Electronic Notes in Theoretical Computer Science 59 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 19 pages

MGS: a Rule-Based Programming Language for Complex Objects and Collections

Jean-Louis Giavitto [1], Olivier Michel [2]

LaMI u.m.r. 8042 du CNRS
Université d'Evry Val d'Essone
91025 Evry Cedex, France.

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 current state of the system. MGS proposes a unified view on several computational mechanisms initially inspired by biological or chemical processes (Gamma and the CHAM, Lindenmayer systems, Paun systems and cellular automata). The basic computation step in MGS replaces in a collection A of elements, some subcollection B, by another collection C. The collection C only depends on 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 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.

1 Motivations

1.1 Dynamical Systems and their State Structures

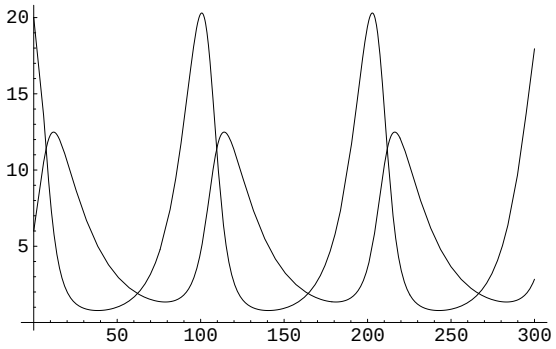
A *dynamical system* (or DS in short) corresponds to a phenomenon that evolves in time. The phenomenon is located on a *system* characterized by "observables". The observables are called the *variables* of the system, and are linked by some relations. The value of the variables evolves with the time. The set of the values of the variables that describe 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.

[1] Email: giavitto@lami.univ-evry.fr
[2] Email: michel@lami.univ-evry.fr

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

Trajectories $x(t)$ and $y(t)$

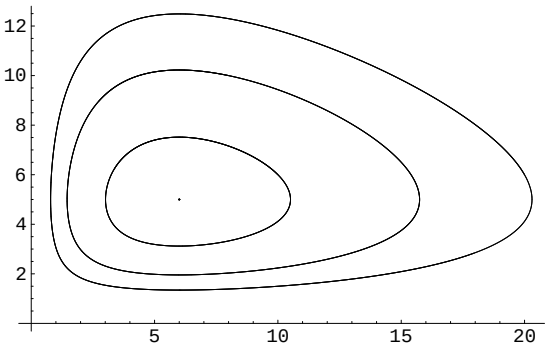


Evolution Constraints for x and y

$$\frac{dx}{dt} = Ax - Bxy$$
$$\frac{dy}{dt} = -Cy + Dxy$$

Solving the constraints gives the trajectory of $x(t)$ and $y(t)$ starting from some initial state. The evolution of a variable is periodic.

Evolution in the phase space (x, y)



The three curves correspond to the cyclic evolution of the system starting from three different initial conditions. A point in this plot corresponds to a state (x, y). A curve corresponds to the evolution $(x, y)(t)$. The periodicity of the trajectories of x and y gives a closed curve. There is a fourth curve reduced to a *fixed point*. The image by the evolution function of this point is itself. This point is characterized by $dx/dt = dy/dt = 0$ (no change).

Fig. 1. Example of the evolution of a predator-prey system (this DS has 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 $\mathbb{R}^2$.

can be very difficult to achieve because of the complexity of the description of the phase space and of the evolution function. However, the more we know about the phase space, the more we know about the DS. 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 is a common situation in biology [9,7,8]. Such DS can be found in the modeling of plant growing, in developmental biology, integrative cell models, protein transport and compartment simulation, etc. This accounts for the fact that the structure of the phase space must be computed jointly with the current state of the system. In this case, we say that the DS has a *dynamical structure*. The description of DS with a dynamical structure are 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.*

1.3 Unifying Several 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 in section 5 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.

2 The Basic Ideas

Our goal is to provide a general support for the notions of "organized set" and "local competing transformations" that can be used to describe uniformly the computation mechanisms of Γ, P and L systems and CA.

We call *collection* a set of elements with some "organization" (to be clarified later). Several kind of organizations are used in programming languages and give raise to several data structures: sets, multisets (or bags), sequences (or list), arrays, trees, terms, etc. The collection type underlying the computations in Γ, CHAM and P system is the multiset, L systems rely on sequences and CA on arrays.

2.1 A Unified Description of Γ, P and L system and CA

A Γ program, a P or a L system and a CA can be themselves viewed abstractly as a discrete dynamical system: a running program can be characterized by a state and the evolution of this state is specified through *evolution rules*. From this point of view, the following characteristics have to be stressed.

Discrete space and time. The structure of the state (the multiset in Γ, the membranes hierarchy in a P system, the word in a L system and the array in a CA) consists of a discrete *collection* of values. This discrete collection of values evolves in a sequence of discrete time steps.

Temporally local transformation. The computation of a new value in the new state depends only on values 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.

"Structural combination", means that the elementary results are combined into a new collection, irrespectively of their precise value. "Small and static subset" makes explicit that only a fixed subset of the initial elements are used to compute a new element value (this is measured for instance by the diameter of the evolution rule of a P systems, the local neighborhood of a CA, the number of variables in the right hand side of a Γ reaction or the context of a rule in a L system).

Considering these shared characteristics, the main difference between the four formalisms appears to be the organization of the collection. The abstract computational mechanism is always the same:

(i) a subcollection A is selected in a collection C;
(ii) a new subcollection B is computed from the collection A;
(iii) the collection B is substituted for A in C.

see Fig. 2. We call these three basic steps a ***transformation***. In addition to transformation specification, there is a need to account for the various

constraints in the selection of the subcollection A and the replacement B. This abstract view makes possible the unification in the same framework of various computational devices. The trick is just to change the organization of the underlying collection.

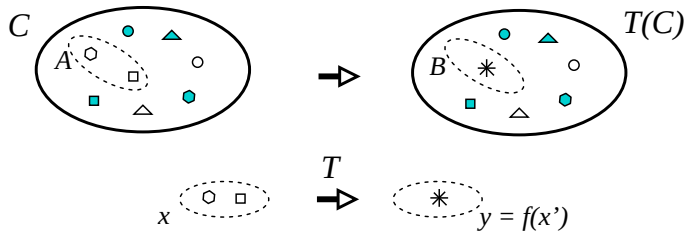


Fig. 2. The basic mechanism of the transformation of a collection. Collection C is of some kind. A rule T specifies that a subcollection A of C has to be substituted by a collection B computed from A. The right hand side of the rule is computed from the subcollection matched by the left hand side x and its possibles neighbors x' in the collection C.

Constraining the Subcollections

There is *a priori* no constraint in the case of Γ: one element or many elements are replaced by zero, one or many elements. In the case of P systems, the evolution of a membrane may affect only the immediate enclosing membrane (by expelling some tokens or by dissolution): there is a *localization* of the changes. This is also the case for L systems: the new collection B is inserted at the place of A and not spread out over C. For CA, the changes are not only localized, but also A and B are constrained to have the same shape: usually A is restricted to be just one cell in the array and B is also one cell to maintain the array structure.

2.2 Collections as Spaces

Considering these constraints and their expression, it is very natural to see a collection as a set of *places* or *positions* organized by 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***.

For instance, one may decide that neighbors of an element in a sequence are their two adjacent elements (except for the first and the last element in the sequence which have only one neighbor). The neighborhood can be specified by a relation denoted by ",". That is to say, x, y means that x is a neighbor of y. If S is a subset of the elements of the collection $\mathcal{C}$, then we say that S is *connected* if the quotient of S by the transitive closure of "," is reduced to only one element. 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. However, a subcollection is always a connected subset of the main collection.

This topological approach formalizing the notion of collection is part of a long term research effort [12] developed for instance in [13] where the focus is on the substructure and in [10] where a general tool for uniform neighborhood definition is developed. 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 5. So, one may ask if the machinery needed is worthwhile. Actually, more complex topologies are needed for some biological modeling applications [11]. And more importantly, the topological framework unify various situations. Our ultimate goal is to develop a generic implementation based on these notions, see [11].

Now, we come back to our initial goal of specifying the dynamical structure of a DS. 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 and 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, in the sense that an interaction between two objects is enabled simply by virtue of their both being present in the multiset. More organized topological collections are used for more sophisticated spatial organization.

2.3 The `MGS` Project and the Organization of the Rest of this Paper

We do not claim that topological collection are a useful theoretical framework encompassing all the previous formalisms. We advocate that few notions and a single syntax can be consistently used to allow the merging of these formalisms *for programming* purposes. This leads to the development of an experimental programming language called `MGS`. `MGS` is the acronym of "*(encore) un Modèle Géneral de Simulation (de système dynamique)*" (yet another General Model for the Simulation of dynamical systems). `MGS` is a vehicle used to investigate general notions of collections and transformations and to study their adequacy to the simulation of various biological processes.

The `MGS` language is presented informally in section 3 through some examples. We review first the notions of collections and then their transformations. Simple examples of `MGS` programs are given in section 4. All examples are processed using the current version of the `MGS` interpreter. Then, in section 5, we sketch how the previous formalisms can be emulated in `MGS`.

3 An MGS Quick Tour

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.

We give here informally the main constructs concerning collections, transformations and their applications. Elements of the MGS syntax are given through examples.

3.1 Collections

In addition to basic values like integers, floats, strings, lambda-expressions, etc., MGS handles records and several kinds of collections. The elements in a collection can be any kind of values: basic, 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 [5]. Collections are (sub-)typed. The tree in Fig. 3 gives the type hierarchy of collections.

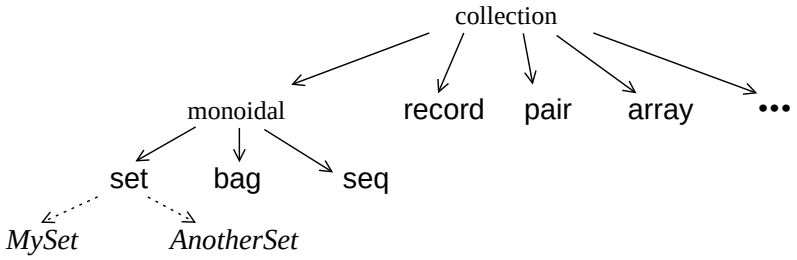


Fig. 3. The subtyping hierarchy of collection kinds. *MySet* and *AnotherSet* are user-defined collection types, Cf. below. The types `collection` and `monoidal` do not correspond to concrete data structures, but to predicates, Cf. below. Conceptually, a record is a set of pairs (*field-name, field-value*) but it is managed through dedicated operators.

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. The join operator with its properties induces the topology of the collection and the neighborhood relationship.

There is a large amount of generic operations available for all collection kinds, based on the function algebra developed for instance in [5]. We do not detail these features as they are not relevant for our purpose here. The table 1 gives the main construction operations for structural recursion.

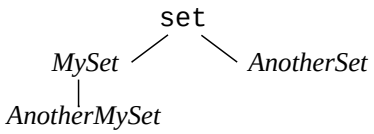
Table 1
Main constructions operations for monoidal collections. The line (*) gives an overloaded syntax (the type of the arguments is used for desambiguation).

	empty	addition	singleton	merge
Set	`set` : ()	`insert`	`single_set`(x)	`union`
Bag	`bag` : ()	`increment`	`single_bag`(x)	`sum`
Seq	`set` : ()	`::`	`single_seq`(x)	`@`
(*)		,		,

User-Defined Subtypes

Often there is a need to distinguish several collections of the same kind (e.g. several multisets nested in one other 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 as a label 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 elements 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 specify 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 3.3) or as a predicate to test if a value is of a given type. A monoidal collection type can also be 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 "*Myset* : ()" denotes the empty *MySet* and "," is the overloaded join operator: x, X creates a new collection with 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 A and B gives a collection with type C corresponding to the common ancestor of A and B (with the previous example, `set` is the common ancestor of *MySet* and *AnotherSet*). Other example, *MySet* is the common ancestor of *AnotherMySet*) and itself.

3.2 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 [18]).

For records, type declarations look like

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

(`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:

$$R(\{a = 2,\ x = 3\}) \qquad \text{or equivalently} \qquad \{a = 2,\ x = 3\} : R$$

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.

3.3 Pattern, Rule and Transformations

A transformation T is a set of rules:

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

When there is only one rule in the transformation, the enclosing braces can be dropped. A rule is a basic transformation taking the following form:

$$\mathit{pattern}\ \texttt{=>}\ \mathit{expression}$$

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.

3.3.1 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 patterns expression 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 evaluating 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.

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, but 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 anywhere in the rest of the rule. E.g., the pattern x, x matches two connected elements with the *same* value (each occurrence of x in a rule denotes the same value).

guard: p/exp matches the collections matched by 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, $x : \mathit{MySet}$ filters an element of type *MySet*. Another example: $y \,/\, y > 3$ matches an element y provided that $y > 3$ holds.

Here is a contrived example. Pattern

$$(x : \mathit{int}/x < 3)+ \ \texttt{as} \ \ S \quad / \quad (\texttt{card}(S) < 5) \quad \& \quad (\texttt{fold}[+](S) > 10)$$

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. a set, a multiset), S denotes a subsequence (resp. a subset, 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.

3.3.2 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 argument: that is, 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. 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. Inclusive rules are better explained by an example. Suppose we have to manipulate records having at least a field x and y. Then,

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

are two inclusive rules (because the arrow is $+=>$) matching respectively a record with at least a field x and a record with at least 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 $\{x = 2,\ y = 3,\ z = 0\}$ is $\{x = 3,\ y = 6,\ z = 0\}$. This result is computed as

$$\Big(\{x = 2,\ y = 3,\ z = 0\} + \{x = 2 + 1\}\Big) + \{y = 2 * 3\}$$
$$\text{or } \Big(\{x = 2,\ y = 3,\ z = 0\} + \{y = 2 * 3\}\Big) + \{x = 2 + 1\}$$

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.

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 a functional language with some imperative features. 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

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

specifies a rule R which is applied at most 5 times (within the evaluations triggered by one application of T). The semicolon 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 rule applications. The initial value of a variable local to a transformation can be overridden when the transformation is applied; for instance the evaluation of $T[a = 3](...)$ triggers at most 2 uses of rule R.

3.4 Managing the Applications of a Transformation

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

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

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

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 example are freely inspired by examples given for Γ, P systems and L systems.

Sorting a Sequence

A kind of bubble-sort is immediate:

$\texttt{trans}\ \mathit{Sort} = (x, y \,/\, y < x)\ \texttt{=>}\ y, x;$

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

Eratosthene's Sieve on a Set

The idea is to generate a set with integers from 2 to N (with rules *Generate* and *Succed*) and to replace an x and an y such that x divides y by x (rule *Eliminate*). The result is the set of the prime integers less than N.

$\texttt{trans}\ \mathit{Generate} = \{x, \mathit{true}\}\ \texttt{=>}\ x, \{x+1, \mathit{true}\};$
$\texttt{trans}\ \mathit{Succed} = \{x, \mathit{true}\}\ \texttt{=>}\ x;$
$\texttt{trans}\ \mathit{Eliminate} = (x, y \,/\, y\,\texttt{mod}\,x = 0)\ \texttt{=>}\ x;$

With these definition, the expression

$$\mathit{Eliminate}[\texttt{fixrule}]\Big(\mathit{Succed}\big(\mathit{Generate}[N](\{2, \mathit{true}\}, \texttt{set} : ())\big)\Big)$$

computes the primes up to N.

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 $\{\mathit{prime} = P_i\}$. This element can receive a candidate number n, which is represented by a record $\{\mathit{prime} = P_i, \mathit{candidate} = n\}$. 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.

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

We have given an explicit name to each rule. The expression

$$Erasto[N]((2, \texttt{seq} : ()))$$

executes N steps of the Erastothene's sieve. For instance $Erasto[100]((2, \texttt{seq} : ()))$ computes the sequence: 42, $\{candidate = 42,\ prime = 2\}$, $\{ok = 41,\ prime = 3\}$, $\{prime = 5\}$, $\{prime = 7\}$, $\{prime = 11\}$, $\{prime = 13\}$, $\{ok = 37,\ prime = 17\}$, $\{prime = 19\}$, $\{prime = 23\}$, $\{prime = 29\}$, $\{prime = 31\}$, $\texttt{seq} : ()$.

5 Comparison with Other Approaches

We want to show that Γ and the CHAM, P systems, L systems and cellular automata (CA) can been handled in MGS. Because they fit harmoniously, we gain confidence that the underlying concepts of topological collection may reveal unifying and covering a broad class of biological DS with a dynamical structure.

5.1 Sets and Multisets: The programming language Γ *and the CHAM*

The computational model underlying Γ [2,1] 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 a 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 happen 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 least 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 [3].

The Topology of Sets

A set V is organized such that each element is neighbor of any other elements in the set (with this definition, an element of V is connected with any other element).

A multiset M of elements $e \in E$ 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 set associated to $\hat{M}$ and any element in the multiset is neighbor of any other element.

With this representation, the application of one Γ rule on a multiset M is also the application of an `MGS` rule. The connection between any two multiset elements accounts the fact that any sub-multiset can be matched and replaced in a Γ rule.

5.2 Nesting of Multisets: P systems

P systems [17,16] are a new distributed parallel computing model based on the notion of a membrane structure. A membrane structure is a nesting of cells represented, e.g, by a Venn diagram without intersection and with a unique superset: the skin. Objects are placed in the regions defined by the membranes and evolve following various transformations: an object can evolve into another object, can pass trough a membrane or dissolve its enclosing membrane. As for 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 nesting of multisets and the locality of a computation step. In this approach, the region associated to a membrane would be a 2 dimensional object (surfaces) and the membranes would be 1 dimensional (curves).

A cruder and simpler 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* (opposed to flat) multiset rewriting. We can handle also this approach, because `MGS` values can be arbitrary combinations of other values.

5.3 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 system can be roughly described as a grammar with an axiom and a set of derivation rules. The productions are applied in parallel in a non deterministic manner. 0L systems are context-free grammars. D0L systems are deterministic context-free grammars: given a letter A there is at most one production rule that can be applied. Parametric L systems deal with *modules* instead of letters: a module is a letter associated with a list of parameters. The production rules are extended with side-conditions on the parameters. For example,

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

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

The topology of a sequence has been sketched in paragraph 2.2. It is the intuitive view of the sequence has a sequence of contiguous cells.

The application of only one production $a \rightarrow b$ of a D0L system is similar to the application of a simple `MGS` rule $(x/x == a) \ => \ b$ on a sequence.

5.4 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 systems in biology [19].

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 $\mathcal{V}$ 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 organization of the cells of an array is the natural one (Von Neuman or Moore neighborhood). A rule of a cellular automata is an MGS rule applying on only one cell. The conditions on the neighbor cells can be expressed using guards and the specific neighbors accessors.

6 Conclusion and Future Work

The technical report [11] gives more details on the topological formalization of collections and transformations and outlines several examples of MGS programs (the tokenization of a sequence of letters, the computation of the convex hull of a set of points in $\mathbb{R}^3$, the computation of the maximal segment sum, a Turing diffusion-reaction process, etc.).

Currently, it exists two versions of an MGS interpreter: one written in OCAML (a dialect of ML and one written in C++. There is 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 [3]. In this current MGS implementations, only 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. [12,10]) are planed in a short term. We also begin the study of a generic implementation of topological chain complex, a suitable formalization of our topological collection, using G-maps [14] to represent arbitrary join/neighborhood structure.

At the language level, the study of the topological collections concepts must continue with a finer study of transformation kinds. Several kinds of restriction can be put on the transformations, leading to various kind of pattern languages and rules. The complexity of matching such patterns has to be investigated. We also want to develop a type system that can handle nested collections, along the lines developed in [4]. At last but not least, we want to known if the topological spaces built by transformations, can be characterized through a non standard type system. The efficient compilation of a MGS program is a long-term research effort.

The applications opened by this preliminary work are numerous. From the applications point of view, we are targeted by the simulation of the topological changes at the early development of the embryo. This is an actual example of tissues formation and fusion requiring complex topology beyond what is accessible using simple data-structures. Another motivating application is the case of a spatially distributed biochemical interaction networks, for which some extension of rewriting have been advocated, see [6,15].

[3] see www.lami.univ-evry.fr/mgs.

Acknowledgments

The comments of the anonymous referees have greatly improved this paper. The authors would like to thanks the members of the "Simulation and Epigenesis" group at Genopole for stimulating discussions and biological motivations. They are also grateful to F. Delaplace and J. Cohen for many questions and encouragements. This research is supported in part by the CNRS, the GDR ALP, IMPG and Genopole/Evry.

References

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

[2] Banatre, J. P. and D. Le Metayer, *A new computational model and its discipline of programming*, Technical Report RR-0566, Inria, 1986.

[3] Berry, G., and Gérard Boudol, *The chemical abstract machine*, *Theoretical Computer Science*, 96:217–248, 1992.

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

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

[6] Fisher, M., G. Malcolm, and R. Paton, *Spatio-logical processes in intracellular signalling*, *BioSystems*, 55:83–92, 2000.

[7] Fontana, W., and L. Buss, *The Arrival of the Fittest": Toward a theory of biological organization*, *Bulletin of Mathematical Biology*, 1994.

[8] Fontana, W., 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.

[9] Fontana, W., *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.

[10] Giavitto, J.-L., 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.

[11] Giavitto, J.-L., and O. Michel, *MGS: a programming language for the transformations of topological collections*, Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001. 85p.

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

[13] Giavitto, J.-L., *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.

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

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

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

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

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

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