An introduction to the theories of bulk data types

Simon Dobson
Informatics Department
Rutherford Appleton Laboratory

This note summarises my understanding to date of two closely-related theories for dealing with "bulk" data types: the Bird-Meertens formalism (henceforth abbreviated to BMF) and its extension in categorical data types (CDT). I've been led to this study for two reasons: as interesting formalisms for program derivation in their own right, and as a possible basis for defining useful operations for shared abstract data types. In this note I cover all the essential ideas I've found so far, and include a bibliography of the theories.

Overview

BMF is a theory of program synthesis in which functions are evolved from abstract specifications using a small number of higher-order constructs. Functions are expressed as combinations of functions over data structures, avoiding the explicit use of recursion. This both simplifies proofs of correctness and allows the possibility for efficient (possibly parallel) implementation of the combination operators over a range of data types. A small-scale theory leads to potentially large-scale applications.

The theory does not, however, start from a standpoint of immediate mechanisation. To do so, in Bird and Meertens' view, would severely limit the many ways in which an algorithm may be evolved. There is a wonderful quote in [10]:

> "...we must abandon our fixation on efficiency if algorithmics [Meertens' name for programming as a mathematical activity] is to enjoy a fruitful development. In general, developing an efficient algorithm will require that we first understand the problem, and for this we need simple algorithmic expressions; but to simplify an expression we have to shed our old habits."

so BMF does not initially worry about the efficiency of functions: instead it makes the concise expression of algorithms possible whilst holding out the possibility for verifiable transformations to improve efficiency.

Categorical data types form the basis for generating types and functions which are very closely related to BMF using category theory. The approach is to derive new types in a manner which are guaranteed (from the underlying categorical basis) to be amenable to treatment in BMF-style functions.

The notation used in this paper is substantially that of [2], altered very slightly to take account of the peculiarities of the word processor used to prepare this note.

The Theory of Lists

The essence of BMF is the use of a small set of higher-order functions and operators to derive programs from specifications (a process which Bird[5] refers to as *program calculation*). The advantages of this approach are four-fold:

- the higher-order of functions chosen are (necessarily) very general, and so form a small set of "syntactic" elements for the programmer to learn;
- functions are defined as compositions of functions rather than by explicit use of recursion wherever possible, so full inductive proofs are seldom needed;
- the semantics of the functions allow for parallel implementation; and
- functions manipulate data types in bulk rather than a piece at a time.

The first property means that functions use only a small number of operators which makes for powerful, if terse, expressions. The second reduces the proof obligation taken on by a programming in deriving a function, as he may make use of the known semantics of the kernel functions rather than having to resort to recursion. The third (which is accomplished by avoiding the usual tail-recursive formulation of the kernel functions) gives the theory its current importance as a vehicle for specifying functions across a range of architectures. The final property is also an aid to parallel evaluation, as functions using BMF avoid the Von Neumann bottleneck.

List operators

BMF is primarily concerned with operations on lists. It uses the notion of a *join-list* defined by three functions whose type signatures are:

$$[] \;::\; 1 \to A^*$$

$$[.] \;::\; A \to A^*$$

$$++ \;::\; A^* \times A^* \to A^*$$

where 1 is the one-point (unit) type, A^* is a list of values of type A and ++ is the list concatenation operator. (This form may be contrasted with the more familiar form of *cons-list* found in Lisp, ML and most other functional languages.) Lists are constrained to be composed of values of a single type.

The simplest operation on lists is list length. The rules for this operator are as expected:

$$\#[] = 0$$

$$\#[x] = 1$$

$$\#(a ++ b) = \#a + \#b$$

Notice that the structure of computing # is recursive and, furthermore, follows closely the recursive structure of the type itself. This point will be important later.

The next operation defined for lists is the map operator, denoted $*$, which applies a function to all elements of a list. The definition of map is that

$$f * [] = []$$
$$f * [x] = [f(x)]$$
$$f * (a + + b) = f(a) + + f(b)$$

The map operator may be applied to a single argument to "lift" a function, so the notation $(f*)$ denotes the function which, when applied to a list, returns a list where f has been applied to every element. Notice again that $*$ is defined in terms of the definition of the list type.

The final operator is the filter operator, denoted $<$. This filters a list through a boolean-valued predicate, returning the list of those elemnts which satisfy the predicate.

Reduction operators

The most basic reduction operator is /, which reduces a list using an operator (the same symbol is used for this purpose in APL):

$$\oplus / [a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \ldots \oplus a_n$$
$$\oplus / [a] = [a]$$

(The case of $\oplus / []$ is deferred until later). Using this operator, many useful functions may be defined by composition (in the same way that * may be used "sectionally"):

$$sum = + /$$
$$product = * /$$
$$flatten = + + /$$
$$all(p) = (\wedge /).(p*)$$
$$some(p) = (\vee /).(p*)$$

Identity elements and fictitious values

As observed earlier,

$$[] + + x = x = x + + []$$

[] is referred to as the *identity element* of the ++ operator. Many operators have such identites, including + (identity element 0), * (1) and $\wedge$ (*true*).

Identity elements have several uses. The first use is that they allow the value of $\oplus / []$ to be defined for any operator $\oplus$ which has an identity element e:

$$\oplus / [] = e$$

and indeed this equation may be seen as *defining* the identity element.

The second use of identity follows from the observation that the reduction operator is defined using a single form. In most function programs there exist two functions *foldl* and *foldr* defined such that

$$foldl(f, z, []) = z$$
$$foldl(f, z, x{:}xs) = foldl(f, f(z, x), xs)$$
$$foldr(f, z, []) = z$$
$$foldr(f, z, x{:}xs) = f(x, foldr(f, z, xs))$$

which reduce a list using an operator and a given basic (identity) element. The need for the two functions is that the operator may not be associative and the element may not be a left and right identity.

BMF states that $\oplus / []$ is only defined for associative operators, and furthermore is only defined when the operator has an identity element. This implies that the identity value e of an operator is both a left and a right identity, and combined with associativity this means that there is no needs for the two reducing forms.

([2] defines two "directed" recursion operators whose definitions match those of *foldl* and *foldr*. These seem to have been abandoned in later treatments of BMF – notably [5] and [10]. It must be said, however, that the directed forms can be very useful in many cases. Note that the undirected form is amenable to parallel implementation whilst the directed forms aren't.)

It is usually possible to either determine an identity element for an operator, but in some cases it may be necessary to extend the domain of the operator with a additional or *fictitious* identity value . For example, a maximum value operator *max*, which has no identity within the domain of finite numbers, may be given one by adjoining the fictitious value ∞ to the domain to form a new operator *max'* such that

$$max'(a,b) = a \text{ if b} = \infty$$
$$= b \text{ if a} = \infty$$
$$= max(a,b) \text{ otherwise}$$

Conditionals and selections

BMF adopts the McCarthy conditional form, so using a predicate p

$$(p\rangle f, g)x = f(x) \text{ if } p(x)$$
$$= g(x) \text{ if } \neg p(x)$$

Another operator defined is the selection operator $\lfloor_f$ which is defined over integer-valued functions f such that

$$a \lfloor_f b = a \text{ if } f(a) < f(b)$$
$$= b \text{ if } f(a) > f(b)$$

A possible problem here is that $\lfloor_f$ is under-specified in the case that $f(a) = f(b)$ but $a \neq b$. This non-determinism can be very useful, but care may be taken in performing equational reasoning on functions using $\lfloor_f$.

Homomorphisms and Categorical Data Types

Let us now turn the focus more towards categorical data types. As mentioned above, this approach is very much in the spirit of BMF. As might be expected from category theory, it seeks to generalise the notions of BMF to more complex types by identifying and exploiting common structure.

The most important single contribution of the CDT discipline is the idea of computing using *homomorphisms* (which are present but under-emphasised in BMF). A homomorphism is an operation which is defined so as to follow the structure of the type of its argument. For example, consider a type A with operations ^ and + (injection and binary join respectively). An operation h is a homomorphism if there exists an operation $\oplus$ such that

$$h(a + b) = h(a) \oplus h(b)$$

h may be described as respecting the structure induced by +: moreover if a and b are also composed of smaller objects then the computation of *h(a)* and *h(b)* may also be expressed in terms of $\oplus$. In other words the computation of h is recursive according to the definition of the type A. This result is true for *any* homomorphism: its computation will always follow the recursive structure of the argument data type. The list length and map operators defined earlier are both homomorphisms

The advantage of expresing a function in terms of homomorphisms is that computation is naturally parallel and largely architecture-independent. The communicarion structure of the computation is determined by the structure of the data type. Any "branch" of the computation may proceed in parallel, either down to values of the fundamental types from which the type A was built or to some level of granularity at which it is advantageous to compute a portion of the homomorphism sequentially. This possibility for different decomposition strategies without impacting on a function's meaning is the major claim which CDT (and BMF, for that matter) have to architecture independence.

Extensions of BMF: D-structures

BMF arose as a theory of lists, but can be generalised to other types. The idea comes initially from [10] with the notion of a D-structure, which is closely related to a more general notion of specifying types using algebra.

A *D-structure* can be created for any type D by defining two functions: a function ^ mapping an element of D into a D-structure and a binary join operator + combining two D-structures into another D-structure. Thus each D-structure is a full binary tree where each interior node has exactly two children and each leaf is an element of D. The D-structure S_D may be represented by the domain equation

$$S_D = D + S_D \times S_D$$

New types may be derived by adding requirements to the form of the initial type D and the join operator. For example to obtain a list type we define:

$$S_D = D + \{0\} + S_D \times S_D$$

$$s + 0 = s = 0 + s$$

i.e. augment the base type D with an additional element which is the identity element of the join operator.

If we now progressively add restrictions that the join operator above is associative, commutative and idempotent we obtain the types sequence, bag and set respectively. This illustrates the important correspondence between common data types and algebraic structures, and allows the full power of algebraic reasoning to be brought to bear on problems relating to complex data types. The hierarchy of types thus obtained is termed the *Boom hierarchy* by Meertens[10].

Bulk algebraic data types from scratch: arrays

Another means of extracting bulk data types is to focus on the algebra which defines a type, and then use notions of homomorphisms to generate suitable operations. This approach may be followed without recourse to category theory, although coming to the same effective result.

To illustrate this approach we shall use Miller's algebraic treatment of arrays. For simplicity we shall concentrate of 2-d arrays, although Miller's work generalises to multiple dimensions.

Arrays are defined using a "join-array" algebra:

$$[.]::A \rightarrow |A|$$

$$\Phi::|A|\times|A|\rightarrow|A|$$

$$\Theta::|A|\times|A|\rightarrow|A|$$

(Compare this with the join-lists of BMF.) These functions respectively create a singleton array or join two arrays "besides" or "above" each other. Arrays joined besides each other must have the same height whilst those joined above must have the same width – the type constructors are partial functions over arrays.

An interesting property of the constructors is that they "abide" with each other:

$$(x\Phi u)\Theta(y\Phi v) = (x\Theta y)\Phi(u\Theta v)$$

providing all the expressions are well-defined. This implies that an array, once constructed, has no "memory" of the sequence of constructors used to create it.

One may now define the usual functions such as map:

$$f^*|a| = |f(a)|$$

$$f*(x\Phi y) = (f*x)\Phi(f*y)$$

$$f*(x\Theta y) = (f*x)\Theta(f*y)$$

Other functions may be defined in a similar framework, including the transposition operator *tr*, a "zip" operator $\nabla_{\oplus}$ which applies the $\oplus$ operator pointwise to two arrays, and a generalised cross product operator $X_{\oplus}$ which generates a cross product of two arrays using $\oplus$. From here we may define the usual laws of matrix algebra.

Conclusion

This note has briefly presented the Bird-Meertens formalism and its close relative categorical data types. Both offer a theory of creating "bulk" data types which are then manipulated *en bloc* using a small number of higher-order operators. The structure of the theories, especially the use of homomorphisms, guarantees that efficient parallel implementations could be built. Such functions are completely imdependent of the number of processors on the target system.

Both theories can be extended from their basis in lists to cope with other data types. Sequences, bags and sets have already been encoded (as well as more exotic types such as molecules), so the theories may be seen to underpin several novel programming models.

The parallelism obtained from the theories is implicit and data-parallel. Each function defined as a composition of maps and homomorphisms can assume that the semantics of each is sequentially consistent. Parallelism comes within each bulk operation, and although a smart compiler might spot optimisations across operations there is no support withi the theory for so doing. This is a far more restrictive form of parallelism than is common in the MIMD world.

With this in mind, however, the ideas of bulk algebraically-specified types provide a good starting point both for mapping and for performance-improving transformations.

Acknowledgements

Many if the ideas in this note only became clear from attending a BCS Workshop on Bulk Data Types for Architecture Independence. Dave Skillicorn's notes and presentations were a great help in straightening-out both categorical data types and homomorphisms.

Annotated Bibliography

[1] C.R. Banger and D.B. Skillicorn, "*Flat arrays as a categorical data type*," Department of Computing and Information Science, Queen's University (1992).

(An alternative treatment of arrays to that of [11].)

[2] Richard Bird, "An introduction to the theory of lists," pp.5-42 in *Proceedings of Logic of Programming and Calculi of Discrete Design* (1986).

(Perhaps the most accessible treatment of BMF, although difficult to get hold of. Substantially the same as [5] but more detailed and with a more sensible structure.)

[3] Richard Bird and Lambert Meertens, "Two exercises found in a book on algorithmics," pp. 451-457 in Program specification and transformation, Elsevier Science Publishers (1987).

(Derivation of two functions using BMF. Possibly the most cited example text on the theory, and very accessible.)

[4] Richard Bird, "Algebraic identities for program calculation,", Computer Journal **32**(2) (1989), pp. 122-126.

(More a description of a transformation system *a la* Burstall and Darlington than true BMF. Includes identities relating to the standard higher-order functions such as *map*, *foldl* and *foldr*.)

[5] Richard Bird, "A calculus of functions for program derivation", pp. 287-307 in *Research topics in functional programming*, ed. David Turner, Addison-Wesley (1990).

(Presents BMF around an example of developing an algorithm for run-length encoding. Contains all the basic ideas.)

[6] Richard Bird and O. de Moor, "List partitions," Formal Aspects of Computing **5**(1) (1993), pp. 61-78.

(Uses BMF to derive functions to solve problems expressed in terms of list partitions, of which there are many.)

[7] Murray Cole, "Parallel programming, list homomorphisms and the maximum segment sum problem," CSR-25-93, Department of Computer Science, University of Edinburgh (May 1993).

(An example of deriving a function using BMF, introducing the notion of an "almost" homomorphism.)

[8] Murray Cole, "List homomorphic parallel algorithms for bracket matching," CSR-29-93, Department of Computer Science, University of Edinburgh (August 1993).

(Another example deriving a family of functions using BMF.)

[9] Paul Hoogendijk, "(Relational) programming laws in the Boom hierarchy of types," pp. 163-190 in *Proceedings of the 2nd International Conference on the Mathematics of Program Construction*, Springer-Verlag (June 1992).

(Totally beyond me at present, I'm afraid!)

[10] Lambert Meertens, "Algorithmics: towards programming as a mathematical activity", pp. 289-334 in *CWI Symposium on Mathematics and Computer Science* (1993).

(A manifesto for BMF and algebraic program development in general. Very close to [5] but more thorough. Includes a description of the Boom hierarchy as found in [9].)

[11] Richard Miller, "A constructive algebra of multidimensional arrays," presented at the BCS Workshop on Bulk Data Types for Architecture Independence, London (20 May 1994), but not included in the proceedings.

(An algebraic treatment of arrays in the spirit of BMF, arising from a practical need to specify array computations without committing to an architecture.)

[12] David Skillicorn, "Models for practical parallel computation," International Journal of Parallel Programming **20**(2) (1991), pp. 133-158.

(Survey article including a brief description of BMF, which it compares favourably against other models.)

[13] David Skillicorn, "*Categorical data types*," presented at the 2nd BCS Workshop on Abstract Machine Models for Highly Parallel Computing, University of Leeds (April 1992).

(Introduction to CDT using molecular modelling as an example problem.)

[14] David Skillicorn, "*Parallelism and the Bird-Meertens formalism*," Department of Computing and Information Science, Queen's University (1992).

(Comparative survey of achitecture-independent models of parallel programming in which BMF scores highly.)

[15] David Skillicorn, "Questions and answers about categorical data types," in *Proceedings on the BCS Workshop on Bulk Data Types for Architecture Independence*, London (20 May 1994).

(A very lucid description of the current state of the art in CDT. Particularly good on homomorphisms.)