using the replication of a growth unit with a variation in size and orientation. However, " external " factors disturb the replication process. For example, walls put constraints or stop the replication. And even the plant itself makes obstructions to its own development.

1.6 Protein transport and Golgi Formation

In the previous examples, the system is decomposed in atomic entities (molecules, growth units) or homogeneous entities (cells with a more or less complex state). However, the decomposition of the system is not always restricted to one level and can be further refined.

Here we sketch an example which is of great importance for the simulation of the cell functioning: the processes by which proteins are physically transported through membranous systems to the plasma membrane or other organelles, or from the cell surface to organelles within the cell. A proposed mechanism consists of small membranous vesicles. A soluble protein is carried within the lumen of a vesicle, and an integral membrane protein is carried within its membrane. The figure 6 illustrates the nature of the budding and fusion events by which the vehicles move between adjacent compartments.

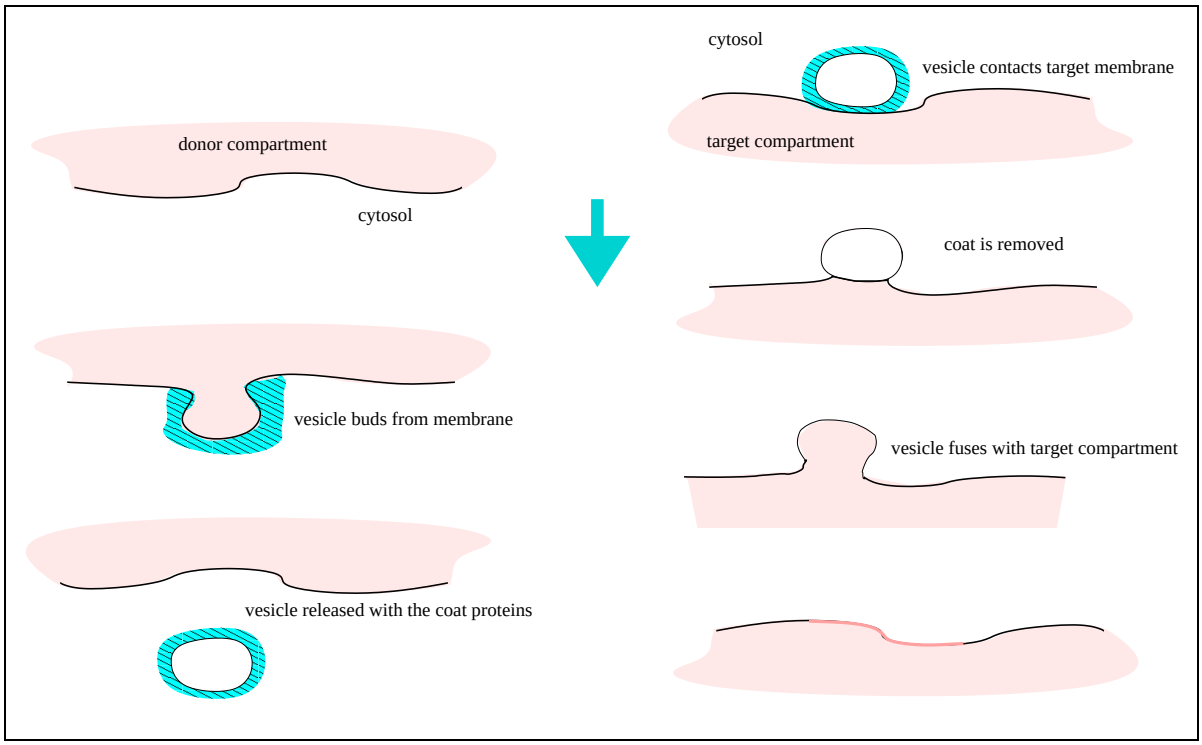


Figure 6: Protein trafficking. Membranes vesicles bud from a donor compartment and are surrounded by coat proteins. The coated vesicle binds to a target compartment, is uncoated, and fuses with the target membrane, releasing its content. (From [Lew97, ch. 34: Protein Trafficking].)

To simulate these mechanisms, we must represent the compartment volumes, the membrane surfaces and the proteins (which may be assimilated to points). In addition to the geometry of these entities, one must represent the density of the proteins on the surfaces coat (it is hypothesized that the curvature of the membrane depends of the presence of some proteins in the coat), or of vesicle in a volume (e.g. to compute the chance of a vesicle to encounter the target membrane and the release level of proteins). In addition, there are some reactions between proteins and mechanical effects on the membrane and vesicle. Furthermore, membrane may be polarized.

This example exhibits a very complex spatial organization. The system can be decomposed in compartments and vesicles. Vesicles are dynamically created (it is also hypothesized that some compartment, like the Golgi, may be created as the dynamic equilibrium of aggregations/separations of the vesicles released by the endoplasmic reticulum). A vesicle or a compartment can be further refined into a membrane, a lumen (the volume inside the compartment) and eventually a coat. A compartment may include other compartments, etc. The topology implied by the representation of these entities is tridimensional (compartments), bidimensional (membranes) and zero dimensional (molecules). Some models also use tensegrity structures to explain the mechanics of the membranes. Then, one-dimensional structures must be introduced to represent the cytoskeletal filaments that allow the cell to resist the distortion of shape when a mechanical stress is applied to it. Obviously, the interaction that must be described depends of the dimension of the entity: for instance, a flow of molecules can be conceived only through a membrane boundary between two compartments, not between a filament and another molecule; conservation laws depend on the topological nature of the entities, etc.

The moral of the story is that the decomposition of a system may exhibit a dynamical complex organization, requiring the description of arbitrary changing topology. The complete description of the system must include several heterogeneous interacting viewpoints (geometry of the system, chemical activity, mechanical state, electrical activity, etc.) at several levels.

1.7 Cell Division

Suppose that we have a system consisting in one cell c floating in a test tube at time t. The cell c is described by a state s. This cell is in a state such that it divides. At time $t+1$ we have 2 cells c_1 and c_2 with states s_1 and s_2 in the test tube. The point is that it would be very irrelevant to say that c_1 (resp. c_2) is the cell c in the new state s_1 (resp. s_2). As a matter of fact[1], the division process is supposed to be symmetric and it will be arbitrary to identify one of the cell at time $t+1$ as " the previous cell" and the others as the " new, children cell". In other word, when an amoeba divides, it is irrelevant to ask which one of the two corresponds to the initial amoeba.

The point we want to focus is a little subtle: confronted to a DS with a static structure, it is easy to decompose the system in fixed parts, to attach some behavior to these parts and to conceive that these parts have a well defined identity in time. This is no longer true when the parts of the system are changing, because it will be very arbitrary to identify one part in the course of the time. This does not mean that only a global description is possible. It means that the right unit of description are the interacting parts and that the corresponding decomposition changes in time. We come back to the problem of identifying a definite parts among the time in section 6.6 when we compare the MGS approach with the multi-agent paradigm).

[1] We suppose that there is no special information in the state to uniquely identify the cell. The structure of the cell state consists solely in the information needed to describe the functioning of the cell and cells are indistinguishable by their functioning.

1.8 Summary of the Examples

The previous examples were very influential for our motivations. To summarize, we are in quest of a programming language dedicated to the simulation of dynamical systems with a dynamical structure. Such a language must take into account the following features:

1. The evolution of the system is specified as a set of evolution rules.
2. An evolution rule gives the evolution of a subpart of the system.
3. We insist that the subparts subject of these rules are in general not reduced to only one element.
4. The elements of the system have a strong spatial organization, which defines a concept of neighborhood.
5. Two elements interact only if they are neighbors (rules are local).
6. The interaction specifying the evolution of an element does not describe necessarily the evolution of the participating neighbor elements.
7. The structure of the DS, that is, the organization of its elements, may depend of external factor, or internal ones.
8. However, the evolution of the system is always specified through local rules. It is the sum of all local changes that gives the global change of the system.
9. The decomposition of a system may exhibit a dynamical complex organization, requiring the description of arbitrary changing topology.
10. The complete description of the system must include several heterogeneous interacting viewpoints (geometry of the system, chemical activity, mechanical state, electrical activity, etc.) at several levels.
11. The parts of the system does not have a remanent identity.

2 MGS Basic Ideas

2.1 The Concept of Transformation in a Collection

Returning to the idea expressed in section 1.2 page 3, our goal is to provide a general support for the notions of " organized set " and " local competing transformations " that arise in the previous examples. From this point of view, the previous examples share the following common characteristics.

Discrete space and time. The structure of the DS state consists of a discrete *collection* of elements. We call *collection* a set of elements with some " organization " (to be clarified later). This discrete collection evolves in a sequence of discrete time steps[2].

If the state of a system consists in a collection, the elements in this collection do not have a remanent identity in time because the organization of the collection may change with the structure of the system. In other words, the collection reduces to a collection of *values*.

It is tempting, and we will do so, to separate the " shape " (i.e. the organization) of the collection and its content (the values).

Complex Organization. Several kinds of organizations (of values) are used in programming languages and give raise to several data structures: sets, multisets (or bags), sequences, arrays, trees, terms, etc. However, the description of a DS often exhibits more complex organizations. For instance, the organization of a DS is often based on the 3-dimensional topological structure of the physical entities in the system.

Temporally local transformation. The computation of a new value in the new state depends only on values computed for a fixed number of preceding steps, and usually just one step.

Spatially local transformation. The computation of a new collection is done by a structural combination of the results of more elementary computations involving only a small and static subset of the initial collection.

" *Small and static subset* " makes explicit that only a fixed subset of the initial elements are used to compute a new element value.

" *Structural combination* ", means that the elementary results are combined into a new collection, irrespectively of their precise value. The global organization of the new collection results of the combination of these local changes.

[2]Independantly of the discrete or continuous nature of the entities that have to be represented in the modeled systems, they must be discretized finally for their computer representation. We have decided to delegate this problem upstream of the programming. It implies that there is no special feature embedded into our framework dedicated to the explicit support of continuous entities (like ODE or PDE solvers, continuous time representation, etc.). For instance, the system of two differential equations of figure 1 must first being discretised (e.g., as a finite difference scheme $x_{t+1} = h_x(x_t, y_t)$ and $y_{t+1} = h_y(x_t, y_t)$) and these discrete equations are the base of the corresponding simulation program.

Considering these characteristics, we propose to idealize the description of a DS evolution by the following abstract computational mechanism:

1. a subcollection A is selected in a collection C;
2. a new subcollection B is computed from the collection A;
3. the collection B is substituted for A in C.

This processus is pictured in Fig. 7. We call these three basic steps a *basic **transformation*** of the collection. A transformation, without the " basic " qualifier, consists in several non interacting basic transformations applied in parallel to a collection. A transformation corresponds to one evolution step of the DS. Then, the iteration of transformations builds the entire DS trajectory, cf. Fig. 8.

In addition to the specification of the underlying organization, the definition of a basic transformation requires the specification of the subcollection A and the replacement B. This specification defines a ***rule*** and must adapt several constraints and variations. We propose to base the specification of the organization and of these constraints on *topological concepts*.

For example, in the game of life, the value of a cell c at time $t+1$ depends on the value of the neighboring cells at time t. If we identify the value of the cell c at time t with the

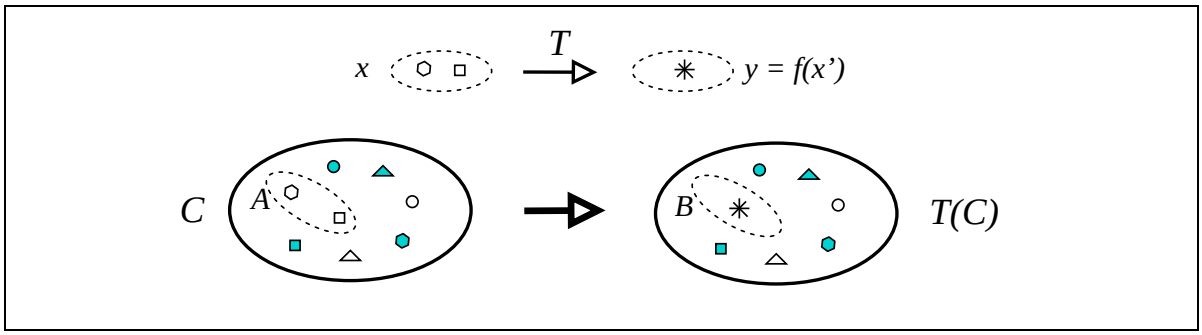


Figure 7: A basic transformation of a collection. Collection C is of some kind (set, sequence, array, cyclic grid, tree, term, etc). A rule T specifies that a subcollection A of C has to be substituted by a collection B computed from A. The right hand side of the rule is computed from the subcollection matched by the left hand side x and its possible neighbors x' in the collection C.

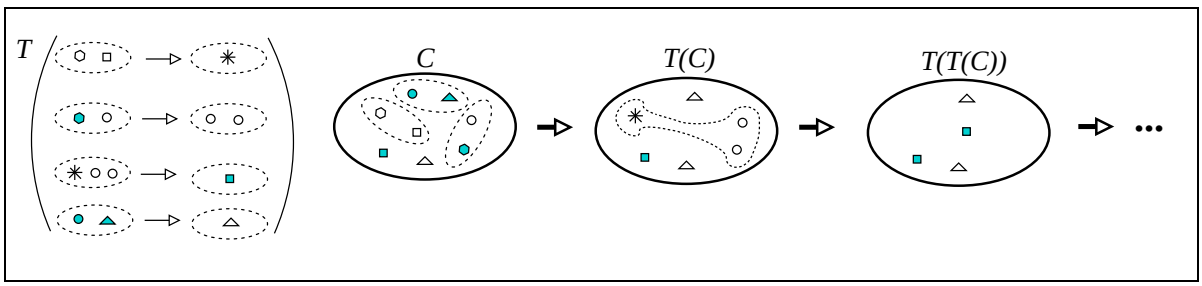


Figure 8: Transformation and iteration of a transformation. A transformation T is a set of basic transformations applied synchronously to make one evolution step. The basic transformations do not interact together. A transformation is then iterated to build the successive states of the system.

subcollection A, then the collection B must be computed not only from A but also from the neighbors of A in the collection C.

This abstract description of one evolution step of a collection makes possible the unification in the same framework of various computational devices. The trick is just to change the organization of the underlying collection. In section 6 we try to reformulate several paradigms (like the CHAM, P systems, L systems and cellular automata), as transformations of some collections.

2.2 Collections as Spaces

As a matter of fact, it is very natural to see these collections as a set of *places* or *positions* labeled by a value. Then, the organization of a collection is seen as a *topology* defining the *neighborhood* of each element in the collection and also the possible subcollections. To stress the importance of the topological organization of the collection's elements, we call them ***topological collection***.

A subcollection is a set of *connected* elements. If the element x in a collection is a neighbor of the element y, we write x, y. Additional conditions can be put to constrain the possible subcollections. A subcollection has itself a topology inherited from the main collection. The topology is used to constrain the possible transformations and the dependances between the collections A, B and C.

For example, one may decide that neighbors of an element in a sequence are its two adjacent elements (except for the first and the last element in the sequence which have only one neighbor). A *subsequence* $\mathcal{C}'$ of $\mathcal{C}$ is a *connected* subset of the elements of $\mathcal{C}$. This means that the possible subsequences of a sequence ℓ are the intervals of ℓ. Additional conditions can be put to constrain the possible subcollections. For instance, one may want to consider only the sequence prefixes or the sequence suffixes for the subcollections, but no arbitrary intervals.

This topological approach to formalizing the notion of collection is part of a long term research effort [GMS96] developed for instance in [Gia00] where the focus is on the substructure, or in [GM01] 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. In this report, we propose a formal framework in section 5 that focuses on the transformation of topological collections, where the basic computation mechanisms is the substitution of subcollections.

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. They are sketched in section 6. So, one may ask if the formal machinery developped in section 5 is worthwhile. Actually, the previous examples show the need of complex topologies. And more importantly, the topological framework unifies various situations. Our ultimate goal is to develop a generic implementation based on these notions.

2.3 The MGS Project: Modeling Biosystems with a Dynamical Structure with Topological Collections and their Transformations

These ideas lead to the development of an experimental programming language called MGS. MGS is the vehicle used to investigate general notions of collections and transformations, and to study their adequacy to the simulation of various biological processes with a dynamical structure.

We will show in section 6 that the notion of topological collection and their transformation are able to take into account in a same unifying framework several biologically and biochemically inspired computational models, namely: Γ and the CHAM, P systems, L systems and cellular automata (CA). We do not claim that topological collection are a useful theoretical framework encompassing all these formalisms. We advocate that few notions and a single syntax can be consistently used to allow their merging *for programming* purposes.

In the current MGS implementations sets, multisets and sequences of elements are supported[3]. This is already a step forward in the quest of a good programming language dedicated to the simulation of biosystems with a dynamical structure. Indeed, even if we restrict to these datatypes, MGS allows some kind of *rewriting* on multisets and sequences. This paradigm is advocated in recent papers for the modeling of biological systems [Man01, FMP00]. To quote[4] Fisher *et al.* [FMP00]:

> « a biological systems is represented as a term of the form $t_1 + t_2 + \cdots + t_n$ where each term t_i represents either an entity or a message [signal, command, information, action, etc.] addressed to an entity. Computation, [i.e., simulation of the physical evolution of the biosystem] is achieved through term rewriting, where the left hand side of a rule typically matches an entity and a message addressed to it, and where the right hand side specifies the entity's updated state, and possibly other messages addressed to other entities. The operator $+$ that joins entities and messages is associative and commutative, achieving an " associative commutative soup", where entities swim around looking for messages addressed to them. [...]
>
> This associative commutative soup allows object to interact in a rather unstructured way, in the sense that an interaction between two objects is enabled simply by virtue of their both being present in the soup. This still does not fully address issues of structural interactions between entities or system parts. »

A severe shortcoming of this view is the *total lack of spatial organization.* The need to represent more structured organizations (than sequence and multiset) of entities and messages is stressed and motivates several extensions of rewriting (see for one example amongst others [BH00]). However, a general drawback with these approaches is that they work with

[3] At the date of may 2001, it exists two versions of an MGS interpreter. One written in OCAML 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 functionnal part of the language. These interpreters are freely available, by sending a demand to [giavitto|michel]@lami.univ-evry.fr.

[4] with some adaptations in the terminology, brackets are our comments

a fixed topology of entities, and it is not obvious at all how to extend this to systems where the number of entities and their relationships are constantly changing.

This is precisely one of the main motivation of the MGS research project. One of our goal is to validate the contribution of the topological approach to the (specification and simulation of the) dynamical organization of biosystems. By superseding the rewriting of terms by the transformation of topological collections, we hope to go beyond the limitations of the preceding formalisms. To paraphrase the previous quotation:

> A collection is used to represent the state of a DS. The elements in the collection represent either entities (a subsystem or an atomic part of the DS) or messages (signal, command, information, action, etc.) addressed to an entity.
>
> A subcollection represents a subset of interacting entities and messages in the system. The evolution of the system is achieved through transformations, where the left hand side of a rule typically matches an entity and a message addressed to it, and where the right hand side 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.

2.4 Organization of the Rest of this Report

The MGS language is presented informally in section 3 through some examples. Simple examples of MGS programs are given in the next section. All examples are processed using the current version of the MGS interpreter.

A possible formalization is presented in section 5 for the theoretically inclined reader. Several formalization of MGS are possible. We present one which is general enough and gives some insights for a generic implementation.

Then, in section 6, we sketch how Γ and the CHAM, P systems, L systems and cellular automata (CA) can be emulated in MGS. Our goal is mainly to sketch the topology of the usual sets, bag, sequence and arrays data structures.

This report finishes by the review of some directions opened by this research.

3 An MGS Quick Tour

MGS is the acronym of *"(encore) un Modèle Général de Simulation (de système dynamique)"* (yet another General Model for the Simulation of dynamical systems). 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. 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.

This section contains a brief description of the main features in MGS concerning functions, collections, transformations and their applications. Elements of the MGS syntax are given through examples. Some examples of actual code are given in this and the following section to give a flavor of the language.

3.1 Functions, Sentences and Programs

MGS is a small higher-order functional language. This means that functions are like any other kinds of primitive values (such as integers, floats, strings, etc.): they are first class values and can be passed as arguments to other functions or returned as results.

Lambdas and Named Lambdas. The denotation of functional values is based on the lambda-calculus; for example:

$$\backslash x.\backslash y.x + y$$

denotes a *curryfied* function expecting one argument value X, that binds to x, and returning a function acting like $\backslash y.X + y$. The syntax of a function application is the usual one with arguments surrounded by brackets: $(\backslash x.\backslash y.x + y)(3)$. Functions may have several arguments. For instance,

$$\backslash (x, y).x + y$$

is a function returning the sum of its two expected arguments. Another syntax is used to give a name to the function:

$$\texttt{fun}\ \mathit{Plus}(x, y) = x + y$$

Such expression creates a functional value and assigns this value to a global variable ($Plus$ in this example), thus the previous definition is equivalent to the expression $Plus = \backslash (x, y).x+y$. We say that *Plus* is a *named lambda*. The name can be used instead of the anonymous lambda-expression in the function application, e.g. $Plus(3, 4)$. Named lambdas enable the direct coding of recursive function:

$$\texttt{fun}\ \mathit{fact}(x)\ =\ \texttt{if}\ (x == 0)\ \texttt{then}\ 1\ \texttt{else}\ x * \mathit{fact}(x - 1)\ \texttt{fi}$$

Primitive Functions. The operator + that appears in the body of *Plus* is an example of a *primitive functional constant* (or " primitive function " in short). There is a rich set of such functions to manage the primitive values. Primitive functions can be used exactly like any other named lambdas, they just are already defined when the interpreter is up. The name of a primitive function always begins with a backquote `'`. Somme primitive functions have an additional special syntax for their application, like + which is the infix form for the application of `'addition`. That is, expression $3 + 4$ is a short hand for the expression `'addition`$(3, 4)$.

Sentences and Programs. An `MGS` program is composed of a sequence of sentences. A sentence finishes by a double semi-colons " ;;". There are three kinds of sentences:

1. expressions,
2. type declarations,
3. commands.

The evaluation of an expression triggers the computation of somme value. The previous definition of function *Plus* is an exemple of an expression with a functional value. The expression $Plus(3, 4)$ is an example of an expression that computes an integer.

A type declaration does not trigger any computation. It just tell the interpreter that some new name can be used at some place in further expressions (see below the use of types as predicates).

Finally, commands are outside the scope of the language and are used to interact with the interpreter beyond the usual *read-eval-print* toplevel loop : one can include some file, divert the output stream, list the named functions or the defined types, save the current session, etc.

3.2 Collections

In addition to basic values like integers, floats, strings, lambda-expressions, etc., `MGS` handles records and several other kinds of collections. The elements in a collection can be any kind of values: basic values, records or arbitrary nesting of collections. The values of the record's fields are also of any kind, thus achieving *complex objects* in the sense of [BNTW95]. Collections are (sub-)typed. The tree in Fig. 9 gives the type hierarchy of collections.

3.2.1 Monoidal Collections

Several kinds of topological collections are supported by `MGS`. We focus here on sets, multisets and sequences. These kinds of collection are called *monoidal* because they can be build as a monoid with operator *join* " ,": a sequence corresponds to a join that has no special property (except associativity), multisets are obtained with commutative joins and sets when the operator is both commutative and idempotent.

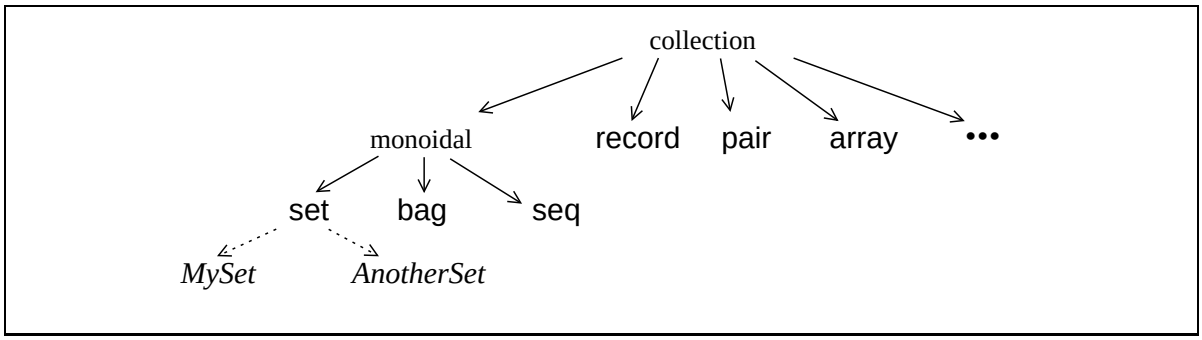


Figure 9: The subtyping hierarchy of collection kinds. *MySet* and *AnotherSet* are user-defined collection types, cf. section 3.2.3. Types `collection` and `monoidal` do not correspond to concrete data structures, but to predicates, cf. below. Conceptually, a record is a set of pairs (*field-name, field-value*) but it is managed through dedicated operators, cf. section 3.3.

There is a large amount of generic operations available for all collection kinds, based on the function algebra developed for instance in [BNTW95]. The following table gives the main constructors for monoidal collections.

	empty	*addition*	*singleton*	*combination*
Set	`set` : ()	`insert`	`single_set`(x)	`union`
Bag	`bag` : ()	`increment`	`single_bag`(x)	`munion`
Seq	`seq` : ()	`::`	`single_seq`(x)	`@`
overloaded syntax		`add` ,		`merge` ,

3.2.2 The topologies of Monoidal Collections

The join operator with its properties directly induces the topology of the collection and the neighborhood relationship. So, it is not a coincidence that the neighborhood relationship in section 2.2 and the join operation here are denoted by the same comma.

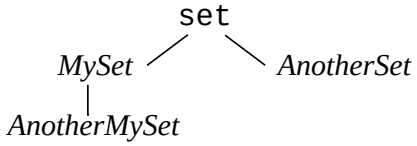
- *Topology of Sets.* In a set, an element x is neighbor of any other element y.
- *Topology of Multisets.* The topology of a multiset is the same as the topology of a set: two arbitrary elements are always neighbors. The difference is, the same element may appear more than one time in the multiset.
- *Topology of Sequences.* The topology of a sequence is the expected one: if the sequence has at least two elements, then all elements except the first and the last have two neighbors (called the *left* and the *right* neighbor). The first and the last element have only one neighbor (respectively a right and a left neighbor). If the sequence is reduced to a singleton, then this singleton as no neighbor.

These neighboring relations are induced by the join operations: if x, y then x is a neighboor of y. For instance, using associtivity and the commutativity of the join of sets, the set " $1,_{\text{set}}\ 2,_{\text{set}}\ 3$ " can be written " $1,_{\text{set}}\ (2,_{\text{set}}\ 3)$ " which shows that 2 and 3 are neighbors, but also it can be written " $(1,_{\text{set}}\ 2),_{\text{set}}\ 3$ " or " $2,_{\text{set}}\ (1,_{\text{set}}\ 3)$ " which show that 1 and 2 are neighboor as well as 1 and 3.

3.2.3 User-Defined Monoidal Subtypes

Often there is a need to distinguish several collections of the same kind (e.g. several multisets nested in another multiset). Various ways can be used to achieve the distinction. For instance, in the P system formalism, each multiset is labeled by a unique integer to reference them unambiguously. We chose to distinguish between collections of the same kind by *types*. The type of a collection must be thought of as a tag that does not change the structure of the collection. Types are organized by a subtyping relationship. The subtyping relation organizes types into a poset. The kind of a collection constitutes the maximal element of this hierarchy. Collection type declarations look like:

$$\begin{aligned} \texttt{collection}\ \mathit{MySet} &= \texttt{set}; \\ \texttt{collection}\ \mathit{AnotherSet} &= \texttt{set}; \\ \texttt{collection}\ \mathit{AnotherMySet} &= \mathit{MySet}; \end{aligned}$$



These three declarations specifies a hierarchy of three types. Type *AnotherMySet* is a subtype of *MySet* which is a subtype of `set`. The type `set` is predefined and corresponds to a collection kind (other predefined types are `seq` for sequences and `bag` for multisets). The type *AnotherSet* is also a subtype of `set` but is not comparable with *MySet*.

A type introduced by a type declaration can later be used in pattern-matching (cf. section 3.4) or as a predicate to test if a value is of a given type. A monoidal collection type can also used in the building of a collection by the enumeration of its elements:

$$1,\ 1+1,\ 2+1,\ 2*2,\ \mathit{MySet} : ()$$

is an expression evaluating to the set of four integers: 1, 2, 3 and 4. The collection kind is a set, and its type is *MySet*. Actually, expression " $\mathit{Myset} : ()$ " denotes the empty *MySet* and " , " is the overloaded join operator: x, X creates a new collection with the element x merged with the elements of collection X; and expression X, Y creates a new collection with elements of both collections X and Y.

The type of a collection is taken into account for several collection operations. For instance, the *join* of two collections of type P and Q gives a collection with type R corresponding to the common ancestor of P and Q. With the previous example, `set` is the common ancestor of *MySet* and *AnotherSet*). Another example, *MySet* is the common ancestor of *AnotherMySet* and itself.

3.2.4 Structural Recursion on Monoidal Collections

The two overloaded operators `oneof` and `rest` are such that for any non empty monoidal collection, we have:

$$C \ = \ \texttt{oneof}(C)\,,\, \texttt{rest}(C)$$

Together with the empty primitive predicate, they makes possible to define a form of structural recursion for monoidal collections:

$$\begin{array}{lcl} \texttt{fun } \mathit{Iter}(e,g) & = & \backslash C.\texttt{if empty}(C) \\ & & \quad \texttt{then } e(C) \\ & & \quad \texttt{else } g(\texttt{oneof}(C), \mathit{Iter}(e,g)(\texttt{rest}(C))) \\ & & \quad \texttt{fi} \end{array}$$

The intent of the expression $\mathit{Iter}(e,g)$ is to define a function h such that:

$$\begin{array}{rcl} h(\texttt{X}:()) & = & e(\texttt{X}:()) \\ h((a,C)) & = & g(a,h(C)) \end{array}$$

where X is the kind of the collection, e a unary function that gives the value of h on the empty collection and g a combining binary function. Please note that h is a unary function, so in expression $h((a,C))$ the function h is applied to the collection built by the join[5] of a with C.

This kind of function definitions (which define homomorphisms) is so common that *Iter* is a primitive function called fold in MGS:

$$h = \texttt{fold}[g,e]$$

Note that square brackets are used instead of braces because the arguments g and e are optional arguments with some default values[6] (the default values are such that fold defines the identity function on collection).

The function fold is called an iterator. An iterator can be used to easely define very useful other functions. We give three examples. The sum of all elements in a collection can be defined by:

$$\texttt{fold}[\backslash(a,c).a+c, \backslash x.0]$$

As a second example, the famous map function is defined in MGS as:

$$\texttt{fun map}(f) = \texttt{fold}[\backslash(a,c).(f(a),c), \backslash x.x]$$

Note that with this definition, we have a generic map that can act on any monoidal collection. On sets for instance, the meaning[7] of map is $\texttt{map}(f)(\{a_1,..,a_n\}) = \{f(a_1),..,f(a_n)\}$. Finally, the generalization of the powerset function to the other collection kinds can be defined as:

$$\mathit{Power} = \texttt{fold}[\backslash(a,C).(C, \texttt{map}(\backslash c.(a,c))(C)), \backslash x.\texttt{add}(x,x)]$$

[5] When there is an ambiguity between the application of a function to several arguments and the join of several arguments, the former interpretation is chosen. Braces can be used to force the other interpretation, as in this case here.

[6] We do not detail further these features as they are not relevant for our purpose here.

[7] Be careful that, for the sake of the explanation, we use the notation $\{a,b,c\}$ to denote the set of the three elements a, b and c. Previously we have used the notation $\{\!|\, a,b,c \,|\!\}$ for a multiset. And below, we use the notation $<a,b,c>$ to express the sequence of the three elements a, b and c. However, these constructions are not part of the MGS syntax. The building of a collection through the enumeration of its elements uses the join operator in MGS.

where add(x, C) adds the element x to the collection C. On sets, the $Power(S)$ build the power set of S. On sequences, $Power(L)$ built a sequence of all the subsequences of the list L; for instance,

$$Power(\text{<1, 2, 3>}) = \text{<<>, <3>, <2>, <2, 3>, <1>, <1, 3>, <1, 2>, <1, 2, 3>>} .$$

3.3 Records

An MGS record is a special kind of collection. An MGS record is a map that associates a value to a name called *field*. The value can be of any type, including records or other collections. Accessing the value of a field in a record is achieved with the dot notation: expression $\{a = 1,\ b = \texttt{"red"}\}.b$ evaluates to the string "red".

Records can be merged with the overloaded $+$ operator. Expression $r_1 + r_2$ computes a new record r having the fields of both r_1 and r_2. Then $r.a$ has the value of $r_2.a$ if the field a belongs to r_2, else the value of $r_1.a$ (asymmetric merge with priority to the second argument [Rém92]).

For records, type declarations look like

$$\begin{aligned} \texttt{state}\ R &= \{a\}; \\ \texttt{state}\ S &= \{b, \tilde{}c\} + R; \\ \texttt{state}\ T &= S + \{a = 1,\ d : \texttt{string}\}; \end{aligned}$$

(state is the keyword used to introduce the definition of a record type in MGS). The first declaration specifies a record type R which consists of the records with at least a field named a. Types can be used as predicates:

$$R(\{a = 2,\ x = 3\})$$

evaluates to true because the record $\{a = 2,\ x = 3\})$ has a field a. The second declaration defines S which has all the fields of R plus a field b and *no* field c. The $+$ operator between record types emulates a kind of inheritance. The definition T specializes type S by constraining the field a to the value 1 and saying that an additional field d must be present and be a string.

3.4 Pattern, Rule and Transformations

A transformation T is a set of basic transformations or *rules*:

$$\texttt{trans}\ T = \{\ \ldots \quad \mathit{rule}; \quad \ldots\ \}$$

When there is only one rule in the transformation, the enclosing brackets can be dropped. A transformation is a first-class value and some operators exist to combine transformations. For instance, the transformation $(T_1 + T_2)$ is the transformation obtained by merging the set of rules of T_1 and the set of rules of T_2.

A rule is a basic transformation taking the following form:

$$pattern \ => \ expression$$

where *pattern* in the left hand side (lhs) of the rule matches a subcollection A of the collection C on which the transformation is applied. The subcollection A is substituted in C by the collection B computed by the *expression* in the right hand side (rhs) of the rule. There are also several kinds of rules, as detailed below.

3.4.1 Patterns

We present the pattern expressions that have a generic meaning, that is, they can be interpreted against any collection kind. The grammar of the pattern expressions is:

$$Pat ::= \ x \ \mid \ \{...\} \ \mid \ p, p' \ \mid \ p+ \ \mid \ p* \ \mid \ p : P \ \mid \ p/exp \ \mid \ p \ \texttt{as} \ x \ \mid \ (p)$$

where p, p' are patterns, x ranges over the pattern variables, P is a predicate and exp is an expression evaluationg to a boolean value. The explanations below give an informal semantics for these patterns.

variable: a pattern variable x matches exactly one element. The variable x can then occur elsewhere in the rest of the rule both as a pattern or in an expression. Actually, the pattern x is the abbreviation of " . as x" where the pattern " ." matches exactly one element.

state pattern: $\{...\}$ are used to match one element which is a record. The content of the braces can be used to match records with or without a specific field (eventually constrained to a given field type or field value). For instance, $\{a, \ b : \texttt{string}, \ c = 3, \ \tilde{} d\}$ is a pattern that matches a record with fields a, b of type `string` and c with value 3, and no field d.

neighbor: p, p' is a pattern that matches two connected collections p and p'. For example, x, y matches two connected elements (i.e., x must be a neighbor of y). The connection relationship depends of the collection kind.

repetition: pattern $p+$ (resp. $p*$) matches a non empty subcollection of elements matched by p (resp. a possibly empty subcollection).

binding: a binding p as x gives the name x to the collection matched by p. This name can be used elsewhered in the rest of the rule. The evaluation of a pattern variable x in an expression returns the subcollection previously matched. When reused as a pattern variable, the pattern x is interpreted as $(y/y == x)$ where y is a fresh variable. For example, x, x is equivalent to $x, (y/y == x)$.

guard: p/exp matches the collections matched by the pattern p verifying *exp*. Pattern $p : P$ is a syntactic suggar for $((p \ \texttt{as} \ x)/P(x))$ where x is a fresh variable. For instance, $x : MySet$ filters an element of type *MySet*. Another example: $y \, / \, y > 3$ matches an integer bigger than 3.

Here is a contrived example. The pattern

$$(x : int/x < 3) + \ as\ S \quad / \quad card(S) < 5 \quad \& \quad Fold[+](S) > 10$$

selects a subcollection S of integers less than 3, such that the cardinality of S is less than 5 and the sum of the elements in S is greater than 10. If this pattern is used against a sequence (resp. set) (resp. multiset), S denotes a subsequence (resp. a subset) (resp. a sub-multiset).

Some pattern constructs are specific to a collection kind. For example, the construct "$\hat{}, x$" is used to select an element which has no left neighbor in a sequence. Such pattern has no meaning when the transformation is applied for instance to a set, and an error is raised. Another example of a specific construct are the operators *left* and *right*. They can be used in the guard of a pattern (or in the rhs of a rule) to refer to the element to the right or to the left of a matched subsequence. These constructions depend on the topology of the collection and we plan to develop a generic and systematic specification of these operators using the notion of boundary.

3.4.2 Rules

A transformation is a set of rules. When a transformation is applied to a collection, the strategy is to apply as many rules as possible in parallel. A rule can be applied if its pattern matches a subcollection. Several features are used to have a finer control over the choice of the rules applied within a transformation.

Exclusive and inclusive rules. *Exclusive rules* consume their arguments: a subcollection matched by an exclusive rule cannot intersect a subcollection matched by any other rule.

Inclusive rules don't have this kind of constraint. They are mainly used to transform independent parts of a complex object[8]. This is best explained by an example:

$$\{x \ \texttt{as}\ v\} \quad +=> \quad \{x = v + 1\}$$
$$\{y \ \texttt{as}\ v\} \quad +=> \quad \{y = 2 * v\}$$

are two inclusive rules (because the arrow is $+=>$) matching respectively a record with a field x and a record with a field y. So they can both apply to the record $\{x = 2,\ y = 3\}$. An inclusive rule of form $r +=> r'$ where r is a record pattern and r' an expression evaluating to a record, replaces the matched record R by $R + r'$. So, the result of applying the two previous rules to $\{x = 2,\ y = 3\}$ is $\{x = 3,\ y = 6\}$. This result is computed as

$$\Big(\{x = 2,\ y = 3\} + \{x = 2 + 1\}\Big) + \{y = 2 * 3\}$$

or

$$\Big(\{x = 2,\ y = 3\} + \{y = 2 * 3\}\Big) + \{x = 2 + 1\}$$

and is independent of the order of application of the two rules. Indeed, the rules work on independent parts of the record, both for accessing or updating the value of a field.

[8] Currently, only a rhs matching a record is allowed in an inclusive rule, but the idea must be extended to nested collections. The concept of inclusive rule may appear very specific. However, it is a very effective way to cut down the combinatorial explosion of the behavior specifications.

Priority. Exclusive rules are applied before any inclusive rules. A priority can be associated to each rule, to specify a precedence order within each class (the priority of inclusive rules may be used to specify the relative order of their applications).

Local variables and conditional rules. MGS is not a purely functional language. Imperative local variables can be attached to a transformation and updated by side effects in the rhs of the rules. These variables can be used in a rule guard allowing the conditional use of a rule. For instance, the transformation

$$\texttt{trans}\ T[a=0] \quad = \{...; \quad R\ =\ x\ =\{\ \texttt{on}\ a<5\ \}=>\ \ (a:=a+1;\ 2*x); \quad ...\}$$

specifies a rule R which is applied at most 5 times (within the evaluations triggered by one application of T). The *body* $=\{\cdots\}=$ of the arrow defines an " on clause ". The expression linked to the on is used to decide if the rule is eligible for a transformation or not. The decision occurs befor any attempt to match a subcollection. The semi-colon in the rhs of the rule denotes the sequencing of two evaluations. As a consequence, the local imperative variable a, initialized to 0 when T is applied, counts the number of applications of rule R and the rule can apply only if a is less than 5. The initial value of a variable local to a transformation can be overridden when the transformation is applied; for instance the evaluation of $T[a=3](...)$ enables at most 2 uses of rule R.

3.5 Managing the Applications of a Transformation

A transformation T is a function like any other function and a *first-class* value. It makes possible to sequence and compose transformations very easily.

The expression $T(C)$ denotes the application of one transformation step 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:

$T[\texttt{'iter} = n]\,(C)$	denotes the application of n transformation steps to C
$T[\texttt{'fixpoint}]\,(C)$	application of the transformation T until a fixpoint is reached
$T[\texttt{'fixrule}]\,(C)$	idem but the fixpoint is detected when no rule applies

In addition to the standard transformation step strategy, two other *application modes* exist. In the *stochastic mode*, the choice of the exclusive rule to apply is made randomly. The priorities of the exclusive rules are then considered as the relative probability of their effective application (when they can apply). In *asynchronous mode*, only one exclusive rule is applied in one transformation step.

4 Examples of MGS Programs

The following examples are freely inspired by examples given for Γ, P systems, L systems and the 81/2 language [Mic96].

4.1 Maximal Element

This example is a fundamental one, because it emphasizes the ability to express in MGS meaningful transformations able to act on several collection kinds. The transformation

$$\begin{array}{l}\texttt{trans}\ Max\ =\ \{\\ \quad x, y/\ (x > y)\ =>\ x\ ;\\ \quad x, y/\ (x < y)\ =>\ y\ ;\\ \}\end{array}$$

can be used both on a set, a multiset or a sequence. On a set, it computes the maximal element in the set; on a sequence it computes the maximal element(s) in the multiset; and on a sequence, it computes a sequence composed only of the maximal element of the initial argument. For instance,

$$\begin{array}{rcl} Max[\texttt{'fixrule}](1,2,2,1,0,2,\texttt{set} : ())) & = & 2,\texttt{set} : () \\ Max[\texttt{'fixrule}](1,2,2,1,0,2,\texttt{bag} : ())) & = & 2,2,2,\texttt{bag} : () \\ Max[\texttt{'fixrule}](1,2,2,1,0,2,\texttt{seq} : ())) & = & 2,2,2,\texttt{seq} : () \end{array}$$

Note that the second rule of the transformation is necessary only to handle sequences in the same manner, because for sets and multisets, if x, y the we have also y, x, see section 6.1.

4.2 Map and Sum

The sum of all elements in a collection of numbers can be computed by transformation

$$\texttt{trans}\ sum =\ a, b => a + b\ ;;$$

It is easy to achieve a function map with the transformation:

$$\texttt{trans}\ MAPF =\ a => f(a)\ ;;$$

This example can be elaborated to be parameterized by the function f:

$$\begin{array}{rcl}\texttt{trans}\ MAP\,[fun = \backslash a.a] & = & a => fun(a)\ ;;\\ \texttt{fun}\ map(f, C) & = & MAP[fun = f](C)\ ;;\end{array}$$

Note that the function *map* is a function wrapper that applies one step of transformation MAP to its argument C. This transformation is parameterized through an optional argument *fun* which takes the function to be applied. The default value for function *fun* is the identity, that is: $MAP(C) \equiv C$. The transformation consists in substituting a by $fun(a)$. Since as many rule instantiations as possible are done in parallel within one step, the only rule of the transformation is applied to each element in the collection.

4.3 Sorting a Sequence

A kind of bubble-sort is immediate:

```
trans Sort = (x, y / y < x) => y, x;
```

(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.)

4.4 Convex Hull

The convex hull of a set P of points in the plane is defined to be the smallest convex polygon containing them all. It is easy to show that the vertices of the convex hull of P are elements of P. The program to compute the convex hull considers a point X and a triple of points U, V and W and eliminates X if it falls inside the triangle U, V, W.

We first define a record *Point* which has a field x and a field y. We define also two variables named *true* and *false* for convenience (however each value can be interpreted as a boolean when needed as in the `C` programming language).

```
state Point = {x, y};;
false := 0;;   true := ~false;;
```

A point X falls inside the points U, V and W iff it exists α, β and γ between 0 and 1 such that: $\alpha U + \beta V + \gamma V = X$ and $\alpha + \beta + \gamma = 1$. This gives a linear system of three equations with three unknowns α, β and γ wich can be solved using the determinant method. This explains the function *inside* defined below. The function *det* computes a 3×3 determinant; the function *check* tests if a value is between 0 and 1; and *inside2* is an auxilliary function that does the real work.

```
fun check(d) = (d >= 1)||(d <= 0);;
fun det(a, b, c, d, e, f, g, h, i) =
    a * (e * i - h * f) - d * (b * i - h * c) + g * (b * f - e * c);;
fun inside(X, U, V, W) =
    inside2(X, U, V, W, det(U.x, V.x, W.x, U.y, V.y, W.y, 1, 1, 1));;
fun inside2(X, U, V, W, d) =
    if d == 0 then false
     else if check(det(X.x, V.x, W.x, X.y, V.y, W.y, 1, 1, 1)/d) then false
     else if check(det(U.x, X.x, W.x, U.y, X.y, W.y, 1, 1, 1)/d) then false
     else if check(det(U.x, V.x, X.x, U.y, V.y, X.y, 1, 1, 1)/d) then false
     else true
     fi fi fi fi;;
```

The function *inside* is used in the guard of the transformation:

$$\texttt{trans}\ \mathit{Convex} = \quad X, U, V, W/\ \mathit{inside}(X, U, V, W) => U, V, W;;$$

To test our program, we compute the convex hull of various points lying inside the square delimited by $(0, 0)$ and $(1, 1)$, including the four corners:

$$\begin{array}{l}
\mathit{Convex}[\texttt{'fixrule}]((\\
\quad \{x = 0,\ y = 0\}, \\
\quad \{x = 0.2,\ y = 0.1\}, \\
\quad \{x = 0.5,\ y = 0.7\}, \\
\quad \{x = 1,\ y = 0\}, \\
\quad \{x = 0.1,\ y = 0.2\}, \\
\quad \{x = 1,\ y = 1\}, \\
\quad \{x = 0.2,\ y = 0.4\}, \\
\quad \{x = 0.4,\ y = 0.6\}, \\
\quad \{x = 0,\ y = 1\}, \\
\quad \texttt{set} : () \\
));;
\end{array}$$

computes the expected result:

$$\{x = 0,\ y = 0\},\ \{x = 0,\ y = 1\},\ \{x = 1,\ y = 0\},\ \{x = 1,\ y = 1\},\ () : \texttt{set}$$

4.5 Eratosthene's Sieve on a Set

The idea is to generate a set with integers from 2 to N (with transformation *Generate* and *Succeed*) and to replace an x and an y such that x divides y by x (transformation *Eliminate*). The results is the set of prime integers.

$$\begin{array}{rcl}
\texttt{trans}\ \mathit{Generate} & = & \{x, true\}\ => \ x, \{x + 1, true\}; \\
\texttt{trans}\ \mathit{Succedd} & = & \{x, true\}\ => \ x; \\
\texttt{trans}\ \mathit{Eliminate} & = & (x, y\ /\ y\,\texttt{mod}\,x = 0)\ => \ x;
\end{array}$$

With this program, the expression

$$\mathit{Eliminate}[\texttt{'fixrule}](\mathit{Succeed}(\mathit{Generate}[N](\{2, true\}, \texttt{set} : ())))$$

computes the primes up to N (and we can turn this expression into a function by abstracting on N).

4.6 Eratosthene's Sieve on a Sequence

The idea is to refine the previous algorithm using a sequence. Each element i in the sequence corresponds to the previously computed ith prime P_i and is represented by a record $\{prime = P_i\}$. This element can receive a candidate number n, which is represented by a record

$\{prime = P_i, candidate = n\}$. If the candidate satisfies the test, then the element transforms itself to a record $r = \{prime = P_i, ok = n\}$. If the right neighbor of r is of form $\{prime = P_{i+1}\}$, then the candidate n skips 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.

$$
\begin{array}{rcl}
\texttt{trans}\ Eratos = \{ & & \\
Genere1 & = & n : integer\ /\ \tilde{}\texttt{right}\ n\ \Rightarrow\ n, \{prime = n\}; \\
Genere2 & = & n : integer, \{prime\ \texttt{as}\ x, \tilde{}candidate, \tilde{}ok\} \\
 & & \Rightarrow\ n+1, \{prime = x, candidate = n\}; \\
Test1 & = & \{prime\ \texttt{as}\ x, candidate\ \texttt{as}\ y, \tilde{}ok\}\ /\ y\,\texttt{mod}\,x = 0\ \Rightarrow\ \{prime = x\}; \\
Test2 & = & \{prime\ \texttt{as}\ x, candidate\ \texttt{as}\ y, \tilde{}ok\}\ /\ y\,\texttt{mod}\,x <> 0 \\
 & & \Rightarrow\ \{prime = x, ok = y\}; \\
Next & = & \{prime\ \texttt{as}\ x1, ok\ \texttt{as}\ y\}, \{prime\ \texttt{as}\ x2, \tilde{}ok, \tilde{}candidate\} \\
 & & \Rightarrow\ \{prime = x1\}, \{prime = x2, candidate = y\}; \\
NextCreate & = & \{prime\ \texttt{as}\ x, ok\ \texttt{as}\ y\}\ \texttt{as}\ s\ /\ \tilde{}\texttt{right}\ s \\
 & & \Rightarrow\ \{prime = x\}, \{prime = y\}; \\
\} & &
\end{array}
$$

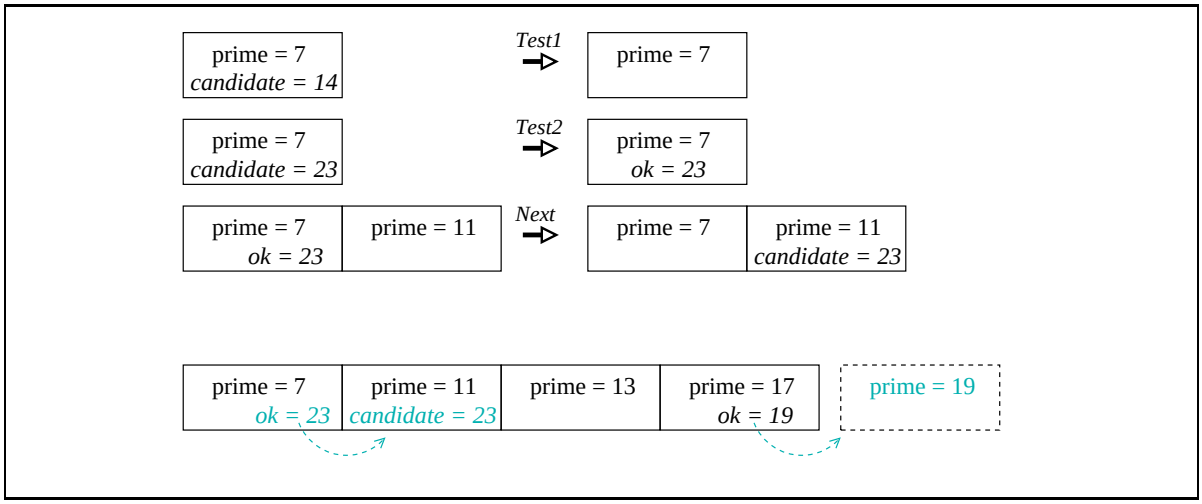


Figure 10: The *Eratos* program. Some rule instantiations and a fragment of the sequence built by the transformation *Eratos*.

We have given an explicit name to each rule. See an illustration on Fig. 10. The expression $Eratos[N]((2, \texttt{seq} : ()))$ executes N steps of the Eratosthene's sieve. For instance $Eratos[100]((2, \texttt{seq} : ()))$ computes the sequence: $42, \{candidate = 42, prime = 2\}, \{ok = 41, prime = 3\}, \{prime = 5\}, \{prime = 7\}, \{prime = 11\}, \{prime = 13\}, \{ok = 37, prime = 17\}, \{prime = 19\}, \{prime = 23\}, \{prime = 29\}, \{prime = 31\}, \texttt{seq} : ()$.

4.7 Maximum Segment Sum

Consider the problem of finding the segment of maximal sum in a sequence of numbers. For instance, in sequence <1, 2, -3, 2, 2, -1> the maximum segment sum is the segment <2, 2>. This optimization problem can be solved by dynamical programming. The corresponding algorithm is easely stated in MGS.

We first transform a sequence of numbers into a sequence of records. A record at position p has a field val which records the number at position p in the initial sequence, a field sum which holds the sum of the current computed maximal segment endings at position p and a field named $indices$ which contains the positions of the elements of the current segment ending at p. Initially, the current segment that ends at position p also begins at position p. Thus:

$$\begin{aligned}\texttt{trans}\ init\,[p=0] \quad &= \quad (x/\ \texttt{record}(x)) \\ &\quad => \ (p := p+1; \{val = x, sum = x, indices = (p, \texttt{set} : ())\})\end{aligned}$$

For instance, $init(<21, -5, 7>)$ computes $<\{val = 21, sum = 21, indices = \{1\}\}, \{val = -5, sum = -5, indices = \{2\}\}, \{val = 7, sum = 7, indices = \{3\}\}>$.

Then, we can combine a segment ending at position p and a segment at position $p+1$ to gives a segment at position $p+1$ if this increase the local score:

$$\begin{aligned}&\texttt{trans}\ all\_max\_sum = \\ &\quad ((x,y)/(y.sum < (x.sum + y.val))) \\ &\quad => x, y + \{val = y.val, sum = x.sum + y.val, indices = x.indices@y.indices\};;\end{aligned}$$

This transformation must be iterated until fixpoint. Then, the maximal segment sum can be extracted:

$$\begin{aligned}&\texttt{trans}\ max\_sum = \{ \\ &\quad x, y/x.sum > y.sum => x; \\ &\quad x, y/x.sum < y.sum => y; \\ &\};;\end{aligned}$$

The whole process can be sumarized in a function:

$$\texttt{fun}\ mss(C) = max\_sum[\texttt{'fixrule}](all\_max\_sum[\texttt{'fixrule}](init(C)));;$$

4.8 Tokenization

The tokenization problem can be stated as follows: it is required to process a sequence of letters to obtain the multiset of words constituting the sequence. A word is a sequence of letters without white space.

The solution, a two transformations long MGS program, relies on a nested collections structure. On the top level, we have a multiset and the elements of this multiset are sequences which finally must be without white space.

We first defines two new types:

$$\texttt{collection } \mathit{Word} = \texttt{seq};;$$
$$\texttt{state } \mathit{Split} = \{\mathit{before}, \mathit{after}\};;$$

The type *Word* is just a distinguished sequence type used to representes the words[9]. The record *Split* will be used to record the two parts of a sequence splitted when a white space is detected. The rule:

$$\begin{aligned}\texttt{trans } \mathit{CutSeq} \;=\;\; & (x/\; x\,!=\text{`` ''})+ \;\texttt{as}\; X,\;\; (y/\; y==\text{`` ''}),\;\; (z+ \;\texttt{as}\; Z)\\ & => \{\mathit{before} = X, \mathit{after} = Z\};;\end{aligned}$$

applied on a sequence, gives a new sequence. If there is a white space " " in the sequence, the the pattern « $(x/\; x\,!=\text{`` ''})+$ as X » filters, in a subsequence named X, all the non-white space letters until the first occurence of a white space binded to y. Then Z binds to the rest of the sequence. The result computed by $\{\mathit{before} = X, \mathit{after} = Z\}$ is a sequence containing only one element, a record of type *Split*. If there is no white space on the sequence, the rule does not apply and the transformation is the identity

Recall that a transformation acts by applying rules on subsequences and the results are gathered in a sequence. This is why the results of applying *CutSeq* is always a sequence, even if the entire sequence is matched by the rule[10].

The second transformation apply *CutSeq* on the elements of a multiset and extract the result of a split from the englobing sequence:

$$\begin{aligned}\texttt{trans } \mathit{Cut} = \;\; & \{\\ & \quad x/\mathit{Split}(\texttt{hd}\, x) \;=>\; (\texttt{hd}\, x).\mathit{before},\;\; (\texttt{hd}x).\mathit{after},\;\; \texttt{bag} : ();\\ & \quad x \;=>\; \mathit{CutSeq}(x);\\ & \}\end{aligned}$$

The first rule of this transformation is applied if the first element of a sequence is a *Split*. In this case, the two fields of the *Split* are extracted and constitute the elements that are added into the multiset in place of the matched sequence. The second rule apply the transformation *CutSeq* to an element. It is important to give the two rules in this order. As a matter of fact, the second rule can always apply (because there is no guard, the pattern x matches any element in the multiset). But we want to apply this rule only if the element is not a split.

For example, see Fig. 11, the expression (the transformation *Cut* applied until fixpoint to a multiset of one element, this element being a sequence *Word*):

$$\mathit{Cut}[\texttt{'fixpoint}](((\text{"}a\text{"},\text{"}b\text{"},\text{"}c\text{"},\text{" "},\text{"}d\text{"},\text{"}e\text{"},\text{" "},\text{"}f\text{"},\text{"}g\text{"},\text{"}h\text{"}, \mathit{Word} : ()), \texttt{bag} : ()));;$$

[9] Instead of letters, we use here strings (written between double quotes) to represent the elements of the words, because the current interpreter does not offer letters as a basic type.

[10] We are devising mechanisms to ease the " dissolving " of a nested collection, in a manner analog of the dissolve operator used in P systems. Here we use a rule in the transformation *Cut*.

evaluates to

$$(\text{"}a\text{"},\text{"}b\text{"},\text{"}c\text{"},():Word),\ (\text{"}d\text{"},\text{"}e\text{"},():Word),\ (\text{"}f\text{"},\text{"}g\text{"},\text{"}h\text{"},():Word),\ \ ():\texttt{bag}$$

that is, a bag of three elements, each element being a word without white space. See figure 11.

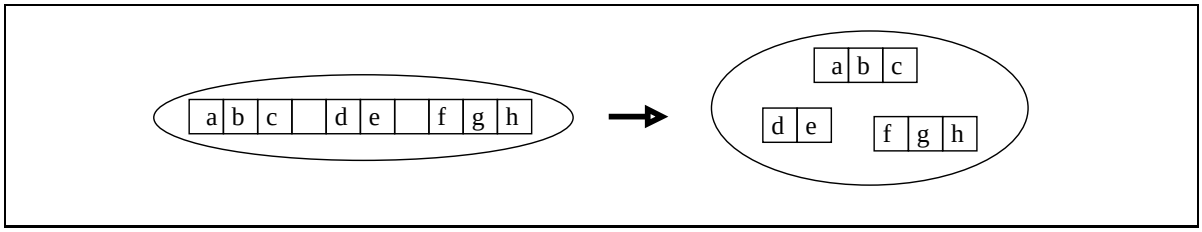


Figure 11: Tokenisation of a sequence of letters

4.9 Token moving on a Ring

The problem is just to propagate a token on a ring. The idea is to use a rule

$$(x/x == 1),\ (y/y == 0) => 0, 1;$$

to say that the token " 1" propagates in a medium of 0. However, the topology of a ring is not directly accessible as a collection kind (not yet). But it can be emulated by a sequence and by managing explicitly what occurs for the begin and the end of the sequence. Instead of written one rule, we have to write three rules. The rule for the first element looks like:

$$z/(z == 0) \quad \& \quad \tilde{}\texttt{left}\ z \quad \& \quad \ldots \quad => \quad 1;$$

where the condition $\tilde{}\texttt{left}\ z$ specifies that z is the first element in the sequence (it has no element to its left) and the condition $z == 0$ ensure that it is not occupied by a token. It remains to check that the last element of the sequence is occupied by a one.

For, we have to refer to the global collection on which the transformation is acting. This is possible, using simply an additional parameter of the transformation. When we apply the transformation, we arrange to pass the collection both as the argument and as the value of the additional parameter of the transformation (using a wrapper). The corresponding program is:

$$\begin{array}{l}
\texttt{trans}\ \mathit{Tore}[\mathit{self}] = \{ \\
\quad (x/x == 1),\ (y/y == 0) \quad => \quad 0, 1; \\
\quad y/(y == 1)\ \&\ \tilde{}\texttt{right}\ y\ \&\ (0 == \texttt{hd}(\mathit{self})) \quad => \quad 0; \\
\quad z/(z == 0)\ \&\ \tilde{}\texttt{left}\ z\ \&\ (1 == \texttt{last}(\mathit{self})) \quad => \quad 1; \\
\};; \\
\texttt{fun}\ \mathit{tore}(t) = \mathit{Tore}[\mathit{self} = t](t);;
\end{array}$$

The operators `hd` and `last` give the first and the last element in a sequence. The function *tore* is the wrapper of the transformation *Tore*. An n-times iteration of the transformation is

then simply obtained by iterating the function *tore* n-times, which is realized with the same syntax as the iteration of a transformation : $tore[n](\dots)$. The 6th first iterations starting from a ring with 5 element and just one token, give:

$0, 0, 1, 0, 0$

$0, 0, 0, 1, 0$

$0, 0, 0, 0, 1$

$1, 0, 0, 0, 0$

$0, 1, 0, 0, 0$

$0, 0, 1, 0, 0$

This program gives an example of the smooth interplay between transformations and functions, and the use of additional arguments in a transformation.

Moving a token on a ring is not very interesting. Instead of moving one token, one can diffuse two morphogenes that, in addition, react together. This process is sketched in the next section. The previous idea is used to diffuse on a ring emulated on a sequence. The results of the MGS program are output in a Mathematica readable form, for the purpose of visualization. The result is plotted in figure 12. We do not give the corresponding MGS code because it simply combines the previous idea with the Turing diffusion-reaction process described below. For information, the MGS code takes 75 lines, including 35 lines dedicated to format the output for Mathematica while an hand-coded C program takes 70 lines to only compute the diffusion-reaction process.

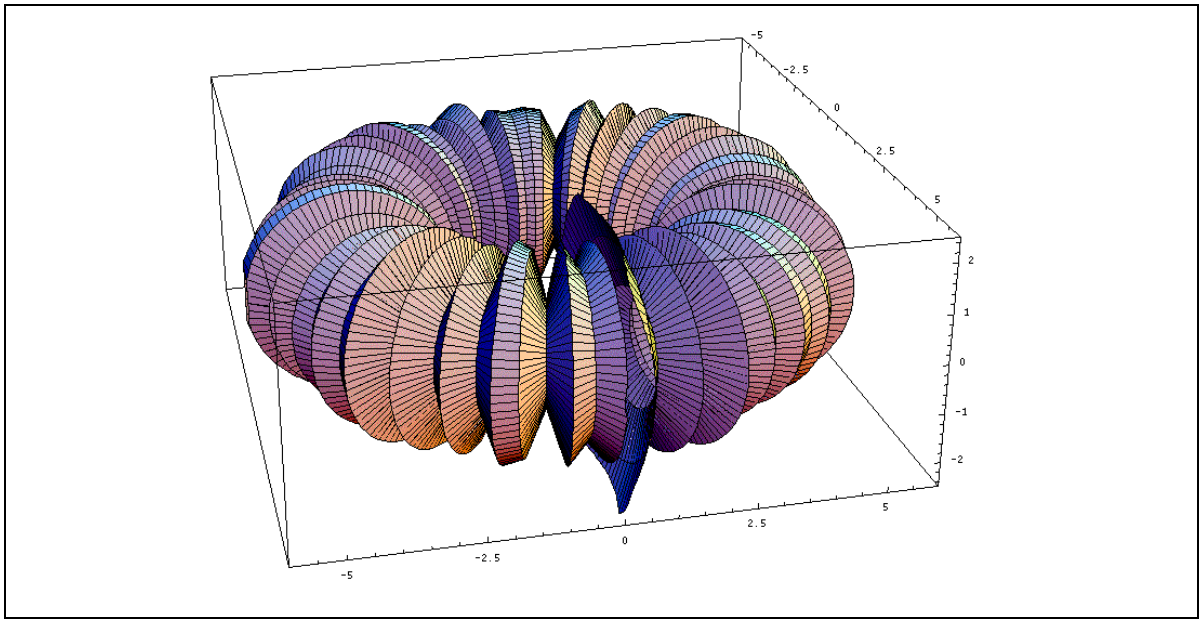


Figure 12: Example of a Turing diffusion-reaction process on a ring. Each cell of the ring is rendered by a slice of the torus. The diameter of the slice is proportional to the b morphogene (cf. text of section 4.10). The results computed by the MGS program are written in a file later read by Mathematica. This file contains both the computed data and a Mathematica program used to compute the coordinate of the torus and to render the 3D objects. This figure plots a gif capture of the graphics rendered by Mathematica when reading the MGS produced file.

4.10 Morphogenesis Triggered by a Turing Diffusion-Reaction Process

Alan Turing proposed a model of chemical reaction coupled with a diffusion processus in cells to explain patterns formation. The system of differential equations [BL74] is:

$$\begin{aligned} da_r/dt &= 1/16(16 - a_r b_r) + (a_{r+1} - 2a_r + a_{r-1}) \\ db_r/dt &= 1/16(16 - b_r - \beta) + (b_{r+1} - 2b_r + b_{r-1}) \end{aligned}$$

where a and b are two chemical reactives that diffuse on a discrete segment of cells indexed by r. This model mixes a continuous phenomena (the chemical reaction in time) and a discrete diffusion process. In MGS we retrive these equations, three times, to handle the cell at the two ends of the segment (rule $evol\_left$ and $evol\_right$) and the cells with two neighboors (rule $evol$).

In addition, we complexify this processus by splitting one cell in two if the level of the morphogen b is greater than a given level (rule $Split$). This process does not correspond to any real biosystems, see however [HP96].

The corrresponding program starts by a transformation used to generate the initial sequence of cells.

$$\begin{array}{l} \texttt{trans}\ init = \\ \quad x => \{ \\ \qquad a = 3.5 + \texttt{random}(1.0) - 0.5, \\ \qquad b = 4.0, \\ \qquad beta = 12.0 + \texttt{random}(0.05 * 2.0) - 0.05, \\ \qquad size = 16 \\ \}; \\ \\ rsp := 1.0/16.0;; \\ diff1 := 0.25;; \\ diff2 := 0.0625;; \\ NbCell := 18;; \\ \\ segment0 := init[1](iota(NbCell, () : \texttt{seq}));; \end{array}$$

The $init$ transformation is used to generate the initial sequence of cells $segment0$. Applied one times to a sequence of n arbitrary elements, it generates a sequence of records. The field a and b of the record corresponds to the morphogens. The field $beta$ is an auxilliary variable of the diffusion-reaction process: it corresponds to a constant with some noise. The field $size$ is used for the 3D output, see figure 13 and annex B. The expression $iota(NbCell, () : \texttt{seq})$ build a sequence made of the integers from 0 to $NbCell$.

The real computation takes place in the $Turing$ transformation. One rule is used to split a cell that reach the adequate level of morphogen b and three other rules are used for the reaction-diffusion process. The functions da and db computes the increases in morphogen a

and b respectively.

```
fun da(a,b,la,ra) = rsp*(16.0-a*b)+diff1*(la+ra-2.0*a);;
fun db(a,b,beta,lb,rb) = rsp*(a*b-b-beta)+diff2*(lb+rb-2.0*b);;

trans Turing = {
  Split =
    (x/x.b > 8) =>
      {a = x.a/2, b = x.b/2, beta = x.beta, size = x.size/2},
      {a = x.a/2, b = x.b/2, beta = x.beta, size = x.size - x.size/2};
  evol =
    (x/(left x)&(right x))+=>
      {a = x.a + da(x.a, x.b, (left x).a, (right x).a),
       b = Max(0.0, x.b + db(x.a, x.b, x.beta, (left x).b, (right x).b))};
  evol_right =
    (x/~left x)+=>
      {a = x.a + da(x.a, x.b, 0, (right x).a),
       b = Max(0.0, x.b + db(x.a, x.b, x.beta, 0, (right x).b))};
  evol_left =
    (x/~right x)+=>
      {a = x.a + da(x.a, x.b, (left x).a, 0),
       b = Max(0.0, x.b + db(x.a, x.b, x.beta, (left x).b, 0))};
};;
```

The rest of the code is used to trigger the computation and to output the results. The output is done in a dedicated language used to visualize 3D scenes. The result is plotted in figure 13. The functions $showBarre$, pre_show and $post\_show$ are detailed in annex B. This code is very short and easy to program, because the language used to produce the scene is very expressive.

```
fun showBarre(barre,t,tmax) = ...;;
fun pre_show() = ...;;
fun post_show(n,c) = ...;;
fun evol(barre,t,tmax) =
(
  showBarre(barre,t,tmax);
  if (t < tmax)    then evol(Turing[iter = 1](barre), t+1, tmax)
   else barre fi
);
fun evolve(n) = (pre_show();  evol(segment0,0,n);  post_show(n,NbCell));;
```

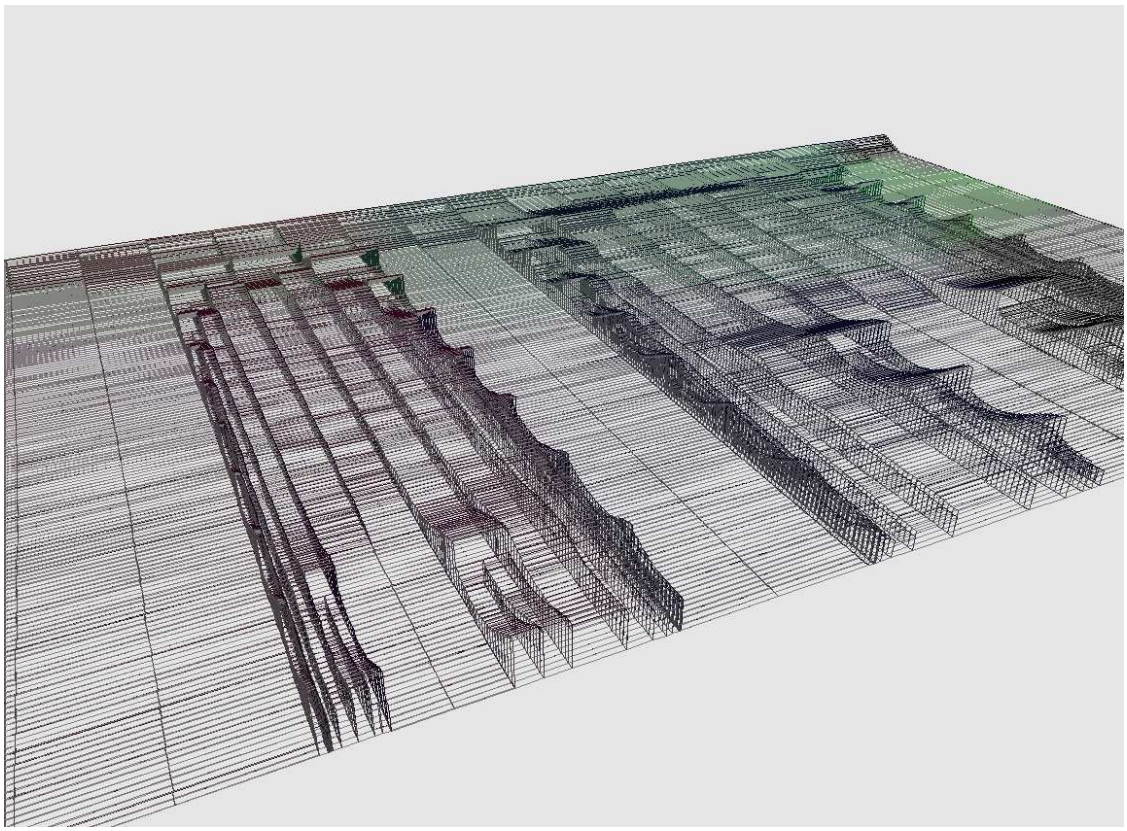


Figure 13: Example of a Turing diffusion-reaction process coupled with a morphogenesis. Each cell is rendered by a block whose height is proportional to the b morphogene (cf. text). When a cell is splitted in two, the width of the two daughter cells is divided by two, such that cells with a common ancestor are in the same parallel line (the axis directed toward the reader, which represents the passing of time). This plot corresponds to 180 time step evolution of an initial sequence of 18 cells.

5 Topological Collections and their Transformations

At this point of our presentation, the interested reader may object that the collection kinds in MGS are not related and that their presence in the same language is more a matter of juxtaposition rather than an integration. In the current prototypes (april 2001), it is true that the implementation of the collection kinds and of the pattern-matching algorithms are ad-hoc. And there is no way to build new collection kinds at user level (beside subtyping cf. section 3.2.3).

However, we show in this section that a formal generic framework can be developped. This formal framework relies on mathematical notions developped in combinatorial algebraic topology. The algebraic and combinatorial definition of the involved concepts makes them particularly suited for a computer implementation and justify our claims in the unifying and generic nature of the MGS approach, *far beyond monoidal collection*. The development of a new version of the MGS prototypes based on this formalization has starded, see 7.

The reader not interested in the formal development may skip this section.

5.1 Organization of this section

The definitions and results given below are standard in combinatorial algebraic topology, and have been gathered from the references [Ale82, Mun84, HY88, Sha90, Hen94, Axe98, Ber00]. Annex C reviews some of the algebraic structures used below.

The algebraic apparatus used here may appear very heavy with respect to our needs. However, the definitions introduced here are only the first elementary notions introduced for starting homology and cohomology theory.

We have tried to give an explanatory introduction of these definitions, following a step by step presentation, with some insights and intuitions talking to a computer scientist. The presentation is then not very straightforward and we have avoided a much more brutal but concise presentation. This explains the length of this chapter and the mix between informal considerations and the algebraic devices.

The organisation of this chapter can be sketched as follow:

1. The basic objects used to construct the space underlying a topological collection, *abstract complexes*, are introduced in section 5.2 and their neigborhood relationships are defined in section 5.3.

2. The previous structure is simple and natural but has some drawbacks when it comes to speak about boundary, or part of a space. The structure of *chain group*, that overcomes these limitation, is then presented in section 5.4. This structure is used to associate a value to each basic object to take into account its neigborhood (section 5.5).

3. We need to relate the notion of abstract complex and chain group together. It implies that only chain groups with a specific form interest us. This is explained in section 5.5.

   A fundamental example is presented in the next section (section 5.6). The structure of these group are more closely investigated in section 5.7.

4. There is an algebraic notion of duality that can be used to extend the notion of chain group. This extension generalizes the notion of chain and also gives a support to the notion of *coboundary*: two elements can be neighboor not only because they share something, but also because they are shared by something. The geometric interpretation is then sketched (section 5.8).

5. Section 5.9 shows that the previous notions can be used not only to render the neighborhood relationships, but also to associate a value with the " places " of a space.

6. We then take all this definitions together and put some additional constraints to define the type of a *topological collection* (section 5.10).

7. Various kind of *transformations* of such objects are specified in the following section (section 5.11).

8. The previous definitions are illustrated in an ad-hoc maner on the specific example of *grids* (section 5.12).

9. We summarize this presentation in the last section.

5.2 Cellular Spaces and Combinatorial Structure of Complexes

Topology is often presented as the geometry of rubber sheet: the properties of a figure that remain true under twisting, pulling, stretching, ..., any deformation of this sort provided the rubber can withstand it without ripping or tearing. Notions like continuity, limit, open set, etc., are developped in *point set* topology and are pertaining analysis and calculus. On the other hand, *combinatorial* topology has developed a strong algebraic flavor. The combinatorial method is used to construct complicated figures from simple ones and to deduce properties of the complicated from the simple. Here we want to speak about a space made of places and the neighborhood of a place in this space. The set of places is discrete and we are not really interested by the " metric " aspect of this space. It does not matter if a place is " far " or " near " another place. What does matter is the connection between places and the decomposition of places into subplaces. So, the combinatorial approach suits particularly well our needs. This sort of space, with its combinatorial structure, will be the carrier of a topological collection.

It is convenient to describe this space as build from basic blocs. This basic blocs are called *k-cells*. **Beware** that we use the same word " cell " for the biological object and the topological notion. In this chapter, we only refer to the topological notion. The fact that a topological 3-cell can be used to represent a biological cell in a simulation may be confusing. A k-cell is an homeomorphic image of an open balls in $\mathbb{R}^k$. In other word, a k-cell c is the image of the set $\mathbb{D}^k = \{x \in \mathbb{R}^k, ||x|| < 1\}$ by a continuous bijection h, such that h^{-1} is also continuous. However, the precise nature of the cell c is not stressed in a purely combinatorial approach until no link is made with point set topology notion. Here, we need only to grad the cells by their dimension and to focus on the connection of cells.

A collection of cells that are fitted together in an appropriate way form larger structures called *complexes*. Examples of complexes are given in Fig. 14 and 15. If an edge e is a side

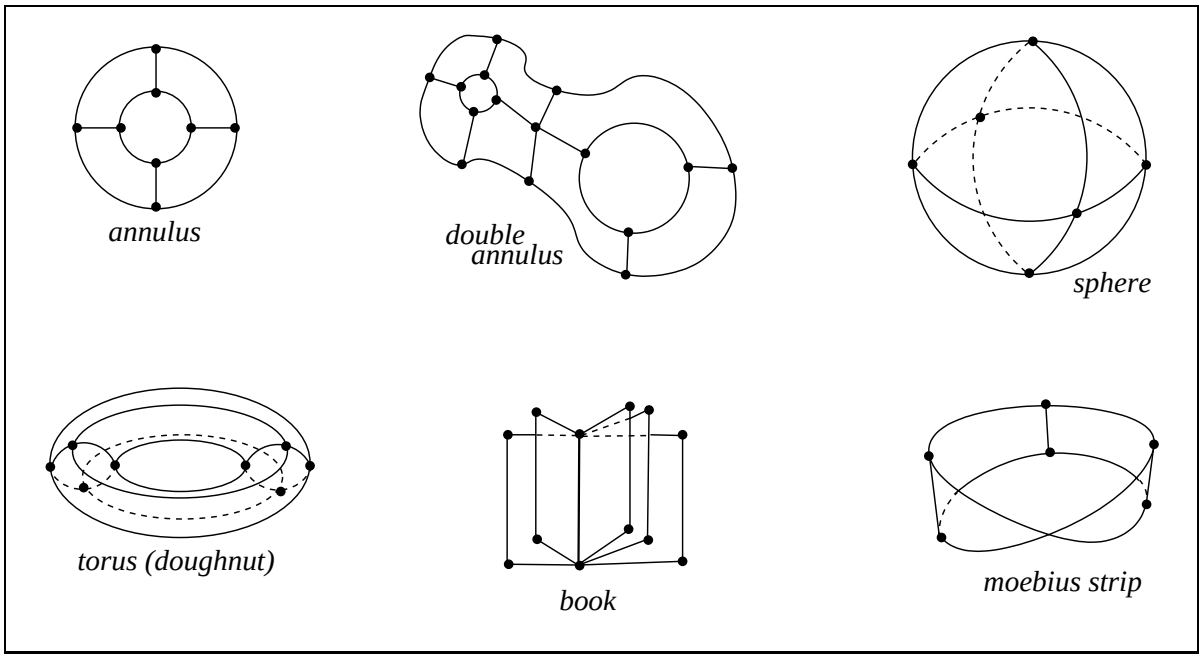


Figure 14: Examples of complex build from polygons. The examples of this figure imply cells of dimension less than 2. A *polygon* is a 2-cell where a finite number of points on the boundary are chosen as vertices. The section of the boundary in between vertices are the edges. A polygon is called a n-gon where n is the number of vertices. Thus the annulus there is composed of four 4-gons while the double annulus is composed of three 4-gons and four 5-gons.

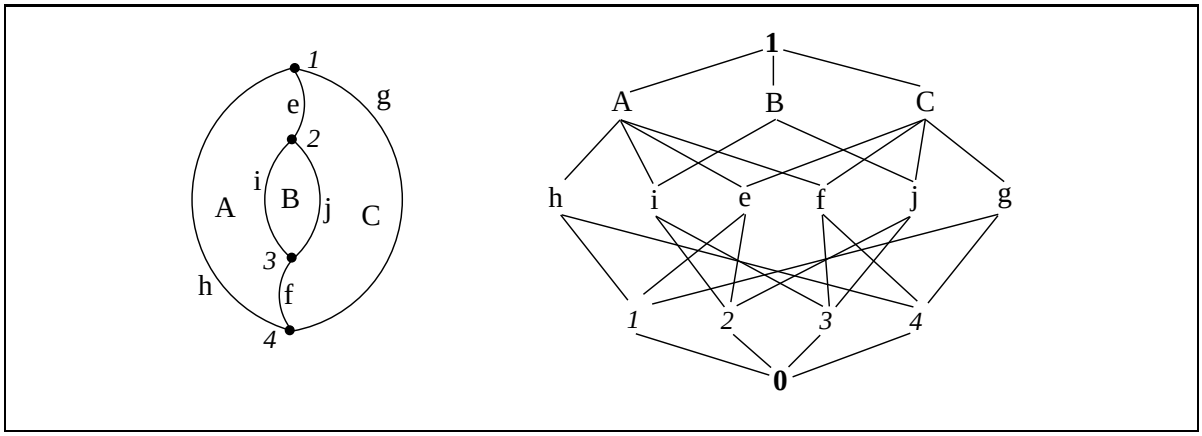


Figure 15: An abstract complex. The schema in the right hand side gives the Hasse diagram of the incidence relation of the complex in the left hand side. Faces are denoted by capital letters A, B and C. Edges are denoted by small letters and vertices by numbers. For instance, the face B is bounded by two edges i and j which are themselves bounded by vertices *2* and *3*. This example shows also that an abstract complex is generally not a *lattice*: there is for instance no least upper bound for edges e and f: both faces A and C are incomparable successors of e and f.

of a face f, we say that e and f are *incident* and we write $e < f$. The relation with the point set notion of a cell as an open ball, is the following. If a cell c is part of the closure of a cell c', we say that c is incident to c' and we write $c < c'$. However, the incidence relation is an order, and that's all we are interested in.

DEFINITION 1 (*Bounded Poset* $(P, <)$). A poset $(P, <)$ is a set P with an antisymmetric and transitive relation $<$ (the partial order). A poset is *bounded* if there are a unique minimal and maximal element $\mathbf{0}$ and $\mathbf{1}$. Let $x, y \in P$ such that $x < y$ and there is no z such that $x < z$ and $z < y$. Then we write $x \prec y$ and we say that x is a *predecessor* of y or that y is a *successor* of x.

DEFINITION 2 (*Abstract Complex*). An *abstract complex* $\mathcal{K}$ is a bounded poset with a function $\dim : \mathcal{K} \to \mathbb{Z}$ defined for the elements $e \in \mathcal{K} - \{\mathbf{0}, \mathbf{1}\}$ such that $e < e'$ implies $\dim e < \dim e'$ and $e \prec e'$ implies $\dim e' = 1 + \dim e$. The set $\mathcal{K}_p = \{e \mid e \in \mathcal{K}, \dim e = p\}$ are the p-cells of $\mathcal{K}$. A 0-cell is also called a *vertex*, a 1-cell is an *edge* and a 2-cell is a *face*. The *dimension* $\dim S$ of a subset $S \subset \mathcal{K}$ is the biggest of the dimensions of the elements of S if it exists.

The minimal and maximal element in the abstract complex definition will not be used at all here (they are useful to make some constructions more smoth). A graph is simply an abstract complex of dimension 1: the vertices are the nodes of the graph and 1-simplices are the edges, with the additional condition that there is exactly two predecessors for each edge.

Note that using an abstract complex, one cannot make a difference between a cylinder and a moebius strip because they give the same poset, see figure 16. These definitions are purely combinatorial and more specialized versions toward a geometric representation are usually used; we can cite *simplicial complex, singular complex, semi-simplicial set, polytope, cellular complex, CW-complex*, etc. They make more constraints in the fitting of cells into a complex.

5.3 Star, Link and Connections

Given a poset and its partial order $<$, we define the derived $\leq$ and $\preceq$ relationships. We defines now some operations on subsets of complexes. For a subset $S \subseteq P$, the smallest poset $\overline{S}$ is its closure.

DEFINITION 3 (*Subcomplex, Star and Shape*). Let $(\mathcal{K}, <)$, an abstract complex and $S \subseteq \mathcal{K}$ a subset of $\mathcal{K}$. Then the set $\overline{S} = \{y \mid y \in \mathcal{K}, y \leq x \in S\}$ with the relation $<$ is the subcomplex generated by S. It is called the *closure* of S. The *star* $\operatorname{St} x$ of a cell $x \in \mathcal{K}$ is $\operatorname{St} x = \{y \mid x \leq y \in \mathcal{K}\}$. We define the star of a subset $S \subseteq \mathcal{K}$ to be $\operatorname{St} S = \bigcup_{x \in S} \operatorname{St} x$ and the *closed star* is $\overline{\operatorname{St}} S = \overline{\operatorname{St} S}$. An element x is *above* a set $S \subset \mathcal{K}$ iff $x \in \overline{S}$ or if the elements of the set $\{y \mid y \prec x\}$ are all above S. The *shape* $\operatorname{Shape}(S)$ of a subset $S \subset \mathcal{K}$ is the set of the elements above S. These notions are illustrated in figure 17 and 18.

There is two ways for a cell x to be connected with a cell y: because they share a common boundary or because they are both boundaries of a " bigger " cell. Then, it appears that the " neighbors " of a cell x are the cells in $\overline{\operatorname{St}} x$.

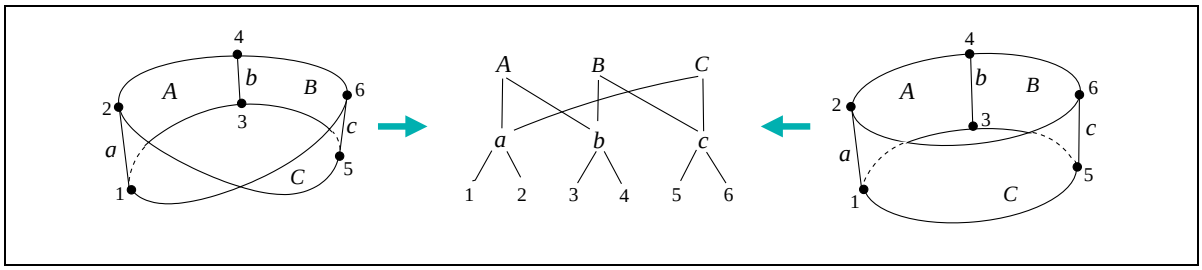


Figure 16: An abstract complex cannot handle orientation. For example, the moebius strip on the left gives the same poset as the cylinder on the right (they are both composed of 3 faces, 3 edges and 6 vertices).

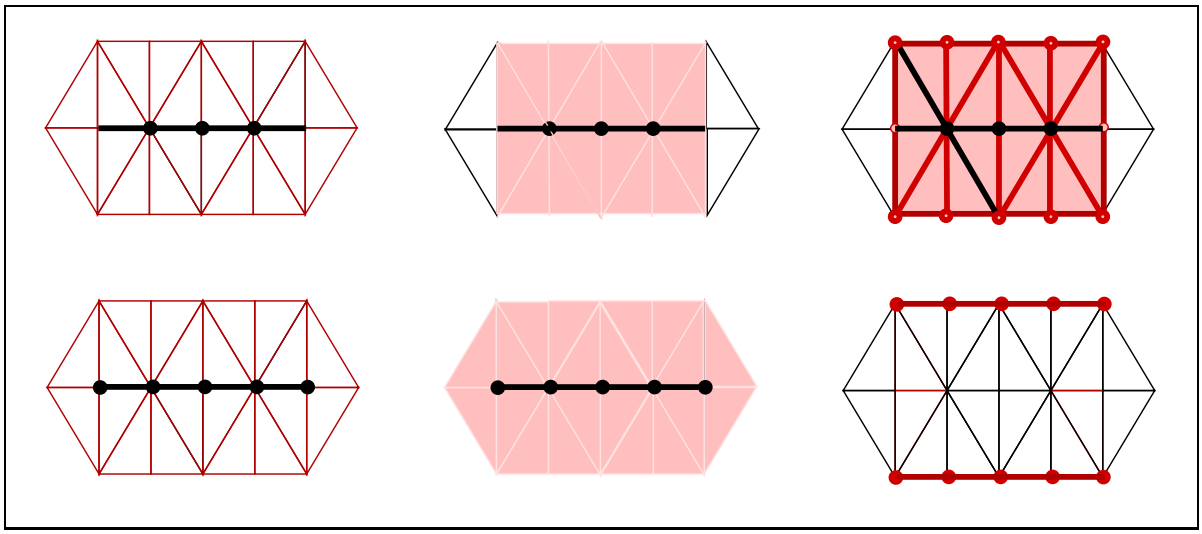


Figure 17: Examples of star and link. The subset S is composed of 4 edges and 3 vertices. On the top we have S pictured on the complex $\mathcal{K}$, then $\operatorname{St} S$ and $\overline{\operatorname{St}}\, S$. On the bottom line we have $\overline{S}$, $\operatorname{St} \overline{S}$ and the *link* of s: $\operatorname{Lk} S = \overline{\operatorname{St}}\, S \setminus \operatorname{St} \overline{S}$ which consists of two components of 4 edges and 5 vertices each (the operator "$\setminus$" denotes the asymmetric set difference, i.e. $A \setminus B = \{x \mid x \in A, x \notin B\}$).

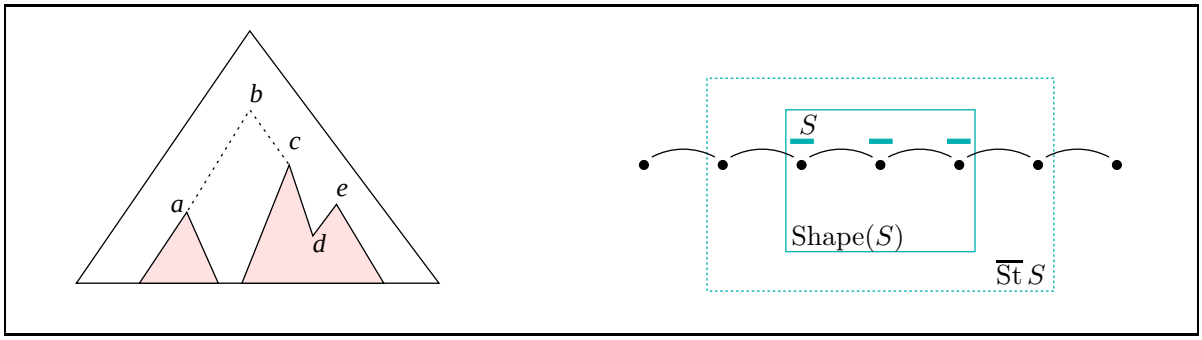


Figure 18: Connection and shape of a set. *Left figure.* We figure symbolically a poset $\mathcal{K}$ by a triangle. The coloured triangle below element a is the subcomplex $\overline{a}$ generated by a. It is also called the *cone* below a. An element x is in the cone below y iff $x \leq y$. The set $\{a, b, c, d, e\}$ is connected because elements are connected two by two. Fo example, a and b are connected because $a \leq b$, idem for c and b. The elements c and e are connected because $d \leq c$ and $d \leq e$. Let $A = \overline{a}$, $C = \overline{c}$ and $E = \overline{e}$ be the closure of $\{a\}, \{c\}$ and $\{e\}$ respectively. Then the set $A \cup C \cup E \cup \{b\}$ is also connected because a closure of a connected set is connected. *Right figure.* The set S consists of three internal vertices of a line graph. We have figured $\overline{\operatorname{St}}(S)$ and $\operatorname{Shape}(S)$.

DEFINITION 4 (*Connections*). Two cells x and y of an abstract complex $\mathcal{K}$ are *connected*, and we write $x \nearrow y$, if $\overline{x} \cap \overline{y} \neq \emptyset$ or if $\mathrm{St}\, x \cap \mathrm{St}\, y \neq \emptyset$. Given a set $S \subseteq \mathcal{K}$, we define $(\nearrow \backslash S)$ as the restriction of $\nearrow$ on S: $(\nearrow \backslash S) = \nearrow \cap (S \times S)$. Let $(\nearrow \backslash S)^*$ be the transitive closure of this relation. A subset S of $\mathcal{K}$ is *connected* if $(\nearrow \backslash S)^*$ has only one equivalence class.

Considering an infinite complex may be useful, for instance to represent an unbounded grid. However, each element (vertex or edge) in this grid is connected to only a finite set of other elements. Then, we say that the grid is locally finite.

DEFINITION 5 (*Locally finite complex*). A complex $\mathcal{K}$ is *closure-finite* if for all cell $x \in \mathcal{K}$, $\overline{x}$ is a finite set. It is *star-finite* if $\mathrm{St}\, x$ is a finite set for all x in $\mathcal{K}$. A complex which is both closure-finite and star-finite, is said to be *locally finite*.

5.4 Chain Complex

Figure 16 shows that the poset structure alone is not enough to represent the connection of cells. The problem can be splitted into two subproblems:

1. how a cell decomposes into subcells;
2. how a cell lies in the complex.

Obviously these problems are related. We will tackle the first question in this section. We shall examine the second in section 5.8.

A cell is not completely described by the simple set of its predecessors. One must represent also some organisation of these predecessors: for example an orientation, or a count if some subcells are identified (see the picture at the right on figure 22). This organisation of the set of the predecessors is represented by the notion of *chain*: a chain is a " structured set " of cells. This structure is specified through an abelian group structure and a boundary operator. The abelian group structure (the annex C gives a quick review of this notion) is used to describe the gluing of two cells using the group operation (written additively). The boundary operator gives the chain that describes the boundary of a cell, and by extension, the boundary of any chain.

Using an abelian group operation to represent the " gluing " c of two cells x and y means that we can write $c = x + y$ or $c = y + x$: the order of the gluing does not matter. The neutral element 0 corresponds to the empty set. And if we add a cell x to a part c, one must be able to " detach " the cell x from c latter. This justify the use of a group structure for the set of chains. Furthermore, one of the main objective of the theory is to compute the boundary of an arbitrary part of a space, from the boudary defined for an " isolated " cell. Thus, it is natural to require the boundary operator ∂ to be an homomorphism: $\partial(x+y) = \partial(x) + \partial(y)$. These considerations motivate the following definition.

DEFINITION 6 (*Chain Complex*). A *chain complex* is a sequence $C = (C_p, \partial_p)_{p \in \mathbb{Z}}$ of abelian groups C_p and connecting homomorphism $\partial_p : C_p \to C_{p-1}$, called *boundary maps*.

The group C_p is called the *p-chain group* and an element c of C_p is called a *p-chain*. C_p represents all the way to glue p-cells together. Sometimes we use a subscript p to indicate that

a chain c is a p-chain: c_p. In the opposite, for convenience in notation, we shall sometimes delete the dimensional subscript p on the boundary operator ∂_p, and rely on the context to make clear which of these operators is intended.

An abelian group C_p is *trivial* when its only p-chain is 0 (the element zero of the group). It this case we write $C_p = 0$. A *finite dimensional* chain complex C is such that the C_p are trivial except for at most a finite number of p. Most often, C_p is the trivial group for $p < 0$ and in this case we say that C is a *non-negative* chain complex. If C_p is a free abelian group for each p, then C is called a *free* chain complex. The chain complex is said *homological* or called a *graded differential group* with operator ∂, iff for all p and for all $c \in C_{p+1}$,

$$(\partial_p \circ \partial_{p+1})(c) = 0$$

This condition reflects the intuitive property that " the border of something has no border itself". Figure 19 gives an example of a non-homological chain complex. Intuitively, the loss of homology comes from identifying parts of the boundary of a cell.

5.5 Chain Group with Coefficient in an Arbitrary Abelian Group

In the two examples of the figure 19, the groups C_p are built from the p-cells in $\mathcal{K}_p$ by saying that C_p is the abelian group generated by the element of $\mathcal{K}_p$ subject to the equations $c+c=0$. In other word, every element in C_p is a formal sum of elements in K_p and any element c in C_p is its own inverse. Example of chains are e, $e+f$, $f+e+g$, etc. So, the relation between a chain and a cell is clear in this case: the chain reduced to only one element x, corresponds to a p-cell x.

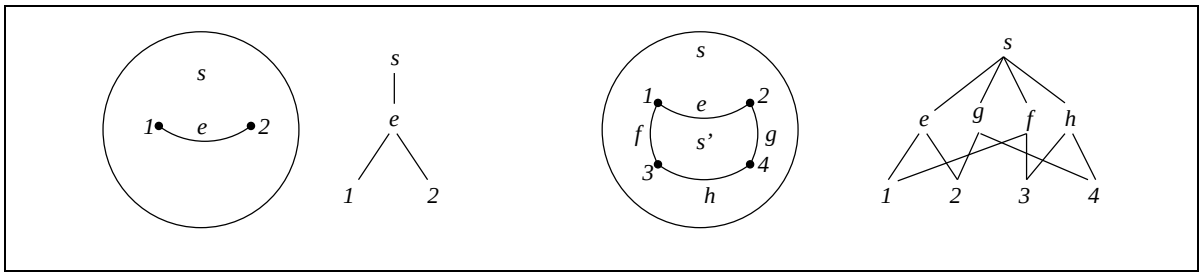


Figure 19: Examples of a non-homological and an homological chain complex. The two figures represent two possible constructions of a sphere. The associated complex $\mathcal{K}$ is defined on the right. The vertices are nammed by an italic number (*1*, *2*, ...), edges are identified by a letter in the begining of the alphabet (e, f, ...) and the faces are called s and s'. A p-chain is simply a " sum" of p-cells, like $e+f$ for instance. In addition, we suppose that the chain groups C_p are *idemgroups*, that is $c+c=0$ for any p-chain. Furthermore, we define $\partial(x) = \sum_{y \prec x} y$ for all p-cell $x \in \mathcal{K}$ and we extend the operator ∂ by linearity, that is $\partial(x+y) = \partial x + \partial y$. The complex pictured to the left is a non-homological complex. Face s is folded and its boundary consists only of one edge e (imagine the border of a disk pathologically stitched on itself to obtain a sphere). The vertices *1* and *2* are the boundary of this edge. This complex is non-homological because $\partial\partial(s) = \partial(e) = (1+2) \neq 0$. This has to be compared with the complex pictured at the right hand-side which is homological. For instance $\partial\partial(s) = \partial(f+g+e+h) = (1+3)+(2+4)+(1+2)+(4+3) = 0$ (recall that $x+x=0$ in an idemgroup). And the result is the same for s' or any other k-cell.

However, the description of the structure of a space by a chain complex is rather abstract and we have in general no explicit notion of the cells in the description of a chain complex. How can we relate an abstract complex $\mathcal{K}$ and a chain complex C?

The idea is that some elements in C_p, called *elementary chains*, must represent a unique p-cell x together with a coefficient g which represents some information about the " gluing" (orientation, count, etc.) of x. Let G be the set of all possible g. Then an elementary chain can be viewed as a pair (g, x). A typical element in C_p is a sum of such elementary chains and C_p must have the structure of a group. Consider the sum of two elementary chains on the same cell x: $c = (g, x) + (g', x)$. The result must be an elementary chain on the same p-cell x, so $c = (g'', x)$. The coefficient g'' must represent the contribution of x coming from a contribution g and a contribution g'. We write $g'' = g + g'$ and is this easy to see that if C_p has an abelian group structure, then G must have an abelian group structure too. And conversely. If we write an elementary chain (g, x) as the formal product gx of an element of the coefficient group G by an elementary p-cell, then a typical chain c_p can be written

$$c_p = \sum_{x \in \mathcal{K}_p} g_x x$$

where g_x is the coefficient that describe the contribution of the p-cell x. But one man think c_p has a function that describes the contribution of x, that is, $c_p : \mathcal{K}_p \to G$ with $c_p(x) = g_x$. Historically, people have first considered chains with a finite number of contributing p-cells, so in the previous sum we consider that only a finite number of g_x are nonzero. Hence the definition.

DEFINITION 7 (*Chain Group with Coefficient in an abelian group G*).

Let $\mathcal{K}$ an abstract complex (finite or not), and let G denotes an arbitrary abelian group written additively. The neutral element of G is written 0. The set $C_p(\mathcal{K}, G)$ of *p-chain on the complex $\mathcal{K}$ with coefficient in G* is the set of total functions c_p from the set $\mathcal{K}_p$ to G that are zero almost everywhere, that is, $c_p(x) = 0$ for all but a finite number of p-cells of $\mathcal{K}$.

The operation used to turn the set of k-chains into a group is the addition of functions if one thinks of them as function, or the componentwise addition if one thinks of them as sums. *Integral chains* are just chains with integer coefficients $C_p(\mathcal{K}, \mathbb{Z})$. The integral chain group, of special importance, is abbreviated $\mathcal{C}_p(\mathcal{K})$. The justification of the sum notation, with the product of a p-cell and a coefficient in G wille be delayed until section 5.7.

Carrier of a Chain. Let $c_p = \alpha_1 x_1 + \cdots + \alpha_n x_n$ be a chain of $C(\mathcal{K}, G)$. Then $\alpha_i \in G$ and we suppose also that $\alpha_i \neq 0$ for all i. Then the *carrier* of c_p is the set of p-cells with a nonzero coefficient in the cochain: $|c_p| = \{x_1, \ldots, x_n, \ldots\}$.

Compatible Boundary Homomorphisms. We have defined the boundary maps of a chain complex $C = (C_p, \partial_p)$ as a sequence of homomorphisms satisfying the signature:

$$C_0 \xleftarrow{\partial_1} C_1 \xleftarrow{\partial_2} C_2 \xleftarrow{\partial_3} \cdots$$

Saying that ∂_p is an homomorphism means that we can define ∂_p on elementary chains and extend the boundary operator on any chains by *linearity*: $\partial(c+c') = \partial c + \partial c'$ In other word, " the boundary of a sum of elements is the sum of the boundaries of the elements ".

The boundary operators embeed more information than the poset structure alone. For example, suppose we work with integral chain groups and we have to describe the moebius band in figure 16. Then the predecessor of C are a and c and we have $\partial C = c - a$. The edge a is counted negatively to account for an oposite orientation. For the cylinder, we have $\partial C = c + a$: the boundary operators makes a difference between two objects that are not distinguished with the poset structure alone.

However, when we use the chain groups $C_p(\mathcal{K}, G)$ in relation with an abstract complex $\mathcal{K}$, we need to relate the connection structure described by the abstract complex $\mathcal{K}$ and the connection described by ∂.

DEFINITION 8 (*Compatible Boundaries*). Let $(C_p(\mathcal{K}, G), \partial_p)$ be a chain complex associated with an abstract complex $(\mathcal{K}, <)$. Then, the boundary maps ∂_p are said *compatible* with $\mathcal{K}$ iff for all $x \in \mathcal{K}_p$, and for all $g \in G$, $g \neq 0$, $|\partial g x| = \{y \mid y \prec_{\mathcal{K}} x\}$.

The elements with nonzero coefficient in $\partial g x$ are exactly the predecessors of x. This condition ensure the coherence between the poset structure of the abstract complex $\mathcal{K}$ and the boundary operations. In the previous example, the compatibility condition is respected because $|\partial C| = \{a, c\}$ both for the moebius band and the cylinder.

Basic Assumption. We are only interested in the case where the abelian groups C_p are related to an abstract complex $\mathcal{K}$. Every chain complex $(C_p, \partial_p)_p$ we consider henceforth is such that $C_p = C_p(\mathcal{K}, G)$ and the boundary maps ∂_p are compatible with $(\mathcal{K}, <)$. We write $C(\mathcal{K}, G, \partial)$ for such chain complex.

5.6 Example of the $C(\mathcal{K}, \mathbb{Z}/2, \partial)$ Chain Complex

Returning back to the examples in figure 19, now we may specify rigorously the group C_p as the functions from $\mathcal{K}_p$ to $\mathbb{Z}/2$ the group of integers modulo 2 (cf. annex C). A chain $c = e + f$ corresponds to the function c defined by $c(e) = c(f) = 1$ and $c(x) = 0$ for $x \neq e$ and $x \neq f$. This chain can also be written $c = 1.e + 1.f + 0.g + 0.h + \ldots$. It is customary not to write the p-cells with a zero coefficient (in accordance with the additive notation). Thus we have $c = 1.e + 1.f$ (or more ambiguously $c = e + f$).

Representation of the Subsets of $\mathcal{K}$ by $C_p(\mathcal{K}, \mathbb{Z}/2)$. Using $\mathbb{Z}/2$ as the chain coefficients enables the representation of the presence, $c_p(x) = 1$, or the absence, $c_p(x) = 0$, of a p-cell x in a chain c_p. A chain of $C(\mathcal{K}, \mathbb{Z}/2)$ is then simply the characteristic function of a subset of $\mathcal{K}$. But the group structure gives some additional capabilities.

The elements of this group are $\{0, 1\}$ and $0 + 0 = 1 + 1 = 0$ and $0 + 1 = 1$. The groups where $x + x = 0$ for all x are qualified as *idemgroups*. So $\mathbb{Z}/2$ is an idemgroup as well as $C(\mathcal{K}, \mathbb{Z}/2)$. Suppose c_1 and c_2 are given k-chains with coefficient in $\mathbb{Z}/2$. Then, $c_1 + c_2$ is defined to be the k-chain made up of the k-cells of c_1 or c_2 *but not in both*. In other words,

the $+$ operation between chains corresponds to the "symmetric set difference". Similarly, $c_1 + c_2 + c_3$ consists of thoses k-cells contained in *just one* or *all three* of the chains c_1, c_2 and c_3. More generally, the sums of n k-chains turn out to consist of those cells contained in an *odd* number of the chain c_1, .., c_n. The zero k-chain (the unique element in C_k such that $c + 0 = 0 + c = c$ for all k-chain c) is the empty chain denoted 0 and containing no k-cell.

Computation of a Boundary in $C(\mathcal{K}, \mathbb{Z}/2)$. We have shown that the use of coefficients in $Z/2$ enables the representation of an arbitrary subset. To turn these chain groups into a chain complex, we have to define the boundary operators ∂_p. We just define the boundary of a single p-cell x as the chain that represents the predecessors of x:

$$\partial_p(x) = \sum_{y \in \mathcal{K}_{p-1},\, y \prec x} y$$

and we extends this definition by linearity on a p-chain $c = \sum_x c(x)\, x$:

$$\partial(c) = \sum c(x) \partial(x) = \sum_x \sum_{y \prec x} c(x)\, y$$

By definition, these boundary operators are homomorphisms and compatible with the poset structure of the abstract complex $\mathcal{K}$ (there is no other possible definition for ∂). Let us see the effect of this boundary operator on such chains.

Suppose that the chain $c \in C_p(\mathcal{K}, \mathbb{Z}/2)$ is composed of two k-cells s' and s'; this is denoted by $c = s + s'$. Suppose than s and s' share only one cell $d \in \mathcal{K}_{p-1}$, see Fig. 20. Then d is not in the border of s because s and s' are glued along d: d is an interior cell. But d is in the boundary of s and in the boundary of s'. Let $\partial_p s = d + \sum x'_j$ and $\partial_p s' = d + \sum x''_k$. Then we must have: $d + \sum x'_j + d + \sum x''_k = \sum x'_j + \sum x''_k$ which is automatically achieved because $d + d = 0$.

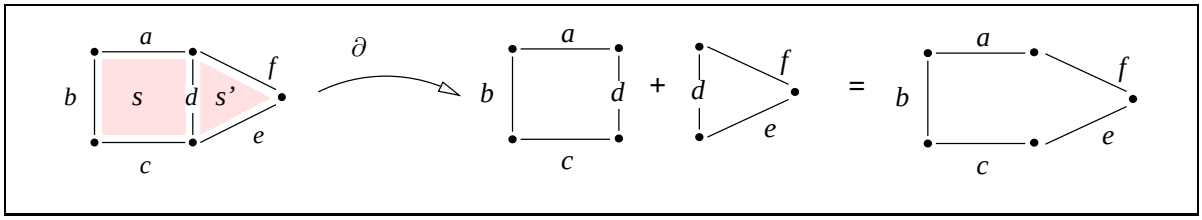


Figure 20: Example of the application of the boundary operator on a $C(\mathcal{K}, \mathbb{Z}/2)$ chain. $\partial(s + s') = \partial s + \partial s' = (a + b + c + d) + (d + e + f) = a + b + c + e + f$ because $d + d = 0$.

All interior cells, i.e. cells that have two successors, will cancel out and only the geometric boundary cells of s will remain. This explain the sense in which the geometric boundary of a set of points is a special case of the more general topological boundary operator defined here.

An Algebra for Counting the Cells in a Boundary. Note that working with $\mathbb{Z}/2$ coefficients avoids the problem of the orientation of each cell and encounters the problem pointed in Fig. 16: orientation is not taken into account, as for abstract complex. This

problem can be handled using integer coefficients $\mathbb{Z}$ that enables the precise counting of each cell, together with its orientation.

The group $\mathbb{Z}/2$ provides an algebra for handling certain simple counting operations where only the evenness or the oddness of the results is important. The integral chain group $C(\mathcal{K}) = C(\mathcal{K}, \mathbb{Z})$ provides an algebra for handling these counting operations without the peculiar restriction to evenness and oddness, see fure 21. The chain $2x + 1y - 3z$ counts the cell x twice, the cell y once and the cell z minus three times. And another abelian coefficient group G provides another kind of counting algebra.

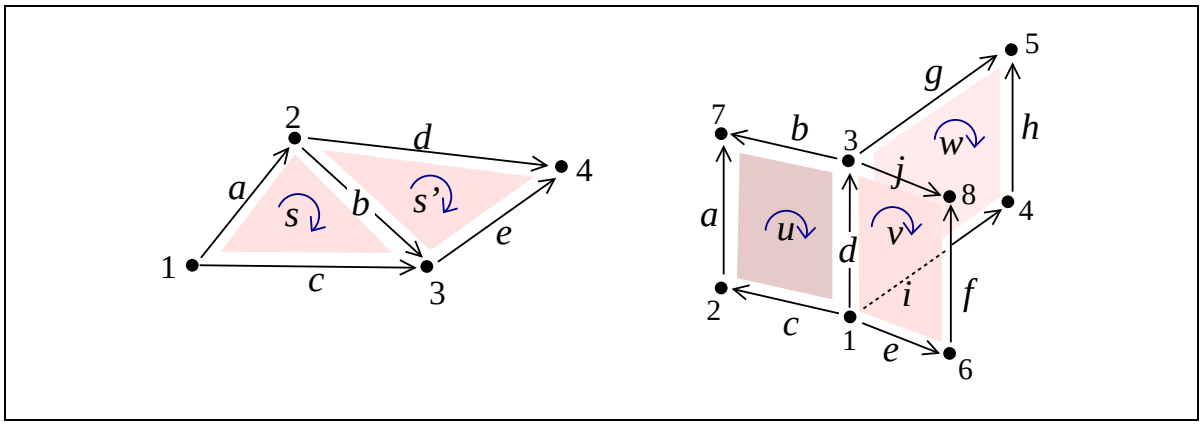


Figure 21: Examples of two oriented complexes. The two figures represents two 2-chains with integral coefficient. The cells used in the left complex are *simplicial cells*, that is, a p-cell is the convex hull of $p+1$ points in $\mathbb{R}^n$. The sign of the coefficient is used to take into account the orientation of edges and faces. For example, $\partial d = 4 - 2$ and $\partial s = a + b - c$. Orientation can be used to cancel out a coefficient: $\partial(s+s') = (a+b-c)+(d-e-b) = a+d-e-c$. The cells in the right complex are polygonal cells. They are oriented by giving an order between vertices. Faces are oriented positively from the low vertex to the high vertex. This example shows that the absolute value of the coefficient is used to count the number of time a cell is used: $\partial(v+w) = (j-f-e+d)+(g-h-i+d) = j-f-e+g-h-i+2d$.

5.7 The Structure of the Chain Group with Coefficient

We want to characterize further the chain groups with coefficient. We first define the notion of free abelian group and review some useful results. Then we show that the integral chain group is free. We show that any chain group with coefficient in G has the structure of a sum of copies of G Finally, we give a representation of $C_p(\mathcal{K}, G)$ in terms of $C_p(\mathcal{K})$.

Constructing a Free Abelian Group From a Basis. An abelian group G is *free* if it has a *basis*, that is, if there is a family $\{g_\alpha\}_{\alpha \in J}$ such that each $g \in G$ can be written *uniquely* as a sum $g = \sum n_\alpha g_\alpha$ where $n_\alpha \in \mathbb{Z}$. If G is free and has a finite basis consisting in n elements, then it is easy to see that every basis for G consists of precisely n elements (this number is called the *rank*). If the basis is not finite, then any basis has the same cardinality. A subgroup of a free group is free.

We give now a specific way of constructing a free abelian group $\mathrm{Abel}(S)$ from a basis S. This construction will be used elsewhere. If the set S is finite, we say that $\mathrm{Abel}(S)$ is *finitely*

generated. Given a set S, not necessarily finite, we define the free abelian group $\mathrm{Abel}(S)$ generated by S to be the set of all total functions $\varphi : S \to \mathbb{Z}$. We add two such functions by adding their values. Given $x \in S$, there is a characteristic function φ_x for x, defined by setting $\varphi_x(x) = 1$ and $\varphi_x(y) = 0$ for $y \neq x$. The set of functions $\{\varphi_x \mid x \in S\}$ form a basis for $\mathrm{Abel}(S)$, that is, each function $\varphi \in \mathrm{Abel}(S)$ can be written uniquely as a sum $\varphi = \sum_{x \in S} a_x \varphi_x$ where $a_x = \varphi(x)$.

We often abuse notation and identify the element $x \in S$ with its characteristic function φ_x. With this notation, the general element of $\mathrm{Abel}(S)$ can be written uniquely as a *formal linear combination* $\varphi = \sum a_x x$ where $a_x \in \mathbb{Z}$ and x are elements of the set S. The set of elements that can be written as a finite sum is a subgroup $\mathrm{Abel}_{\mathrm{finite}}(S)$ of $\mathrm{Abel}(S)$. If $\mathrm{Abel}(S)$ is finitely generated, then $\mathrm{Abel}_{\mathrm{finite}}(S) = \mathrm{Abel}(S)$ but this is not in general the case.

We can relate the group $\mathrm{Abel}(S)$ and the constructions of direct product and external direct sum of groups introduced in annex C. It is easy to show that $\mathrm{Abel}(S) = \prod_{x \in S} G_x$ where $G_x = \mathrm{Abel}(\{x\})$. And $\mathrm{Abel}_{\mathrm{finite}}(S)$ is the so-called external direct sum of the groups G_x: $\mathrm{Abel}_{\mathrm{finite}}(S) = \bigoplus_{x \in S} G_x$. Conversely, if G is a direct sum or a direct product of infinite cyclic subgroups, then G is a free abelian group. The qualifier " direct" in the product or in the sum, is for the uniqueness of the sum denoting an element. If $G = \prod G_x$, then this product is direct if and only if the equation $0 = \sum g_x$ implies that $g_x = 0$ for each x. This in turn occurs if and only if for each fixed x, one has $G_x \cap \left(\sum_{y \neq x} G_y\right) = \{0\}$.

The Free Structure of $C(\mathcal{K})$. The resemblance of $C_p(\mathcal{K}) = C_p(\mathcal{K}, \mathbb{Z})$ to free abelian groups is strong. And indeed $C_p(\mathcal{K})$ is a *direct sum* of infinite cyclic groups

$$C_p(\mathcal{K}) = \bigoplus_{x \in \mathcal{K}_p} \mathrm{Abel}(\{\widehat{x}\}) \simeq \bigoplus_{x \in \mathcal{K}_p} \mathrm{Abel}(\{x\}) \simeq \bigoplus_{\mathcal{K}_p} \mathbb{Z}$$

where the function $\widehat{x} : \mathcal{K}_p \to \mathbb{Z}$ is specified by $\widehat{x}(x) = 1$ and $\widehat{x}(y) = 0$ for $y \neq x$. Usually, we identify $\mathrm{Abel}(\{\widehat{x}\}$ and $\mathrm{Abel}(\{x\})$ as well as $\widehat{x}$ and x. In addition, $\mathrm{Abel}(\{x\}) \simeq \mathbb{Z}$ (just use the isomorphism $nx \mapsto n$). Then, if $\mathcal{K}_p$ is finite with cardinality n, we have $C_p(\mathcal{K}) \simeq \mathbb{Z}^n$.

DEFINITION 9 (The *free Chain Group*). The direct sum of a sequence of free abelian groups is again a free abelian group. Using this fact we may form *the* integral chain group :

$$\mathit{Chains}(\mathcal{K}) = C_0(K) \oplus C_1(K) \oplus ... \oplus C_n(K) \oplus ...$$

Note that if there is no cells in $\mathcal{K}_q$, then we set $C_q(K) = 0$ (the trivial group).

By definition (see annex C), each element of $C(\mathcal{K})$ is a sequence $(c_0, c_1, ..., c_n, ...)$ where c_p is an integral p-chain of $\mathcal{K}_p$ and where there is only a finite number of c_k that are nonzero. Such a weak direct sum is often called a *graded group*.

The Sum Structure of the $C_p(\mathcal{K}, G)$. It is easy to show that $C_p(\mathcal{K}, G)$ has also the structure of a direct sum. Consider the set of the total functions $\widehat{x} : \mathcal{K}_p \to G$ such that $\widehat{x}(y) = 0_G$ for all $y \neq x$. This set is a group G_x for the addition of functions and $C_p(\mathcal{K}, G) = \bigoplus G_x$. However, $C_p(\mathcal{K}, G)$ is not free because the G_x are not necessarily free. Consider for

example the group $G = \mathbb{Z}/2$; then $1.\widehat{x} + 1.\widehat{x} = 0_{(\mathcal{K}_p \to \mathbb{Z}/2)}$ for all cell x, which show that G_x is not free.

Now, each G_x is obviously isomorphic to G (by the mapping $c \mapsto c(x)$). Then, we have in general $C_p(\mathcal{K}, G) \simeq \bigoplus_{\mathcal{K}_p} G$ and this justify the sum notation for an element. An elementary chain c which associates $g \in G$ to the p-cell x is written gx. If $\mathcal{K}_p$ is finite with cardinality n, then $C_p(\mathcal{K}, G) \simeq G^n$. *The* chain group with coefficient in G is defined by: $Chains(\mathcal{K}, G) = C_0(\mathcal{K}, G) \oplus C_1(\mathcal{K}, G) \oplus \ldots$

5.8 Duality: Cochain, Coboundary and Cochain Complex

We want now give a very slight generalization of the notion of chain by defining *cochains*. This generalization has several motivations: it allows the handling of " infinite " chains; it makes able to relate the chain group with arbitrary coefficient with the integral chain group; and finally it introduces naturally a dual of the ∂ operator.

Chains with Coefficient in G as Homomorphisms from $C_p(\mathcal{K})$ to G. We defined a chain to be a function from the p-cells to an abelian group G, but using linear extension we can and will consider a chain to be a function on integral chains.

For let $c^p = \sum \alpha_i x_i$ where each α_i is in G and each x_i is in $\mathcal{K}_p$. Let $d_p = \sum n_j x_j$ be an integral chain (i.e. n_j belongs to $\mathbb{Z}$). We may then define the *value of c_p on d_p* by

$$c_p(d_p) = c_p\Big(\sum_j n_j x_j\Big) = \sum_j n_j \,.\, c^p(x_j) = \sum_j n_j \alpha_j$$

Clearly $\sum_j n_j \alpha_j$ is an element of G since $n_j \alpha_j$ is the n_j-fold sum $\alpha_j + \cdots + \alpha_j$.

For a fixed chain c_p, this operation yields a homomorphism of $C_p(\mathcal{K})$ into G. However, if $\mathcal{K}_p$ is not finite, then the set $C_p(\mathcal{K}, G)$ of chains with coefficient in G does not contain all the homomorphisms between $C(\mathcal{K}_p)$ and G. For example, suppose $g \in G, g \neq 0$, then h defined by $h(x) = g$ for all x is an homomorphism which cannot be represented by a finite sum: $h = \sum_{x \in \mathcal{K}_p} g\, x$ contains as many terms as $\mathcal{K}_p$ has elements. This motivate to consider infinite sums to retrieve all the homomorphisms.

DEFINITION 10 (*Cochains*). *A p-cochain on the complex $\mathcal{K}$ with coefficient in G* is a total function c^p from the set $\mathcal{K}_p$ to the abelian group G. The set of p-cochains on the complex $\mathcal{K}$ with coefficient in G is a free abelian group (for the pointwise addition of functions) written $C^p(\mathcal{K}, G)$, and we have: $C^p(\mathcal{K}, G) = \mathrm{Hom}(C_p(\mathcal{K}), G)$.

The notation $\mathrm{Hom}(A, B)$ denotes the set of homomorphisms between a group A and a group B. This set is a group for the pointwise addition of functions. The difference between a cochain c^p and a chain d_p is that c^p is not necessarily zero almost everywhere. Then:

- Every chain is a cochain but not conversely.
- The set of chains $C_p(\mathcal{K})$ is a subgroup of $C^p(\mathcal{K})$.

- However, the two groups $C^p(\mathcal{K}, G)$ and $C_p(\mathcal{K}, G)$ are identical in the case of a finite complex $\mathcal{K}$ or if $\mathcal{K}_p$ is finite.

To distinguish between the chains and cochains (if needed), we are following the current practice in using subscripts to indicate the dimension of chains and superscripts to give the dimensions of cochains.

The cochain c^p can also be written as a sum $c^p = \sum_{x \in \mathcal{K}_p} \alpha_x x$ but this sum is not necessarily finite. Saying that $c^p = \sum_{x \in \mathcal{K}_p} \alpha_x x$ is equivalent of saying that $c^p(x) = \alpha_x$ for all x in $\mathcal{K}_p$.

The Characterization $C^p(\mathcal{K}, G) = \mathrm{Hom}(C_p(\mathcal{K}), G)$ and the Sum Notation. From the presentation of cochains as (possibly infinite) sum, we deduces that $C^p(\mathcal{K}, G)$ is the group $\mathrm{Hom}(C_p(\mathcal{K}), G)$. We can chose this latter result as the definition of cochains and recover the representation of a cochain as a (possibly infinite) sum. The group $C_p(\mathcal{K})$ is the free abelian group generated by the element of $\mathcal{K}_p$ and therefor, $C_p(\mathcal{K}) = \bigoplus_{x \in \mathcal{K}_p} G_x$ where G_x is the free group generated by x. Then[11]: $\mathrm{Hom}(C_p(\mathcal{K}), G) = \mathrm{Hom}(\bigoplus_{x \in \mathcal{K}_p} G_x, G) \simeq \prod_{x \in \mathcal{K}_p} \mathrm{Hom}(G_x, G)$. An element of $\mathrm{Hom}(G_x, G)$ is an homomorphism h that associates to an element $n.x$ of G_x an element $n.g_x$ where $g_x = h(1.x)$. We denote this element by $g_x x$. An element of $\prod_{x \in \mathcal{K}_p} \mathrm{Hom}(G_x, G)$ can be written as a sum of elements belonging to the factors, because these groups are distincts. Thus, an element c^p of $C^p(\mathcal{K}, G)$ can be written as a sum $\sum_{x \in \mathcal{K}_p} g_x x$ where $g_x = c^p(x)$ and this gives the previous sum notation for a cochain.

The Kronecker Index. In place of the functional notation, it is often convenient to use a product notation. That is, we use $c^p.d_p$ to denote the value of c^p on d_p rather than the more familiar notation $c^p(d_p)$. The result of this " product" is called the *Kronecker index* of c^p and d_p.

Consider two p-cells x and y. They can be viewed as the two elementary integral chains $x_p = 1.x$ and $y_p = 1.y$ and also as the two elementary integral cochains x^p and y^p (a chain is a cochain). Then we have: $x^p.y_p = y^p.x_p = 0$ if $x \neq y$ and $x^p.x_p = 1$ elsewhere.

We have mentionned the dimension in subscript or in superscript to make clear what object is at hand, but we shall delete them when there is no confusion.

Dual Homomorphisms, Coboundary Operators and Cochain Complex. The abelian group $C_p(\mathcal{K})$ is a $\mathbb{Z}$-module and the group $\mathrm{Hom}(C_p(\mathcal{K}), \mathbb{Z})$, wich is also the set of integral cochains $C^p(\mathcal{K})$, is a $\mathbb{Z}$-module called the *dual* of $C_p(\mathcal{K})$ (see annex C). This notion of duality is the direct generalization for modules of the dual of a vector space. Intuitively, if one consider a chain as " vectors", then the cochains are the " linear forms". We can go further in the analogy with the notion of dual homomorphism.

DEFINITION 11 (*Dual Homomorphism*). A homomorphism $\sigma : A \to B$ gives rise to a *dual homomorphism*

$$\mathrm{Hom}(A, G) \xleftarrow{\widetilde{\sigma}} \mathrm{Hom}(B, G)$$

[11] We use the following result: the homomorphisms from the weak direct sum of the P_i to a group G is isomorphic to the direct product of the homomorphisms from P_i to G.

going in the reverse direction and defined by: $\widetilde{\sigma}(\varphi) = \varphi \circ \sigma$.

DEFINITION 12 (*Coboundary Operator* δ). We define the *coboundary operator* δ as the dual of ∂: $\delta = \widetilde{\partial}$. The operator ∂_{p+1} on integral chains is an homomorphism from $C_{p+1}(\mathcal{K})$ to $C_p(\mathcal{K})$, thus

$$\delta^p = \widetilde{\partial_{p+1}} : C^p(\mathcal{K}, G) \longrightarrow C^{p+1}(\mathcal{K}, G)$$

so that δ^p raises dimension by one. The effect of operator δ^p is defined by:

$$(\delta^p c^p) \,.\, d_{p+1} = c^p \,.\, (\partial_{p+1} d_{p+1})$$

DEFINITION 13 (*Cochain Complex*). We define the (homological) abstract cochain complex similarly to the chain complex. We write $C(\mathcal{K}, G, \delta)$ for the sequence $(C^p(\mathcal{K}, G), \delta^p)_{p \geq 0}$ where the coboundary operators are homomorphisms with signature as follows:

$$C^1(\mathcal{K}, G) \xrightarrow{\delta^1} C^2(\mathcal{K}, G) \xrightarrow{\delta^2} C^3(\mathcal{K}, G) \xrightarrow{\delta^3} \cdots$$

The abelian group $Cochains(\mathcal{K}, G) = C^0(\mathcal{K}, G) \oplus C^1(\mathcal{K}, G) \oplus \ldots$ is called *the* cochain group. An abstract cochain complex $C(\mathcal{K}, G, \delta)$ is said *homological* or called a *graded differential group* with operator δ, iff for all p, $\delta^{p+1} \circ \delta^p = 0$.

Defining ∂, then there is a unique δ dual operator, and vice-versa. If $\partial_p \circ \partial_{p+1} = 0$, then by duality we have also $\delta_{p+1} \circ \delta_p = 0$. Thus, if $C(\mathcal{K}, G, \partial)$ is an homological abstract chain complex, then $C(\mathcal{K}, G, \delta)$, where δ is the dual of ∂, is an homological abstract cochain complex.

A fundamental difference between ∂ and δ is that ∂x depends only on (the closure of) x while δx depends on how x lies in the complex $\mathcal{K}$. Furthermore, it is possible that x is a cell belonging to the boundary of infinitely many cells, even if the complex $\mathcal{K}$ is closure-finite. Thus δx is not necessarily a finite sum. However, in the following we want to consider only locally-finite complex $\mathcal{K}$. Then if we take the boundary or the coboundary of a chain, we obtain a chain again.

Geometric Interpretation of δ. The definition of δ is highly algebraic in nature. But it is possible to figure the geometric meaning of δ. The *dual of a poset* $(E, <)$ is the poset $(E, >)$ with the reverse order between elements. Then we can define the analog of $\partial : \mathcal{K}_p \to C(\mathcal{K})$ for the dual poset. Let δ' be this operator. Following the definition of ∂, we must have $|\delta'(x)| = \{y \mid x \prec y\}$.

When considering both ∂ and δ' together, we need to ensure some consistency between the coefficients associated to each boundary or coboundary element. Let x be an element of $\mathcal{K}_{p-1}$ and y an element of $\mathcal{K}_p$. Then the coefficient of x in the chain ∂y is $(\partial y)(x) = (\partial y).x$ using the Kronecker notation. If x is in the boundary of y, then $x \in |\partial y|$ and $(\partial y).x \neq 0$. However, if $x \in |\partial y|$ then $x \prec y$ and then $y \in |\delta' x|$. This means also that $(\delta' x).y \neq 0$. The problem is to relate $(\partial y).x$ and $(\delta' x).y$.

Remark that both $(\partial y).x$ and $(\delta' x).y$ are zero or nonzero together. A natural and simple constraint is to set $(\partial y).x = (\delta' x).y$. If this constraint is satisfied, we say that ∂ and δ' are *dual* operators. Figure 22 shows some example of this constraint.

We rewrite the property by remarking that $(\partial y).x = x.(\partial y)$ which makes the statment of the equality more symmetric and we recover the definition 12 of δ by linearity. We can summarize: the coboundary operator coincides with the boundary operator in the dual abstract complex. This gives also the interpretation of δ as a transport operation, see next section and illustration in figure 24.

One comes to recognize the relation $c^p.(\partial d_{p+1}) = (\delta c^p).d_{p+1}$ as a combinatorial form of *Stokes'theorem* [Sha90]. The Stoke's formula links the differential of a form ω and the boundary operator of a domain V: $\int_{\partial V} \omega = \int_V d\omega$. Take for example $p = 2$, then d_{p+1} is a volume. Interpret c^p as the integral (the sum) of a form (the coefficients of the cells in c^p) on some domain (the integral chain d_p). The equality says that the value taken by the function c^p on the surface boundary equals the value taken by the new form δc^p on the volume. This remark can be greatly refined, see for instance [Ton74, Ton76, CS00].

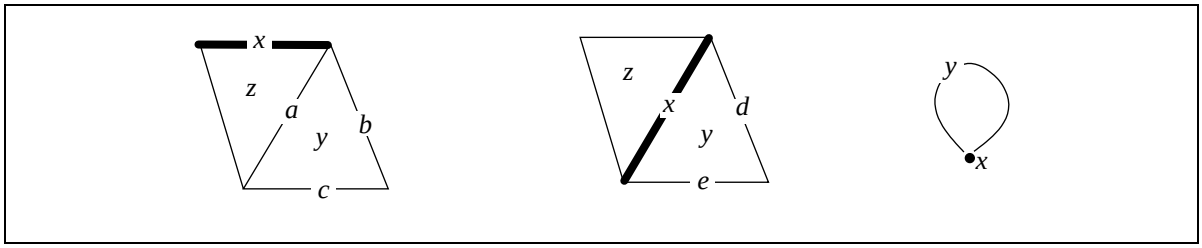


Figure 22: Example of dual ∂ and δ operators. We work in the integral chain group. For the complex to the left, we have $\delta x = z$ and $\partial y = a + b + c$. Then, $(\delta x).y = x.(\partial y) = 0$. For the complex in the middle, we have $|\delta x| = \{y, z\}$ and $|\partial y| = \{x, d, e\}$. If we define $\partial y = x + d + e$, then $x.(\partial y) = 1$ which implies that $(\delta x).y$ is nonzero. Duality further fixes the coefficient $(\delta x).y = 1$. For the complex to the right, we may specify $\partial y = 2x$ to state that the vertex x is encounter two times, at the two ends of the edge y. But then, we must fix $(\delta x).y = 2$, that is $\delta x = 2y$ to ensure the duality of ∂ and δ. The condition $\delta x = 2y$ can be interpreted as: the edge y is connected two times to the vertex x.

5.9 Arbitrary Labeling the Cells of a Complex

Suppose we want to label *some* of the cells of a complex with values taken in an arbitrary set Val. Such labeling can be represented by a *partial* function ℓ from $\mathcal{K}$ to Val. This partial function can be extended into a total function given the value $\perp$, $\perp \notin Val$, to the cells that have no image by ℓ. Then, the function ℓ can be seen as a chain if we give an abelian group structure to $Val \cup \{\perp\}$. We review two possibilities amongst others.

Labeling with $\mathrm{Idem}(Val)$. We can use the abelian idemgroup generated by Val. This group is denoted by $\mathrm{Idem}(Val)$. It contains all the subsets of Val written as sums, and the element 0 is the the empty set. We identify $\perp$ with 0 and a value $v \in Val$ with the corresponding singleton in $\mathrm{Idem}(Val)$. Then we can write $\ell = \sum_{x \in \mathcal{K}} \ell_x \, x$ where $\ell_x = \ell(x)$ if $\ell(x)$ is defined and 0 otherwise. An example is given in figure 23.

Note that using the group $\mathrm{Idem}(Val)$ instead of the set Val associates actually a subset of Val to each cell. By indentifying the singletons with the elements of Val, we represent the desired labeling in a natural way. Partiality is handled using 0 to represent $\perp \notin Val$.

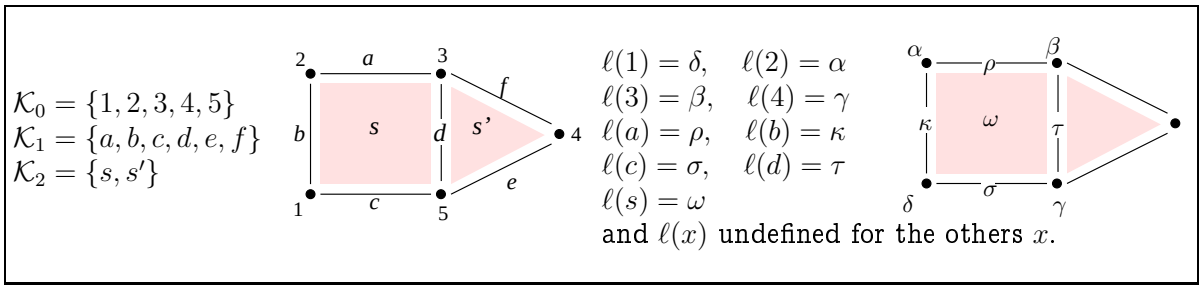


Figure 23: The labeling of the cell of an abstract complex. The figure in the left gives the abstract complex $\mathcal{K}$ and its p-cells $\mathcal{K}_p$ (for $p = 0, 1, 2$). The labeling ℓ is defined on the right. In this diagram, we indicate the images of the function ℓ by writing next to each cell the value of the function on that cell. This function has for codomain the set $Val = \{\alpha, \beta, \gamma, \delta, \rho, \tau, \sigma, \kappa, \omega\}$ which do not have an a priori abelian group structure. The function ℓ can be written as a chain of $C(\mathcal{K}, \mathrm{Idem}(Val))$: $\ell = \delta.1 + \alpha.2 + \beta.3 + \gamma.4 + \rho.a + \kappa.b + \sigma.c + \tau.d + \omega.s$. Note however that in $C(\mathcal{K}, \mathrm{Idem}(Val))$ there are also chains like $(\alpha +_{\mathrm{Idem}(Val)} \beta).1$ which would represents a function f such that $f(1) = \{\alpha, \beta\}$ and undefined elsewhere.

Labeling with $\mathrm{Abel}(Val)$**.** One can also use $\mathrm{Abel}(Val)$ instead of $\mathrm{Idem}(Val)$. We rely on the injection $x \mapsto x$ to represent an element of Val by an element of $\mathrm{Abel}(Val)$. This group has a richer structure and enables the association of a cell to a " generalized multiset " of Val elements. In a generalized multiset, an element can have a negative multiplicity.

Remark that if Val has already a group structure $+$, the operation in $\mathrm{Abel}(Val)$ does not coincide with the operation $+_{\mathrm{Abel}}$ in $\mathrm{Abel}(Val)$. Take for example $Val = \mathbb{Z}$, then $x +_{\mathrm{Abel}} (-x) \neq 0_{\mathrm{Abel}}$. Indeed, both x and $(-x)$ are generators of $\mathrm{Abel}(\mathbb{Z})$ and they are distinct.

Boundary and Coboundary as Transport Operation. In an arbitrary labeling of a complex, we can interpret the ∂ and δ operations as *transport* operations, see figure 24 and the references [Ton74, Ton76, PS93].

Suppose that we want to valuate the cells of the chains by an element of Val. We use the previous encoding based on $\mathrm{Abel}(Val)$ for the chain coefficients. We define the boundary of a cell x by:

$$\partial x = \sum_{y \prec x} y \qquad \text{and extend } \partial \text{ linearly:} \qquad \partial(\sum \alpha_x x) = \sum \alpha_x \partial x$$

Consider a cell x that have several successors in the chain. Then the effect of ∂ as a transport operation is to send to x the coefficients of theses successors. The result is conveniently gathered as a formal sum in $\mathrm{Abel}(Val)$ and no coefficients are lost (using $\mathrm{Idem}(Val)$ instead of $\mathrm{Abel}(Val)$ then we can record only the coefficients that appear an odd number of times). We can then further interpret " the collision at cell x of the transported values " using an homomorphism to resolve the " collisions " and to compute the final value of x.

To be more concrete, suppose that the cells in figure 24 (left) are valuated by reals, that is, we consider chains in $C(\mathcal{K}, \mathrm{Abel}(\mathbb{R}))$. For instance, take $\omega = 1.6$ and $\omega' = 3.1$ in chain ℓ_2. Then

$$\partial(1.6s + 3.1s') = 1.6a + 1.6b + 1.6c + (1.6 +_{\mathrm{Abel}} 3.1)d + 3.1f + 3.1e$$

We say that the value 1.6 coming from s and the value 3.1 coming from s', collide at cell d.

We want to combine colliding values into a real to get again a real valued chain. Suppose that the combination function is the sum of reals. Then we would use the homomorphism h from $\mathrm{Abel}(\mathbb{R})$ to $(\mathbb{R},+)$ that interpret the $+_{\mathrm{Abel}}$ as the usual $+_{\mathbb{R}}$. The homomorphism h between the groups of values, is easely extended into an homomorphism on chains, by defining $h(\alpha x) = h(\alpha)x$ for all cell x and then using linearity.

Instead of using a function h to combine the colliding values, we can work directly with chains in $C(\mathcal{K},(\mathbb{R},+))$. In this way, the combining function is directly the group operation of the chain coefficients. However, using $\mathrm{Abel}(\mathbb{R})$ and then an a posteriori homomorphism h is more general. For instance, suppose that we work with coefficients in $(\mathbb{R},+)$ but we want to combine the colliding values by multiplication. This is not easely expressed. But using $\mathrm{Abel}(\mathbb{R})$ at the first place, we have just to change the function h.

Intuitively, one can see the interest of using an abelian group for the coefficients. The combination function must not depend on the order of the combinations and then the chain $(\alpha+\beta)x$ must be equal to the chain $(\beta+\alpha)x$.

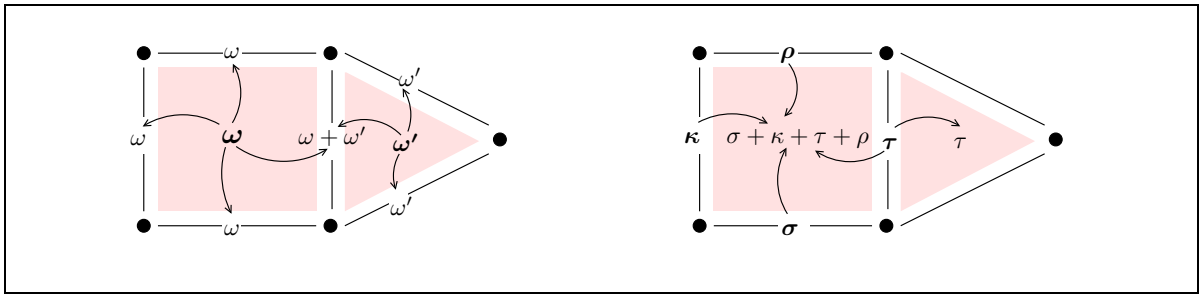


Figure 24: Depiction of the boundary and coboundary operation on chains. We consider the abstract complex already used in figure 23. The effect of taking the boundary operator ∂ on $\ell_2 = \omega.s + \omega'.s'$ is pictured by the diagram in the left. The figure in the right gives the effect of taking the coboundary δ of the 1-chain $\ell_1 = \rho.a + \kappa.b + \sigma.c + \tau.d$. In these two figures, the curved arrow indicate values (in bold) being transferred from a p-cell to the preceding $(p-1)$-cells (for ∂) and from a $p-1$-cell to the succeeding p-cells (for δ).

5.10 Topological Collections

A topological collection associates a value to some cells of a complex. In addition, we must be able to speak of the carrier of the collection (the cell that have a value), of the neighbor of an element, of subcollection and of the boundary of a subcollection. All these notions can be developped on top of the notion of chain complex presented above. The previous paragraph showed how arbitrary values can be associated to the cells using the notion of chain (or cochain). But then, it misses the representation of the coefficients used to compute the boundary structure.

The idea is naturally to represent both the coefficients in B and the label in $\mathrm{Abel}(Val)$. However, using the group $G = B \times \mathrm{Abel}(Val)$ seems at first sight not adequate: all the cells $(0_B,\alpha)$ are distinguished although they represent the same absence of a cell in a chain (because the coefficient 0_B) and then the value α does not matter. However, the definition

of an alternative to the cartesian product is not easy at all. For example, the construction $\big(B \times \mathrm{Abel}(Val)\big) \,/\, \big(\{0_B\} \times \mathrm{Abel}(Val)\big)$ collapses all the values $(0_B, \alpha)$ to $0_{B\times \mathrm{Abel}(Val)}$. But, all values (g, α) are collapsed on $(g, 0)$ which is certainly not what we want. So, does exists a product with projection π_1 and π_2 such that $\pi_2 x = 0$ whenever $\pi_1 x = 0$? Suppose that the group of coefficients used to compute the boundary is $\mathbb{Z}/2$; and suppose we have three 2-cells a, b and c such that $\partial a = 1+2$, $\partial b = 1+3$ and $\partial c = 1+4$ (imagine a graph with three edges and four vertices, the edges are linked by one end to the vertex 1 and to the other end to a unique vertex). We have $\partial(a+b+c) = 1+2+3+4$ so it seems that it is natural to have $\partial(\alpha.a + \beta.b + \gamma.c) = (\alpha+\beta+\gamma).1 + \alpha.2 + \beta.3 + \gamma.4$. But the term $(\alpha+\beta+\gamma).1$ is obtained as $\alpha.1 + \beta.1 + \gamma.1$. If the values $(0, \varepsilon)$ are identified with $(0, 0)$, then by computing first $(\alpha.1 + \beta.1) + \gamma.1$ we obtain the result $\gamma.1$ while computing $\alpha.1 + (\beta.1 + \gamma.1)$ we obtain $\alpha.1$. So, there is no product having the wanted property and we use simple the cartesian product.

Definition 14 (*Topological Collection*).

A *topological collection shape* is a triple $\mathcal{S} = (\mathcal{K}, B, \partial)$ such that $\mathcal{K}$ is a locally finite abstract complex of finite dimension and $C(\mathcal{K}, B, \partial)$ is an homological chain complex. A *topological collection type* is a pair of $\mathcal{T} = (\mathcal{S}, Val)$ where $\mathcal{S}$ is a shape and Val is an arbitrary set. A *topological collection* is a pair $(\mathcal{T}, c)$ where $\mathcal{T}$ is a topological collection type $((\mathcal{K}, B, \partial), Val)$ and c is a cochain: $c \in \mathit{Cochains}(\mathcal{K}, B \odot Val)$. The product $B \odot Val$ denotes the cartesian product $B \times \mathrm{Abel}(Val)$. The set of collections with a given type $\mathcal{T}$ is denoted by $TC(\mathcal{T})$; the set of collections with a given shape shape $\mathcal{S}$ is written $TC(\mathcal{S})$; the set of collection on a given complex $\mathcal{K}$ is written $TC(\mathcal{K})$, etc.

Often we omit to mention the shape or the type $\mathcal{T}$ of the topological collection when it is clear from the context; we says directly that a chain c is a topological collection and we write $c \in \mathcal{T}$ or $c \in \mathcal{S}$ if $\mathcal{S}$ is the shape and $\mathcal{T}$ the type of c.

The cochain group $\mathit{Cochains}(\mathcal{K}, B \odot Val)$ is called the *full cochain group* associated to the type $\mathcal{T}$. The cochain group $\mathit{Cochains}(\mathcal{K}, B)$ is called the *shape cochain group* associated to $\mathcal{T}$. And $\mathit{Cochains}(\mathcal{K}, \mathrm{Abel}(Val))$ is called the *value cochain group*.

If c is a collection, and $x \in \mathcal{K}_p$, then $c(x) = (g, u)$ with $g \in B$ and $u \in \mathrm{Abel}(Val)$ and we say that *the value of c at x is u*. The functions c_b and c_v are the first and second projection of c. That is, $c_b(x) = g$ and $c_v(x) = u$ for $c(x) = (g, u)$. The functions c_b and c_v associate an element of a group to a cell and then are cochains: $c_b \in \mathit{Cochains}(\mathcal{K}, B)$ and $c_v \in \mathit{Cochains}(\mathcal{K}, \mathrm{Abel}(Val))$.

For all collection c we have $|c_v| \subset |c|$ and $|c_b| \subset |c|$. The set $\mathrm{Residu}(c) = \{x \in \mathcal{K} \mid c_b(x) = 0_B \text{ and } c_v(x) \neq 0_{\mathrm{Abel}(Val)}\}$ is called the *residu* of the collection. We usually omit the subscripts of 0 and rely on the context to make clear on which group 0 belongs. A collection c is *residu-free* if $\mathrm{Residu}(c) = \emptyset$.

A topological collection c is *flat* if $c_v(x) = 0$ or $c_v(x) \in Val$ for all $x \in \mathcal{K}$. It is *monolayer* if c_b is a p-cochain for some p, i.e. it exists an integer p such that $|c_b| \subset \mathcal{K}_p$.

Integral and Modulo 2 Shapes. An important case is when $B = \mathbb{Z}$ or $B = \mathbb{Z}/2$. In this case, we say that a chain c has an *integral shape* or a *modulo 2 shape* respectively. We use

a special notation for integral and modulo 2 chain:

$$c = \sum \alpha_x.n_x x$$

where $n_x \in \mathbb{Z}$ or $\mathbb{Z}/2$, and $\alpha_x \in \mathrm{Abel}(Val)$. But the terms $\alpha.(-1)x$ are written simply $-\alpha.x$ and $\alpha.x$ is for $\alpha.1x$. For instance, $c = \texttt{"abc"}.2x - \texttt{"def"}.y + \texttt{"rosae"}.z$ stands for a chain c such that: $c(x) = (2, \texttt{"abc"})$, $c(y) = (-1, \texttt{"def"})$ and $c(z) = (1, \texttt{"rosae"})$.

Subcollections. We need now to introduce the notion of subcollection of a collection. The *restriction* $c\backslash S$ of a topological collection c by a set S is the chain $c\backslash S$ defined by $(c\backslash S)(x) = c(x)$ if $x \in S$ and else $(c\backslash S)(x) = 0$. A restriction is too general to represent a subcollection: a subcollection is a connected part of a collection. It must be represented by a chain too.

Definition 15 (*Split, Patch and Subcollection*). Let c be a cochain and c' and c'' be two cochains such that $|c'| \cap |c''| = \emptyset$ and $c = c' + c''$. Then we say that c' and c'' are a *split* of the cochain c and we write $c \trianglerighteq c'$, $c \trianglerighteq c''$ and $c'' = \complement_c c'$ or $c' = \complement_c c''$. A cochain c' is a *patch* of the cochain $c \in \mathit{Cochains}(\mathcal{K}, G)$, if $c \trianglerighteq c'$ and if $\mathrm{Shape}\,|c'|$ is a connected set of $\mathcal{K}$. Let c be a collection; a collection c' is a subcollection of c if $c' = c\backslash|c'|$ and if c'_b is a patch of c_b.

5.11 Transformations

We want now define several kinds of transformations of a topological collection.

Definition 16 (*Shape-preserving, Pointwise and Local Operations*).

A function f from $TC(\mathcal{S}, Val)$ to $TC(\mathcal{S}, Val')$ is *shape preserving* iff for all c, $(fc)_b = c_b$. It is *pointwise* if it is shape preserving and if it exists a function $g : \mathrm{Abel}(Val) \to \mathrm{Abel}(Val')$ such that $(fc)_v = g \circ c_v$. It is *local* if it is shape preserving and if it exists a function $g' : TC(\mathcal{S}, Val) \to \mathrm{Abel}(Val')$ such that $\big(fc\big)_v(x) = g'\big(c\backslash(\overline{\mathrm{St}}\,x)\big)$.

Variations on the notion of locality are obtained by changing $\overline{\mathrm{St}}\,x$ for $\mathrm{St}\,x$ or $\mathrm{Lk}\,x$ or $|\overline{x}|$, etc.

Definition 17 (*Renaming Operations*). Let h be a bijection from $\mathcal{K}$ to $\mathcal{K}'$. Then, the *renamed complex* $\mathcal{K}' = h(\mathcal{K})$ is such that $\dim_{\mathcal{K}'} x' = \dim_{\mathcal{K}} h^{-1}(x')$ and $x' \prec_{\mathcal{K}'} y'$ iff $h^{-1}(x') \prec_{\mathcal{K}} h^{-1}(y')$. If $\mathcal{S}$ is a collection shape $(\mathcal{K}, B, \partial)$, then the *renamed shape* $\mathcal{S}' = h(\mathcal{S})$ is defined by $\mathcal{S}' = (\mathcal{K}', B, \partial')$ where the boundary operator ∂' is defined by: $\partial' x' = \sum g_y h(y)$ if $\partial h^{-1}(x') = \sum g_y y$. The *renaming of the collection* c into $h(c)$ is a function from $TC(\mathcal{S})$ to $TC(h(\mathcal{S}))$ such that $h(c)(x) = c(x).h(x)$.

We can now define the basic transformation described in section 2.1 page 12. The basic intuition hidden behind this definition is sketched in figure 25. Note that we do not describe a device to select a subcollection into a collection, neither we give condition on the gluing of the substituted subcollection. We just specify that untouched parts of the collection must remain untouched, both from the value point of view (condition 1) and the shape point of view (condition 2).

DEFINITION 18 (*Split, Patch and Non-Local Substitutions*). Let c and d be collections with respective subcollections c' and d'. Then d is a *patch substitution* of c' by d' if the two following conditions hold:

1. $\mathsf{C}_c c' = \mathsf{C}_d d'$
2. $\mathrm{Shape}\,|\mathsf{C}_c c'| = \mathrm{Shape}\,|\mathsf{C}_d d'|$

If we relax the connectivity condition on d', then we say that we have a *split substitution*. If the condition is also relaxed for c', then we have a *distributed* (split or patch) *substitution*. If it exists a function f such that $d' = f(c\backslash|\,\overline{\mathrm{St}}\,c'|)$ then the substitution is said *computed by* f. In addition, the substitution is *coboundary preserving* if $\delta c'_b = \delta d'_b$ and *boundary preserving* if $\partial c'_b = \partial d'_b$.

The figure 26 gives several examples of various kinds of substitution.

Note that the operator ∂ and its dual δ defined for a collection do not appear explicitly in the straight definition of a substitution. However, they comes into play when one has to specify precisely the process of gluing d' and c'' into the new collection d.

There is several variations on the notion of " computed by f " to accomodate the possible variation on the neighborhood notion.

DEFINITION 19 (*Simple Transformation*). We say that d is a *simple transformation of type* n of c iff:

— *type I:* it exists a pointwise or a local function f such that $d = f(c)$;

— *type II:* d is a renaming of c;

— *type III:* it exists subcollections c' and d' of c and d such that d is a patch substitution;

— *type IV:* idem but with a split substitution;

— *type V:* idem but with a non-local substitution.

A pointwise function is a patch-, boundary and coboundary preserving- substitution computed by a function. The current version of the MGS interpreters allow only this kind of substitutions, see section 6.

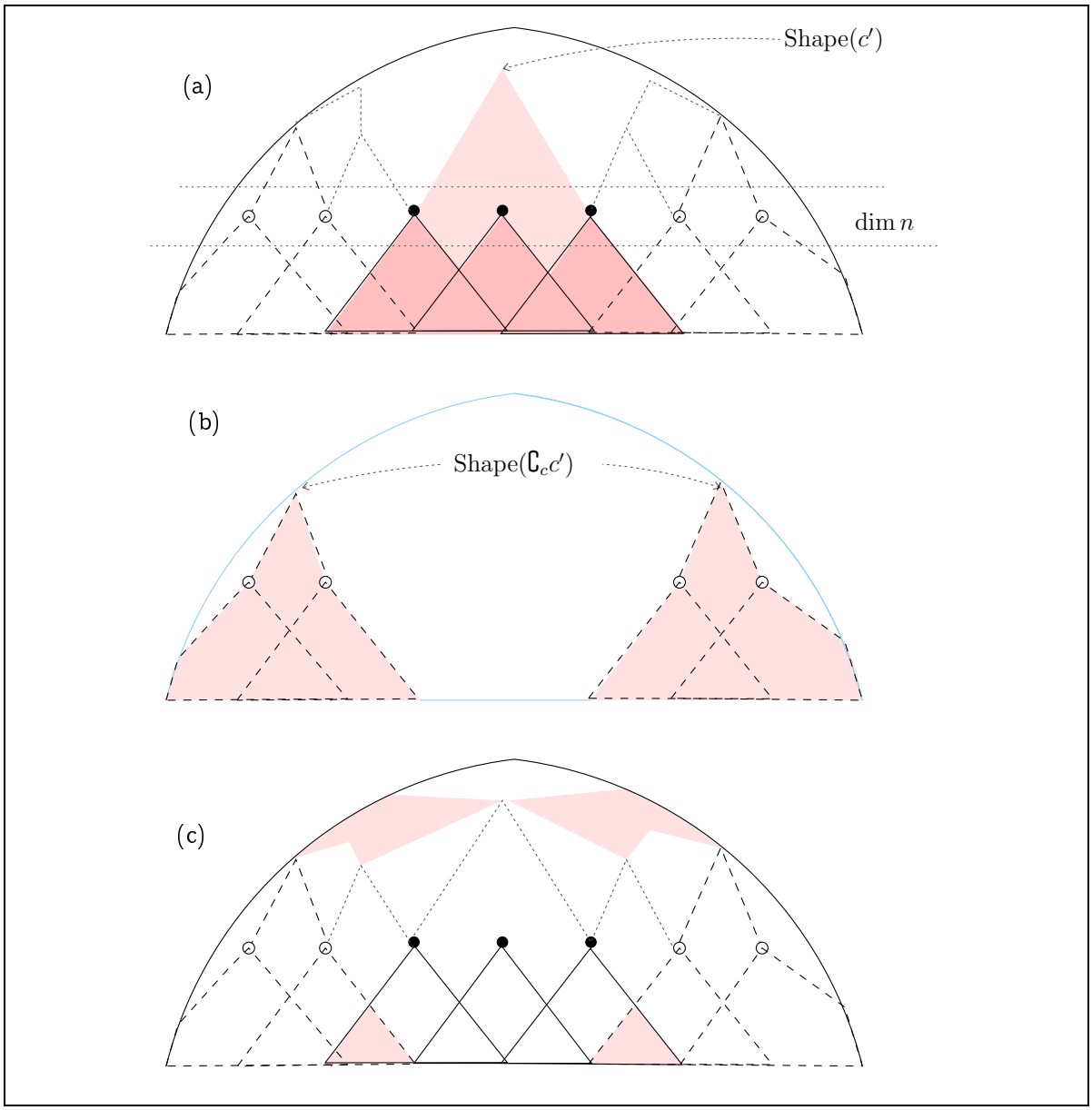


Figure 25: Parts of a complex involved in a substitution.
We have pictured symbolically the abstract complex $\mathcal{K}$ as a Hasse diagram (cf. Fig. 18). The carrier of the monolayer chain c consists in all the n-cells pictured as circle (diagram (a)). The three black circles in the middle specify the carrier of the subcollection c'. Consequently, the four empty circles are the carrier of $c'' = \complement_c c'$.
The shape $\mathrm{Shape}(c')$ of c' is sketched as the gray region in diagram (a): the subcomplex $\overline{|c'|}$ spanned by c' is in dark gray while the p-cells above this subcomplex are in light gray. The shape $\mathrm{Shape}(c'')$ is sketched in gray in diagram (b). This part of the complex must remain unchanged across the substitution.
The diagram (c) has two gray regions, one near the top and one near the bottom (each is composed of several parts). The region near the bottom, corresponds to the intersection $\mathrm{Shape}(c') \cap \mathrm{Shape}(c'')$. Cells in this region have a dimension less than n. The definition of a substitution says that this region must remain unchanged in the final result (because the belongs to the shape of c'' and then must not be touched by the transformation).
The region near the top corresponds to the p-cells x, $p > n$, such that $\overline{x}$ has an intersection both in $\overline{|c'|}$ and $\overline{|c''|}$. The definition of a substitution does not say anything about such cells. However, if the $n+1$-cells remain identical accross the transformation, then the transformation is said coboundary preserving.

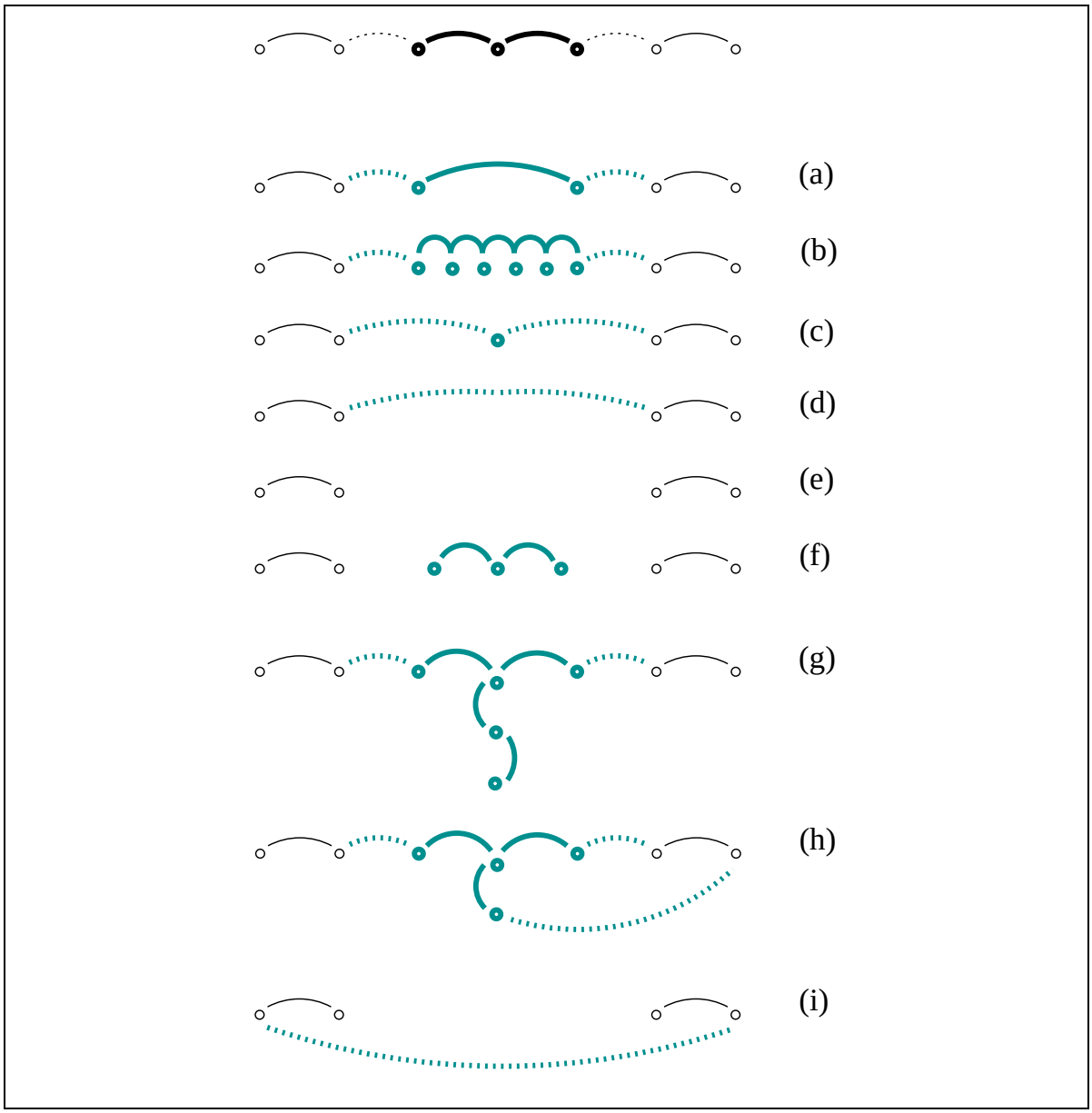


Figure 26: Substitutions in a line graph.
The shape of the collection c is a line graph with 7 vertices. The collection is monolayer and we assume that all vertices have a value. The first diagram indicates the subcollection c' with shape $\mathrm{Shape}(c')$ indicated in bold (bold edges and/or bold vertices, however note that only the vertices appear in the chain c'). The split $c'' = \complement_c c'$ are the vertices draw as empty circles. The dotted edges are the 1-cells that are not in $\mathrm{Shape}(c')$ nor in $\mathrm{Shape}(\complement_c c')$. The other diagrams give several possible substitutions of the subcollection c'. The shape of the substituted collection d' is pictured in gray. The dotted edges do not belongs to $\mathrm{Shape}(c'')$ or to $\mathrm{Shape}(d')$: they are dependant of the substitution process. One can imagine that they come from the handling of the p-cells, $p > n$ that are neither in $\mathrm{Shape}(c')$ nor in $\mathrm{Shape}(\complement_c c')$ (in the current version of MGS, this handling is fixed and depends of the collection kind, i.e. the type of the underlying topology). Because a collection is a 0-chain, there is no intersection between $\mathrm{Shape}(c')$ and $\mathrm{Shape}(c'')$, so there is no constraint in the collection d'. All examples are non-distributed substitutions. Examples (e) and (f) are split substitution. Example (d) and (i) are examples where the chain d' is reduced to 0 but the underlying topology is nevertheless changed. Because chains are 0-chains, all transformations are necessarily boundary preserving (because the boundary of a 0-cell is 0). Examples (a – c, g) are coboundary preserving: this implies that the dotted edges are identified with the dotted edge in the initial collection.

5.12 The Example of a 2D Grid

To illustrate the previous notions, we gives here a possible model for 2D arrays. Topological structure for set, multiset and sequence are sketched in the next section.

Two dimensional grids will be rendered by flat, monolayer topological collections. From this point of view, an array is a labeled graph: values are carried by vertices and the connections rely on edges.

For, we define the abstract complex $(\mathcal{G}, <)$ by

$$\begin{aligned}
&\mathcal{G}_0 = \mathbb{Z} \times \mathbb{Z},\\
&\mathcal{G}_1 = \big\{\{x, y\} \mid x, y \in \mathcal{G}_0, x - y = (0, 1) \text{ or } x - y = (1, 0)\big\}\\
&x \prec \{x, y\} \quad \text{and} \quad y \prec \{x, y\} \qquad \text{for } x, y \in \mathcal{G}_0
\end{aligned}$$

The abstract complex $\mathcal{G}$ is not finite but locally-finite. The shape of a 2D grid is the triple $(\mathcal{G}, \mathbb{Z}/2, \partial)$ with ∂ defined by:

$$\partial x = 0, \text{ for } x \in \mathcal{G}_0 \qquad \text{and} \qquad \partial\{x, y\} = x + y, \text{ for } x, y \in \mathcal{G}_0$$

The shape of a 2D grid is homological, because, for $u \in \mathcal{G}_1$, $\partial^2 u = \partial(\sum_{x \in \mathcal{G}_0} x) = \sum 0 = 0$ because $\partial x = 0$ for $x \in \mathcal{G}_0$.

A *data field* with element in Val is an element of the full 0-cochain group $C^0(\mathcal{G}, \mathbb{Z}/2 \odot Val)$. A data field generalizes the notion of array considering non rectangular shapes for functional arrays, see [GMS96, Lis93]. A data-field is a monolayer collection.

Let $x = (a, b)$ be in $\mathcal{G}_0$, then

$$\begin{aligned}
\overline{\mathrm{St}}(x) = \big\{ &(a, b),\ (a - 1, b),\ (a + 1, b),\ (a, b - 1),\ (a, b + 1),\\
&\{(a, b), (a - 1, b)\},\ \{(a, b), (a + 1, b)\},\ \{(a, b), (a, b - 1)\},\ \{(a, b), (a, b + 1)\}\ \big\}
\end{aligned}$$

Then, it must be obvious that a type I transformation replaces the value of a vertex x by a value computed from the **4-neighbors** (the so-called *Von-Neuman* neighborhood) of x. See figure 27.

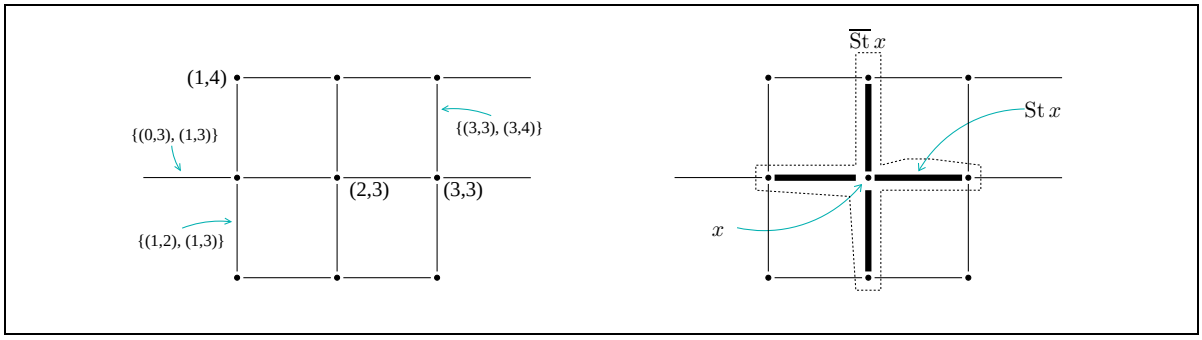


Figure 27: Modelling of 2D grids.

5.13 Summary

We have defined a topological collection c to be a chain on a given chain complex that describes the topology of the collection and a labeling of the cells. A substitution replaces a

subchain c' by another subchain, preserving the topological structure of the complement of c' in c: $\complement_c c'$. What remains to be done is:

- to devise various devices to specify the subcollection to be substituted;
- to design several constructions, *available at user-level*, to specify how the new collection d' must be injected into the old one at the place of c'.

Actually, the strategy implemented in the current version of the MGS interpreters embeeds the new collection into the old one using a fixed strategy depending on the collection type. These strategies are described in the next section. Thus, it is not possible to change the topological structure by the application of a transformation. The motivating example presented in section 1.5 is still out of reach.

6 Comparaison with Other Approaches

We want to show that some widely used computational models can be seen as specific instances of transformations of some topological collections. The level of the discussion is informal.

Four Biologically Inspired Computational Models. One of our additional motivations is the ability to describe generically the basic features of four models of computation: Γ and the CHAM, P systems, L systems and cellular automata (CA). They have been developed with various goals in mind, e.g. parallel programming for Γ, semantic modeling of nondeterministic processes for the CHAM, calculability and complexity issues for P systems, formal language theory and biological modeling for L systems, parallel distributed model of computation for CA (this list is not exhaustive). We assume that the reader is familiar with the main features of these formalisms but a short description of these computational models is given below for the readers convenience.

All these computational models rely on a biological or biochemical metaphor. It is then natural to require their integration in a uniform framework. Because they fit harmoniously, we gain confidence that the underlying concepts of topological collection may reveal as unifying and covering a broad class of biological DS with a dynamical structure.

The Multi-Agent Paradigm. This section ends by comparing the appraoch of topological collections with the multi-agent modeling paradigm. We show that the main difference relies on the entity on which the evolution function (i.e. transformation in the case of topological collections and the behavior in the case of multi-agent) is linked.

6.1 The topology of Sets and Multisets: the programming language Γ and the CHAM

The computational model underlying Γ [BM86, BCM87] is based on the chemical reaction metaphor; the data are considered as a multiset M of molecules and the computation is a succession of chemical reactions according to particular rules. A rule (R, A) indicates which kind of molecules can react together (a subset m of M that satisfies predicates R) and the product of the reaction (the result of applying function A to m). Several reactions may be possible at the same time. No assumption is made on the order on which the reactions occurs. The only constraint is that if the reaction condition R holds for at last one subset of elements, at least one reaction occurs (the computation does not stop until the reaction condition does not hold for any subset of the multiset).

The CHemical Abstract Machine (CHAM) extends these ideas with a focus on the expression of semantic of non deterministic processes [BB89]. The CHAM is an elaboration on the original Γ formalism introducing the notion of subsolution enclosed in a membrane. It is shown that models of algebraic process calculi can be defined in a very natural way using a CHAM: the fact that concurrency (between rule application) is a primitive built-in notion makes proof far easier than in the usual process semantics.

The Topology of Sets. Informally, an element in a set (or in a multiset) is a neighboor of any other element. Hence, the MGS pattern x, y in a rule selects an arbitrary pair in the set, and the pattern $x+$ selects an arbitrary non-empty subset.

Using the technical notions introduced in section 5, we can describe this situation more formally. A set V is represented by a topological 0-collection on a one dimensional shape with vertices V and only one edge $\top$. The function ∂_1 is defined by $\partial_1 \top = \sum V$. With this definition, an element of V is connected with any other element. The chain group describing a set is then particularly simple: $C_p = 0$ for $p \neq 0$, $K_0 = V$ and $C_0 = C_0(\mathcal{K}, \mathbb{Z}/2 \odot V)$. A set V corresponds to the chain $\sum_{x \in V} x.x$ using the notation described in page 57.

Let c' be the subcollection to be replaced by d' into the collection c to give a new collection d. The fixed strategy used to build d from d' and $c'' = \complement_c c'$, is simply to set $\top_d = |c''| \cup |d'|$.

This description is only combinatorial and does not admit a geometric realization. Indeed, a geometric 1-cell is homeomorphic to the interval $[0, 1]$ and then admits only two 0-cells in its boundary. If one insists to have a geometric realization of topological sets, then it is enough to shift the dimension of the cells by one: the elements of V are the many edges of one unique face.

The Topology of Multisets. A multiset M of element $e \in E$ can be represented by a set $\hat{M} \subseteq \mathbb{N} \times E$. If $e \in M$ with multiplicity n, then the n elements $(1, e), (2, e), ..., (n, e)$ belong to $\hat{M}$. The multiset M is represented as the 1-collection associated to the set $\hat{M}$. With this encoding, two arbitrary multiset elements are connected, in accordance with the fact that any submultiset can be matched and replaced in a Γ rule. Furthermore, the application of one Γ rule on a multiset M is a local, boundary preserving, patch substitution.

6.2 Nesting of Multisets: P systems

P systems [Pau98, Pau00] 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 pas through a membrane or dissolve its containing membrane. As for Γ, the computation is finished when no object can further evolve.

The P Systems Topology. The case of P systems is more interesting, because the topology can be used to take into account the locality of a computation step. In this approach, the region associated to a membrane would be a 2-cell and the membranes would be 1-chain; then a P system is viewed as a 2-collection with a 2-complex organization. Note that membrane systems are sometimes described by a sequence of well balanced parenthesis, which specify only the relative inclusion of membrane and not their connection in one level. The corresponding topology is then weaker. In the opposite, the organisation enabled by the two dimensional mapping of the membrane in a plane is weaker than the combinaisons enabled by 3D membranes, etc.

A cruder approach just associates a multiset M to the region associated with the skin of a P system. The difference with Γ is that the elements of M can be multiset themselves, associated to the inner membranes. In this approach, P systems are viewed as a theory of nested multiset rewriting.

6.3 The Topology of Sequences: L systems

L systems are a formalism introduced by A. Lindenmayer in 1968 for simulating the development of multicellular organism. Related to abstract automata and formal language, this formalism has been widely used for the modeling of plants. A L systems can be roughly described as a grammar. The productions are applied in parallel in a non deterministic manner. 0L system are context-free grammar. D0L system are deterministic context-free grammar: given a letter A there is at most one production that can be applied. Parametric L systems deal with *module* instead of letters: a module is a letter associated with a list of parameters. The production rules are extended with conditions on the parameters. For example,

$$A(x,y) : y \leq 3 \quad \longrightarrow \quad A(2x, x+y)$$

is a rule that can be applied to the module $A(2,5)$ to gives the module $A(4,7)$. This rule cannot be applied on $A(7,1)$ because the first parameter x, does not match the condition.

The Topology of Sequences. Section 5.12 gives a formalization of the topology of a grid. The model can be weakened to gives the topology of a sequence.

A sequence $\ell = <\ell_1, \ell_2, \ldots, \ell_n>$ is a 0-collection whose shape is a chain complex of dimension 1. Let i_k be n reals in increasing order; the underlying complex $\mathcal{K}$ is defined by

$$\begin{aligned} \mathcal{K}_0 &= \{i_1, \ldots, i_n\} \qquad \text{such that } i_j < i_{j+1} \\ \mathcal{K}_1 &= \big\{(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)\big\} \\ \partial(k, k') &= k +_C k' \end{aligned}$$

(this last sum is a formal sum). The shape is $C(\mathcal{K}, \mathbb{Z}/2, \delta)$. Hence, ℓ is represented by the chain $\sum_{1 \leq j \leq n} \ell_j.i_j$ (using the notation described in page 57).

An MGS rule $c' => d'$ applied to a topological sequence c corresponds to a substitution with result d. The strategy used to glue the new subcollection d' and $c'' = \complement_c c'$ into the result d is the following:

- if $d' = 0$ (that is, the MGS rule cancel c') then $\mathrm{Shape}(d) = \mathrm{Shape}(c'')$;
- if $d' \neq 0$, then $\delta c' = \delta d'$ (the δ in the left hand side must be taken in the shape of c while the δ in the right hand side must be taken in d). This condition, together with $d = d' + c''$, is enough to specify completely $\mathrm{Shape}(d)$: $\mathrm{Shape}(d) = \mathrm{Shape}(d') \cup \mathrm{Shape}(c'') \cup |\delta c'|$.

In a MGS rule, the sequence d' is computed solely as a function from the subsequence c'. Thus, if $d \neq 0$, the MGS rule is a patch-, boundary and coboundary preserving- substitution computed by a function.

Topology of Context-Free Sequence. However, we can propose an alternative. Indeed, for D0L-system, the right hand side of a production rule is limited to only one element: there is no interaction with the neighborhood and the corresponding grammar is context-free. In MGS terms, it means that all rule have the form:

$$(x/\dots) \quad => \quad \dots$$

This property can be enforced using a more precise model, that forbids a dependance between an element and its neighboor in the substitution process.

A sequence $v_1, ..., v_n$ of n values is then represented by a 1-collection with shape $C(K, \mathbb{Z}/2, \partial)$ defined by: $K_0 = \{0, 1, .., n\}$ and $K_1 = \{(0,1), (1,2), ..., (n-1, n)\}$. We have $\partial_0 x = \emptyset$ and $\partial_1(i,j) = i + j$ (where the sum in the right is the group operation of C_1). And a collection c is a monolayer 0-chain $c = v_1.(0,1) + ... + v_n.(n-1, n)$.

In this formalization, the application of only one production $x \rightarrow f(x)$ of a D0L system is a local, boundary and coboundary preserving, patch substitution computed by f. Rigorously, the argument of f is the the chain $c\backslash\overline{\mathrm{St}}\, x$, but with our topology, $c\backslash\overline{\mathrm{St}}\, x = c\backslash x$ which show that the new value replacing x depends only on x.

6.4 The Topology of Arrays: Cellular Automata

Cellular automata (CA) have been invented many times under different names: tessalation automata, cell spaces, iterative arrays, etc. However, a fair fraction of the computer research on two-dimensional cellular automata has its ultimate origins in the work of J. Von Neumann to provide a more realistic model for the behavior of complex system in biology [VN66].

In a simple case, a 2D cellular automaton consists in a grid of cells or sites, each with a value taken in a finite set $\mathcal{V}$. The values are updated in a sequence of discrete time steps, according to a definite, fixed, rule. Denoting the value of a site at position (i,j) by $a_{i,j}$, a simple rule gives its new value as $a'_{i,j} = \varphi(a_{i,j}; a_{k_1}, ..., a_{k_p})$, where φ is a function from $\mathcal{V}^{p+1}$ to $\mathcal{V}$ which specifies the rule, and where the a_{k_j} are the values of the p neighbors of site (i,j). For example, the Von Neumann neighbors of a cell (i,j) are the four cells $(i-1,j), (i+1,j), (i,j-1)$ and $(i,j+1)$.

Many variations are possible: organization of the cells in a regular lattice of any dimensions or even in a general graph, variable neighborhood, various finite set $\mathcal{V}$. However the main characteristics of CA are largely unaffected by such additional complications.

The Topology of Arrays. The topology of arrays has been introduced in section 5.12. A rule of a cellular automata is a local, shape preserving, patch substitution computed by a function (the evolution function of an elementary cell).

6.5 Production Systems, Rewriting systems and All That

Production systems is a term used in artificial intelligence to describe systems specified as a set of production rules acting on a global database under the supervision of a control

system [Nil80]. Rules are associated with applicability conditions and the control system chooses the next rule to apply. The termination of the computations is determined by a global termination condition.

More specifically, in the field of grammar systems, rewriting systems or formal language theory, the idea of many possibly different local derivations relations integrated according to specific strategies, has been extensively studied and applied with different patterns in the search for new modeling approaches of biosystems [Man01, Pau00, Pau98, FMP00]. In these approaches, simple component are specified together with their integration: monolithic complex systems are reduced, by means of cooperation and distribution, in terms of simpler parts.

MGS participates of these approaches. For instance, an MGS program can be seen as a production system, the termination condition holding when the fixpoint has been reached. The originality of the MGS approach lies in the emphasis put on the topological view of the rules and on the database, while production system are often based on logical inferences or grammatical formalisms. The mechanisms used to describe the integration of the different parts rely heavily on the topological structure of the system, which is a natural tools for describing such complex systems.

6.6 A Comparison with the Multi-Agent Modeling Paradigm

The multi-agent paradigm is often advocated for the modelization of complex dynamical systems. Thus, we want to compare the approach of topological collections and the multi-agent approach. To make the comparison more concrete, we turn our attention again on the neurulation example introduced in section 1.5.

Let us simplify drastically the description of the neurulation for the sake of its simulation. We consider the neural plate in isolation and we assume that the system evolves by discrete steps. Furthermore, we suppose that there is no cell creation nor destruction; the neural plate is modeled as a linear sequence of n cells in the plane and the left and right extremities of this sequence are glued together at the end of the neurula stage; see Fig. 28

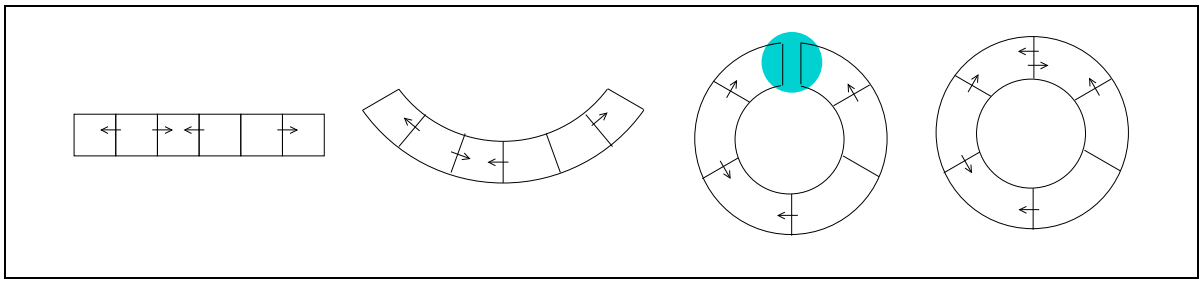


Figure 28: A simplification of the neurulation for simulation purposes.

The chemical state $s_i^c(t+1)$ of a cell i of the neural plate at time $t+1$ depends only of the own cell activity and of the signals received from the neighboring cells at time t. This can be written:

$$s_i^c(t+1) = h_i(s_i^c(t)\,;\; s_{i_1}^c(t), ..., s_{i_k}^c(t))$$

where the cells $i_1, ..., i_k$ are the cells in the neighborhood of i and h_i the evolution function of the cell i. Let $\mathcal{V}(t; i)$ denotes the set of neighbors of cell i at time t. The function $\mathcal{V}$ depends of t because there is a change in the neighborhood of the cells at the extremities.

Now, we have to face the problem of building the representation the state S of the entire system from the chemical state of the cells. As a matter of fact, the chemical state of the cells does not describe completely the system. One has to describe also its structure, that is, the organization of the cells. We can make three choices for the representation of the structure:

- avoid the description of the structure,
- implicitly distribute the description of the structure over the state of the cells,
- explicitly specify the structure beside the chemical state.

We will see that the first choice is not an option when dynamical structures come into the play. The second approach is the approach taken to the multi-agent paradigm. The last one corresponds to the MGS way of thinking.

Avoiding the description of the structure. The complete state S of the entire system is just a *set* of n chemical state s^c, where n is the number of cells in the systems.

This approach is not satisfactory at all because it misses the information related to the neighboring of each cell. The function $\mathcal{V}$ is simply ignored. But without this information we cannot fill h with the right arguments and hence, we cannot compute the trajectory of the DS.

Note however that this approach is the standard one when the structure of the DS is static: instead of a set, one use any organisation relevant to describe the fixed organisation of the cells (e.g. a vector). In this case, we write $s_i^c(t+1) = h_i(s_1^c(t), ..., s_n^c(t))$ and the function $\mathcal{V}$ is " hard-coded" in the functional dependencies between the arguments of h_i. It is because the time-dependance of $\mathcal{V}$ that this approach fails.

The multi-agent approach: Implicit Distributed Representation of the Organization. The complete state s of a cell corresponds to its chemical state s^c together with the information s^n relevant to describe its neighborhood: $s = (s^c, s^n)$. The complete state S of the entire system is then the *set* of the complete state of the cells.

This approach fits with the " multi-agent paradigm" for system modeling: each cell is an autonomous agent interacting with the others. The s^n part of each agent state corresponds to a *distributed representation* of the function $\mathcal{V}$. There is several possible drawbacks with such an approach:

1. Although complete, the description of the organisation of the cells is implicit and spreaded on the s_i^n. This does not ease any reasoning on the evolution of the system nor its implementation.

   For instance, suppose that s_i^n represents the position (x, y) of the cell i in the plane. The connection between the cells must be recovered by examining all the cells position, in

contradiction with the local nature of the evolution process. The global scan of all the cell states can be avoided if we store the adjacency relationship instead of the absolute position of each cell in the plane. But the problems listed below continue to hold.

2. The information scattered in the s_i^n is redundant. For example, assuming that the space of cells is isotropic, if s_i^n indicates that cell j is a neighbor, then s_j^n must indicate that i is a neighbor. However, there is no special way to ensure the coherency of the redundant informations.

3. The distributed representation of the function $\mathcal{V}$ is to loose. If we want to avoid the multiplication of cell types, a cell with two neighbors must be represented in the same manner as a cell with only one neighbor. Then, there is no mechanism that prevent a misuse of such cell.

4. The behavior h_i of the agent i embeeds the bookkeeping needed to compute the information s^n which inhibits its reuse in another context where the chemical behavior of the cell is the same but with another spatial organisation.

5. Most of the time, there is no change in the organisation of the cells, only the chemical state is changing. However, the changes in the organisation are handled at the same level as the changes in the own cell state.

6. Interaction, that is the influence of an entity to the evolution of another, is implicit. Even if strategies fixing the interaction between agents exist, the set of participants of an interaction is not a first citizen object of the language.

   For instance, there is no natural object in the basic multi-agent paradigm that represent a pair of (interacting) agents. This kind of entity can be modeled by a new kind of compound agent, but the managment of the aggregation will be tedious and " by hand " (for instance, suppose that the aggregation of two agents in a pair is valid only when the two agents are not far one from the other). In other word, there is no support for *spatial or logical aggregation.*

MGS: **Explicit Specification of the Organisation.** The complete state S of the systems at time t consists in the set of the $s_\cdot^c(t)$ together with the function $\mathcal{V}(t,\cdot)$. That is, at time t the $s_\cdot^c(t)$ are organized in a collection with topology given by $\mathcal{V}(t,\cdot)$. We recover the MGS model. The advantages are the following:

1. The description of the organization is explicit through the topology of the collection. And the relationships between the evolution function and the topology of the collection are explicitly given in the MGS rules, through the appearance of the neighborhood operators.

   It is then more easier to reason on the evolution process. Static analysis (through typing, abstract interpretation, structural results on homomorphisms, etc.) is possible.

2. The specification of the evolution function is still *local* (w.r.t. the topology, using transformation rules).

Locality is of paramount importance. As a matter of fact, it enables the application of transformation rules on an unknown global structure and the construction/computation of a global dynamical structure with only a static set of fixed local rules of changes.

3. It is more easy to specify generic h functions that can be used with several topology. A rule in MGS specifies an *interaction* between *several* entities. For example a pattern "x, y" in the left hand side of a rule specifies an interaction between two entities. The global structure do not appear in the rule, neither the exact representation of the topological link between x and y. Only the logical neighboring relationship between x and y is mentioned. This makes possible the use of the same rule with several topology, cf. section 4.1.

4. More generally, the separate specification of the structure and the evolution rules is an effective way to reuse evolution rules and thus to cut down the combinatorial explosion of the *behavior* $\times$ *structure* specifications.

7 Conclusion

In the current implementation, records, sets, multisets and sequences of elements are supported. Elements are of any types, allowing arbitrary nesting. Implementation of arrays is in progress and group-based data fields (GBF which generalizes functional arrays, cf. [GMS96, GM01]) are planed in a short term. We also plan to study a generic implementation of topological collections based on G-maps [Lie91].

The perspectives opened by this preliminary work are numerous. Here are some of them.

- The topological formalization of the MGS computation mechanism must be developped. For instance, the formal notions developped here are purely descriptive, in the sense that there is no prescription of the device used to select a subcollection. The situation is analogous as the description of the lambda-calculi at the point where one has defined the beta-reduction as a relation: it remains, to effectively compute a normal form, to define a strategy of reductions and to study the interaction between this strategy and the reduction, etc.

- Furthermore, the current characterization of the transformations are rather poor and we are very confident that they can be greatly improved. Our main goal in section 5 was to introduce for a reader with a background in computer science, some of the topological notions on which a theory of transformations of topological collections can be build.

- Based on the topological background, it must be possible to design some constructions to let the programmer specify the gluing of the new replacement subcollection into the old one. This is necessary if one want to compute a new topology from a old one (the " drastic changes " evocated in page 3).

- We claim that " by changing the underlying topology, one changes the computational model". This claim must be supported by developping the topologies needed to describe the λ-calculus, first-order dataflow, petrinets, etc.

- Given some topology, it must be possible to defines new ones by standard constructions: several *products* are possible for instance. Products are very interesting because they enable the (more or less) orthogonnal description of several (more or less) independant view points of a system.

  The *quotient* constructions are of particular interest. As a matter of fact, these constructions can be a basis to describe a system at several scale.

  Nesting is another possible approach of this problem and must be studied to enable the uniform description of the depth of an organization.

- The composition of transformations and the building of composed transformations for composed topologies must be investigated. Currently, the transformations in MGS are applied on monoids where a rich algebra of functions exists. We musts study if this algebra can be defined in topological terms and then extended to others topologies.

- Several kinds of restrictions can be put and the transformations, leading to various kinds of pattern languages and rules. The complexity of matching such patterns has to be investigated.
- We also want to develop a type system that can handle nested collections, along the lines developed in [Ble93]. At last but not least, we want to known if the topology spaces build by transformations, can be characterized through a non standard type system.
- One very important question is the efficient implementation of MGS. One approach is to developp a (non standard) type system that can be further used to make the evaluation process more efficient or to guide the compilation. Here are some questions: can be the pattern expressions used to select a subcollection typed " by complexity of the matching"? Can we type a transformation with respect to the topology of the input argument and the output argument? etc.
- We must validate the adequation of the MGS concepts to some real application. Two of them are particularly motivating: the simulation of the topological changes at the early development of the embryo (see section 1.5 and 6.6) and the case of the Golgi formation (see section 1.6). These two applications are very challenging and require complex topologies going far beyond monolayer flat collection. These applications are also very attractive because their potential importance for biologists.
- One of the motivation behind the MGS project is to develop a *domain-specific language* (DSL) dedicated to the simulation of biological systems with a dynamical structure.

  DSLs are programming languages for solving problems in a particular domain. To this end, a DSL provides abstractions and notations for the domain at hand. DSLs are usually small, and more declarative than imperative. Moreover, DSLs are more attractive for programming in the dedicated domain than general-purpose languages because of easier programming, systematic reuse, better productivity, reliability, maintainability, and flexibility. MGS must be validated on these software engineering goals. Problems like: module systems for reusing simulation parts and capitalizing MGS code, dedicated semantic framework to validate MGS programs, observation and test theory of MGS programs, etc., are long term research goals.

Acknowledgments

The authors would like to thanks the members of the " Simulation and Epigenesis " workgroup at Genopole for stimulating discussions and biological motivations. We are indebt to Francois Letierce for the development of the 3D graphic viewer imoview. We are also very grateful to Franck Delaplace and Julien Cohen for their numerous questions, warm encouragements and the constant providing of sweet cookies.

This research takes place in the SPECIF team of the LaMI umr 8042 CNRS, in University of Evry Val d'Essone, and is supported in part by the CNRS, the GDR ALP, the GDR IMPG and the Genopole/Evry.

A An MGS Grammar

We give in this section the grammar used in some MGS interpreter (the C++ version at the date of april 2001). We give the grammar in a yacc-like form because it would give some ideas of the constructions that can be formulated in MGS (see for instance the pattern sub-language). Note however that this is only a first prototype and the user can expect drastic changes in the near futur. Already now, the functional constructs available in the ocaml version of the MGS interpreter are richer thant those available in the C++ version.

```
// Operator precedence: from the weakest to the strongest binding. See the YACC documentation

%right B_SEMI_COLON           // virtual token to indicate a priority less than semi-column
%right SEMI_COLON             //   ;
%right A_SEMI_COLON           // idem but greater
%right EQUAL                  //   =
%right LET                    //   :=
%right DOT_LAMBDA_MARK        // virtual token for priority
%right ASSERTION_MARK         //   !!
%left  AS                     //   as
%right MAP FOLD               //   map, fold
%left  EQ NEQ                 //   ==, != or ~=
%right B_COMMA                // virtual token for priority
%right COMMA3                 // various sort of comma operators
%right COMMA2                 //
%right COMMA                  // the comma ‘‘,’’
%right A_COMMA                // virtual token for priority
%left  INF2                   //   <<
%left  OR AND                 //   | or ||,  & or &&
%left  LT LE GT GE            //   <  <=  >  >=
%left  MIN MAX                //   min max
%left  COLUMN                 //   :
%right COLUMN2                //   ::
%right APPEND                 //   @
%left  PLUS MINUS             //   +, -
%left  TIME DIV MOD           //   *, /, %
%nonassoc NOT                 //   ~
%right TL                     //   tl
%nonassoc HD EMPTY            // hd, empty
%left  SIGN_OP                // virtual token for the sign of a number
%left  LEFT RIGHT             //   left, right
%nonassoc  LBRACKET RBRACKET  //   {, }
%nonassoc  LPAREN RPAREN      //   (, )
%nonassoc  LCROCHET RCROCHET  //   [, ]
%left  DOT                    //   .

// --- SENTENCES -----------------------------------------------------

top_level:  /* nothing */ | input ;

input: def
     | TERMINATOR
     | input TERMINATOR
     | input def ;

def: type_declaration TERMINATOR
   | command TERMINATOR
   | exp TERMINATOR ;

type_declaration: collection | arrow_spec | state ;

command: ...  ;
```

```
// --- COLLECTION -----------------------------------------

collection: COLLECTION user_id EQUAL id ;

// --- STATE ----------------------------------------------------

state: STATE optional_id  EQUAL state_body ;
state_body: id | state_body PLUS state_body | state_enumeration ;
state_enumeration: LBRACKET sid_list RBRACKET ;
sid_list:  /* nothing */ |  field_def |  sid_list COMMA field_def ;
field_def:  id |  NOT id |  id EQUAL exp %prec A_COMMA ;

// --- ARROW ----------------------------------------------------

arrow_spec: ARROW_SPEC ident_arrow EQUAL arrow_body ; // definition of arrow names
arrow_body: arrow_sep_begin  blist  arrow_sep_end ;   // and arrow kinds
arrow_sep_begin: ARROW_BEGIN  | LBRACKET ;            // not presented here
arrow_sep_end: ARROW_END  | RBRACKET ;

// --- EXPRESSION -----------------------------------------------------

exp: LPAREN exp RPAREN
   | ASSERTION_MARK exp
   | exp SEMI_COLON exp   // expression sequencing
   | exp COMMA exp        // Join of collections:
   | exp COMMA2 exp       // different kind of neighborhood
   | exp COMMA3 exp       // in the building of a collection
   | HD exp         // head
   | TL exp         // tail of a collection
   | EMPTY exp      // empty predicate
   | exp INF2 exp   // Scalar (integers, string, bool, ....) and collection Arithmetics
   | exp PLUS exp
   | exp TIME exp
   | exp DIV  exp
   | exp MOD  exp
   | exp MINUS exp
   | exp LE exp
   | exp LT exp
   | exp GE exp
   | exp GT exp
   | exp EQ exp
   | exp NEQ exp
   | exp AND exp
   | exp OR  exp
   | NOT exp
   | exp CONS exp
   | exp APPEND exp
   | MIN LPAREN exp COMMA exp RPAREN
   | MAX LPAREN exp COMMA exp RPAREN
   | IF exp THEN exp ELSE exp ENDIF
   | exp LPAREN exp_list RPAREN                          // Function application
   | exp LCROCHET integer RCROCHET LPAREN exp_list RPAREN // apply with optional arguments
   | exp LCROCHET TIME RCROCHET LPAREN exp_list RPAREN
   | exp LCROCHET blist RCROCHET LPAREN exp_list RPAREN
   | MAP LCROCHET exp RCROCHET exp                       // abbreviations for map and fold
   | FOLD LCROCHET fold_bin COMMA exp RCROCHET exp
   | id LET exp                        // assigning an imperative local variable
   | id EQUAL exp                      // binding (constantly) a value to a variable
   | exp DOT id                        // accessing the neighborhood
   | LEFT id
   | RIGHT id
   | id
   | fun_exp                           // constants of various type...
   | transformation
   | record
   | integer
   | real
   | STRING
   | UNDEF
   | LPAREN RPAREN COLUMN id            // empty collections
   | id COLUMN LPAREN RPAREN ;

exp_list: /* nothing */ | exp %prec A_COMMA | exp_list COMMA exp %prec A_COMMA ;

record: LBRACKET blist RBRACKET ;
blist: /* nothing */
     | id EQUAL exp %prec A_COMMA
     | id  %prec A_COMMA
     | blist COMMA id EQUAL exp  %prec A_COMMA
     | blist COMMA id  %prec A_COMMA ;

fold_bin: fun_exp | MAX  | MIN  | PLUS  | TIME  | AND  | OR ;

integer: INT | MINUS INT %prec SIGN_OP | PLUS INT  %prec SIGN_OP ;
real: REAL | MINUS REAL %prec SIGN_OP | PLUS REAL  %prec SIGN_OP ;

// --- FUNCTION -------------------------------------------------

fun_exp: FUN optional_id optional_arg LPAREN arg_list RPAREN EQUAL exp
       | LAMBDA LPAREN arg_list RPAREN DOT exp %prec DOT_LAMBDA_MARK
```

```
       | LAMBDA optional_arg arg_list DOT exp  %prec DOT_LAMBDA_MARK
       | FUN error TSEP
       | LAMBDA error TSEP ;

arg_list:  /* nothing */ |  id  |  arg_list COMMA id ;
optional_arg: /* nothing */ | LCROCHET blist RCROCHET ;
TSEP: SEMI_COLON | TERMINATOR ;

// --- TRANSFORMATION --------------------------------------------------

transformation: TRANSFORM transbody
              | TRANSFORM user_id optional_arg EQUAL transbody ;

transbody: LBRACKET rule_list OPT_SC RBRACKET ;

rule_list: rule
         | transformation    // nesting of transformations: not presented here
         | rule_list SEMI_COLON rule
         | rule_list SEMI_COLON transformation ;

OPT_SC: /* nothing */ | SEMI_COLON ;

// --- IDENTIFIER ------------------------------------------------------

id: ID  | QID /* quoted id */ ;
user_id: ID ;
optional_id: /* nothing */ | user_id  ;
ident_arrow: IDENT_ARROW ;
```

```
// --- RULES -----------------------------------------------------------

rule: pattern arrow a_exp
    | user_id EQUAL pattern arrow a_exp ;

a_exp: exp  %prec A_SEMI_COLON
     | id COLUMN exp  %prec A_SEMI_COLON ; // abstraction rule not presented here

arrow: ARROW            // arrows can be abstract or qualified, not presented here
     | PLUS_ARROW
     | ABSTRACT_ARROW
     | PLUS_ABSTRACT_ARROW
     | ident_arrow
     | arrow_body ;

pattern: user_id                // naming
       | user_id COLUMN id      // guard
       | LPAREN pattern RPAREN  // precedence
       | pattern DIV exp        // guard
       | pattern TIME           // iteration
       | pattern PLUS
       | pattern AS user_id     // naming
       | pattern COMMA pattern  // neigbhborhood
       | filter_state ;         // record pattern

filter_state: LBRACKET fid_list RBRACKET ;
fid_list: /* nothing */
        | id
        | id AS id
        | NOT id
        | id EQUAL id
        | fid_list COMMA id
        | fid_list COMMA id AS id
        | fid_list COMMA NOT id
        | fid_list COMMA id EQUAL id ;
```

B Full Code of the Turing+Morphogenesis Example

We give here verbatim the code used to produce the figure 13. This complete code uses the output facilities of MGS to write a file, called `tmp.turing.m` which contain `theomview` orders. The language `theomview` is used to describe graphical 3D scene composed of objects with automatic placement facilities. We give here an extract of the produced file:

```
Scaled{ Scale <0.1, 0.1, 0.1>
        Geometry Grid1{ Axis<1,0,0> GridList[

           Grid1{ Axis <0,0,1>  GridList [
                      Box { Size <1, 4, 16> },
                      Box { Size <1, 4, 16> },
                      Box { Size <1, 4, 16> },
                      ...
                      Box { Size <1, 4, 16> },
                      Box { Size <1, 4, 16> },
                      Box { Size <1, 4, 16> }]
           },
           Grid1{ Axis <0,0,1>  GridList [
                      Box { Size <1, 3.71071, 16> },
                      Box { Size <1, 3.94391, 16> },
                      Box { Size <1, 3.9798, 16> },
                      ...
                      Box { Size <1, 3.87611, 16> },
                      Box { Size <1, 3.90443, 16> },
                      Box { Size <1, 3.65924, 16> }] },
        ...
        ]}
}
```

The order `Box` is used to draw a cube. The order `Grid1` is used to automatically align the element of the list `GridList`, following an axis. In figure 13, the scene is rendered using a framewire mode, but one may chose interactively another rendering mode under graphical viewer.

To understand the composition of the file `tmp.turing.m` within the MGS program, one must known that:

- `"file" <<` exp write in the file *file* the value of expression exp.
- the primitive function `print_coll` takes 6 arguments: `print_coll(`$file$, col, f, $s1$, $s2$, $s3$`)`:
  - $file$ is an expression that evaluates to the name of the file where to save the collection;
  - col is the collection to be saved;
  - f is a function that is applied to each element of the collection col: it is the value returned by f that is written in the file $file$;
  - the value of expression $s1$ is written at the very beginning;
  - the value of $s2$ is written between two elements of the collection;
  - the value of $s3$ is written at the very end;
- The primitive function `close` is used to free any resources used to write in a file.
- The function `system` is used to start a shell command from the MGS interpreter. `imoview` is the name of the *theomview* viewer.

Here is the complete MGS code. It begins with a transformation used to produce the initial sequence of cells, cf. section 4.10.

```
trans init =
{
    x => { a = 3.5 +  random(1.0) -0.5, // 4.0,
           b = 4.0,
           beta = 12.0 + random(0.05 * 2.0) - 0.05,
           size = 16 };
};;
rsp := 1.0/16.0;;
diff1 := 0.25;;
diff2 := 0.0625;;
NbCell := 18;;
tore0 := init[1](iota(NbCell, ():seq));;
```

The transformation Turing is the core of the computation. It makes use of the auxilliary evolution function da and db.

```
fun da(a, b, la, ra) = rsp * (16.0 - a * b) + diff1*(la + ra - 2.0*a);;
fun db(a, b, beta, lb, rb) = rsp*(a*b - b - beta) + diff2*(lb + rb - 2.0*b);;

trans Turing =
{
    (x / x.b > 8)
    => { a = x.a/2, b = x.b/2, beta = x.beta, size = x.size/2},
       { a = x.a/2, b = x.b/2, beta = x.beta, size = x.size - x.size/2};

    (x / (left x) & (right x))
    +=> { a = x.a + da(x.a, x.b, (left x).a, (right x).a),
          b = Max(0.0, x.b + db(x.a, x.b, x.beta, (left x).b, (right x).b))
        };

    (x / ~(left x))
    +=> { a = x.a + da(x.a, x.b, 0, (right x).a),
          b = Max(0.0, x.b + db(x.a, x.b, x.beta, 0, (right x).b))
        };

    (x / ~(right x))
    +=> { a = x.a + da(x.a, x.b, (left x).a, 0),
          b = Max(0.0, x.b + db(x.a, x.b, x.beta, (left x).b, 0))
        };
};;
```

The transformation Turing is wrapped in several functions used to output the results:

```
fun minimal(x) = if (x <= 0) then 0.1 else x fi;;

fun showX(x) = "Box { Size <1, "
                    + minimal(x.b) + ", "
                    + minimal(x.size) + "> }";

fun showBarre(barre, t, tmax) =
(
    print_coll("tmp.turing.m",
               barre, showX,
               ("Grid1{ Axis <0,0,1>  GridList [\n"),
               ",\n\t ",
```

```
              "] }");
   if (t ~= tmax)
   then ("tmp.turing.m" << ",\n\n")
   else ("tmp.turing.m" << "\n\n") fi
);

fun pre_show() =
(
    "tmp.turing.m" << "Scaled{ Scale <0.1, 0.1, 0.1>\n"
                   << " Geometry Grid1{ Axis<1,0,0> GridList[\n\n"
);;

fun post_show(n, c) =
(
    "tmp.turing.m" << "] }}\n";
    close("tmp.turing.m");
    system("imoview tmp.turing.m")
);;

fun evol(barre, t, tmax) =
(
         showBarre(barre, t, tmax);
         if (t < tmax)
         then evol(Turing[iter=1](barre), t+1, tmax)
         else barre fi
);

fun evolve(n) = (pre_show(); evol(tore0, 0, n); post_show(n, NbCell));;

evolve(180);;  // run the evolution of the sequence of cells for 180 time steps

!quit;;
```

C Review of Some Notions Related to the Group Structure

The Group Structure. A group $(G,+)$ is a set G with a binary operation $+$ taking two elements of G into a third denoted by $a+b$. The operation is required to satisfy the following conditions:

- *Associativity*: $a+(b+c)=(a+b)+c$;
- *Existence of zero*: there exists an element $0 \in G$ such that $a+0=0+a=a$ for every a;
- *Existence of negative*: for any a there exists an element $(-a)$ such that $a+(-a)=0$.

If g is an element of a group G and n is an integer, then ng denotes the n-fold sum $g+...+g$ (the element g added n times) and $(-n)g$ denotes $n(-g)$.

If each $g \in G$ can be written as a finite sum $g=\sum n_\alpha g_\alpha$ where the g_α belong to a set S, we say that the set S *generates* G. If the set S is finite, then we say that G is *finitely generated* by S.

Group Homomorphisms. Let $(G,+_G)$ and $(H,+_H)$ be two groups. Then a function $f: G \to H$ is a *homomorphism* iff $f(a+_G b)=f(a)+_H f(b)$ for every a and b. The set of homomorphisms between G and H is denoted by $\mathrm{Hom}(G,H)$.

If f is a bijection then we say that f is an isomorphism and that the group G and H are *isomorphic*.

Let f and g two elements of $\mathrm{Hom}(G,H)$. Then we may define the function $(f+_{\mathrm{Hom}(G,H)} g): G \to H$ by $(f+_{\mathrm{Hom}(G,H)} g)(x)=f(x)+_H g(x)$. It is easy to check that $(f+_{\mathrm{Hom}(G,H)} g)$ is an homomorphism. It is also easy to check that $\mathrm{Hom}(G,H)$ together with $+_{\mathrm{Hom}(G,H)}$ is a group.

Direct Product of Groups. Of all the methods of constructing groups, we mention here the simplest. Let $(G_x)_{x\in X}$ be a family of groups indexed by indices in a set X. The set X can be finite or not. The *direct product* $\prod_{x\in X} G_x$ is the group H whose underlying set is the cartesian product of the sets G_x and whose group operation is the component-wise addition.

External Direct Sum of Groups. If $X=\{1,2\}$, then we write simply $G_1 \times G_2$ instead of $\prod_{i\in\{1,2\}} G_i$. If e_1 and e_2 are the neutral elements of G_1 and G_2, then the maps $g_1 \mapsto (g_1,e_2)$ and $g_2 \mapsto (e_1,g_2)$ are isomorphisms of G_1 and G_2 with subgroups of $G_1 \times G_2$. We suppose now that the groups G_1 and G_2 are distinguished. Usually the elements of G_1 and G_2 are then identified with their images under these isomorphisms, that is g_1 is written for (g_1,e_2) and g_2 is written for (e_1,g_2). Then G_1 and G_2 can be considered as subgroups of $H=G_1 \times G_2$.

In H we have $g_1+g_2=g_2+g_1$ if $g_i \in G_i$, $i=1,2$. Any element $h \in H$ can be written $h=g_1+g_2$ with $g_i \in G_i$: we say that the subgroups G_1 and G_2 generates H or that H is the (internal) *sum* of subgroups G_1 and G_2 of H. The subgroup's intersection $G_1 \cap G_2$ is equal to $\{0\}$ (the subgroups G_i are distinguished), and then, the sum $h=g_1+g_2$ uniquely define the g_i in G_i: we say that the sum is *direct*.

This explain why the direct product $G_1 \times G_2$ of the distinguished groups G_1 and G_2 is also written $G_1 \oplus G_2$ and called the *external direct sum* of G_1 and G_2.

The notion of external direct sum can be extended to an arbitrary product, but with a slight constraint. Let $H=\prod_{x\in X} G_x$ the direct product of the groups G_x. The *external direct sum* of the groups G_x is the subgroup G of the direct product H consisting of all tuples $(g_x)_{x\in X}$ such that $g_x=0_{G_x}$ for all but finitely many values of x (here 0_{G_x} is the zero element of G_x). The subgroup G is also called the *weak direct product* or the *weak direct sum* and is written: $\bigoplus_{x\in X} G_x$.

Abelian Groups. If $a+b=b+a$ for every elements a and b, then the group is said *abelian* or commutative. The abelian group G is *free* if it exists a set of generators S such that each $g \in G$ can be written as a *unique* finite sum. Then we say that the set of generators S is a *basis*. For example, the integers with the usual addition is a free abelian group denoted by $\mathbb{Z}$ and the basis is the singleton $\{1\}$.

If G and H are abelians, then $\mathrm{Hom}(G,H)$ is an abelian group too. The direct product $\prod_{x\in X} G_x$ of abelian groups G_x is abelian. And if all the G_x are free, then their direct product is also free. Furthermore, the direct product corresponds to the direct sum of modules, see below.

Let a function h defined on the basis $S = \{g_\alpha\}$ of a group G and with value in a group H, i.e. $h : S \to H$. Then, h can be extended to an homomorphism $\overline{h} : G \to H$ uniquely defined by $\overline{h}(\sum n_\alpha g_\alpha) = \sum n_\alpha h(g_\alpha)$. The functions h and $\overline{h}$ coincide on S. Usually we make no notational difference between h and its linear extension $\overline{h}$ and use h in both case, relying on the context to make clear which of these function is intended.

Ring and Modules. A *ring* R is an abelian group, written additively, with a multiplication operation satisfying two axioms:

- *Associativity*: $r.(s.t) = (r.s).t$
- *Distributivity*: $r(s+t) = r.s + r.t$ and $(r+s).t = r.t + s.t$.

If there is an element 1 in R such that $r.1 = 1.r = r$ for all r, then 1 is called a *unity element* in R. A ring is commutative if $r.s = s.r$ for all r and s. The only ring we consider is the ring of integers $(\mathbb{Z}, +, .)$.

An abelian group A has the structure of *module over a commutative ring* R with unity element 1, or more simply is a R-module, if there is a binary operation $R \times A \to A$, called the *scalar multiplication*, such that for $r, s \in R$ and $a, b \in A$, we have:

- $r(a+b) = ra + rb$
- $(a+b)r = ra + rb$
- $r(sa) = (r.s)a$
- $1a = a$

Homomorphism of Module, Dual of a Module. If A and B are R-modules, a *module homomorphism* is a group homomorphism $\varphi : A \to B$ such that $\varphi(ra) = r\varphi(a)$ for $r \in R$ and $a \in A$. The set of module homomorphisms is denoted by $\mathrm{Hom}_R(A, B)$.

If A is a module over a ring R then the set $\widetilde{A}$ of all homomorphism $\mathrm{Hom}_R(A, R)$ of A to R is a R-module, if we define operations by

$$\begin{aligned} (f+g)(a) &= f(a) + g(a) \qquad \text{for } f, g \in \widetilde{A} \text{ and } a \in A \\ (rf)(a) &= r\, f(a) \qquad \text{for } f \in \widetilde{A},\ a \in A \text{ and } r \in R \end{aligned}$$

This module is called the *dual* module of A.

Direct Sum of Modules. Let M and N two modules over a ring R. Consider the module consisting of pairs (m, n) for $m \in M, n \in N$, with addition and multiplication by elements of R given by

$$(m, n) + (m', n') = (m + m', n + n') \qquad \text{and} \qquad r(m, n) = (rm, rn)$$

This module is called the *direct sum* of M and N and is denoted by $M \oplus N$. The direct sum of any number of modules can be defined in the same way. The sum of n copies of the module M is denoted by M^n and is called the *free module of rank* n. This is the most direct generalization of a n-dimensional vector space.

$\mathbb{Z}$-modules. Any abelian groups can be considered as $\mathbb{Z}$-modules, by defining ng as the n-fold sum $g + \cdots + g$. The direct sum of these $\mathbb{Z}$-modules coincides with the direct product of the groups and $\mathrm{Hom}_{\mathbb{Z}}(A, B) = \mathrm{Hom}(A, B)$.

A *one dimensional* $\mathbb{Z}$-module is an abelian group denoted by $(\mathbb{Z}/n, +)$ where $n \in \mathbb{N}$. Formally, this group is the quotient of $\mathbb{Z}$ by the subgroup $n\mathbb{Z}$ of the multiples of n. Intuitively, this quotient group has n elements and each element is a subset of $\mathbb{Z}$. These sets form a partition of $\mathbb{Z}$. These sets are named by one of their members: the element $\overline{k} \in \mathbb{Z}/n$ denotes the set $\{\ldots, k-n, k, k+n, k+2n, \ldots\}$. The addition law is compatible with the addition on $\mathbb{Z}$: $\overline{p} + \overline{q} = \overline{p+q}$.

The module $\mathbb{Z}/0$ is isomorphic to $\mathbb{Z}$ and said to be a *free module*. The other modules $\mathbb{Z}/n$, $n \neq 0$, are called *torsion modules*.

The previous $\mathbb{Z}$-modules are of dimension 1 (they are generated by only one generator). $\mathbb{Z}$-modules of dimension greater than one are direct products of 1-dimensional $\mathbb{Z}$-modules.

The Fundamental Theorem of Finitely Generated Abelian Groups. The fundamental theorem of abelian groups says that every finitely generated abelian group G is isomorphic to:

$$G \simeq \mathbb{Z}^n \times \mathbb{Z}/t_1 \times \mathbb{Z}/t_2 \times \ldots \times \mathbb{Z}/t_q$$

where t_i divides t_{i+1} (see any standard text on groups; for a computer oriented handling cf. [Coh93]). This theorem shows that the study of abelian groups splits naturally into, on one hand the study of free $\mathbb{Z}$-modules of finite rank (i.e. $\mathbb{Z}^n$), and on the other hand the study of finite $\mathbb{Z}$-modules.

This isomorphism gives in some sense a "canonical representation" for G. The coefficient t_i of an abelian group defined by its generators and the relations between them (the finite presentation of the group) can be computed using the *Smith Normal Form* of the group presentation [Smi66]. The references [KB79, CC82, Ili89, HHR93] give a lot of considerations about the implementation, the complexity of the normalization algorithm and its optimizations.

There is another such canonical form, derived as follows. If m and n are relatively prime positive integers, then $\mathbb{Z}/m \times \mathbb{Z}/n$ is isomorphic to $\mathbb{Z}/mn$. It follows that any finite cyclic group can be written as the product of cyclic groups whose orders are powers of primes. Then

$$G \simeq \mathbb{Z}^n \times \mathbb{Z}/a_1 \times \mathbb{Z}/a_2 \times \ldots \times \mathbb{Z}/a_r$$

where each a_i is a power of a prime.

The Case of the Free Abelian Groups. In the case of a free abelian group, the torsion modules collapse to the trivial group and then a finitely generated abelian group with n generators is simply isomorphic to $\mathbb{Z}^n$. In other words, an element of a free abelian group with a basis of size n, can be represented by a n-uple of integers.

References

[Ale82] P. Alexandroff. *Elementary concepts of topology*. Dover publications, New-York, 1982.

[Axe98] Ulrike Axen. *Topological Analysis using Morse theory and auditory display*. PhD thesis, University of Illinois at Urbana Champaign, 1998.

[BB89] Gerard Berry and G. Boudol. The chemical abstract machine. Technical Report RR-1133, Inria, Institut National de Recherche en Informatique et en Automatique, 1989.

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

[Ber00] Guntram Berti. *Generic Software Components for Scientific Computing*. PhD thesis, Fakultät für Mathematik, Naturwissenschaften und Informatik der Brandenburgischen Technischen Universität Gottbus, 2000.

[BH00] Ronald Brown and Anne Heyworth. Using rewriting systems to compute left kan extensions and induced actions of categories. *Journal of Symbolic Computation*, 29(1):5–31, January 2000.

[BL74] J. Bard and I. Lauder. How well does turing's theory of morphogenesis work ? *Journal of Theoretical Biology*, 45:501–531, 1974.

[Ble93] Guy Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[BM86] J. P. Banatre and Daniel Le Metayer. 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.

[CC82] Tsu-Wu J. Chou and George E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM Journal on Computing*, 11(4):687–708, November 1982.

[Coh93] H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Text in Mathematics*. Springer, 1993.

[CS00] Jeffrey Chard and Vadim Shapiro. A multivector datastructure for differential forms and equation. *Mathematics and Computers in Simulation*, (54):33–64, 2000.

[FB94] W. Fontana and L. Buss. "the arrival of the fittest": Toward a theory of biological organization. *Bulletin of Mathematical Biology*, 1994.

[FB96] W. Fontana and L. Buss. *Boundaries and Barriers, Casti, J. and Karlqvist, A. edts,*, chapter The barrier of objects: from dynamical systems to bounded organizations, pages 56–116. Addison-Wesley, 1996.

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

[Fon92] Walter Fontana. Algorithmic chemistry. In Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Proceedings of the Workshop on Artificial Life (ALIFE '90)*, volume 5 of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 159–210, Redwood City, CA, USA, February 1992. Addison-Wesley.

[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.

[GM01] 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.

[GMS96] J.-L. Giavitto, O. Michel, and J. Sansonnet. Group-based fields. In *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068, pages 209–215, 1996.

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

[HHR93] George Havas, Derek F. Holt, and Sarah Rees. Recognizing badly presented Z-modules. *Linear Algebra Appl.*, 192:137–163, 1993.

[HP96] Mark Hammel and Przemyslaw Prusinkiewicz. Visualization of developmental processes by extrusion in space-time. In Wayne A. Davis and Richard Bartels, editors, *Graphics Interface '96*, pages 246–258. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 1996. ISBN 0-9695338-5-3.

[HY88] J. G. Hocking and G.S. Young. *Topology*. Dover publications, New-York, 1988.

[Ili89] C. S. Iliopoulos. Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the hermite and smith normal forms of an integer matrix. *SIAM Journal on Computing*, 18(4):658–669, August 1989.

[KB79] Ravindran Kannan and Achim Bachem. Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. *SIAM Journal on Computing*, 8(4):499–507, November 1979.

[Lew97] Benjamin Lewin. *Genes (VI)*. Oxford University Press, 1997. 6th. edition.

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

[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 Press, June 1993.

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

[Mic96] O. Michel. Design and implementation of 81/2, a declarative data-parallel language. *Computer Languages*, 22(2/3):165–179, 1996. special issue on Parallel Logic Programming.

[Mun84] James Munkres. *Elements of Algebraic Topology*. Addison-Wesley, 1984.

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga publishing company, 1980.

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

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

[PS93] Richard S. Palmer and Vadim Shapiro. Chain models of physical behavior for engineering analysis and design. *Research in Engineering Design*, 5:161–184, 1993. Springer International.

[Rém92] Didier Rémy. Syntactic theories and the algebra of record terms. Technical Report 1869, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

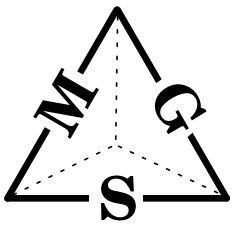
[Sha90] Igor' Shafarevich. *Basic Notions of Algebra*. Springer, 1990.

[Smi66] D. Smith. A basis algorithm for finitely generated abelian groups. *Math. Algorithms*, 1(1):13–26, January 1966.

[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.

[Ton76] Enzo Tonti. The reason for analogies between physical theories. *Appl. Math. Modelling*, 1:37–50, June 1976.

[VN66] J. Von Neumann. *Theory of Self-Reproducing Automata*. Univ. of Illinois Press, 1966.


M
G
S