LaMI

Laboratoire de Méthodes Informatiques

Pattern-matching and Rewriting Rules for Group Indexed Data-Structure

Jean-Louis Giavitto, Olivier Michel & Julien Cohen

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

Rapport de Recherche n^o **76-2002**

Juin 2002

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

Pattern-matching and Rewriting Rules for Group Indexed Data-Structure

Jean-Louis Giavitto, Olivier Michel & Julien Cohen

LaMI u.m.r. 8042 du CNRS, Université d'Évry Val d'Essone – GENOPOLE
Tour Évry2, 523 place des terasses de l'agora.
91000 Évry, France.
Tel: +33 1.60.87.39.04
[giavitto,michel,jcohen]@lami.univ-evry.fr

LaMI technical report N° 76-2002, June 2002

Abstract

In this report, 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 of functions. This framework is implemented in an experimental language called **MGS**. The underlying notions funding our framework have a topological nature and make possible to extend the case-based definition of functions found in modern functional languages beyond algebraic data-structures.

Keywords

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

The authors of this research report can be contacted at:

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

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

The **MGS** interpreters are freely available from the **MGS** home page located at url :
`http://www.lami.univ-evry.fr/mgs` .

Versions of this report:

- Initial Version: June 2002.

Pattern-matching and Rewriting Rules for Group Indexed Data-Structure

Jean-Louis Giavitto, Olivier Michel & Julien Cohen

LaMI u.m.r. 8042 du CNRS
Université d'Evry Val d'Essone
91000 Evry, France.
[giavitto,michel]@lami.univ-evry.fr

LaMI technical report N° 76-2001, June 2002.

1 Introduction

One of the achievement and success of current functional languages is the ability to define functions by case 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 a data-structure beyond the current capability is for example the *array data-type*: it is not possible to define a function by case on an array.

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 of 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-structure 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 structure generalizes the notion of array. We give a geometric interpretation of GBFs 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 such kind of cellular automata, rules of forms $\beta \Rightarrow f(\beta)$ 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 square lattice (see the two diagrams at left 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. Note that in this

case, the pattern β does not filter a subarray but an arbitrary subset (a cross). Such 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* `=` { $\beta \Rightarrow f(\beta)$; }

The transformation *Turn* is defined by case (here there is only one case corresponding to one rule in the transformation *Turn*). The case β specifies a sub-domain which is replaced by $f(\beta)$. However, in opposition with case-based function definition acting on algebraic data-type, the cases do not correspond to constructors nor exhaust the data-structure.

It is usual for physicists to work with an hexagonal lattice, because such 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 tiling, cf. the two diagrams at the right of figure 1. In this case, the pattern β involves a 7 cells sub-domain.

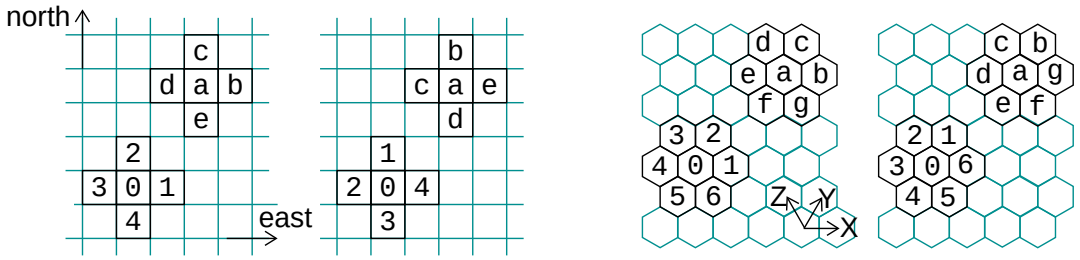

Figure 1: Application of transformation *Turn* on the array to the left or to the hexagonal subdivision at the right. In contrast with cellular automata, the evolution concerns a multi-cell domain.

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 sub-domain in an array,
3. specify a function using rules like $\beta \Rightarrow f(\beta)$ that specifies the substitution of non-intersecting occurrences of subdomains matched by β by a replacement computed as $f(\beta)$.

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 show 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-structure. 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 with generalizes the concept of array. Such data structure admits a geometrical interpretation which is the basis of the language of filters presented in section 5.

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 indices $\mathcal{I} = [1, n] \times [1, m]$ 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 indices $\mathcal{I} = \mathbb{Z} \times \mathbb{Z}$ (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 indices $\mathcal{I}$ 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 indices in scientific programs [GG95]. As a consequence, we have proposed to provide the set of indices with a *group structure* [GMS96]. Such 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 indices is the group $(\mathbb{Z}^n, +)$. The advantage of providing the set of indices 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 indices. 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 $e_k = e'_k$:

`gbf` ***G*** `=` `<` g_1`,` `...,` g_n`;` $e_1 = e'_1$`,` `...,` $e_p = e'_p$ `>`

We use the following typographical conventions: if G is a GBF, we write $\boldsymbol{G}$ (a finite group presentation) for its type and $\mathcal{G}$ (the group of indices 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 $\boldsymbol{G}$ because **MGS** is a dynamically typed language and heterogeneous values can be recorded in a GBF.

In this paper we deal only with abelian group 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 Indices and Topological Representation

A Cayley graph is a graph representation of the presentation $\boldsymbol{G}$ 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 $\boldsymbol{G}$ such that $x + \mathtt{u} = y$. See figure 2. This representation support the following *topological interpretation* of a GBF:

- The group of indices $\mathcal{G}$ of a GBF type $\boldsymbol{G}$ 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 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 $\boldsymbol{G}$ 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 relationships* 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 position u_0 and ends at 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 enlighten 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 inverse. 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 kind of 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 same set of elementary moves. Spaces that can be described as GBFs 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].

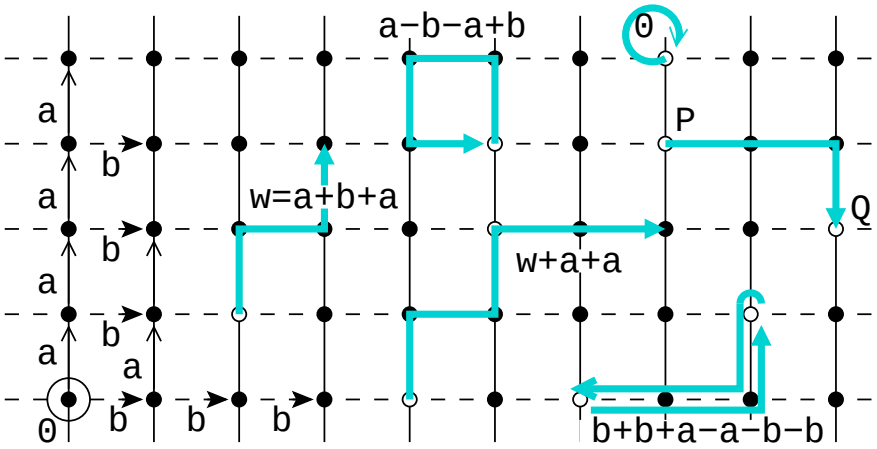

Figure 2: Graphical representation of the relationships between Cayley graphs and group theory. A vertex is a group element. An edge labeled a is a generator a of the group. A word (a formal sum of generators) is a path. Path composition corresponds to group addition. A closed path (a cycle) is a word equal to 0 (the identity of the group operation). An equation $v = w$ can be rewritten $v - w = e$ 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 x to y, is equivalent to say that there is always a solution v to equation $x + v = y$.

5 A Generic Filter Languages 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
REP	::=	BFILT | BFILT/*exp* | BFILT DIR `+` | BFILT DIR `*`
BFILT	::=	cte | id | _ | `<undef>`
DIR	::=	`,` | `|`$\mathtt{u}_1$`, ..., `$\mathtt{u}_n$`>`

where cte is a literal value, id ranges over the pattern variables, *exp* is a boolean expression, and $\mathtt{u}_i$ is a word 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.

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. The position of a is accessible through the expression $pos(x)$. The identifier of a pattern variable can be used only once in the position of a filter. 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.

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.

neighbor: b DIR p is a pattern that matches a path with first element matched by b and continuing as a path matched by p with the first element p_0 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 `|`$\mathtt{u}_1$`, ..., `$\mathtt{u}_n$`>` means that p_0 must be a $\mathtt{u}_0$-neighbor *or* a $\mathtt{u}_1$-neighbor *or* ... *or* a $\mathtt{u}_n$-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 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 `,+`".

6 Examples

We give immediately some examples of path patterns and complete **MGS** programs.

Finding its way in a labyrinth. Suppose 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:

$$\texttt{trans}\ \mathit{FindPath}\ =\ \{\quad 1, 2\texttt{*}, 3\ \Rightarrow\ \texttt{Print("there is a path")}\,;\quad \}$$

The pattern $1, 2\texttt{*}, 3$ matches a path beginning with 1 and ending with 3 after a sequence of 2. If one wants to record the sequence of positions corresponding to the travel from the entry to the exit door, one can use the rule

$$(1, 2\texttt{*}, 3)\ \texttt{as}\ P\ \Rightarrow\ \mathit{pos}(P)$$

the construction `as` is used to give a name to a path matched by an arbitrary sub-pattern.

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

$$\begin{array}{l}\texttt{trans}\ \mathit{Turn}\ =\ \{\\ \quad a\ \texttt{|east>}\ b\ \texttt{|north-east>}\ c\ \texttt{|-east-north>}\ d\ \texttt{|east-north>}\ e\ \Rightarrow\ a, e, b, c, d\,;\\ \}\end{array}$$

To understand why the right hand side (r.h.s.) of the rule specifies a 90°-rotation of the cross matched in the left hand side (l.h.s.) of the rule, one must know that the comma operator in an expression corresponds to the sequence constructor. Thus, the comma denotes ambiguously the neighborhood relationships in the l.h.s. of a rule and building of a sequence in the r.h.s. (The two interpretations agree because two elements in a sequence are neighbors if they are argument of a constructor). Moreover, in a rule $p \Rightarrow \mathit{sexp}$, where the expression *sexp* computes *a sequence* s of elements, the sequence s is used to replace *point-wise* the elements matched by p. (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 specification of *Turn* is also straightforward in ***H2***:

$$\texttt{trans}\ \mathit{Turn}\ =\ \{\ a\ \texttt{|X>}\ b\ \texttt{|Z>}\ c\ \texttt{|-X>}\ d\ \texttt{|-Y>}\ e\ \texttt{|-Z>}\ f\ \texttt{|X>}\ g\ \Rightarrow\ a, g, b, c, d, e, f\,;\ \}$$

Eden's growing process. We consider a simple model of growth sometimes called the Eden model (specifically, a type B Eden model [YPQ58]). 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 in empty or 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:

$$\texttt{trans}\ \mathit{Eden}\ \ =\ \ \{\quad x, \texttt{<undef>}\ \Rightarrow\ x, \texttt{true}\ \texttt{;}\quad \}$$

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 figure 3 for an illustration.

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

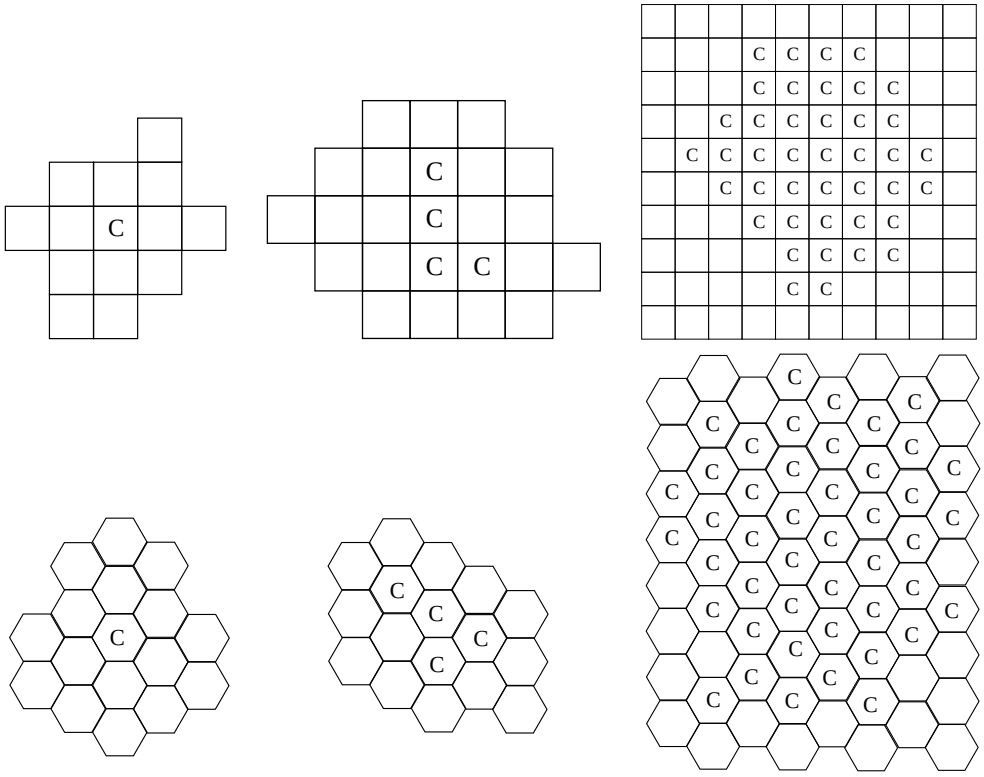

Figure 3: Eden's model on a grid and on an hexagonal mesh (initial state, and states after the 3 and the 7 time steps). *Exactly the same* transformation is used for both cases. These shapes correspond to a Cayley graph of ***G2*** and ***H2*** with the following conventions: a vertex is represented as a face and two neighbors in the Cayley graphs share an edge in this representation. An empty cell has an undefined value. Only a part of the infinite domain is figured.

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

$$\begin{array}{rcl} \textsc{Pattern} & ::= & \textsc{Atom} \mid \textsc{Atom Dir Pattern} \\ \textsc{Atom} & ::= & a\texttt{/}exp \mid \text{id } \textsc{Dir} * \\ \textsc{Dir} & ::= & \texttt{<}\mathtt{u}_1\texttt{, ..., }\mathtt{u}_n\texttt{>} \end{array}$$

Note 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 unnamed filter "_" can be coded as a`/true` where a is a fresh variable. The neighborhood relation `,` can be recovered as the direction `<`$\mathtt{g}_1$`, ..., `$\mathtt{g}_n$`, -`$\mathtt{g}_1$`, ..., -`$\mathtt{g}_n$`>` where the $\mathtt{g}_i$ are the generators of the GBF type. The non-empty repetition `+` can be recovered using `*`, e.g. p ***dir*****`+`** can be rewritten as `p` ***dir*** p ***dir*****`*`** using fresh variables where needed.

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 $S - x$ denotes the set S without the element x. `List`(X) represents the set of lists of elements of X; $[\,]$ is the empty list; $\ell @ \ell'$ is the concatenation

of lists ℓ and ℓ'. The *distribution* $S \otimes e$ of an expression e over elements of a set S is defined as $\{l@[e], l \in S\}$. An *environment* is a partial function defined for a set of identifier $i_1, ..., i_n$ with value $v_1, ..., v_n$, and elsewhere undefined; E ranges over environments; the *augmentation* of an environment E with identifier i_{n+1} and value v_{n+1} is a new environment $E' = E + [i_{n+1} \rightarrow v_{n+1}]$, such that $E'(i_{n+1}) = v$ and $\forall k, k \neq n+1, E'(i_k) = E(i_k)$.

***Derivatives* of a path pattern.** A pattern-matching expression is an element of PAT. The *derivative* of a pattern-matching expression P with respect to a position p, given a set C of pairs *(position, value)* (i.e., a GBF) and an environment E is written

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

and represents the set of paths in a GBF C 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 $\partial P / \partial p(C, E)$ is a set of lists ℓ of positions. Let ε be the empty environment, then all the occurrences of a path pattern P in a GBF C are computed by:

$$\bigcup_{p \in \{q \mid \exists v,\, (q,v) \in C\}} \frac{\partial P}{\partial p}(C, \varepsilon)$$

In the definition of the function $\partial \cdot / \partial \cdot (\cdot, \cdot)$ that follows, three additional functions are required: $\texttt{val}(C, p)$ is a function that requires a GBF C and a position p and returns the value of C at position p; $\texttt{eval}(E, C, \text{expr})$ is a predicate that holds when expression expr evaluates to the boolean true value in environment E with respect to C; $\texttt{neighbor}(C, \text{DIR}, p)$ is a function that computes, given a list of directions and a GBF C, all (defined) neighbors of a position p:

$$\texttt{neighbor}(C, \texttt{<u}_1\texttt{, ..., u}_n\texttt{>}, p) = \left\{p + \texttt{u}_i \mid 1 \leq i \leq n \text{ and } \exists v \text{ s.t. } (p + \texttt{u}_i, v) \in C\right\}$$

The definition of the derivatives is given by induction on the path pattern P and the GBF C:

$$\frac{\partial\, \text{id/expr}}{\partial p}(C, E) = \texttt{if eval}(E + [\text{id} \rightarrow p], C, \text{expr}) \texttt{ then } \left\{[p]\right\} \texttt{ else } \emptyset$$

$$\frac{\partial\, \text{id dir*}}{\partial p}(C, E) = \left\{[\,]\right\} \cup \frac{\partial\, \text{id dir id dir*}}{\partial p}(C, E)$$

$$\frac{\partial\, \text{id/expr dir P}}{\partial p}(C, E) = \texttt{let } E' = E + [\text{id} \rightarrow p] \texttt{ and } C' = C - (p, \texttt{val}(C, p))$$
$$\texttt{in if eval}(E', C, \text{expr})$$
$$\texttt{then} \bigcup_{p' \in \texttt{neighbor}(C,\, \text{dir},\, p)} \left(\left(\frac{\partial P}{\partial p'}(C', E') \right) \otimes p \right)$$
$$\texttt{else } \emptyset$$

$$\frac{\partial\, \text{id dir} *\ \text{dir}'\ P}{\partial p}(C, E) = \left\{[\,]\right\} \cup \frac{\partial\, \text{id dir id dir} *\ \text{dir}'\ P}{\partial p}(C, E)$$

The rule for the repetition $a\ \texttt{d*}$ simply specifies that the corresponding paths is a 0 length path or begins with an element matched by a and then follows direction d to match a path satisfying the same pattern $a\ \texttt{d*}$.

8 Conclusions

The array data structure is not smoothly handled in functional languages because they cannot be described convincingly as instances of an algebraic data type. Therefore, there are no means to specify by case 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 on trees and arrays in the same framework [GM01a]. The GBF approach put the emphasis on the logical neighborhood of the data structure elements [GM02]. 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 find all the paths matched by a pattern is given, inspired by the notion of derivative developed for the recognition of regular expression 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://www.lami.univ-evry/mgs`). This interpreter handles the examples proposed in section 6. Several other examples of the programming style allowed by **MGS** rules on GBF are developed in [GGMP02] in the context of biological simulations.

Pattern matching in arrays has been considered in the functional language community as back as [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 languages. 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 for the unions appearing in the definition of the derivatives, we can parameterize a strategy for the enumeration of paths. We are currently developing a pattern compiler for **MGS** based on pattern transformations.

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. Banatre and Daniel Le Metayer. 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 structures 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] Jean-Louis Giavitto and Olivier Michel. MGS: a rule-based programming language for complex objects and collections. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

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

[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] Jean-Pierre 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.