Declarative definition of group indexed data structures and approximation of their domains

Jean-Louis Giavitto
giavitto@lami.univ-evry.fr

Olivier Michel
michel@lami.univ-evry.fr

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

ABSTRACT

We introduce a new high-level programming abstraction which extends the concept of data collection. The new construct, called GBF (for Group Based Data-Field), is based on an algebra of index sets, called a *shape*, and a functional extension of the array type, the *field* type. Shape constructions are based on group theory and put the emphasis on the logical neighborhood of the data structure elements. A field is a function from a shape to some set of values. In this study, we focus on *regular* neighborhood structures and we show that arrays of any dimensions, cyclic array and trees are special kind of GBF.

The recursive definitions of a GBF are then studied and we provide some elements for an implementation and some computability results in the case of recursive definition.

Keywords

recursive definition of data-structures, data-field, Cayley graph, extension analysis

1. INTRODUCTION AND MOTIVATIONS: DATA STRUCTURE AS SPACES

In Haskell or CAML, or more generally in functional languages, the array type is very different in nature from the algebraic data types that can be specified by the programmer. For instance, there is no pattern for case-based function definition on an array. The reason is that there is no natural constructor for the array type. In contradiction with this fact, it is possible to define in a natural way the notion of *catamorphisms* [9] for the arrays types. Therefore, there is obviously a need for a unified formalism that enable the definition of such functions smoothly on both data structures. In this paper, we present a possible approach to answer this need in a declarative framework.

In [13] we have developed a general framework for the recursive definition of data structures. In this framework, we rely upon the following intuitive meaning of a data structure: a data structure s is an *organization* or an *arrangement* o performed on a data set D. It is customary to consider the pair $s = (o, D)$ and to say that s is a structure o of D (for instance a *list* of *int*, an *array* of *float*, etc.). However, we want to stress the structure o as a set of *places* or *positions*, independently of their occupation by elements of D. Following this perspective, a data structure in [13] is a function from a set of places to a set of values.

Now, we are interested to study various "set of places" independently of the set of values. For example, one of our motivations is to define in the same framework the set of places representing a tree or an array.

1.1 Data Structure and Elementary Moves

In order to separate sets places from values they contain, our main idea is to abstract *the data and computation* ***movements*** that occur within a data structure. The point of view is geometric: *a data structure can be seen as a space*, the set of places or positions between which the programmers, the computation and the values, move.

The notion of move relies on some notion of *neighborhood*: moving from one point to a neighbor point. Although speaking of *neighborhood* in a data structure is not usual, the relative accessibility from one element to another is a key point usually considered in a data structure. For example:

- In a simply linked list, the elements are accessed linearly (the second after the first, the third after the second, etc.).
- In a circular buffer, or in a double-linked list, computation goes from one element to the following *or* to the previous one.
- From a node in a tree, we can access the sons.
- The neighbors of a vertex V in a graph are visited after V when traveling through the graph.
- In a record, the various field are locally related and this localization can be named by an identifier.
- Neighborhood relationships between array elements are left *implicit* in the array data-structure. Implementing neighborhood on arrays relies on an index algebra: index computations are used to code the access to a neighbor.

For example $(i-1, j)$ is the index used to access the "north neighbor" of point (i, j) (we assume that the "north" direction is mapped to the first element of the index tuple). The standard example of index algebra is integer tuples with linear mappings $\lambda x.x \pm 1$ along each dimension (called "Von Neumann" or "Moore" neighborhoods). More than 99% of array references are affine functions of array indexes in scientific programs [11].

This list of examples can be continued to convince ourselves that a notion of *logical neighborhood* is fundamental in the definition of a data structure. The concept of logical neighborhood in a data structure is not only an abstraction perceived by the programmer and vanishing at the execution, but it does have an actual meaning for the computation. The computation indeed complies with the logical neighborhood structure of the elements. For example, the recursive definition of the `map` function on lists propagates an action to be performed from the head to the tail. More generally, recursive computations on data structure respect so often the logical neighborhood, that standard high-order functions can be automatically defined from the data structure organization (think about catamorphisms and others polytypic functions on inductive types [9, 24]).

1.2 Formalizing the Elementary Displacements in a Data Structure

Our goal is to *make the neighborhood definition explicit* by specifying several spatial elementary *moves* (we will call them equivalently *shifts* or *displacements*) to define the neighborhood for each element.

Such a structure of displacements will be called a *shape*. A shape is part of the type of a data structure type, like `[100]` is part of the `C` vector type `int [100]`. However, the shape embeds much more information than just a size.

What we want is to give a uniform description of the shapes appearing in various data structures focusing on the geometrical nature of a shape. The purpose is to enable the explicit representation and the reasoning on the data movements and to develop *a geometry of computation patterns*. The expected benefits are twofold:

- From the programmer's point of view, describing various shapes in a uniform manner enhances the language expressiveness and proposes a new programming style.
- From the implementor's point of view, a uniform handling of the shapes enables to reason on dependencies and data movements independently of the data structure.

In the following we restrict ourselves to *regular data structures*. A data structure is called *regular* if every element of the data structure has the same neighborhood structure (like for example a "right neighbor" and a "left neighbor"). The consequence of this assumption is examined below.

The Group Structure of Elementary Moves. To stress the analogy made between a data structure and a (discrete) space, we call *points* the elements of a data structure. Let "a", "b", "c", ... the directions taken on a point to go to the point's neighbors and let $P\!<\!a\!>$ be the "a" neighbor of a point P. One can think about a as the displacement from a point towards one of its neighbors (see Fig. 1). Displacement operations can be composed: using a multiplicative notation, we write $P\!<\!a.b\!>$ for $(P\!<\!a\!>)\!<\!b\!>$. Displacement composition is associative. We note e the null displacement, i.e. $P\!<\!e\!> = P$. Furthermore we will define a unique inverse displacement a^{-1} for each displacement a such that $P\!<\!a.a^{-1}\!> = P\!<\!a^{-1}.a\!> = P$.

In other words, the displacements constitute a *group* for the displacement composition, and the application of the displacements to points is the *action of the group over the data structure elements*.

1.3 Rationales of Using a Group Structure to Model the Displacements

The reader who follows our analogy between space and data structure may be surprised by the choice of a group structure to formalize the displacements. For instance, why choosing a group structure instead of a *monoid*? Another example, is the approach taken in [10], that rephrased in our perspective, uses a *regular language* to model the displacements resulting from following pointers in `C` data structures. The group structure seems to have two drawbacks:

1. A group structure implies inverse displacements. But in a simply linked list, if we know how to go from the head to the tail, we cannot go back from the tail to the head (else, the structure will be a doubly linked list).

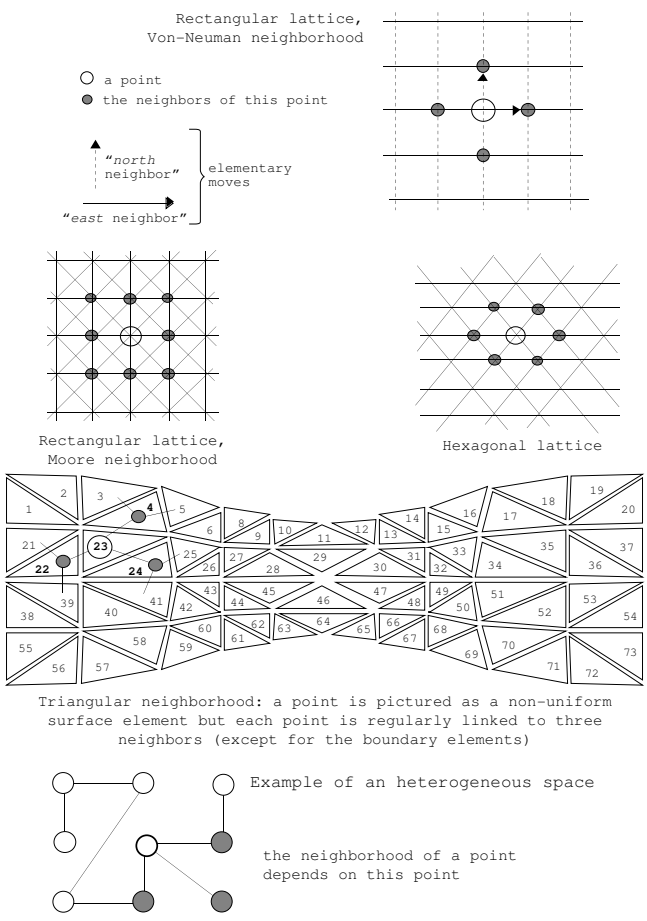

Figure 1: Four examples of regular spaces and one example of a non regular space.

2. The group structure implies regular displacement: each displacement must apply on every point (e.g. on every element of the data structure). This does not seem to be the case for trees for example, where a distinction is usually made between interior nodes (from which a displacement is possible) and leaves (which are dead ends).

The first remark relies implicitly on the idea that *all* the possible displacements are coded in some way in the data structure itself (e.g. by pointers). This is not the case: when reversing a simply linked list, the inverse displacement is represented in a stack which dynamically records from where the computation comes. This makes possible to access the previous cons cell although there is only a forward pointer. In a vector, accessing the element next to the element indexed by i is done by computing its index, e.g. $i+1$. The inverse of function $\lambda i.i+1$ can be computed given access to the previous element (and at the same cost).

The second remark outlines that the parts of a (recursive) data structure are generally not of the same kind and considering regular displacements is a rough approximation. However, consider more closely the case of a binary tree data type T defined by:

$$T = A \cup (B \times T \times T) \tag{1}$$

The interior nodes are valuated by elements of type B and the leaf by elements of type A. Intuitively, the corresponding displacements are g_l = "go to the left son" and g_r = "go to the right son" corresponding to the two occurrences of T on the right hand side of the equation (1). These two displacements cannot be applied to the leaves nodes. Now, note that in an updatable data structure, a leaf may be substituted to a sub-tree. So, from the shape point of view, which focuses on the geometry of the structure and not on the type of the elements, the organization of the elements is similar to a regular binary tree

$$T = C \times T \times T \tag{2}$$

where $C = A \cup B$. In a point valuated by A, applying a displacement g_l or g_r is an error. Errors are exceptional cases that derogate from the regular case. Checking at run time if the value is of type A or B to avoid an error is not different from checking if the node is of type A or $B \times T \times T$ (in languages like `ML`, this check is done through the dispatch mechanism of pattern matching the arguments of a function).

What we have lost between equation (1) and equation (2) is the relationship between the A type and the inapplicability of the displacement. But we have gained a regular description of the displacement structure.

To summarize the previous discussion, the idea is to embed an irregular structure into a regular one and to avoid some moves. In other words, the group structure does not overconstrain the elementary displacements that can be expressed. In addition, the group structure is sufficiently general and provides an adequate framework to unify data-structures like arrays and trees (Cf. sections 2.2 and 2.3).

1.4 The Representation of the Points

The first important decision we have made is to consider regular displacements. We have now to decide on what kind of sets operates the group of displacements.

Our idea is that the value of an element, or point, P may depend only on the value at the points reachable from P. That is to say, the value at a point may depend only on the value at the points of its orbit. The orbit of the point $P \in E$ under the action of the elements of the group G is the set $\{P<g>, g \in G\}$. The action of G on E is said to be *transitive* if all elements of E have the same orbit. If there are several distinct orbits, then the computation involved in these sub-data structures are *always* completely independent, and therefore, it is rather artificial to merge all these sub data structures into a bigger one.

This leads to considering a set of points on which the group of displacements acts transitively, which means that there is a possible path between any two points.

The simplest choice is to consider the group itself as this set of points and let

$$P<a> \;=\; P.a$$

as the group action on itself.

1.5 Collection, Data Field and Group Based Field

We have now all the necessary notions to define a data structure: informally, a data structure $\mathcal{D}$ associates a value of some type $\mathcal{V}$ to the element of a group G. The group G represents both the places of the data structure and the displacements allowed between these places.

In consequence, a data structure s is a partial function: $s \in S_G = G \to \mathcal{V}$ and a data structure type S_G is the set of partial functions from a given G to some set $\mathcal{V}$. Because the set G is a group, we call our model of data structures: **GBF** for *Group Based Field*.

Because we use *partial* functions, a concrete data structure represents a bounded domain in the space defined by its shape: the element of the shape with a definite image. For instance, this enable the representation of usual (bounded) arrays over an infinite grid defined by an abelian group.

The formalization of a data structure as a function is not new; it constitutes for instance, the basement of the theory of *data fields* [20] and is heavely used in [13]. In computer science, it is usual to think about a function as a rule to be performed in order to obtain a result starting from an argument. This is the *intensional* notion of functions studied for instance by the λ calculus. However, the current standard definition of a function in mathematics is a set of pairs relating the argument and the result. This representation is termed as *extensional* and is closer to the concept of a data structure. For example, an array tabulates the relationship between the set of indices and the array elements. So, we insist here that the view of data structures as functions is only logical and appears only at the level of the data structure definition. It does not assume anything on the data structure implementation.

Organization of the paper. The rest of this work is devoted to a first study of the consequences of considering a data structure under the geometric point of view of a group operating on itself. It can be conceived as a study in data field theory, where we have equipped the domain of the function with a group structure.

Shapes are defined in section 2. GBF and their operations are introduced in section 3.

In section 4 we consider the *recursive definition of GBF*.

A clear distinction is made between GBF and functions, so we do not accept any recursive definition scheme and we consider only recursions that propagate the computations along a natural displacement.

The implementation problems of recursive GBF are considered in section 6. The basis for an optimized implementation dedicated to *abelian GBF* are provided (a possible underlying parallel virtual machine is described in [14]).

The tabulation of the GBF values require the computation or the approximation of the definition domain. Some theoretical results are provided for this problem in section 7.

Finally, section 8 reviews some related works on data fields, collections and the representation of discrete spaces in computer science.

2. THE DEFINITION OF A SHAPE

Let the group G represent the set of all possible moves within a data structure. Furthermore, we characterize a subset $S \subset G$ of *elementary displacements*.

Let $Shape(G, S)$ denotes the directed graph having G as its set of vertices and $G \times S$ as its set of (directed) edges. For each edge $(g, s) \in G \times S$, the starting vertex is g and the target vertex is $g.s$. The *direction* or the *label* of edge (g, s) is s. Each element of the subgroup generated by S corresponds at the same time to a *path* (a succession of elementary displacements) and to a point: the point reached starting from the identity point e of G and following this path:

$$e<P> = P<e> = P$$

(from here we use $P.s$ instead of $P<s>$ for the s neighbor of P). In other words, $Shape(G, S)$ is a graph where:

1. each vertex represents a group element,
2. an edge labeled s is between the nodes P and Q if $P.s = Q$, and
3. the labels of the edges are in S.

This graph is called a *Cayley graph*. The following dictionary, illustrated in figure 2, gives the translation between graph theory and group related concepts:

Cayley graphs		*Groups*
vertex	$\leftrightarrow$	group element
labeled edge	$\leftrightarrow$	generator
path composition	$\leftrightarrow$	word multiplication
closed path (cycle)	$\leftrightarrow$	word equating to e
connexity	$\leftrightarrow$	solvability of $P.x = Q$

We can state some properties that link the global structure of $Shape(G, S)$ and the relations between G and S. Let us say that S is a *basis* of G if an element of G is a product of elements of S. Let $S^{-1} = \{s^{-1}, s \in S\}$. We say that S *generates* G if $S \cup S^{-1}$ is a basis of G (this terminology is not standard). Then, the following facts are well known:

- For $Shape(G, S)$ to be connected, it is necessary and sufficient that S generates G. The connected components of $Shape(G, S)$ are the cosets $g.H$ where H is the subgroup generated by S (a coset $g.H$ is the set $\{g.h : h \in H\}$).
- For $Shape(G, S)$ to contain a loop (a directed cycle of length 1), it is necessary and sufficient that e belongs to S.
- A circuit is a directed cycle. $Shape(G, S)$ has no circuit of length ≤ 2, if and only if $S \cap S^{-1} = \emptyset$.

In the following, we restrict ourselves to the case where the subset S generates G. Usually the name *Cayley graph* for $Shape(G, S)$ is used if S is a basis of G. If S is not a basis of G, $Shape(G, S)$ is a subgraph of the Cayley graph of G. Note that there exist *regular connected graphs*, i.e., graphs where each vertex has the same number of adjacent nodes, which are not the Cayley graphs of a group [30].

2.1 Specification of a Shape by a Presentation

What we want is to specify $Shape(G, S)$, that is, the group G and the generator set S, in an abstract manner.

We use a finite *presentation* to specify the group. A finite presentation gives a finite set of group generators and a finite set of equations constraining the equality of two words. An equation takes the following form: $v = w$ where v and w are products of generators and their inverses.

The presentation of a group is given between enclosing $\langle\!|$ and $|\!\rangle$:

$$\langle\!| \, g_1, \ldots, g_d \, ; \; w_1 = w'_1, \ldots, w_p = w'_p \, |\!\rangle$$

where g_i are the generators of the presentation and $w_j = w'_j$ are the equations. A *free group* is a group without equation.

We associate to a presentation $G = \langle\!| \, S \, ; \, \ldots \, |\!\rangle$ the shape $Shape(G, S)$. So the generators in the presentation are the distinguished group elements representing the elementary displacements from a point towards its neighbors in a shape.

In the following, a presentation denotes the corresponding shape or the underlying group following the context.

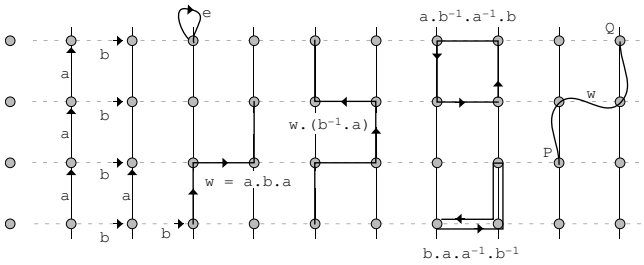

Figure 2: Graphical representation of the relationships between Cayley graphs and group theory. A vertex is a group element. The label a of an edge corresponds to the generator a of the group. There is an edge between vertices P and Q labelled by a iff $P.a = Q$. A word (a product of generators) can be seen a path. Starting from vertex P, a path w ends in $P.w$. Path composition corresponds to word multiplication. A closed path (a cycle) is a word equal to e (the identity of the multiplication). An equation $v = w$ can be rewritten $v.w^{-1} = e$ 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 $b.a.a^{-1}.b^{-1}$) and closed paths specific to the own group equations (e.g.: $a.b^{-1}.a^{-1}.b$). The graph connexity (there is always a path going from P to Q) is equivalent to say that there is always a solution x to equation $P.x = Q$.

2.2 Examples of Abelian Shapes

Abelian groups are groups with a commutative law (that is, the product of two generators commutes). Abelian groups are of special interest and we specifically use the $\langle\ \rangle$ brackets for the presentation of abelian groups, skipping the commutation equations as they are implicitly declared.

For example,

$$G2 \quad = \quad \langle North,\ East,\ West,\ South;\\ South = North^{-1},\ West = East^{-1} \rangle$$

Because the last two equations, *South* and *West* are aliases for the inverses of *North* and *East* and only two generators are necessary to enumerate the group element. The corresponding abstract group can be presented without equation by

$$G2' \ = \langle\ North,\ East\ \rangle$$

and therefore, is a free group. These shapes correspond to an infinite NEWS grid. The difference between *G2* and *G2'* is that in the shape *G2*, two adjacent nodes are linked by an edge and its opposite (the grid is "bidirectional"), while in the shape *G2*', there is only one edge between two neighbors.

Here is another example that shows that the effect of adding an equation to a presentation is to identify some points. We start from the free abelian group with one generator: $\langle\ a\ \rangle$ that describes a discrete line. If we add the equation $a^N = e$, the presentation becomes:

$$\langle a\ ;\ a^N = e \rangle$$

which specifies a cyclic group of order N. The shape can be pictured by the discretization of a circle where N is the number of points of the discretization. Along the circle, we can always move in the same direction a and after N a-moves, we are back to the starting position. The points $\{a^{k.N}, k \in \mathbb{Z}\}$ are all identified with the point e.

Since arrays (like PASCAL arrays) are essentially finite grids, our definition of group-based fields naturally embeds the usual concept of array as the special case of a bounded region over a free abelian shape. For example, multidimensional LUCID fields, systolic arrays, Lisper's data-fields [21] and even lazy lists, fit into this framework. Furthermore, this allows the reuse of most of the achievements in the implementations of arrays (e.g. [8, 28]) to implement (bounded regions over) infinite abelian fields, and with some additional work, to adapt them to the handling of finite abelian fields.

2.3 An Example of a Non Abelian Shape

Abelian groups are an important but special case of groups. We give here one significant example of a non abelian shape.

The first example is simply a free group. The free non abelian shape:

$$F2 = \langle\!\langle\ x, y\ \rangle\!\rangle$$

is pictured in Fig. 3. We see that the corresponding shape can be pictured as a tree (i.e. a connected non-empty graph without circuit). Actually, there is a general result stating that if $Shape(G, S)$ is a tree, then G is a free group generated by S.

This enables the embedding of some class of trees in our framework. Let $Shape(G, S)$ where G is a free group and S is a minimal set of generators, i.e. no proper subset of S generates G. Then $Shape(G, S)$ is a tree. Observe that this tree has no node without predecessor. This situation is unusual in computer science where (infinite) trees have a root and "grow" by the leaves, but this graph embeds any finite binary tree by rooting them at some point. Figure 3.b gives an illustration of the points accessed starting from a point w in *F2*: it is a binary tree with root w. We cannot find a unique generator acting as the father accessor (for node $w.x$, the father accessor is x^{-1}, while it is y^{-1} for the node $w.y$).

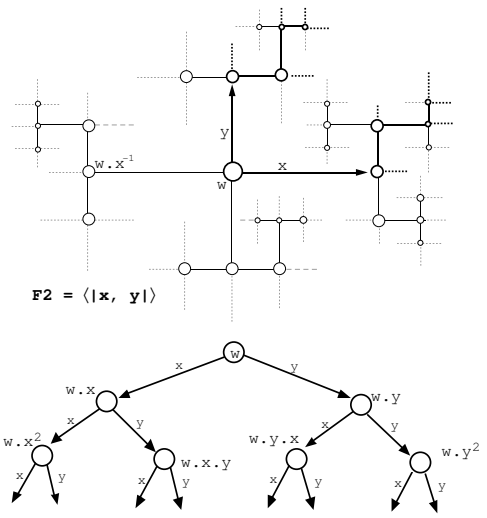

Figure 3: A free non abelian group with two generators. Bold lines correspond to the points that can be reached starting from a point w and following the elementary displacements x and y.

3. GROUP BASED FIELDS

A group based field (or field in short) is a partial function from a shape to some values set. The elements of the shape with a well defined value are called the index set of the GBF. If $g : F \to \mathcal{V}$, we write $g[F]$ to specify that g is a GBF on shape F and $g(x)$ denotes the value of g at point $x \in F$.

Because a shape F is simply a graph, a GBF is a function over the vertices of this graph. The supplementary structure of the graph is used to specify automatically some operations that are available on a GBF over F.

Operations defined on fields are intensional. We present three kinds of GBF expressions: extensions of scalar functions, geometric operations and reductions.

These operations are given as a first account to show how a rich algebra of shape parameterized operations can be introduced on GBF. In addition, all these operations have a data parallel interpretation because they lead to manage GBF as a whole.

3.1 Extension

Extension of a scalar function is just the point-wise application of the function to the value of a field at each point. So, if F has shape G, $f(F)$ denotes the field of shape G which has value $f(F(w))$ for each point $w \in G$. Similarly, n-ary scalar functions are extended over fields with the same shape.

3.2 Geometric operations

A geometric operation on a collection consists in rearranging the collection values or in selecting some part of the collection to build a new one.

Translation. The first geometric operation is the translation of the field values along the displacement specified by a generator: $F.a$ where $a \in S$. The shape of $F.a$ is the shape of F. The value of $F.a$ at point w is $(F.a)(w) = F(w.a)$. When the field F is non-abelian, it is necessary to define another operation $a.F$ specified as: $(a.F)(w) = F(a.w)$. Obviously, this definition extends to the case where $a \notin S$: if $u = a_1.\ldots.a_n, a_i \in S$, then $(F.u)(w) = ((\ldots(F.a_1).\ldots).a_n)(w) = F(w.u)$.

Direct Product. Several group constructions enable the construction of a group from previous ones. We just mention the *direct product* of two groups that gives rise to the direct product of two fields: $F_1[G_1] \times_h F_2[G_2]$. Its shape is the direct product $G_1 \times G_2 = \{(u_1, u_2) : u_1 \in G_1, u_2 \in G_2\}$ equipped with multiplication $(u_1, u_2).(v_1, v_2) = (u_1.v_1, u_2.v_2)$. The value of the direct product $F_1 \times_h F_2$ at point (u, v) is $h(F_1(u), F_2(v))$. This operation corresponds to the *outer product* on vector space.

Restriction and Asymmetric Union. We say that a shape $F = \mathit{Shape}(G, S)$ is infinite if G is not a finite set. Only the values of a field on a finite set are practically computable. This raises the problem of specifying the parts of a field where the field values have to be computed. Our approach is similar to the one of B. Lisper for data fields on $\mathbb{Z}^n$: we introduce an operation of *restriction* that specifies the domain of a field.

The restriction $g|p$ of a field g by a boolean valuated field p, specifies a field undefined for the point x where $p(x)$ is false. For the point x where $p(x)$ is true, the restriction coincides with g. We define also the restriction of a field g to a coset C: $g|C$ where $C = u.H$. The result is a GBF of shape H such that $(g|C)(x) = g(u^{-1}.x)$.

It is convenient to introduce simultaneously to the restriction, an operator for *asymmetric union*: $(f\#g)(x) = f(x)$ if f has a defined value at point x and $g(x)$ elsewhere.

Remark. In [14], we do not admit any predicate p but we restrict to expressions corresponding to some simple domains with good properties: the points of such a domain can be enumerated, and predicate expressions are closed for domain intersection.

Translation, restriction and asymmetric union of such domains are the basis of the implementation of data fields on $\mathbb{Z}^n$ studied in [14, 7].

3.3 Reductions

Reduction of a n-dimensional array in APL is parameterized by the *axis* of the operation [16] (e.g. a matrix can be reduced by row or by column). The projection of the array shape along the axis is another shape, of dimension n-1, and this shape is the shape of the reduction. We generalize this situation in the following way (consider Fig. 4).

Normal Subgroup and Quotient Group. Let H be a subgroup of G, specified by its set of generators S'; we write $H = S' : G$. H will be the axis of the reduction.

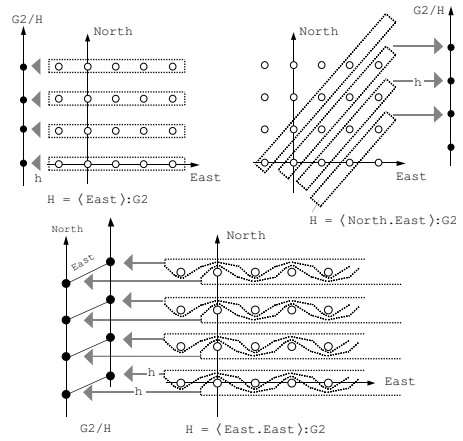

Figure 4: Three examples of reduction over the *G2* shape.

For $u, v \in G$, we define the relation $u \equiv_H v$ if there exists $x \in H$ such that $u.x = v$. Let the quotient of G by H, denoted by G/H, be the equivalence classes of $\equiv_H$. An element w of G/H is the set $u.H$ where u is any element in w.

We need to ensure that G/H is a group. This is always possible, through a standard construction, *if* we assume that H is a *normal subgroup* of G, that is, for each $x \in G$, $x.H = H.x$ (for an abelian group, any subgroup is normal). Then, a possible presentation of G/H is the presentation of G augmented by the set of equations $\{g = e,\ g \in S'\}$.

The Reduction. The expression $h\backslash H\ F$ denotes the reduction of a field $F[G]$ following the axis H and using a combining function h.

It is assumed that H is a normal subgroup of G and that h is a commutative and associative binary function. The shape of $h\backslash H\ F$ is G/H. The value of $h\backslash H\ F$ on a point $w \in G/H$ is the reduction of $\{F(v) : v \in w\}$ by h (this set has no canonical order, this is why we impose the commutativity of h).

See figure 4 for some examples of reductions over the *G2* shape. Only the first example can be expressed in APL. An interesting point is that H is not restricted to be generated by only one generator; as an example, $+\backslash G\ F$ where G is the shape of F computes the global sum of all elements in G (G is always normal in itself).

Remark. As usual in data fields, there is a problem with the handling of reductions over an infinite domain. The idea is that undefined values are not taken into account. So $h\backslash H\ (g|p)$ is defined even if G is infinite, if the set $\{x, p(x) = \mathit{true}\}$ is finite.

4. RECURSIVE DEFINITION OF A GBF

We can see scan operations [5], or catamorphisms and their variations, as computations propagating along the data structure neighborhood. The recursive definition of a GBF, introduced in the next section, is then a possible generalization of such operations.

Here *declarative definitions* of GBF are considered. So

we restrict to recursive definitions of GBF *preserving the neighborhood relationships*. This kind of GBF specification induces computation flowing from a point to the neighbor points, in a way reminiscent from the systolic computation paradigm.

Let $g[F]$ be a GBF such that $F = \mathit{Shape}(G, \{s_1, \ldots, s_n\})$. If g *complies with the elementary neighborhood* specified by F, then the value of g on a point x depends only on the value of g at points $x.s_i$ via a fixed function h. That is

$$\exists h,\ \forall x \in G, \quad g(x) \ = \ h(g(x.s_1), \ldots, g(x.s_n)) \tag{3}$$

where h is a scalar function that establishes the functional relationship between the value at a point and the values at its neighbors.

Equation (3) holds for all $x \in G$ so we make that implicit and write

$$g[F] = h(g.s_1, \ldots, g.s_n) \tag{4}$$

(the generators $s_1, \ldots, s_n$ appearing in the equation are not always sufficient to infer the shape of g, for instance in $g = 0$; this is why we may explicitly indicate $[F]$). This equation is a functional equation between GBF and not between values. The GBF g is said to be recursively defined or simply a "recursive GBF". An example is given in Fig. 5.

Quantification of Definitions. Obviously equation (4) is a kind of recursive definition and we need some "base case" to stop the recursion. So, we introduce *quantified definitions*; the two equations:

$$g@C \ = \ 0 \tag{5}$$
$$g[F] \ = \ 1 + g.d \tag{6}$$

define a GBF g on shape F. The equation (5) specifies the value of $g(x)$ on a point $x \in C$. In our example, the value of g on C is 0. For point $x \notin C$, the equation (6) is used and $g(x) = (1 + g.d)(x)$.

We say that equation (5) is *quantified* and that equation (6) is the *default equation*. It is the set of these two equations that makes the definition of g.

Using quantified definitions do not enhance the expressive power of recursive GBF. Indeed, equations (5+6) are equivalent to

$$g[F] \ = \ (0 \,|\, C) \,\#\, (1 + g.d)$$

Coset Quantified Definition. The problem is to specify the kinds of domains we admit for the expression of C. Ideally, we would make a partition of the shape and define the field giving an equation for each element of the partition. It implies that each element of the partition can be viewed as a shape itself. We may use subgroups of the initial group to split the initial domain, but this is somewhat too restrictive, thus we will use *cosets*.

A coset $g.H = \{g.h, h \in H\}$ is the "translation" by g of the subgroup H. In a non-abelian group, we distinguish the right coset $g.H$ and the left coset $H.g$. To specify a coset we give the word g and the subgroup H. The notation $\{g_1, g_2, \ldots, g_p\} : G$ defines a subgroup of G generated by $\{g_1, g_2, \ldots, g_p\}$ (the g_i are words of G). There is no specific equation linking the generators of the subgroup but they are subject to the equations of the enclosing group, if applicable.

Well formed shape partitions. The intersection of two cosets is empty or a coset. For that reason, in a coset quantified definition like

$$\begin{cases} g@C_1 = \ldots \\ \quad \ldots \\ g@C_n = \ldots \\ g[G] = \ldots \end{cases} \tag{7}$$

there are ambiguities in the definition of g if $C_i \cap C_j \neq \emptyset$ for $i \neq j$. To avoid these ambiguities, we suppose that if $C_i \cap C_j \neq \emptyset$ for $i \neq j$, then there exists k such that $C_i \cap C_j = C_k$. That is, the set $\{C_i\}$ is closed for the intersection. Then, the value of g on a point $x \in C_i$ is defined by the equation corresponding to the smallest C_k containing x.

Remarks:

- Note that the set of points where the default definition applies is not a coset but the complement of a union of cosets.
- The ambiguities involved by multiple cosets quantification is similar to the ambiguities involved by the definition of a function through overlapping patterns. For instance, in the following `ML`-like function definition

  ```
  let f = function (true, _) -> 0 | (_,_) -> 1
  ```

  the value of `f(true, true)` is either `0` or `1`. An additional rule giving the precedence to the first pattern that matches in the pattern list, is used to fix the ambiguity. The rule of cosets inclusion is used in the case of GBF, but a rule based on the definition order can be used if checking the inclusion of cosets has to be avoided.
- The form (4) extends obviously to handle arbitrary translation. This does not contradict the neighborhood compliance because the introduction of intermediate fields recovers the locality. For example,

  $$g = 1 + g.d^3$$

  can be rewritten as

  $$\begin{cases} g' = g.d \\ g'' = g'.d \\ g = 1 + g''.d \end{cases}$$

5. A DENOTATIONAL SEMANTICS FOR RECURSIVE GBF DEFINITIONS

As a matter of fact, a GBF is a function. Then, the semantics of a system of recursive equations defining a set of GBF is the same as the semantics of a system of recursive equations defining functions in the framework of denotational semantics [29].

Let $\mathcal{F}$ be the Scott domain of functions over a group F. The recursive expression $g[F] = \varphi(g)$ defines a *continuous* operator φ on $\mathcal{F}$, because φ is a composition of continuous operators like: translation, restriction, asymmetric union and extension of continuous functions. Therefore, solutions of $g[F] = \varphi(g)$ exist and are called fixpoints of φ. The least fixed point of φ can be computed by fixpoint iteration from $\lambda x.\ \bot$ and is the limit of $\varphi^n(\lambda x.\ \bot)$ when n goes to infinity.

Computability. An immediate question is to know if the fixpoint iteration converges on a point in a finite number of steps. For general functions this amounts to solve the halting problem but here we are restricted to group based fields. However, the expressive power of group based fields is enough to confront to the same problem: suppose a field defined by:

$$g[F] = h(g.a, g.b, \dots)$$

the points accessed for the computation of the value of w are: $w.a, w.b, \dots, w.a.a, w.a.b, \dots$. As a matter of fact, if the computation of a field value on a point w depends on itself, the fixpoint iteration cannot converge; so we face the problem of deciding if $w.a = w$, $w.b = w$, $\dots, w.a.b = w$, etc. That is to say, we have to decide if two words in a finite presentation represent the same group element. This problem is known as the *word problem for groups* and is not decidable (but it is decidable for finitely presented abelian groups, free groups and some other interesting families).

An Example. A possible program for a field on a one-dimensional line, where the value at a point increases by one between two neighbors, is:

$$\begin{aligned} G1 &= \langle left \rangle && (8)\\ A &= left^2.(\langle\rangle : G1) && (9)\\ iota@A &= 0 && (10)\\ iota[G1] &= 1 + iota.left && (11) \end{aligned}$$

Equation (8) defines a one-dimensional, one-directional line. Equation (9) defines the coset $A = \{left^2\}$ because the subgroup $\langle\rangle : G1$ is reduced to $\{e\}$ by convention. Equation (10) specifies that the field *iota* has the value 0 for each point of coset A and equation (11) is valid for the remaining points.

To define a field *iota* with the value 0 fixed at the point e, we set "$iota@\langle\rangle = 0$" instead of (10). We write $\langle\rangle$ for $e.(\langle\rangle : G1)$ because a subgroup H is also the coset $e.H$ and because here, after $iota@$, $\langle\rangle$ denotes necessarily a subgroup of $G1$.

The previous equations for *iota* define a function over $G1$ that can be specified in a `ML`-like style as:

```
let rec iota(left^n) = if n == 2 then 0
                       else 1 + iota(left^(n+1));;
```

This function has a defined value for the points $\{left^n, n \leq 2\}$ and the value $\perp$ for the other points. Note that the use of a displacement a instead of a^{-1} is mainly a convention.

6. IMPLEMENTING THE COMPUTATION OF A RECURSIVE GBF

For the sake of simplicity, we suppose that field definitions take the following form:

$$\begin{cases} g@C_1 = c_1 \\ \quad \dots \\ g@C_n = c_n \\ g[G] = h(g.r_1,\ g.r_2,\ \dots,\ g.r_p) \end{cases}$$

where C_i are cosets, c_i are constants and h is some extension of a scalar function. The set $\mathrm{R}_g = \{r_1, \dots, r_p\}$ is called the dependency set of g.

We assume the existence of a mechanism for ordering the cosets and to establish if a given word $w \in G$ belongs to some coset. We also suppose that we have a mechanism to decide if two words are equal. For example, these mechanisms exist for free groups and for abelian groups. There is no general algorithm to decide word equality in any non-abelian groups. So our proposal is that non abelian shapes are part of a library and come equipped with the requested mechanisms. A future work is then to develop useful families of (non abelian) shapes.

With these restrictions, a first strategy to tabulate the field values is the use of memoized functions. A field $g[G]$ is stored as a dictionary with entries $w \in G$ associated to values $g(w)$. If the value $g(w)$ of w is required, we first check if w is in the dictionary (this is possible because we have a mechanism to check word equality). If not, we have to decide which definition applies, that is, if w belongs to some C_i or not. In the first case, we finish returning c_i and storing (w, c_i) in the dictionary. In the second case, we have to compute the value of g at points $w.r_1, \dots, w.r_p$, (that is recursion) and then the results are combined by h.

Optimization when a Word Normal Form Exists. We can do better if each word w can be reduced to a normal form $\overline{w}$. A normal form can be computed for abelian groups (the Smith Normal Form) or for free groups. In this case, the dictionary can be optimized into an hash-table with key $\overline{w}$ for w.

Implementation of Recursive Abelian GBF. In the case of an abelian group G, we can even improve the implementation using the fundamental isomorphism between G and a product of $\mathbb{Z}$-modules, see [6, 15]. As a matter of fact, a function over a $\mathbb{Z}$-module is simply implemented as a vector. The difficulty here is to handle the case of $\mathbb{Z}^n$ which corresponds to an unbounded array. The computation and implementation of data fields over $\mathbb{Z}^n$ is studied in [22, 12, 14].

7. APPROXIMATION OF THE DOMAIN OF A RECURSIVE GBF

The algorithm presented in section 6 corresponds to a *demand-driven evaluation strategy*. For example, to evaluate $iota(e)$, we have to compute $iota(left)$ which triggers the computation of $iota(left^2)$ which returns 0. So, there is a dependency between the computation of $iota(e)$ and $iota(left)$ that can be pictured by a dependency between e and $left$.

More generally, for a definition $g[G] = h(g.r_1, \dots)$ we can associate to each point $w \in G$ a set $\mathcal{P}_w$ of directed paths corresponding to the points visited to compute $g(w)$. An element p of $\mathcal{P}_w$ is a word of the subgroup generated by $\mathrm{R}_g = \{r_1, \dots\}$ (the converse is not true). These notions are illustrated in figure 5.

The evaluation of $g(w)$ fails if some $p \in \mathcal{P}_w$ has an infinite length. Two cases can arise:

- p is cyclic;
- p has an infinite number of distinct vertices.

Bounding the number of vertices in a computation path is similar to the "stack overflow" limit. Static analysis can be used to characterize the domains of G with finite paths

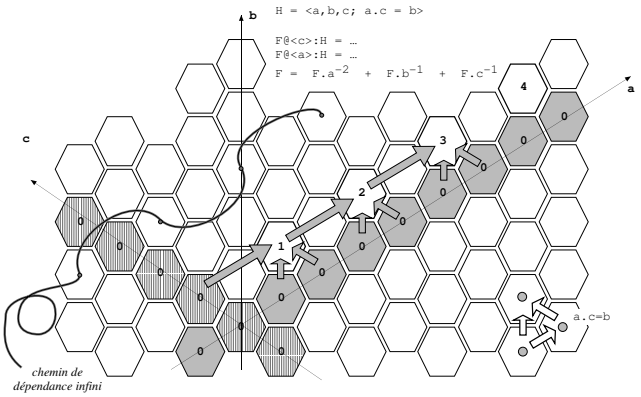

Figure 5: This schema figures a GBF based on an hexagonal shape $H = \langle a, b, c\,;\ b = a.c\rangle$. The field F is defined by a recursive expression. The cosets $\langle a\rangle : H$ and $\langle c\rangle : H$ are the base case of the recursion. The dependency set is $\mathrm{R}_F = \{a^{-2}, b^{-1}, c^{-1}\}$. The integer that appears in a cell corresponds to the maximal length of a dependency path starting from the cell and reaching a coset. This integer can be thought as the early instant where the cell value can be produced (in a free schedule). The arrows picture the inverse of a dependency: this translation can be used to compute new points value starting from known points. In this example, only one value can be produced at each time. The cells that have a value different from $\perp$ are in bold: they correspond to the *definition domain* of F. The infinite path that starts from one cell shows the beginning of an infinite dependency path: this path "jump" over the cosets and goes to infinity, that is, the starting cell does not have a defined value.

(Cf. [21] for a study in this line). Sufficient conditions can also be checked at compile-time to detect cyclic paths (e.g. a raw criterion can be $\mathrm{R}_g \cap \mathrm{R}_g^{-1} = \emptyset$) and/or it can be detected at run-time using an occur-check mechanism.

The development of a *data driven* evaluation strategy, or the development of some optimizations in the computation of a GBF, require the computation or the characterization of the *definition domain* of a recursive GBF. The definition domain of a GBF is the set of points w such that $\forall p \in \mathcal{P}_w$, p is finite (the set $\mathcal{P}_w$ is finite if each of its element is finite, because on any point w there is only a finite number of neighbors that can be used to continue a path).

In the rest of this section, we give some results about this problem.

7.1 Computability

The decidability of the word problem is not a sufficient condition to decide if a GBF g has a defined value on point x. For instance, in $\mathbb{Z}^n$ where the word problem is decidable (Cf. section 6), the problem of deciding if a GBF g defined by an equation of form (4) has a defined value on a point x is still undecidable.

Informally, the example of *iota* shows that some kind of primitive recursion is implementable in the GBF formalism. The equations $g[\langle right\rangle] = if\,p\ then\ c\ else\ g.right$ shows that some kind of minimization is also possible. Thus, intuitively, arithmetic functions can be coded in the GBF formalism. Note that for minimization, we use a conditional which is the extension of a non strict function.

Note also that for finite groups this problem is decidable because it is sufficient to explicitly compute the dependency graph between the group elements. This graph is finite and it is sufficient to check for the absence of cycle.

7.2 Approximation of the Definition Domain of a strict GBF

We call *strict GBF* a recursive GBF g specified by case on cosets but without using restriction, asymmetric union and with the help of only *strict functions* h in right hand side of equation (4). The computability of a strict GBF does not become a trivial problem. We give here some results on the approximation of the definition domain of a strict GBF g defined by

$$\begin{cases} g@C_1 = c_1 \\ \quad\cdots \\ g@C_p = c_p \\ g[G] = h(g.r_1, \ldots, g.r_q) \end{cases} \tag{12}$$

where h is a strict function. Let:

$$\mathrm{R}_g = \{r_1, \ldots, r_q\} \qquad \text{and} \qquad \mathrm{D}_0 = \bigcup_j C_j$$

In the sequel, we reserve the index j to enumerate the cosets C_j and the index i to enumerate the shifts r_i.

We know that the solution g of equation (12) is the least fixpoint of φ defined by:

$$\varphi(f) = \lambda x.\ \text{if } x \in C_i \text{ then } c_i \text{ else } h(f.r_1, \ldots, f.r_q)$$

$Def(\mathrm{g})$ denotes the definition domain of g. As a matter of fact, $\varphi(f)(x)$ is defined if $x \in \mathrm{D}_0$. Because h is strict, if $x \notin \mathrm{D}_0$ then $x \in Def(g) \Rightarrow x.r_i \in Def(g)$. That is, the definition domain $Def(g)$ is the least solution (for the inclusion order) of equation

$$D \ = \ \mathrm{D}_0 \ \cup \bigcap_i (D/\mathrm{D}_0).r_i \tag{13}$$

where $D/\mathrm{D}_0 = \{x \text{ such that } x \in D \wedge x \notin \mathrm{D}_0\}$.

7.2.1 *The Lower Approximation* D_n

The solution g is the limit of the sequence $g_n = \varphi^n(\lambda x.\perp)$. If $x \in Def(g_n)$, then we have two possibilities: $x \in \mathrm{D}_0$ or $x.r_i \in Def(g_{n-1})$ because h is a strict function. In the last case, it means that $x \in Def(g_{n-1}).r_i^{-1}$.

Suppose that the domain of g_{n-1} is a set D_{n-1}. We can propagate the value to $D_{n-1}.r_i^{-1}$ and because of the strictness of h we need to satisfy all the dependencies r_i. Thus, we may compute new values on the set $\bigcap_i D_{n-1}.r_i^{-1}$.

We then obtain the definition domain of g as the limit D_∞ of the sequence:

$$\mathrm{D}_0 = \bigcup_j C_j \tag{14}$$

$$\mathrm{D}_{n+1} = \mathrm{D}_n \ \cup \bigcap_i \mathrm{D}_n.r_i^{-1} \tag{15}$$

Starting from the definition of D_n we have immediately:

$$\mathrm{D}_0 \ \subseteq \ \mathrm{D}_1 \ \subseteq \ \ldots \ \subseteq \ \mathrm{D}_n \ \subseteq \ \ldots \ \subseteq \ \mathrm{D}_\infty = Def(g) \tag{16}$$

Therefore, the sequence D_n gives a lower approximation of $Def(g)$.

7.2.2 The Greater Approximation E_n

A Geometric Interpretation. To obtain a greater approximation of $Def(g)$, we first interpret geometrically the property of belonging to the definition domain of g. To each point $w \in G$ we associate a set $\mathcal{P}_w$ of directed paths corresponding to the points visited for the computation of $g(w)$. An element p of $\mathcal{P}_w$ is a word of the *monoid* $\mathcal{R}_g$ generated by R_g:

$$\mathcal{R}_g = \{\ r_{i_1}^{\alpha_{i_1}} . .... . r_{i_k}^{\alpha_{i_k}},\ \text{with } r_{i_l} \in R_g \text{ and } \alpha_{i_l} \in \mathbb{N}\ \}$$

The computation of $g(w)$ fails if there exists a $p \in \mathcal{P}_w$ with an infinite length. We have already noted that there are two classes of infinite path: cyclic paths and the others.

Computing a Greater Approximation E_0. If $g(w)$ is defined, then all the paths $p \in \mathcal{P}_w$ starting from w must end on a coset C_j. Amongst all these paths, there are some paths made only with r_i shifts. Let:

$$R_i = \{\ r_i^{-n}, n \in \mathbb{N}\ \} \tag{17}$$
$$E_0 = D_0 \cup \bigcap_i D_0.R_i$$

The set R_i is the monoid generated by r_i^{-1} (*warning:* we take the inverse of the dependency). The set E_0 is made of the points $w \in G$ that either belong to D_0 or are such that there exists a path made only from r_i starting from w and reaching D_0. This last property is simply expressed as: $\forall i, \exists n_i,\ w.r_i^{-n_i} \in D_0$. This property is true for all $w \in Def(g)$ and then:

$$Def(g) \subseteq E_0 \quad .$$

Refining the Approximation E_0. The greater approximation E_0 is a little rude. We can refine them on the basis of the following remark. If $w \in Def(g)$, then we have either $w \in D_0$ or $w.r_i \in Def(g)$. We can deduce that:

$$Def(g) \ \subseteq\ E_1 = D_0 \cup (E_0 \cap \bigcap_i E_0.r_i)$$

Obviously $E_1 \subseteq E_0$. Moreover, this construction starting from E_0 can be iterated, which introduces the sequence

$$E_0 = D_0 \cup \bigcap_i D_0.R_i \tag{18}$$
$$E_{n+1} = D_0 \cup (E_n \cap \bigcap_i E_n.r_i) \tag{19}$$

We always have $Def(g) \subseteq E_{n+1} \subseteq E_n$.

Let E_∞ be the limit of E_n. For each $w \in E_\infty$, we have either $w \in D_0$ or $w.r_i \in E_\infty$. Therefore, E_∞ is a solution of the equation (13). It should be checked that it is the *least* solution which we admit (intuitively, the element of G are equivalence classes of *finite words* of generators and then, if $x \in E_\infty$ it can be checked by induction on the number of occurrences of r_i in x that $x \in Def(g)$).

7.3 Summary and a Conjecture

We can summarize the previous results by the formula:

$$D_0 \subseteq ... \subseteq D_n \subseteq ... \subseteq D_\infty = Def(g) = E_\infty \subseteq ...$$
$$... \subseteq E_n \subseteq ... \subseteq E_0 \tag{20}$$

These results hold for any strict GBF (abelian or non abelian). The recursive definition of a GBF $g[G]$ can be generalized without difficulty by considering more general base case domains. That is, we may replace the coset C_i by arbitrary set S_i in equation (7). Relations (20) remain true.

A *monoid* M generated by element $g_1, ..., g_p$ of a group G is the set of elements that can be written as product of positive powers of the g_i's. We call *comonoid* the translation of a monoid, that is, a set $x.M = \{x.m, m \in M\}$ where M is a monoid. For all the examples we have worked out on $\mathbb{Z}^n$, we have verified that the definition domain of a GBF g is a *finite union of comonoids.* We conjecture that this is always true.

8. CONCLUSION

This paper reports some efforts to extend the concept of collection, in the line of [13], by specifying a structure for an index set and independently of the element's value.

Considering a data structure independently of its underlying set is interesting for many purposes. For instance, this is the essence of the approach taken in the theory of *species of structures* [3] for combinatorial enumeration. The approach is functorial, which is also the case in [17], where B. Jay develops a concept of *shape polymorphism.* In his point of view, a data structure is also a pair (*shape*, *set of data*). As above, the shape describes the *organization* of the data structure and the set of data describes the *content* of the data structure. However, his main concern is the development of *shape-polymorphic functions* and their typing. Examples of shape polymorphic functions are the generalized `map` or the generalized `scan`, that can be computed without changing the data structure organization. More generally, the shape of the result of a shape-polymorphic function application depends only on the shape of the argument, not of its content.

The framework presented here unifies the tree and the array data structures. There is a number of researches to extend the concept of array: Indexical Lucid [1], Infidel [27], AMR++ [2]. These approaches consider more general shapes for arrays than n-dimensional bounding box, but always rely on grids (that is, a point is indexed by a tuple of integers). This forbids for example the natural representation of a tree or a triangular lattice. Cellular automata on a Cayley graph have been studied by [25] but no field algebra is worked out and the problem of recursive definition is out of their scope.

Here, we propose to consider a collection as a partial function over a finitely presented group. This approach makes the definition of point neighborhood explicit and gives a very rich algebra of function built on group theoretic constructions: examples of direct product, free product and quotient have been given. It remains to extend these constructions (e.g. defining an amalgamated product) and to check if, starting from groups owning the required properties (e.g. existence of a mechanism to test coset membership), these properties can be constructively lifted through the group constructions. Computational group theory is an

extensively studied area, see for example [26] and a large corpus of results is available. The reader may find in [22, 12] a review of the theoretical tools needed to solve the implementation problems we have discussed for abelian fields. These constitutes the basis of a parallel platform in JAVA [14] for the computation of data fields.

Shape specification and construction fit naturally the framework of type theory. For instance, presentations correspond to ground types and group constructions to type expressions. The parameterized shape $\mathrm{D}(N)$ in section 2.2 is an example of a value dependent type. The group foundation of shapes offers several tools to formulate various type equality. For example, group isomorphism, which is solvable for finite abelian presentation, is a candidate for observational equality. But group isomorphism does not preserve the neighborhood structure of a shape, which is central in our approach: we have to check in addition that the image of a generator is a generator, and conversely. There is also a natural interpretation for subtyping: a shape S' is a subtype of shape S if they define the same group and if the generators of S are included in those of S' (all field operations defined on S are available on S'). This is decidable for abelian presentation. These examples corroborate our opinion that, in addition to the gain in expressive power for the programmer, the use of group theory gives also a gain for managing the type structure.

For recursive field definitions, the decomposition of a field into subfields is a fundamental mechanism. The need of powerful decomposition mechanisms appears in quantification of definitions and in reduction expressions. We use respectively cosets and normal subgroups. It is interesting to compare this situation with the approach of Bird-Meertens algebra [4] or with the power-list algebra [23]. These theories develop a basis for the (recursive) definition of lists or arrays. The decomposition relies on the concatenation: appending two lists gives another list and concatenating two homogeneous arrays gives another array, leading to a divide-and-conquer computation strategy. In group-based fields, the decomposition relies on cosets (the sets L_t giving the decomposition of the computations) or on a normal subgroup (which decomposes naturally the group into a product). A direction for future work is to investigate other possible and useful decompositions of shapes. An analogous for the concept of list-homomorphism must also be worked out for group based fields.

9. REFERENCES

[1] E. A. Ashcroft, A. Faustini, R. Jagannathan, and W. Wadge. *Multidimensional Programming*. Oxford University Press, February 1995. ISBN 0-19-507597-8.

[2] D. Balsara, M. Lemke, and D. Quinlan. *Adaptative, Multilevel and hierachical Computational strategies*, chapter AMR++, a C++ object-oriented class library for parallel adaptative mesh refinment in fluid dynamics application, pages 413–433. Amer. Soc. of Mech. Eng., Nov. 1992.

[3] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*, volume 67 of *Encyclopedia of mathematics and its applications*. Cambridge University Press, 1997. isbn 0-521-57323-8.

[4] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design, NATO ASI Series, vol. F36*, pages 217–245. Springer-Verlag, 1987.

[5] G. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, Nov. 1989.

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

[7] D. De Vito. *Conception et implémentation d'un modèle d'exécution pour un langage déclaratif data-parallèle*. Thèse de doctorat, Université de Paris-Sud, centre d'Orsay, June 1998.

[8] P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, Feb. 1991.

[9] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, Florida, 21–24 Jan. 1996.

[10] P. Fradet and D. L. Métayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

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

[12] J.-L. Giavitto. *Scientific Repport for the HDR*. PhD thesis, LRI, Université de Paris-Sud, centre d'Orsay, Sept. 1999. Research Report 1226.

[13] J.-L. Giavitto. A framework for the recursive definition of data structures. In *ACM-Sigplan 2nd International Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 45–55, Montréal, Sept. 2000. ACM-press.

[14] 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 *Lecture Notes in Computer Science*, pages 742–??, Sept. 1998.

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

[16] K. E. Iverson. A dictionnary of APL. *APL quote Quad*, 18(1), Sept. 1987.

[17] C. B. Jay. A semantics of shape. *Science of Computer Programming*, 25(2–3):251–283, 1995.

[18] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega calculator and library, version 1.1.0*. College Park, MD 20742, 18 november 1996.

[19] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its application. Technical Report UMIACS-TR-95-48, CS-TR-3457, Univ. of Maryland, College Park, MD 20742, 14 Aprils 1994.

[20] B. Lisper. On the relation between functional and data-parallel programming languages. In *Proc. of the 6th. Int. Conf. on Functional Languages and*

Computer Architectures. ACM, ACM Press, June 1993.

[21] B. Lisper and J.-F. Collard. Extent analysis of data fields. Technical Report TRITA-IT R 94:03, Royal Institute of Technology, Sweden, January 1994.

[22] O. Michel. *Représentations dynamiques de l'espace dans un langage déclaratif de simulation*. PhD thesis, Université de Paris-Sud, centre d'Orsay, Dec. 1996. N°4596, (in french).

[23] J. Misra. Powerlist: a structure for parallel recursion. *ACM Trans. on Prog. Languages and Systems*, 16(6):1737–1767, November 1994.

[24] S. Nishimura and A. Ohori. A calculus for exploiting data parallelism on recursively defined data (Preliminary Report). In *International Workshop TPPP '94 Proceedings (LNCS 907)*, pages 413–432. Springer-Verlag, Nov. 94.

[25] Z. Róka. One-way cellular automata on Cayley graphs. *Theoretical Computer Science*, 132(1–2):259–290, 26 Sept. 1994.

[26] M. Schnert. GAP 3.3. ftp samson.math.rwth-aachen.de:/pub/gap, 7 Nov. 1993.

[27] L. Semenzato. *An abstract machine for partial differential equations*. PhD thesis, U. of California at Berkeley, 1994.

[28] T. Torgersen. Parallel scheduling of recursively defined arrays: Revisited. *Journal of Symbolic Computation*, 16:189–226, 1993.

[29] J. van Leeuwen, editor. *Handbook in theoretical computer science*. Elsevier Science Publishers, 1990.

[30] A. White. *Graphs, groups and surfaces*. Mathematics Studies. North-Holland, 1973.

APPENDIX

A. COMPUTING THE DOMAIN APPROXIMATIONS OF AN ABELIAN GBF USING THE OMEGA CALCULATOR

Equations (14, 15, 17, 18, 19) enable the explicit construction of D_n and E_n if it is known how to compute intersection, union and product of *comonoid*. We call comonoid a set $x.M = \{x.m, m \in M\}$ where M is a monoid.

Indeed, a coset is a special kind of comonoid. Note that the intersection of a comonoid is either empty or a comonoid. If the product $D.M$ of a comonoid D by a monoid M is also a monoid (which is the case for abelian shape or if the r_i commutes with all group elements), then all arguments of the intersections and unions in the previous equations are comonoids. We may then express D_n and E_n for a given n has a finite union of comonoids. It is then clear that the definition domain of g is an union of comonoids. The conjecture only says that this union is finite.

We have used the `omega calculator`, a software package [18] that enables the computation of various operations on convex polyhedra to make linear algebra in $\mathbb{Z}^n$ and represent comonoids. Linear algebra is not enough to compute D_n and E_n because we have to compute the R_i. Fortunately, the `omega calculator` is able to determine in some cases the *transitive closure* of a relation [19] which enables the computation of R_i as the transitive closure of `[x, x.r`$_i$`]` (we use here the syntax of the `omega calculator`). We plan to develop a dedicated library under `Mathematica` to compute these approximations systematically.

Here is in example, based on the definition illustrated in figure 5. Please refer to [18] for the `omega calculator` concepts and syntax. We first define the cosets in $\mathbb{Z}^2$

```
C1 := { [n, 0] };
C2 := { [0, n] };
```

then three relations that correspond to the dependencies:

```
r1 := { [x, y] -> [x, y+1] };
r2 := { [x, y] -> [x+2, y] };
r3 := { [x, y] -> [x+1, y+1] };
```

and we need also the inverse of the dependencies:

```
ar1 := { [x, y] -> [x, y-1] };
ar2 := { [x, y] -> [x-2, y] };
ar3 := { [x, y] -> [x-1, y-1] };
```

We may now defines the D_i:

```
D0 := C1 union C2;

H1 := r1(D0) intersection r2(D0) intersection r3(D0);
D1 := D0 union H1;

H2 := r1(D1) intersection r2(D1) intersection r3(D1);
D2 := D1 union H2;

H3 := r1(D2) intersection r2(D2) intersection r3(D2);
D3 := D2 union H3;
```

We can ask `omega` to compute a representation of D_3

```
{[x,0]} union {[0,y]} union {[4,1]} union
                            {[6,1]} union {[2,1]}
```

which is what it is expected. For the approximation E_i we need to represent the monoids R_i which is done through a transitive closure:

```
R1 := r1*;
R2 := r2*;
R3 := r3*;
```

The definition of E_0 raise the computation of

```
E0 := R1(D0) intersection R2(D0) intersection R3(D0);
```

(we have ommited the union with D_0 to avoid too complicated term in the result). The evaluation of this definition returns

```
{[x,y]:  Exists (alpha :  0 = x+2alpha
   && 1 <= y && 2 <= x)} union {[x,0]} union {[0,y]}
```

This approximation is too large, we may refine it by computing E_1:

```
E1:= r1(ar1(E0) intersection E0) intersection
     r2(ar2(E0) intersection E0) intersection
     r3(ar3(E0) intersection E0);
```

The evaluation of E_1 gives:

```
{[x,1]:  Exists ( alpha :  0 = x+2alpha
                        && 4 <= x)} union {[2,1]}
```

which is also E_∞ minus D_0.

Extensions. We may extend the result (20) to non abelian forms simply by carefully taking care of the right or left applications of a shift r_i. We may also extend the previous results to the case of a *system* of recursive strict GBF $g, g', g'', ...$ by using $D_n, D'_n, D''_n, ...$ and $E_n, E'_n, E''_n, ...$ instead of only D_n and E_n.