Group-based fields[1]

Jean-Louis Giavitto, Olivier Michel, Jean-Paul Sansonnet
LRI u.r.a. 410 du CNRS, Bâtiment 490, Université de Paris-Sud,
F-91405 Orsay Cedex, France.
email: {`michel|giavitto|jps`}`@lri.fr`

[1] This paper is a revised version of a paper presented at PSLS'95 [1]. A shorter version must appears in the final proceedings. This version appears as a part of [**?**].

Table of Contents

Group-based fields

Jean-Louis Giavitto, Olivier Michel, Jean-Paul Sansonnet

LRI u.r.a. 410 du CNRS, Bâtiment 490, Université de Paris-Sud,
F-91405 Orsay Cedex, France.
email: {michel|giavitto|jps}@lri.fr

Abstract. This paper reports the preliminary work on extending the concept of collection in the declarative language 81/2. We propose to consider a collection as a function over a group (eventually tabulated in a distributed fashion). The group is the group of motions of the underlying space. This extension is called "group based fields". This approach makes the definition of point neighbourhood explicit and the various permutations on collections become now motions in a space. It also enables the definition of much richer shapes than in previous data-parallel languages. Examples are given to illustrate the involved concepts. The application of the fundamental structure theorem of abelian groups shows that group based fields naturally embed the multidimensional arrays. In addition, group based fields offers the same unifying framework for interpreting and handling others well-known structures like trees. We then sketch some of the problems implied by the implementation of recursively-defined group-based fields.

1 Introduction

This paper reports the preliminary work on extending the concept of *collection* in 81/2. 81/2 is a declarative language that allows the functional definition of *streams* and *collections* [2, 3]. In this paper, we focus our interest on a high-level programming abstraction which extends the concept of collection in 81/2. The new construct is based on an algebra of index set, called *shape*, and an extension of the array type, the *field* type.

The rest of this paper has the following structure. Section 2 gives some background on collections and arrays. Some shortcomings of data-parallel arrays are sketched. Section 3 describes the 81/2 answers to the previous problem and introduces group-based shapes. Section 4 gives some examples of shapes. Section 5 is devoted to fields and section 6 to field expressions. Section 7 examines the problem of boundary and introduces the lazy management of group-based fields. Section 8 gives examples of field definitions and section 9 sketches the implementation. Related and future works are discussed in the last section.

2 Arrays and collections

This paper reports the preliminary work on extending the concept of *collection* in 81/2. 81/2 is a declarative language that allows the functional definition of *webs* [2, 3]. A web combines the concepts of *stream* and collection. A stream is a temporal succession of values whereas a collection is a structured set of values that can be handled as a whole by the programmer. A web can be viewed as a stream of collections and is used for example to represent the trajectory of a dynamical system (a state that changes in time through the application of an evolution function).

For the sake of simplicity, we forget here the stream viewpoint of a web and we concentrate on the collection aspects. A 81/2 collection is a recursively defined nested vector. For example, the following equation

$$iota = 0\#(1 + iota) : [9] \tag{1}$$

defines a vector of 10 elements. The operator # stands for the concatenation, so this definition is very similar to the recursive definition of the infinite lazy list

$$\text{let rec } l = \text{cons}(0,\ \text{map}(1+,\ l))$$

The differences are: 1) in the *take* operator : [9] which selects the first nine elements of its argument; 2) in the implicit coercion of the scalar constant 1 into a vector; and 3) in the implicit extension of the operation + over vectors (a type system described in [4] makes things go right).

We argue that 81/2 vectors are collections because they are handled as a whole: no index manipulation or iteration loop appear in equation (1). Collections have been advocated as a good support for data-parallelism [5] whilst the declarative definition of webs supports implicit control-parallelism. Usual structures of collections are *sets* (SETL [6]), *bags* (Gamma [7]), *relations* (set of tuples, e.g. in SQL), *vectors* (*LISP), *nested vectors* (NESL [8], 81/2), and *multidimensional arrays* (HPF, MOA [9], new Lucid [10]). Typical operations on "collections structured as arrays" are point-wise applied scalar functions, reductions, scans and various permutations or rearranging operations that can be interpreted as communication operations in a data-parallel implementation. Recursive definitions of multidimensional arrays are mainly studied in systolic programming [11] and in automatic parallelization of loops [12], using exclusively tools from linear algebra. In the field of functional programming, recursive definitions of infinite lazy list have mainly been studied [13].

Arrays are a basic and versatile data-structure in computer science. It can be used for example to implement queues, graphs and so on. Nowadays, simulation of large dynamical systems (resolution of PDE, discrete events simulations, etc.) represents the majority of supercomputer applications. Collections are often used in these algorithms to represent the variation of some quantity over a bounded spatial or temporal domain: for example a vector can be used to record the temperature at the discretisation points of a uniform rod in the simulation of heat diffusion. Indeed, collection managed as a whole are very well fitted to such computation because the same physical laws apply homogeneously to each point in space or in time. In the previous example, the array structure is most effective than other collection structures because it matches naturally the grid structure of the rod×time discretisation. However, it presents several shortcomings:

- Natural operations upon spaces, like taking the value of the neighbour elements, must be implemented in term of index manipulations.
- Arrays have static bounds: traditional arrays are shaped like n-dimensional box, defined by a lower and an upper bound in each dimension, but grids may have more complex shapes. And simulation of growing processes (like plant growing) requires dynamically bounded arrays.
- Arrays provide a natural representation in the simple case of multidimensional grids. For example, to implement a circular buffer, or to discretise a circle, additional management must be included in the index manipulation (e.g. increasing or decreasing the index modulo the length of the buffer or the size of the discretisation).
- The topology of the space implemented as an array is implicit. The ability to support several space topologies using the same array structure relies mainly on the uniform access to an array element and in the "encoding" of the topology in terms of index manipulations.
- Space formalisms (e.g. geometry, linear algebra, tensor calculus, differential algebra) do not match array formalisations (e.g. product domain in denotational semantics).
- Arrays have a simple and fast implementation on homogeneous random-access memory architectures. However, high-performance architectures do not have a homogeneous memory model. On vector architectures, access to sequential elements is faster than to random elements. The optimal storage layout for an array depends on its access pattern, and a poor layout can have a dramatic impact on execution speed. Extracting access patterns from index operation nested in iteration loops requires difficult and not always successful analysis.

This motivates the development of a new collection structure.

81/2 abandons the concept of a general-purpose array type, and specializes it towards two directions. The first one is a specialization towards finite difference algorithms and space discretisations by considering more general grid topology and grid shape. The second specialization we consider is towards the simulation of growing processes by considering partial data-structure.

The goal of theses extensions is to relieve the programmer from making many low-level implementation decisions and to concentrate in a sophisticated data-structure the complexity of the algorithms. Certainly this implies some loss of run-time performance but in return for programming convenience. Future work must establish how much loss we can tolerate and and what we do get in exchange.

3 Extending the concept of collection

81/2 introduces two new primitive types: *shapes* and *fields*. A *shape* represents a set of coordinates. An example of coordinates is integer tuples, but more generally, 81/2 uses a *group* element to index a point. A *field* is an array whose index set is an arbitrary set in a shape. Operations on fields are data-parallel ones. A field is virtually defined over its entire shape, even if the shape has an infinite number of elements, but the values of the field are *computed only if needed*: that is, a field is a lazy data-structure.

In this section, we examine the concept of shape. A shape must specify both the elements of an index set and the neighbourhood of an element. The motivations of considering a group to specify the index set are introduced through an example.

3.1 Explicit definition of the neighbourhood of a point

Consider the recursive equation (1) for the definition of *iota*. This definition is valid for every element in *iota*. So,we can state that

$$iota.i = (0\#(1 + iota) : [9]).i \quad 0 \le i < 10$$

The bounds for index i come from the operator take and from the properties of the concatenation. The vector elements, numbered from 0, are accessed using the operator dot. Thus, by virtue of concatenation, we have:

$$\begin{aligned} iota.0 &= 0 \\ iota.i &= 1 + iota.(i-1) \quad 1 \le i < 10 \end{aligned}$$

That is, to compute the value of $iota.i$ we need to know the value of $iota.(i-1)$. We will say that the elements $i-1$ and i are *neighbours*[1]. From this point of view, the neighbours of an element P are the elements accessed to compute the value of P. In (1), the neighbours are *implicitly* defined by the action of catenating 0 to the left, which shifts the collection to the right and creates a linear order between elements.

We will make neighbourhood definition *explicit* in the type of a collection by specifying several spatial shifts to define the neighbourhood for each point. Such a structure will be called a *space* or a *shape*. A shape is part of the type of a collection, like `[100]` is part of the `C` vector type `int [100]`.

3.2 Displacement and displacement composition

In the following, we restrict ourselves to *regular* space, that is, to spaces where two points have the same configuration of neighbours (e.g. "one to the left" and "one to the right"). Let give a name "a", "b", "c", ... to each neighbouring direction and let $P{<}a{>}$ be the "a" neighbour of a point P. We can consider a as the displacement from a point towards one of its neighbours (see figure

[1] This terminology is motivated by the fundamental postulate in physical science that actions are local in space (no instantaneous distant action) and in time (causality). That is, the state of a point depends only on the states of its neighbours. Therefore, if a point value is required to compute another point value, these two points have to be neighbours in some space, the (perhaps abstract) space where the phenomenon takes place.

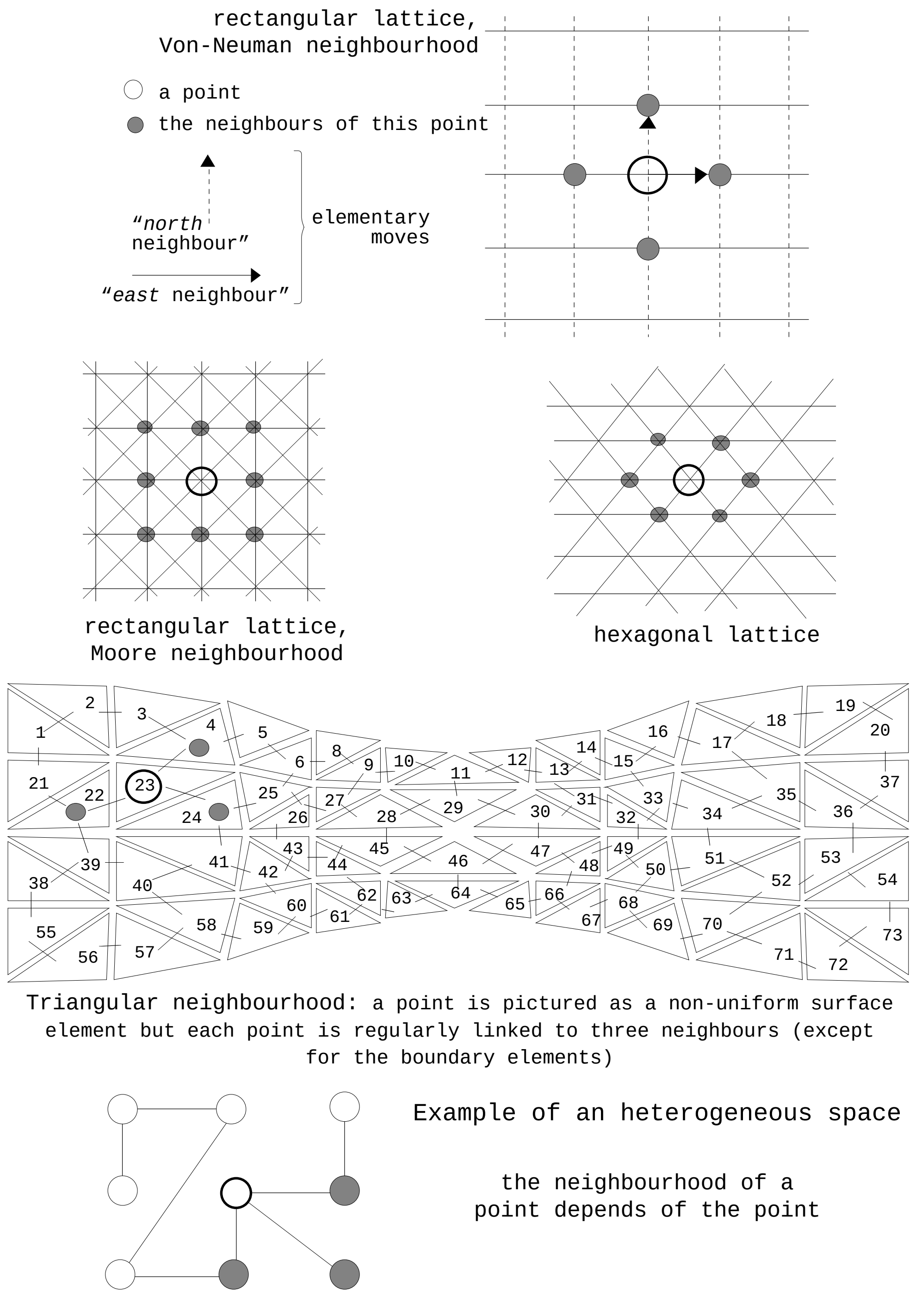

rectangular lattice,
Von-Neuman neighbourhood
a point
the neighbours of this point
"north neighbour"
"east neighbour"
elementary moves
rectangular lattice,
Moore neighbourhood
hexagonal lattice
Triangular neighbourhood: a point is pictured as a non-uniform surface element but each point is regularly linked to three neighbours (except for the boundary elements)
Example of an heterogeneous space
the neighbourhood of a point depends of the point

Fig. 1. Four examples of homogeneous spaces and one example of a non homogeneous space.

1). Displacement operation 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 an inverse displacement a^{-1} for each displacement a such that $P{<}a.a^{-1}{>} = P{<}a^{-1}.a{>} = P$. In other words, the displacements form a *group* for the composition, and the application of the displacements to points is the action of the group over the point space.

3.3 Spaces as groups

Regarding the previous section, regular spaces have to do with groups. Indeed, we make no distinction between points and displacements using the following scheme. Let G be a group and S a subset of G. *Space*(G, S) denotes the directed graph having G as its set of vertices and $G \times S$ as its set of 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 either to a *path* (a succession of elementary displacements) and a point (the point reached starting from the identity point e of G and following this path). Then we use $P.s$ instead of $P{<}s{>}$ for the s neighbour of P. In other words, *Space*(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.

The following dictionary gives the translation between graph theory and group related concepts:

Cayley graph		*Group*
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 $P.x = Q$
(there is a path from any P to any Q)		

We propose to consider a collection as a function over a group G (eventually tabulated in a distributed fashion) which complies with a space structure *Space*$(G,\ S)$. This extension is called a *group based field* (field is an alternative name for collection in the functional language community [14]). This approach makes the definition of point neighbourhood explicit, the various permutations on collections becoming motions in the underlying space.

Before going into group based fields, we will investigate further the concept of shape and shape specifications. Let us say that S is a *basis* of G if an element of G is a product of elements of S and that S *generates* G if $S \cup S^{-1}$ is a basis of G. The following propositions make a link between the global structure of *Space*(G, S) and the relations between G and S:

- For *Space*(G, S) to be connected, it is necessary and sufficient that S generates G. The connected components of *Space*(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 *Space*(G, S) to contain a loop, it is necessary and sufficient that e belongs to S.
- *Space*(G, S) has a circuit of length ≥ 2, if and only if $S \cap S^{-1} = \emptyset$.

In the following, we implicitly restrict to the case where the subset S generates G. If S is a basis of G, *Space*(G, S) is called the Cayley graph of the group G. If S is not a basis of G, *Space*(G, S) is a subgraph of the Cayley graph of G. Nota Bene: there exist regular connected graphs which are not the Cayley graph of a group [15].

We use a finite *presentation* to specify a group. A finite presentation gives a finite list of group generators and a finite list 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 not unique: different presentations may define the same group. However, a presentation uniquely defines *Space*(G, S): we use the generator list in the presentation to specify S. So the generators in the presentation are the distinguished group elements representing the elementary displacements from a point towards its neighbours.

We give two simple examples. Let C a cyclic group of order n generated by $S = \{a\}$. *D1* $=$ *Space*(G, S) represents the discretisation of a circle where n is the number of points of the discretisation. To specify *D1*, we give between brackets the list of the generators followed by the equation list:

$$D1 = \langle a \; ; \; a^n = e \rangle$$

In the following, a presentation denotes the corresponding space or the underlying group following the context. In the circle *D1*, we can move always in the same direction a. If we want to move in the reverse direction too, a^{-1} has to be added to the list of generators. A two-dimensional "bidirectional" grid is specified by:

$$\begin{aligned} G2 = \langle \; & North, East, South, West \; ; \\ & North.East = East.North, \; South = North^{-1}, \; West = East^{-1} \rangle \end{aligned}$$

There are four generators but, because the last two equations, $South$ and $West$ are just renamings for convenience of the inverses of $North$ and $East$. The first equation specifies that $North$ and $East$ commute, that is, the specified group is *abelian*. Abelian groups are of special interest and we specifically use the enclosing brackets for the presentation of abelian groups, skipping the commutation equations in bracket as they are implicitly declared. The figure 2 gives several other examples of abelian shapes.

Observe that *D1* has a finite number of elements and that this is not true for *G2*. If we use *G2* as the underlying domain of some fields, we must add the specification of a finite subset of the group elements. For the sake of simplicity, we forget for the moment this problem.

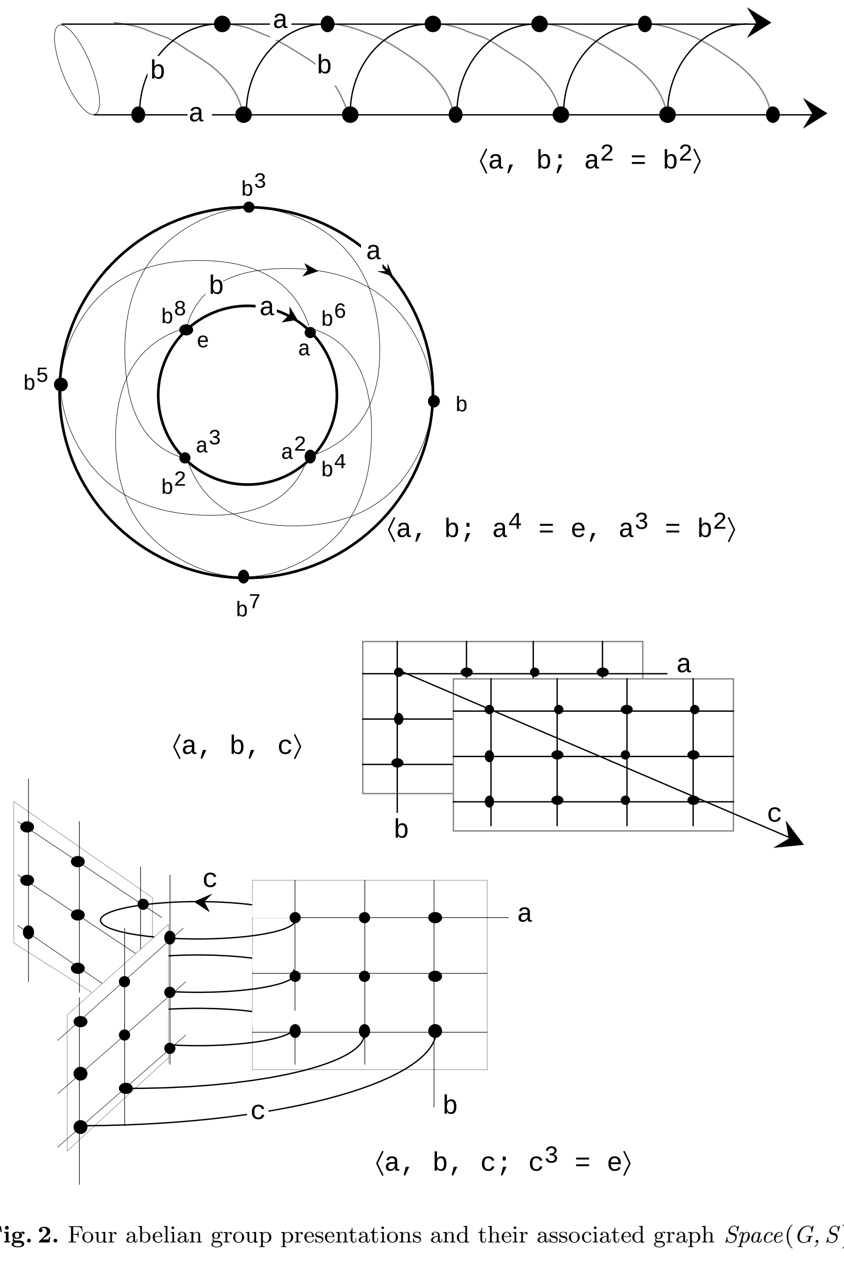


Fig. 2. Four abelian group presentations and their associated graph *Space*(G, S).

4 Abelian and non abelian shapes

This section reviews some interesting families of shapes.

4.1 Abelian shapes

A fundamental theorem of abelian groups says that each abelian group G is isomorphic to

$$G \simeq \mathbb{Z}^n \;\times\; \mathbb{Z}/n_1\mathbb{Z} \;\times\; \mathbb{Z}/n_2\mathbb{Z} \;\times\; \ldots \;\times\; \mathbb{Z}/n_q\mathbb{Z}$$

where n_i divides n_{i+1} (see any standard text on groups; for a computer oriented handling Cf. [16]). This theorem shows that the study of abelian shapes splits naturally into, on the one hand the study of free $\mathbb{Z}$-modules of finite rank (i.e. $\mathbb{Z}^n$), and on the other hand the study of finite $\mathbb{Z}$-modules. In other words, abelian shapes correspond to a mix of n-dimensional grids and n-dimensional torus.

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 (bounded region over a free) abelian shape. For example, multidimensional LUCID fields, systolic arrays, Lisper's data-fields [17] and even lazy lists, fit into this framework. Furthermore, this allows to reuse most of the achievements in the (parallel) implementation of arrays (e.g. [12], [18]) to implement (bounded regions over) infinite abelian fields, and with some additional work, to adapt them to the handling of finite abelian fields. The basic tool to explicit the isomorphism between the finite presentation of an abelian group and $\mathbb{Z}$-modules is the computation of the Smith Normal Form (Cf. [19]).

Here is an example. $H2 = \langle a, b, c\, ;\; b = a.c \rangle$ corresponds to an hexagonal grid. Figure 3 presents $H2$ and $G2$ in a different mode: a vertex is figured by a cell instead of a point and neighbour vertices are adjacent cells (such hexagonal partitions of the plane have interesting applications in digital topology). $H2$ is isomorphic to $G2$ by the following injections:

$$\begin{array}{l} i : H2 \longrightarrow G2,\; a^p.b^q.c^r \rightarrow North^{p+q}.East^{q+r} \\ j : G2 \longrightarrow H2,\; North^s.East^t \rightarrow a^s.c^t \end{array}$$

Note that if $H2$ and $G2$ are group isomorphic, $H2$ and $G2$ have not the same shape[2] in the sense that the image of $H2$-neighbours of a point P in $H2$ are not the $G2$-neighbours of the image of P in shape $G2$. Other examples are given by the first example of figure 2:

$$\langle a, b\, ;\; a^2 = b^2 \rangle \simeq \langle a, c\, ;\; c^2 = e \rangle = \mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$$

[2] From the implementation point of view, it does not matter because $G2$ corresponds to an array and we can access uniformly to any array element in a shared-memory implementation (in a distributed memory implementation, we can use a Valiant routing strategy to achieve statistically the uniformity of accesses).

(just take $c = a.b^{-1}$) and for the second example

$$\langle a, b\,;\; a^4 = e, a^3 = b^2\rangle \simeq \langle d\,;\; d^8 = e\rangle = \mathbb{Z}/8\mathbb{Z}$$

(take $d = a^{-2}.b$).

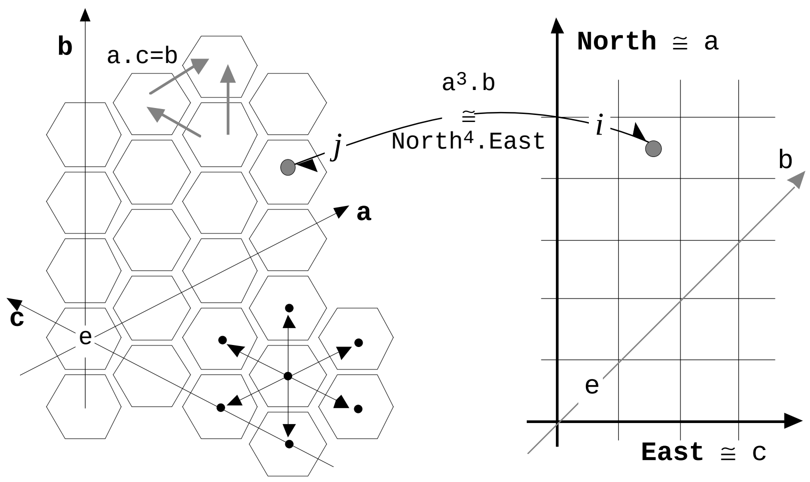


Fig. 3. Representation of *H2* and *G2* and the isomorphism between them.

4.2 Non abelian shapes

Abelian groups are an important but special case of groups. We give here two significant examples of a non abelian shape. Presentations of a non abelian shape are given between $\langle\!|$ and $|\!\rangle$. The first example is a *triangular neighbourhood*: the vertices of T are at the centre of equilateral triangles, and the neighbours of a vertex are the nodes located at the centre of the triangles which are adjacent side by side:

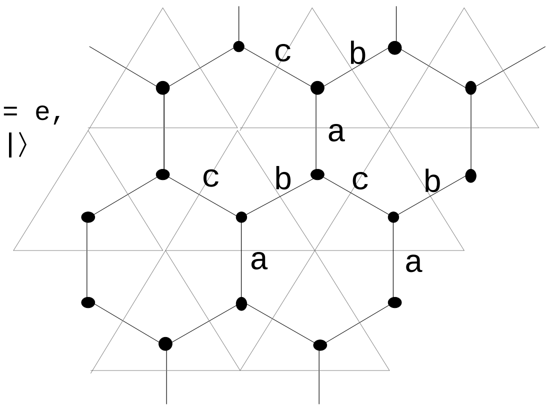
$$T = \langle\!| a, b, c\,;\; a^2 = b^2 = c^2 = e, (a.b.c)^2 = e |\!\rangle$$

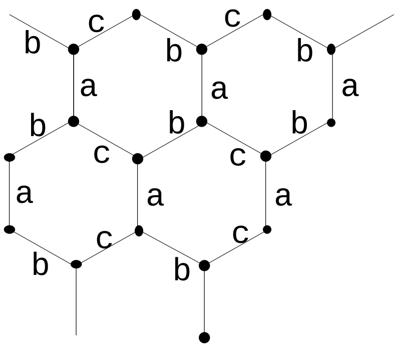
Such a lattice occurs for example in flow dynamics because its symmetry matches well the symmetry of fluid laws. Figure 4 gives a picture of T and shows two other possible presentations for a triangular partition of the plane. It is easy to see that, for instance, $a.b \neq b.a$ so this group is not abelian.

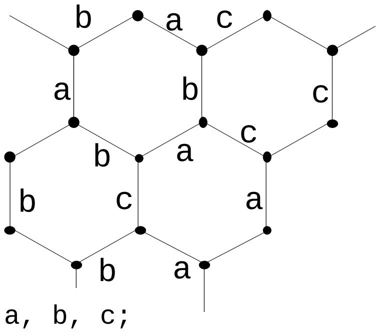
Our second example is simply a free group. A free group is a group without constraint between generators. Note that free abelian groups add just the commutation equations between generators and that they correspond to grids. The free non abelian shape

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

```
⟨|a, b, c;
  a.a = b.b = c.c = e,
  a.b.c.a.b.c = e |⟩
```







```
⟨| a, b, c;
      a.a = b.b = c.c = e,
      b.a.b = c.a.c |⟩
```

```
⟨| a, b, c;
    a.a = b.b = c.c = e,
    (b.c)³ = e,
    (c.a)³ = e,
    (a.b)³ = e |⟩
```

Fig. 4. Three examples of a 3-neighbourhood shape. These shapes are non abelian.

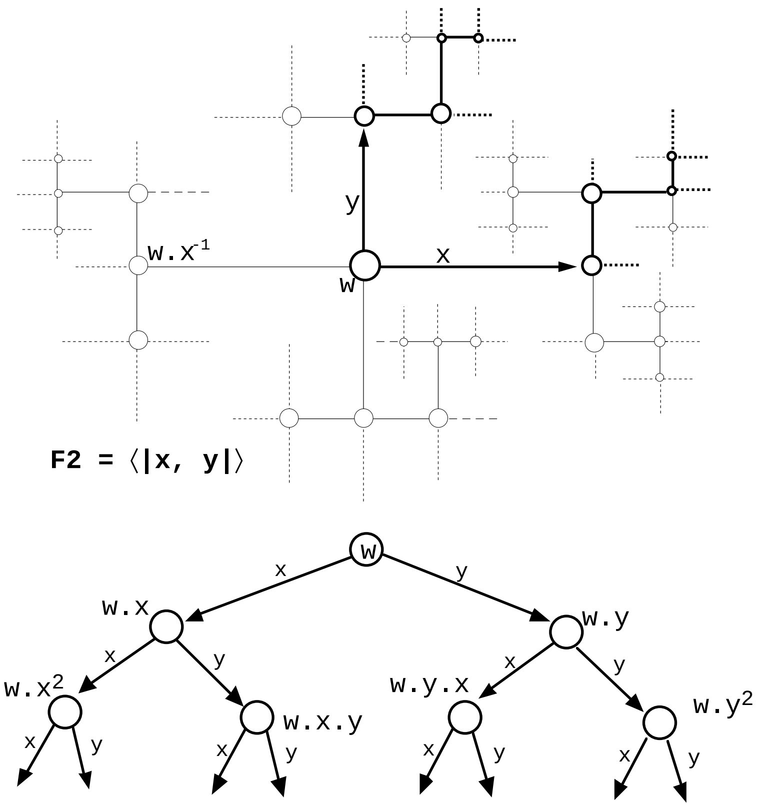


Fig. 5. A free non abelian group with two generators. Bold lines correspond to the points that can be reached starting from a point w and following the elementary displacements x and y.

is pictured in figure 5. We see that the corresponding space can be pictured as a tree (i.e. a connected non-empty graph without circuit). Actually, there is a more general result stating that if $\mathit{Space}(G, S)$ is a tree, then G is a free group generated by S.

This enables the embedding of some class trees in our framework. Let $\mathit{Space}(G, S)$ where G is a free group and S is a minimal set of generators, i.e. no proper subset of S generates G. Then $\mathit{Space}(G, S)$ 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. Figure 5.b gives an illustration of the points accessed starting from a point w in $\mathit{F2}$: it is a binary tree with root w. We cannot link to a generator the meaning of the father accessor (for node $w.x$, the father accessor is x^{-1}, whilst it is y^{-1} for the node $w.y$; in addition they are not in S).

5 Defining group based fields

5.1 Neighbourhood compliant definitions

A field F can be thought as a function over a space that complies with the space structure. That is, for each point P of $\mathit{Space}(G, S)$ we have

$$F(P) = f(F(P.a),\ F(P.b),\ \ldots)$$

with a, b, ..., in S and f the functional dependency between a point value and the values of its neighbours. Because such a relationship must hold for every point P, we make it implicit and write:

$$F = f(F.a,\ F.b,\ \ldots)$$

making this equation an equation about fields and not about field values. Operations on fields are reviewed below. The generators a, b, ... are not always sufficient to infer the domain of F (for instance, if the generator names are shared by several groups or if they do not appear at all, like in the equation "$F = 0$"). So we write:

$$F[E] = f(F.a,\ F.b,\ \ldots)$$

for a field F over a space E. The syntax $\ldots[E]$is borrowed from the usual notation for specifying the shape of an array.

With these conventions, a possible program for a field on a one-dimensional line, where the value of a point increases by one between two neighbours, is:

$$\mathit{G1} = \langle \mathit{left} \rangle$$
$$\mathit{iota}[\mathit{G1}] = 1 + \mathit{iota.left}$$

We obviously need to set the value of iota at some point. More generally, we will make a partition of the shape and define the field giving an equation for each element of the partition.

5.2 Coset quantified definitions

It implies that each element of the partition can be viewed as a shape in 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. Going back to the *iota* example, we write:

$$\begin{aligned} \mathit{G1} &= \langle \mathit{left} \rangle \\ A &= \mathit{left}^2.(\langle\rangle : \mathit{G1}) \qquad (2) \\ \mathit{iota}@A &= 0 \qquad (3) \\ \mathit{iota}[\mathit{G1}] &= 1 + \mathit{iota}.\mathit{left} \qquad (4) \end{aligned}$$

Equation (2) defines the coset $A = \{\mathit{left}^2\}$ because the subgroup $\langle\rangle : \mathit{G1}$ is reduced to $\{e\}$ by convention. Equation (3) specifies that the field *iota* has the value 0 for each point of coset A and equation (4) is valid for the remaining points. We say that equation (3) is *quantified* over A and that (4) is the *general* definition of *iota*. To define a field *iota* with the value 0 fixed at point e, we set "$\mathit{iota}@\langle\rangle = 0$" instead of (3). We write $\langle\rangle$ for $e.(\langle\rangle : \mathit{G1})$ because a subgroup H is also the coset $e.H$ and because here, after $\mathit{iota}@$, $\langle\rangle$ denotes necessarily a subgroup of $\mathit{G1}$.

5.3 Well formed shape partitions

In the *iota* example, A is included in $\mathit{G1}$ hence equations (3) and (4) both apply to define the values for the A points. To leave the ambiguity, it is assumed that the equation over the smaller domain is the valid one. The domains are ordered by inclusion, which is a partial order. So, if $A \cap B \neq \emptyset$ it is assumed that a domain $C = A \cap B$ has been defined (the intersection of two cosets is either empty or a coset). Note that the set of points where the general definition applies is not a coset but the complement of a union of cosets.

6 Operations on field

Operations on fields are of three kinds: extension of scalar functions, geometric operations and reductions.

6.1 Extension

Extension of a scalar function is just the point-wise application of the function to the value of a field in each point. We do not consider here nested fields, therefore the extension of a function can be implicit without ambiguity (for an example of possible ambiguity in the case of nested fields, consider the application of the function *reverse* over a nested list and its implicit extension [5]). 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.

6.2 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. 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: $(a.F)(w) = F(a.w)$.

Another geometric operation is the restriction $F|C$ of the field F to a coset C. If $C = x.H$, the shape of $F|C$ is H. This operation enables the access to the value(s) of a (coset of) element(s): for example, $F|(w.\langle\rangle)$ is a field over the single point domain $\{e\}$ and valued by $F(w)$.

Other useful geometric operations can be defined. We just mention 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 $h(F_1(u), F_2(v))$.

6.3 Reductions

Reduction of an n-dimensional array in APL is parameterised by the *axis* of the operation [20] (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. Let H be a subgroup of G (H will be the axis of the reduction). For $u, v \in G$, we define the relation $u \equiv v$ if it 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 $\equiv$. 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, each subgroup is normal).

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 commutative and associative. The shape of $h\backslash H\ F$ is G/H. The value of $h\backslash H\ F$ on a point w 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 6 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. The problem of handling reductions over an infinite domain are reviewed below.

Scan operations [21] seems more problematic to define. For instance, what would be a scan with an axis H with more than one generator?

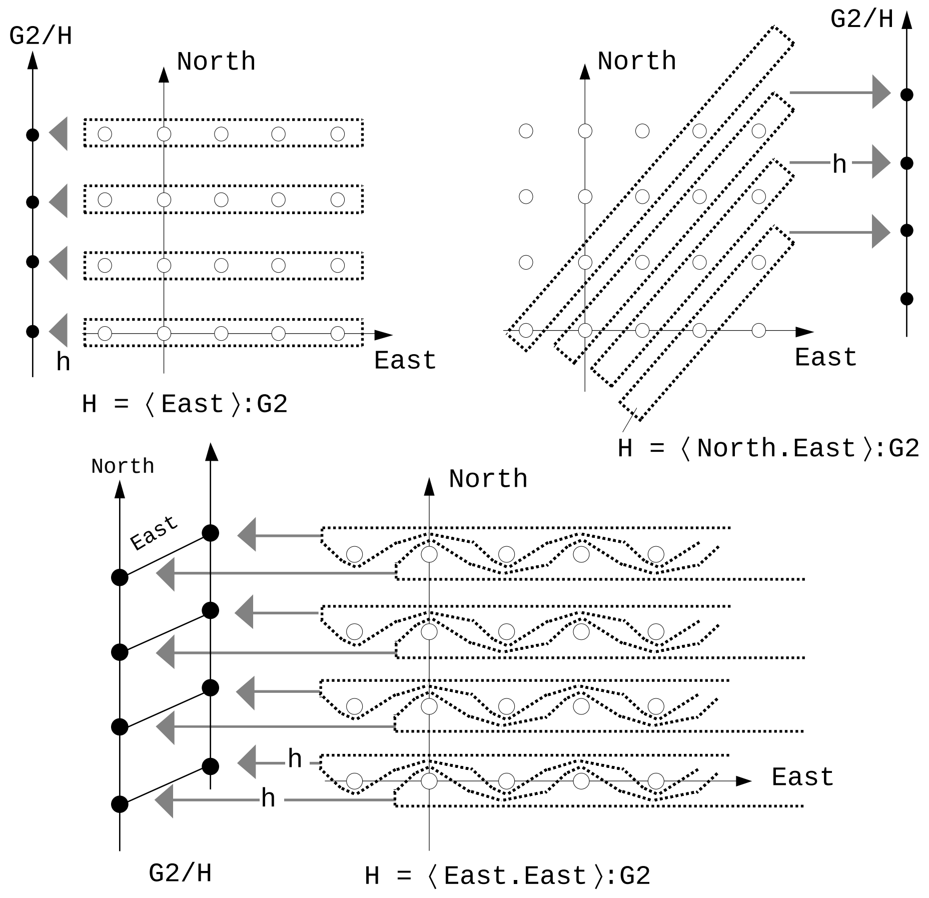


Fig. 6. Three examples of reduction over the *G2* shape.

7 Bounded versus lazy fields

We consider now the problem of handling infinite shapes. As a matter of fact, only the values of a field over a finite domain are of practical interest. One approach to tackle the problem is to specify explicitly this finite domain. However this approach is unsatisfactory.

First, the answer to: "what is a suitable language for specifying finite subsets in a group?" seems far from obvious.

Secondly, what does it mean to specify a finite domain over a finite group? To fix the idea, suppose we define a field over *D1*(4) the discretised circle with 4 elements. We will restrict it to the interval $\{a^\alpha : 0 \leq \alpha < 3\}$. The cyclic structure of *D1*(4) is lost and the programmer may use indifferently *G1* as the underlying domain for this field. On the contrary, if we restrict to $\{a^\alpha : 0 \leq \alpha < 8\}$ we "wrap around" two times (see figure 7). This is certainly not a desirable behaviour because the choice of the subset contradicts the structure of the shape.

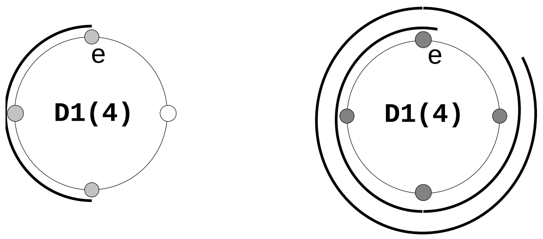


Fig. 7. Selecting a sub-domain in a shape.

Third, if we know for certain where we want a result (in heat diffusion example Cf. section 8.3, it could be the temperature at the middle point of the rod at time 100) we do not know the figure of the points that must be evaluated to compute the required results. In such a context, specifying explicitly a sub-domain of the field leads either to "access out of bounds" errors during the computation or to somewhat useless computations.

So we advocate a *lazy approach* to compute a field: a field is virtually defined over its entire shape, even if the shape has an infinite number of element, but the values of the field are *computed only if needed*.

If the use of a lazy data structure and of a lazy evaluation strategy relieve the burden of specifying a working subdomain, it does not solve the problem of specifying the requested results because we certainly need the value of a field in more than one point. However, the problem is much less compelling because we now deal with a clean and straightforward mathematical object; the specification of the required subset of values is just like the problem of specifying a good format for the "print" function (just like `"%.8f"` in the `C` language allows to print the first 8 decimal places of a `float`).

For a first "formatting language", we propose the following. The elements of an abelian group are all enumerated by varying $g_1^{n_1}.g_2^{n_2}. \ldots .g_p^{n_p}$ where the g_i are the generators of the shape. So a subset is specified by setting an interval for each n_i. For non abelian fields, all the group element are enumerated by $g_{i_1}^{n_1}.g_{i_2}^{n_2}. \ldots .g_{i_p}^{n_p}. \ldots$ where g_{i_j} is some generator and $g_{i_j} \neq g_{i_{j+1}}$. So we specify a maximal number p of generators and an interval for the power of each generator.

Figure 8.a shows a bounded domain in *H2* where each point has form $a^\alpha.b^\beta.c^\gamma$ with $0 \leq \alpha < 4$, $-1 \leq \beta < 2$, $0 \leq \gamma < 2$. Figure 8.b shows such a domain in T, the maximal number of generators being 2 and with $0 \leq n_j < 2$ (anyway in T, $g_i^2 = e$).

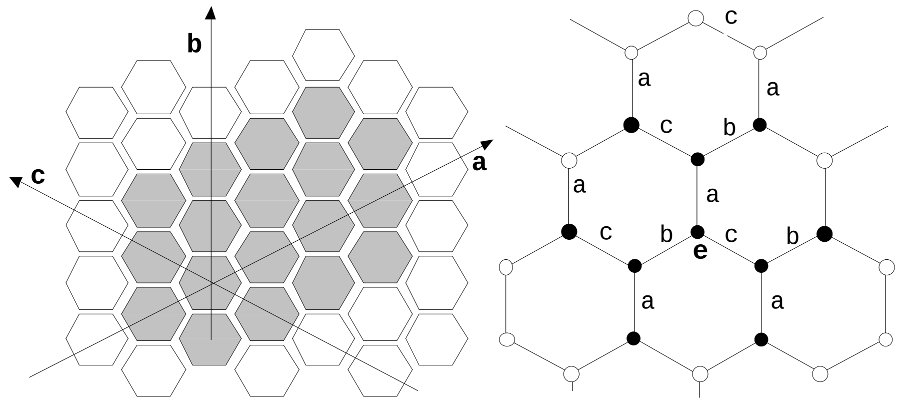


Fig. 8. Two domains described by the format of a "print" function dedicated to lazy fields.

Using (lazy) infinite shapes, may result in the combination of an infinite amount of values in a reduction computation. Such an operation may have sense, for example, when there is only a finite number of defined values in the field. In this case, making a global sum really means: "make the sum of all defined values and ignore the undefined ones". The situation is the same for a dynamically bounded array: it is potentially infinite but the sum of all elements means the sum of the current elements. With this approach, the programmer implicitly works on a finite portion of the infinite field in order not to be burdened with the handling of the bounds and because the required topology matches better the topology of this infinite field. The problem is then to explicitly define, or to infer at compile-time, or to maintain at run-time, a bounding region outside which all values are necessarily undefined. Another possible meaning of an infinite reduction is the existence of the result from a mathematical point of view (e.g. when computing a convergent series). So we can compute an approximation of the result if we know that, outside some bounded region, the values to be reduced are vanishing.

8 Examples of group based fields

8.1 The iota example continued

Going back to the *iota* example, we can now say that equations (3)+(4) define the function:

$$\text{let rec } iota(left^n) = if\ n == 2\ then\ 0\ else\ 1 + iota(left^{n+1})\ fi$$

over *G1*. This function has a defined value for the points $\{left^n, n \leq 2\}$ and the value $\perp$ for the other points, see figure 9. Note that the use of a displacement a instead of a^{-1} is mainly a convention but it does matter when specifying the domain of required values.

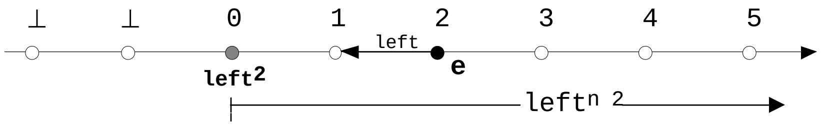


Fig. 9. The field *iota*.

8.2 Combining subfield definitions

Here, we give a more sophisticated example corresponding to a field F over a cylinder. We use the free product $*$ of two groups which is simply defined by joining their respective presentations and identifying the identity elements: the free product coincides with the direct product for abelian groups.

$$\begin{aligned} D1(n) &= \langle a\ ;\ a^n = e \rangle \\ G1 &= \langle left, right\ ;\ left = right^{-1} \rangle \\ Cyl(n) &= D1(n) * G1 \end{aligned}$$

$$F@\langle\rangle = 0 \tag{5}$$
$$F@D1(4) = 1 \tag{6}$$
$$(F@G1)\ as\ L = L.left + 1 \tag{7}$$
$$f[Cyl(4)] = F.left + F.a \tag{8}$$

This example introduces parameterised shapes (e.g. $D1(n)$) and uses a local name (the construction "...*as* L" to denote a subfield. A group H denotes implicitly the coset $e.H$ if needed and the elements of a product are implicitly subgroups of the product. So $D1(4)$ in equation (6) really means $e.(D1(4) : Cyl(4))$. In the definition of F, we have $e.G1 \cap e.D1(4) = e.(\langle\rangle : Cyl(4)) = \{e\}$, thus the shapes are well defined (and in equation (5), $\langle\rangle$ really means $e.(\langle\rangle : Cyl(4))$). See figure 10 to visualise the corresponding partition.

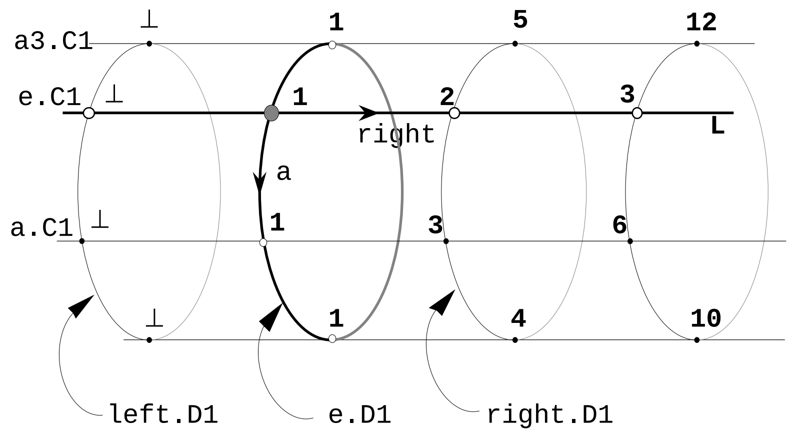


Fig. 10. Definition of a field over a more complex partition of the shape.

8.3 A more concrete example

The problem consists in the simulation of the diffusion of heat in a thin uniform rod. Both extremities of the rod are held to $0°C$. The equation of the diffusion takes the following form: $\partial heat/\partial t = k\, \partial^2 heat/\partial x^2$ where x is the coordinate of a point on the rod. For its numerical resolution, this partial differential equation is discretised as

$$\frac{heat_{i,t+1} - heat_{i,t}}{k} = \frac{heat_{i+1,t} - 2heat_{i,t} + heat_{i-1,t}}{h^2}$$

where i represents the coordinate of the point on the discretised rod and t the discrete time coordinate (the constant h depends on the discretisation). In fact, $t+1$ is a trick to address the instant following t (which is the meaning of ∂t) and $i+1$ and $i-1$ are the encoding of the left and right neighbours of point i on the rod (in other words, the additive group on integers is used to code the group of translations on a one dimensional line).

This example can be easily turned into the formalism of group based field. To define the space-time domains we use the free product of simpler domains.

$$\begin{aligned} Time &= \langle past \rangle && \text{in time we can only progress, so no inverse of generators appears} \\ Rod &= \langle left, right\,;\ left = right^{-1} \rangle && \text{an infinite rod} \\ SpaceTime &= Time * Rod \end{aligned}$$

We define the cosets corresponding to the initial and boundary conditions (Cf. figure 11).

$$\begin{aligned} InitialLeftBorder &= e.(\langle\rangle : SpaceTime) \\ InitialRightBorder &= right^n.(\langle\rangle : SpaceTime) \\ Initial &= e.(Rod : SpaceTime) && \text{the rod at time 0} \end{aligned}$$

$$\begin{aligned} \mathit{LeftBorder} &= e.(\mathit{Time} : \mathit{SpaceTime}) \quad \text{left border along the time} \\ \mathit{RightBorder} &= \mathit{right}^n.(\mathit{Time} : \mathit{SpaceTime}) \quad \text{right border along the time} \end{aligned}$$

Now we are able to define the field *heat*:

$$\begin{aligned} \mathit{heat}@\mathit{InitialLeftBorder} &= 0 \\ \mathit{heat}@\mathit{InitialRightBorder} &= 0 \\ \mathit{heat}@\mathit{Initial} &= \mathit{start} \quad \text{some initial heat distribution} \\ \mathit{heat}@\mathit{LeftBorder} &= 0 \\ \mathit{heat}@\mathit{RightBorder} &= 0 \\ \mathit{heat}[\mathit{SpaceTime}] &= 0.4 * \mathit{heat.past} + \\ &\qquad 0.3 * (\mathit{heat.past.left} + \mathit{heat.past.right}) \end{aligned}$$

The general equation of *heat* states simply that the current value of heat in some point is a linear combination of the heat of the neighbourhood at the previous instant. The field *start* is a parameter of the program and corresponds to some initial heat distribution. The following definition can be used to specify a symmetrical distribution:

$$\begin{aligned} \mathit{ramp}@\langle\rangle &= 0 \\ \mathit{ramp}[\mathit{Rod}] &= 1 + \mathit{ramp.left} \\ r &= \mathit{ramp}/n \\ \mathit{start}[\mathit{Rod}] &= r * (1 - r) \end{aligned}$$

Note that an infinite shape is used for the modelisation of the finite rod discretisation. Following the remarks at the end of the previous section, we ideally would have undefined values outside the domain implementing the rod. This can be achieved using an explicit restriction operator, like:

$$\mathit{start}[\mathit{Rod}] = r * (1 - r) \text{ in } \{\mathit{right}^p : 0 \leq p < n\} \text{ otherwise } \perp$$

9 Implementing fields

9.1 Solving field equations

Because a field can be viewed as a function over a shape, the semantic of a system of field equations is the same as the recursive definition of functions in denotational semantics [22]. Solving field definitions over the Scott domain on flat values, ensures the existence of a solution and the computation of the least solution by means of a fixpoint iteration.

An immediate question is to know if the fixpoint iteration converges on a point in a finite number of steps. For general functions this amount to solve the halting problem but here we are restricted to group based fields. However,

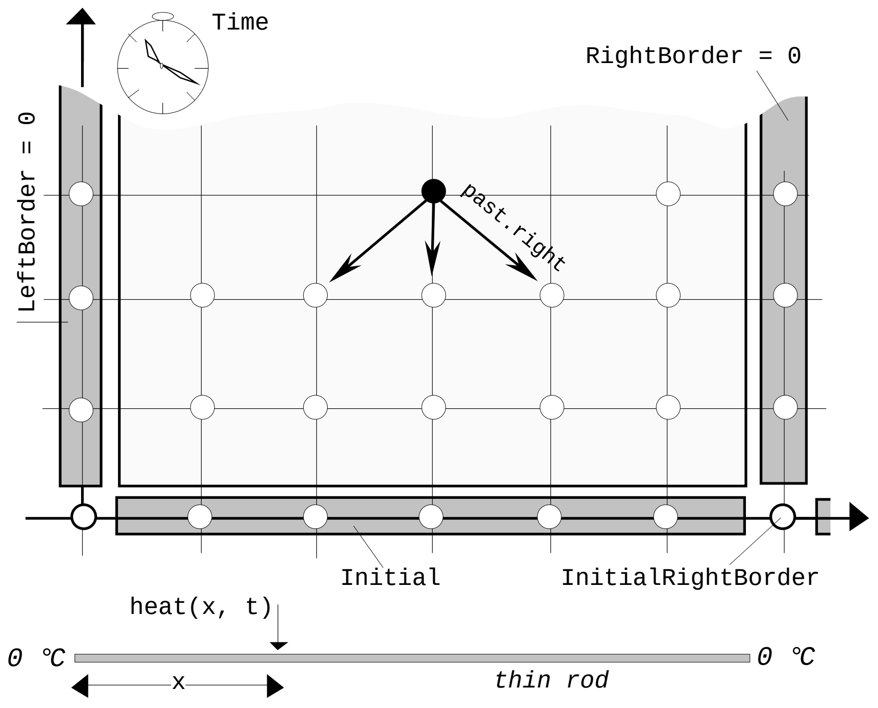


Fig. 11. Field encoding of the explicit numerical resolution of the parabolic partial differential equation governing the heat diffusion in a thin uniform rod.

the expressive power of group based fields is enough to confront to the same problem: suppose a field defined by

$$F[E] = f(F.a, F.b, \ldots)$$

the points accessed for the computation of the value of w are: $w.a, w.b, \ldots$. 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 to the problem of deciding if $w.a = w$, $w.b = w$, etc. In other words, 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).

9.2 Implementation

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

$$\begin{aligned} F@C1 &= c_1 \\ &\ldots \\ F@Cn &= c_n \\ F[G] &= h(F.g_1, F.g_2, \ldots, F.g_p) \end{aligned}$$

where Ci are cosets, c_i are constants and h is some extension of a scalar function. The set $dF = \{g_1, \ldots, g_p\}$ is called the dependency set of F.

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 suppose further 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 general 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 implement fields is the use of memoised functions. A field $F[G]$ is stored as a dictionary with entry $w \in G$ and value $F(w)$. If the value $F(w)$ of w is required, we check first 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 Ci 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 F at points $w.g_1, \ldots, w.g_p$, recurring the process, and then the results are combined by h.

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

In case of an abelian group G, we can further improve the implementation using the fundamental isomorphism between G and a product of $\mathbb{Z}$-modules. Confer [16, 23]. 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 previous evaluation scheme corresponds to a demand-driven evaluation strategy: e.g. to evaluate $iota(e)$, we have to compute $iota(left)$ which triggers the computation of $iota(left^2)$ which returns 0. So, we can associate to each point $w \in G$ a set $p(w)$ of directed paths corresponding to the points visited to compute $F(w)$. An element p of $p(w)$ is a word of the subgroup generated by dF. The evaluation of $F(w)$ fails if some $p \in p(w)$ has an infinite length. Two cases can arise: p is cyclic or p has infinitely many 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 (Cf. [17]). Sufficient condition can also be checked at compile-time to detect cyclic paths (e.g. a raw criterion can be $dF \cap dF^{-1} = \emptyset$ and/or it can be detected at run-time using an occur-check mechanism.

Tools developed for the scheduling of uniform recurrence equations can also be used to implement a data-driven evaluation strategy (but a data-driven strategy is not well fitted to a lazy evaluation strategy). The idea is to propagate the values from defined points to undefined ones, like a wavefront. At time 0, the value of F is known for the points in domain

$$Def_0 = C1 \cup C2 \cup \ \ldots \ \cup Cn$$

Thus, at time 1, it is possible to compute *in parallel* the values for points in domain

$$L_0 = Def_0.g_1 \cap \ \ldots \ \cap Def_0.g_p$$

So, at time 1, the values for F are known for

$$Def_1 = Def_0 \cup L_0$$

More generally, it is possible at time t to know the values for Def_t and to compute the values of L_t)

$$\begin{aligned} Def_t &= Def_{t-1} \cup L_{t-1} \\ L_t &= Def_t.g_1 \cap \ \ldots \ \cap Def_t.g_p \end{aligned}$$

The computations of the values of L_t are independent and can be carried in parallel.

A compiled approach of a data-driven evaluation strategy would try to infer at compile-time the frame of L_t but in general this is difficult: for example, L_t is not necessarily a coset. A possible approach would be to find a tractable approximation of L_t. In the case of recursively defined arrays (which corresponds to free abelian fields), the hyperplane scheduling method makes the assumption that the time at which an array element will be computed is given by a linear combination of the array subscripts [18] (that is, the approximation of L_t is an hyperplane, or in our context, a coset).

10 Conclusions and future work

This paper reports the very preliminary work on extending the concept of collection in the 81/2 language. We propose to consider a collection as a partial function over a group compliant with a shape. A shape is a directed graph defining the possible dependencies in the computation of a point value. This approach makes the definition of point neighbourhood explicit. It also enables the definition of much richer shapes than in previous data-parallel languages.

The specification of a shape is directly done using a finite presentation or build through a group theoretic construction: 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 [24] and a large corpus of results is available.

We currently implement a `C++` library for the management of sets of bounded rectangular regions in $\mathbb{Z}^n$. This library will be used for the implementation of abelian fields. It is itself based on AVTL [25], a portable MPI [26] based parallel vector template library.

There is a small number of research efforts to extend the concept of array: Lucid [10], LPARKX [27], Infidel [28], AMR++ [29]. They all 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 integer). This forbids for example the natural representation of a tree or a triangular lattice.

In 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 [30] or with the power-list algebra [31]. 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 catenating 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 equivalence for the concept of list-homomorphism must also be worked out for group based fields.

The use of a group as the underlying domain of a field gives a rich structure to the computation dependencies: they can be interpreted as paths in well-handled spaces. Another direction of work is then the use of tools from algebraic topology to characterise the domain of computation (homotopy theory, etc.). Such mathematical tools have already be proved useful [32–34].

Acknowledgements. We are grateful to the members of the Parallel Architectures team in LRI for many fruitful discussions, and we thank especially Dominique De Vito and Abderrahmane Mahiout.

References

1. J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In R. H. Halstead, I. Takayasu, and C. Queinnec, editors, *Proceedings of the Parallel Symbolic Languages and Systems (PSLS'95)*, volume ? of *LNCS*, page ?, Beaune (France), 2-4 October 1995. Springer-Verlag. to be published.
2. J.-L. Giavitto. A synchronous data-flow language for massively parallel computer. In D. J. Evans, G. R. Joubert, and H. Liddell, editors, *Proc. of Int. Conf. on Parallel Computing (ParCo'91)*, pages 391–397, London, 3-6 September 1991. North-Holland.
3. O. Michel and J.-L. Giavitto. Design and implementation of a declarative data-parallel language. In *post-ICLP'94 workshop W6 on Parallel and Data Parallel Execution of Logic Programs*, S. Margherita Liguria, Italy, 17 June 1994. Uppsala University, Computing Science Department.
4. J.-L. Giavitto. Typing geometries of homogeneous collection. In *2nd Int. workshop on array manipulation, (ATABLE)*, Montréal, 1992.
5. J. M. Sipelstein and G. E. Belloch. Collection-oriented languages. *Proc. of the IEEE*, 79(4), April 1991.
6. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets: and introduction to SETL*. Springer-Verlag, 1986.
7. J.-P. Bânatre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4:133–144, 1988.
8. G. E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.
9. G. Hains and L. M. R. Mullin. An algebra of multidimensional arrays. Technical Report 782, Université de Montréal, 1991.
10. E. Ashcroft, A. Faustini, R. Jagannatha, and W. Wadge. *Multidimensional Programming*. Oxford University Press, February 1995. ISBN 0-19-507597-8.
11. C. Mauras. Definition of Alpha: a language for systolic programmation. Technical Report 482, INRIA, June 1989.
12. P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.
13. B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programming Languages and Systems*, 11(4):633–649, October 1989.
14. 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.

15. A. White. *Graphs, groups and surfaces.* Mathematics Studies. North-Holland, 1973.
16. H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Text in Mathematics.* Springer-Verlag, 1993.
17. 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.
18. T. Torgersen. Parallel scheduling of recursively defined arrays: Revisited. *Journal of Symbolic Computation*, 16:189–226, 1993.
19. D. Smith. A basis algorithm for finitely generated abelian groups. *Math. Algorithms*, 1(1):13–26, January 1966.
20. K. E. Iverson. A dictionnary of APL. *APL quote Quad*, 18(1), September 1987.
21. G. E. Blelloch. Scans primitive parallel operation. *IEEE Trans. on Computers*, 38(11), November 1989.
22. J. van Leeuwen, editor. *Handbook in theoretical computer science.* North-Holland, 1990.
23. C. S. Iliopoulos. Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the hermite and smith normal forms of an integer matrix. *SIAM Journal on Computing*, 18(4):658–669, August 1989.
24. M. Schönert. Gap 3.3. ftp samson.math.rwth-aache.de:/pub/gap, 7 November 1993.
25. T. J. Scheffler. A portable MPI-based parallel vector template library. Technical Report 95.04, RIACS, 1995.
26. Message-Passing Interface Forum. MPI: a message-passing interface standard, May 1994.
27. S. R. Kohn and S. B. Baden. A robust parallel programming model for dynamic non-uniform scientific computation. Technical Report TR-CS-94-354, U. of California at San-Diego, March 1994.
28. L. Semenzato. *An abstract machine for partial differential equations.* PhD thesis, U. of California at Berkeley, 1994.
29. 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., November 1992.
30. 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.
31. J. Misra. Powerlist: a structure for parallel recursion. *ACM Trans. on Prog. Languages and Systems*, 16(6):1737–1767, November 1994.
32. E. Goubault and T. P. Jensen. Homology of higher-dimensional automata. In *Proc. of CONCUR'92.* Springer-Verlag, 1992.
33. C. C. Squiers and Y. Kobayashi. A finiteness condition for rewriting systems. *Theoretical Computer Science*, 131(2):271–294, 12 September 1994.
34. M. Herlihy and S. Rajsbaum. *Computer Science Today - Recent Trends and Developments*, chapter Algebraic Topology and Distributed Computing: A Primer, pages 203–217. Number 1000 in Lecture Notes in Computer Sciences. Springer-Verlag, 1995.