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

1 Introduction

This paper reports the preliminary work on extending the concept of *collection* in 8_{1/2}. 81/2 is a declarative language that allows the functional definition of *streams* and *collections* [1, 2]. 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 and fields. Section 4 is devoted to the shape algebra. Section 5 introduces the main field operations and field definitions. Section 6 sketches the implementation. Related and future works are discussed in the last section.

2 Arrays and collections

A collection is an aggregate of elements handled as a whole: no index manipulation or iteration loop appear in expressions over collections. Collections have been advocated as a good support for data-parallelism [3]. Usual structures of aggregation are *sets* (SETL [4]), *bags* (Gamma [5]), *relations* (set of tuples, e.g. in SQL), *vectors* (*LISP), *nested vectors* (NESL [6]), and *multidimensional arrays* (HPF, MOA [7], new Lucid [8]). Typical operations on "arrays as collections" are pointwise applied scalar functions, reductions, scans and various permutations or rearranging operations that can be interpreted as communication operations in a data-parallel implementation.

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. The array data structure is the most expressive (with respect to set, bag...) to implement space or time discretisation because it matches canonically the grid lattice. They have a simple and fast implementation on homogeneous random-access memory architectures. Yet this generality has its costs. 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. Moreover, while traditional arrays are shaped like n-dimensional box, defined by a lower and an upper bound in each dimension, grids may have more complex shapes. And simulation of growing processes (like plant growing) requires dynamically bounded arrays.

3 Shapes and fields

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.

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.

4 Shape constructs

A shape specify both the group used to denote the array elements and the neighbourhood of an element. 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). We use $P.s$ 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 labelled s is between the nodes P and Q if $P.s = Q$, and 3) the labels of the edges are in S. If S is a basis of G, $\mathit{Space}(G, S)$ is called the Cayley graph of the group G.

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 the shape $\mathit{Space}(G, S)$: 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 gives some example of shapes. A free abelian groups corresponds to a n-dimensional grid (n is the number of generators). The hexagonal lattice: $\mathit{H2} = \langle a, b, c\,;\ b = a.c\rangle$ is an abelian shape that can be used for example in image processing (the underlying space has the Jordan property, which is not the case for NEWS meshes). A (non abelian) free group is simply a tree (n generators for n sons). Another example of non abelian shape is the *triangular neighbourhood*: the vertices of T are at the center of equilateral triangles, and the neighbours of a vertex are the nodes located at the center of the triangles which are adjacent side by side. A possible shape is: $T = \langle\!| a, b, c\,;\ a^2 = b^2 = c^2 = e, (a.b.c)^2 = e |\!\rangle$. Such a lattice occurs for example in flow dynamics because its symmetry matches well the symmetry of fluid laws.

5 Field definitions

A field F can be thought as a function over a group that complies with the shape structure: the value of a field in some point depends only on the values of the neighbours points. 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:

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

for a field F over a shape E. Field expressions f are of three kinds: extension of scalar functions, geometric operations and reductions.

Extension of a scalar function is just the pointwise application of the function to the value of a field in each point.

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. A main 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: $(a.F)(w) = F(a.w)$.

Reduction of an n-dimensional array in APL is parameterised by the *axis* of the operation [9] (e.g. a matrix can be reduced by row or by column). A *normal subgroup* is used for axis in the case of group based shape. More details are given in [10].

When using recursive definition, "terminal cases" stop the recursion. For group-based fields, we will 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 in itself. We use *cosets* to partition the shape. Cosets may overlap, so additional constraints are put on the partition, Cf. [10].

6 Implementation

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

$$F@C1 = c_1, \ \ldots, \ F@Cn = c_n, \ F[G] = h(F.g_1, F.g_2, \ldots, F.g_p)$$

where Ci are cosets, c_i are constants and h is some extension of a scalar function. $F@Ci = \ldots$ is the equation defining the field F on coset Ci whilst $F[G] = \ldots$ is the general definition valid for the remaining points.

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 lazy 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 $\bar{w}$. For instance, 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 [11, 12]. As a matter of fact, a function over a $\mathbb{Z}$-module is simply implemented as a vector. The only difficulty here is to handle the case of $\mathbb{Z}^n$ which corresponds to an unbounded array.

7 Conclusions

We advocate in this paper the use of theoretical group constructions for the index set of an array. The resulting data-structure, called *group-based field*, is managed in a lazy way and extends the traditional array type. More details are given in [10]. 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 [13], a portable MPI [14] based parallel vector template library.

There is a small number of research efforts to extend the concept of array: Lucid [8], LPARKX [15], Infidel [16], AMR++ [17]. They all consider more general shapes for arrays 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 [18] or with the power-list algebra [19]. These theories develop a basis for the (recursive) definition of lists or arrays. The decomposition relies on the concatenation leading to a divide-and-conquer computation strategy. In group based fields, the decomposition relies on cosets 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. 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 [20, 21].

Acknowledgments. 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. 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.
2. 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.
3. J. M. Sipelstein and G. E. Belloch. Collection-oriented languages. *Proc. of the IEEE*, 79(4), April 1991.

4. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with sets: and introduction to SETL.* Springer-Verlag, 1986.
5. 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.
6. 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.
7. G. Hains and L. M. R. Mullin. An algebra of multidimensional arrays. Technical Report 782, Université de Montréal, 1991.
8. E. Ashcroft, A. Faustini, R. Jagannatha, and W. Wadge. *Multidimensional Programming.* Oxford University Press, February 1995. ISBN 0-19-507597-8.
9. K. E. Iverson. A dictionnary of APL. *APL quote Quad*, 18(1), September 1987.
10. O. Michel. A guided tour to 81/2 and its dynamical extensions. Technical report, Laboratoire de Recherche en Informatique, December 1995.
11. H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Text in Mathematics.* Springer-Verlag, 1993.
12. 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.
13. T. J. Scheffler. A portable MPI-based parallel vector template library. Technical Report 95.04, RIACS, 1995.
14. Message-Passing Interface Forum. MPI: a message-passing interface standard, May 1994.
15. 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.
16. L. Semenzato. *An abstract machine for partial differential equations.* PhD thesis, U. of California at Berkeley, 1994.
17. 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.
18. 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.
19. J. Misra. Powerlist: a structure for parallel recursion. *ACM Trans. on Prog. Languages and Systems*, 16(6):1737–1767, November 1994.
20. E. Goubault and T. P. Jensen. Homology of higher-dimensional automata. In *Proc. of CONCUR'92.* Springer-Verlag, 1992.
21. C. C. Squiers and Y. Kobayashi. A finiteness condition for rewriting systems. *Theoretical Computer Science*, 131(2):271–294, 12 September 1994.