Pattern-matching and Rewriting Rules for Group Indexed Data Structures

Jean-Louis Giavitto giavitto@lami.univ-evry.fr
Olivier Michel michel@lami.univ-evry.fr
Julien Cohen jcohen@lami.univ-evry.fr

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

Abstract

In this paper, we present a new framework for the definition of various data structures (including trees and arrays) together with a generic language of filters enabling a rule-based programming style for functions. This framework is implemented in an experimental language called MGS. The underlying notions funding our framework have a topological nature and enable to extend the case-based definition of functions found in modern functional languages beyond algebraic data structures.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; E.1 [**Data Structures**]; F.1.1 [**Theory of Computation**]: Models of Computations; F.4.2 [**Formal Languages**]: Grammar and Other Rewriting Systems

Keywords

group-based data fields, group indexed data structure, path pattern, combinatorial matching, array pattern matching, Cayley graphs, rule based array function.

1 Introduction

One of the achievements and successes of the current functional languages is the ability to define functions by cases using filters and pattern-matching. However, this possibility is restricted to pattern-matching of algebraic data types, which is now well understood. An example of data structure beyond the current capability is for example the *array data type*: it is not possible to define a function by cases on arrays.

In this paper, we present a new framework for the definition of various data structures, including trees and arrays, together with a generic language of filters enabling a rule-based programming style for functions. This framework is implemented in an experimental language called MGS.

The underlying notions funding our framework have a topological nature and unify several programming paradigm like Gamma [BM86] and the CHAM [BB92], Lindenmayer systems [RS92], Paun systems [Pau99] and cellular automata [VN66]. Gamma, CHAM and Paun systems are based on multiset rewriting and Lindenmayer systems on string rewriting. These kind of data structures are qualified as *monoidal* [Man01, GM01b] and their rewriting theories are now mastered. In this paper, we focus on non-monoidal data structures and especially array-like data structures for which there is no clear agreement on a rule-based rewriting mechanism.

The rest of this paper is organized as follows. The next section introduces a motivating example. Section 3 details the notion of group indexed data structure or GBF (for *group-based data fields*). Such structures generalize the notion of array. We give a geometric interpretation of GBF in section 4. This interpretation underlies the design of a generic pattern language described in section 5. Some examples are worked out in section 6. The corresponding pattern-matching algorithm is developed section 7, before reviewing some related and future works.

2 A Motivating Example

This example is loosely inspired from lattice gas automata. In these kinds of cellular automata, rules of the form $\beta \Rightarrow f(\beta)$ are used to specify the local evolution of a set of particles distributed on a regular subdivision of the plan. The expression β is a pattern that matches a configuration (typically two particles in two neighbor cells that would collide at the next time step) and $f(\beta)$ is used to specify the evolution of the particles.

In our arbitrary example, we want to specify the 90°-rotation of a cross in a square lattice (see the two diagrams on the left side of figure 1). An array-like data structure can be used to record the lattice state and the rule $\beta \Rightarrow f(\beta)$ is used to specify the rotation of a single cross. Notice that in this case, the pattern β does not filter a sub-array but an arbitrary subset (a cross). This rule must be applied to each occurrence of a cross in the data structure. The result is an array function, called here a *transformation*. We write:

```
trans Turn = { β ⇒ f(β); }
```

The transformation *Turn* is defined by cases (here there is only one case corresponding to the single rule in the transformation *Turn*). The case β specifies a sub-domain

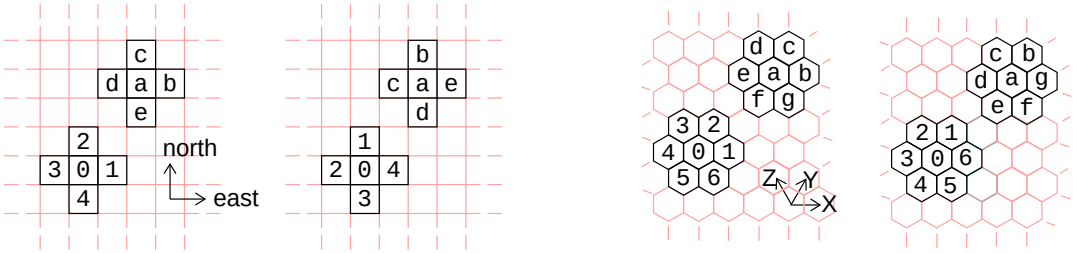


Figure 1. Application of the transformation *Turn* to an array on the left or to an hexagonal subdivision on the right. In contrast with cellular automata, the evolution concerns a multi-cell domain.

which is replaced by $f(\beta)$. However, unlike case-based function definitions acting on algebraic data types, the cases do not correspond to constructors nor exhaust the data structure.

A transformation is a function taking a *collection* as argument. A collection is an organized set of elements. The MGS language handles several kinds of collections including sets, bags, sequences and array-like data structures called GBF. A square lattice, as pictured on the left of figure 1 is a special case of GBF.

It is usual for physicists to work with an hexagonal lattice, because such a tiling of the plane respect more symmetries in the expression of fundamental physical laws than a square lattice. We can transpose our transformation in such a tiling, cf. the two diagrams on the right of figure 1. In this case, the pattern β involves a 7 cells sub-domain.

To turn the description of the transformation *Turn* into a real program, one must dispose of some new constructs in a language in order to

1. define the type of a data structure representing a 2D array (or better, some generalization like an hexagonal tiling),
2. define a pattern β that matches an arbitrary subdomain in an array,
3. specify a function using rules like $\beta \Rightarrow f(\beta)$ that specify the substitution of non-intersecting occurrences of subdomains matched by β by a replacement computed by $f(\beta)$.

Such devices are available in MGS, an experimental declarative language. One of the objectives of the MGS project is to investigate the use of a rule-based approach for the simulation of dynamical systems (this explains the choice of our examples). In [GM01c] we have shown how MGS unifies multiset and string based rewriting paradigms. In this paper, we extend further this unification towards array-like data structure. In section 3 we show how to describe such data structures. The problem of specifying a pattern β in this kind of data structure is examined in section 4 and 5.

3 Group Indexed Data Structures

In this section, we introduce the concept of GBF which generalizes the concept of array. These data structures admit a geometrical interpretation which is the basis of the language of filters presented in section 5. As a matter of fact, a collection type always admit a topological interpretation in terms of neighborhood (cf. [GM02a, GM02b]) and the notions introduced in section 5 are uniformly applicable to all collection types.

An $n \times m$ array A associates a well defined value to an index (i, j) for $1 \leq i \leq n$ and $1 \leq j \leq m$. Thus, an array can be seen abstractly as a *total function* from the set of indexes $I = [1,n] \times [1,m]$ to some set of values. The *data field approach* extends this notion by considering the array A as a *partial function with a finite support* from a larger set of indexes $I = \mathbb{Z} \times \mathbb{Z}$ (the *support* of a partial function is the subset of its domain for which the function takes a well defined value). This enables the representation of "arrays with holes", "triangular arrays", etc. The notion of data field appears in the development of recurrence equations and goes back at least to [KMW67]. The term itself seems to appear for the first time in [YC92, CiCL91] and its investigation in a functional and data parallel context has been mainly made by Lisper [Lis96] (see also [GDVS98]).

Our starting point to extend further the notion of data field, is the remark that the set of indexes I is provided with some operations. The standard example of index algebra is integer tuples with linear mappings. For instance, more than 99% of array references are affine functions of array indexes in scientific programs [GG95]. As a consequence, we have proposed to provide the set of indexes with a *group structure* [GMS96]. Such a data structure, a partial function with a finite support from a group to a set of values, is called a **GBF** for group-based data field. The basic example is the data fields themselves, where the group of indexes is the group $(\mathbb{Z}^n, +)$. The advantage of providing the set of indexes with a group structure and several examples of GBF are detailed in [GM01a].

GBF are introduced in the MGS language using a type declaration specifying the underlying group of indexes. The definition of the group is given using a finite presentation listing a set of generators g_i for the group and a set of equations $e_k = e'_k$ where the e_k are formal sums of the g_i :

```
gbf G = < g1, ..., gn;
          e1 = e'1, ..., ep = e'p >
```

A formal sum of the generators is simply a linear com-

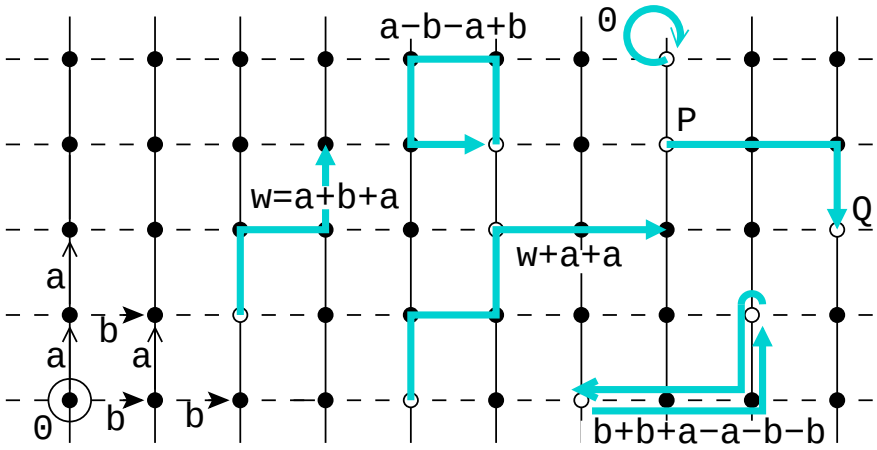


Figure 2. Graphical representation of the relationships between Cayley graphs and group theory. A vertex is a group element. An edge labeled a is a generator a of the group. A word (a formal sum of generators and of inverses of generators) is a path. Path composition corresponds to group addition. A closed path (a cycle) is a word equal to 0 (the identity of the group operation). An equation $v = w$ can be rewritten $v - w = 0$ and then corresponds to a cycle in the graph. There are two kinds of cycles in the graph: the cycles that are present in all Cayley graphs and corresponding to group laws (intuitively: a backtracking path like $b+a-a-b$) and closed paths specific to the own group equations (e.g.: $a-b-a+b$ for Abelian groups). The graph connectivity, i.e. there is always a path going from P to Q, is equivalent to say that there is always a solution to the equation $P+x=Q$.

bination as for example:

```
3g1 + 2g3 - (5g4 + g5)
```

We use the following typographical conventions: if G is a GBF, we write **G** (a finite group presentation) for its type and $\mathcal{G}$ (the group of indexes of G) for its domain. Beware that a group admits various presentations, so a GBF type contains more information than just the group structure. The set of values of a GBF G is not mentioned in the type declaration for **G** because `MGS` is a dynamically typed language and heterogeneous values can be recorded in a GBF.

In this paper we deal only with Abelian groups and we use an additive notation for the group operation. By convention a finite presentation starting with "<" and ending with ">" introduces an Abelian group, that is: the set of equations is completed implicitly with the equations specifying the commutation of the generators $g_i + g_j = g_j + g_i$.

Examples of GBF Types

The two examples of figure 1 correspond to the two GBF types:

```
gbf G2 = < north, east >
gbf H2 = < X, Y, Z; X+Z=Y >
```

The type **H2** defines an hexagonal lattice that tiles the plane. This geometrical interpretation of the presentation relies on the notion of *Cayley graph*.

4 Group of Indexes and Topological Representation

A Cayley graph is a graph representation of the presentation **G** of a group $\mathcal{G}$: each vertex in the Cayley graph is an element of the group $\mathcal{G}$ and vertex x and y are linked if there is a generator `u` in the presentation **G** such that $x + \mathtt{u} = y$. See figure 2. This representation supports the following *topological interpretation* of a GBF:

- The group of indexes $\mathcal{G}$ of a GBF type **G** is the set of *positions* of a discrete space.
- A GBF G associates a value to some positions. As a partial function with finite support, G can be seen as a finite set of pairs (*position*, *value*). An element a of G, written $a \in G$, is such a pair and we use the sentences "position of a" and "value of a" to speak about the first and the second elements of this pair.
- A generator g of the group presentation **G** is also an *elementary translation* (we use equivalently the words *move*, *shift* or *direction*) from a position p to a position $p+g$.
- More generally, an element $x \in \mathcal{G}$ can be seen both as a position and as a translation (technically, we consider the left-action of $\mathcal{G}$ on itself).
- The set of elementary translations provide a *neighborhood relationship* to the set of positions: y is g-neighbor of x *iff* $x+g=y$. Two elements u and v are said neighbors, and we write "u,v" if there is a generator g such that u is a g-neighbor of v or v is a g-neighbor of u.
- A *path* is a sequence of positions u_i. It starts at the position u_0 and ends at the position u_n. Usually u_i

and u_{i+1} are neighbors, but we do not enforce this constraint. Paths can be translated by a translation t simply by adding t to each u_i.

- A *relative path* is a sequence r_i of positions. A relative path is a path but it is intended to be applied to a base position. The application of a relative path r_i to a position p_0 gives an actual path p_i defined as $p_{i+1} = p_i + r_i$.

The graphical representations of **G2** and **H2** in figure 1 can be enlightened from this topological point of view. In these diagrams, a vertex of the Cayley graph is pictured as a polygonal cell and two neighbors share an edge in this representation. For **G2**, each position (i.e. cell) has 4 neighbors corresponding to the `north` and `east` directions and their inverses. In **H2**, each cell has six neighbors (following the three generators and their inverses). The equation `X + Z = Y` specifies that a move following `Y` is the same has a move following the `X` direction followed by a move following the `Z` direction (or equivalently, the translations corresponding to the relative paths `Y` and `X,Z` are the same).

The spaces that can be described by a finite presentation are *uniform* in the sense that each position has the same number of neighbors reachable by the set of elementary moves. Spaces that can be described as GBF include:

- *n-ary trees* as the Cayley graph of a presentation of a *free group* with *n* generators [Ser77];
- *n-dimensional grids* as the Cayley graph of a presentation of a *free Abelian group* with *n* generators;
- *grids with circular dimension* and *screwed grids* corresponding to *Abelian groups*;
- *archimedian partitions of the plane* [Cha95].

5 A Generic Filter Language for Path Patterns

In a rule $\beta \Rightarrow f(\beta)$, the expression β is a pattern used to select a "part of a GBF". We call the part that can be matched and replaced a *sub-collection*. Our idea is to specify this pattern as a *path pattern* that matches *in some order* the elements of the sub-collection. A path is a sequence of elements and thus, a path pattern *Pat* is a sequence or a repetition *Rep* of *basic filters Bfilt*. A basic filter matches one element in a GBF. The grammar of path patterns reflects this decomposition:

Pat	::=	*Rep* | *Rep Dir Pat* | *Pat* `as` id | (*Pat*)
Rep	::=	*Bfilt* | id/*exp* | *Bfilt Dir*+ | *Bfilt Dir**
Bfilt	::=	`cte` | id | `_` | `<undef>`
Dir	::=	`,` | `|`u_1`, ..., `u_n`>`

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

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

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

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

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

neighbor: *b dir p* is a pattern that matches a path with its first element matched by *b* and continuing as a path matched by *p* whitch first element p_0 is such that p_0 is neighbor of *b* following the *dir* direction. The specification *dir* of a direction is interpreted as follows:

- the comma "`,`" means that p_0 and *b* must be neighbors;
- `|u>` means that p_0 must be a `u`-neighbor of *b*;
- the direction `|`u_1`, ..., `u_n`>` means that p_0 must be a u_0-neighbor *or* a u_1-neighbor *or* ... *or* a u_n-neighbor of *b*.

For example, *x*,*y* matches two connected elements (i.e., *x* must be a neighbor of *y*). The pattern

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

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

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

repetition: pattern *b dir*$*$ matches a possibly empty path *b dir b dir...dir b*. If the basic filter *b* is a variable, then its value refers to the sequence of matched elements and not to one of the individual values. The repetition *b dir*+ is similar but enforces a non-empty path. The pattern *x*+ is an abbreviation for "*x* `,`+".

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

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

Suppose that the pattern *Pat* `as` *P* is used to match a path in a GBF *G*. The value of a pattern variable *x*

used as a basic filter in *Pat* denotes a value found in G. The position of the matched value is denoted by $pos(x)$ which is an ad-hoc syntactic construct and not the call of a function *pos*. The value of the pattern variable P denotes the entire path matched by *Pat*. The value of P is a GBF of the same type of G containing only the matched elements. Thus, the construct $pos(P)$ denotes a GBF with the same domain as P and such that if $(p,v) \in P$, then $(p,p) \in pos(P)$. The elements in P have been matched following some order induced by the pattern expression *Pat*. The construct $seq(P)$ can be used to access to the sequence of the matched values and $seqpos(P)$ to the sequence of the positions of the matched elements.

6 Examples

We give immediately some examples of path patterns and complete MGS programs. The syntax and some specific features of MGS are sketched and explained through these examples.

Sequences

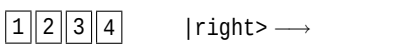
The sequence is a predefined collection type in MGS corresponding to the *list* algebraic data type in ML. However, we can specify as an exercise a similar collection type using the following GBF declaration:

```
gbf L = < right >
```

This example shows also the difference between the term rewriting approach of the algebraic data types and the path rewriting approach developed in MGS. A value of type **L** can be built using an enumeration: expression

```
L = 1 |right> 2 |right> 3 |right> 4
```

creates a new GBF of type **L** (the type is inferred from the generators used in the enumeration) with value 1, 2, 3 and 4. The value 1 is at the position 0|right>. The value 2 is at the right of the value 1 and then is at the position 1|right>. The value 4 is at position 3|right>. We can picture this GBF by:

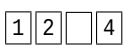


(the right direction extents to the horizontal right of the page; there is an infinite number of undefined elements that are not represented to the left of the element 1 and to the right of the element 4).

The main difference between an **L** and a value of the algebraic data type *list* is that an **L** is a partial data structure. One can then define a list "with holes":

```
L' = 1 |right> 2 |right> <undef> |right> 4
```

is pictured as:



The <undef> keyword is used to specify that the corresponding position must be left empty and an empty box ☐ is used in the picture (the empty boxes corresponding to the infinite number of undefined elements at the right and at the left are not represented).

Transformations can be used to program the usual functions on lists. For the head function *hd* that takes the head of a list in ML, we can write:

```
trans hd = {
    <undef> |right> x ⇒ return(x);
}
```

The statement return indicates that if the left hand side (l.h.s) matches, then the argument of return must be evaluated and returned as the global result of the entire transformation (instead of inserting the result in the collection and looking for others applications of the rule). The pattern <undef> |right> x matches an element x with an undefined neighbor at its left. Applied to a sequence without holes, there is only one such element x that can be matched. However, if the data structure has holes, like L', then every element at the right of an undefined element can match the rule. The result of the application of *hd* on such a structure is then one of these elements chosen in a non-deterministic manner. That is, $hd(L')$ returns either 1 or 4.

The code of the *last* function is very simple to specify because the last element in a sequence is "the element without a right neighbor":

```
trans last = {
    x |right> <undef> ⇒ return(x);
}
```

The definition of the *map* function is also very simple because it is enough to replace each value x in the GBF by $f(x)$:

```
trans mapf = { x ⇒ f(x); }
```

In this example, there is no return statement in the right hand side (r.h.s.) of the unique rule of the transformation. Then, the strategy for the transformation application is to apply in parallel as many occurrences of the rule as possible to the collection, provided that the sub-collection matched by an occurrence does not intersect a sub-collection matched by another occurrence. In this case, it means that every element x in the collection is replaced by $f(x)$.

We need a way to parameterize the transformation with the function f to be applied. This is easily done using an additional argument:

```
trans map(f) = { x ⇒ f(x); }
```

This transformation takes an additional argument f in addition to the collection. The result *map* is a curryfied function and

```
map (\x.x+1) L'
```

computes the GBF 2 3 ☐ 5.

The *fold* operator is written in the same way:

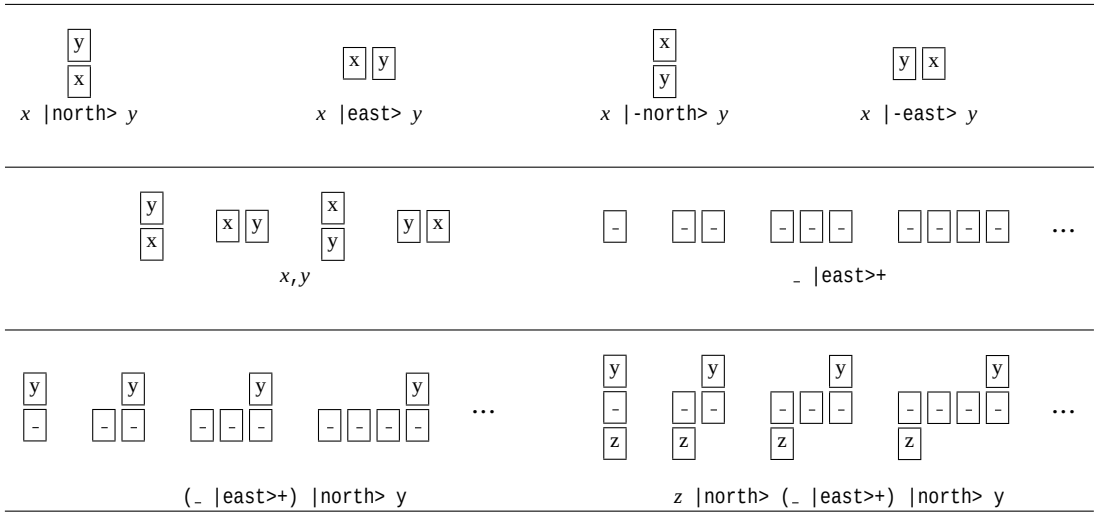


Figure 3. Several patterns and the corresponding path shapes in G2. For example, filter x,y matches four possible configurations as indicated.

```
trans fold(op) = {
  x |right> y |right> <undef>
  ⇒ op(x,y),<undef>,<undef>;
}
```

The transformation *fold* just replaces the last two elements x and y of the sequence by $op(x,y)$. Indeed, in a rule $p \Rightarrow sexp$, where the expression *sexp* computes a *built-in sequence* s of elements, the sequence s is used to replace *point-wise*[1] the elements matched by p. In addition, the comma operator in an expression corresponds to the built-in sequence constructor. Thus, the comma denotes ambiguously the neighborhood relationships in the l.h.s. of a rule and the building of a sequence in the r.h.s. (The two interpretations agree because two elements in a built-in sequence are neighbors if they are arguments of the comma constructor).

Thus *fold* $(\backslash x,y.x+y)\ L'$ evaluates to $\boxed{3}\boxed{\ }\boxed{\ }\boxed{4}$ (the element 4 cannot be matched by the rule because it is an isolated element). The expression *fold* $(\backslash x,y.x^*y)\ L$ evaluates to $\boxed{1}\boxed{2}\boxed{12}$. To obtain the full reduction, the transformation must be iterated until a fixed point is reached. This is provided in the MGS language using a special syntax for the iteration:

```
fold[iter=fixpoint] (\x,y.x*y) L
```

[1] If the r.h.s. computes a GBF g, then the GBF is inserted in place of the sub-collection matched by p if the "borders" of p and g agree, else it is an error. The notion of "border" is induced by the neighborhood relationship of the collection. This strategy agrees with the standard behavior of a rule in term rewriting where a term is replaced by another term.

The substitution behavior sketched in the text coexists gracefully with the standard one. Both are meaningful because a pattern specifies both a path, i.e. a sequence of elements, and a sub-collection. In this paper, we use only the substitution strategy presented in the text where the r.h.s. evaluates to a sequence of elements.

the optional named parameters in the brackets are used to tune the application strategy of a transformation. The `iter` parameter controls the iteration of a transformation [GM01c]: `fixpoint` indicates the iteration of the transformation until a fixed point is reached; `fixrule` specifies the same behavior but the fixed point is detected when no rule applies; an integer n stands for n iterations; etc. The result of the previous expression is $\boxed{24}$ (a GBF of type **L** with only one element).

The *cons* function used to add an element a in front of a sequence l can be defined as the transformation:

```
trans cons(a) = {
    <undef> |right> x ⇒ a,x;
}
```

This transformation works as follow: all the elements without a left neighbor gain a new element a located at their left. So, *cons* `9` L evaluates to $\boxed{9}\boxed{1}\boxed{2}\boxed{9}\boxed{4}$.

Path Patterns in a NEWS Grid

We assume working in **G2**. Then, the pattern

```
x |north> y
```

matches two elements x and y with y at the north of the element x. Using the convention used in the left diagram in figure 1, this filter can be represented as a vertical domino. Figure 3 depicts several other filters in **G2**. In this figure, a box $\boxed{\_}$ indicates a matched element in a GBF which is not binded to a pattern variable.

Finding One's Way in a Labyrinth

Consider a labyrinth represented as a GBF where the value 1 denotes the entry doors, the value 2 codes the corridors and the value 3 the exit doors. Then finding a path between the entry and the exit doors is simply

specified as:

```
(1, (2,*), 3)
```

this pattern matches a path beginning with 1 and ending with 3 after a sequence of 2. This path can be used in a transformation

```
trans FindPath = {
  (1,(2,*),3) as P ⇒ return(seqpos(P));
}
```

The statement `return` indicates that the transformation must stop and return the argument value as soon as this rule matches. The returned value is the sequence of the positions of the path *P* matched by the l.h.s.

Rotation of the Cross

The transformation *Turn* on the square lattice **G2** in section 2 can be specified as:

```
trans Turn = {
  a |east> b
    |north-east> c
    |-east-north> d
    |east-north> e
  ⇒   a,e,b,c,d ;
}
```

The sequence *s* computed in the r.h.s. of the rule is used to replace *point-wise* the elements matched by the l.h.s. Then, the first element *a* of the sequence *s* replace the element named *a* in the pattern. The second element, which is *e*, replace the element named *b*, etc. The net result is a 90°-rotation of the cross matched in the l.h.s. of the rule, leaving the center *a* unmodified.

The specification of the rotation is also straightforward in **H2**:

```
trans Turn_h = {
  a |X> b
    |Z> c
    |-X> d
    |-Y> e
    |-Z> f
    |X> g
  ⇒   a,g,b,c,d,e,f ;   }
```

Eden Growing Process

We consider a simple model of growth sometimes called the Eden model (a type B Eden model [YPQ58] to be more precise). The model has been used since the 60's as a model for things such as tumor growth and growth of cities. In this model, a 2D space is partitioned into empty and occupied cells (we use the value `true` for an occupied cell and left undefined the unoccupied cells). We start with only one occupied cell. At each step, occupied cells with an empty neighbor are selected, and the corresponding empty cell is made occupied. The Eden's aggregation process is simply described as the following `MGS` global transformation:

```
trans Eden = { x,<undef> ⇒ x,true ; }
```

We assume that the boolean value `true` is used to represent an occupied cell, other cells are simply left undefined. The special symbol `<undef>` is used to match an undefined value. Then the previous rule can be read: an occupied element *x* and an undefined neighbor are transformed into two occupied elements. The transformation *Eden* defines a function that can then be applied to compute the evolution of some initial state. See the first evolution steps in figure 4.

One of the advantages of the `MGS` approach, is that this transformation can be applied indifferently on grid or hexagonal lattices, or *any* other collection kind (this also holds for the transformation *FindPath*).

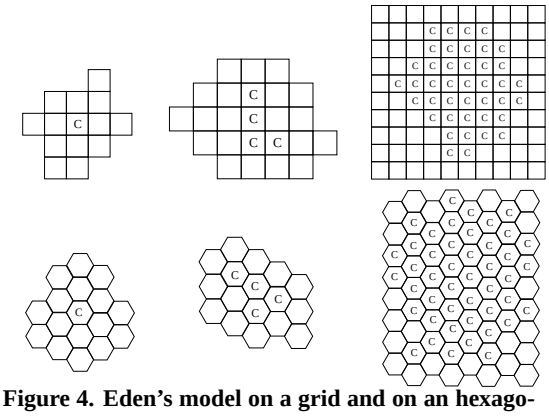


Figure 4. Eden's model on a grid and on an hexagonal mesh (initial state, and states after 2 and 6 time steps). Exactly the same `MGS` transformation is used for both cases. An empty cell has an undefined value and only a part of the infinite domain is figured.

7 A Generic Pattern-Matching Algorithm

We present in this section a simplified pattern-matching algorithm for GBF path patterns. This algorithm is inspired from the approach taken by J. A. Brzozowski for the computation of the *derivatives of regular expressions* [Brz64]. We recall in the next paragraph the notion of derivative of a regular expression. Then we restrict the language of pattern expression to its fundamental core and we introduce the notations used before defining the derivative of a path pattern. This section ends by a very simple but complete example of path computations.

The Derivatives of a regular Expression

Let R be a regular expression and L_R the language recognized by R. For any letter $a \in A$ the derivative of R with respect to a is denoted by $\partial R/\partial a$ and is

$$\frac{\partial R}{\partial a} = \{m \text{ such that } am \in L_R\}$$

The idea of derivative with respect to a letter can be defined generally for a set L but it turns out that the derivative of a regular expression can be defined by a regular

expression. For example,

$$\frac{\partial a.(a+b)^*}{\partial a} = \varepsilon + (a+b)^*$$

In words: if am is a word recognized by $a.(a+b)^*$ then m is either empty or recognized by $(a+b)^*$. The derivative of a regular expression R is another regular expression that can be derived using simple rule on the structure of R. These symbolic rules formally mimic the classical rules of the derivation of real functions, hence the name.

The notion of derivative has been used in word recognition because if $m = m_1 m_2 \dots m_n$, then $m \in L_R$ iff $m_2 \dots m_n \in \partial R / \partial m_1$. By iteration, the membership problem is then reduced to the membership of the empty word ε to the language recognized by a regular expression.

The annulator $[R]$ of a regular expression is defined by:

$$[R] = \begin{cases} \emptyset & \text{if } \varepsilon \notin L_R \\ \{\varepsilon\} & \text{if } \varepsilon \in L_R \end{cases}$$

and can also be computed by symbolic rules on the structure of R. This gives a canonical decomposition of the words of L_R:

$$L_R = [R] \cup \bigcup_{a \in A} a \otimes \frac{\partial R}{\partial a}$$

where $a \otimes L = \{a.m \text{ where } m \in L\}$. Remark that $a \otimes \emptyset = \emptyset$ and that $a \otimes \{\varepsilon\} = \{a\}$.

We want to adapt these ideas to our case: a path pattern will play a role similar to a regular expression and the GBF will correspond to the vocabulary A. Several differences have to be taken into account:

- The notion of derivative of a regular expression is traditionally used to check if a word belongs to a language defined by a regular expression. In our case, we want to enumerate the paths matched by a path pattern in a GBF.
- A path and a path pattern exhibit both a canonical order over their elements. However, there is no such canonical order between the elements of a GBF.
- There is only one possible letter following another letter in a word. There are several possible neighbor of a given element in a GBF.
- Path patterns include logical expressions involving the value of the matched elements through the binding of some variables.

The Pattern Expressions

For the sake of the simplicity, we restrict the grammar of path patterns to the following abstract syntax:

Pattern	::=	*Atom* | *Atom Dir Pattern*
Atom	::=	id/*exp* | *Dir* $*$
Dir	::=	`|`u_1`, ..., `u_n`>`

Notice that a literal pattern *cte* can be rewritten a `/` $a =$ *cte* where a is a fresh variable. A variable is systematically guarded but one can use the pattern a`/true` if there is no check to do. The neighborhood relation `,` can be recovered as the direction `|`g_1`, ..., `g_n`, -`g_1`, ..., -`g_n`>` where the g_i are the generators of the GBF type. There is no naming in a repetition pattern to simplify the handling of the variable bindings. The unnamed filter "`_`" in the previous syntax can be coded as a`/true` where a is a fresh variable and "`_ |`u_1`, ...,`u_n`>*`" in the old syntax is coded as `|`u_1`, ...,`u_n`>*` in the new syntax.The non-empty repetition `+` can be recovered using `*`, e.g. p *dir*`+` can be rewritten as

```
p dir dir*
```

using fresh variables where needed. The handling of the naming of a sub-pattern presents no special difficulties but would burden a lot the presentation. For the same reason, we drop the handling of the `<undef>` basic filter[2].

For example, the path pattern

x`, (_ |north>+) |east>` y

in **G2** can be rewritten in the new syntax:

```
(x/true)
|north,east,-north,-east>
(u/true)
|north>
(|north>*)
|east>
(y/true)
```

Notations

We use brackets to enumerate the elements in a set and for set comprehension. The symbol $\emptyset$ is for the empty set. The expression $S - e$ denotes the set S without the element e. $[]$ is the empty list; $\ell @ \ell'$ is the concatenation of lists ℓ and ℓ'. The *distribution* $e \otimes S$ of an expression e over the elements of a set S of lists is defined by $\{[e]@l, l \in S\}$. An *environment* is a partial function defined for a set of identifiers $i_1, ..., i_n$ with values $v_1, ..., v_n$, and elsewhere undefined; E ranges over the environments; the *augmentation* of an environment E with identifier i_{n+1} and value v_{n+1} is a new environment $E' = E + [i_{n+1} \to v_{n+1}]$, such that $E'(i_{n+1}) = v_{n+1}$ and $\forall k, i_k \neq i_{n+1}, E'(i_k) = E(i_k)$.

[2] The handling of `<undef>` is complicated and would burden a lot our exposition. We sketch two examples to show the difficulties. A rule like `<undef>` $\Rightarrow$ `1` is forbidden in `MGS` because it implies the replacement of all undefined elements by a `1` and there is possibly an infinite number of such elements. Other example: in the processing of a rule like `<undef>,`$x \Rightarrow$ `1,`x we cannot start by looking for an undefined element (because there could be an infinite number of such elements) but rather we have to look for a defined element x that has an undefined neighbor.

$$\frac{\partial \,\mathrm{dir}*}{\partial\, p}(G,E,\emptyset) = \{[]\} \quad (1)$$

$$\frac{\partial P}{\partial\, p}(G,E,\emptyset) = \emptyset \quad \text{provided that } P \neq \mathrm{dir}* \quad (2)$$

$$\frac{\partial\, \mathrm{id/expr}}{\partial\, p}(G,E,\Pi) = \texttt{if eval}(E+[\mathrm{id}\to p],G,\mathrm{expr})\ \texttt{then}\ \{[p]\}\ \texttt{else}\ \emptyset \quad (3)$$

$$\frac{\partial\, \mathrm{dir}*}{\partial\, p}(G,E,\Pi) = \{[]\} \cup \frac{\partial\,(\mathrm{id/}\texttt{true}\ \mathrm{dir}\ \mathrm{dir}*)}{\partial\, p}(G,E,\Pi) \quad \text{where id is a fresh variable} \quad (4)$$

$$\begin{aligned}\frac{\partial\, \mathrm{id/expr\ dir\ P}}{\partial\, p}(G,E,\Pi) = {} & \texttt{let } E' = E+[\mathrm{id}\to p] \quad \texttt{and} \quad \Pi' = \Pi - p \\ & \texttt{in if eval}(E',G,\mathrm{expr}) \\ & \quad \texttt{then } p \otimes \left(\bigcup_{p'\in \texttt{neighbor}(\Pi',\mathrm{dir},p)} \frac{\partial P}{\partial\, p'}(G,E',\Pi') \right) \\ & \quad \texttt{else } \emptyset\end{aligned} \quad (5)$$

$$\begin{aligned}\frac{\partial\, \mathrm{dir}*\ \mathrm{dir}'\ P}{\partial\, p}(G,E,\Pi) = {} & \bigcup_{p'\in \texttt{neighbor}(\Pi,\mathrm{dir}',p)} \left(\frac{\partial P}{\partial\, p'}(G,E,\Pi)\right) \quad \text{where id is a fresh variable} \\ & \cup \quad \frac{\partial\,(\mathrm{id/}\texttt{true}\ \mathrm{dir}\ \mathrm{dir}*\ \mathrm{dir}'\ P)}{\partial\, p}(G,E,\Pi)\end{aligned} \quad (6)$$

Figure 5. Specification of the derivatives of a path pattern. We suppose that $\Pi \neq \emptyset$ in the equations.

Derivatives *of a Path Pattern*

A pattern-matching expression is an element of *Pattern*. The *derivative* of a pattern-matching expression P with respect to a position p, given a set G of pairs *(position, value)* (i.e., a GBF), an environment E and a set of available positions Π is written

$$\frac{\partial P}{\partial\, p}(G,E,\Pi)$$

and represents *the set of paths in a GBF G starting at position p and matched by the path pattern P*. The environment E is an additional argument used to record the variable bindings used in the evaluation of guards in a pattern. The result of $\partial P/\partial\, p(G,E,\Pi)$ is a set of lists ℓ of positions. Such a list ℓ records the sequence of the elements of the GBF that match the path pattern P.

Let ε be the empty environment, and $dom(G)$ the set of positions which have a value in G then all the occurrences of a path pattern P in a GBF G are computed by:

$$\bigcup_{p\in dom(G)} \frac{\partial P}{\partial\, p}(G,\varepsilon,dom(G)) \quad (7)$$

The derivatives of a path pattern is a 5-ary function $\partial\cdot/\partial\cdot(\cdot,\cdot,\cdot)$ defined by induction on the path pattern P and the GBF G. The specification is given in figure 5 and use two additional functions: $\texttt{eval}(E,C,\mathrm{expr})$ is a predicate that holds when the expression expr evaluates to the boolean true value in the environment E with respect to G; $\texttt{neighbor}(\Pi,dir,p)$ is a function that computes, given a set of positions Π and a list of directions , the neighbor positions of a position p in Π:

$$\begin{aligned}&\texttt{neighbor}(\Pi, \texttt{|u}_1\texttt{, ..., u}_n\texttt{>}, p) \\ &\qquad = \{p+\mathtt{u}_i \mid 1\le i\le n \text{ and } p+\mathtt{u}_i \in \Pi\}\end{aligned}$$

The equations in figure 5 can be intuitively explained as follow:

1. There is only one empty path in an empty GBF.
2. There is no non-empty path in an empty GBF.
3. A path reduced to only one element matches an element at position p if the condition expr is met. In this case, there is only one possible path with only one element at position p. If the condition is not met, there is no singleton path starting at p.
4. A path specified by dir$*$ starting at position p is either empty or begins with the value at position p and continues following the direction dir as a path specified by dir$*$.
5. The paths starting at position p and beginning with an element id satisfying condition exp and then following direction dir to continue as a path P can exist only if the condition is satisfied. This condition is checked by $\texttt{eval}(E',G,\mathrm{expr})$ using the

augmented environment E': E' contains the previous bindings together with the binding of id with the position p.

If the condition is satisfied, then such a path can be obtained by computing the paths starting from a dir-neighbor p' of p and matching P and then adding the position p in front of these paths thanks to the $\otimes$ operator.

6. The last rule decomposes into two sets the paths starting at position p beginning with a repetition dir$*$ and continuing following direction dir$'$ by a path matched by P.

   The first set corresponds to an empty repetition. So, we want to match the paths specified by P starting from a dir$'$-neighbor of P.

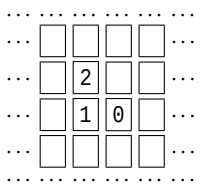
   The second set corresponds to a non empty-repetition and we just unfold the repetition one time.

Example of Derivative Computation

To make these definitions more concrete, we compute the path matching the pattern "`_, 1 |north> x`". This pattern is first transformed into

$P =$ `u/true`
`|north,east,-north,-east>`
Q
$Q =$ `v/v=1`
`|north>`
`x/true`

(for convenience, we introduce a meta-variable Q to name a sub-pattern). We look for some paths in the GBF G of type **G2**



which is represented as the set of pairs (*position*, *value*). To spare the notation, we write a couple (n,e) for a position "n`|north> + `e`|east>`".

$$G = \{\ ((0,0),\texttt{1}),\ ((0,1),\texttt{0}),\ ((1,0),\texttt{2})\ \}$$

We have arbitrarily fixed the value `1` at position $(0,0)$. There is only one path matching P in G: $[(0,1);(0,0);(1,0)]$. Indeed $(0,0)$ is a neighbor of $(0,1)$ and its value is `1`. Moreover, at north of $(0,0)$, i.e. at position $(1,0)$, there is a value.

The domain of G is called Π :

$$\Pi = dom(G) = \{(0,0),\ (0,1),\ (1,0)\}$$

All the paths matched by P are computed using the definition (7):

$$\frac{\partial P}{\partial(0,0)}(G,\varepsilon,\Pi) \cup \frac{\partial P}{\partial(0,1)}(G,\varepsilon,\Pi) \cup \frac{\partial P}{\partial(1,0)}(G,\varepsilon,\Pi) \quad (8)$$

Then we have:

$$\frac{\partial P}{\partial(0,0)}(G,\varepsilon,\Pi) = (0,0) \otimes \bigcup_{p' \in \{(1,0),(0,1)\}} \frac{\partial Q}{\partial p'}(G,[u \to (0,0)],\Pi')$$

where $\Pi' = \{(0,1),(1,0)\}$. The union is composed of two terms. The first one evaluates to $\emptyset$:

$$\frac{\partial Q}{\partial(1,0)}(G,[u \to (0,0)],\Pi') = (1,0) \otimes \bigcup_{p' \in \emptyset} \frac{\partial x/\texttt{true}}{\partial p'}(...)$$

where the union is made on an empty set of indexes, so:

$$\frac{\partial Q}{\partial(1,0)}(G,[u \to (0,0)],\Pi') = (1,0) \otimes \emptyset = \emptyset$$

The second term $\frac{\partial Q}{\partial(0,1)}(G,[u \to (0,0)],\Pi')$ gives a similar result and then:

$$\frac{\partial P}{\partial(0,0)}(G,\varepsilon,\Pi) = (0,0) \otimes \emptyset = \emptyset$$

This result is also true for $\frac{\partial P}{\partial(1,0)}(G,\varepsilon,\Pi)$.

There is a difference in the computation of:

$$\frac{\partial P}{\partial(0,1)}(G,\varepsilon,\Pi) = (0,1) \otimes \bigcup_{p' \in \{(0,0)\}} \frac{\partial Q}{\partial p'}(G,[u \to (0,1)],\Pi'')$$

where $\Pi'' = \{(0,0),(1,0)\}$. The union term does not reduce to the empty set:

$$\frac{\partial Q}{\partial(0,0)}(G,[u \to (0,1)],\Pi'') = (0,0) \otimes \frac{\partial x/\texttt{true}}{\partial(1,0)}(G,[u \to (0,1), v \to (0,0)],\Pi''')$$

where $\Pi''' = \Pi'' - (0,0) = \{(1,0)\}$. Because

$$\frac{\partial x/\texttt{true}}{\partial(1,0)}(G,...,\Pi''') = \{[(1,0)]\}$$

we have then that

$$\begin{aligned} (8) &= (0,1) \otimes \Big((0,0) \otimes \big\{[(1,0)]\big\}\Big) \\ &= \big\{[(0,1);(0,0);(1,0)]\big\} \end{aligned}$$

which is what was expected.

8 Conclusions

The array data structure is not smoothly handled in functional languages because it cannot be described convincingly as instances of an algebraic data type. Therefore, there are no means to specify by cases a function on an array. This annoying situation is summarized by Wadge: "We spent a great deal of efforts trying to find a simple algebra of arrays (...) with little success" [WA85].

In this work, we have presented a framework, the group-based data fields, that allows a uniform description of

trees and arrays in the same framework [GM01a]. The GBF approach puts the emphasis on the logical neighborhood of the data structure elements [GM02a]. This topological point of view allows the definition of path patterns used to match a sub-collection in an array or a tree. A first algorithm to enumerate all the paths matched by a pattern is given, inspired by the notion of derivative developed for the recognition of regular expressions on sequences. This algorithm has been extended to handle a more complete pattern language and is used in the current version of the `MGS` interpreter (see the web home page `http://mgs.lami.univ-evry`). This interpreter handles the examples proposed in section 6 as well as more intricate ones like:

$x,\ (y+\ /\ x\ >\ \mathtt{Sum}(y))$

that looks for a path beginning with an x that is greater than the sum of the rest of the matched elements (the function `Sum` is an auxiliary function that computes the sum of all elements in an collection of numbers). A remarkable feature is that the same algorithm sketched here is used to find the occurrences of a pattern in a set, a multiset, a sequence or a GBF. We think that this demonstrates the usefulness and the unifying nature of our topological framework.

Several other examples of the programming style allowed by `MGS` rules on GBF are developed in [GGMP02] in the context of biological simulations. Many mathematical models of objects and processes are based on a notion of state that specifies the object or the process by assigning some data to each point of a physical or abstract space. The goal of `MGS` is to support this approach by offering several mechanisms to build complex and evolving spaces and handling the mappings between these spaces and the data in a functional framework. In this context, GBF are used to model the uniform and regular discretization of spaces.

Pattern matching in arrays has been considered in the functional languages community from [Bir77, Bak78] and more recently in [Jeu92] but the problem is then restricted to determine an occurrence of a rectangular sub-array. For example, if P is a $p \times q$ rectangular two-dimensional array (a pattern of literals), and G is a $n \times m$ array, the problem handled is to find a pair (i, j) such that for all k and l such that $1 \leq k \leq p$ and $1 \leq l \leq q$, we have $G[i-p+k, j-q+l] = P[k,l]$.

Compared to these previous works, our algorithm is more general in two directions: it handles group-indexed data structures and it allows a more expressive pattern language. Obviously, there is a large room for optimizations. For instance, we do not compute all paths before applying a rule but we stop the search as soon as one matching path has been found. By specifying an order over the unions appearing in the definition of the derivative Fig. 5, we can parameterize a strategy for the enumeration of paths. We are currently developing a pattern compiler for `MGS` based on pattern transformations.

Acknowledgments

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 P. Prusinkiewicz and F. Delaplace for numerous questions, encouragements and thoughtful remarks. This research is supported in part by the CNRS, the GDR ALP and IMPG, the University of Evry and GENOPOLE-Evry.

9 References

[Bak78] Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.*, 7(4):533–541, 1978.

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[Bir77] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, October 1977.

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

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, 1964.

[Cha95] Thomas Chaboud. About planar cayley graphs. In *Fundamentals of Computation Theory (FCT '95)*, volume 965 of *LNCS*, pages 137–142, 1995.

[CiCL91] Marina Chen, Young il Choo, and Jingke Li. Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 7, pages 255–308. ACM Press, New York, 1991.

[GDVS98] J.-L. Giavitto, D. De Vito, and J.-P. Sansonnet. A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$. In *EuroPar'98 Parallel Processing*, volume 1470 of *LNCS*, pages 742–??, September 1998.

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

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

[GM01a] J.-L. Giavitto and O. Michel. Declarative definition of group indexed data struc-

tures and approximation of their domains. In *Proceedings of the 3nd Imternational ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, September 2001.

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

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

[GM02a] J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49:107–129, 2002.

[GM02b] J.-L. Giavitto and O. Michel. Data Structure as Topological Spaces. In *3th Int. Conf. on Unconventional Models of ComputationFundamenta Informaticae*, Himeji, Japan. To be published in the LNCS serie. Spinger, 2002.

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

[Jeu92] J. Jeuring. The derivation of a hierarchy of algorithms for pattern matching on arrays. In G. Hains and L. M. R. Mullin, editors, *Proceedings ATABLE-92, Second international workshop on array structures*, 1992.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[Lis96] B. Lisper. Data parallelism and functional programming. In *Proc. ParaDigme Spring School on Data Parallelism*. Springer-Verlag, March 1996. Les Ménuires, France.

[Man01] Vincenzo Manca. Logical string rewriting. *Theoretical Computer Science*, 264(1):25–51, August 2001.

[Pau99] G. Paun. Computing with membranes: An introduction. *Bulletin of the European Association for Theoretical Computer Science*, 67:139–152, February 1999.

[RS92] G. Rozenberg and A. Salomaa. *Lindenmayer Systems*. Springer, Berlin, 1992.

[Ser77] J.-P. Serre. *Arbres, Amalgames, SL_2*. Number 46 in Astérisque. Société Mathématique de France, 1977.

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

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Data flow programming language*. Academic Press U. K., 1985.

[YC92] J. Allan Yang and Young-il Choo. Data fields as parallel programs. In *Proceedings of the Second International Workshop on Array Structure*, Montreal, Canada, June/July 1992.

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