Separation of Shape and Data

Thomas Nitsche

***Abstract*—In parallel and distributes systems we have to consider aspects of communication, synchronization and data movement besides the actual algorithmic solution. In order to enable the clear separation of these aspects [29] we have to separate the structure or shape of data structures from the actual data elements themselves. While the latter can be used to describe algorithmic aspects, the former enables us to derive the communication [27].**

In this paper we formalize the notion of shape and data elements and how they can be separated as well as used to reconstruct a valid data structure again for arbitrary container types, i.e. parameterized, nested algebraic data types and arrays. The *shape* function removes the data elements from the container type and replaces them by a dummy element * (or alternatively, their identifier). The *data* function extracts the list of data elements contained in the data structure; while a *re_cover* function can be used to reconstruct a data structure from its shape and a consistent list of data elements. All these functions can be characterized as special cases of a general traversal function operating on the data structure or its shape.

We have used this as the semantical basis for handling overlapping data distributions not only over arrays as in commonly used approaches but over arbitrary (container) types.

***Index Terms*—Parallel Programming, Separation of Concerns, Data Types, Container Types, Shape of Data Structures.**

I. Introduction

Programming parallel and distributed systems requires programmers to consider the distribution of the work respectively the data onto the different processors, as well as synchronization and data exchange, i.e. communication, between them. For this reason the extraction of the data elements from a given data structure, their distribution and communication has to be handled in a parallel program. This can be done explicitly by the programmer in a low-level way as in MPI [31], by the system or combinations thereof.

The decision of which data will be communicated and when depends largely on the characteristics of the parallel computer such as its network topology, bandwidth, etc. To achieve maximum efficiency, many low-level machine- and algorithm-specific details have to be considered. The resulting parallel program is highly problem- and machine-specific, which makes it error-prone if programmed by explicit message-passing [12] and difficult to port to another parallel machine or to reuse the code for another program. Since one of the ideas behind grid computing is to make parallel computing power as easily available on the market as today's electricity or telephony services [11], parallel programs should not be written for a specific parallel machine but rather parameterized using certain architectural parameters. The number of available processors is one such architectural parameter; others are the computing power of each processor, its memory, the network parameter, etc.

In order to achieve a higher level of programming effectiveness and coordination of parallel activities, we can use algorithmic skeletons [3], [7], [8], [9], [30] as a kind of collective parallel operation instead of directly using low-level (point-to-point) communication operations. Well known examples for such generic operations include the *map* skeleton, which applies a function to all data elements, the *zip* skeleton, which combines two (or more) data structures, and the *reduce* skeleton, which combines all data elements to a "sum" using some binary operator:[1]

$$\text{map}(f)([a_1, \ldots, a_N]) = [f(a_1), \ldots, f(a_N)] \quad (1)$$

$$\text{zip}(\otimes)([a_1, \ldots, a_N], [b_1, \ldots, b_N]) = [a_1 \otimes b_1, \ldots, a_N \otimes b_N] \quad (2)$$

$$\text{reduce}(\oplus)([a_1, \ldots, a_N]) = a_1 \oplus \ldots \oplus a_N \quad (3)$$

In an array processing language this would correspond to *f(A), A⊗B,* and *⊕(A)* as used in, e.g., *max(abs(A) – B).*

Operations on locally available data elements now correspond to purely local operations operating merely on the data elements themselves, while accesses to elements residing on other processors imply communication requirements, where the structure of the data type determines the communication structure. If we thus separated the data elements from the data structure (i.e. shape) itself, we could describe the parallel operations independently from the actual data types and use the shape for deriving the communication [27].

The remainder of this paper is organized as follows. Section II describes the concept of shape in an informal way, while section III gives its formal definition and derives related properties. Finally, section IV discusses related work and concludes.

II. Shape and Content of Data Structures

Every data structure has two aspects: its shape and its content (or data). Moreover, there is a mapping from shape to content. The content, for example, is the set of elements in the fields of a matrix, in the nodes of a tree, and so on, while the shape corresponds to the index range of a matrix or the node

Thomas Nitsche was with Technical University Berlin, Germany. He is now with the Research Institute for Communication, Information Processing and Ergonomics (FGAN/FKIE), Neuennahrer Straße 20, 53343 Wachtberg, Germany; e-mail: nitsche@fgan.de.

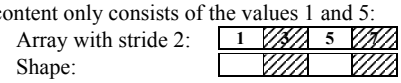
[1] The data elements need an order if the binary operator ⊕ is not associative and commutative.

T. Sobh (ed.), Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, 455–461.

structure of a tree. Separating shape from content allows us to handle the computational operations on the data elements independently of their mapping and ordering aspects.

Let us look at some examples. The simplest one is a single data element, e.g. a real value like π. Its shape is a placeholder for a real value, while its content corresponds to the value of the data element. Similarly, the shape of a tuple, e.g. a pair *&(3.1415, 2)* consisting of a real (*3.1.415*) and an integer (*2*) value, corresponds to the storage place for the values. Its content is the list of the data values. Because the values have different types (*real* and *int*, respectively), we must encode them in a list of the sum of the data types. Thus, formally, the content $\langle in_1(3.1415), in_2(2)\rangle$ is of type $list[real + int]$.

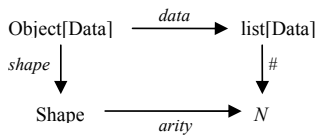
In these simple examples, the shape corresponds directly to the type of the data.[2] The same still holds for arrays. However, in this case it is also sufficient to use the size of the array together with the type of the data elements as shape information. MPI data types allow strides in arrays, meaning that not all array elements are used but only those in a regular interval. In the example below, the shape only contains the elements with even index, while the elements with odd index are omitted. Thus the content only consists of the values 1 and 5:



For nested, inhomogeneous arrays, the shape is merely the nested array structure. The list of data elements can be obtained by flattening [20] the nested arrays, i.e. by putting all data elements into a single list. For algebraic data types, we have to consider the structure, i.e. the graph of their cells, as the shape. In the case of linear lists, this is merely the sequence of cons cells, while for trees the traversal order of the elements within the data structure determines the order of the data elements within the content.

A. Shape Theory

Shape theory [18], [16] formally describes the separation of shape and data and how data is stored within data structures. Shapely types can be formally described as a pullback in the sense of category theory.



(4)

The semantics of (4) are as follows. Given two morphisms *arity: Shape* → *N* and *#: list[Data]* → *N* (where # computes the length of a list), then there exists an – up to isomorphisms – uniquely determined object *Object[Data]* with morphisms *shape: Object[Data]* → *Shape* and *data: Object[Data]* → *list[Data]* such that the above diagram commutes, i.e. it satisfies $arity \circ shape = \# \circ data$. This means that the data structure *Object[Data]* can be constructed from its shape and the list of its data elements.[3]

B. Representing Shapes

In the language FISh (Functional + Imperative = Shape) [17] and its predecessor Vec the shape of a data structure is merely its size, i.e. the number of data elements, together with the shape of the elements. This allows the description of nested homogeneous vectors, all subvectors being required to have the same shape and hence the same length. The shape information thus describes the memory consumption of a certain element because here the shape corresponds to the size (required bytes) of a data element in the case of basic data types like *bool*, *nat* or *real*, and otherwise to the size of a vector (and the shape, i.e. the size of its components.) Shape analysis can be used for memory management and program optimization.

This is similar to the language SAC (Single Assignment C) [14]. It offers multidimensional, homogeneous arrays and – as in APL – dimension-invariant array operations [13]. Arrays in SAC consist of two parts: a data vector containing the data values, and a separate shape vector holding the array sizes in the different dimensions. Access to the shape vector is allowed within the language SAC. Consider, for example, a three-dimensional 2×2×3 matrix *A* with data elements 1, …,12. It is defined as *A = reshape([2,2,3], [1,2,3,4,5,6,7,8,9,10,11,12])*. Then, *dim(A)* returns the number of dimensions of the matrix *A*, i.e. the value 3, while *shape(A)* yields the shape vector *[2, 2, 3]* with the sizes within each dimension. This shape vector can be used to generate other shapes and hence new arrays, e.g. *genarray(min(shape(A), shape(B)))* computes the minimal size within each dimension of matrices *A* and *B*. Note, however, that only homogeneous vectors are allowed in SAC, i.e. all subvectors of a nested matrix must be of the same size.

In Nesl [4] and Nepal [5], [23], the subvectors of a nested array may have different lengths, thus allowing us to model not only dense but also sparse matrices. Internally, the nested vectors are subject to the flattening transformation and are represented by a data vector and a shape vector containing the sizes of the different subvectors [20]. For example, the nested vector [[1,2,3,4,5], [], [6], [7,8,9,10], [11,12]] is represented by the data vector [1,2,3,4,5,6,7,8,9,10,11,12] and the size vector [5,0,1,4,2], which contains the sizes of each sublist of the data.

ZPL [6] offers the concepts of regions that are abstract index sets with no associated data and grids as abstractions of processor sets [10]. Since ZPL is an array processing language, regions are based on arrays and indexes.

C. Shapes of Parameterized Algebraic Data Types

We do not want to deal only with arrays or lists but also with arbitrary algebraic data types, so the size or the index

[2] We will later modify the notion of shape slightly. Instead of storing a dummy value of the corresponding data type, we replace the data values by $* \in 1 = \{*\}$ (cf. Def. 3).

[3] Note that the morphisms have to take into account the traversal order within the data structure. Otherwise, the elements can be arbitrarily re-ordered within *Object[Data]* because permutations are isomorphisms on lists.