# Monads, Shapely Functors and Traversals

E. Moggi [a], G. Bellè [a] and C.B. Jay [b]

[a] *DISI - Univ. di Genova, via Dodecaneso 35, 16146 Genova, Italy* [1]

[b] *SOCS - Univ. of Tech. Sydney, P.O. Box 123 Broadway, 2007, Australia*

**Abstract**

This paper demonstrates the potential for combining the polytypic and monadic programming styles, by introducing a new kind of combinator, called a *traversal*. The natural setting for defining traversals is the class of shapely data types. This result reinforces the view that shapely data types form a natural domain for polytypism: they include most of the data types of interest, while to exceed them would sacrifice a very smooth interaction between polytypic and monadic programming.

*Keywords:* functional/monadic/polytypic programming, shape theory.

## 1 Introduction

Monadic programming has proved itself extremely useful as a means of encapsulating state and other computational effects in a functional programming setting (see e.g. [12,14]). Recently, interactions between monads and data structures have been studied as a further way for structuring programs. Initially focusing on lists, the studies have been extended to the class of regular datatypes (see e.g. [4,11,1]), with the aim to embody another kind of polymorphism into programs, that is, having combinators parameterized with respect to a class of datatypes. Thus generic properties of many of the usual combinators of the Bird-Meertens formalism, such as mapping, folding and zipping, can be extended by programming with monads.

The novelty of this work is our categorical characterization of the *traversal* constructor, which, among other things, leads us to having such a combinator defined uniformly for a large class of data types, namely those corresponding to functors shapely over lists.

[1] Research partially supported by MURST progetto cofinanziato “Tecniche formali per la specifica,. . . di sistemi software”, ESPRIT WG APPSEM.

**Background and related work.** Since their expressiveness and ability in structuring programs, the *map* and *fold* combinators have been the ideal candidates for studying the interactions between monads and data structures. Meijer and Jeuring [11] propose *monadic folds* as a useful pattern for structuring programs, and gives several examples of their use. Fokkinga [4] gives a definition of monadic fold for regular datatypes via an adjunction between the category of algebras and another category of algebras built upon the Kleisli category. This formalization requires an assumption on the monad, that is not valid for several monads (e.g. the state monad). The type of a monadic fold for lists is

$$mfold\colon (X \to Y \to TY) \to TY \to LX \to TY \quad \text{or equivalently}$$

$$mfold\colon ((1 + X \times Y) \to TY) \to LX \to TY$$

where $T$ can be any *monad* and $L$ is the list datatype. Another form of interaction between the list data type $L$ and a monad $T$ is given by *monadic map* (definable in terms of monadic fold, see [11])

$$mmap\colon (X \to TY) \to LX \to T(LY)$$

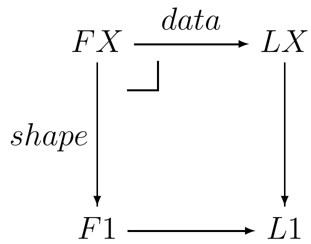
In contrast to the usual *map*, for a monadic map the order in which a list is traversed matters. In fact, every strategy for traversing a list induces a different monadic map. A simple application of monadic map is for labeling the elements of a list $(x_i|i\colon n)$ with their position, to produce the list $(\langle i, x_i\rangle|i\colon n)$. The idea is to use the state monad $TX = N \to (X \times N)$, whose state is a counter, and apply monadic map to the function $f\colon X \to T(N \times X)$, where the effect of $f(x)$ is to increment the counter and then return the pair consisting of the value of the counter and $x$. There is also another combinator that we call *traversal*

$$traverse\colon L(TX) \to T(LX)$$

obtained by supplying the identity function to the monadic map. Although monadic map and monadic folds are more useful in programming, traversals are more convenient for theoretical studies (e.g. for investigating naturality properties). It is clear that a traversal is a mechanism for commuting of functors, and such mechanisms have been studied elsewhere:

– Beck distributive laws $S(TX) \to T(SX)$ between monads $S$ and $T$ (see [2]) endow the composite functor $TS$ with a canonical monad structure.
– Arbib & Manes have considered distributive laws $F(TX) \to T(FX)$ between a functor $F$ and a monad $T$ (see [13]), and shown that they are in bijective correspondence with *extensions* of $F$ to the Kleisli category for $T$.
– Hoogendijk & Backhouse (see [6,5]) have investigated (generalized) zips $F(GX) \rightsquigarrow G(FX)$ between *relators* $F$ and $G$ in a relational setting.

**Functors shapely over lists.** Functors shapely over lists [7] correspond to those datatype constructors that can be split into the shape and data. Intuitively a shape can be thought of as a structure with a finite number of holes into which data elements, represented by a list, can be inserted. Formally, the characterization of a shapely type constructor $F$ uses pullback diagrams such as




Regular data types are shapely. Of course, not all type constructors are shapely; function types, for example, are not shapely, and neither are types of sets, since the cardinality of a set depends on knowing which elements are equal, i.e. depends on the data.

**Results.** From a programming language viewpoint, the main result of this paper is that traversals can be defined *uniformly* for a large class of data types, namely those corresponding to functors *shapely over lists*. This result is summarized by the existence of a polytypic combinator

$$\begin{aligned} \mathit{traverse} : & \forall m{:}\,\mathbf{N}.\forall F{:}\,\mathbf{ShFunctor}(m).\forall T{:}\,\mathbf{Monad}.\forall X_{i:m}{:}\,\mathbf{Type}. \\ & F(TX_i)_{i:m} \to T(F(X_i)_{i:m}) \end{aligned}$$

where $m$ is an arity, $F$ is a functor of arity $m$ shapely over lists, $T$ is a monad, and the $X_i$ are types. Furthermore, one can recover the more interesting *polytypic* monadic map and monadic fold from (the polytypic versions of *map* and $fold$ and) a polytypic traversal. This suggests the introduction (in Haskell) of constructor classes for functors shapely over lists. The power of traversals is exemplified by implementing a generic alpha-conversion function for an extensible type of lambda-terms. However, the mathematical contents of the paper is not adequately summarized by the above result. In fact, we address the following issues also:

– What makes a natural transformation $F(GX) \to G(FX)$ a traversal? We identify a key shape-preservation property: the *shape* of an $F$-data structure is not changed by a traversal. The simplest way of formalizing this property it to say that for each $F$-shape $s{:}\,F1$ one has a map $F_s(GX) \to G(F_sX)$, where $F_s(X)$ is the set of $F$-data structures with shape $s$ and data in $X$. Given such a family of maps one recovers a map $F(GX) \to G(FX)$.

- What properties should a polytypic traversal $F(TX) \to T(FX)$ have? We identify several higher-order naturality properties. They exploit the fact that $F$ is a functor shapely over lists and $T$ is a strong monad.
- Can we extend the polytypic traversal beyond functors shapely over lists and strong monads? Strong monads can be replaced by *monoidal functors*, which are more general. It seems unlikely that one can go beyond functors shapely over lists (but we have only a conjecture).
- What distinguishes traversals from zips and distributive laws a la Arbib and Manes? We formalize in a functional setting a shape-preservation property of zips, which is derivable from the definition of zip given in [6]: a zip of $F$ by $G$ is given by a family of natural isos $F_s(G_t X) \tilde{\to} G_t(F_s X)$, where $s\colon F1$ and $t\colon G1$. Given such a family one recovers a *span* (often a relation) $F(GX) \rightsquigarrow G(FX)$, which is defined on arguments where all $G$-shapes within the $F$-shape are the same. This common value is the outer shape of the result. By contrast, a traversal only considers the shape of $F$.

**Contents.** The structure of the paper is as follows. Section 2 reviews the categorical concepts of functor, monad and functor shapely over lists in the category **Set** of sets. Section 3 explores the implications of a polytypic traversal in a programming language. Section 4 provides a categorical semantics for traversals and Section 5 for zips in the simplified setting of **Set**. Section 6 outlines the definition of traversal and zip in a locos.

## 2 Preliminaries

This section reviews functors, monads and functors shapely over lists. For simplicity we will work in the category **Set** of (small) sets and functions. However, definitions (suitably adapted) and results can be extended to the more general setting of a locos (see [3]).

**Notation.** A sequence $X_0, X_1, \ldots, X_{m-1}$ of $m$ types, may be written as $X_{i:m}$ or even $\overline{X}$ when $m$ is either clear from the context, or irrelevant. Similar notation will be used for other sequences below, of functors, terms, etc.

Types for combinators of the form $X_0 \to X_1 \to \ldots \to X_{m-1} \to Y$ may be written as sequences $X_0, X_1, \ldots, X_{m-1} \to Y$ or $X_{i:m} \to Y$.

We write **Functor**$(m)$ for the (large) category $\mathbf{Set}^m \to \mathbf{Set}$ of $m$-ary functors $F\colon \mathbf{Set}^m \to \mathbf{Set}$ and natural transformations. Functors support the polytypic combinator of mapping

$$\begin{aligned} map : \forall m\colon \mathbf{N}.\forall F\colon \mathbf{Functor}(m).\forall X_{i:m}, Y_{i:m}\colon \mathbf{Type}.\\ (X_i \to Y_i)_{i:m}, F(\overline{X}) \to F(\overline{Y}) \end{aligned}$$

and are closed under composition $Comp_{m,n}\colon \mathbf{Functor}(m), \mathbf{Functor}(n)^m \to \mathbf{Functor}(n)$. However, general set-theoretic functors are not closed under formation of initial algebra functors, therefore they are not suitable for modeling "inductive datatypes".

An alternative to overcome this deficiency is the category **wFunctor**$(m)$ of $\omega$-colimit preserving $m$-ary functors and natural transformations. **wFunctor**$(m)$ is a full sub-category of **Functor**$(m)$, which is closed not only under composition, but also under formation of initial algebra functors. This means that we have functors $\mu_m\colon \mathbf{wFunctor}(m+1) \to \mathbf{wFunctor}(m)$ and polytypic combinators capturing the initial algebra structures

$$\begin{aligned} intro : \forall m\colon \mathbf{N}.\forall F\colon \mathbf{wFunctor}(m+1).\forall X_{i:m}\colon \mathbf{Type}.\\ F(\overline{X}, \mu_m F(\overline{X})) \to \mu_m F(\overline{X})\\ fold : \forall m\colon \mathbf{N}.\forall F\colon \mathbf{wFunctor}(m+1).\forall X_{i:m}, Y\colon \mathbf{Type}.\\ (F(\overline{X}, Y) \to Y), \mu_m F(\overline{X}) \to Y \end{aligned}$$

Another alternative is provided by the category **RegFunctor**$(m)$ of regular $m$-ary functors and natural transformations. Regular functors are functors which are isomorphic to those built from constant functors, projection functors, sum and product functors, by closing under composition and the formation of initial algebra functors (see [4,10]). A formal grammar for the $m$-ary regular functors $F\colon \mathbf{RegFunctor}(m)$, or simply $F^{(m)}$, is:

| | | |
|---|---|---|
| $F^{(m)}$ ::= | $0 \mid 1$ (only if $m = 0$) | constant functors |
| $\mid$ | $\Pi_i^{(m)}$ | $m$-ary extraction, $i = 0, \ldots, n-1$ |
| $\mid$ | $+ \mid \times$ (only if $m = 2$) | binary sum and product functor |
| $\mid$ | $F^{(k)}\langle F_0^{(m)}, \ldots, F_{k-1}^{(m)}\rangle$ | functor composition |
| $\mid$ | $\mu F^{(m+1)}$ | the initial algebra functor induced by $F$ |

Regular functors support the same combinators ($map$, $intro$ and $fold$) as $\omega$-colimit functors, as they are closed under formation of initial algebra functors. In the sequel we will show that functors shapely over lists provide an even better alternative.

The versatility and usefulness of monadic programming has been demonstrated by several researchers, and this has led the Haskell language designers to support this programming style by introducing suitable qualified kinds (see [8]). It is important to keep a clear distinction between computational monads and datatypes. Computational monads are mainly for structuring control, while the main purpose of datatypes is for structuring data. We write **Monad** for the category of monads $T$ on **Set** and monad morphisms. There are two equivalent definitions of monad:

(i) the first defines a monad as a 1-ary functor $T$ equipped with two natural transformations $\eta_T: X \to TX$ and $\mu_T: T^2X \to TX$ satisfying three equational laws;

(ii) the other defines a monad (more precisely a Kleisli's triple) as an action on objects $T$ equipped with two polymorphic operations $\eta_T: X \to TX$ and $-^*_T: (X \to TY) \to TX \to TY$ satisfying three equational laws.

Both definitions are easy to formalize in a calculus.

(i) The first definition inherits the *map* combinator for **Functor**(1) and adds the combinators

$$\begin{aligned} sng &: \forall T{:}\,\mathbf{Monad}.\forall X{:}\,\mathbf{Type}.X \to TX \\ flat &: \forall T{:}\,\mathbf{Monad}.\forall X{:}\,\mathbf{Type}.T(TX) \to TX \end{aligned}$$

satisfying the equational laws (the last two express naturality)

$$\begin{aligned} flat_T\ (sng_T\ u) &= u \\ flat_T\ (map_T\ sng_T\ u) &= u \\ flat_T\ (map_T\ flat_T\ u) &= flat_T\ (flat_T\ u) \\ map_T\ f\ (sng_T\ x) &= sng_T\ (f\ x) \\ flat_T\ (map_T\ (map_T\ f)\ u) &= map_T\ f\ (flat_T\ u) \end{aligned}$$

Here and in the sequel, when instantiating a polytypic combinator, we make explicit the arity and functor parameters (while type parameters are left implicit).

(ii) The second definition is more direct and simply adds the combinators

$$\begin{aligned} val &: \forall T{:}\,\mathbf{Monad}.\forall X{:}\,\mathbf{Type}.X \to TX \\ let &: \forall T{:}\,\mathbf{Monad}.\forall X,Y{:}\,\mathbf{Type}.(X \to TY), TX \to TY \end{aligned}$$

satisfying the equational laws

$$let_T\ val_T = id$$

$$let_T\ f\ (val_T\ x) = f\ x$$
$$(let_T\ g) \circ (let_T\ f) = let_T\ ((let_T\ g) \circ f)$$

We take the Kleisli's triple definition as primitive, and adopt the syntax of [12], namely

$$[x]_T \triangleq val_T\ x \qquad \mathrm{let}_T\ x{\Leftarrow}e_1\ \mathrm{in}\ e_2 \triangleq let_T\ (\lambda x.e_2)\ e_1$$

Moreover, we write $\mathrm{let}_T\ x_{i:n}{\Leftarrow}e_{i:n}$ in $e$ as shorthand for

$$\mathrm{let}_T\ x_0{\Leftarrow}e_0\ \mathrm{in}\ (\ldots(\mathrm{let}_T\ x_{n-1}{\Leftarrow}e_{n-1}\ \mathrm{in}\ e)).$$

In this setting a **monad morphism** $\sigma: S \to T$ is simply a family of functions $\langle \sigma_X: SX \to TX | X \in \mathbf{Set}\rangle$ satisfying two equational laws

$$\sigma_X\ [x]_S = [x]_T \qquad \sigma_X\ (\mathrm{let}_S\ x{\Leftarrow}e_1\ \mathrm{in}\ e_2) = \mathrm{let}_T\ x{\Leftarrow}(\sigma_X\ e_1)\ \mathrm{in}\ (\sigma_X\ e_2)$$

The combinators *map*, *sng* and *flat* can be defined using *val* and *let* as follows

$$map_T\ f\ t \triangleq \mathrm{let}_T\ x{\Leftarrow}t\ \mathrm{in}\ [f\ x]_T$$
$$sng_T\ t \triangleq [t]_T$$
$$flat_T\ t \triangleq \mathrm{let}_T\ x{\Leftarrow}t\ \mathrm{in}\ x$$

### *2.3 Functors shapely over lists*

The notion of functor shapely over lists makes sense in any locos (see [7]), but for simplicity we consider it only in **Set**. The paradigmatic example of functor shapely over lists is the list functor $LX = \coprod_{n:\mathbf{N}} X^n$ itself.

**Definition 2.1** *A natural transformation $\delta: F \to G: \mathbf{C}_1 \to \mathbf{C}_2$ is* **cartesian** *iff the naturality following squares are pullbacks*

$$\begin{array}{ccc} FY & \xrightarrow{\delta_Y} & GY \\ {\scriptstyle Ff}\uparrow & & \uparrow{\scriptstyle Gf} \\ FX & \xrightarrow[\delta_X]{} & GX \end{array}$$

*The functor $L_m: \mathbf{Set}^m \to \mathbf{Set}$ is defined as $L_m(\overline{X}) = L(\coprod_{i:m} X_i)$. An m-ary* **functor shapely over list** *is a pair $(F, \delta)$ s.t. $F: \mathbf{Set}^m \to \mathbf{Set}$ and $\delta: F \to L_m$ is a cartesian natural transformation. A* **shapely morphism**

*$\tau\colon (F,\delta) \to (F',\delta')$ between functors shapely over lists is a (cartesian) natural transformation $\tau\colon F \to F'$ s.t. $\delta' \circ \tau = \delta$ (which implies cartesianity of $\tau$).*

**ShFunctor**$(m)$ *is the category of $m$-ary functors shapely over lists and shapely morphisms.*

**Remark 2.2** In [7] there are two definitions of $m$-ary functor shapely over lists. One requires a cartesian natural transformation $\delta\colon F \to L_m$ as above, the other requires a cartesian natural transformation $\delta'\colon F \to L^m$, where $L^m(\overline{X}) = \prod_{i:m} L(X_i)$. The existence of cartesian natural transformations

$$\prod_{i:m} L(X_i) \xrightarrow{in} L(\coprod_{i:m} X_i) \xrightarrow{out} \prod_{i:m} L(X_i)$$

rendered the two definitions *interchangeable* for the purposes of that paper. However, the existence of such transformations is not enough to establish an equivalence between categories, since the definition of morphism between functors shapely over lists is fairly restrictive. For our purposes the definition in terms of $L(\coprod_{i:m} X_i)$ is preferable, because it provides a global ordering for traversing the data in $F(\overline{X})$.

The functors shapely over lists enjoy many desirable closure properties, like those we have stated for $\omega$- and regular functors, suitably extended to handle the cartesian natural transformation $\delta$ (see [7]). Therefore, functors shapely over lists are good candidates for modeling datatype constructors. Section 4.1 further reinforces this claim, by showing that they support interesting polytypic combinators (which are unlikely to be available for wider classes of functors). Moreover, there are functors shapely over lists which are not regular, particularly those representing array types, such as matrices. This is because the regular functors have a very close relationship to context-free languages that is not shared by the array functors. Let us elaborate.

**Definition 2.3** *Given $(F,\delta)$ functor shapely over list of arity $n$, the language $\mathcal{L}_F$ over the finite alphabet $n$ (i.e. the set of predecessors of $n$) is the image of $\#_F = \delta_{\overline{1}}\colon F(\overline{1}) \to L(n)$. We say that $(F,\delta)$ has* **context-free size** *iff $\mathcal{L}_F$ is a context-free language.*

**Proposition 2.4** *Functors having context-free size are closed under composition and formation of initial algebras. Regular functors have context-free size.*

**Proof** Let $G$ and each $F_i$ be such functors. Each of the grammars for the corresponding context-free languages can be chosen so that their sets of non-terminal symbols are pairwise disjoint. The grammar for $G\langle\overline{F}\rangle$ is obtained by taking the union of all their productions, with the modification that whenever the terminal symbol $i$ appears in a production of $G$ then it is replaced by the start symbol of $F_i$. Let $F$ have context-free size and arity $n+1$. The grammar for $\mathcal{L}_{\mu F}$ is obtained from that of $\mathcal{L}_F$ by replacing all occurrences of the symbol $n$ in productions by the start symbol. The rest is trivial. $\square$

**Corollary 2.5** *The square matrix functor given by* $M(A) = \coprod_{n:\mathbf{N}} A^{n*n}$ *is shapely over lists but not regular.*

**Proof** Clearly $M$ is shapely over lists. The language $\mathcal{L}_M$ is isomorphic to the set of squares $\{n^2 | n \geq 0\}$ which is not context-free by a classical application of the pumping lemma. Hence, $M$ does not have context-free size and so cannot be a regular functor (in the case of a unary functor $F$ any choice of cartesian natural transformation $\delta: F \to L$ induces the same $\mathcal{L}_F$). $\square$

A functor shapely over lists $(F, \delta)$ is determined *up to iso* by the *object of shapes* $F(\overline{1})$ and the map $\delta_{\overline{1}}: F(\overline{1}) \to L_m(\overline{1})$, where $L_m(\overline{1})$ is just $L(m)$ by definition of $L_m$. This observation is technically very useful, since one can work with the simpler category $\mathbf{Set}^{L(m)}$ (or $\mathbf{Set}/L(m)$), which is equivalent to that of $m$-ary functors shapely over lists, and transfer (categorical) properties of the first to the latter. For instance, we can say that $\mathbf{ShFunctor}(m)$ is locally small (i.e. the hom-sets are small), because $\mathbf{Set}^{L(m)}$ is locally small.

**Proposition 2.6** *The category* $\mathbf{Set}^{L(m)}$ *is equivalent to* $\mathbf{ShFunctor}(m)$*, and the equivalence functor from the first to the latter is defined as follows*

- *a family* $\langle C_l | l: L(m) \rangle = \langle C_{\langle n,i \rangle} | \langle n, i \rangle: \coprod_{n:\mathbf{N}} m^n \rangle$ *of sets is mapped to the functor* $(F, \delta)$ *shapely over* $L_m$*, which we call in* ***canonical form****.*

  $F: \mathbf{Set}^m \to \mathbf{Set}$ *is given by:*

  · *on objects* $X_i$ *for* $i: m$

$$F(\overline{X}) = \coprod_{n:\mathbf{N}, i:m^n} (C_{n,i} \times \prod_{j:n} X_{i(j)})$$

  · *on morphisms* $f_i: X_i \to Y_i$ *for* $i: m$

$$F\ \overline{f}\ \langle n: \mathbf{N}, i: m^n, c: C_{n,i}, x: \prod_{j:n} X_{i(j)} \rangle = \langle n, i, c, (\lambda j: n. f_{i(j)}\ x_j) \rangle$$

  $\delta: F \to L_m$ *is given by:*

$$\delta_{\overline{X}}\ \langle n: \mathbf{N}, i: m^n, c: C_{n,i}, x: \prod_{j:n} X_{i(j)} \rangle = \langle n, (\lambda j: n. in_{i(j)}\ x_j) \rangle$$

- *a family* $\langle h_{\langle n,i \rangle}: C_{\langle n,i \rangle} \to D_{\langle n,i \rangle} | \langle n, i \rangle: L(m) \rangle$ *of functions is mapped to the shapely morphism* $\tau: F \to G$ *given by*

$$\tau_{\overline{X}}\ \langle n: \mathbf{N}, i: m^n, c: C_{n,i}, x: \prod_{j:n} X_{i(j)} \rangle = \langle n, i, (h_{n,i}\ c), x \rangle$$

  *where* $F$ *and* $G$ *are the functors corresponding to* $\langle C_{\langle n,i \rangle} | \langle n, i \rangle: L(m) \rangle$ *and* $\langle D_{\langle n,i \rangle} | \langle n, i \rangle: L(m) \rangle$*.*

The result generalizes to a locos $\mathbf{C}$, provided one takes $\mathbf{C}/L(m)$ as the equivalent category.

The list functor preserves all $\omega$-colimits. In **Set** such colimits are preserved by pulling back, so that all functors shapely over list have this property, too. That these functors form a proper subclass of the $\omega$-colimit preserving functors is illustrated by the finite powers set functor, $\mathcal{P}_f$, whose object of shapes $\mathcal{P}_f(1) = 2$ is too small to represent all possible shapes of finite sets.

## 3 Polytypic traversal in programming

Suppose we have a language supporting a polytypic programming style with the usual polytypic combinators *map* and *fold*, i.e.

$$\begin{aligned}
&\mathit{map}: \forall m\colon \mathbf{N}.\forall F\colon \mathbf{Datatype}(m).\forall X_{i:m}, Y_{i:m}\colon \mathbf{Type}.\\
&\qquad (X_i \to Y_i)_{i:m}, F(\overline{X}) \to F(\overline{Y})\\
&\mathit{fold}: \forall m\colon \mathbf{N}.\forall F\colon \mathbf{Datatype}(m+1).\forall X_{i:m}, Y\colon \mathbf{Type}.\\
&\qquad (F(\overline{X}, Y) \to Y), \mu_m F(\overline{X}) \to Y
\end{aligned}$$

For the developments in this section, it is irrelevant what class of functors corresponds to the qualified kinds $\mathbf{Datatype}(m)$, provided it is closed under the formation of initial algebra functors. We assume that the language supports also monadic programming, in particular it has a qualified kind **Monad** (no relation is assumed between $\mathbf{Datatype}(1)$ and **Monad**) and combinators

$$\begin{aligned}
&\mathit{val}: \forall T\colon \mathbf{Monad}.\forall X\colon \mathbf{Type}.X \to TX\\
&\mathit{let}: \forall T\colon \mathbf{Monad}.\forall X, Y\colon \mathbf{Type}.(X \to TY), TX \to TY.
\end{aligned}$$

We outline the advantages of having also a polytypic traversal:

$$\begin{aligned}
&\mathit{traverse}: \forall m\colon \mathbf{N}.\forall F\colon \mathbf{Datatype}(m).\forall T\colon \mathbf{Monad}.\forall X_{i:m}\colon \mathbf{Type}.\\
&\qquad\qquad F(TX_i)_{i:m} \to T(F(\overline{X}))
\end{aligned}$$

We illustrate the expressiveness of this polytypic combinator by deriving polytypic combinators for monadic fold (e.g. [11]) and monadic map. More surprisingly, the existence of *traverse* implies that every $F\colon \mathbf{Datatype}(m)$ is equipped with operations capable of extracting data, and to combine data and shape into values, frequently those of a functor shapely over lists (see Section 4). Finally, we consider a simple programming exercise exemplifying the usefulness of monadic map.

**Example 3.1** The types of the polytypic combinators for monadic map and monadic fold are:

$$mmap : \forall m{:}\, \mathbf{N}. \forall F{:}\, \mathbf{Datatype}(m). \forall T{:}\, \mathbf{Monad}. \forall X_{i:m}, Y_{i:m}{:}\, \mathbf{Type}.$$
$$\quad (X_i \to TY_i)_{i:m}, F(\overline{X}) \to T(F(\overline{Y}))$$
$$mfold : \forall m{:}\, \mathbf{N}. \forall F{:}\, \mathbf{Datatype}(m+1). \forall T{:}\, \mathbf{Monad}. \forall X_{i:m}, Y{:}\, \mathbf{Type}.$$
$$\quad (F(\overline{X}, Y) \to TY), \mu_m F(\overline{X}) \to TY$$

We show that both of them are definable using *traverse* (and the polytypic combinators *map* and *fold*). Again, when instantiating a polytypic combinator, we make explicit the arity and functor parameters (while type parameters are left implicit).

$$mmap_{F,T}\ \overline{f}\ t = traverse_{F,T}\ (map_F\ \overline{f}\ t)$$
$$mfold_{F,T}\ f = fold_F\ f' \text{ where } f'{:}\, F(\overline{X}, TY) \to TY \text{ is given by}$$
$$f\ u = \mathrm{let}_T\ v \Leftarrow (mmap_{F,T}\ (\lambda x{:}\, X_i.[x]_T)_{i:m}\ id\ u) \text{ in } fv$$

Notice that the definition of $mmap_{F,T}$ does not exploit the monad structure of $T$, while the definition of $mfold_{F,T}$ makes essential used of it. Furthermore the definition of $mmap_{F,T}$ depends only on the instance $traverse_{F,T}$ of the polytypic traversal, and similarly $mfold_{F,T}$ depends only on $traverse_{F,T}$. Finally, one can recover $traverse_{F,T}$ from $mmap_{F,T}$ (because *map* preserves identities)

$$traverse_{F,T} = mmap_{F,T}\ \overline{(\lambda u{:}\, TX_i.u)}.$$

**Example 3.2** We show how the existence of traversals allows us to define the decomposition of a data structure into the list of data, $\delta_X{:}\, FX \to LX$, and the shape, $\#_X{:}\, FX \to F1$ (we consider for simplicity the unary case). This decomposition is at the basis of the definition of functors shapely over lists (see section 2.3 and [7]).

Let's consider the following monad $TY = M \times Y$, where $M$ is the monoid $(LX, @, [])$, with @ the concatenation of lists and [] the empty list. Note that the choice of $T$ varies with each choice of datatype $X$.

The data-shape decomposition for $FX$ is given by:

$$\langle \delta_X, \#_X \rangle = mmap_{F,T}\ f : FX \to LX \times F1$$

where $f{:}\, X \to LX \times 1$ is defined as $f\ x\ = \langle [x], * \rangle$

Assuming the naturality of *traverse* w.r.t. the monad parameter (see Section 4.1 and Theorem 4.8), we can deduce the naturality of the transformation $\delta$ from $F$ to $L$. Naturality is one of the properties, beside cartesianity, required by the definition of functors shapely over lists.

**Example 3.3** We can define also the partial inverse to the shape-data decomposition. Let's consider the following monad $TY = M \to (M \times Y) + 1$, where $M$ is the monoid $(LX, @, [])$. Given a shape and a list of data the function

$insert: F1 \to LX \to FX + 1$ fills the shape with the data, failing if either there is not enough data or there is any data left.

$$insert = h \circ (mmap_{F,T}\ g): F1 \to LX \to FX + 1$$

where

$$mmap_{F,T}: (1 \to (LX \to LX \times X + 1)) \to F1 \to LX \to LX \times FX + 1$$

and $g: 1 \to LX \to LX \times X + 1$ is defined as $\begin{cases} g\ u\ nil = in_1\ u \\ g\ u\ (x::xs) = in_0\ \langle xs, x \rangle \end{cases}$, i.e. $g$ is the iso which attempts to decompose a list into its head and tail.

So $mmap_{F,T}\ g: F1 \to LX \to LX \times FX + 1$ takes a shape and a list and returns, if there is enough data to fill the shape, a pair consisting of the rest of the data and a datatype corresponding to the shape. If there is any data left over in the list then *insert* fails, so we represent this by another function $h: LX \times FX + 1 \to FX + 1$.

### *3.1 An alpha-conversion algorithm for a generic λ-calculus*

Alpha-conversion takes a lambda abstraction $\lambda x.t$ and renames the bound variable to a variable which is not free in $t$. Generally speaking, $\alpha$-conversion denotes the contextual and transitive closure of the relation defined above.

We define a function that renames all the bound variables in a term with fresh variables (chosen in a suitable way). This guarantees that there will be no conflict when the term is $\beta$-reduced. The function is described using a pseudo-language which supports traversal and polytypic definitions. Polytypic definitions allows us to define a function that works not only for terms of a particular lambda calculus but for a class of extensions of the basic calculus.

The syntax of terms follows the Combinatory Reduction Systems notation (see [9]), so a term is either a variable, or an abstraction, or a $n$-ary function symbol applied to $n$ terms. Thus we define the following type that is parameterized over a type constructor $F$, which takes into account the function symbols:

$$Term = \text{var}\ N |\ \text{bind}\ N\ Term\ |\ \text{other}\ (F(Term))$$

where variables are represented by natural numbers. Examples of $F$ include $F(X) = \text{lam}\ X |\ \text{app}\ X\ X$, for the basic lambda calculus.

In order to define the function that renames all the bound variables, we need to supply a source of new variable names and a storage for keeping the information about the names that have to rename variable occurrences. This is obtained using the following monad:

$$S(X) = N \times R \to \text{Maybe}(X \times N)$$

where $\text{Maybe}(X) = X + 1$ is the error monad, and $R = N \to \text{Maybe}(N)$ is the type of a "renaming" function. We can see the monad $S$ as the combination of a side-effect monad $S_1(X) = N \to (X \times N)$ and a state-reading monad $S_2(X) = R \to \text{Maybe}(X)$. The side-effect monad supplies the source of new variable names and the state-reading monad keeps the information about the names that have to "rename" variable occurrences.

The functions $val_S$ and $let_S$ associated to the monad $S$ are defined as follows:

$$\begin{aligned} [x]_S\ \langle m, r\rangle &= [\langle x, m\rangle]_{\text{Maybe}} \\ (\text{let}_S\ x \Leftarrow u \text{ in } f)\ \langle m, r\rangle &= \text{let}_{\text{Maybe}}\ \langle y, n\rangle \Leftarrow u\langle m, r\rangle \text{ in } (\lambda x.f)\ y\ \langle n, r\rangle. \end{aligned}$$

The function

$$aconv : \forall F : \textbf{Functor}(1).\ Term \to S(Term)$$

takes a term and renames each free variable according to the "renaming" function in the state and each bound variable with a fresh variable according to the state in the side-effect monad. It distinguishes three cases: if the term is a variable then it applies the "renaming" function; if the term is an abstraction then it renames the bound variable with a new fresh variable; otherwise it traverses the term computing the $\alpha$-conversion of the sub-terms.

The source of errors comes from the initial state given in input to the $aconv$ function. The initial state should be the pair $\langle m+1, id_m\rangle : N \times R$, where $m$ is the maximum variable in the term and $id_m\ i = \text{if } i \leq m \text{ then } [i]_{\text{Maybe}} \text{ else fail}$. This means that an error arises when, visiting the term, we find a variable greater than the maximum fixed in the initial state.

The definition is as follows:

$$\begin{aligned} aconv\ (\text{var } i) &= \lambda\langle n, r\rangle.map_{\text{Maybe}}\ (\lambda j.(\text{var } j, n))\ (r\ i) \\ aconv\ (\text{bind } i\ t) &= \lambda\langle n, r\rangle.map_{\text{Maybe}}\ (\lambda(u, m).(\text{bind } n\ u, m)) \\ &\qquad\qquad (aconv\ t\ (n+1, update\ i\ n\ r)) \\ aconv\ (\text{other } u) &= map_S\ \text{other}\ (mmap\ aconv\ u) \end{aligned}$$

where $update : N \to N \to R \to R$ is

$$update\ i\ j\ r\ x\ = if\ i == x\ then\ [j]_{\text{Maybe}}\ else\ (r\ x)$$

i.e. it updates a "renaming" function $r$ with $j$ for $i$.

## 4 Traversals

This section investigates the semantics of traversals in **Set**. For the sake of simplicity, we consider only unary functors (endofunctors), but definitions and results extend to functors of any arity. Furthermore, the following definitions and results can be recast in greater generality (see Section 6).

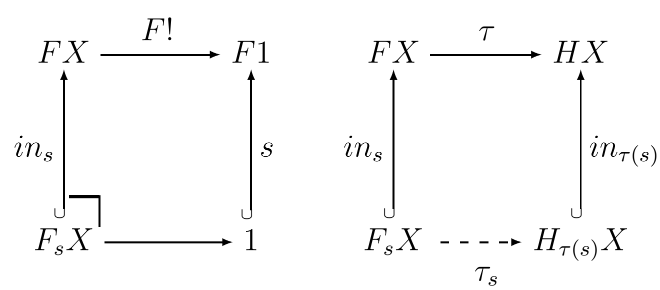
The paradigmatic example of traversal is the traversal (from left to right) of the list functor $L$ by a monad $T$. This is the family of maps $\zeta\colon L(TX) \to T(LX)$ mapping the list $(u_i\colon TX|i\colon n)$ to $\mathrm{let}_T\ x_{i:n}{\Leftarrow}u_{i:n}$ in $[(x_i\colon X|i\colon n)]_T$.

This traversal suggests several properties that a *traversal* of a functor $F$ by a monad $T$ (or by another functor $G$) ought to satisfy. In particular, the length-preservation property can be recast as a shape-preservation property. More precisely, we expect that a traversal $\zeta\colon F(GX) \to G(FX)$ of a functor $F$ by a functor $G$ should satisfy the property "the $F$-shape in the result is the same as that of the argument". The simplest way of formalizing this property is to introduce notation for $F$-data structures with a given shape.

**Notation 4.1** Given an endofunctor $F\colon \mathbf{Set} \to \mathbf{Set}$ and an $F$-shape $s\colon F1$, the endofunctor $F_s\colon \mathbf{Set} \to \mathbf{Set}$ and the natural transformation $in_s\colon F_s \to F$ are defined as follows:

– $F_sX$ is the subset $\{u\colon FX|F!u = s\}$ of the elements of $FX$ *with shape* $s$,
– $in_{sX}$ is its inclusion into $FX$, and
– $F_sf$ is the restriction of $Ff$ to elements of shape $s$.

Furthermore, a natural transformation $\tau\colon F \to H$ and an $F$-shape $s\colon F1$ induce a natural transformation $\tau_s\colon F_s \to H_{\tau(s)}$ obtained by restricting $\tau_X\colon FX \to HX$ to $F_sX$. More abstractly, we define $F_s$, $in_s$ and $\tau_s$ as follows




is immediate to see that $[in_{sX}|s\colon F1]\colon \coprod_{s:F1} F_sX \to FX$ is an iso natural in $X$.

**Definition 4.2 (Traversal)** *Given endofunctors $F$ and $G$, a* traversal of $F$ by $G$ *is a natural transformation* $\zeta_X\colon F(GX) \to G(FX)$ induced *by a family* $\zeta_{sX}\colon F_s(GX) \to G(F_sX)$ *of natural transformations indexed by* $s\colon F1$*, i.e. for* *every* $s{:}\,F1$

$$\begin{array}{ccc} F(GX) & \xrightarrow{\ \zeta\ } & G(FX) \\ {\scriptstyle in_s}\uparrow & & \uparrow{\scriptstyle G(in_s)} \\ F_s(GX) & \xrightarrow[\ \zeta_s\ ]{} & G(F_sX) \end{array}$$

*The natural transformation* $\zeta$ *is uniquely determined by the family* $\zeta_{s:F1}$*, but the converse does not hold in general (namely when* $G$ *does not preserve monos like* $in_s$*), hence it is mathematically more convenient to work with the family* $\zeta_{s:F1}$*. In the sequel we use traversal to refer (ambiguously) to both* $\zeta$ *and* $\zeta_{s:F1}$*.*

Note that in **Set** almost all monos split, and the functors that do not preserve monos are quite odd, e.g. $GX = \begin{cases} A & \text{if } X = \emptyset \\ 1 & \text{otherwise} \end{cases}$

**Example 4.3** We reconsider in the light of the above definition the paradigmatic traversal of the list functor $L$ by a monad $T$. The set of $L$-shapes is $L1 = N$, whereas the functor $L_m$ is $L_mX = X^m$. Therefore a traversal of $L$ by $T$ is given by a family $\zeta_{m_X}{:}\,(TX)^m \to T(X^m)$ of natural transformations. In particular, the family inducing the traversal from left to right is

$$u_{i:m} \xmapsto{\zeta_{m_X}} \text{let}_T\ x_{i:m}{\Leftarrow}u_{i:m} \text{ in } [x_{i:m}]_T$$

Actually what is used for defining $\zeta_m$ is not the monad structure on $T$, but the induced monoidal functor structure, i.e. the natural transformations $\phi{:}\,1 \to T1$ and $\phi{:}\,TX \times TY \to T(X \times Y)$. Other traversals of $L$ by $T$ can be obtained by choosing a permutation on $m$ for each $m{:}\,\mathbf{N}$, in particular the traversal defined above corresponds to taking the identity permutation for each $m$.

### *4.1 Traversal of functors shapely over lists by monads*

In Section 3 we have shown the usefulness of the polytypic combinator

$$\begin{aligned} &\mathit{traverse} : \forall m{:}\,\mathbf{N}.\forall F{:}\,\mathbf{Datatype}(m).\forall T{:}\,\mathbf{Monad}.\forall X_{i:m}{:}\,\mathbf{Type}. \\ &\qquad F(TX_i)_{i:m} \to T(F(\overline{X})). \end{aligned}$$

In this section we show that such a combinator has a semantic counterpart when $\mathbf{Datatype}(m)$ is the class $\mathbf{ShFunctor}(m)$ of functors shapely over lists. The definition is a simple generalization of the paradigmatic example of traversal from left to right of the list functor by a monad (see Example 4.3). This is done by exploiting the equivalence between $\mathbf{ShFunctor}(m)$ and $\mathbf{Set}^{L(m)}$ (thus considering only functors shapely over lists in canonical form).

**Definition 4.4** *Given a functor $F$ shapely over lists (in canonical form) and a monad $T$, the* traversal $\zeta_{F,T_X} : F(TX) \to T(FX)$ from left to right of $F$ by $T$ *is the natural transformation induced by the family $\langle \zeta_{F,T,s_X} : F_s(TX) \to T(F_sX) | s: F1\rangle$ of natural transformations s.t.*

$$u_{i:m} : F_{\langle m,c\rangle}(TX) \xmapsto{\zeta_{F,T,\langle m,c\rangle_X}} \mathrm{let}_T\ x_{i:m} \Leftarrow u_{i:m}\ \mathrm{in}\ [x_{i:m}]_T$$

*where $F$ corresponds to the family of sets $\langle C_m | m \in N\rangle$, i.e. $FX = \coprod_{m:\mathbf{N}} C_m \times X^m$, therefore a shape $s: F1$ is a pair $\langle m: \mathbf{N}, c: C_m\rangle$ and $F_{\langle m,c\rangle}(X) = X^m$.*

**Remark 4.5** Let $\delta: F \to L$ be a cartesian natural transformation over $\mathbf{Set}$ and $\zeta$ the corresponding family of traversals, then the data-shape decomposition induced by $\zeta$ as described in Example 3.2 is $\langle \delta, F!\rangle$, i.e. one recovers $\delta$ from $\zeta$.

The properties of the family $\langle \zeta_{F,T} | F: \mathbf{ShFunctor}(1), T: \mathbf{Monad}\rangle$ of traversals given above are summarized by the following theorems. We consider both higher-order naturality properties, relating traversals for different $F$s and $T$s, and equational properties relating $\zeta_{F,T}$ to the monad structure on $T$.

**Theorem 4.6 (Local properties)** *The traversal $\zeta_{F,T_X} : F(TX) \to T(FX)$ satisfies the properties:*

– *preservation of values, i.e.*

$$\begin{array}{ccc} & & FX \\ & \swarrow^{F(val_T)} & \downarrow{\scriptstyle val_T} \\ F(TX) & \xrightarrow[\zeta_{F,T}]{} & T(FX) \end{array}$$

– *preservation of Kleisli compositions, i.e.*

$$\begin{array}{ccccc} F(T^2X) & \xrightarrow{\zeta_{F,T}} & T(F(TX)) & \xrightarrow{T(\zeta_{F,T})} & T^2(FX) \\ \downarrow{\scriptstyle F(flat_T)} & & & \swarrow_{flat_T} & \\ F(TX) & \xrightarrow[\zeta_{F,T}]{} & T(FX) & & \end{array}$$

*provided the monad $T$ is commutative, i.e.*
$\mathrm{let}_T\ x_1, x_2 \Leftarrow e_1, e_2\ \mathrm{in}\ e = \mathrm{let}_T\ x_2, x_1 \Leftarrow e_2, e_1\ \mathrm{in}\ e$.

**Remark 4.7** The above result says that when $T$ is commutative, the traversal $\zeta_{F,T_X} : F(TX) \to T(FX)$ is a distributive law in the sense of Arbib and Manes, and therefore $F$ extends to a functor on the Kleisli category $\mathbf{Set}_T$ for $T$, as done in [4] (without commutativity we can define an action on morphisms of $\mathbf{Set}_T$, but it fails to preserve composition). Many interesting monads (e.g. side-effects and exceptions) are not commutative, therefore traversals represent a useful generalization. For some applications (e.g. parallel programming or databases) it is quite convenient to restrict to commutative monads, since one may rearrange the order of evaluation without affecting the final result. On one hand this leaves greater opportunities for optimization, on the other it allows to extend traversals beyond functors shapely over lists (e.g. bags).

**Theorem 4.8 (Global properties)** *The family $\zeta_{F,T_X}\colon F(TX) \to T(FX)$ satisfies the properties:*

- *naturality in $F$, i.e. for any shapely morphism $\tau\colon F \to G$*

$$\begin{array}{ccc} F(TX) & \xrightarrow{\zeta_{F,T}} & T(FX) \\ {\scriptstyle\tau}\downarrow & & \downarrow{\scriptstyle T(\tau)} \\ G(TX) & \xrightarrow[\zeta_{G,T}]{} & T(GX) \end{array}$$

- *naturality in $T$, i.e. for any monad morphism $\sigma\colon S \to T$*

$$\begin{array}{ccc} F(SX) & \xrightarrow{\zeta_{F,S}} & S(FX) \\ {\scriptstyle F(\sigma)}\downarrow & & \downarrow{\scriptstyle \sigma} \\ F(TX) & \xrightarrow[\zeta_{F,T}]{} & T(FX) \end{array}$$

**Remark 4.9** In the definition of the category $\mathbf{ShFunctor}(m)$ we have taken as objects pairs $(F, \delta)$ consisting of a functor and a cartesian natural transformation. For instance, $(L, id)$ and $(L, rev)$, where $rev\colon LX \to LX$ is the map reversing a list, are two different objects of $\mathbf{ShFunctor}(1)$. Also the notion of shapely morphism should not be overlooked, it is far more restrictive than a natural transformation, e.g. $rev\colon L \to L$ is the only shapely morphism from$(L, id)$ to $(L, rev)$ (in fact these objects are terminal in $\mathbf{ShFunctor}(1)$). On the other hand, there are infinitely many (cartesian) natural transformation $\tau\colon L \to L$.

One may ask whether the family $\zeta_{F,T}$ satisfies a stronger form of naturality in $F$, allowing any natural transformation $\tau\colon F \to G$. The following counter-example shows that such a requirement is incompatible with many monads.

Let $FX = X^n$, $GX = 1$, and $!: FX \to 1$ be the unique natural transformation. The stronger naturality property would imply that

$$\begin{array}{ccc} (T1)^n & \xrightarrow{\zeta_{F,T}} & T1 \\ {\scriptstyle !}\downarrow & & \Vert\, {\scriptstyle T!} \\ 1 & \xrightarrow{val_T} & T1 \end{array}$$

i.e. executing $n$ computations has no effect.

The properties we have established for the family of traversals $\zeta_{F,T}$ do not characterize it uniquely. However, the way the list functor is traversed by a monad $T$, fully determines the traversal of other functors shapely over lists.

**Proposition 4.10** *If* $\langle \zeta'_{F,s_X}: F_s(GX) \to G(F_sX) | F: \mathbf{ShFunctor}(1), s: F1\rangle$ *is a family of traversals by an endofunctor G which is natural in F (in the sense of Theorem 4.8), then* $\zeta'_L$ *uniquely determines* $\zeta'_F$*.*

**Proof** Let $\delta: F \to L$ be the cartesian natural transformation which makes $F$ shapely over lists, i.e. $\delta: (F, \delta) \to (L, id)$ is a morphism in the category $\mathbf{ShFunctor}(1)$. Since $\zeta'$ is natural in $F: \mathbf{ShFunctor}(1)$, we have that

$$\begin{array}{ccc} F_s(GX) & \xrightarrow{\zeta'_{F,G,s}} & G(F_sX) \\ {\scriptstyle \delta_s}\downarrow & & \downarrow{\scriptstyle G(\delta_s)} \\ L_{\delta(s)}(GX) & \xrightarrow[\zeta'_{L,G,\delta(s)}]{} & G(L_{\delta(s)}X) \end{array}$$

Moreover $\delta_s: F_s \to L_{\delta(s)}$ is a natural iso, because $\delta$ is cartesian. Therefore $\zeta'_{F,G,s}$ is uniquely determined by $\zeta'_{L,G,\delta(s)}$. $\Box$

## 5 Zips revised

In the introduction we mentioned zips as examples of distributive laws, whose purpose is to *commute* the order of two data structures. A well-known example of zip is the function mapping two lists $[a_i|i: m]$ and $[b_i|i: m]$ of the same length to the list of pairs $[(a_i, b_i)|i: m]$, this is a zip of the product functor (of arity 2) by the list functor (of arity 1). A *polytypic* notion of zip has been investigated in a relational setting by [5,6]. Roughly speaking, a *zip* between two *relators* (i.e. endofunctors in the category of relations) $F$ by $G$ is a natural transformation $\xi_X : F(GX) \rightsquigarrow G(FX)$, satisfying certain additional properties. The paradigmatic example of zip is given by the zip of the list functor/relator $L$ by itself. This is the family of relations $\xi_X : L(LX) \rightsquigarrow L(LX)$ relating the list of lists $((x_{i,j}|j{:}\,n)|i{:}\,m)$ to $((x_{i,j}|i{:}\,m)|j{:}\,n)$. If we fix the lengths $m$ and $n$ this amounts to transposition of $m \times n$ matrices.

As prerequisite for comparing the notion of zip and traversal, we recast the definition of zip in a functional setting. By analogy with our definition of traversal, we take as fundamental a shape-preservation property.

**Definition 5.1 (Zip)** *A* zip of $F$ by $G$ *is a family* $\xi_X : F(GX) \rightsquigarrow G(FX)$ *of spans* induced *by a family* ${\xi_{s,t}}_X : F_s(G_tX) \tilde{\to} G_t(F_sX)$ *of natural isos indexed by* $s{:}\,F1$ *and* $t{:}\,G1$*, i.e. the span* $F(GX) \leftarrow \coprod_{s,t} F_s(G_tX) \to G(FX)$ *s.t. for each* $s{:}\,F1$ *and* $t{:}\,G1$

$$F(GX) \xleftarrow{F(in_t)} F(G_tX) \xleftarrow{in_s} F_s(G_tX) \overset{\xi_{s,t}}{=\!=} G_t(F_sX) \xrightarrow{in_t} G(F_sX) \xrightarrow{G(in_s)} G(FX)$$

*The span* $\xi_X : F(GX) \rightsquigarrow G(FX)$ *is uniquely determined by the family* $\xi_{s:F1,t:G1}$*, but the converse does not hold in general, hence it is mathematically more convenient to work directly with the family* $\xi_{s:F1,t:G1}$ *of natural isos. In the sequel we use zip to refer (ambiguously) to both* $\xi$ *and* $\xi_{s:F1,t:G1}$*.*

**Remark 5.2** We have defined zip differently from [5,6] (and in a simpler setting), nevertheless we have captured the *shape preservation* property (see Section 5.3.2 in [6]) and the symmetry between $F$ and $G$ (which explains why we require the $\xi_{s,t}$ to be isos). The other properties of zip given in [6] involve a class of relators, and therefore cannot be captured in our definition. Once the definition of zip has been recast in terms of a family of natural isos $\xi_{s,t}$, it is immediate to see the difference with the definition of traversal. Zips are inherently symmetric, a zip $\xi_{s,t}$ of $F$ by $G$ induces a zip $\xi_{s,t}^{-1}$ of $G$ by $F$. When there is only one $G$-shape, i.e. $G1 = 1$, then a zip of $F$ by $G$ is also a traversal of $F$ by $G$ (but the converse fails).

**Example 5.3** We reconsider in the light of the above definition the paradigmatic zip of the list functor $L$ by itself. Such a zip is induced by the family of the natural isos ${\xi_{m,n}}_X : (X^n)^m \tilde{\to} (X^m)^n$ given by transposition of $m \times n$ matrices. Other zips of $L$ by $L$ can be obtained by choosing a permutation on $m \times n$ for each $m, n{:}\,\mathbf{N}$.

It is immediate to generalize the paradigmatic example of zip to functors shapely over lists.

**Definition 5.4** *Given two endofunctors* $F$ *and* $G$ *shapely over lists (and in canonical form), the* zip ${\xi_{F,G}}_X : F(GX) \rightsquigarrow G(FX)$ of $F$ by $G$ *is the family of relations induced by the family* $\langle {\xi_{F,G,s,t}}_X : F_s(G_tX) \tilde{\to} G_t(F_sX) | s{:}\,F1, t{:}\,G1\rangle$ *of natural isos given by transposition* $(X^n)^m \tilde{\to} (X^m)^n$*, where* $F_{\langle m,c\rangle}(X) = X^m$

*and* $G_{\langle n,d\rangle}(X) = X^n$.

**Remark 5.5** We now compare the family of zips $\xi_{F,G,s,t}$ defined above with the family of traversals $\zeta_{F,T,s}$ given in Definition 4.4. Firstly, the family $\xi_{F,G,s,t}$ of zips satisfies a stronger naturality property in $F$ (and $G$), namely for any natural transformation $\tau\colon F \to H$ between functors shapely over lists

$$\begin{array}{ccc} F_s(G_t X) & \overset{\xi_{F,G,s,t}}{=\!=\!=\!=} & G_t(F_s X) \\ {\scriptstyle \tau_s}\big\downarrow & & \big\downarrow{\scriptstyle G_t(\tau_s)} \\ H_{\tau(s)}(G_t X) & \underset{\xi_{H,G,\tau(s),t}}{=\!=\!=\!=} & G_t(H_{\tau(s)} X) \end{array}$$

On the other hand, the family $\zeta_{F,T,s}$ of traversals is natural in $F$ only w.r.t. shapely morphisms $\tau\colon F \to H$. Secondly, it is quite easy to extend $\xi_{F,G,s,t}$ beyond functors shapely over lists, e.g. by considering functors of the form $FX = \coprod_{s:S} X^{E_s}$ where $E\colon S \to \mathit{Set}$ is any family of sets, while it seems unlikely that $\zeta_{F,T,s}$ can be extended. Finally, when $GX = TX = X^n$, and so there is only one $G$-shape, we have that $\xi_{F,G,s,*} = \zeta_{F,T,s}$.

## 6 Traversals and Zips in a Locos

In this section we outline how to extend the notion of traversal and zip to a locos (see [3]). Conceptually there are no difficulties to extend the results from **Set** to any locos, once the right definitions are in place.

Functors shapely over lists make sense in a locos, but the notion of traversal can be defined in the more general setting of a lex-category $\mathbf{C}$ (i.e. a category with finite limits). When **Set** is replaced by $\mathbf{C}$, a traversal $\zeta$ of $F$ by $G$ will be determined by a family $\zeta_s$ of natural transformations indexed by an object of $\mathbf{C}$. Therefore, we need to think in terms of fibrations over $\mathbf{C}$.

**Definition 6.1** *Given a lex-category* $\mathbf{C}$*, we write* $\mathit{Fib}(\mathbf{C})$ *for the 2-category of fibrations over* $\mathbf{C}$*.*

*Given an object* $S \in \mathbf{C}$*, we write* $\_^S$ *for the 2-functor on* $\mathit{Fib}(\mathbf{C})$ *mapping a* *fibration* $p: \mathbf{E} \to \mathbf{C}$ *to the fibration* $p^S: \mathbf{E}^S \to \mathbf{C}$ *given by*

$$\begin{array}{ccc} \mathbf{E}^S & \dashrightarrow & \mathbf{E} \\ {\scriptstyle p^S} \downarrow & \lrcorner & \downarrow {\scriptstyle p} \\ \mathbf{C} & \xrightarrow[S \times \_]{} & \mathbf{C} \end{array}$$

*(therefore the fiber* $\mathbf{E}^S_I$ *is* $\mathbf{E}_{S\times I}$*), and similarly for* $\mathbf{C}$*-fibered functors and natural transformations.*

*For every* $\mathbf{C}$*-fibration* $\mathbf{E}$ *there is a* $\mathbf{C}$*-fibered functor* $\Delta_S: \mathbf{E} \to \mathbf{E}^S$ *s.t.* $\Delta_{S,I} \triangleq \pi_1^*: \mathbf{E}_I \to \mathbf{E}_{S\times I}$.

*By abuse of language, we write* $\mathbf{C}$ *for "*$\mathbf{C}$ *fibered over itself", i.e. the* $\mathbf{C}$*-fibration* $cod: \mathbf{C}^{\to} \to \mathbf{C}$.

**Remark 6.2** There is an equivalence between endofunctors on **Set** and **Set**-fibered endofunctors on **Set** fibered over itself, and so one may safely confuse functors and natural transformations with their **Set**-fibered counterparts. For this reason we have decided to confine ourselves to **Set** in the first part of the exposition.

**Proposition 6.3** *The* $\mathbf{C}$*-fibered functor* $\Delta_S: \mathbf{C} \to \mathbf{C}^S$ *has a* $\mathbf{C}$*-fibered left adjoint* $\Sigma_S: \mathbf{C}^S \to \mathbf{C}$*. Every* $\mathbf{C}$*-fibered endofunctor* $F: \mathbf{C} \to \mathbf{C}$ *can be described as a composite* $\mathbf{C} \xrightarrow{\widehat{F}} \mathbf{C}^{S_F} \xrightarrow{\Sigma_{S_F}} \mathbf{C}$*, where* $S_F \triangleq F_1(1)$ *is the object of* $F$-shapes *and* $\widehat{F}$ *preserves the unit objects in the fibers.*

**Proof** The $\mathbf{C}$-fibered functor $\Sigma_S: \mathbf{C}^S \to \mathbf{C}$ is given by composition, namely it maps the object $X \xrightarrow{x} S \times I$ in the fiber $\mathbf{C}^S_I = \mathbf{C}_{S\times I}$ to the object $X \xrightarrow{x} S \times I \xrightarrow{\pi_1} I$ in $\mathbf{C}_I$.

There is only one way to define $\widehat{F}$ to ensure the required properties, since $\widehat{F}_I$ must map the morphism

$$\begin{array}{ccc} X & \xrightarrow{!} & I \\ & {\scriptstyle x} \searrow & \| \\ & & I \end{array}$$

in the fiber $\mathbf{C}_I$ to

$$\begin{array}{ccc} F_I(X) & \xrightarrow{F_I(!)} & F_I(1) \\ & {\scriptstyle \widehat{F}_I(x)} \searrow & \| \\ & & S_F \times I \end{array}$$

in the fiber $\mathbf{C}^{S_F}_I = \mathbf{C}_{S_F \times I}$. $\square$

**Remark 6.4** Informally speaking, the objects of the $\mathbf{C}$-fibration $\mathbf{E}^S$ are $S$-indexed families of objects in $\mathbf{E}$. Therefore, the $\mathbf{C}$-fibered functor $\widehat{F}$ maps objects in $\mathbf{C}$ to $S_F$-indexed families of objects. When $\mathbf{C}$ is $\mathbf{Set}$, to give a functor $\widehat{F}\colon \mathbf{Set} \to \mathbf{Set}^{F1}$ is the same as to give a family of functors $F_{s:F1}\colon \mathbf{Set} \to \mathbf{Set}$. These considerations suggest that one should recast the set-theoretic definition of traversal and zip in terms of $\widehat{F}$.

**Definition 6.5 (Traversal)** *Given* $\mathbf{C}$*-fibered endofunctors* $F$ *and* $G$ *on* $\mathbf{C}$*, a* traversal of $F$ by $G$ *is a* $\mathbf{C}$*-fibered natural transformation*

$$\begin{array}{ccc} \mathbf{C} & \xrightarrow{\widehat{F}} & \mathbf{C}^{S_F} \\ {\scriptstyle G}\downarrow & \overset{\widehat{\zeta}}{\Rightarrow} & \downarrow{\scriptstyle G^{S_F}} \\ \mathbf{C} & \xrightarrow[\widehat{F}]{} & \mathbf{C}^{S_F} \end{array}$$

*where* $S_F$ *is the object of* $F$*-shapes.*

**Definition 6.6 (Zip)** *Given* $\mathbf{C}$*-fibered endofunctors* $F$ *and* $G$ *on* $\mathbf{C}$*, a* zip of $F$ by $G$ *is a* $\mathbf{C}$*-fibered natural iso*

$$\begin{array}{ccc} \mathbf{C} & \xrightarrow{\widehat{F}} & \mathbf{C}^{S_F} \\ {\scriptstyle \widehat{G}}\downarrow & \overset{\widehat{\xi}}{\Rightarrow} & \downarrow{\scriptstyle \widehat{G}^{S_F}} \\ \mathbf{C} & \xrightarrow[\widehat{F}^{S_G}]{} & \mathbf{C}^{S_F \times S_G} \end{array}$$

*where* $S_F$ *is the object of* $F$*-shapes and* $S_G$ *is the object of* $G$*-shapes.*

## References


[1] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming - an introduction. Lecture notes for the summer school on Advanced Functional Programming, 1998. In press for LNCS, 1999.

[2] M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer Verlag, 1985.

[3] J.R.B. Cockett. List-aritmetic distributive categories: locoi. *Journal of Pure and Applied Algebra*, 66:1–29, 1990.

[4] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.

[5] P. Hoogendijk and R. Backhouse. When do datatypes commute? In E. Moggi and G. Rosolini, editors, *CTCS*, LNCS 1290, pages 242–260. Springer-Verlag, September 1997.

[6] Paul F. Hoogendijk. *A Generic Theory of Data Types*. PhD thesis, Technische Universiteit Eindhoven, 1997.

[7] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[8] Mark P. Jones. A theory of qualified types. In *ESOP'92*, LNCS 582. Springer Verlag, 1992.

[9] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI, Amsterdam, 1980.

[10] L. Meertens. Calculate Polytypically! In Kuchen and Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, LNCS 1140, pages 1–16. Springer-Verlag, 1996.

[11] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Proceedings of the First International Springschool on Advanced Functional Programming Techniques*, LNCS 925, pages 228–266. Springer-Verlag, 1995.

[12] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[13] J. Vinárek. Extensions of symmetric hom-functors to the Kleisli category. In Ferenc Gécseg, editor, *Proceedings of the 1981 International Conference on Fundamentals of Computation Theory*, LNCS 117, pages 394–399, Szeged, Hungary, August 1981. Springer-Verlag.

[14] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 1993. Special issue of selected papers from 6'th Conference on Lisp and Functional Programming, 1992.