Constructing Applicative Functors

Ross Paterson

City University London, UK

Abstract. Applicative functors define an interface to computation that is more general, and correspondingly weaker, than that of monads. First used in parser libraries, they are now seeing a wide range of applications. This paper sets out to explore the space of non-monadic applicative functors useful in programming. We work with a generalization, lax monoidal functors, and consider several methods of constructing useful functors of this type, just as transformers are used to construct computational monads. For example, coends, familiar to functional programmers as existential types, yield a range of useful applicative functors, including left Kan extensions. Other constructions are final fixed points, a limited sum construction, and a generalization of the semi-direct product of monoids. Implementations in Haskell are included where possible.

1 Introduction

This paper is part of a tradition of applying elementary category theory to the design of program libraries. Moggi [16] showed that the notion of monad could be used to structure denotational descriptions of programming languages, an idea carried over to program libraries by Wadler [20]. It turns out that the monads useful in semantics and programming can be constructed from a small number of monad transformers also identified by Moggi [17].

Applicative functors [15] provide a more limited interface than monads, but in return have more instances. All monads give rise to applicative functors, but our aim is to explore the space of additional instances with applications to programming. We are particularly interested in general constructions, with which programmers can build their own applicative functors, knowing that they satisfy the required laws. It is already known that applicative functors, unlike monads, can be freely composed. We identify a number of further general constructions, namely final fixed points, a limited sum construction, a generalization of semi-direct products of monoids, and coends (including left Kan extensions). By combining these constructions, one can obtain most of the computational applicative functors in the literature, with proofs of their laws. General constructions also clarify the relationships between seemingly unrelated examples, and suggest further applications.

Elementary category theory provides an appropriately abstract setting for the level of generality we seek. An idealized functional language corresponds to a type of category with first-class functions (a cartesian closed category). Applicative functors on such a category are equivalent to a simpler form called lax monoidal functors, which are more convenient to work with. We can build up lax monoidal functors in more general ways by ranging across several different categories, as long as the end result acts on the category of our functional language, and is thus applicative. Familiarity with the basic definitions of categories and functors is assumed. The other notions used are mostly shallow, and will be explained along the way.

In the next section, we introduce applicative and lax monoidal functors. The rest of the paper describes the general constructions, illustrated with examples in Haskell where possible. Two proof styles are used throughout the paper. When making statements that apply to any category, we use standard commuting diagrams. However many statements assume a cartesian closed category, or at least a category with products. For these we use the internal language of the category, which provides a term language with equational reasoning that will be familiar to functional programmers.

2 Applicative Functors

The categorical notion of "functor" is modelled in Haskell with the type class

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Instances include a variety of computational concepts, including containers, in which `fmap` modifies elements while preserving shape. Another class of instances are "notions of computation", including both monads and applicative functors, in which terms of type $F\,a$ correspond to computations producing values of type a, but also having an "effect" described by the functor F, e.g. modifying a state, possibly throwing an exception, or non-determinism. The requirement that F be a functor allows one to modify the value returned without changing the effect.

The applicative interface adds pure computations (having no effect) and an operation to sequence computations, combining their results. It is described by a type class:

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

If we compare this with the type class of monads:

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

we see that `pure` corresponds to `return`; the difference lies in the sequencing operations. The more powerful `>>=` operation available with monads allows the choice of the second computation to depend on the result of the first, while in the applicative case there can be no such dependency. Every monad can be made an applicative functor in a uniform way, here illustrated with the `Maybe` monad:

```
instance Functor Maybe where
    fmap f m = m >>= \ x -> return (f x)

instance Applicative Maybe where
    pure = return
    mf <*> mx = mf >>= \ f -> mx >>= \ x -> return (f x)
```

For functors that are also monads the monadic interface is often more convenient, but here we shall be more interested in applicative functors that are not also monads. A simple example is a constant functor returning a monoid [15]. Here is that functor expressed in Haskell using the `Monoid` class, which defines an associative binary operation `<>` with identity `mempty`:

```
newtype Constant a b = Constant a

instance Functor (Constant a) where
    fmap f (Constant x) = Constant x

instance Monoid a => Applicative (Constant a) where
    pure _ = Constant mempty
    Constant x <*> Constant y = Constant (x <> y)
```

The more limited applicative interface has many more instances, some of which will be presented in later sections. For example, the constrained form of sequencing offered by the applicative interface makes possible instances in which part of the value is independent of the results of computations, e.g. parsers that pregenerate parse tables [18]. Unlike monads, applicative functors are closed under composition.

However many applications of monads, such as traversal of containers, can be generalized to the applicative interface [15].

2.1 Lax Monoidal Functors

The applicative interface is convenient for programming, but in order to explore relationships between functors we shall use an alternative form with a more symmetrical sequencing operation:

```
class Functor f => Monoidal f where
    unit :: f ()
    mult :: f a -> f b -> f (a, b)
```

This interface, with identity and associativity laws, is equivalent to the applicative interface—the operations are interdefinable:

```
pure x = fmap (const x) unit
a <*> b = fmap (uncurry id) (mult a b)

unit = pure ()
mult a b = fmap (,) a <*> b
```

If we uncurry the operation `mult` of the `Monoidal` class, we obtain an operation $\circledast : F\,a \times F\,b \to F\,(a \times b)$. This suggests generalizing from products to other binary type constructors, a notion known in category theory as a monoidal category.

A *monoidal category* [13] consists of a category $\mathcal{C}$, a functor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ and an object $\top$ of $\mathcal{C}$, with coherent natural isomorphisms

$$
\begin{array}{ll}
\lambda : \top \otimes a \cong a & \text{(left identity)} \\
\rho : a \otimes \top \cong a & \text{(right identity)} \\
\alpha : a \otimes (b \otimes c) \cong (a \otimes b) \otimes c & \text{(associativity)}
\end{array}
$$

A *symmetric monoidal category* also has

$$\sigma : a \otimes b \cong b \otimes a \qquad \text{(symmetry)}$$

Both products and coproducts are examples of monoidal structures, and both are also symmetric. Given a monoidal category $\langle \mathcal{C}, \top, \otimes, \lambda, \rho, \alpha \rangle$, the category $\mathcal{C}^{\mathrm{op}}$, obtained by reversing all the morphisms of $\mathcal{C}$, also has a monoidal structure: $\langle \mathcal{C}^{\mathrm{op}}, \top, \otimes, \lambda^{-1}, \rho^{-1}, \alpha^{-1} \rangle$. The product of two monoidal categories is also monoidal, combining the isomorphisms of the two categories in parallel.

Often we simply refer to the category when the monoidal structure is clear from the context.

Some functors preserve this structure exactly, with $\top' = F\,\top$ and $F\,a \otimes' F\,b = F\,(a \otimes b)$; a trivial example is the identity functor. Others, such as the product functor $\times : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ preserve it up to isomorphism:

$$
\begin{array}{c}
1 \cong 1 \times 1 \\
(a_1 \times a_2) \times (b_1 \times b_2) \cong (a_1 \times b_1) \times (a_2 \times b_2)
\end{array}
$$

We obtain a larger and more useful class of functors by relaxing further, requiring only morphisms between the objects in each pair, thus generalizing the class `Monoidal` above from products to any monoidal category.

A *lax monoidal functor* between monoidal categories $\langle \mathcal{C}, \otimes, \top \rangle$ and $\langle \mathcal{C}', \otimes', \top' \rangle$ consists of a functor $F : \mathcal{C} \to \mathcal{C}'$ with natural transformations

$$
\begin{array}{ll}
u : \top' \to F\,\top & \text{(unit)} \\
\circledast : F\,a \otimes' F\,b \to F\,(a \otimes b) & \text{(multiplication)}
\end{array}
$$

such that the following diagrams commute:

$$
\begin{tikzcd}
\top \otimes F\,a \arrow[r, "\lambda"] \arrow[d, "u \otimes F\,a"'] & F\,a \\
F\,\top \otimes F\,a \arrow[r, "\circledast"'] & F\,(\top \otimes a) \arrow[u, "F\,\lambda"']
\end{tikzcd}
\qquad
\begin{tikzcd}
F\,a \otimes \top \arrow[r, "\rho"] \arrow[d, "F\,a \otimes u"'] & F\,a \\
F\,a \otimes F\,\top \arrow[r, "\circledast"'] & F\,(a \otimes \top) \arrow[u, "F\,\rho"']
\end{tikzcd}
$$

$$
\begin{tikzcd}
F\,a \otimes (F\,b \otimes F\,c) \arrow[r, "F\,a \otimes \circledast"] \arrow[d, "\alpha"'] & F\,a \otimes F\,(b \otimes c) \arrow[r, "\circledast"] & F\,(a \otimes (b \otimes c)) \arrow[d, "F\,\alpha"] \\
(F\,a \otimes F\,b) \otimes F\,c \arrow[r, "\circledast \otimes F\,c"'] & F\,(a \otimes b) \otimes F\,c \arrow[r, "\circledast"'] & F\,((a \otimes b) \otimes c)
\end{tikzcd}
$$

The first two diagrams state that u is the left and right identity respectively of the binary operation $\circledast$, while the last diagram expresses the associativity of $\circledast$.

2.2 Weak Commutativity

Although the definition of a lax monoidal functor neatly generalizes the `Monoidal` class, it lacks the counterpart of `pure`. We will also want an associated axiom stating that pure computations can be commuted with other computations. (There is a notion of symmetric lax monoidal functor, but requiring the ability to swap any two computations would exclude too many functors useful in computation, where the order in which effects occur is often significant.)

Thus we define an *applicative functor* on a symmetric monoidal category $\mathcal{C}$ as consisting of a lax monoidal functor $F : \mathcal{C} \to \mathcal{C}$, with a natural transformation $p : a \to F\,a$ (corresponding to the `pure` function of the `Applicative` class) satisfying $p_\top = u$ and $p \circ \circledast = \circledast \circ p \otimes p$, plus a weak commutativity condition:

$$\begin{tikzcd}
a \otimes F\,b \arrow[r, "p \otimes F\,b"] \arrow[d, "\sigma"'] & F\,a \otimes F\,b \arrow[r, "\circledast"] & F\,(a \otimes b) \arrow[d, "F\,\sigma"] \\
F\,b \otimes a \arrow[r, "F\,b \otimes p"'] & F\,b \otimes F\,a \arrow[r, "\circledast"'] & F\,(b \otimes a)
\end{tikzcd}$$

We could also express the weak commutativity condition as a constraint on functors with a tensorial strength, but here we shall avoid such technicalities by assuming that function spaces are first-class types, with primitives to perform application and currying, or in categorical terms that we are working in a cartesian closed category (ccc). In particular, if $\mathcal{A}$ is a ccc, any lax monoidal functor $F : \mathcal{A} \to \mathcal{A}$ is also applicative. To show this, we make use of another advantage of working in a ccc, namely that we can conduct proofs in the internal λ-calculus of the category [12], in which variables of type a stand for arrows of $\mathcal{A}(1, a)$, and we write $f\,(e_1, \ldots, e_n)$ for $f \circ \langle e_1, \ldots, e_n \rangle$. The result is a convenient language that is already familiar to functional programmers. When working in categories with products we shall calculate using the internal language; when products are not assumed we shall use diagrams.

In the internal language, we can define $p : I \dot{\to} F$ with the counterpart of the above definition of `pure` for any `Monoidal` functor:

$$p\,x = F\,(const\,x)\,u$$

The proof of weak commutativity is then a simple calculation in the internal language:

$$
\begin{array}{rlr}
F\,\sigma\,(p\,x \circledast y) & = F\,\sigma\,(F\,(\mathit{const}\,x)\,u \circledast y) & \text{definition of } p \\
& = F\,(\sigma \circ (\mathit{const}\,x) \times \mathit{id})\,(u \circledast y) & \text{naturality of } \circledast \\
& = F\,(\sigma \circ (\mathit{const}\,x) \times \mathit{id})\,(F\,\lambda^{-1}\,y) & \text{left identity} \\
& = F\,(\mathit{id} \times (\mathit{const}\,x) \circ \sigma)\,(F\,\lambda^{-1}\,y) & \text{naturality of } \sigma \\
& = F\,(\mathit{id} \times (\mathit{const}\,x) \circ \sigma \circ \lambda^{-1})\,y & \text{functor} \\
& = F\,(\mathit{id} \times (\mathit{const}\,x) \circ \rho^{-1})\,y & \text{symmetry} \\
& = F\,(\mathit{id} \times \mathit{const}\,x)\,(F\,\rho^{-1}\,y) & \text{functor} \\
& = F\,(\mathit{id} \times \mathit{const}\,x)\,(y \circledast u) & \text{right identity} \\
& = y \circledast F\,(\mathit{const}\,x)\,u & \text{naturality of } \circledast \\
& = y \circledast p\,x & \text{definition of } p
\end{array}
$$

It is also known that lax monoidal functors in a ccc are equivalent to closed functors [5], which resemble the `Applicative` interface, but again the lax monoidal form is more convenient for defining derived functors.

Thus our strategy will be to construct a lax monoidal functor over the product structure of a ccc, but we may construct it from constituents involving other monoidal categories. As a simple example, we have seen that the product functor $\times : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is lax monoidal, and we can compose it with the diagonal functor from $\mathcal{A}$ to $\mathcal{A} \times \mathcal{A}$ (also lax monoidal) to obtain a lax monoidal functor from $\mathcal{A}$ to $\mathcal{A}$:

$$F\,a = a \times a$$

though in this case the resulting functor is also monadic. In Section 5 we also use auxiliary categories with monoidal structures other than products.

3 Fixed Points, Limits and Colimits

A standard example of a computational monad is the list monad, which may be used to model backtracking. There is another lax monoidal functor on lists, with a unit constructing infinite lists and multiplication forming the zip of two lists [15]:

```
data ZipList a = Nil | Cons a (ZipList a)

instance Functor ZipList where
    fmap f Nil = Nil
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)

instance Monoidal ZipList where
    unit = Cons () unit
    mult (Cons x xs) (Cons y ys) = Cons (x,y) (mult xs ys)
    mult _ _ = Nil
```

It turns out that this instance, and the proof that it satisfies the lax monoidal laws, follow from a general construction. We can observe that `ZipList` is a fixed point through the second argument of the binary functor $F(a,b) = 1 + a \times b$. That is, F is the functor $\texttt{Maybe} \circ \times$, a composition of two lax monoidal functors and therefore lax monoidal.

There are two canonical notions of the fixed point of a functor, the initial and final fixed points, also known as data and codata. Initial fixed points can be used to define monads; here we use final fixed points to define lax monoidal functors. Recall that a parameterized final fixed point of a functor $F : \mathcal{A} \times \mathcal{B} \to \mathcal{B}$ consists of a functor $\nu F : \mathcal{A} \to \mathcal{B}$ with an isomorphism $c : F\,(a, \nu F\,a) \cong \nu F\,a$ and an unfold operator $[\!(\cdot)\!]$ constructing the unique morphism satisfying

$$\begin{tikzcd}
b \arrow[r, dashed, "{[\!(f)\!]}"] \arrow[d, "f"'] & \nu F\,a \\
F\,(a,b) \arrow[r, dashed, "{F\,(a,[\!(f)\!])}"'] & F\,(a, \nu F\,a) \arrow[u, "c"']
\end{tikzcd}$$

If F is lax monoidal, we can define the unit and multiplication morphisms of a lax monoidal structure on νF as two of these unfolds:

$$\begin{tikzcd}
\top \arrow[r, dashed, "{u_{\nu F} = [\!(u_F)\!]}"] \arrow[d, "u_F"'] & \nu F\,\top \\
F\,(\top,\top) \arrow[r, dashed, "{F\,(\top, u_{\nu F})}"'] & F\,(\top, \nu F) \arrow[u, "c"']
\end{tikzcd}$$

$$\begin{tikzcd}
\nu F\,a_1 \otimes \nu F\,a_2 \arrow[r, dashed, "{\circledast_{\nu F} = [\!(\circledast_F \circ c^{-1} \otimes c^{-1})\!]}"] \arrow[d, "c^{-1} \otimes c^{-1}"'] & \nu F\,(a_1 \otimes a_2) \\
F\,(a_1, \nu F\,a_1) \otimes F\,(a_2, \nu F\,a_2) \arrow[d, "\circledast_F"'] & \\
F\,(a_1 \otimes a_2, \nu F\,a_1 \otimes \nu F\,a_2) \arrow[r, dashed, "{F\,(a_1 \otimes a_2, \circledast_{\nu F})}"'] & F\,(a_1 \otimes a_2, \nu F\,(a_1 \otimes a_2)) \arrow[uu, "c"']
\end{tikzcd}$$

In particular, for $F = \texttt{Maybe} \circ \times$, this construction yields a lax monoidal functor equivalent to `ZipList` above.

One can prove using fusion that this definition does indeed satisfy the lax monoidal laws, but we shall prove a more general result instead.

3.1 Limits

Ignoring the parameter $\mathcal{A}$ for the moment, another way to define the final fixed point of a functor $F : \mathcal{B} \to \mathcal{B}$ starts with the terminal object 1. Using with the unique morphism $!_{F\,1} : F\,1 \to 1$, we can define a chain of objects and morphisms:

$$\cdots \longrightarrow F^3\,1 \xrightarrow[F^2\,!_{F\,1}]{} F^2\,1 \xrightarrow[F\,!_{F\,1}]{} F\,1 \xrightarrow[!_{F\,1}]{} 1$$

The final fixed point $\nu\,F$ is defined as the limit of this chain, an object with a commuting family of morphisms (a *cone*) to the objects of the chain:

$$\begin{tikzcd}
\nu F \arrow[dr] \arrow[drr] \arrow[drrr] \arrow[drrrr] & & & & \\
\cdots \arrow[r] & F^3\,1 \arrow[r, "F^2\,!_{F\,1}"'] & F^2\,1 \arrow[r, "F\,!_{F\,1}"'] & F\,1 \arrow[r, "!_{F\,1}"'] & 1
\end{tikzcd}$$

such that any other such cone, say from an object B, can be expressed as a composition of a unique morphism $B \to \nu F$ and the cone from νF.

This construction is sufficient for final fixed points of regular functors like `ZipList`, but for the general case we need to lift the whole thing to the category $\mathbf{Fun}(\mathcal{A}, \mathcal{B})$, whose objects are functors $\mathcal{A} \to \mathcal{B}$, and whose morphisms are natural transformations. Given a functor Φ on this category, we can repeat the above construction in the functor category, starting with the constant functor $\overline{1}$:

$$\begin{tikzcd}
\nu \Phi \arrow[dr] \arrow[drr] \arrow[drrr] \arrow[drrrr] & & & & \\
\cdots \arrow[r] & \Phi^3\,\overline{1} \arrow[r, "\Phi^2\,!_{\Phi\,\overline{1}}"'] & \Phi^2\,\overline{1} \arrow[r, "\Phi\,!_{\Phi\,\overline{1}}"'] & \Phi\,\overline{1} \arrow[r, "!_{\Phi\,\overline{1}}"'] & \overline{1}
\end{tikzcd}$$

A standard result holds that limits in $\mathbf{Fun}(\mathcal{A}, \mathcal{B})$ may be constructed from pointwise limits in $\mathcal{B}$ [13, p. 112].

On the way to defining the final fixed point of Φ as a lax monoidal functor, we wish to require that Φ preserve lax monoidal functors. To state this, we need a specialized notion of natural transformation for lax monoidal functors: a *monoidal transformation* between lax monoidal functors $\langle F, \circledast, \top \rangle$ and $\langle F', \circledast', \top' \rangle$ is a natural transformation $h : F \stackrel{.}{\to} F'$ that preserves the lax monoidal operations:

$$\begin{tikzcd}
 & \top \arrow[dl, "u"'] \arrow[dr, "u'"] & \\
F\,\top \arrow[rr, "h"'] & & F'\,\top
\end{tikzcd}
\qquad
\begin{tikzcd}
F\,a \otimes F\,b \arrow[r, "h \otimes h"] \arrow[d, "\circledast"'] & F'\,a \otimes F'\,b \arrow[d, "\circledast'"] \\
F\,(a \otimes b) \arrow[r, "h"'] & F'\,(a \otimes b)
\end{tikzcd}$$

Then given monoidal categories $\mathcal{A}$ and $\mathcal{B}$, we can define a category $\mathbf{Mon}(\mathcal{A}, \mathcal{B})$ with lax monoidal functors as objects and monoidal transformations between them as morphisms. Now suppose we have a diagram in $\mathbf{Mon}(\mathcal{A}, \mathcal{B})$, e.g. the chain

$$\cdots \longrightarrow F_3 \xrightarrow[f_2]{} F_2 \xrightarrow[f_1]{} F_1 \xrightarrow[f_0]{} F_0$$

We can construct a limit in $\mathbf{Fun}(\mathcal{A}, \mathcal{B})$ (from pointwise limits in $\mathcal{B}$):

$$
\begin{tikzcd}
& F \arrow[d, "t_3"'] \arrow[dr] \arrow[drr] \arrow[drrr, "t_0"] \\
\cdots \arrow[r] & F_3 \arrow[r, "f_2"'] & F_2 \arrow[r, "f_1"'] & F_1 \arrow[r, "f_0"'] & F_0
\end{tikzcd}
$$

To extend F to a limit of this diagram in $\mathbf{Mon}(\mathcal{A}, \mathcal{B})$, we want to define operations u and $\circledast$ on F such that the t_i are monoidal transformations, i.e. satisfying the following equations in $\mathcal{B}$:

$$
\begin{tikzcd}
& \top \arrow[dl, dashed, "u"'] \arrow[dr, "u_i"] & \\
F\,\top \arrow[rr, "t_i"'] & & F_i\,\top
\end{tikzcd}
\qquad
\begin{tikzcd}
F\,a \otimes F\,b \arrow[r, "t_i \otimes t_i"] \arrow[d, dashed, "\circledast"'] & F_i\,a \otimes F_i\,b \arrow[d, "\circledast_i"] \\
F\,(a \otimes b) \arrow[r, "t_i"] & F_i\,(a \otimes b)
\end{tikzcd}
$$

These equations imply that u and $\circledast$ are mediating morphisms to the limits in $\mathcal{B}$, and thus uniquely define them. It remains to show that $\circledast$ is a natural transformation, and that u and $\circledast$ satisfy the identity and associativity laws. Each of these four equations is proven in the same way: we show that the two sides of the equation are equalized by each t_i, as a consequence of the corresponding equation on F_i, and thus, by universality, must be equal. For example, for the left identity law we have the diagram

$$
\begin{tikzcd}
\top \otimes F\,a \arrow[rrr, "\lambda"] \arrow[dr, "\top \otimes t_i"] \arrow[ddd, "u \otimes F\,a"'] & & & F\,a \arrow[dl, "t_i"] \\
& \top \otimes F_i\,a \arrow[r, "\lambda"] \arrow[d, "u_i \otimes F_i\,a"'] & F_i\,a & \\
& F_i\,\top \otimes F_i\,a \arrow[r, "\circledast_i"'] & F_i\,(\top \otimes a) \arrow[u, "F_i\,\lambda"'] & \\
F\,\top \otimes F\,a \arrow[ur, "t_i \otimes t_i"] \arrow[rrr, "\circledast"'] & & & F\,(\top \otimes a) \arrow[ul, "t_i"'] \arrow[uuu, "F\,\lambda"']
\end{tikzcd}
$$

The central panel is the left identity law for F_i, while the four surrounding panels follow from the definitions of u and $\circledast$ and the naturality of λ and t_i. Thus the two morphisms $\top \otimes F\,a \to F\,a$ on the perimeter of the diagram is equalized by t_i. Since the universality of the limit implies that such a morphism is unique, they must be equal. We have proven:

Proposition 1. *If $\mathcal{B}$ is complete, then so is* $\mathbf{Mon}(\mathcal{A}, \mathcal{B})$.

Applying this to the chain of the fixed point construction, we have the immediate corollary that the final fixed point of a higher-order functor Φ on lax monoidal functors is a uniquely determined extension of the final fixed point of Φ on ordinary functors. For example, ZipList is the final fixed point of the higher-order functor Φ is defined by $\Phi\, Z\, a = \mathtt{Maybe}\,(a \times Z\, a)$.

3.2 Sums

The dual notion, colimits, is not as easily handled. We can construct sums in special cases, such as adding the identity functor to another lax monoidal functor:

```
data Lift f a = Return a | Others (f a)

instance Functor f => Functor (Lift f) where
    fmap f (Return x) = Return (f x)
    fmap f (Others m) = Others (fmap f m)

instance Monoidal f => Monoidal (Lift f) where
    unit = Return ()
    mult (Return x) (Return y) = Return (x, y)
    mult (Return x) (Others my) = Others (fmap ((,) x) my)
    mult (Others mx) (Return y) = Others (fmap (flip (,) y) mx)
    mult (Others mx) (Others my) = Others (mult mx my)
```

Here pure computations (represented by the identity functor and the constructor `Return`) may be combined with `mult`, but are converted to the other functor if either computation involves that functor.

Applying this construction to the constant functor yields a form of computations with exceptions that collects errors instead of failing at the first error [4, 15]:

```
type Except err a = Lift (Constant err) a
```

That is, in a computation `mult e1 e2`, after a failure in `e1`, the whole computation will fail, but not before executing `e2` in case it produces errors that should be reported together with those produced by `e1`.

The fixed point $L \cong \mathtt{Lift}\,(I \times L)$ expands to non-empty lists combined with a "long zip", in which the shorter list is padded with copies of its last element to pair with the remaining elements of the longer list, as suggested by Jeremy Gibbons and Richard Bird[1]:

```
data PadList a = Final a | NonFinal a (PadList a)

instance Functor PadList where
    fmap f (Final x) = Final (f x)
    fmap f (NonFinal x xs) = NonFinal (f x) (fmap f xs)
```

[1] Personal communication, 5 July 2011.

```
instance Monoidal PadList where
    unit = Final ()
    mult (Final x) (Final y) = Final (x, y)
    mult (Final x) (NonFinal y ys) =
        NonFinal (x, y) (fmap ((,) x) ys)
    mult (NonFinal x xs) (Final y) =
        NonFinal (x, y) (fmap (flip (,) y) xs)
    mult (NonFinal x xs) (NonFinal y ys) =
        NonFinal (x, y) (mult xs ys)
```

A straightforward generalization is $L \cong \texttt{Lift}\,(I \times (F \circ L))$ for any lax monoidal F, defining forms of long zip for various kinds of tree.

The `Lift` construction is able to combine computations in the identity functor with those of another lax monoidal functor F because there is a monoidal transformation between the two, namely the arrow `pure`. We can generalize:

Proposition 2. *If $\mathcal{J}$ is an upper semi-lattice and $\mathcal{B}$ is a ccc with finite coproducts, a diagram $\Delta : \mathcal{J} \to \mathbf{Mon}(\mathcal{A}, \mathcal{B})$ has a colimit.*

Proof. Define a functor F by

$$F\,a = \sum_{j \in \mathcal{J}} C_j\,(\Delta_j\,a)$$
$$F\,f\,(C_j\,x) = C_j\,(f\,x)$$

where the $C_j : \Delta_j \dot{\to} F$ are tagging injections (constructors) marking the terms of the sum. Then we can define a lax monoidal structure on F as follows:

$$u = C_\perp\,u_\perp$$
$$C_j\;a \circledast C_k\;b = C_{j \sqcup k}\;(\Delta_{j \leq j \sqcup k}\;a, \Delta_{k \leq j \sqcup k}\;b)$$

Naturality of $\circledast$ and the identity and associativity laws follow from simple calculations. $\square$

For example, in the case of `Lift`, $\mathcal{J}$ is a two-element lattice $0 \leq 1$, with $\Delta_0 = I$, $\Delta_1 = F$ and $\Delta_{0 \leq 1} = p$.

4 Generalized Semi-direct Products

A pioneering instance of the applicative interface was the parser combinator library of Swierstra and Duponcheel [18], which we here rehearse in a greatly cut-down form.

These parsers are applied to the output of lexical analysis. Given a type `Symbol` enumerating symbol types, parser input consists of a list of `Token`s, recording the symbol type and its text:

```
type Token = (Symbol, String)
```

For example, there might be a `Symbol` for numeric literals, in which case the corresponding `String` would record the text of the number. Parsers take a list of tokens and return either an error string or a parsed value together with the unparsed remainder of the input:

```
newtype Parser a = P ([Token] -> Either String (a, [Token]))
```

This type is a monad (built by adding to an exception monad a state consisting of a list of tokens), and therefore also an applicative functor. Parsers can be built using primitives to peek at the next symbol, to move to the next token returning the string value of the token read, and to abort parsing reporting an error:

```
nextSymbol :: Parser Symbol
advance :: Parser String
throwError :: String -> Parser a
```

In order to construct recursive descent parsers corresponding to phrases of a grammar, one needs to keep track of whether a phrase can generate the empty string, and also the set of symbols that can begin a phrase (its *first* set). Swierstra and Duponcheel's idea was to define a type containing this information about a phrase, from which a deterministic parser for the phrase could be constructed:

```
data Phrase a = Phrase (Maybe a) (Map Symbol (Parser a))
```

The two components are:

- The type `Maybe a` indicates whether the phrase can generate the empty string, and if so provides a default output value.
- The type `Map Symbol (Parser a)` records which symbols can start the phrase, and provides for each a corresponding deterministic parser.

The `Functor` instance for this type follows from the structure of the type:

```
instance Functor Phrase where
    fmap f (Phrase e t) = Phrase (fmap f e) (fmap (fmap f) t)
```

The idea, then, is to build a value of this type for each phrase of the grammar, with the following conversion to a deterministic parser:

```
parser :: Phrase a -> Parser a
parser (Phrase e t)
  | null t = def
  | otherwise = do
        s <- nextSymbol
        findWithDefault def s t
  where
    def = case e of
        Just x -> return x
        Nothing -> throwError ("expected " ++ show (keys t))
```

A parser for a single symbol, returning its corresponding text, is

```
symbol :: Symbol -> Phrase String
symbol s = Phrase Nothing (singleton s advance)
```

Alternatives are easily built:

```
(<|>) :: Phrase a -> Phrase a -> Phrase a
Phrase e1 t1 <|> Phrase e2 t2 =
    Phrase (e1 `mplus` e2) (t1 `union` t2)
```

In a realistic library, one would want to check that at most one of the alternatives could generate the empty string, and that the first sets were disjoint. The information in the `Phrase` type makes it possible to determine this check before parsing, but we omit this in our simplified presentation.

Now the lax monoidal structure corresponds to the empty phrase and concatenation of phrases. A phrase $\alpha\beta$ can generate the empty string only if both the constituent phrases can, but the emptiness information for α also determines whether the initial symbols of $\alpha\beta$ include those of β in addition to those of α:

```
instance Monoidal Phrase where
    unit = Phrase unit empty
    mult (Phrase e1 t1) (~p2@(Phrase e2 t2)) =
        Phrase (mult e1 e2) (union t1' t2')
      where
        t1' = fmap (`mult` parser p2) t1
        t2' = maybe empty (\ x -> fmap (fmap ((,) x)) t2) e1
```

In Haskell, a tilde marks a pattern as lazy, meaning it is not matched until its components are used. It is used here so that `Phrase` values can be recursively defined, as long as one avoids left recursion.

We might wonder whether this definition is an instance of a general construction. We note that the `Phrase` type is a pair, and the first components are combined using the lax monoidal operations on `Maybe`, independent of the second components. This is similar to a standard construction on monoids, the semi-direct product, which takes a pair of monoids $\langle A, *, 1\rangle$ and $\langle X, +, 0\rangle$ with an action $(\cdot) : A \times X \to X$, and defines a monoid on $A \times X$, with binary operation

$$(a, x) \odot (b, y) = (a * b, x + (a \cdot y))$$

and identity $(1, 0)$. For example Horner's Rule for the evaluation of a polynomial $a_n x^n + \cdots + a_1 x + a_0$ can be expressed as a fold of such an operation over the list $[(x, a_0), (x, a_1), \ldots, (x, a_n)]$, with the immediate consequence that the calculation can be performed in parallel (albeit with repeated calculation of the powers of x).

We shall consider a generalization of the semi-direct product on lax monoidal functors, requiring

– a lax monoidal functor $\langle F, \circledast, u\rangle : \langle \mathcal{A}, \otimes, \top\rangle \to \langle \mathcal{B}, \times, 1\rangle$

– a functor $G : \mathcal{A} \to \mathcal{B}$ with a natural family of monoids $\oplus : G\,a \times G\,a \to G\,a$ and $\emptyset : 1 \to G\,a$.
– an operation $\rtimes : G\,a \times (F\,b \times G\,b) \to G\,(a \otimes b)$ distributing over G:

$$\emptyset \rtimes q = \emptyset \tag{1}$$
$$(x \oplus y) \rtimes q = (x \rtimes q) \oplus (y \rtimes q) \tag{2}$$

– an operation $\ltimes : (F\,a \times G\,a) \times G\,b \to G\,(a \otimes b)$ distributing over G:

$$p \ltimes \emptyset = \emptyset \tag{3}$$
$$p \ltimes (x \oplus y) = (p \ltimes x) \oplus (p \ltimes y) \tag{4}$$

also satisfying

$$(p \ltimes y) \rtimes r = p \ltimes (y \rtimes r) \tag{5}$$

Proposition 3. *Given the above functors and operations, there is a lax monoidal functor* $\langle H, \circledast_H, u_H \rangle : \langle \mathcal{A}, \otimes, \top \rangle \to \langle \mathcal{B}, \times, 1 \rangle$ *defined by*

$$\begin{aligned} H\,a &= F\,a \times G\,a \\ u_H &= (u, \emptyset) \\ (a, x) \circledast_H (b, y) &= (a \circledast b, (x \rtimes (b, y)) \oplus ((a, x) \ltimes y)) \end{aligned}$$

provided that $\ltimes$ *and* $\rtimes$ *are left and right actions on* G*, i.e.*

$$u_H \ltimes z = z \tag{6}$$
$$(p \circledast_H q) \ltimes z = p \ltimes (q \ltimes z) \tag{7}$$
$$x \rtimes u_H = x \tag{8}$$
$$x \rtimes (q \circledast_H r) = (x \rtimes q) \rtimes r \tag{9}$$

Proof. It follows from their definitions that H is a functor and $\circledast_H$ a natural transformation. Next, we show that u_H is the left and right identity of $\circledast_H$:

$$\begin{aligned} u_H \circledast_H (b, y) &= (1 \circledast b, (\emptyset \rtimes (b, y)) \oplus (u_H \ltimes y)) && \text{definition of } \circledast_H,\ u_H \\ &= (1 \circledast b, \emptyset \oplus y) && \text{equations (1) and (6)} \\ &= (b, y) && \text{monoid laws} \end{aligned}$$

$$\begin{aligned} (a, x) \circledast_H u_H &= (a \circledast 1, (x \rtimes u_H) \oplus ((a, x) \ltimes \emptyset)) && \text{definition of } \circledast_H,\ u_H \\ &= (a \circledast 1, x \oplus \emptyset) && \text{equations (8) and (3)} \\ &= (a, x) && \text{monoid laws} \end{aligned}$$

Finally, we must show that $\circledast_H$ is associative:

$$
\begin{array}{ll}
((a, x) \circledast_H (b, y)) \circledast_H (c, z) & \\
= (a \circledast b \circledast c, (((x \rtimes (b, y)) \oplus ((a, x) \ltimes y)) \rtimes (c, z)) \oplus & \text{definition of } \circledast_H \\
\qquad\qquad (((a, x) \circledast_H (b, y)) \ltimes z)) & \\
= (a \circledast b \circledast c, ((x \rtimes (b, y)) \rtimes (c, z)) \oplus (((a, x) \ltimes y) \rtimes (c, z)) \oplus & \text{equation (2)} \\
\qquad\qquad (((a, x) \circledast_H (b, y)) \ltimes z)) & \\
= (a \circledast b \circledast c, (x \rtimes ((b, y) \circledast_H (c, z))) \oplus & \text{(9), (5) and (7)} \\
\qquad\qquad ((a, x) \ltimes (y \rtimes (c, z))) \oplus ((a, x) \ltimes ((b, y) \ltimes z))) & \\
= (a \circledast b * c, (x \rtimes ((b, y) \circledast_H (c, z))) \oplus & \text{equation (4)} \\
\qquad\qquad ((a, x) \ltimes (y \rtimes (c, z)) \oplus ((b, y) \ltimes z))) & \\
= (a, x) \circledast_H ((b, y) \circledast_H (c, z)) & \text{definition of } \circledast_H
\end{array}
$$

□

In particular, Proposition 3 identifies the properties we need to establish to demonstrate that `Phrase` is lax monoidal, and thus applicative.

5 Existentials and Coends

Many applicative functors are constructed using existential quantification, hiding a private representation type. We shall consider the corresponding categorical notion, called a coend.

Abbott, Altenkirch and Ghani [1] consider containers of the form

$$L\, c = \exists\, m.\, K\, m \to c$$

Here m is drawn from a set of shapes $\mathcal{M}$ (a discrete category), the functor $K : \mathcal{M} \to \mathcal{C}$ assigns to each shape a set of positions within containers of that shape, and the function provides a value for each of these positions. For example, the shape of a list is a natural number n giving its length, which K maps to the set of positions $\{0, 1, \ldots, n-1\}$, the indices of the list.

There are several ways that we can extend container functors to obtain useful lax monoidal functors.

If we let $\mathcal{M}$ be the natural numbers plus an upper bound ω, and $K\, n = \{i \mid i < n\}$, then L represents finite and infinite lists. We can define lax monoidal operations:

$$
\begin{aligned}
u &= (\omega, \mathit{const}\, \top) \\
(m, f) \circledast (n, g) &= (m \sqcap n, \langle f, g \rangle)
\end{aligned}
$$

That is, u yields an infinite list, and $\circledast$ constructs a list of pairs, whose length is the smaller of the lengths of the arguments. We recognize this functor as another version of the `ZipList` functor defined in Section 3. More generally, if $\mathcal{M}$ has a monoidal structure that is a lower semi-lattice, and $K\, (m_1 \sqcap m_2) \subseteq K\, m_i$, then the lax monoidal structure on L computes zips on containers.

A type comprising arrays of different dimensions can be represented using a shape functor K satisfying $K\,\top \cong 1$ and $K\,(a \otimes b) \cong K\,a \times K\,b$. Then we can define lax monoidal operations with with u constructing a scalar and $\circledast$ being cartesian product:

$$u = (\top, \mathit{const}\,())$$
$$(m, f) \circledast (n, g) = (m \otimes n, f \times g)$$

We can approximate such multi-dimensional arrays using a Haskell existential type (specified by using the quantifier keyword `forall` before the data constructor):

```
data MultiArray a = forall i. Ix i => MA (Array i a)

instance Functor MultiArray where
    fmap f (MA a) =
        MA (array (bounds a) [(i, f e) | (i, e) <- assocs a])
```

The `unit` operation constructs a scalar, while `mult` forms the cartesian product of two arrays:

```
instance Monoidal MultiArray where
    unit = MA (array ((), ()) [((), ())])
    mult (MA xs) (MA ys) =
        MA (array ((lx, ly), (hx, hy))
            [((i, j), (x, y)) | (i, x) <- assocs xs,
                                (j, y) <- assocs ys])
          where
            (lx, hx) = bounds xs
            (ly, hy) = bounds ys
```

We could extend multi-dimensional arrays by adding a distinguished position, i.e. a cursor within the container:

$$L\,c = \exists\, m.\, K\,m \times (K\,m \to c)$$

When two arrays are combined with `mult`, their cursors are also paired to form a cursor on the product array.

Another example arises in Elliott's analysis of fusion [6], where folds are reified using a type

```
data FoldL b a = FoldL (a -> b -> a) a
```

The type constructor `FoldL` is not a functor, because its argument `a` occurs in both the domain and range of function types. Wishing to apply functorial machinery to these reified folds, Elliott introduced a related type that could be defined as a functor:

```
data WithCont z c = forall a. WC (z a) (a -> c)
```

```
instance Functor (WithCont z) where
    fmap g (WC z k) = WC z (g . k)
```

Although `FoldL` is not a functor, it nevertheless has operations similar to `unit` and `mult`. These can be described using a type class similar to `Monoidal`, but without the `Functor` superclass:

```
class Zip z where
    zunit :: z ()
    zmult :: z a -> z b -> z (a, b)
```

The above type constructor `FoldL` is an instance:

```
instance Zip (FoldL b) where
    zunit = FoldL const ()
    zmult (FoldL f1 z1) (FoldL f2 z2) =
        FoldL (\ (x,y) b -> (f1 x b, f2 y b)) (z1, z2)
```

This class is sufficient to define `WithCont` as a lax monoidal functor:

```
instance Zip z => Monoidal (WithCont z) where
    unit = WC zunit (const ())
    mult (WC t1 k1) (WC t2 k2) =
        WC (zmult t1 t2) (\ (x,y) -> (k1 x, k2 y))
```

5.1 Left Kan Extensions

We now consider the general case. Kan extensions are general constructions that have also found applications in programming. The right Kan has been used to construct generalized folds on nested types [10], to fuse types [7], and to construct a monad (the codensity monad) that can be used for program optimization [19, 8]. The lax monoidal functors discussed above are instances of the other variety, the left Kan.

Kan extensions have an elegant description at the level of functors. Given a functor $K : \mathcal{M} \to \mathcal{C}$, the left and right Kan extensions along K are defined as the left and right adjoints of the higher-order functor $(\circ K)$ that maps to each functor $\mathcal{C} \to \mathcal{A}$ to a functor $\mathcal{M} \to \mathcal{C}$ [13]. That is, the left Kan extension a functor $T : \mathcal{M} \to \mathcal{A}$ along K is a functor $L : \mathcal{C} \to \mathcal{A}$ with a universal natural transformation $\eta : T \dot{\to} L \circ K$:

$$\begin{tikzcd} \mathcal{M} \arrow[rr, "K"] \arrow[dr, "T"'] & \arrow[d, phantom, "\xRightarrow{\eta}"] & \mathcal{C} \arrow[dl, dashed, "L"] \\ & \mathcal{A} & \end{tikzcd}$$

For our purposes, it will be more convenient to use the standard pointwise construction of the left Kan extension as a coend, corresponding to existential quantification in programming languages. For convenience, we assume that the category $\mathcal{A}$ is cartesian closed, and that $\mathcal{C}$ is an $\mathcal{A}$-category [11], i.e. that the "hom-sets" of $\mathcal{C}$ are objects of $\mathcal{A}$, with identity and composition morphisms satisfying the usual laws. Using the more familiar notation $\exists$ in place of the integral sign favoured by category theorists, the left Kan extension of $T : \mathcal{M} \to \mathcal{A}$ along $K : \mathcal{M} \to \mathcal{C}$ is the functor $L : \mathcal{C} \to \mathcal{A}$ defined by

$$L\, c = \exists\, m.\, T\, m \times \mathcal{C}\,(K\, m, c)$$

The examples discussed above are instances of left Kan extensions:

- In the container example, T is the constant functor mapping to 1, and the monoidal structure on $\mathcal{M}$ has $\top = \omega$ and with $\otimes$ as minimum.
- In the example of arrays with cursors, T is identified with K.
- In the `WithCont` example, $\mathcal{M}$ is the subcategory of isomorphisms of $\mathcal{A}$. T can model any type constructor, as although type constructors (like `FoldL` above) need not be functorial, they still preserve isomorphisms.

To state that $L\, c$ is a coend is to say that there is an initial dinatural transformation $\omega : T\, m \times \mathcal{C}\,(K\, m, c) \to L\, c$. This dinaturality of ω is expressed by the equation

$$\omega\,(T\, h\, x, k) = \omega\,(x, k \circ K\, h)$$

That is, the existentially qualified type m is abstract: we can change the representation without affecting the constructed value. The natural transformation $\eta : T \dot{\to} L \circ K$ is defined as

$$\eta\, x = \omega\,(x, id)$$

Initiality of ω means that a natural transformation from L is uniquely determined by its action on terms of the internal language of the form $\omega\,(x, k)$. For example, we can define the action of L on arrows as

$$L\, f\,(\omega\,(x, k)) = \omega\,(x, f \circ k)$$

In order to make L lax monoidal, we shall assume that the functor $K : \mathcal{M} \to \mathcal{C}$ is a *colax monoidal functor*, or equivalently a lax monoidal functor $\mathcal{M}^{\mathrm{op}} \to \mathcal{C}^{\mathrm{op}}$. That is, there are natural transformations

$$\begin{aligned} s &: K\,(a \otimes_{\mathcal{M}} b) \to K\, a \otimes_{\mathcal{C}} K\, b \\ n &: K\, \top_{\mathcal{M}} \to \top_{\mathcal{C}} \end{aligned}$$

such that the following diagrams commute:

$$\begin{tikzcd} K\,(\top \otimes a) \arrow[r, "s"] \arrow[d, "K\,\lambda"'] & K\,\top \otimes K\, a \arrow[d, "n \otimes K\, a"] \\ K\, a & \top \otimes K\, a \arrow[l, "\lambda"] \end{tikzcd} \qquad \begin{tikzcd} K\,(a \otimes \top) \arrow[r, "s"] \arrow[d, "K\,\rho"'] & K\, a \otimes K\,\top \arrow[d, "K\, a \otimes n"] \\ K\, a & K\, a \otimes \top \arrow[l, "\rho"] \end{tikzcd}$$

$$\begin{tikzcd}
K\,(a \otimes (b \otimes c)) \arrow[r, "s"] \arrow[d, "K\,\alpha"'] & K\,a \otimes K\,(b \otimes c) \arrow[r, "K\,a \otimes s"] & K\,a \otimes (K\,b \otimes K\,c) \arrow[d, "\alpha"] \\
K\,((a \otimes b) \otimes c) \arrow[r, "s"'] & K\,(a \otimes b) \otimes K\,c \arrow[r, "s \otimes K\,c"'] & (K\,a \otimes K\,b) \otimes K\,c
\end{tikzcd}$$

In the special case where the monoidal structure on $\mathcal{C}$ is that of products, there is only one choice for n, namely the unique arrow $K\,\top \to 1$. Moreover in that case $s : K\,(a \otimes b) \to K\,a \times K\,b$ can be broken down into two components: $s = \langle s_1, s_2 \rangle$.

Proposition 4. *If $\mathcal{M}$ and $\mathcal{C}$ are monoidal and $\mathcal{A}$ has finite products, K is colax monoidal and T is lax monoidal, then L is lax monoidal, with*

$$u_L = \omega\,(u_T, n_K)$$
$$\omega\,(x_1, k_1) \circledast_L \omega\,(x_2, k_2) = \omega\,(x_1 \circledast_T x_2, k_1 \times k_2 \circ s_K)$$

This is a special case of Proposition 5, which we shall prove in the next section.

A degenerate example has $\mathcal{M}$ as the trivial category with one object and one morphism, so that T defines a monoid and K selects some object, with $s_i = id$. This describes computations that write output and also read an environment, but in which the output is independent of the environment:

$$L\,c = T \times (K \to c)$$

This applicative functor is a composition of two applicative functors that are also monads, but the composition is not a monad.

Another simple case arises when $\mathcal{M}$ is a cartesian category, in which case an arbitrary functor $K : \mathcal{M} \to \mathcal{C}$ can be made colax monoidal by setting $s_i = K\,\pi_i$. Thus we obtain the following Haskell version of the left Kan:[2]

```
data Lan t k c = forall m. Lan (t m) (k m -> c)

instance (Functor t, Functor k) => Functor (Lan t k) where
    fmap f (Lan x k) = Lan x (f . k)

instance (Monoidal t, Functor k) => Monoidal (Lan t k) where
    unit = Lan unit (const ())
    mult (Lan x1 k1) (Lan x2 k2) =
        Lan (mult x1 x2)
            (\ y -> (k1 (fmap fst y), k2 (fmap snd y)))
```

Although this implementation has the form of a general left Kan extension, it is limited to the Haskell category.

A richer example occurs in the modelling of behaviours of animations using applicative functors by Matlage and Gill [14]. The basic functor comprises a function over a closed interval of time, which can be modelled as pairs of times:

[2] In fact the `Functor` instance requires no assumptions about `t` and `k`, and in the `Monoidal` instance `Zip t` could replace `Monoidal t`.

```
data Interval = Between Time Time

instance Monoid Interval where
    mempty = Between inf (-inf)
    Between start1 stop1 <> Between start2 stop2 =
        Between (min start1 start2) (max stop1 stop2)
```

We would like to represent continuous behaviours by a type $\exists i. K\, i \to T$, for a functor K mapping pairs of times to closed intervals of time, with s_i mapping from larger intervals to smaller by truncation. We cannot express this directly in Haskell, which lacks dependent types, but we can approximate it with a type

```
data Behaviour a = B Interval (Time -> a)
```

provided we hide the representation and provide only an accessor function:

```
observe :: Behaviour a -> Time -> a
observe (B (Between start stop) f) t =
    f (max start (min stop t))
```

Now this type can be made monoidal with the following definitions, which preserve the abstraction:

```
instance Functor Behaviour where
    fmap f (B i g) = B i (f . g)

instance Monoidal Behaviour where
    unit = B mempty (const ())
    mult b1@(B i1 f1) b2@(B i2 f2) =
        B (i1 <> i2) (\ t -> (observe b1 t, observe b2 t))
```

The final functor used by Matlage and Gill can be obtained by adding constant behaviour using `Lift`:

```
type Active = Lift Behaviour
```

Thus a value of type `Active a` is either constant or a function of time over a given interval. A combination of such behaviours is constant only if both the arguments were.

5.2 The General Case

Our final example is a generalization of the type used by Baars, Löh and Swierstra [3] to construct parsers for permutations of phrases, which we express as

```
data Perms p a = Choice (Maybe a) [Branch p a]
data Branch p a = forall b. Branch (p b) (Perms p (b -> a))
```

This implementation is too subtle to explain in full detain here, but the `Perms` type is essentially an efficient representation of a collection of all the permutations of a set of elementary parsers (or actions, in other applications). The type in the original paper is equivalent to restricting our version of the `Perms` type to values of the forms `Choice (Just x) []` and `Choice Nothing bs`, allowing a single elementary parser to be added to the collection at a time. In contrast, the `mult` methods allows the interleaving of arbitrary collections of actions, allowing us to build them in any order.

The functor instances for these two types are straightforward:

```
instance Functor p => Functor (Perms p) where
    fmap f (Choice def bs) =
        Choice (fmap f def) (map (fmap f) bs)

instance Functor p => Functor (Branch p) where
    fmap f (Branch p perm) = Branch p (fmap (f .) perm)
```

Assuming that `p` is lax monoidal, we will construct instances for `Perms p` and `Branch p`. These types are mutually recursive, but we know that final fixed points preserve applicative functors.

We define an operator `***` as

```
(***) :: Monoidal f => f (a1 -> b1) -> f (a2 -> b2) ->
                       f ((a1,a2) -> (b1,b2))
p *** q = fmap (\ (f,g) (x,y) -> (f x, g y)) (mult p q)
```

This is an example of the construction of static arrows from applicative functors [15]. Now, assuming that `Perms p` is lax monoidal, we can construct an instance for `Branch p` as a generalized left Kan extension:

```
instance Monoidal p => Monoidal (Branch p) where
    unit = Branch unit (pure id)
    mult (Branch p1 perm1) (Branch p2 perm2) =
        Branch (mult p1 p2) (perm1 *** perm2)
```

The instance for `Perms p` is constructed from the instance for `Branch p` as a generalized semi-direct product, which builds all the interleavings of the two collections of permutations:

```
instance Monoidal p => Monoidal (Perms p) where
    unit = Choice unit []
    mult (t1@(Choice d1 bs1)) (t2@(Choice d2 bs2)) =
        Choice (mult d1 d2)
            (map (`mult` include t2) bs1 ++
             map (include t1 `mult`) bs2)
      where
        include :: Monoidal p => Perms p a -> Branch p a
        include p = Branch unit (fmap const p)
```

To encompass examples such as this, we need a generalization of the left Kan. Suppose the functor K factors through a monoidal category $\mathcal{B}$:

$$\begin{tikzcd}
& \mathcal{B} \arrow[dr, "J"] \arrow[d, phantom, "=" near start] & \\
\mathcal{M} \arrow[ur, "H"] \arrow[rr, "K"] \arrow[dr, "T"'] & & \mathcal{C} \arrow[dl, dashed, "L"] \\
& \mathcal{A} \arrow[u, phantom, "\overset{\eta}{\Rightarrow}" near start] &
\end{tikzcd}$$

We also assume a natural operator

$$\boxtimes : \mathcal{C}(J\,a, J\,b) \times \mathcal{C}(J\,c, J\,d) \to \mathcal{C}(J\,(a \otimes c), J\,(b \otimes d))$$

(corresponding to `***` above) satisfying unit and associativity laws:

$$\begin{aligned}
J\,\lambda \circ f \boxtimes \top &= f \circ J\,\lambda \\
J\,\rho \circ \top \boxtimes f &= f \circ J\,\rho \\
J\,\alpha \circ f \boxtimes (g \boxtimes h) &= (f \boxtimes g) \boxtimes h \circ J\,\alpha
\end{aligned}$$

The situation of ordinary left Kan extensions is the special case where J is the identity functor and $\boxtimes$ is $\otimes_{\mathcal{C}}$. However in general we do not require that $\boxtimes$ be a functor. The key example of a structure with such an operator is an enriched premonoidal category, or "arrow" [2, 9].

Proposition 5. *If $\mathcal{M}$ and $\mathcal{B}$ are monoidal, $\mathcal{A}$ has finite products, H is colax monoidal and T is lax monoidal, then $F = L \circ J$ is lax monoidal, with*

$$\begin{aligned}
F\,a &= \exists\,m.\,T\,m \times (J\,(H\,m) \to J\,a) \\
F\,f\,(\omega\,(x,k)) &= \omega\,(x, J\,f \circ k) \\
u_F &= \omega\,(u_T, J\,n_H) \\
\omega\,(x_1,k_1) \circledast_F \omega\,(x_2,k_2) &= \omega\,(x_1 \circledast_T x_2, k_1 \boxtimes k_2 \circ J\,s_H)
\end{aligned}$$

Instead of proving this directly, we show that the functor $G : \mathcal{M}^{\mathrm{op}} \times \mathcal{M} \times \mathcal{A}$ defined by

$$G\,(m', m, a) = T\,m \times (J\,(H\,m') \to J\,a)$$

is itself lax monoidal, and then use a general result about coends of lax monoidal functors. To see that G is lax monoidal, we note that T is lax monoidal, so we only need to show that the second component is. The left identity case is

$$\begin{aligned}
F\,\lambda \circ id \boxtimes k \circ J\,(n_H \times 1 \circ s_H) &= k \circ J\,(\lambda \circ n_H \times 1 \circ s_H) && \text{left identity of } \boxtimes \\
&= k \circ J\,(H\,\lambda) && \text{left identity of } H
\end{aligned}$$

The right identity case is similar. Associativity relies on the associativity of $\boxtimes$:

$$\begin{aligned}
& J\,\alpha \circ k_1 \boxtimes (k_2 \boxtimes k_3 \circ J\,s_H) \circ J\,s_H \\
&= J\,\alpha \circ k_1 \boxtimes (k_2 \boxtimes k_3) \circ J\,(id \times s_H \circ s_H) && \text{naturality of } \boxtimes \\
&= (k_1 \boxtimes k_2) \boxtimes k_3 \circ J\,(\alpha \circ id \times s_H \circ s_H) && \text{associativity of } \boxtimes \\
&= (k_1 \boxtimes k_2) \boxtimes k_3 \circ J\,(s_H \times id \circ s_H \circ H\,\alpha) && \text{associativity of } s_H
\end{aligned}$$

Thus it suffices to show that coends preserve lax monoidal functors, which is our final result.

Proposition 6. *Given monoidal categories $\mathcal{A}$ and $\mathcal{B}$ and a ccc $\mathcal{C}$, with a lax monoidal functor $G : \mathcal{A}^{\mathrm{op}} \times \mathcal{A} \times \mathcal{B} \to \mathcal{C}$, then the coend $F\,b = \exists\,a.\,G\,(a,a,b)$ is also lax monoidal, with*

$$\begin{aligned}
F\,b &= \exists\,a.\,G\,(a,a,b) \\
F\,f\,(\omega\,x) &= \omega\,(G\,(id,id,f)\,x) \\
u_F &= \omega\,u_G \\
\omega\,x_1 \circledast_F \omega\,x_2 &= \omega\,(x_1 \circledast_G x_2)
\end{aligned}$$

Proof. It is a standard result that a parameterized coend such as F defines a functor. Naturality of $\circledast_F$ follows from naturality of $\circledast_G$:

$$\begin{array}{ll}
F\,(f_1 \otimes f_2)\,(\omega\,x_1 \circledast_F \omega\,x_2) & \\
= F\,(f_1 \otimes f_2)\,(\omega\,(x_1 \circledast_G x_2)) & \text{definition of } \circledast_F \\
= \omega\,(G\,(id,id,f_1 \otimes f_2)\,(x_1 \circledast_G x_2)) & \text{definition of } F \\
= \omega\,(G\,(id,id,f_1)\,x_1 \circledast_G G\,(id,id,f_2)\,x_2) & \text{naturality of } \circledast_G \\
= \omega\,(G\,(id,id,f_1)\,x_1) \circledast_F \omega\,(G\,(id,id,f_2)\,x_2) & \text{definition of } \circledast_F \\
= F\,f_1\,(\omega\,x_1) \circledast_F F\,f_2\,(\omega\,x_2) & \text{definition of } F
\end{array}$$

Similarly the left identity law for F follows from the corresponding law for G:

$$\begin{array}{rll}
F\,\lambda\,(u_F \circledast_F \omega\,x) & = F\,\lambda\,(\omega\,u_G \circledast_F \omega\,x) & \text{definition of } u_F \\
& = F\,\lambda\,(\omega\,(u_G \circledast_G x)) & \text{definition of } \circledast_F \\
& = \omega\,(G\,(id,id,\lambda)\,(u_G \circledast_G x)) & \text{definition of } F \\
& = \omega\,(G\,(id,\lambda \circ \lambda^{-1},\lambda)\,(u_G \circledast_G x)) & \text{isomorphism} \\
& = \omega\,(G\,(\lambda^{-1},\lambda,\lambda)\,(u_G \circledast_G x)) & \text{dinaturality of } \omega \\
& = \omega\,x & \text{left identity of } G
\end{array}$$

The right identity case is similar.

Finally, the associativity law for $\circledast_F$ follows from the associativity of $\circledast_G$:

$$\begin{array}{ll}
F\,\alpha\,(\omega\,x \circledast_F (\omega\,y \circledast_F \omega\,z)) & \\
= F\,\alpha\,(\omega\,x \circledast_F \omega\,(y \circledast_G z)) & \text{definition of } \circledast_F \\
= F\,\alpha\,(\omega\,(x \circledast_G (y \circledast_G z))) & \text{definition of } \circledast_F \\
= \omega\,(G\,(id,id,\alpha)\,(x \circledast_G (y \circledast_G z))) & \text{definition of } F \\
= \omega\,(G\,(id,\alpha \circ \alpha^{-1},\alpha)\,(x \circledast_G (y \circledast_G z))) & \text{isomorphism} \\
= \omega\,(G\,(\alpha^{-1},\alpha,\alpha)\,(x \circledast_G (y \circledast_G z))) & \text{dinaturality of } \omega \\
= \omega\,((x \circledast_G y) \circledast_G z) & \text{associativity of } G \\
= \omega\,(x \circledast_G y) \circledast_F \omega\,z & \text{definition of } \circledast_F \\
= (\omega\,x \circledast_F \omega\,y) \circledast_F \omega\,z & \text{definition of } \circledast_F
\end{array}$$

□

As a further example, we have the coend encoding of the final fixed point $\nu\,F$ of a functor $F : \mathcal{A} \times \mathcal{B} \to \mathcal{B}$:

$$\nu\,F\,a \cong \exists\,b.\,b \times (b \to F\,(a,b))$$

which is a coend of $G\,(b', b, a) = b \times (b' \to F\,(a, b))$, and yields the same applicative functor as discussed in Section 3.

6 Conclusion

We have established a number of general constructions of lax monoidal functors, and therefore of applicative functors. In examples such as the permutation phrases of Section 5.2, we showed that by combining these constructions we could account for quite complex (and useful) applicative functors, avoiding the need for specific proofs of their laws. By breaking the functors down into simple building blocks, we have clarified their relationships, as well providing the tools to build more applications. The next stage is to examine the possible combinations, and to consider other constructions.

References

1. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew D. Gordon, editor, *FoSSaCS*, volume 2620 of *LNCS*, pages 23–38, 2003.
2. Robert Atkey. What is a categorical model of arrows? *Electronic Notes on Theoretical Computer Science*, 229(5):19–37, 2011.
3. Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Parsing permutation phrases. *Journal of Functional Programming*, 14(6):635–646, 2004.
4. Duncan Coutts. Arrows for errors: Extending the error monad, 2002. Unpublished presentation at the Summer School on Advanced Functional Programming.
5. S. Eilenberg and G.M. Kelly. Closed categories. In S. Eilenberg, D. K. Harrison, H. Röhrl, and S. MacLane, editors, *Proceedings of the Conference on Categorical Algebra*, pages 421–562. Springer, 1966.
6. Conal Elliott. Denotational design with type class morphisms. Technical Report 2009-01, LambdaPix, 2009.
7. Ralf Hinze. Type fusion. In Dusko Pavlovic and Michael Johnson, editors, *Proceedings of the Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, volume 6486 of *LNCS*, pages 92–110. Springer, 2010.
8. Ralf Hinze. Kan extensions for program optimisation, or: Art and dan explain an old trick. In Jeremy Gibbons and Pablo Nogueira, editors, *Mathematics of Program Construction*, 2012.
9. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.
10. Patricia Johann and Neil Ghani. A principled approach to programming with nested types in Haskell. *Higher-Order and Symbolic Computation*, 22(2):155–189, 2009.
11. G. M. Kelly. *Basic concepts of enriched category theory*, volume 64 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1982.
12. J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, 1986.
13. Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, New York, 1971.

14. Kevin Matlage and Andy Gill. Every animation should have a beginning, a middle, and an end: A case study of using a functor-based animation language. In Rex Page, Zoltán Horváth, and Viktória Zsók, editors, *Proceedings of the 11th International Conference on Trends in Functional Programming*, volume 6546 of *LNCS*, pages 150–165. Springer, 2011.
15. Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.
16. Eugenio Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science*, pages 14–23. IEEE Computer Society Press, 1989.
17. Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, 1990.
18. S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS*, pages 184–207. Springer, 1996.
19. Janis Voigtländer. Asymptotic improvement of functions over free monads. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction*, volume 5133 of *LNCS*, pages 388–403. Springer, 2008.
20. Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.