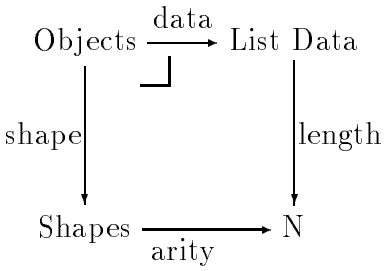
Introduction to Shape Analysis

Milan Sekanina

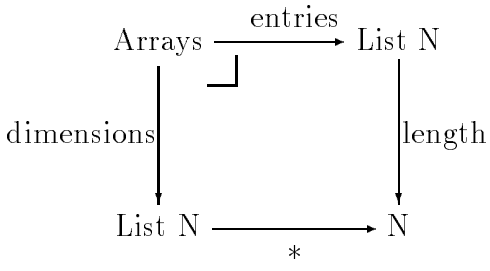
August 1, 1995

1 What is a shape?

The starting point of my thesis is a simple observation: most objects commonly used in computer science, such as arrays, trees or graphs, can be separated into two parts - a "shape" part, describing the "structure" of the object, and a list of data. In this way we can split an array into the list of dimensions and the list of entries, a tree with labeled nodes into the underlying unlabeled tree and a list of labels, a graph with weighted edges into an unlabeled graph and a list of weights (in fact, in case of graphs we need an implicit ordering of edges as well). A closer look at this idea reveals that, for example, not every tuple consisting of a list of dimensions and a list of entries gives us an array - the product of dimensions has to be equal to the length of the list of entries. The same condition holds for trees - the number of nodes has to be the same as the length of the list of labels. So for every "shape" (an unlabeled tree, a list of dimensions) we have its "arity" - the number of data that fits into it. We can think about shape as a structure with some holes in it - the number of these holes tells us how many pieces of data we have to put in to get a complete object. Categorically, this situation is captured by a pullback



So, for example, if we take arrays of natural numbers of arbitrary dimensions, we get



($*$ stands for multiplying all the entries in a list). We could draw similar diagrams for all the other types mentioned before.

Most objects used in the world of computing can be split into their shape and their data. Examples include all inductive types (lists, trees and so on), graphs, records, arrays, sparse matrices (the shape of a sparse matrix is the distribution of its non-zero entries - its "sparsity pattern") and so on. We'll say that such types are "shapely". A formal definition of a shapely type is given in [JC94]. Most prominent among non-shapely types are the function types - it is difficult to imagine what the shape of a function could be.

2 What is it good for?

The idea to separate the shape from the data is not new. Many programming languages (APL, Nial, ...) represent arrays in exactly this fashion - by a list of dimensions followed by a list of data (although often the length of the data list and doesn't have to match the arity). But so far the implications following from this separation hasn't been properly investigated. It seems that possible benefits fall into two main categories: shape polymorphism and shape analysis.

2.1 Shape polymorphism

In languages like ML or Miranda the programmer has the ability to define a function which is polymorphic, i.e. which can be applied to arguments of various types. an example is a function map . The type of map is $(\alpha \to \beta) \to (\text{List } \alpha \to \text{List } \beta)$ – given a function f from α to β and a list as of type List α, map $f(as)$ applies f to every element in list as. The types α and β are type variables which can be then instantiated to any type. So the *data* part varies – this kind of polymorphism is therefore called *data polymorphism* and is common and reasonably well understood. But if we look at map from the viewpoint of shape, then we realize that we don't have to limit ourselves just to lists – for example, we can also take a tree with nodes of type α and apply f to every node and get a tree with the same structure but with nodes of type β. In the same way we may want to apply f to the data of any object of a shapely type. What we have in this case is a different type of polymorphism where the data is fixed, but the shape varies – so called *shape polymorphism*. The two kinds of polymorphism often coexist (map is both data and shape polymoprhic), but sometimes they don't – map f for given f has the data part fixed (given by the type of f), and only the shape can vary. Of course, the problem how to represent shapes now arises. What should be the type of map if its instantiations should include the following types?

$$\text{map} \quad (\alpha \to \beta) \to (\text{Matrix } \alpha \to \text{Matrix } \beta) \tag{1}$$

$$\text{map} \quad (\alpha \to \beta) \to (\text{Tree } \alpha \to \text{Tree } \beta) \tag{2}$$

Obviously, there is no type for such a term in existing languages. To solve this problem is one of the main goals of the shape project. A language P2 supporting shape polymorphism on polynomial types and their fixpoint types was implemented by B. Jay [Jay95].

The main benefits fo shape polymorphism are basically the smae benefits we get from using data polymorphism – greater reuseability of code, shorter and clearer programs.

2.2 Shape analysis

The other major application of shape is in the area of shape analysis. If we can separate the shape from the data, then we might be able to process the shape information separately from, and preferably before, processing the data part. Might is in this case really the right word. Ideally, when there is no interaction between the shape and the data (or at least the interaction goes only one-way – from the shape to the data), and we can infer the shape of the output just from the shapes of the input, independently from the data (such functions and computatitons are called *shapely*), we can first process the shape part, determine all the intermediate shapes, check whether shapes which are supposed to be the same really are the same, in case that some value required in the computation with the data depends only on the shape, compute it and store it – in other words, we can do the *shape analysis* of the program. Let's look, for example, at multiplication of matrices – we take two matrices with dimensions (m_1, n_1), resp. (m_2, n_2), and returns their product with dimensions (m_1, n_2). This is an example of a shapely operation – the dimensions of the output are determined just from the dimensions of the input. So, when we are given the dimesions of the input matrices, we can check whether the dimesions match ($m_2 = n_1$) and infer the dimensions of the product. This is one of the simplest cases of shape analysis. This might seem not to be very useful, but let's suppose that we want to multiply not just two, but several thousands large matrices (as sometimes happens in real applications). This operation can take a very long time, and if the dimensions don't match properly, we would like to know as soon as possible – the shape analyser would determine this in just a few seconds. Moreover, given just the dimensions of the input we can infer other useful information, such as the most efficient order of multiplication (so-called the optimal matrix-parenthesization problem, [KGGK94]), a problem which every efficient algorithm has to solve anyway since it greatly reduces the workload. So we see that program processing now splits into three phases: the compiler phase when we produce an executable code as well as do the type checking and so on, shape analyzing phase when we are given the shapes of the inputs and we infer as much information as possible from them, and finally the data-processing phase when we compute with the actual data.

This seems interesting, but can it have any practical applications? Isn't it the case that at the time when we know the shape, we know the data as well? In that case, what could be the use of separating shape analysis from the actual computation with the data? The time spent on the whole computation would surely be the same? Not quite. There are two points worth mentioning. Firstly, a great deal of shape analysis can be done at compile time, symbolicly, without using the actual shapes. We can determine which shapes have to match, produce a list of such constraints, simplify them if possible (an operation $A \times A^T$ which multiplies a matrix by its transpose is always shouldn't produce any constraints), in some cases infer that no inputs can satisfy the requirements (e.g. operation `zip as (a:as)` trying to zip together two lists of (provably) unequal lengths should produce an error at compile time), determine which shape-dependent information will be required at run-time and so on. Secondly, in many applications (solving sparse systems of linear equations, finite element analysis, above mentioned matrix-parenthesization problem), the part of computation which uses only the shapes is often performed once and the results are then applied to several systems with the same shape [KGGK94].

As we said before, the full use of shape analysis can be exploited only when the operations concerned are shapely, i.e. the shape of the output can be infered just from the shapes on the inputs. There is, of course, a lot of important algorithms which are not shapely. Algorithms such as finding the shortest path in a weighted graph where the re-

sulting path has to be determined from the wieghts. But even in these cases it seems likely that shape analysis can be of some use. Even in non-shapely algorithms we can determine that `zip as(a:as)` is not well formed. It seems that the more the shape interacts with the data, the smaller part of the computation cen be done at the shape-analysing phase, but that there is only a very few cases when the shape analysis would be completely superfluous.

3 What has already been done?

Is shape already used somewhere? Basically, the answer depends on whether we are looking just at algorithms or at languages and compilers used for their implementation.

3.1 Algorithms and shape

In the world around us is a plenty of widely used algorithms which work with shapes independently from the data. Here is several examples of algorithms which has a "shape preprocessing" phase when they work purely with the shapes of inputs.

3.1.1 The optimal matrix-parenthesization problem

Let's consider the problem of multiplying n matrices $A_1, \ldots, A_n$. The order in which the matrices are multiplied greatly affects the total number of operations needed to evaluate their product. As an extreme example, let's take three matrices A, B, C with dimensions (10,1), (1,10), and (10,1), respectively. The total number of multiplications required to evaluate $(A \times B) \times C$ is $10 * 10 * 10 = 1000$, while evaluation of $A \times (B \times C)$ requires only $10 + 10 = 20$ multiplications! Clearly, the second parenthesization is much more efficient way how to evaluate the product.

Solving this problem is an important part of any efficient matrix multilication algorithm, and is an operation computed just from the dimensions of the matrices. Moreover, in many situations we know the dimensions prior to knowing the data.

3.1.2 Solving sparse systems of linear equations

When solving large sparse systems of linear equations, the shape of the system (the sparsity pattern, i.e. the distribution of its non-zero entries) plays a very important role in reducing the number of operations required. The "direct method" (based on Gaussian elimination) used for solving such systems has four phases:

1. **Ordering**, when we permute the rows and columns of the original matrix in order to get a matrix which leads to a system which can be solved faster. The aim is to minimize the *fill-in*, i.e. the number of originally zero positions which would, during factorization, become nonzero. There is many different approaches which can be chosen when solving this problem, see [DER86].

2. **Symbolic factorization**, when we determine the sparsity patterns of resulting matrices, set up the data structures for storing them, allocate memory and so on.

3. **Numerical factorization**, when we, for the first time, work with the data of the matrix and reduce it to a triangular form (using Gaussian elimination). And finally,

4. **Solving the triangular system.**

The first two of these four phases are operations based purely on shape. Often they are performed once and then several systems with the same sparsity pattern are solved. A nice example can be found in [DER86]: in the design of safety features in cars, the dynamics of human body is modelled by a set of time-dependent differential equations. The body is represented by a (sparse) graph of body segments. These segments can't move independently because they are connected at joints. This leads to a set of linear constraints on the positions of body segments. At each step of the simulation (typically there is several thousands of these), this system of linear equations has to be solved. The non-zero entries are different each time, but the sparsity pattern is given by the representation graph, and is fixed. The following table indicates the number of operations required to solve this system (in this example the human body was represented by a graph with 14 nodes) using three different methods: Gaussian elimination which doesn't take into account the sparsity pattern of the matrix, Gaussian elimination which works with the sparsity pattern, and finally Gaussian elimination on a matrix reordered so as to minimize the number of operations:

	The whole matrix	The sparse matrix	The ordered sparse matrix
No. of operations	2106	502	153

We can see that the number of operations required to solve the system decreased very rapidly when we first reordered the matrix. Since the sparsity pattern remains the same throughout the computation, this is done just once at the beginning and the total gain is then very significant.

3.1.3 Finite element method on a parallel architecture

Probably even more then in the world of sequential algorithms, shape analysis could have an impact on the world of parallel programming, mainly in areas such as load balancing, memory allocation, in other words, everywhere where we are dealing just with sizesof the data, its structure or communication patterns . For example, let's suppose we want to solve finite element analysis on a parallel architecture.

Finite element method is a method used for deriving approximate numerical solutions to partial differential equations over a discretized domain. This domain is usually represented by a graph (where adjacent nodes correspond to neighbouring elements). At each step, the value at each point depends on the values of its neighbours at the previous step. So edges of the graph represent necessary communication links. To ensure efficient parallel implementation, it is important to distribute the nodes onto the processors in a way which mimimizes the load imbalance while requires a minimal amount of interprocessors communication. So the aim is to carve the graph into pieces of similar (or, even better, the same) size while cutting through as few edges as possible. This is an NP-hard problem which depends purely on the shape of the graph, and can, therefore, be done as a part of shape analysis.

Basically any parallel program which aims to be efficient (and producing non-efficient parallel programs seems like wasting time) has to deal in some way or another with the problem how to distribute the data onto the processors and avoid any greater imbalance. If some part of the problem could be solved at compile time (and this is where shape analysis could step in), a lot of things could be simplified.

3.2 Languages, compilers and shape

We have seen that we can find some form of shape analysis in many algorithms, both sequential and parallel. Is this fact somehow reflected in language design? Barely. We'll try to summarize some aspects of shape polymorphism and analysis which can be found in existing languages and compilers.

Probably the best place to start looking for some sort of shape analysis and polymorphism could be arrays and their manipulation. Arrays are used everywhere and, therefore, most languages support them, the shape of an array is very simple, just a list of natural numbers, so shape analysis could be relatively easy. Here is a brief summary:

- Virtually no shape analysis is done at compile time.
- Some languages (e.g. Fortran), when accessing an element in an array, check whether the element really lies in the array, i.e. whether its coordinates specify an element of the array or not. This is done at run-time, though.
- Some languages (APL, Nial) support some form of shape polymorphism on arrays: a function can be applied to an array of arbitrary dimensions. But since these languages are not typed, this can't help us very much.

All in all, it looks that support for shape in existing languages is (to say least) very meagre. Bits and pieces can be found, but nothing systematic or going into depth.

What about compilers? Is shape used somewhere? Not often, but there is a few examples.

In [BCF91] is described a method for implementing data-parallel algorithms on MIMD multiprocessors. Data-parallel algorithms have many pleasant properties, but their implementation on MIMD architectures has to overcome some problems connected with reducing incurring overheads. A method suggested in the paper uses technique called *size inference* to analyze the loops in the data-parallel program and to infer which of them have provably same structure. This information then helps us to transform the original algorithm (we won't go into detail here since the only place where something akin to shape analysis takes place is in the size inference phase). Size inference works with symbolic sizes of vectors. Given that we know how the sizes of output vectors depend on the sizes of input vectors for some set of primitive vector operations (e.g. + which adds elementwise two vectors of the same size), we infer the dependency for the size of output vectors of more complex operations. Size inference also checks (for the primitive operations) whether the sizes of inputs satisfy given constraints (since the primitive operations are simple, these constraints are also very simple, e.g. the constraint for + is $s_1 = s_2$, where s_1 and s_2 are the sizes of the inputs), and is able to detect some shape errors at compile time. Some of the methods described in [BCF91] where used in constructing the NESL language [Ble92].

I think that our search for existing use of shape can be summarized as follows: it has a great potential, but so far no systematic approach has been taken to tackle it, and consequently its use in existing languages is very limited and ad hoc. Our aim is to contribute to putting it on firmer foundations and making it a part of language design.

References

[BCF91] G.E. Blelloch, S. Chatterjee, and A.L. Fisher. Size and access inference for data-parallel programs. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 130–144, 1991.

[Ble92] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, January 1992.

[DER86] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press Oxford, 1986.

[Ive62] K.E. Iverson. *A Programming Language.* Wiley, New York, NY, 1962.

[Jay95] C.B. Jay. Polynomial polymorphism. In R. Kotagiri, editor, *Proceedings of the Eighteenth Australasian Computer Science Conference: Glenelg, South Australia 1–3 February, 1995*, volume 17, pages 237–243. Australian Computer Science Communications, 1995.

[JC94] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, Lecture Notes in Computer Science, pages 302–316. Springer Verlag, 1994.

[JJ93] M.A. Jenkins and W.H. Jenkins. *Nested Interactive Array Language Version 6.0.* Nial Systems Limited, 1993.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing : design and analysis of algorithms.* Benjamin/Cummings Pub. Co., Redwood City, California, 1994.

[KLS+94] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, Massachusetts, 1994.

[Mac87] M.E. Mace. *Memory storage patterns in parallel processing.* Kluwer Academic, Boston, 1987.