Shapely Types and Shape Polymorphism

C. Barry Jay[1] and J.R.B. Cockett[2]

[1] School of Computing Sciences, University of Technology, Sydney, PO Box 123
Broadway, 2007, Australia

[2] Department of Computer Science, University of Calgary, Calgary, Alberta,
T2N 1N4, Canada

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. They include both arrays and the usual algebraic types (of trees, graphs, etc.), and are closed under the formation of initial algebras.

1 Introduction

Consider the operation map which applies a function to each element of a list. In existing functional languages, its type is

$$(\alpha \to \beta) \to \alpha\,\texttt{list} \to \beta\,\texttt{list}$$

where α and β may range over any types. This *data polymorphism* allows the data (α and β) 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

$$(A \to B) \to A\,\texttt{tree} \to B\,\texttt{tree}$$

and

$$(A \to B) \to A\,\texttt{matrix} \to B\,\texttt{matrix}$$

In each case map(f) applies f to the data (the leaves or entries), while leaving the shape fixed. Typically, both kinds of polymorphism co-exist, so that map can vary both its data and its shape.

Shape polymorphism applies to the usual algebraic types, (and others, such as matrices) but, being restricted to covariant constructions, does not address contravariant types of, say, functions.

The appropriate class of types are the *shapely types,* whose shape and data can be separated. With only one kind of data, this separation is represented by a pullback

$$\begin{tikzcd}
FA \arrow[r, "\texttt{data}"] \arrow[d, "\texttt{shape}"'] \arrow[dr, phantom, very near start, "\lrcorner"] & A\,\texttt{list} \arrow[d, "\texttt{length}"] \\
\texttt{Shapes} \arrow[r, "\texttt{arity}"'] & N
\end{tikzcd}$$

Values of a shapely type FA are uniquely determined by a list of A's and a shape of type **Shapes**, such that the arity of the shape equals the length of the list. Shape polymorphism arises when the operations on the data and shape are completely independent of each other.

It oftens happens, particularly in scientific computation and data-processing, that the shape can influence the data, but not conversely. If computation is restricted to such *shapely operations* then the shape information can be treated as if it is part of the type system, with all shape computations, including shape checking, performed before executing the list operations. The latter can then be optimised, or run in parallel, without sacrificing (and indeed strengthening) the benefits of typing.

These ideas can be understood within a semantics based on sets, or of bottomless c.p.o's, but for generality (and flexibility) are presented in a locos [Coc90], which provides a minimal setting for working with both lists and pullbacks. In particular, no higher types are assumed (as is shown feasible in the language Charity[CF92]) since they contribute nothing to the theory of shape. A calculus of shapely functors and natural transformations is introduced, with the shapely type constructors being those functors which have a shapely transformation to the list functor (or a multi-parameter analogue of it).

Most of this paper is devoted to the introduction of concepts. The main result of the paper is that the shapely type constructors are closed under the construction of initial algebras. That is, once we have lists then we have all the other "algebraic" types of trees, graphs, queues, etc. The additional presence of arrays, and other types defined by pullbacks, means that the shapely types lie between these algebraic types, and the class of initial algebras for arbitrary functors. Full details of proofs can be found in [JC94].

2 Locoses

The types and operations are modelled by the objects and arrows of a category $\mathcal{C}$. 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 finitely complete, distributive category (or 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 c.p.o.'s, or even topological spaces, any one of which will suffice to illustrate the ideas below.

Let us fix some notation. If $f : C \rightarrow A$ and $g : C \rightarrow B$ are morphisms then $\langle f, g \rangle : C \rightarrow A \times B$ is their pairing. The left and right projections from the product are $\pi_{A,B}$ and $\pi'_{A,B}$ (respectively) and the unique morphism to the terminal object is $!_A : A \rightarrow 1$. The symmetry for the product is denoted $c_{A,B} : A \times B \rightarrow B \times A$. Dually, the coproduct inclusions are given by $\iota_{A,B} : A \rightarrow A+B$ and $\iota'_{A,B} : B \rightarrow A+$

B. If $f : A \rightarrow C$ and $g : B \rightarrow C$ then their case analysis is given by $[f, g] : A + B \rightarrow C$. The functors $\Pi, \Sigma : \mathcal{C}^n \rightarrow \mathcal{C}$ denote chosen n-fold products and coproducts, respectively, and $\Delta : \mathcal{C} \rightarrow \mathcal{C}^n$ 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 $[\mathtt{id} \times \iota, \mathtt{id} \times \iota']$.

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

A *pullback* is a commuting square

$$\begin{tikzcd}
P \arrow[r, "q"] \arrow[d, "p"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "g"] \\
A \arrow[r, "f"'] & C
\end{tikzcd}$$

such that, for every pair of morphism $x : X \rightarrow A$ and $y : X \rightarrow B$ such that $f \circ x = g \circ y$ there is a unique morphism $z : X \rightarrow P$ such that $p \circ z = x$ and $q \circ z = y$. We denote z by $\langle x, y \rangle$ (the usual pairing is a special case).

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

$$\mathtt{nil} : 1 \rightarrow LA \tag{1}$$

$$\mathtt{cons} : A \times LA \rightarrow LA \tag{2}$$

$$\mathtt{foldr}(x, h) : LA \rightarrow C \tag{3}$$

where $x : B \rightarrow C$ and $h : A \times C \rightarrow C$ are morphisms. $\mathtt{nil}$ and $\mathtt{cons}$ are the usual constructors, while $\mathtt{foldr}(x, h)$ is the unique morphism making the following diagram commute

$$\begin{tikzcd}
B \arrow[r, "\langle \mathtt{nil}, \mathtt{id} \rangle"] \arrow[dr, "x"'] & LA \times B \arrow[d, "\mathtt{foldr}(x, h)"] & A \times LA \times B \arrow[l, "\mathtt{cons} \times \mathtt{id}"'] \arrow[d, "\mathtt{id} \times \mathtt{foldr}(x, h)"] \\
 & C & A \times C \arrow[l, "h"]
\end{tikzcd}$$

It follows that $[\mathtt{nil}, \mathtt{cons}] : 1 + (A \times LA) \rightarrow LA$ 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 $\mathtt{map}(f)$ for $f : A \rightarrow B$
$\eta : A \rightarrow LA$	makes singleton lists
$@ : LA \times LA \rightarrow LA$	is append
$\mathtt{snoc} : LA \times A \rightarrow LA$	is $\mathtt{cons}$ on the tail of the list
$\mu : L^2 A \rightarrow LA$	flattens a list of lists
$g^* : LA \rightarrow LB$	is $\mu \circ Lg$ for $g : A \rightarrow LB$.

Many elementary results about lists in locoses can be found in [Jay93b].

$L1$ is a natural numbers object N with zero 0 and successor S given by $\mathtt{nil}$ and $\mathtt{cons}$ respectively. Then $\eta = \mathtt{one}$ and $@ = +$ is addition and $\mu : LN \rightarrow N$ is summation. Let $\mathtt{Eq}$ be the equality on N. The *length* of a list object LA is $L! = \# : LA \rightarrow N$.

Define $\mathtt{shunt} : LA \times LA \rightarrow LA \times LA$ to be

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

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

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

is given by $\mathtt{foldr}(\langle \mathtt{nil}, \mathtt{id} \rangle, \mathtt{shunt})$. 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

$$\mathtt{take} = \pi \circ \mathtt{split} : N \times LA \rightarrow LA \qquad (4)$$

$$\mathtt{drop} = \pi' \circ \mathtt{split} : N \times LA \rightarrow LA \;. \qquad (5)$$

Lemma 1. $@ \circ \mathtt{split} = \pi'$. *Hence, we have a pullback*

$$\begin{tikzcd}
LC \times LC \arrow[r, "\langle \# \circ \pi{,} @ \rangle"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & N \times LC \arrow[d, "\mathtt{Eq} \circ \langle \pi{,} \# \circ \mathtt{take} \rangle"] \\
1 \arrow[r, "\mathtt{true}"'] & \mathtt{bool}
\end{tikzcd}$$

Proof. Both sides of the equation equal $\mathtt{foldr}(\mathtt{id}, \mathtt{id})$.

Given $x : X \rightarrow N \times LC$ for which $\mathtt{Eq} \circ \langle \pi, \mathtt{take} \rangle \circ x = \mathtt{true}$ then the induced morphism into the pullback is $\mathtt{split} \circ x$. □

The powers $\mathcal{C}^n$ of $\mathcal{C}$ are also locoses, and they have a right $\mathcal{C}$-action. That is, a functor $\mathcal{C}^n \times \mathcal{C} \rightarrow \mathcal{C}^n$ which maps $A = (A_1, A_2, \ldots, A_n)$ and B to

$$A \times B = (A_1 \times B, A_2 \times B, \ldots, A_n \times B)$$

and has the obvious action on morphisms.

3 Shapely Functors and Transformations

The essential ideas of this section can be found in [Jay93a].

A *strength* for a functor $F : \mathcal{C}^m \to \mathcal{C}$ is a natural transformation

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

which satisfies the usual associativity and unicity axioms. More generally, a strength for $F : \mathcal{C}^m \to \mathcal{C}^n$ is given by a strength for each of its projections onto $\mathcal{C}$. See [CS92] for an account of the connections between strength and fibrations.

(F, τ) is a *shapely functor* if F is *stable* (preserves all pullbacks) and τ is a strength for it. Then $F1$ is the *object of F-shapes* and $\# = F! : FA \to F1$ is the *shape* of FA (generalising the length of a list). As F is stable, the following diagram is a pullback

$$\begin{tikzcd}
F(A \times B) \arrow[r, "F\pi'"] \arrow[d, "F\pi"'] \arrow[dr, phantom, "\lrcorner", very near start] & FB \arrow[d, "\#"] \\
FA \arrow[r, "\#"'] & F1
\end{tikzcd}$$

If $FA \times_{\#} FB$ is a canonical choice of pullback (representing pairs that have the same shape) then there is an isomorphism

$$\mathtt{zip} : FA \times_{\#} FB \to F(A \times B)$$

which is inverse to $\langle F\pi, F\pi' \rangle$ and generalises the usual `zip` of lists.

The shapely functors include the product, coproduct and list functors, and are closed under composition and pairing.

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

$$\begin{tikzcd}
FA \arrow[r, "\alpha_A"] \arrow[d, "Ff"'] \arrow[dr, phantom, "\lrcorner", very near start] & GA \arrow[d, "Gf"] \\
FB \arrow[r, "\alpha_B"'] & GB
\end{tikzcd}$$

A *strong* natural transformation $(F, \tau_1) \Rightarrow (G, \tau_2)$ between strong functors is a natural transformation $\alpha : F \Rightarrow G$ that commutes with the strengths, i.e.

$$\begin{tikzcd}
FA \times B \arrow[r, "\alpha_A \times \mathtt{id}"] \arrow[d, "\tau_1"'] & GA \times B \arrow[d, "\tau_2"] \\
F(A \times B) \arrow[r, "\alpha_{A \times B}"'] & G(A \times B)
\end{tikzcd}$$

If, further, F and G are shapely and α is cartesian, then α is a *shapely transformation* and F is *shapely over* G by α.

A consequence of being an extensive category is that the coproduct inclusions are shapely; cartesian-ness is by definition, and the strength is given by the distributive law.

Projections from the product $\times : \mathcal{C}^2 \rightarrow \mathcal{C}$, though strong, are never shapely. (If it were so then all the transformations of interest would be shapely.) Instead, given $f : A \rightarrow C$ and $g : B \rightarrow D$ we have the pullback

$$\begin{tikzcd}
A \times D \arrow[r, "\pi"] \arrow[d, "f \times \mathtt{id}"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "f"] \\
C \times D \arrow[r, "\pi"'] & C
\end{tikzcd}$$

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

The shapely transformations are closed under vertical and horizontal composition, so that for each locos $\mathcal{C}$ we have a 2-category of shapely functors and natural transformations. They are also closed under pairing and case analysis. Here are two more general results.

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

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

is a cartesian in both A and B.

Proof. Consider a commuting square

$$\begin{tikzcd}
X \arrow[r, "\langle x, y \rangle"] \arrow[d, "z"'] & GA \times HB \arrow[d, "Gg \times Hh"] \\
F(A', B') \arrow[r, "\langle \alpha, \beta \rangle"'] & GA' \times HB'
\end{tikzcd}$$

Then x and z induce a unique morphism $x' : X \rightarrow F(A, B')$ by the cartesian-ness of α. Similarly y and z induce a morphism $y' : X \rightarrow F(A', B)$. As F is shapely, x' and y' induce the desired morphism $X \rightarrow F(A, B)$. □

Theorem 3. *If $\alpha : F \Rightarrow G$ and $\beta : H \times G \Rightarrow G$ are shapely transformations, then*

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

is a shapely transformation.

Proof. The result generalises [Jay93a, Theorem 2.6] without changing the proof. □

Corollary 4. $@ = \mathtt{foldr}(\mathtt{nil}, \mathtt{cons})$ *and* $\mu = \mathtt{foldr}(\mathtt{nil}, @)$ *are shapely.* □

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

Lemma 5. *If $\alpha : F \Rightarrow G$ is cartesian and (G, τ_2) is shapely then there is a unique strength τ_1 for F such that (F, τ_1) and α are both shapely.*

Proof. If F has a strength τ_1 that makes α strong then the following diagram must commute.

$$\begin{tikzcd}
FA\times B \arrow[r, "\alpha\times\mathtt{id}"] \arrow[dr, "\tau_1"] \arrow[ddr, "\pi"'] & GA\times B \arrow[dr, "\tau_2"] & \\
& F(A\times B) \arrow[r, "\alpha"'] \arrow[d, "F\pi"] \arrow[dr, phantom, "\lrcorner", very near start] & G(A\times B) \arrow[d, "G\pi"] \\
& FA \arrow[r, "\alpha"'] & GA
\end{tikzcd}$$

Since the square is a pullback, this determines τ_1 uniquely. Conversely, this pullback can be used to define τ_1. Its naturality, associativity and unicity are all inherited from that of τ_2. □

4 Shapely Types

A functor $F : \mathcal{C} \rightarrow \mathcal{C}$ is a shapely type constructor if it is shapely over L. More generally, a functor $F : \mathcal{C}^m \rightarrow \mathcal{C}^n$ is a *shapely type constructor* if it is shapely over $\Delta \Pi L^m$. Equivalently, each of the projections of F onto $\mathcal{C}$ must be shapely over ΠL^m. If δ is the relevant shapely transformation then we have the pullback

$$\begin{tikzcd}
F(A_i) \arrow[r, "\delta"] \arrow[d, "\#"'] \arrow[dr, phantom, "\lrcorner", very near start] & \Pi(LA_i) \arrow[d, "\#"] \\
F1 \arrow[r, "\delta_1"'] & N^n
\end{tikzcd}$$

$F(A_i)$ is a (tuple of) shapely types, and δ_1 is also known as the *arity* of $F1$.

The shapes can be thought of as having fixed numbers of holes or entries of each type, which are filled in by the data.

Of course, lists are shapely by the identity transformation. Also, $\mu : L^2 \Rightarrow L$ makes L^2 a shapely type constructor, with shapes given by lists of numbers.

The shape of a matrix is given by its dimensions, which are of type N^2. The defining pullback is

$$\begin{tikzcd}
MA \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & LA \arrow[d, "\#"] \\
N^2 \arrow[r, "*"'] & N
\end{tikzcd}$$

where MA is the type of matrices with entries from A. (They can also be defined as vectors of vectors [Jay93a]. Matrix multiplication, and general operations of linear algebra can all be defined in this way.

Binary trees with leaves labelled by A's have shapes given by (unlabelled) trees, whose arity is their number of leaves, and data given by their list of leaves.

$$\begin{tikzcd}
TA \arrow[r, "\mathtt{leaves}"] \arrow[d, "\#"'] \arrow[dr, phantom, "\lrcorner", very near start] & LA \arrow[d, "\#"] \\
T1 \arrow[r, "\mathtt{leaves}"'] & N
\end{tikzcd}$$

The pullback construction does not determine the order of elements in the list of leaves, which could be listed from left to right, right to left, or in some more arcane fashion. Let us stick with left-to-right order, unless specifically changed.

map on trees is given by the induced morphism into the pullback

$$\begin{tikzcd}
TA \arrow[rr] \arrow[dd] \arrow[dr, dotted, "Tf"] & & LA \arrow[dd] \arrow[dr, "Lf"] & \\
& TB \arrow[rr] \arrow[dd] \arrow[dr, phantom, "\lrcorner", very near start] & & LB \arrow[dd, "\#"] \\
T1 \arrow[rr] \arrow[dr, "\mathtt{id}"'] & & N \arrow[dr, "\mathtt{id}"'] & \\
& T1 \arrow[rr, "\mathtt{leaves}"'] & & N
\end{tikzcd}$$

Note that the shape has remained fixed while the data changes. Of course, this construction generalises to define map for any shapely type constructor.

Other operations change the shape and leave the data fixed. For example, the balancing of a binary tree is given by an operation $\mathtt{bal}_A : TA \rightarrow TA$ which is independent of the labels

$$\begin{tikzcd}
TA \arrow[rr] \arrow[dd] \arrow[dr, dotted, "\mathtt{bal}_A"] & & LA \arrow[dd] \arrow[dr, "\mathtt{id}"] & \\
& TA \arrow[rr] \arrow[dd] \arrow[ddrr, phantom, "\lrcorner", very near start] & & LA \arrow[dd, "\#"] \\
T1 \arrow[rr] \arrow[dr, "\mathtt{bal}_1"'] & & N \arrow[dr, "\mathtt{id}"'] & \\
& T1 \arrow[rr, "\mathtt{leaves}"'] & & N
\end{tikzcd}$$

Finally, both data and shape may change together, as occurs with the node operation, which creates a tree from two subtrees.

$$\begin{tikzcd}
(TA)^2 \arrow[rr] \arrow[dd] \arrow[dr, dotted, "\mathtt{node}_A"] & & (LA)^2 \arrow[dd] \arrow[dr, "@"] & \\
& TA \arrow[rr] \arrow[dd] \arrow[ddrr, phantom, "\lrcorner", very near start] & & LA \arrow[dd, "\#"] \\
(T1)^2 \arrow[rr] \arrow[dr, "\mathtt{node}_1"'] & & N^2 \arrow[dr, "+"'] & \\
& T1 \arrow[rr, "\mathtt{leaves}"'] & & N
\end{tikzcd}$$

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` represents the leaves from right to left then the order of the arguments must be swapped to define @. In some sense, the choice of @ here fixes the representation of the leaves.

Trees whose nodes are also labelled ar shapely, too. The defining pullback is given by

$$\begin{tikzcd}
T(A,B) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & LA \times LB \arrow[d, "\# \times \#"] \\
T1 \arrow[r, "\langle \mathtt{leaves}, \mathtt{nodes} \rangle"'] & N \times N
\end{tikzcd}$$

Most of the usual first-order data types of functional languages can be constructed in this way, and arrays are available within the same framework, too. Note that, since shapely type constructors are always covariant functors, contravariant constructions, such as function types, cannot be shapely.

4.1 An Alternative Approach

In defining shapely type constructors, the data is given by a tuple of lists, one for each kind of data whose type is $LA_1 \times LA_2 \times \ldots \times LA_n$. What happens if these lists are interleaved to resemble input strings, of type $L(A_1 + A_2 + \ldots + A_n)$? In other words, one could consider the functors which are shapely over $L\Sigma$ instead of ΠL. In fact, the resulting class of functors is unchanged, as shown by the following

Proposition 6. *ΠL^m and $L\Sigma$ are each shapely over the other.*

Proof. Clearly, there is shapely natural transformation $\Pi L^m \Rightarrow L\Sigma$ obtained by concatenation. Conversely, define the natural transformation $\mathtt{check}_{A,B}$ by

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

It is shapely in A whence $\kappa_1 = \mathtt{foldr}(\mathtt{nil}, \mathtt{check})$ is, too. This can be generalised to define $\kappa_i : L(A_1 + A_2 + \ldots + A_n) \rightarrow LA_i$ which strips from a list all entries which are not from A_i. It is shapely in A_i. The obvious n-fold generalisation of Proposition 2 shows that

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

is a shapely transformation. □

5 Shape Polymorphism

Shape polymorphism arises when the operations on the shape and on the data are independent of each other. Currently, such operations must redefined for each new type, increasing the bulk of the code and reducing clarity. Aside from `map`, the shape polymorphic operations include

$$\mathtt{zip} : FA \times_{\#} FB \rightarrow F(A \times B) \quad (6)$$

$$\tau : FA \times B \rightarrow F(A \times B) \quad (7)$$

$$\tau' : A \times FB \rightarrow F(A \times B) \quad (8)$$

$$\tau'' = F\tau' \circ \tau : FA \times FB \rightarrow F^2(A \times B) \quad (9)$$

$$\mathtt{copies} : F1 \times A \rightarrow FA \quad (10)$$

$$\mathtt{square} = \tau \circ \langle \#, \mathrm{id} \rangle : FA \rightarrow F^2 A \ . \quad (11)$$

τ' is dual to τ and the operation `copies` instantiates all entries of the shape to the given value, while `square` replaces each entry with a copy of the whole.

Closely related to `map` 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 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. Their use in defining the discrete Fourier transform *op. cit.* shows it to be shape polymorphic.

6 Shapely Operations

In general computation, the shape of the result is influenced by that of the data. Examples include filtering of a list, or graph reduction. There is, however, a large body of computations where the shape of the result depends only on that of the input, without reference to the data. As well as the shape polymorphic operations, these include many operations where the shape affects the data, but not conversely. Examples include averaging of entries, pointwise operators, and many algorithms, such as the DFT.

For such *shapely operations* it is profitable to separate the internal representations of shape and data. Then shape processing can be treated as part of compilation, in which shape errors are detected (e.g. attempting to `zip` two different shapes), the shape of the result is computed, and any shape information required by the data is supplied. Data can then be stored and processed in arrays. In this way, the clarity of type (and shape!)-checking is combined with the efficiency of array-processing.

The main points are illustrated by the decomposition of a tree into either a leaf or a pair of sub-trees.

$$\begin{tikzcd}
TA \arrow[r, "\simeq"] \arrow[d] & A + (TA)^2 \arrow[r, "\mathtt{id} + \delta^2"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & A + (LA)^2 \arrow[r, "{[\eta, @]}"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & LA \arrow[d] \\
T1 \arrow[r, "\simeq"'] & 1 + (T1)^2 \arrow[r, "\mathtt{id} + \delta^2"'] & 1 + N^2 \arrow[r, "{[\mathsf{one}, +]}"'] & N
\end{tikzcd}$$

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 subtree is required. Shape processing would add the computed value of n to the environment prior to the array-processing.

7 Initial Algebras

7.1 Endofunctors

Let F be shapely over L. The initial algebra F_0 will be constructed as an object of well-formed expressions. The alphabet is given by $\Omega = F1$. The well-formed expressions are described by a pullback

$$\begin{tikzcd}
F_0 \arrow[r, "\phi"] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & L\Omega \arrow[d, "\chi_1"] \\
1 \arrow[r, "{\langle \mathtt{nil}, \mathtt{one} \rangle}"'] & L\Omega \times N
\end{tikzcd}$$

where χ_1 attempts to recognise the well-formed expressions. More precisely, its second component counts the number of well-formed expressions created, while the first component yields that part of the input string (if any) which could not be recognised.

Observe that $! : F1 \to 1$ makes 1 an F-algebra. The recogniser χ_1 is a special case of a more general "parser"

$$\chi_C : L\Omega \to L\Omega \times LC$$

to be defined for any F-algebra $\gamma : FC \to C$. The terminology is motivated by the case when C represents the parse trees, as given by F_0 below.

$\chi_C = \mathtt{foldr}(\langle \mathtt{nil}, \mathtt{nil}\rangle, \theta_C)$ where $\theta_C : \Omega \times L\Omega \times LC \to L\Omega \times LC$ performs one step of the parse, as will now be described.

First, the middle component of the source is $\mathtt{nil}$ unless the parse has already failed. That is, we can re-express the source (using the appropriate isomorphism) as $(\Omega \times LC) + (\Omega \times \Omega \times L\Omega \times LC)$ and then $\theta_C = [\zeta_C, \mathtt{cons} \circ (\mathtt{id} \times \mathtt{cons}) \times \mathtt{id}]$ where ζ_C remains to be described, by cases.

Lemma 7. *The test* $\mathtt{Eq} \circ \langle \pi, \mathtt{take}\rangle \circ (\delta \times \mathtt{id}) : \Omega \times LC \to \mathtt{bool}$ *recognises the sub-object*

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

Proof. In brief, the test picks out those pairs where the arity of the Ω is no greater than the length of the list. In that case, we have the resources to construct something of type FC with a list of C's left over. □

Let $\iota' : QC \to \Omega \times LC$ be the pullback of the above test along $\mathtt{false}$. Then ζ_C is given by decomposing $\Omega \times LC$ into the specified coproduct followed by

$$[\langle \mathtt{nil}, \mathtt{cons} \circ (\gamma \times \mathtt{id})\rangle, (\eta \times \mathtt{id}) \circ \iota'] : (FC \times LC) + QC \to L\Omega \times LC \ .$$

Lemma 8. *If* $h : (C, \gamma) \to (C', \gamma')$ *is an* F*-algebra homomorphism then*

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

Proof. Clearly Q is a functor and ι' is a natural transformation. Hence ζ, θ and χ are natural with respect to F-algebra homomorphisms. □

Hence, the construction of F_0 can be given in stages by

$$\begin{tikzcd}
F_0 \arrow[r, "\phi"] \arrow[d, dotted, "h_C"'] \arrow[dr, phantom, very near start, "\lrcorner"] & L\Omega \arrow[d, "\chi_C"] \\
C \arrow[r, "\langle \mathtt{nil}, \eta_C\rangle"] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & L\Omega \times LC \arrow[d, "\mathtt{id} \times \#"] \\
1 \arrow[r, "\langle \mathtt{nil}, \mathtt{one}\rangle"'] & L\Omega \times N
\end{tikzcd}$$

Now we will show that F_0 is an initial F-algebra with h_C the unique algebra homomorphism to C.

Lemma 9. $\chi_C \circ \phi^* = \langle \mathtt{nil}, Lh_C \rangle : LF_0 \rightarrow L\Omega \times LC$

Proof. It suffices to show that both sides of the equation are foldright of $\langle \mathtt{nil}, \mathtt{nil} \rangle$ and $\mathtt{foldr}(\mathtt{id}, \theta_C) \circ (\phi \times \mathtt{id})$. The **cons** case for the right-hand-side is not trivial. See [JC94] for details.

□

Consider the following cube.

$$\begin{tikzcd}
FF_0 \arrow[rr, "\langle \#, \delta \rangle"] \arrow[ddd, "Fh_C"'] \arrow[drr, dotted, "\gamma_0"] & & \Omega \times LF_0 \arrow[r, "\mathtt{id} \times \phi^*"] & \Omega \times L\Omega \arrow[dr, "\mathtt{cons}"] \arrow[dd, "\mathtt{id} \times \chi_C"] & \\
& & F_0 \arrow[rr, "\phi"] \arrow[dd, "h_C"'] \arrow[drr, phantom, very near start, "\lrcorner"] & & L\Omega \arrow[dd, "\chi_C"] \\
& & & \Omega \times L\Omega \times LC \arrow[dr, "\theta_C"'] & \\
FC \arrow[uurrr, phantom] \arrow[urrr, "\langle \#, \mathtt{nil}, \delta \rangle"] \arrow[rr, "\gamma"'] & & C \arrow[rr, "\langle \mathtt{nil}, \eta \rangle"'] & & L\Omega \times LC
\end{tikzcd}$$

Lemma 9 implies the commutativity of its rear face. The right and bottom faces commute by the definitions of χ and θ. Hence, there is an induced F-action γ_0 that makes h_C a homorphism. (Of course, the definition of γ_0 and its action is not dependent on the particular choice of C, since we can always work over the algebra $C = 1$.)

It remains to prove its uniqueness. Let $h : F_0 \rightarrow C$ be any F-algebra homomorphism. Then the pullback defining h_C factorises it as $h \circ h_0$ as in Fig. 1. (We abuse notation by denoting h_{F_0} by h_0 and χ_{F_0} by χ_0 etc.) Hence it suffices to prove that $h_0 = \mathtt{id}$.

The following lemma shows that parsing into F_0 is reversible.

Lemma 10. $@ \circ (\mathtt{id} \times \phi^*) \circ \chi_0 = \mathtt{id}_{L\Omega}$ □

Hence

$$\begin{aligned}
\phi \circ h_0 &= @ \circ (\mathtt{id} \times \phi^*) \circ \langle \mathtt{nil}, \eta \rangle \circ h_0 && (12) \\
&= @ \circ (\mathtt{id} \times \phi^*) \circ \chi_0 \circ \phi && (13) \\
&= \phi && (14)
\end{aligned}$$

and so $h_0 = \mathtt{id}$ since ϕ is a monomorphism.

$$\begin{tikzcd}
F_0 \arrow[r, "\phi"] \arrow[d, "h_0"'] \arrow[dr, phantom, "\lrcorner", very near start] & L\Omega \arrow[d, "\chi_0"] \\
F_0 \arrow[r, "\langle\mathtt{nil},\eta\rangle"] \arrow[d, "h"'] \arrow[dr, phantom, "\lrcorner", very near start] & L\Omega\times LF_0 \arrow[d, "\mathrm{id}\times Lh"] \\
C \arrow[r, "\langle\mathtt{nil},\eta\rangle"'] & L\Omega\times LC
\end{tikzcd}$$

Fig. 1. Factorisation of h_C

7.2 The General Case

A functor $F : \mathcal{C}^m\times\mathcal{C}^n\rightarrow\mathcal{C}^n$ 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 $\alpha_A : F(A,F^\dagger A)\rightarrow F^\dagger A$ for the functor $F(A,-)$. If such 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

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

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

Theorem 11. *If $\phi : F\Rightarrow\Delta\Pi L^{m+n} : \mathcal{C}^m\times\mathcal{C}^n\rightarrow\mathcal{C}^n$ is a shapely type constructor then $F^\dagger$ exists and is one, too. Further, if $\beta : F\langle\mathtt{id},G\rangle\Rightarrow G : \mathcal{C}^m\rightarrow\mathcal{C}^n$ is a shapely transformation, then so is $\beta^\dagger$.*

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

The cartesian-ness of $\beta^\dagger$ follows from that of χ_G which, by Theorem 3, reduces to that of θ_G. Examination of the cases reduces this to the cartesian-ness of $\mathtt{cons}, \eta$ and β.

The strength for $F^\dagger$ is defined using the defining pullback for $F^\dagger(A\times B)$ and the strength of $LF(A,1)\times LGA$. It follows that $\beta^\dagger$ is shapely.

Now, taking β to be

$$F(A,\Pi L^m A) \xrightarrow{\phi} \Pi L^m A\times\Pi L^m A \cong \Pi(LA\times LA) \xrightarrow{\Pi @} \Pi L^m A$$

(where $\cong$ permutes the arguments) induces a shapely transformation $F^\dagger\Rightarrow\Pi L^m$ which shows that $F^\dagger$ is a shapely type constructor. □

8 Related and Further Work

Banger and Skillicorn [BS93] give a categorical semantics for arrays, which are represented by their dimensions and some unbounded lists. They do not use pullbacks to represent their types, nor is there a general theory of shape.

Those shapely types, such as matrices, which are not part of the usual functional programming approach, can be represented using dependent types, but their type checking must be performed dynamically, whereas most shape computation should be performed statically. The exact relationship to dependent types, and the internal logic of locoses, require further exploration.

Further work will consider how to incorporate shape ideas into programming language design, and compiler construction, using classes in a functional or object-oriented setting.

References

[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**. University of Calgary preprint, 1992.

[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.

[CS92] J. R. B. Cockett and D. Spencer. Strong categorical datatypes. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. American Mathematics Society, Montreal, 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.

[JC94] C.B. Jay and J.R.B. Cockett. Shapely types and shape polymorphism: Extended version. Technical Report UTS-SOCS-94-??, University of Technology, Sydney, 1994.

[Jon90] G. Jones. Deriving the fast fourier transform algorithm by calculation. In *Functional programming, Glasgow 1989*, Springer Workshops in Computing. Springer Verlag, 1990.

[Man92] E. Manes. *Predicate Transformer Semantics*, volume 33 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[SP82] M. Smith and G. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11, 1982.