# A Semantics for Shape

C. Barry Jay \*

#### Abstract

Shapely types separate data, represented by lists, from shape, or structure. This separation supports shape polymorphism, where operations are defined for arbitrary shapes, and shapely operations, for which the shape of the result is determined by that of the input, permitting static shape checking. The shapely types are closed under the formation of fixpoints, and hence include the usual algebraic types of lists, trees, etc. They also include other standard data structures such as arrays, graphs and records.

## 1 Introduction

The values of a shapely type are uniquely determined by their shape and their data. The shape can be thought of as a structure with holes or positions, into which data elements (stored in a list) can be inserted. The use of shape in computing is widespread, but till now it has not, apparently, been the subject of independent study. The body of the paper presents a semantics for shape, based on elementary ideas from category theory. First, let us consider some examples and possible applications. Three classes of examples are inductive types, arrays and records.

All inductive types are shapely. For example, a tree with leaves of type A has shape given by the corresponding unlabelled tree, and data given by its list of leaves (in, say, left-to-right order). Shape also provides another approach to the semantics of parametric polymorphism. Of particular interest for the inductive types is the existence of shape polymorphism (see below) in which a program can be used with arbitrary shapes.

This representation of inductive types also supports greater use of parallelism, since the data is held in a list. W.F. McColl writes of the language GPL : "Such [inductive] types have been deliberately excluded from GPL since they often lead to representations for which it is hard to obtain high degrees of concurrency." [McC94].

The shape of an array is its size. For a matrix this is a pair of natural numbers. More generally, the arrays of dimension k have shape given by a k-tuple, while arrays of arbitrary dimension have shapes given by lists of numbers. Unlike the inductive types, the representation of the shape is very small compared to that of the data, and tends to be quite stable. Hence, shape analysis (see below) may yield substantial benefits in error detection or optimisation, particularly in parallel programming.

Function types are not shapely in general. This may account for some of the tension between the use of higher-order functions and that of arrays. For example, the core of the type systems underlying many sequential functional languages, e.g. [MT91, HPJW92], does not support array types. By contrast, parallel functional languages *must* emphasise arrays, and so often compromise the function types, e.g. by restricting them to be first-order as in Sisal [FCO90], or second-order, using, say, skeletons [Col93]. Shape theory may provide a context in which to explore the trade-offs.

Sparse arrays are also shapely. They can be represented by a list of position-data pairs; the list of positions is the shape. Symbolic computation, as used in Gauss-Jordan elimination, uses the shape to try to minimise the number of non-zero entries [DER86].

Graphs which have a given order on the nodes can be represented as sparse matrices. More directly, their shapes are unlabelled graphs, i.e. relations.

<sup>\*</sup>School of Computing Sciences, University of Technology, Sydney, PO Box 123 Broadway, 2007, Australia.

Programming based on shape would allow the graph, or topology, of a problem to be handled explicitly, instead of being embedded within the structure of a sparse array. Then shape polymorphism would support code re-use despite varying geometries. Also, the processor architecture may be expressible in the same terms, so that compilation could be expressed as a mapping of the shape of the problem onto the shape of the processor.

The shape of a record is the set of its fields (represented as an ordered list). Since records are fundamental to both databases and to object-oriented programming, it may well clarify their semantics. For example, in database theory, missing fields are recorded in the shape. In object-oriented languages, e.g. Eiffel [Mey94], container classes are designed to represent shape.

Having considered a variety of shapes, let us consider how they might be used. Three applications are: code re-use, error-detection and optimisation. The latter two are collectively called shape analysis. While examining them, we will consider whether existing languages and type systems are capable of expressing the desired benefits.

### 1.1 Shape Polymorphism

Shape polymorphism is a novel form of parametric polymorphism which allows operations to be parametrised over shapes, rather than over data. Consider the operation map which applies a function to each element of a list. In existing functional languages, its type is

$$(\alpha {\rightarrow} \beta) {\rightarrow} \alpha$$
 list ${\rightarrow} \beta$  list

where  $\alpha$  and  $\beta$  may range over any types. This *data polymorphism* allows the data ( $\alpha$  and  $\beta$ ) to vary, but uses a fixed shape, **list**. Shape polymorphism fixes the data, but allows the shape to vary, so that, for types A and B, instances of map include

$$(A \rightarrow B) \rightarrow A \texttt{tree} \rightarrow B \texttt{tree}$$

and

$$(A \rightarrow B) \rightarrow A \text{ matrix} \rightarrow B \text{ matrix}$$
.

In each case map f applies f to the data (the leaves or entries), while leaving the shape fixed.

Shape polymorphic operations need not fix the shape; the shape S may be replaced by something constructed from it, such as  $S \times S$ .

It is common for both kinds of parametric polymorphism to co-exist, as for **map**, but neither implies the other: appending of lists is data polymorphic but not shape polymorphic, while mapping of a particular function (e.g. square root) may be shape polymorphic without being data polymorphic.

Shape polymorphism will have several benefits. It will allow operations to be used with arbitrary shapes, including those defined by the user. In array-based languages, it will liberate function definition from issues of size and dimension. In record types, adaptation for missing and additional fields in records can be handled automatically.

Shape polymorphism is not supported in type system **F** [GLT89] which underlies the dominant functional languages. In the extension  $\mathbf{F}_1$  of  $\mathbf{F}$ , the shape could be viewed as a connective  $F : \Omega \to \Omega$  which maps types to types (e.g. maps A to trees of A). Then a possible type for map is:

$$\forall F : \Omega \to \Omega . \forall XY . (X \to Y) \to (FX \to FY) .$$

However, this type is empty. One difficulty is that connectives may be contravariant, in which case the natural type for map would be  $\forall XY.(X \rightarrow Y) \rightarrow (FY \rightarrow FX)$ . More fundamentally, there is no single algorithm for map in **F** that will work for all connectives (even the covariant ones).

The existence of this meaningful (but uninhabited) type for map means that its type can be checked, but the algorithm used to implement it must depend on the type, so that the polymorphism is ad hoc, rather than parametric. If map is supported through type classes, as in Haskell [HPJW92] and Gofer [Jon94], the algorithm is supplied by the user. In Charity [CF92] the algorithm is inferred automatically from the type.

A parametrically polymorphic version of map (and of fold) has been implemented in P2 [Jay95] for the polynomial types (built using products and sums) and their fixpoints, such as lists and trees. A language for all inductive types, such as forests, is currently under construction. It seems likely that some shape polymorphism could be incorporated into existing functional languages.

Some approximations to shape polymorphism can be found in existing languages. Array languages in which nested arrays are represented by their shape and a flat array, e.g. VCODE [BC90] and Qnial [JG89] allow operations over a variety of shape classes. Object-oriented languages with container classes [Mey94] can support shape polymorphic mapping; a routine for performing map fover a variety of containers is given by:

```
from
start
until
off
loop
item.f
forth
end
```

### 1.2 Shape Analysis

Some computations require a high degree of interaction between the data and the shape, e.g. in graph reduction. However, there is a large class of operations in which the interaction is minimal, or even non-existent.

Sometimes the shape of the result is determined by the shape of the arguments, without reference to the input data, though the data of the result may depend on the shape. This is common in data parallel computation [Ski94] and in systolic array computations [Kun82]. For such *shapely operations* it is possible to compute all of the intermediate shapes, as well as that of the result, before examining the data.

This phase-distinction [HMM90] is similar to that occuring in static type checking. As there, we can expect early error detection, before computation on the data begins. The compiler will generate shape constraints from the program, simplifying them where possible. Occasionally, the constraints will be shown to be unsatisfiable, and the error reported before any input is considered. Otherwise, when inputs are provided, their shapes must be shown to satisfy the constraints before the data is processed, e.g. the matrix dimensions must match before multiplication is attempted.

An additional benefit is that knowledge of the shapes of all the intermediate values can be used to optimise large-scale computations, which is particularly important for parallel processing. Shapes carry size information with them, so that some load balancing can be pre-determined. Also, shapes may allow complexity estimates to be made for various sub-tasks, leading to improved scheduling, or determination of non-deterministic algorithms. For example, symbolic computation on sparse matrices manipulates the positions of the non-zero entries to maximise efficiency [DER86].

Even when operations are not shapely, the separation of shape from data may be useful in runtime algorithms. For example, the optimal matrix parenthetisation problem [Ka94, Chapter 9.4.1] uses the shapes alone. Again, when a task (and its data) is split so that it can be shared between two processors, it may be desirable to transmit the smaller portion of data, whose size is easily computed from the shape.

## 1.3 The Semantics

This work is based on the results reported in [JC94].

The setting is a locos, a lextensive category [CLW93] with list objects. (Although this setting is quite weak (e.g. cartesian closure and subobject classifiers are not assumed) the assumption of all finite limits is not reasonable computationally, and can probably be weakened. The characterisation of the shapely type constructors F uses pullbacks such as:



That is, values of type FA are uniquely represented by a shape (of type S) and some data (of type LA) where the length of the data list equals the arity of the shape. The construction of FA is functorial in A and  $\delta$  is a natural transformation. F inherits many of the properties of the list functor; it is a shapely functor. Similarly, the definition of  $\delta$  by a pullback confers additional properties on it, making it a shapely (natural) transformation. The shapely functors and transformations form an attractive setting, which is explored below.

The data could be represented by other structures, such as lazy lists, streams and multi-sets, with varying degrees of success. The emphasis on finite lists is justified by the main theorem of the paper, which asserts that the shapely types are closed under the formation of fixpoints. Hence, the theorem establishes the existence of the inductive types, such as trees, from that of lists alone.

The proof is based on the observation that inputs to sequential computers are given by lists, which are parsed to produce values of fixpoint type. This algorithm, together with those for recognition of the language, and for folding, or reduction, out of it, are instances of a single operation on lists. This link from fixpoint types to parsing is not an accident, but a witness to the link from parsing to context-free grammars. Another link is that data storage (in a shape) becomes equivalent to data entry (using a language, Section 5).

#### 1.4 Towards a shapely type system

There are some open questions about how a type system might be built upon this semantics. One method would represent types exactly as they appear in the semantics, as pullbacks. That is, values would be represented by a shape and a data list, subject to the constraint that the arity of the shape equals the length of the list. This constraint on inputs must be checked by the compiler, and would involve techniques similar to those used in shape analysis. It is not yet clear what limitations should be imposed on the shape and arity to make this feasible.

A shapely type can be represented as a dependent type, i.e. a sigma type:

$$\Sigma_{s:S} A^{a(s)}$$

whose values are given by a shape s: S and some data, of type  $A^{a(s)}$  (where a is the arity).

Of course there are languages that support dependent types, e.g. the Calculus of Constructions [CH88], but the dependence of types on values means that type-checking is performed at run-time, whereas it is our intent that type (and shape) checking be performed as early as possible, preferably during compilation.

Another possibility arises if the lists LA can be represented as  $\sum_{n:N} A^n$  (which is really a type of vectors). Then we can represent the shapely type by the power series

$$\Sigma_{n:N}S_n \times A^n$$

where  $S_n$  represents all shapes having arity n. This approach emphasises the connections with polynomial types [Jay95] and also with combinatorics, particularly the theory of species [Joy81]. For example, if S represents tree shapes then  $S_n$  is the number of trees with n leaves, or if Srepresents all possible arrays, then  $S_n$  is the number of factorisations of n. Unfortunately, this approach is unlikely to be useful algorithmically, since the arity is a poor discriminator among shapes. The paper is structured as follows. Section 1 is the introduction; Section 2 reviews the categorical setting, and establishes some notation and lemmas; Sections 3–7 introduce shapely functors, transformations and types, shape polymorphism, and shape analysis; Section 8 constructs initial algebras, or fixpoints; and Section 9 draws conclusions.

## 2 Locoses

The types and operations are modelled by the objects and arrows of a category C. It must have lists (and the underlying products and coproducts required to define them) and enough pullbacks to work with shapes. Specifying such a class of pullbacks (as was done for the Boolean categories of [Man92]) at this stage would impose an unwelcome burden so, to simplify slightly, we will assume that we have all pullbacks, and work in a lextensive category [CLW93] which has all list objects, i.e. a *locos* [Coc90]. Being extensive is equivalent to requiring that all coproduct diagrams have disjoint (monomorphic) inclusions, and are stable under pulling back. Examples include the usual semantic categories, including those of sets, bottomless complete partial orders, or even topological spaces, any one of which will suffice to illustrate the ideas below.

Let us fix some notation. If  $f: C \to A$  and  $g: C \to B$  are morphisms then  $\langle f, g \rangle : C \to A \times B$  is their pairing. The left and right projections from the product are  $\pi_{A,B}$  and  $\pi'_{A,B}$  (respectively). The unique morphism to the terminal object is  $!_A : A \to 1$ . The symmetry for the product is denoted  $c_{A,B} : A \times B \to B \times A$ . Dually, the coproduct inclusions are given by  $\iota_{A,B} : A \to A + B$  and  $\iota'_{A,B} : B \to A + B$ . If  $f: A \to C$  and  $g: B \to C$  then their case analysis is given by  $[f,g] : A + B \to C$ . The functors  $\Pi, \Sigma : C^n \to C$  denote chosen *n*-fold products and coproducts, respectively, and  $\Delta : C \to C^n$  is the diagonal functor.

The distributive law is witnessed by a natural isomorphism

$$d_{A,B,C}: A \times (B+C) \rightarrow (A \times B) + (A \times C)$$

whose inverse is  $[id \times \iota, id \times \iota']$ .

Subscripts on natural transformations will be omitted unless required to disambiguate an expression.

A *pullback* is a commuting square



such that, for every pair of morphism  $x : X \to A$  and  $y : X \to B$  such that  $f \circ x = g \circ y$  there is a unique morphism  $z : X \to P$  such that  $p \circ z = x$  and  $q \circ z = y$ . When C = 1 the pullback is  $A \times B$  and  $z = \langle x, y \rangle$ . The same pairing notation will be used in the general case.

The list constructor is a functor  $L : \mathcal{C} \rightarrow \mathcal{C}$ . Its constructors are:

nil : 
$$1 \rightarrow LA$$
  
cons :  $A \times LA \rightarrow LA$ 

Further, for any choice of objects B and C, and morphisms  $x : B \to C$  and  $h : A \times C \to C$  there is a unique morphism  $\texttt{foldr}(x,h) : LA \times B \to C$  (called *foldright* of x and h) that makes the following

diagram commute:



It follows that  $[nil, cons] : 1 + (A \times LA) \rightarrow LA$  is an isomorphism, which expresses LA as a coproduct.

From these primitives we can construct the usual family of list operations, whose notation is a mixture of the list notation of [BW88] and categorical notation for monads:

Lf	:	$LA \rightarrow LB$	is map $f$ for $f: A \rightarrow B$
#	:	$LA \rightarrow N$	is the length $map$ !
$\eta$	:	$A \rightarrow LA$	makes singleton lists
0	:	$LA \times LA \rightarrow LA$	is append
snoc	:	$LA \times A \rightarrow LA$	appends a singleton
$\mu$	:	$L^2 A {\rightarrow} L A$	flattens a list of lists
$g^*$	:	$LA \rightarrow LB$	is $\mu \circ Lg$ which extends
			$q: A \rightarrow LB$ to act on lists.

L1 is a natural numbers object N with zero 0 and successor S given by **nil** and **cons** respectively. Then  $\eta =$ **one** and @ = + is addition and  $\mu : LN \rightarrow N$  is summation. Let **Eq** be the equality on N. Many elementary results about lists in locoses can be found in [Jay93b].

The lemma which concludes this section will be needed to prove the main theorem below. Define shunt:  $LA \times LA \rightarrow LA \times LA$  to be

$$LA \times LA \cong LA + (LA \times A \times LA) \xrightarrow{[(\texttt{id}, \texttt{nil}), \texttt{snoc} \times \texttt{id}]} LA \times LA$$

where the isomorphism is given by the coproduct decomposition of the second list and the distributive law. Then

 $\texttt{split}: N \times LA \rightarrow LA \times LA$ 

is given by foldr((nil, id), shunt). It divides a list into two segments, whose first, initial segment, has length given by the first projection (if the list is long enough). Define

take = 
$$\pi \circ \text{split} : N \times LA \rightarrow LA$$
  
drop =  $\pi' \circ \text{split} : N \times LA \rightarrow LA$ 

**Lemma 2.1** The following equations hold:

$$\begin{array}{rcl} @\circ {\tt split} & = & \pi' \\ {\tt split} \circ \langle \# \circ \pi, @ \rangle & = & {\tt id} \end{array}$$

Hence, we have a pullback:



**Proof** Both sides of the first equation equal foldr(id, id). The second is proved similarly.

Given  $x: X \to N \times LA$  for which  $Eq \circ \langle \pi, take \rangle \circ x = true$  then the induced morphism into the pullback is split  $\circ x$ . That  $@\circ split \circ x = \pi' \circ x$  follows from the first equation above while

$$\# \circ \pi \circ \texttt{split} \circ x = \# \circ \texttt{take} \circ x$$
$$= \pi \circ x$$

follows from the assumption about x. The induced morphism into  $LA \times LA$  is unique since  $\langle \# \circ \pi, @ \rangle$  is a monomorphism by the second equation above.

# 3 Shapely Functors

Shapely functors will be defined using two properties of the list functor, its strength [Koc72] and stability, which we will now review.

Construct  $\tau_0$  as

where  $h = \langle \operatorname{cons} \circ (\operatorname{id} \times c), \pi' \rangle$ . Define the *strength* of L to be  $\tau_{A,B} = \pi \circ \tau_0 : LA \times B \to L(A \times B)$ . In **Sets** it maps  $\langle [a_i], b \rangle$  to  $[\langle a_i, b \rangle]$ . It allows parameters to be introduced to a list.

Let us generalise. If  $A = (A_0, \ldots, A_{m-1})$  is an object of  $\mathcal{C}^m$  and B is an object of  $\mathcal{C}$ , then define  $A \times B = (A_0 \times B, \ldots, A_{m-1} \times B)$ .

A strong functor is a functor  $F: \mathcal{C}^m \to \mathcal{C}$  equipped with a natural transformation

$$\tau_{A,B}: FA \times B \to F(A \times B)$$

called its *strength* which satisfies the usual associativity and unicity axioms. These ensure that the result is independent of whether parameters are introduced one at a time, or as a tuple.

More generally still, a strength for  $F : \mathcal{C}^m \to \mathcal{C}^n$  is given by a strength for each of its projections onto  $\mathcal{C}$ .

The list functor preserves all pullbacks, i.e. is *stable* [Coc90]. It does not preserve products, however. The terminal object is mapped to the natural numbers object N which represents the length (or shape) of lists. Also, we can construct a binary product as a pullback over 1 and then apply L to get:



This induces an isomorphism

$$zip: LA \times_{\#} LB \rightarrow L(A \times B)$$
.

from the canonical choice of pullback (indicated by  $\times_{\#}$ ) into  $L(A \times B)$ .

**Definition 3.1** Let  $F : \mathcal{C}^m \to \mathcal{C}^n$  be a functor with a given strength  $\tau$ . Then it is a shapely functor if F is a stable functor. Then F1 is the object of F-shapes and  $\# = F! : FA \to F1$  is the shape of FA. Also  $\mathtt{zip} : FA \times \#FB \to F(A \times B)$  is the canonical isomorphism.

As all finite limits can be constructed from pullbacks and the terminal object 1, it follows that shapely functors preserve as many finite limits as possible while having a non-trivial shape. Here are some examples of shapely functors.

**Example 3.2** If FX = K is a constant functor then it is shapely.

**Example 3.3** The coproduct functor  $+ : \mathcal{C}^2 \to \mathcal{C}$  has strength given by the distributive law. Its stability follows directly from extensivity.

**Example 3.4** The product functor  $\times : \mathcal{C}^2 \to \mathcal{C}$  has strength given by the canonical morphism:

$$(A \times A') \times B \to (A \times B) \times (A' \times B)$$

Stability follows since products commute with all finite limits.

**Example 3.5** Composites of shapely functors are shapely: if  $(F, \sigma) : \mathcal{A} \to \mathcal{B}$  and  $(G, \tau) : \mathcal{B} \to \mathcal{C}$  are shapely functors then  $G \circ F$  is shapely. Stability is immediate, while the strength is given by:

$$GFA \times B \xrightarrow{\tau} G(FA \times B) \xrightarrow{G\sigma} GF(A \times B)$$

**Example 3.6** If  $(F, \sigma) : \mathcal{A} \to \mathcal{B}$  and  $(G, \tau) : \mathcal{A} \to \mathcal{C}$  are shapely then so is  $\langle F, G \rangle : \mathcal{A} \to \mathcal{B} \times \mathcal{C}$  with its strength having components  $\sigma$  and  $\tau$ . Stability reduces to that of F and G separately.

**Example 3.7** Combining the last two results, we see that if  $F, G : \mathcal{A} \to \mathcal{B}$  are both shapely, then so are  $F + G = + \circ \langle F, G \rangle$  and  $F \times G = \times \circ \langle F, G \rangle$ .

**Example 3.8** The list functor is shapely.

Now let us consider some alternatives to lists as a means of storing data. Some of these functors preserve the terminal object, as well as pullbacks. Usually, such additional properties are to be welcomed, but now the object of shapes is trivial, as is the resulting shape theory. Here are two examples.

**Example 3.9** Let SA be the streams (or infinite lists) of A's. They are given by the final coalgebra [Hag83] for the functor  $A \times (-)$ . That is, for each co-algebra  $\alpha : C \rightarrow A \times C$  there is a unique co-algebra homomorphism  $C \rightarrow SA$ . It follows that S has a strength, and S preserves all finite limits. Every stream has the same (infinite) length.

**Example 3.10** Let X be an object such that the exponential (or function type)  $A^X$  exists for all objects A. The functor  $(-)^X$  is strong, and preserves all limits since it is a right adjoint. Combining this with lists yields the shapely functors  $(L-)^X$  and  $L(-^X)$ .

Here are some alternatives to lists with non-trivial shapes.

**Example 3.11** Let  $L^{\infty}A$  be the object of finite and infinite lists, i.e. the final co-algebra for the functor  $FX = 1 + A \times X$ . Its shapeliness follows directly from its definition. Its shape object is  $N^{\infty}$  which in **Sets** is  $N \cup \{\infty\}$ .

**Example 3.12** Let BA be the bags or multi-sets of elements of A. Then B is shapely, but B1 = N so that bags have the same shapes as lists. In other words, the shape does not record the multiplicities of the elements, since these dependent on the data. For this we must turn to the next example.

**Example 3.13** Let W be an object (of *weights*). We can define weighted lists by the functor  $L(W \times -)$  whose object of shapes is LW. If W = N is used to represent multiplicities, then we have an approximation to bags in which the same element may appear twice within a bag. If the weights are positions, then we have, say, a sparse matrix. Note that the weights in  $L(W \times A)$  may be considered as part of either the shape or the data. Hence, the shape must be given explicitly.

Here are a couple of non-examples.

**Example 3.14** The covariant functor  $P_f$ : Sets $\rightarrow$ Sets which constructs finite sets of elements does not preserve pullbacks.  $P_f 1 = 2$  merely determines whether a set is empty or not, which is too little information.

**Example 3.15** The functor  $X^{(-)}$  is contravariant, and so cannot be shapely. Example 3.10 showed how exponentials could be used to construct shapely functors. The functor  $Y^{X^{(-)}}$  is covariant but it does not always preserve pullbacks, e.g. X = Y = 2 in **Sets**.

## 4 Shapely Transformations

A strong natural transformation  $(F, \sigma) \Rightarrow (G, \tau)$  between strong functors is a natural transformation  $\alpha: F \Rightarrow G$  that commutes with the strengths. That is:



A natural transformation  $\alpha : F \Rightarrow G$  is *cartesian* if, for every morphism  $f : A \rightarrow B$ , the following square is a pullback:



Finally, a cartesian, strong natural transformation  $\alpha : (F, \sigma) \Rightarrow (G, \tau)$  between shapely functors is a *shapely transformation*, in which case F is *shapely over* G by  $\alpha$ .

**Example 4.1** Coproduct inclusions are shapely; the strength is given by the distributive law, and cartesian-ness follows from extensivity of the coproduct. Hence type constructors, such as nil and cons are shapely.

**Example 4.2** Projections from the product  $\times : \mathcal{C}^2 \to \mathcal{C}$ , though strong, are never shapely. (Not all transformations of interest are shapely!) Instead, given  $f : A \to C$  and  $g : B \to D$  we have the pullback



which shows that the transformation  $\pi : (-) \times D \Rightarrow id : C \rightarrow C$  is cartesian. That is,  $\pi_{A,B}$  is shapely in A and  $\pi'_{A,B}$  is shapely in B.

**Example 4.3** If  $\alpha : F \Rightarrow G$  and  $\beta : F \Rightarrow H$  are shapely transformations, then their pairing  $\langle \alpha, \beta \rangle : F \Rightarrow G \times H$  is. Similarly, if  $\alpha : F \Rightarrow H$  and  $\beta : G \Rightarrow H$  are shapely, then so is  $[\alpha, \beta] : F + G \Rightarrow H$ .

**Theorem 4.4** Let C be a locos. The powers of C, and the corresponding shapely functors and shapely transformations between them, form a 2-category with respect to the usual operations. **Proof** If  $\alpha : (E, \sigma) \Rightarrow (G, \tau)$  and  $\beta : (G, \tau) \Rightarrow (H, \rho)$  are shapely then so is  $\beta \circ \alpha : F \Rightarrow H$  (whose

**Proof** If  $\alpha : (F, \sigma) \Rightarrow (G, \tau)$  and  $\beta : (G, \tau) \Rightarrow (H, \rho)$  are shapely then so is  $\beta \circ \alpha : F \Rightarrow H$  (whose components are  $(\beta \circ \alpha)_A = \beta_A \circ \alpha_A$ . If  $\alpha : F_1 \Rightarrow F_2$  and  $\beta : G_1 \Rightarrow G_2$  are shapely then so is their horizontal composite  $\beta * \alpha : G_1F_1 \Rightarrow G_2F_2$  (whose components are given by  $(\beta * \alpha)_A = \beta_{F_2A} \circ G_1\alpha_A$ ). In each case, strength and cartesian-ness follow by pasting.

**Proposition 4.5** Let  $F : \mathcal{C}^2 \to \mathcal{C}$  be a shapely functor and let  $G, H : \mathcal{C} \to \mathcal{C}$  be any functors. Suppose that for each object B the transformation  $\alpha_{A,B} : F(A,B) \to GA$  is cartesian in A, and for each object A the transformation  $\beta_{A,B} : F(A,B) \to HB$  is cartesian in B. Then

$$\langle \alpha_{A,B}, \beta_{A,B} \rangle : F(A,B) \rightarrow GA \times HB$$

is cartesian in both A and B.

**Proof** Consider a commuting square:

$$\begin{array}{c} X \xrightarrow{\langle x, y \rangle} GA \times HB \\ \downarrow \\ z \\ \downarrow \\ F(A', B') \xrightarrow{\langle \alpha, \beta \rangle} GA' \times HB' . \end{array}$$

Then x and z induce a unique morphism  $x': X \to F(A, B')$  by the cartesian-ness of  $\alpha$ . Similarly y and z induce a morphism  $y': X \to F(A', B)$ . Now the commutativity of

$$\begin{array}{c} X \xrightarrow{x'} F(A,B') \\ \downarrow \\ y' \downarrow \\ F(A',B) \xrightarrow{F(\mathsf{id},h)} F(A',B') \end{array}$$

and the stability of F induce the desired morphism  $X \rightarrow F(A, B)$ .

**Theorem 4.6** If  $\alpha: F \Rightarrow G$  and  $\beta: H \times G \Rightarrow G$  are shapely transformations, then

$$\gamma_A = \texttt{foldr}(\alpha_A, \beta_A) : LHA \times FA \rightarrow GA$$

is a shapely transformation.

**Proof** Adapt the proof of the special case [Jay93a, Theorem 2.6].

Hence the main operation by which new operations are constructed preserves shapeliness. For example, @ = foldr(nil, cons) and  $\mu = \texttt{foldr(nil, @)}$  are shapely, as is  $\tau_0$ . However  $\tau_{A,B} = \pi \circ \tau_0$  is shapely in A but not B.

The following lemma shows how strength and cartesian-ness interact.

**Theorem 4.7** If  $\alpha : F \Rightarrow G$  is cartesian and  $(G, \tau)$  is shapely then there is a unique strength  $\sigma$  for F such that  $(F, \sigma)$  and  $\alpha$  are both shapely.

**Proof** If F has a strength  $\sigma$  that makes  $\alpha$  strong then Fig. 1 must commute. Since the square is a pullback, this determines  $\sigma$  uniquely. Conversely, this pullback can be used to define  $\sigma$  whose desired properties are all inherited from  $\tau$ .



Figure 1: The strength of F.

**Corollary 4.8** There is an equivalence between functors shapely over  $(G, \tau)$  and morphisms into its shape G1.

**Proof** If  $\alpha : (F, \sigma) \Rightarrow (G, \tau)$  is shapely then  $\alpha_1 : F1 \rightarrow G1$  is the desired morphism. Conversely, given  $a : S \rightarrow G1$  define a functor F by the pullback:



The action of F extends to a functor as follows: given  $f : A \rightarrow B$  then  $\# \circ Gf$  equals # which implies that the pullback defining FA can be constructed in stages, as in Fig. 2.



Figure 2: Cartesian-ness of  $\alpha$ .

The equations for functoriality all follow directly from the universal properties, and the cartesian-ness of  $\alpha$  is immediate from the diagram. Now the theorem shows that F and  $\alpha$  are shapely. The constructions are inverse (up to isomorphism).

## 5 Shapely Types

A functor  $F : \mathcal{C} \to \mathcal{C}$  which is shapely over lists is a shapely type constructor. More generally, a functor  $F : \mathcal{C}^m \to \mathcal{C}^n$  is a shapely type constructor if it is shapely over  $\Delta \Pi L$ .

The latter functor may need some unravelling. The list functor on  $\mathcal{C}^m$  acts on each component separately. Hence, if  $A = (A_0, \ldots, A_{m-1})$  is an object of  $\mathcal{C}^m$  then

$$\Pi LA = LA_0 \times \ldots \times LA_{m-1}$$

Then  $\Delta$  makes one copy of this for each component of  $\mathcal{C}^n$ .

Thus, if  $F = \langle F_0, \ldots, F_{n-1} \rangle$  and  $\delta = \langle \delta^0, \ldots, \delta^{n-1} \rangle$  is the given shapely transformation then there are pullbacks:



 $\delta_1^i$  is also known as the *i*th *arity* of F1. The significance of this pullback is that values of type FA are given by a shape (of type F1) and some data (of type  $\Pi LA$ ) for each *i*, such that the arity of the shape equals the amount of data supplied. In other words, shapes can be thought of as having fixed numbers of holes or entries of each type, which must be filled in by data.

Corollary 4.8 shows that shapely type constructors can be specified by giving their shape with its arity. Let us consider a single example in some detail.

Trees with labelled leaves and nodes are described by:



The shape of a tree is the corresponding unlabelled tree. It is worth emphasising here that pullbacks are only defined up to isomorphism, so that they only provide a specification of an object, not an implementation. This level of abstraction can be a real benefit, but unanswered questions can remain. In the current case, the pullback does not determine whether the list of leaves represents them in left-to-right or right-to-left order, or in some more arcane fashion. This issue will only be resolved when the constructors for the type are given, which in turn are determined by their action at the level of shapes.

For notational simplicity, we will illustrate this by trees TA with leaves of type A but unlabelled nodes. Define  $leaf_A : A \rightarrow TA$  as in Fig. 3.

If the leaves are listed from left-to-right then  $node_A$  is given by Fig. 4. The number of leaves in the result is the sum of those in the sub-trees, while the lists of leaves must be appended. Note that if **leaves** were to represent the leaves from right to left then the order of the lists must be swapped before appending. Thus, the choice of @ for the **node** constructor fixes the representation of the leaves.

Of course, these constructions all depend on the prior existence of the shape, its arity and constructors. The existence of such inductive types in an arbitrary locos will be established in Subsection 8.3. Here are some examples which are not inductive types.

**Example 5.1** Languages generated by a grammar are shapely, with the data given by lists of terminal symbols, and the shape given by the production.



Figure 3: leaf.



Figure 4: node.

**Example 5.2** Define the matrices MA with entries of type A by the following pullback:



That is, a matrix is a list with a given factorisation of its length. The underlying shape of a matrix is its dimensions. Corollary 4.8 shows that M and entries are shapely. Matrix multiplication, and general operations of linear algebra can all be produced from this definition [Jay93a].

**Example 5.3** Arrays of dimension k generalise the matrices. They are constructed from the arity  $N^k \rightarrow N$  which computes the product of the sizes. The types of arrays of all possible dimension have shapes given by lists of numbers LN. The length of the list determines the dimensionality of the array. The usual array operations of updating, etc. can be defined using operations on the data list, using the shape to determine the necessary positions.

Banger and Skillicorn [BS93] give a categorical semantics for arrays, which are represented by their dimensions and a stream. The lack of a constraint linking sizes and data limits the potential for error-checking.

Manes [Man92] interprets matrices as the morphisms of a category, whose objects are sizes. The result is a universe of matrices, without distinguishing the matrices as one data type among many.

**Example 5.4** Sparse arrays can be represented as a list of position-datum pairs, the result of zipping a position list, the shape, with a data list.

**Example 5.5** The underlying shape of a graph is an unlabelled graph or relation. There is no canonical order on the nodes of a graph (or elements of a set), so that one must be imposed. Then a relation can be represented as a symmetric boolean matrix. A more efficient representation uses an upper triangular matrix. Thus relations (on finite orders) are given by the following pullback



where **choose2** maps n to n(n-1)/2. It follows that  $\# : R \to N$  determines the number of nodes, and  $\# \circ \delta$  determines the number of edges in an unlabelled graph.

Thus graphs with nodes of type A and edges of type B are given by the pullback:



Note that once an order has been chosen for the nodes then there is an implicit order on the edges.

There is a second means of representing these graphs, where the shape is given by the number of nodes:

$$\begin{array}{c} G(A,B) & \longrightarrow & LA \times L(B+1) \\ & & \downarrow \\ & \downarrow \\ & & \downarrow \\ & & \downarrow \\ & & \downarrow \\ & & \downarrow$$

Now there is an "edge" between any pair of nodes, but those labelled by  $\iota' : 1 \rightarrow B + 1$  are only dummy edges. The result is a sparse matrix with dummy labels whenever there is no edge.

Directed graphs are handled the same way, except that there are now  $n^2$  possible edges. For directed multi-graphs it is necessary to have an order on the edges connecting a single pair of vertices. Then for each pair of nodes there is a list of labels. One representation is



where s is the squaring function.

**Example 5.6** Consider records whose field names are of type X. Assume that the fields have a canonical order (e.g. lexicographic). For simplicity, assume that all the fields must have the same type. The shape of a record is the finite set of its fields, with arity given by cardinality. We can represent such a set by a list of fields, in correct order. Then the data list represents the values assigned to the fields. When there are several types of data then X must be partitioned into subobjects consisting of fields that must have the same type. The usual operations of adding, deleting and changing fields can be defined using the properties of pullbacks.

**Example 5.7** Weighted lists are shapely over lists in the obvious way. Hence, any functor shapely over weighted lists is automatically shapely over lists.

**Example 5.8** Shapely type constructors are closed under composition. If  $\alpha : F \Rightarrow L$  and  $\beta : G \Rightarrow L$  are shapely type constructors then

$$GFA \xrightarrow{(\beta * \alpha)_A} LLA \xrightarrow{\mu_A} LA$$

makes GF a shapely type constructor. For example, trees of arrays form a shapely type.

Let us consider what happens if lists are replaced by one of the other candidates for data storage presented in the previous section. The last example above shows that neither weighted lists nor graphs add anything new.

**Example 5.9** Consider a shapely natural transformation  $\alpha : F \Rightarrow B$  over the bag functor. Values in FA are determined by a shape, and by a bag of data, but the only connection between them is that the number of items in the bag equals the arity of the shape. This does not seem very interesting.

**Example 5.10** If G preserves the terminal object then any shapely natural transformation  $\alpha$ :  $F \Rightarrow G$  makes F isomorphic to  $F1 \times G(-)$  since the pullback defining  $\alpha$  reduces to a product diagram. That is, there is no constraint linking the shape and the data. For example, matrices which are infinite in both dimensions have no shape; they are isomorphic to streams. Similarly, necessarily infinite trees have non-trivial shapes, but the data is always a stream; there is a shape but no constraint.

**Example 5.11** If the data is stored in a lazy list  $L^{\infty}A$  then we can construct lazy data types such as lazy trees, whose shapes are given by possibly infinite trees, and lazy arrays, which may be infinite in some (or all) dimensions. Of course, some care must be taken in choosing the order of the entries in the data list.

The definition of shapely types is based on the image of a structure with holes in which different types of data can be stored; this is represented by functors which are shapely over a product  $\Pi LA = LA_0 \times LA_2 \times \ldots \times LA_{m-1}$  of lists, one for each type of data. An alternative image, to be exploited below, takes data entry as the primitive notion. That is, an input string is of type

$$L\Sigma A = L(A_0 + \ldots + A_{m-1})$$

where the data of different types may be mixed together. This leads to the consideration of functors which are shapely over  $L\Sigma$  instead of  $\Pi L$ . Both intuitions are useful, so which is to be preferred? Fortunately, the resulting notions of shapeliness are equivalent, as the following proposition shows.

**Proposition 5.12**  $\Pi L$  and  $L\Sigma$  are each shapely over the other. **Proof** Clearly, there is shapely natural transformation  $\Pi L \Rightarrow L\Sigma$  given by

$$\Pi LA_k \xrightarrow{\Pi L\iota_k} \Pi L\Sigma(A_i) \xrightarrow{a} L\Sigma(A_i)$$

where  $\iota_k$  is the kth inclusion to the sum, and a is the m-fold append of the lists. As each of these transformations is shapely, so is the result.

Conversely, a list whose entries are of all the different types can be filtered to produce a tuple of lists whose entries are all of the same type. Define the natural transformation  $check_{A,B}$  by:

$$(A + B) \times LA \xrightarrow{d} (A \times LA) + (B \times LA) \xrightarrow{[cons, \pi']} LA$$

It is shapely in A whence  $\kappa_1 = \text{foldr}(\text{nil}, \text{check})$  is, too. This can be generalised to define the shapely transformation  $\kappa_i : L\Sigma A \rightarrow LA_i$  which strips from a list all entries which are not from  $A_i$ . Then the obvious *m*-fold generalisation of Proposition 4.5 shows that

$$\kappa = \langle \kappa_i \rangle : L\Sigma \Rightarrow \Pi L$$

is a shapely transformation.

## 6 Shape Polymorphism

The separation of data from shape in shapely types allows operations to be defined by giving their action on each part separately, as occurred in defining the node operation on trees. Parametric polymorphism arises when one of these operations can be given independently of the types involved for the other. One version of this, *data polymorphism* is already well understood. It occurs when the action on the shape is independent of the data. An example of this is given by the balancing of a binary tree in Fig. 5 where the shape is fixed to be binary trees, but the data may be of any type.



Figure 5: Balancing a tree.

The other form of parametric polymorphism, called *shape polymorphism* is completely new. In this case the data is fixed while the shape can vary. For example, summing the data values fixes the data type to be the numbers N but can be defined for any shape, as in Fig. 6.

Of course, it often happens that data and shape polymorphism co-exist; the canonical example is map. Typically, it is only applied to lists, but can be defined quite generally, as in Fig. 7, since the shape remains fixed while the data changes. It has been implemented for a large class of inductive types in **P2** [Jay95].

It is not yet clear what an appropriate class of shape polymorphic operations might be. Other shape polymorphic operations include  $zip : FA \times_{\#} FB \rightarrow F(A \times B)$  and the strength  $\tau$  as well as the basic operations such as projections and inclusions.

Other examples are the *pointwise operators* introduced by example in [Jon90] and defined in [Jay93a]. These iterate an endomorphism at each entry in a shape. The number of iterations at each entry is determined by a *weight* on the shape i.e. a morphism  $F1 \rightarrow FN$ . Particular shapes



Figure 6: Summing over a shapely type.



Figure 7: Shape polymorphic map.

may have special weights (e.g. one can weight each leaf in a tree by its depth) but weights on lists yield shape polymorphic operations. Examples include weighting each entry by the length of the list, or by its position. When the discrete Fourier transform is defined using pointwise operators (*ibid.*) then it is seen to be shape polymorphic.

# 7 Calculating with shape

Interaction between shape and data in a computation may be a major consideration (e.g. in graph reduction) or be non-existent (e.g. when mapping). The less interaction there is, the greater the benefits of separating one from the other. The simplest case is when there is no interaction. Though few in number, such operations are used often, e.g. map, zip.

More realistically, we would like to be able to perform all shape computations before looking at the data, though the shape could influence the data. For example, the size of an array may appear as a parameter in the data calculations, as in the Fourier transform. These are the *shapely operations*. Semantically, they are given by operations  $f : FA \rightarrow GB$  between shapely types, for which there is an operation  $u : F1 \rightarrow G1$  between their shapes such that the following diagram commutes:



For example, consider the decomposition of a tree into either a leaf or a pair of sub-trees:

The shape of the result is determined by that of the input, but in order to know where to break the list of leaves, the number n of leaves in the left sub-tree is required. Shape processing would add the computed value of n to the environment prior to the data-processing.

If a program is built from shapely operations then all of the intermediate shapes can be performed before considering any data. Such information can be used to optimise run-time code, e.g. by performing load balancing or determining communication patterns. For example, symbolic computation is an important technique in optimising Gauss-Jordan elimination on sparse matrices, since a bad choice of pivots may dramatically increase the number of non-zero entries in the array. The structure of the shape (such as the depth of a tree) may even be useful in making complexity estimates.

Even when the shape depends on the data, their separation may be productive, if the benefits obtained from having the shape outweigh the overheads of maintaining it. For example, consider a distributed divide-and-conquer algorithm in which one part of the divided problem must be passed to another processor. The choice of part may be easy if the shapes are known.

## 8 Initial Algebras

The main purpose of this section is to show how the existence of lists can be used to infer the presence of all the other inductive types, constructed as initial algebras for shapely type constructors. The underlying intuition is that an inductive type T can be identified as a language in some alphabet  $\Omega$ . That is, T is a sub-object of  $L\Omega$  which is distinguished by a recogniser, represented by a morphism  $\chi : L\Omega \rightarrow bool$  which maps T to true and all else to false. The situation is captured by the following pullback:



Here true is represented by  $\iota : 1 \rightarrow 1 + 1 = 2$ . The recogniser  $\chi$  will be constructed using techniques from parsing.

#### 8.1 Endofunctors

The theory of context-free languages and parsing is typically introduced without considering much data. For example, the language of trees is handled by considering the problem of matching

brackets, i.e. of identifying unlabelled trees. We will mimic this approach, by first considering shapely endofunctors.

Let  $\delta: F \Rightarrow L$  make F a shapely type constructor. An F-algebra is given by an object C and an F-action  $\gamma: FC \rightarrow C$ . If  $\gamma_0: FF_0 \rightarrow F_0$  makes  $F_0$  an initial F-algebra then there is a unique F-algebra homomorphism fold $\gamma: F_0 \rightarrow C$ .

Define  $\Omega = F1$ . Then  $L\Omega$  represents words in Polish notation and the initial algebra  $F_0$  will be the sub-object of  $L\Omega$  of well-formed words. For example, if  $FX = 1 + X \times X$  then  $F_0 = T1$ is the unlabelled binary trees and  $\Omega \cong 2 \cong \{l, n\}$  where *l* represents a leaf, and *n* represents a node. For example, *nlnll* represents a tree whose left branch is a leaf, and whose right branch is the smallest possible tree with a node.

While trying to recognise well-formed expressions (elements of  $F_0$ ) it is necessary to keep track of how many well-formed sub-expressions have already been produced. This will be done using a morphism  $\chi_1 : L\Omega \rightarrow L\Omega \times N$  which maps a word v to a pair  $\langle w, n \rangle$  where n represents the number of well-formed expressions found, and w is that part of v which could not be parsed. Then the initial algebra can be given by:



That is, if  $\chi_1$  parses everything and produces a single expression then the string represents an expression in  $F_0$ . (Then  $\chi$  is given by composing  $\chi_1$  with the appropriate test  $L\Omega \times N \rightarrow bool$ .)

Actually,  $\chi_1$  is a special case of the operation  $\chi_C : L\Omega \to L\Omega \times LC$  which can be defined for any *F*-algebra  $(C, \gamma)$  (e.g. the terminal object). Instead of just producing the number of well-formed expressions,  $\chi_C$  constructs their images in *C* under fold  $\gamma$ .

## 8.2 Parsing

Before constructing  $\chi_C$  let us preview some of its uses. First, the restriction of  $\chi_C$  to  $F_0$  will yield fold  $\gamma: F_0 \rightarrow C$ . Second, when C is itself  $F_0$  then  $\chi_{F_0}: L\Omega \rightarrow L\Omega \times LF_0$  is the standard notion of a parser, since  $F_0$  is the type of parse trees. In short, from  $\chi_C$  is derived both the recogniser and parser for the initial algebra, and also the algebra homomorphisms fold  $\gamma$  out of it. For these reasons, we will generalise the usual terminology, and call  $\chi_C$  a parser.

**Lemma 8.1** The test  $Eq \circ \langle \pi, take \rangle \circ (\delta \times id) : \Omega \times LC \rightarrow bool$  recognises the subobject

$$(\mathsf{id} \times @) \circ (\langle \#, \delta \rangle \times \mathsf{id}) : FC \times LC \to \Omega \times LC .$$
(1)

**Proof** First show that the following diagram is a pullback:

$$\begin{array}{c|c} FC \times LC & \underbrace{(\operatorname{id} \times @) \circ (\langle \#, \delta \rangle \times \operatorname{id})}_{\delta \times \operatorname{id}} & \Omega \times LC \\ & & \downarrow & \downarrow & \downarrow \\ \delta \times \operatorname{id} & & \downarrow & \delta \times \operatorname{id} \\ & & LC \times LC & \xrightarrow{\langle \# \circ \pi, @ \rangle} & N \times LC \end{array}$$

Then paste it to that in Lemma 2.1.

The test just constructed picks out those pairs where the arity of the  $\Omega$  is no greater than the length of the list. Then there are enough resources to construct something of type FC with a list

of C's left over. Let  $\iota' : QC \rightarrow \Omega \times LC$  be the pullback of this test along false. Then we can define  $\zeta_C : \Omega \times LC \rightarrow L\Omega \times LC$  to be:

$$(FC \times LC) + QC \xrightarrow{[(\texttt{nil}, \texttt{cons} \circ (\gamma \times \texttt{id})), (\eta \times \texttt{id}) \circ \iota']} L\Omega \times LC .$$

In words, the action of  $\zeta_C$  is as follows. If the arity of the shape is no more than the length of the list, then take enough of the list to form something of type FC, apply  $\gamma$  and cons the result onto the remaining list; the list of  $\Omega$ 's is **nil**. If the arity exceeds the length then make the shape a singleton list, and leave that of C's alone.

From  $\zeta_C$  we can construct an action  $\theta_C : \Omega \times L\Omega \times LC \rightarrow L\Omega \times LC$  as follows. Decompose its source as  $(\Omega \times LC) + (\Omega \times \Omega \times L\Omega \times LC)$  (by splitting  $L\Omega$  along nil and cons) and then

 $heta_C = [\zeta_C, \mathtt{cons} \circ (\mathtt{id} imes \mathtt{cons}) imes \mathtt{id}]$  .

The sense is that if the middle component is not **nil** then the "parse" has already failed, so just cons the new  $\Omega$  onto the existing list. Otherwise, apply  $\zeta_C$ . Finally,  $\chi_C = \texttt{foldr}(\langle \texttt{nil}, \texttt{nil} \rangle, \theta_C)$ .

**Lemma 8.2** If  $h: (C, \gamma) \rightarrow (C', \gamma')$  is an *F*-algebra homomorphism then

$$(\operatorname{id} \times Lh) \circ \chi_C = \chi_D$$
 .

Hence,  $F_0$  can be constructed in stages, as in Fig. 8.



Figure 8: The initial algebra.

**Proof** Clearly Q is a functor and  $\iota'$  is a natural transformation. Hence  $\zeta, \theta$  and  $\chi$  are natural with respect to F-algebra homomorphisms.

A couple of lemmas will be required before proving that  $F_0$  is an initial *F*-algebra with h =**fold**  $\gamma$  the unique algebra homomorphism to *C*.

**Lemma 8.3**  $\chi_C \circ \phi^* = \langle \texttt{nil}, Lh \rangle : LF_0 \rightarrow L\Omega \times LC$ .

**Proof** It suffices to show that both sides of the equation are foldright of  $\langle nil, nil \rangle$  and foldr(id,  $\theta_C$ )  $\circ$  ( $\phi \times id$ ). The nil case is trivial. The cons case for the left-hand-side is in Fig. 9.

The comparable diagram for the right-hand side is in Fig. 10. All of its cells commute, except the left-hand cell on the lower edge. To resolve this, a digression is required.

The lower edge of this cell is  $foldr(\langle nil, id \rangle, \theta_C)$  now denoted by f. The following equations may be proved in sequence, using elementary arguments.

$$\begin{array}{rcl} \chi & = & f \circ \left< \texttt{id}, \texttt{nil} \right> \\ f \circ \zeta & = & \zeta \end{array}$$



Figure 9: Representing  $\chi_C \circ \phi^*$ .



Figure 10: Representing  $\langle nil, Lh \rangle$ .

$$\begin{array}{rcl} f \circ \theta & = & \theta \circ (\operatorname{id} \times f) \\ f \circ f & = & f \\ f \circ \chi & = & \chi \\ f \circ (\operatorname{id} \times @) \circ (f \times \operatorname{id}) & = & f \circ (\operatorname{id} \times @) \end{array}.$$

It follows that both sides of the lemma are fixed by post-composing with f. Hence, it suffices to show that the recalcitrant cell commutes upon post-composition with f. Now

$$\begin{array}{lll} f \circ (\texttt{id} \times @) \circ (\chi \times \texttt{id}) &=& f \circ (\texttt{id} \times @) \circ (f \circ \langle \texttt{id}, \texttt{nil} \rangle \times \texttt{id}) \\ &=& f \circ (\texttt{id} \times @) \circ (f \times \texttt{id}) \circ (\langle \texttt{id}, \texttt{nil} \rangle \times \texttt{id}) \\ &=& f \circ (\texttt{id} \times @) \circ (\langle \texttt{id}, \texttt{nil} \rangle \times \texttt{id}) \\ &=& f \ . \end{array}$$

The following lemma shows how to de-parse, i.e. reverse the parse into  $F_0$ . For notational clarity, the subscript  $F_0$  will be contracted to 0 from now on, e.g.  $\chi_{F_0}$  becomes  $\chi_0$ .

Lemma 8.4  $@\circ(\operatorname{id}\times\phi^*)\circ\chi_0=\operatorname{id}_{L\Omega}.$ 

**Proof** The commutativity of the lower square in Fig. 11 follows by a case analysis of the definition of  $\theta_0$ .



Figure 11: De-parsing.

Now let us return to the *F*-algebra structure of  $F_0$ . Consider Fig. 12. Lemma 8.3 implies the commutativity of its rear face. The right and bottom faces commute by the definitions of  $\chi$  and  $\theta$ . Hence, there is an induced *F*-action  $\gamma_0$  that makes *h* a homomorphism. (Of course, the definition of  $\gamma_0$  and its action is not dependent on the particular choice of *C*, since we can always work over the algebra C = 1.)



Figure 12: The action of the initial algebra.

It remains to prove its uniqueness. Let  $h: (F_0, \gamma_0) \rightarrow (C, \gamma)$  be any *F*-algebra homomorphism. Then  $\operatorname{fold} \gamma = h \circ \operatorname{fold} \gamma_0$  by Fig. 13. Hence it suffices to prove that  $\operatorname{fold} \gamma_0 = \operatorname{id}$ .



Figure 13: Factorisation of h.

Now

$$\begin{array}{lll} \phi \circ \texttt{fold}\gamma_0 &=& @\circ(\texttt{id} \times \phi^*) \circ \langle \texttt{nil}, \eta \rangle \circ \texttt{fold}\gamma_0 \\ &=& @\circ(\texttt{id} \times \phi^*) \circ \chi_0 \circ \phi \\ &=& \phi \end{array}$$

where the last equation holds by Lemma 8.4. Hence  $fold\gamma_0 = id$  since  $\phi$  is a monomorphism.

### 8.3 The General Case

The construction of  $F_0$  in the previous section shows how to build particular types, but in order to obtain type constructors we must construct initial algebras in a parametrised fashion.

A functor  $F : \mathcal{C}^m \times \mathcal{C}^n \to \mathcal{C}^n$  can be used to represent a system of (parametrised) domain equations [SP82], whose solution is can be found by constructing, for each object A in  $\mathcal{C}^m$ , an initial algebra  $\alpha_A : F(A, F^{\dagger}A) \to F^{\dagger}A$  for the functor F(A, -).

For example, if  $F(A, X) = A + X \times X$  then  $F^{\dagger}A = TA$  is the binary trees on A; the leaf and node constructors are given by the coproduct inclusions

 $A \longrightarrow F(A, TA) \longleftarrow TA \times TA$ 

followed by the structure morphism for the initial algebra.

If such initial algebras always exist, then  $F^{\dagger}$  extends to a functor whose action on  $f : A \rightarrow B$  is the F(A, -)-algebra homomorphism induced by the action:

$$F(A, F^{\dagger}B) \xrightarrow{F(f, \mathrm{id})} F(B, F^{\dagger}B) \xrightarrow{\alpha_B} F^{\dagger}B$$

Further, if  $\beta : F \langle id, G \rangle \Rightarrow G : \mathcal{C}^m \to \mathcal{C}^n$  is a natural transformation, then the unique algebra homomorphisms induce a natural transformation  $\beta^{\dagger} : F^{\dagger} \Rightarrow G$ .

**Theorem 8.5** If  $F : \mathcal{C}^m \times \mathcal{C}^n \to \mathcal{C}^n$  is a shapely type constructor then  $F^{\dagger}$  exists and is one, too. Further, if  $\beta : F \langle id, G \rangle \Rightarrow G : \mathcal{C}^m \to \mathcal{C}^n$  is a shapely transformation, then so is  $\beta^{\dagger}$ .

**Proof** F is determined by its projections onto C which are all shapely over  $\Pi L$ . By the Bekic Lemma, we can treat these individually, or, equivalently, assume that n = 1. Then for each object A in  $C^m$  the initial algebra  $F^{\dagger}A$  for F(A, -) is constructed as above.

That  $F^{\dagger}$  is shapely over  $\Delta \Pi L$  will be a consequence of the second part of the theorem applied to the composite transformation  $\beta$ 

$$F(A, \Delta \Pi LA) \xrightarrow{\delta} \Delta \Pi L(A, \Delta \Pi LA) \longrightarrow \Delta \Pi LA$$

where  $\delta$  makes F shapely over  $\Delta \Pi L$  and the second transformation is built from natural isomorphisms,  $\mu$ , @ and the transformation  $\langle L\Pi_i \rangle : L\Pi L \Rightarrow \Pi LL$  which is shapely by Proposition 4.5. Hence,  $\beta^{\dagger} : F^{\dagger} \Rightarrow \Delta \Pi L$  is shapely as required.

Some additional notation will clarify the proof of the second assertion, that  $\beta^{\dagger}$  is shapely. Define  $\Omega_{(-)} = F(-, 1)$  so that  $\Omega_A = F(A, 1)$  etc. Observe that the following square is a pullback

$$\begin{array}{c|c} \Omega_A \times L\Omega_A \times LGA & \xrightarrow{\Omega_f \times L\Omega_f \times LGf} & \Omega_B \times L\Omega_B \times LGB \\ & & & & & & \\ \theta_{GA} & & & & & \\ L\Omega_A \times LGA & \xrightarrow{L\Omega_f \times LGf} & L\Omega_B \times LGB \end{array}$$

so that  $\theta_G$  is cartesian, and in fact is shapely. Hence  $\chi_G$  is also shapely, by Theorem 4.6. For each morphism  $f: A \rightarrow B$  in  $\mathcal{C}^m$  we must show that the left-hand square of

$$\begin{array}{c} F^{\dagger}A \xrightarrow{F^{\dagger}f} F^{\dagger}B \xrightarrow{\phi^{B}} L\Omega_{B} \\ \beta^{\dagger}_{A} & \beta^{\dagger}_{B} & \downarrow \\ GA \xrightarrow{Gf} GB \xrightarrow{\langle nil, \eta \rangle} L\Omega_{B} \times LGB \end{array}$$

is a pullback. As the right-hand square is a pullback by definition, it suffices to observe that the outer square is one. But this can be re-drawn as:

The strength for  $F^{\dagger}$  is defined in Fig. 14 using the defining pullback for  $F^{\dagger}(A \times B)$  and the strength of  $L\Omega_A \times LGA$ . Its right face commutes because  $\chi_G$  is strong. Taking GA = 1 shows that the strength for  $F^{\dagger}$  does not depend on  $\beta$ . The diagram also shows that  $\beta^{\dagger}$  is strong, and so is shapely.

Note that the theorem asserts that if F is shapely over lists then so is  $F^{\dagger}$ . It does not establish the stronger conjecture, that if F is merely shapely then so is  $F^{\dagger}$ . This is because the proof of shapeliness, like that of existence for  $F^{\dagger}$ , relies on a parsing argument.

## 9 Conclusions

A semantic notion of shape has been presented, and used to prove that, under mild assumptions, the existence of lists is enough to establish the existence of all the other inductive types, such as trees. It also indicates how shape polymorphic operations, e.g. mapping, can be introduced for such types.

Of much broader significance is that the same semantic notion embraces many of the other fundamental data types, such as arrays, graphs and records, which are not inductive types, and hence outside the core of many languages.



Figure 14: The strength for  $F^{\dagger}A$ .

A type system, and programming language, based on shape should yield many further benefits, including shape polymorphism, the detection of shape errors, and optimisation of run-time code based on shape analysis of the inputs.

## Acknowledgements

I would like to thank the anonymous referees, and D. Clarke, J. Crossley, J. Edwards, D. Mahler and M. Sekanina for their constructive criticism.

# References

- [BC90] G.E. Blelloch and S. Chatterjee. VCODE: A data-parallel intermediate language. In Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation, pages 471-480, October 1990.
  [BS93] C.R. Banger and D.B. Skillicorn. A foundation for theories of arrays. Queen's University, Canada, 1993.
  [BW88] R. Bird and P. Wadler. Introduction to Functional Programming. International Series in Computer Science. Prentice Hall, 1988.
- [CF92] J.R.B. Cockett and T. Fukushima. About charity. Technical Report 92/480/18, University of Calgary, 1992.

- [CH88] T. Coquand and G. Huet. The calculus of constructions. Information and Computation, 73(2/3), 1988.
- [CLW93] A. Carboni, S. Lack, and R.F.C. Walters. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, 84:145–158, 1993.
- [Coc90] J.R.B. Cockett. List-aritmetic distributive categories: locoi. Journal of Pure and Applied Algebra, 66:1-29, 1990.
- [Col93] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. In D. Trystram, editor, *Proceedings of Parco 93*, Advances in Parallel Computing. Elsevier, 1993.
- [DER86] I.S. Duff, A.M. Erisman, and J.K. Reid. Direct Methods for Sparse Matrices. Clarendon Press Oxford, 1986.
- [FCO90] J.T. Feo, D.C. Cann, and R.R. Oldehoeft. A report on the sisal language project. Journal of Parallel and Distributed Computing, 10:349-366, 1990.
- [GLT89] J-Y. Girard, Y. Lafont, and P. Taylor. Proofs and Types. Tracts in Theoretical Computer Science. CUP, 1989.
- [Hag83] T. Hagino. A Categorical Programming Language. PhD thesis, University of Edinburgh, 1983.
- [HMM90] R. Harper, J. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In 17th POPL. ACM, 1990.
- [HPJW92] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the programming language haskell: a non-strict, purely functional language. SIGPLAN Notices, 1992.
- [Jay93a] C.B. Jay. Matrices, monads and the fast fourier transform. Technical Report UTS-SOCS-93.13, University of Technology, Sydney, 1993.
- [Jay93b] C.B. Jay. Tail recursion through universal invariants. Theoretical Computer Science, 115:151-189, 1993.
- [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.
- [JG89] M.A. Jenkins and J.I. Glasgow. A logical basis for nested array data structures. Computer Languages Journal, 14(1):35-51, 1989.
- [Jon90] G. Jones. Deriving the fast fourier transform algorithm by calculation. In Functional programming, Glasgow 1989, Springer Workshops in Computing. Springer Verlag, 1990.
- [Jon94] M. Jones. The implementation of the gofer functional programming system. Technical Report YALEU /DCS/RR-1030, Yale University, 1994.
- [Joy81] A. Joyal. Une théorie combinatoire des séries formelles. Advances in Mathematics, 42:1-82, 1981.

- [Ka94] V. Kumar and all. Introduction to Parallel Computing: Design and Analysis of Algorithms. The Benjamin/Cummings Publishing Company, Inc., 1994.
- [Koc72] A. Kock. Strong functors and monoidal monads. Archiv der Mathematik, 23, 1972.
- [Kun82] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37-46, January 1982.
- [Man92] E. Manes. Predicate Transformer Semantics, volume 33 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
- [McC94] W.F. McColl. BSP Programming. In Proceedings DIMACS Workshop on Specification of Parallel Algorithms, 1994.
- [Mey94] B. Meyer. Eiffel: the libraries. Prentice-Hall, 1994.
- [MT91] R. Milner and M. Tofte. Commentary on Standard ML. MIT Press, 1991.
- [Ski94] D.B. Skillicorn. Foundations of Parallel Programming. Number 6 in Cambridge Series in Parallel Computation. Cambridge University Press, 1994.
- [SP82] M. Smith and G. Plotkin. The category-theoretic solution of recursive domain equations. SIAM Journal of Computing, 11, 1982.

This article was processed using the  $IAT_{EX}$  macro package and P. Taylor's diagrams package.