
This is the accepted version of the paper.

This version of the publication may differ from the final published version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/1141/

Link to published version:

Copyright: City Research Online aims to make research outputs of City, University of London available to a wider audience. Copyright and Moral Rights remain with the author(s) and/or copyright holders. URLs from City Research Online may be freely distributed and linked to.

Reuse: Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.
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 sequenc-
ing 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 pre-
generate 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 applica-
tive 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 \( \text{mult} \) of the \textbf{Monoidal} class, we obtain an operation \( \otimes : F a \times F b \to F (a \times b) \). This suggests generalizing from products to other binary type constructors, a notion known in category theory as a monoidal category.

A \textit{monoidal category} \cite{13} consists of a category \( C \), a functor \( \otimes : C \times C \to C \) and an object \( \top \) of \( C \), with coherent natural isomorphisms

\[
\begin{align*}
\lambda : \top \otimes a & \cong a \quad \text{(left identity)} \\
\rho : a \otimes \top & \cong a \quad \text{(right identity)} \\
\alpha : a \otimes (b \otimes c) & \cong (a \otimes b) \otimes c \quad \text{(associativity)}
\end{align*}
\]

A \textit{symmetric monoidal category} also has

\[
\sigma : a \otimes b \cong b \otimes a \quad \text{(symmetry)}
\]

Both products and coproducts are examples of monoidal structures, and both are also symmetric. Given a monoidal category \( \langle C, \otimes, \top, \lambda, \rho, \alpha \rangle \), the category \( C^{\text{op}} \), obtained by reversing all the morphisms of \( C \), also has a monoidal structure: \( \langle C^{\text{op}}, \otimes, \top, \lambda^{-1}, \rho^{-1}, \alpha^{-1} \rangle \). 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 \( \times : A \times A \to A \) preserve it up to isomorphism:

\[
1 \cong 1 \times 1 \\
(a_1 \times a_2) \times (b_1 \times b_2) \cong (a_1 \times b_1) \times (a_2 \times b_2)
\]

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 \textbf{Monoidal} above from products to any monoidal category.

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

\[
\begin{align*}
u : \top' & \to F \top \\
\otimes' : F a \otimes' F b & \to F (a \otimes b) \\
\rho : F a \otimes \top' & \to F a \\
\alpha : F a \otimes (b \otimes c) & \cong (F a \otimes b) \otimes F c
\end{align*}
\]

such that the following diagrams commute:
The first two diagrams state that \( u \) is the left and right identity respectively of the binary operation \( \otimes \), while the last diagram expresses the associativity of \( \otimes \).

### 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 \( C \) as consisting of a lax monoidal functor \( F : C \to C \), with a natural transformation \( p : a \to Fa \) (corresponding to the `pure` function of the `Applicative` class) satisfying \( p_\top = u \) and \( p \circ \otimes = \otimes \circ p \otimes p \), plus a weak commutativity condition:

\[
\begin{align*}
  a \otimes F b & \xrightarrow{p \otimes F b} Fa \otimes F b \xrightarrow{\otimes} F (a \otimes b) \\
  F b \otimes a & \xrightarrow{F b \otimes p} F b \otimes Fa \xrightarrow{\otimes} F (b \otimes a)
\end{align*}
\]

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 \( A \) is a ccc, any lax monoidal functor \( F : A \to A \) 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 \( \lambda \)-calculus of the category \([12]\), in which variables of type \( a \) stand for arrows of \( A(1, a) \), 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 \( p : I \to F \) with the counterpart of the above definition of `pure` for any `Monoidal` functor:

\[
p x = F (\text{const } x) u
\]
The proof of weak commutativity is then a simple calculation in the internal language:

\[
F \sigma (px \otimes y) = F \sigma (F (\text{const } x \otimes u) \otimes y) \\
= F (\sigma \circ (\text{const } x \times \text{id}) \otimes (u \otimes y)) \quad \text{definition of } p \\
= F (\sigma \circ (\text{const } x \times \text{id}) \otimes (\lambda^{-1} y)) \quad \text{naturality of } \otimes \\
= F (\text{id} \times (\text{const } x \circ \sigma) \otimes (\lambda^{-1} y)) \quad \text{naturality of } \sigma \\
= F (\text{id} \times (\text{const } x \circ \sigma \circ \lambda^{-1}) \otimes y) \quad \text{functor} \\
= F (\text{id} \times (\text{const } x \circ \rho^{-1}) \otimes y) \quad \text{symmetry} \\
= F (\text{id} \times (\text{const } x \circ \rho^{-1}) \otimes y) \quad \text{functor} \\
= F (\text{id} \times (\text{const } x) \otimes (\rho^{-1} y)) \quad \text{right identity} \\
= y \otimes F (\text{const } x) u \quad \text{naturality of } \otimes \\
= y \otimes px \quad \text{definition of } p
\]

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]:

\[
data \text{ZipList } a = \text{Nil } | \text{Cons } a \text{ (ZipList } a) \\
\]

instance Functor ZipList where
\[
fmap f \text{ Nil } = \text{Nil} \\
fmap f \text{ (Cons } x \text{ xs) } = \text{Cons } (f \: x) \text{ (fmap } f \text{ xs)}
\]

instance Monoidal ZipList where
\[
\text{unit } = \text{Cons } () \\text{ unit} \\
\text{mult } (\text{Cons } x \text{ xs) } (\text{Cons } y \text{ ys) } = \text{Cons } (x, y) \\text{ (mult } xs \text{ ys) } \\
\text{mult } _ _ = \text{Nil}
\]
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 \texttt{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 \( F : A \times B \to B \) consists of a functor \( \nu F : A \to B \) with an isomorphism \( c : F(a, \nu Fa) \cong \nu Fa \) and an unfold operator \( \{ \cdot \} \) constructing the unique morphism satisfying

\[
F(a, b) \xrightarrow{\{ f \}} \nu F a \\
F(a, b) \xrightarrow{\{ F(a, f) \}} F(a, \nu F a)
\]

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

\[

\begin{array}{c}
\top \xrightarrow{u_{\nu F} = \{ u_F \}} \nu F \top \\
\top - \xrightarrow{u_F} - - F(\top, \top) - \xrightarrow{\{ F(\top, u_{\nu F}) \}} F(\top, \nu F)
\end{array}

\]

\[
\nu F a_1 \otimes \nu F a_2 \xrightarrow{\otimes_{\nu F} = \{ \otimes_F \circ c^{-1} \otimes c^{-1} \}} \nu F (a_1 \otimes a_2) \\
\otimes c^{-1} \circ c^{-1} \\\n\otimes_F
\]

\[
\begin{array}{c}
F(a_1, \nu Fa_1) \otimes F(a_2, \nu Fa_2) \xrightarrow{c} \nu F (a_1 \otimes a_2) \\
F(a_1 \otimes a_2, \nu F (a_1 \otimes a_2)) \xrightarrow{c} F(a_1 \otimes a_2, \nu F (a_1 \otimes a_2))
\end{array}
\]

In particular, for \( F = \texttt{Maybe} \circ \times \), this construction yields a lax monoidal functor equivalent to \texttt{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 \( A \) for the moment, another way to define the final fixed point of a functor \( F : B \to B \) starts with the terminal object 1. Using with the
unique morphism \( !_F : F \rightarrow 1 \), we can define a chain of objects and morphisms:

\[
\cdots \to F^3 1 \xrightarrow{!_{F^3 1}} F^2 1 \xrightarrow{!_{F^2 1}} F 1 \xrightarrow{!_{F 1}} 1
\]

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:

\[
\cdots \to F^3 1 \xrightarrow{\nu F 1} F^2 1 \xrightarrow{\nu F 2} F 1 \xrightarrow{\nu F 1} 1
\]

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 \( \nu F \).

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

\[
\cdots \to \Phi^3 1 \xrightarrow{\nu \Phi 1} \Phi^2 1 \xrightarrow{\nu \Phi 2} \Phi 1 \xrightarrow{\nu \Phi 1} 1
\]

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

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

\[
F a \otimes F b \xrightarrow{h \otimes h} F' a \otimes F' b
\]

\[
F (a \otimes b) \xrightarrow{h} F' (a \otimes b)
\]

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

\[
\cdots \to F^3 \xrightarrow{f_3} F^2 \xrightarrow{f_2} F^1 \xrightarrow{f_1} F_0
\]
We can construct a limit in $\text{Fun}(A, B)$ (from pointwise limits in $B$):

$$
\begin{array}{c}
F \\
t_3 \\
\vdots \\
t_0 \\
F_3 f_3 \\
F_2 f_2 \\
F_1 f_1 \\
F_0 f_0 \\
\end{array}
$$

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

$$
\begin{array}{c}
F a \otimes F b \xrightarrow{t_i \otimes t_i} F_i a \otimes F_i b \\
\uparrow 1 \quad \downarrow \otimes \varepsilon_i \\
F (a \otimes b) \xrightarrow{t_i} F_i (a \otimes b)
\end{array}
$$

These equations imply that $u$ and $\otimes$ are mediating morphisms to the limits in $B$, and thus uniquely define them. It remains to show that $\otimes$ is a natural transformation, and that $u$ and $\otimes$ 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

$$
\begin{array}{c}
\begin{array}{c}
\top \otimes F a \\
\uparrow \top \otimes t_i \\
\top \otimes F_i a \\
\uparrow \top \otimes \lambda \\
F_i a \\
\uparrow F_i \lambda \\
F a \\
\end{array} \\
\begin{array}{c}
u \otimes F a \\
\uparrow \otimes \varepsilon_i \\
F_i \top \otimes F_i a \\
\uparrow \otimes \varepsilon_i \\
F_i (\top \otimes a) \\
\uparrow t_i \\
F (\top \otimes a)
\end{array}
\end{array}
$$

The central panel is the left identity law for $F_i$, while the four surrounding panels follow from the definitions of $u$ and $\otimes$ and the naturality of $\lambda$ 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 $B$ is complete, then so is $\text{Mon}(A, B)$.

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 $\Phi$ on lax
monoidal functors is a uniquely determined extension of the final fixed point of $\Phi$ on ordinary functors. For example, ZipList is the final fixed point of the higher-order functor $\Phi$ defined by $\Phi Z a = \text{Maybe} (a \times Z a)$.

### 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:

```haskell
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]:

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

That is, in a computation $\text{mult } e1 e2$, 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 $L \cong \text{Lift} (I \times L)$ 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:

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

```haskell
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 \( L \cong \text{Lift}(I \times (F \circ L)) \) for any lax monoidal \( F \), defining forms of long zip for various kinds of tree.

The \text{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 \text{pure}. We can generalize:

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

**Proof.** Define a functor \( F \) by

\[
F a = \sum_{j \in J} C_j (\Delta_j a)
\]

\[
F f (C_j x) = C_j (f x)
\]

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

\[
u = C_\bot \ u_\bot
\]

\[
C_j a \otimes C_k b = C_{j \sqcup k} (\Delta_{j \sqcup k} a, \Delta_{k \sqcup j} b)
\]

Naturality of \( \otimes \) and the identity and associativity laws follow from simple calculations. \( \square \)

For example, in the case of \text{Lift}, \( J \) 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 Tokens, recording the symbol type and its text:

\[
\text{type Token} = (\text{Symbol}, \text{String})
\]
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:

```haskell
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:

```haskell
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:

```haskell
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:

```haskell
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:

```haskell
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

\[
\text{symbol :: Symbol -> Phrase String} \\
\text{symbol s = Phrase Nothing (singleton s advance)}
\]

Alternatives are easily built:

\[
(\langle|\rangle) :: \text{Phrase a -> Phrase a -> Phrase a} \\
\text{Phrase e1 t1 <|> Phrase e2 t2 =} \\
\text{Phrase (e1 `mplus` e2) (t1 `union` t2)}
\]

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 \text{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 \(\alpha\) also determines whether the initial symbols of \(\alpha\beta\) include those of \(\beta\) in addition to those of \(\alpha\):

\[
\text{instance Monoidal Phrase where} \\
\text{unit = Phrase unit empty} \\
\text{mult (Phrase e1 t1) (~p2@(Phrase e2 t2)) =} \\
\text{Phrase (mult e1 e2) (union t1' t2')} \\
\text{where} \\
\text{t1'} = \text{fmap ('mult' parser p2) t1} \\
\text{t2'} = \text{maybe empty (\ \_ x -> fmap (fmap ((,) x)) t2) e1}
\]

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 \text{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 \text{Phrase} type is a pair, and the first components are combined using the lax monoidal operations on \text{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 \(\langle F, \odot, u \rangle : \langle A, \odot, \top \rangle \to \langle B, \times, 1 \rangle\)
– a functor $G : A \to B$ with a natural family of monoids $\oplus : Ga \times Ga \to Ga$ and $\emptyset : 1 \to Ga$.
– an operation $\times : Ga \times (Fb \times Gb) \to G(a \otimes b)$ distributing over $G$:

$$\emptyset \times q = \emptyset$$  \hspace{1cm} (1)
$$x \oplus y \times q = (x \times q) \oplus (y \times q)$$  \hspace{1cm} (2)

– an operation $\ltimes : (Fa \times Ga) \times Gb \to G(a \otimes b)$ distributing over $G$:

$$p \ltimes \emptyset = \emptyset$$  \hspace{1cm} (3)
$$p \ltimes (x \oplus y) = (p \ltimes x) \oplus (p \ltimes y)$$  \hspace{1cm} (4)

distributed over $G$ also satisfying

$$p \ltimes y \ltimes r = p \ltimes (y \ltimes r)$$  \hspace{1cm} (5)

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

$$H a = Fa \times Ga$$
$$u_H = (u, \emptyset)$$
$$(a, x) \otimes_H (b, y) = (a \oplus b, (x \times (b, y)) \oplus ((a, x) \times y))$$

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

$$u_H \ltimes z = z$$  \hspace{1cm} (6)
$$(p \oplus_H q) \ltimes z = p \ltimes (q \ltimes z)$$  \hspace{1cm} (7)
$$x \ltimes u_H = x$$  \hspace{1cm} (8)
$$x \ltimes (q \oplus_H r) = (x \times q) \ltimes r$$  \hspace{1cm} (9)

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

$$u_H \oplus_H (b, y) = (1 \oplus b, (\emptyset \times (b, y)) \oplus (u_H \times y))$$ \hspace{1cm} definition of $\oplus_H$, $u_H$
$$= (1 \oplus b, \emptyset \oplus y)$$ \hspace{1cm} equations (1) and (6)
$$= (b, y)$$ \hspace{1cm} monoid laws

$$(a, x) \oplus_H u_H = (a \oplus 1, (x \times u_H) \oplus ((a, x) \times \emptyset))$$ \hspace{1cm} definition of $\oplus_H$, $u_H$
$$= (a \oplus 1, x \oplus \emptyset)$$ \hspace{1cm} equations (8) and (3)
$$= (a, x)$$ \hspace{1cm} monoid laws
Finally, we must show that $\odot_H$ is associative:

\[
\begin{align*}
((a, x) \odot_H (b, y)) \odot_H (c, z) &= (a \odot b \odot c, (((x \times (b, y)) \odot ((a, x) \times y)) \times (c, z)) \odot \\
&= (a \odot b \odot c, ((x \times (b, y)) \times (c, z)) \odot (((a, x) \times y) \times (c, z)) \odot \\
&= (a \odot b \odot c, (x \times ((b, y) \odot_H (c, z))) \odot \\
&= (a \odot b \odot c, (x \times ((b, y) \odot_H (c, z))) \odot \\
&= (a, x) \odot_H ((b, y) \odot_H (c, z))
\end{align*}
\]

definition of $\odot_H$

\[
\begin{align*}
= (a \odot b \odot c, ((x \times (b, y)) \times (c, z)) \odot ((a, x) \times (b, y) \times z))
&= (a \odot b \odot c, (x \times ((b, y) \times (c, z))) \odot ((a, x) \times ((b, y) \times z)))
&= (a, x) \odot_H ((b, y) \odot_H (c, z))
&= (a, x) \odot_H ((b, y) \odot_H (c, z))
\end{align*}
\]

equation (2)

\[
\begin{align*}
&= (a \odot b \odot c, (x \times ((b, y) \times (c, z))) \odot ((a, x) \times ((b, y) \times z)))
&= (a, x) \odot_H ((b, y) \odot_H (c, z))
&= (a, x) \odot_H ((b, y) \odot_H (c, z))
\end{align*}
\]

equation (4)

definition of $\odot_H$
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 $u$ constructing a scalar and $\otimes$ being cartesian product:

$$u = (\top, \text{const } ())$$

$$(m, f) \otimes (n, g) = (m \otimes n, f \times g)$$

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

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

```haskell
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:

```haskell
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

```haskell
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:

```haskell
data WithCont z c = forall a. WC (z a) (a -> c)
```
Constructing Applicative Functors

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 $\eta : T \to L \circ K$:

\[
\begin{array}{c}
\mathcal{M} \\
\downarrow T \\
\mathcal{C} \\
\downarrow K \\
\mathcal{A} \\
\end{array}
\xrightarrow{L} \begin{array}{c}
\mathcal{C} \\
\downarrow K \\
\mathcal{A} \\
\end{array}
\]

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 $A$ is cartesian closed, and that $C$ is an $A$-category [11], i.e. that the “hom-sets” of $C$ are objects of $A$, 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 $T : M \to A$ along $K : M \to C$ is the functor $L : C \to A$ defined by

$$Lc = \exists m. T m \times C (K m, c)$$

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 $M$ has $\top = \omega$ and with $\otimes$ as minimum.
- In the example of arrays with cursors, $T$ is identified with $K$.
- In the WithCont example, $M$ is the subcategory of isomorphisms of $A$. $T$ can model any type constructor, as although type constructors (like FoldL above) need not be functorial, they still preserve isomorphisms.

To state that $Lc$ is a coend is to say that there is an initial dinatural transformation $\omega : T m \times C (K m, c) \to Lc$. This dinaturality of $\omega$ 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 $\eta : T \to L \circ K$ is defined as

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

Initiality of $\omega$ 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

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

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

$$s : K (a \otimes_M b) \to K a \otimes_C K b$$
$$n : K \top_M \to \top_C$$

such that the following diagrams commute:

\[
\begin{array}{ccc}
K (\top \otimes a) & \xrightarrow{s} & K \top \otimes K a \\
K \lambda & \downarrow & \downarrow n \otimes K a \\
K a & \xleftarrow{\lambda} & \top \otimes K a
\end{array}
\quad
\begin{array}{ccc}
K (a \otimes \top) & \xrightarrow{s} & K a \otimes K \top \\
K \rho & \downarrow & \downarrow K a \otimes n \\
K a & \xleftarrow{\rho} & K a \otimes \top
\end{array}
\]
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: \( s = (s_1, s_2) \).

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

\[
\begin{align*}
\omega(x_1, k_1) \otimes_L \omega(x_2, k_2) &= \omega(x_1 \otimes_T x_2, k_1 \times k_2 \circ s_K) \\
\end{align*}
\]

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:

\[ Lc = T \times (K \to c) \]

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:

\[
\begin{align*}
data \text{Lan } t \ k \ c = \text{forall } m. \text{Lan } (t \ m) \ (k \ m \to c) \\
\text{instance } (\text{Functor } t, \text{Functor } k) \Rightarrow \text{Functor } (\text{Lan } t \ k) \text{ where} \\
fmap f (\text{Lan } x \ k) &= \text{Lan } x \ (f \circ k) \\
\text{instance } (\text{Monoidal } t, \text{Functor } k) \Rightarrow \text{Monoidal } (\text{Lan } t \ k) \text{ where} \\
\text{unit} &= \text{Lan } \text{unit} \ (\text{const } () ) \\
\text{mult} (\text{Lan } x1 \ k1) \ (\text{Lan } x2 \ k2) &= \\
&\text{Lan} \ (\text{mult} \ x1 \ x2) \\
&\ (\ y \to (k1 \ (\text{fmap} \ \text{fst} \ y), k2 \ (\text{fmap} \ \text{snd} \ y))) \\
\end{align*}
\]

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:

\[2\text{In fact the Functor instance requires no assumptions about } t \text{ and } k, \text{ and in the Monoidal instance Zip } t \text{ could replace Monoidal } t.\]
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 \( \text{Lift} \):

type Active = Lift Behaviour

Thus a value of type \( \text{Active} a \) 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 detail 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:

```haskell
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

```haskell
(***) :: 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:

```haskell
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:

```haskell
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 $B$:

![Diagram](image)

We also assume a natural operator

$$\boxtimes : C(Ja, Jb) \times C(Jc, Jd) \to C(J(a \otimes c), J(b \otimes d))$$

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

$$J \lambda \circ f \boxtimes T = f \circ J \lambda$$

$$J \rho \circ T \boxtimes f = f \circ J \rho$$

$$J \alpha \circ f \boxtimes (g \boxtimes h) = (f \boxtimes g) \boxtimes h \circ J \alpha$$

The situation of ordinary left Kan extensions is the special case where $J$ is the identity functor and $\boxtimes$ is $\otimes_C$. 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 $M$ and $B$ are monoidal, $A$ has finite products, $H$ is colax monoidal and $T$ is lax monoidal, then $F = L \circ J$ is lax monoidal, with

$$Fa = \exists m. Tm \times (J(Hm) \to Ja)$$

$$Ff(\omega(x,k)) = \omega(x, Jf \circ k)$$

$$u_F = \omega(u_T, J n_H)$$

$$\omega(x_1, k_1) \otimes_F \omega(x_2, k_2) = \omega(x_1 \otimes_T x_2, k_1 \boxtimes k_2 \circ J s_H)$$

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

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

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

$$F \lambda \circ id \boxtimes k \circ J(n_H \times 1 \circ s_H) = k \circ J(\lambda \circ n_H \times 1 \circ s_H) \quad \text{left identity of } \boxtimes$$

$$= k \circ J(H \lambda) \quad \text{left identity of } H$$

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

$$J \alpha \circ k_1 \boxtimes (k_2 \boxtimes k_3 \circ J s_H) \circ J s_H$$

$$= J \alpha \circ k_1 \boxtimes (k_2 \boxtimes k_3) \circ J(id \times s_H \circ s_H) \quad \text{naturality of } \boxtimes$$

$$= (k_1 \boxtimes k_2) \boxtimes k_3 \circ J(\alpha \circ id \times s_H \circ s_H) \quad \text{associativity of } \boxtimes$$

$$= (k_1 \boxtimes k_2) \boxtimes k_3 \circ J(s_H \times id \circ s_H \circ H \alpha) \quad \text{associativity of } s_H$$
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 $G : \mathcal{A}^{\text{op}} \times \mathcal{A} \times \mathcal{B} \to \mathcal{C}$, then the coend $F b = \exists a. G(a, a, b)$ is also lax monoidal, with

$$F b = \exists a. G(a, a, b)$$

$$F f(\omega x) = \omega(G(id, id, f)x)$$

$$u_F = \omega u_G$$

$$\omega x_1 \otimes_F \omega x_2 = \omega(x_1 \otimes_G x_2)$$

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

$$F(f_1 \otimes f_2)(\omega x_1 \otimes_F \omega x_2)$$

$$= F(f_1 \otimes f_2)(\omega(x_1 \otimes_G x_2))$$

$$= \omega(G(id, id, f_1 \otimes f_2)(x_1 \otimes_G x_2))$$

$$= \omega(G(id, id, f_1)x_1 \otimes_G G(id, id, f_2)x_2)$$

$$= \omega(G(id, id, f_1)x_1) \otimes_F \omega(G(id, id, f_2)x_2)$$

$$= F f_1(\omega x_1) \otimes_F F f_2(\omega x_2)$$

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

$$F \lambda(u_F \otimes_F \omega x) = F \lambda(\omega u_G \otimes_F \omega)$$

$$= F \lambda(\omega u_G \otimes_F x)$$

$$= \omega(G(id, id, \lambda)(u_G \otimes_G x))$$

$$= \omega(G(id, \lambda \circ \lambda^{-1}, \lambda)(u_G \otimes_G x))$$

$$= \omega(G(\lambda^{-1}, \lambda, \lambda)(u_G \otimes_G x))$$

$$= \omega x$$

The right identity case is similar.

Finally, the associativity law for $\otimes_F$ follows from the associativity of $\otimes_G$:

$$F(\alpha (\omega x \otimes_F (\omega y \otimes_F \omega z)))$$

$$= F(\alpha (\omega x \otimes_F \omega(y \otimes_G z)))$$

$$= F(\alpha (\omega(x \otimes_G (y \otimes_G z))))$$

$$= \omega(G(id, id, \alpha)(x \otimes_G (y \otimes_G z)))$$

$$= \omega(G(id, \alpha \circ \alpha^{-1}, \alpha)(x \otimes_G (y \otimes_G z)))$$

$$= \omega(G(\alpha^{-1}, \alpha, \alpha)(x \otimes_G (y \otimes_G z)))$$

$$= \omega((x \otimes_G y) \otimes_G z)$$

$$= \omega(x \otimes_G y \otimes_F \omega z)$$

$$= (\omega x \otimes_F \omega y) \otimes_F \omega z$$

$$= \omega y \otimes_F \omega z$$

$$= \omega y$$

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


