Notions of Computation as Monoids

EXEQUIEL RIVAS MAURO JASKELIOFF
Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas
CONICET, Argentina FCEIA, Universidad Nacional de Rosario, Argentina

Abstract

There are different notions of computation, the most popular being monads, applicative functors, and arrows. In this article we show that these three notions can be seen as monoids in a monoidal category. We demonstrate that at this level of abstraction one can obtain useful results which can be instantiated to the different notions of computation. In particular, we show how free constructions and Cayley representations for monoids translate into useful constructions for monads, applicative functors, and arrows. Moreover, the uniform presentation of all three notions helps in the analysis of the relation between them.

1 Introduction

When constructing a semantic model of a system or when structuring computer code, there are several notions of computation that one might consider. Monads (Moggi, 1989; Moggi, 1991) are the most popular notion, but other notions, such as arrows (Hughes, 2000) and, more recently, applicative functors (McBride & Paterson, 2008) have been gaining widespread acceptance.

Each of these notions of computation has particular characteristics that makes them more suitable for some tasks than for others. Nevertheless, there is much to be gained from unifying all three different notions under a single conceptual framework.

In this article we show how all three of these notions of computation can be cast as a monoid in a monoidal category. Monads are known to be monoids in a monoidal category of endofunctors (Mac Lane, 1971; Barr & Wells, 1985). Moreover, strong monads are monoids in a monoidal category of strong endofunctors. Arrows have been recently shown to be strong monoids in a monoidal category of profunctors by Jacobs et al. (2009). Applicative functors, on the other hand, are usually presented as lax monoidal functors with a compatible strength (McBride & Paterson, 2008; Jaskelioff & Rypacek, 2012; Paterson, 2012) However, in the category-theory community, it is known that lax monoidal functors are monoids with respect to the Day convolution, and hence applicative functors are also monoids in a monoidal category of endofunctors using the Day convolution as a tensor (Day, 1973).

Therefore, we unify the analysis of three different notions of computation, namely monads, applicative functors, and arrows, by looking at them as monoids in a monoidal category. In particular, we make explicit the relation between applicative functors and monoids with respect to the Day convolution, and we simplify the characterisation of arrows. Unlike

the approach to arrows of Jacobs et al. (2009), where the operation first is added on top of the monoid structure, we obtain that operation from the monoidal structure of the underlying category. Furthermore, we show that at the level of abstraction of monoidal categories one can obtain useful results, such as free constructions and Cayley representations.

Free constructions are often used in programming in order to represent abstract syntax trees. For instance, free constructions are used to define deep embeddings of domain-specific languages. Traditionally, one uses a free monad to represent abstract syntax trees, with the bind operation (Kleisli extension) acting as a form of simultaneous substitution. However, in certain cases, the free applicative functor is a better fit (Capriotti & Kaposi, 201
The free arrow, on the other hand, has been less explored and we know of no publication that has an implementation of it.

The Cayley representation theorem states that every group is isomorphic to a group of permutations (Cayley, 1854). Hence, one can work with a concrete group of permutations instead of working with an abstract group. The representation theorem does not really use the inverse operation of groups so one can generalise the representation to monoids, yielding a Cayley representation theorem for monoids (Jacobson, 2009).

In functional programming, the Cayley theorem appears as an optimisation by change of representation. We identify two known optimisations, namely difference lists (Hughes, 198 and the codensity monad transformation (Voigtländer, 2008; Hutton *et al.*, 2010) as being essentially the same, since both are instances of the general Cayley representation of monoids in a monoidal category. Moreover, we obtain similar transformations for applicative functors and arrows by analysing their Cayley representations.

Given the three notions of computation, one may ask what is the relation between them. Lindley et al. (2011) address this question by studying the equational theories induced by each calculus. Since the different notions are monoids in a monoidal category, a categorical approach could be to ask about the relation between the corresponding categories of monoids. However, another consequence of having a unified view is that we can ask a deeper question instead and analyse the relation between the different monoidal categories that support them. Then, we obtain the relation between their monoids as a corollary.

Concretely, the article makes the following contributions:

- We present a unified view of monads, applicative functors, and arrows as monoids in a monoidal category. Although the results are known in other communities, the case of the applicative functors as monoids seems to have been overlooked in the functional programming community.
- We show how the Cayley representation of monoids unifies two different known optimisations, namely difference lists and the codensity monad transformation. The similarity between these two optimisations has been noticed before, but now we make the relation precise and demonstrate that they are two instances of the *same*

change of representation.

- We apply the characterisation of applicative functors as monoids to obtain a free construction and a Cayley representation for applicative functors. In this way, we clarify the construction of free applicative functors as explained by Capriotti and Kaposi (2014). The Cayley representation for applicative functors is entirely new.

- We clarify the view of arrows as monoids by introducing the strength in the monoidal category. In previous approaches, the strength was added to the monoids, while in this article we consider a category with strong profunctors. Our approach leads to a new categorical model of arrows and to the first formulation of free arrows.
- We analyse the relation between the monoidal categories that give rise to monads, applicative functors, and arrows, by constructing monoidal functors between them.

The rest of the article is structured as follows. In Section 1.1 we introduce the Cayley representation for ordinary monoids. In Section 2, we introduce monoidal categories, monoids, free monoids and the Cayley representation for monoids in a monoidal category. In Section 3, we instantiate these constructions to a category of endofunctors, with composition as a tensor and obtain monads, free monads, and the Cayley representation for monads. In Section 5, we do the same for applicative functors. Before that, we introduce in Section 4 the notions of ends and coends needed to define and work with the Day convolution. In Section 6, we work in a category of Profunctors to obtain pre-arrows, their free constructions, and their Cayley representations. In section 7, we turn to arrows, analyse the relation between arrows and pre-arrows, and construct free arrows and an arrow representation. Finally, in section 8, we analyse the relation between the different monoidal categories considered in the previous sections, and conclude in Section 9.

The article is aimed at functional programmers with knowledge of basic category theory concepts, such as categories, functors, and adjunctions. We provide an introduction to more advanced concepts, such as monoidal categories and ends.

In frames like the one surrounding this paragraph, we include Haskell implementations of several of the categorical concepts of the article. The idea is not to formalise these concepts in Haskell, but rather to show how the category theory informs and guides the implementation.

1.1 Cayley representation for monoids

We start by stating the Cayley representation theorem for ordinary monoids, i.e. monoids in the category of sets and functions Set. A monoid is a triple $(M, \oplus, e)$ of a set M, a binary operation $\oplus : M \times M \to M$ which is associative ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$), and an element $e \in M$ which is a left and right identity with respect to the binary operation (i.e. $e \oplus a = a = a \oplus e$.) Because of the obvious monoid $(\mathbb{N}, \cdot, 1)$, the element e and the binary operation $\oplus$ are often called the *unit* and *multiplication* of the monoid.

For every set M we may construct the *monoid of endomorphisms* $(M \to M, \circ, \mathsf{id})$, where $\circ$ is function composition and id is the identity function.

Up to an isomorphism, M is a *sub-monoid* of a monoid $(M', \oplus', e')$ if there is an injection $i : M \hookrightarrow M'$ such that $i(e) = e'$ and $i(a \oplus b) = i(a) \oplus' i(b)$ for some $\oplus$ and e. This makes

$(M, \oplus, e)$ a monoid and i a *monoid morphism*.

Theorem 1.1 (Cayley representation for (Set*) monoids)*
Every monoid $(M, \oplus, e)$ is a sub-monoid of the monoid of endomorphisms on M.

Proof We construct an injection $\mathsf{rep} : M \to (M \to M)$ by currying the binary operation $\oplus$.

$$\mathsf{rep}(m) = \lambda m'.\, m \oplus m'.$$

The function rep is a monoid morphism:

$$\begin{aligned}
\mathsf{rep}(e) &= \lambda m'.\, e \oplus m' \\
&= \mathsf{id} \\
\mathsf{rep}(a \oplus b) &= \lambda m'.\, (a \oplus b) \oplus m' \\
&= \lambda m'.\, a \oplus (b \oplus m') \\
&= (\lambda m.\, a \oplus m) \circ (\lambda n.\, b \oplus n) \\
&= \mathsf{rep}(a) \circ \mathsf{rep}(b)
\end{aligned}$$

Moreover, rep is an injection, since we have a function $\mathsf{abs} : (M \to M) \to M$ given by

$$\mathsf{abs}(k) = k(e)$$

and, $\mathsf{abs}(\mathsf{rep}(m)) = (\lambda m'.\, m \oplus m')\, e = m \oplus e = m$.

□

When M lifts to a group (i.e. it has a compatible inverse operation), then the monoid of endomorphisms on M lifts to the traditional Cayley representation of a group M.

How can we use this theorem in Haskell? Lists are monoids $([a], +\!\!+, [\,])$ so we may apply Theorem 1.1. Let us define a type synonym for the monoid of endomorphisms:

$$\mathbf{type}\ \mathsf{EList}\ a = [a] \to [a]$$

The functions rep and abs are

$$\begin{aligned}
&\mathsf{rep} \quad :: [a] \to \mathsf{Elist}\ a \\
&\mathsf{rep}\ xs = (xs +\!\!+) \\
&\mathsf{abs} \quad :: \mathsf{Elist}\ a \to [a] \\
&\mathsf{abs}\ xs = xs\ [\,]
\end{aligned}$$

By the theorem above, we have that $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$. The type $\mathsf{Elist}\ a$ is no other than difference lists! (Hughes, 1986). Concatenation for standard lists is slow, as it is linear on the first argument. A well known solution is to use a different representation of lists: the so-called “difference lists” or “Hughes’ lists”, in which lists are represented by endofunctions of lists. In difference lists, concatenation is just function composition, and the empty list is the identity function. Hence we can perform efficient concatenations on difference lists, and when we are done we can get back standard lists by applying the empty list.

2 Monoidal Categories

The ordinary notion of monoid in the category Set of sets and functions is too restrictive, so we are interested in generalising monoids to other categories. In order to express a monoid a category should have a notion of

1. a pairing operation for expressing the type of the multiplication,
2. and a type for expressing the unit.

In Set (in fact, in any category with finite products), we may define a binary operation on X as a function $X \times X \to X$, and the unit as a morphism $1 \to X$. However, a given category $\mathscr{C}$ may not have finite products, or we may be interested in other monoidal structure of $\mathscr{C}$, so we will be more general and abstract the product by a $\otimes$ operation called a *tensor*, and the unit 1 by an object I of $\mathscr{C}$. Categories with a tensor $\otimes$ and unit I have the necessary structure for supporting an abstract notion of monoid and are known as *monoidal categories*.

Definition 2.1 (*Monoidal Category*)
A *monoidal category* is a tuple $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$, consisting of

- a category $\mathscr{C}$,
- a bifunctor $\otimes \colon \mathscr{C} \times \mathscr{C} \to \mathscr{C}$,

- an object I of $\mathscr{C}$,
- natural isomorphisms $\alpha_{A,B,C} : A \otimes (B \otimes C) \to (A \otimes B) \otimes C$, $\lambda_A : I \otimes A \to A$, and $\rho_A : A \otimes I \to A$ such that $\lambda_I = \rho_I$ and the following diagrams commute.

$$\begin{tikzcd}
A \otimes (B \otimes (C \otimes D)) \arrow[r, "\alpha"] \arrow[d, "\mathsf{id} \otimes \alpha"'] & (A \otimes B) \otimes (C \otimes D) \arrow[r, "\alpha"] & ((A \otimes B) \otimes C) \otimes D \\
A \otimes ((B \otimes C) \otimes D) \arrow[rr, "\alpha"'] & & (A \otimes (B \otimes C)) \otimes D \arrow[u, "\alpha \otimes \mathsf{id}"']
\end{tikzcd}$$

$$\begin{tikzcd}
A \otimes (I \otimes B) \arrow[rr, "\alpha"] \arrow[dr, "\mathsf{id} \otimes \lambda"'] & & (A \otimes I) \otimes B \arrow[dl, "\rho \otimes \mathsf{id}"] \\
& A \otimes B &
\end{tikzcd}$$

A monoidal category is said to be *strict* when the natural isomorphisms α, λ and ρ are identities. Note that in a strict monoidal category the diagrams necessarily commute.

A *symmetric* monoidal category, is a monoidal category with an additional natural isomorphism $\gamma_{A,B} : A \otimes B \to B \otimes A$ subject to some coherence conditions.

The idea of currying a function can be generalised to a monoidal category with the following notion of exponential.

Definition 2.2 (*Exponential*)
Let A be an object of a monoidal category $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$. An *exponential* $-^A$ is the right

adjoint to $- \otimes A$. That is, the exponential to A is characterised by an isomorphism

$$\lfloor - \rfloor : \mathscr{C}(X \otimes A, B) \cong \mathscr{C}(X, B^A) : \lceil - \rceil$$

natural in X and B. We call the counit of the adjunction $\mathsf{ev}_B = \lceil \mathsf{id}_{B^A} \rceil : B^A \otimes A \to B$ the *evaluation morphism* of the exponential. When the exponential to A exists, we say that A is *an exponent*. When the exponential exists for every object we say that the monoidal category *has exponentials* or that it is a *right-closed* monoidal category.

The next lemmata will be used in the proofs that follow.

Lemma 2.3
Let A, B, C be objects of a monoidal category $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$, such that the exponential $-^A$ exists. For every $f : B \otimes A \to C$, we have

$$\mathsf{ev} \circ (\lfloor f \rfloor \otimes \mathsf{id}) = f$$

Lemma 2.4
Let A, B, C, D be objects of a monoidal category $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$, such that the exponential $-^C$ exists. For every $f : B \otimes C \to D$ and $g : A \to B$, we have

$$\lfloor f \circ (g \otimes \mathsf{id}) \rfloor = \lfloor f \rfloor \circ g$$

2.1 Monoids in Monoidal Categories

With the definition of monoidal category in place we may define a monoid in such a category.

Definition 2.5 (*Monoid*)
A *monoid* in a monoidal category $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$ is a tuple (M, m, e) where $M \in \mathscr{C}$ and m and e are morphisms in $\mathscr{C}$

$$I \xrightarrow{e} M \xleftarrow{m} M \otimes M$$

such that the following diagrams commute.

$$\begin{tikzcd}
(M \otimes M) \otimes M \arrow[rr, "m \otimes \mathrm{id}"] & & M \otimes M \arrow[d, "m"] \\
M \otimes (M \otimes M) \arrow[u, "\alpha"] \arrow[r, "\mathrm{id} \otimes m"'] & M \otimes M \arrow[r, "m"'] & M
\end{tikzcd}
\qquad
\begin{tikzcd}
M \otimes M \arrow[dr, "m"] & M \otimes I \arrow[l, "\mathrm{id} \otimes e"'] \arrow[d, "\rho"] \\
I \otimes M \arrow[u, "e \otimes \mathrm{id}"] \arrow[r, "\lambda"'] & M
\end{tikzcd}$$

A *monoid homomorphism* is an arrow $M_1 \xrightarrow{f} M_2$ in $\mathscr{C}$ such that the diagrams

$$\begin{tikzcd}
& M_1 \arrow[d, "f"] & M_1 \otimes M_1 \arrow[l, "m_1"'] \arrow[d, "f \otimes f"] \\
I \arrow[ur, "e_1"] \arrow[dr, "e_2"'] & & \\
& M_2 & M_2 \otimes M_2 \arrow[l, "m_2"]
\end{tikzcd}$$

commute.

In the same manner that A^* (the words on A) is the free monoid on a set A, we can define the notion of free monoid in terms of monoidal categories.

Definition 2.6 (*Free Monoid*)
Let $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$ be a monoidal category. The *free monoid* on an object X in $\mathscr{C}$ is a monoid (F, m_F, e_F) together with a morphism $\mathsf{ins} : X \to F$ such that for any monoid (G, m_G, e_G) and any morphism $f : X \to G$, there exists a unique monoid homomorphism

$\mathsf{free}\ f : F \to G$ that makes the following diagram commute.

$$\begin{tikzcd} X \arrow[r, "\mathsf{ins}"] \arrow[dr, "f"'] & F \arrow[d, dashed, "\mathsf{free}\ f"] \\ & G \end{tikzcd}$$

The morphism ins is called the *insertion of generators* into the free monoid.

Monoids in a monoidal category $\mathscr{C}$ and monoid homomorphisms form the category $\mathsf{Mon}(\mathscr{C})$. When the left-adjoint $(-)^*$ to the forgetful functor $U : \mathsf{Mon}(\mathscr{C}) \to \mathscr{C}$ exists, it maps an object X to the free monoid on X. There are several conditions that guarantee the existence of free monoids (Dubuc, 1974; Kelly, 1980; Lack, 2010). Of particular importance to us is the following:

Proposition 2.7
Let $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$ be a monoidal category with exponentials. If $\mathscr{C}$ has binary coproducts, and for each $A \in \mathscr{C}$ the initial algebra for the endofunctor $I + A \otimes -$ exists, then the

monoid A^* exists and its carrier is given by $\mu X\,.\,I + A \otimes X$.

Proof A multiplication on A^* has the form $m : A^* \otimes A^* \to A^*$. By definition 2.2, it is equivalent to define a morphism $A^* \to A^{*A^*}$, and then use $\lceil - \rceil$ to get m. Exploiting the universal property of initial algebras, we define such morphism by providing and algebra $I + A \otimes A^{*A^*} \to A^{*A^*}$, given by[1] $[\lfloor \lambda_{A^*} \rfloor, \lfloor \delta \circ \mathsf{inr} \circ (\mathsf{id} \otimes \mathsf{ev}) \circ \alpha \rfloor]$ where $\delta : I + A \otimes A^* \cong A^*$ is the initial algebra structure over A^*.

The monoid structure on A^* is then

$$
\begin{aligned}
e &= \delta \circ \mathsf{inl} \\
m &= \lceil (\!| [\lfloor \lambda_{A^*} \rfloor, \lfloor \delta \circ \mathsf{inr} \circ (\mathsf{id} \otimes \mathsf{ev}) \circ \alpha \rfloor] |\!) \rceil
\end{aligned}
$$

where the banana brackets $(\!| - |\!)$ denote the universal morphism from an initial algebra (Meijer *et*

The insertion of generators and the universal morphism from the free monoid to the monoid (G, m_G, e_G) for $f : A \to G$ are:

$$
\begin{aligned}
\mathsf{ins} &= \delta \circ \mathsf{inr} \circ (\mathsf{id} \otimes e) \circ \rho^{-1} \\
\mathsf{free}\ f &= (\!| [e_G, m_G \circ (f \otimes \mathsf{id})] |\!)
\end{aligned}
$$

$\square$

It is well known that the free monoid over a set A is the set of lists of A. Unsurprisingly, when implementing in Haskell the formula of proposition 2.7 for the case of Set monoids, we obtain lists.

$$\textbf{data}\ \mathsf{List}\ a = \mathsf{Nil} \mid \mathsf{Cons}\ (a, \mathsf{List}\ a)$$

[1] For given $f : A \to C$ and $g : B \to C$, then $[f,g]$ is the unique morphism $A + B \to C$ such that $[f,g] \circ \mathsf{inl} = f$ and $[f,g] \circ \mathsf{inr} = g$.

Definition 2.8 (*Sub-monoid*)
Given a monoid (M,e,m) in $\mathscr{C}$, and a monic $i : M' \hookrightarrow M$ in $\mathscr{C}$, such that for some (unique) maps e' and m', we have a commuting diagram

$$\begin{tikzcd}
I \arrow[r, "e"] \arrow[d, equal] & M & M \otimes M \arrow[l, "m"'] \\
I \arrow[r, "e'"'] & M' \arrow[u, "i"'] & M' \otimes M' \arrow[l, "m'"] \arrow[u, "i\otimes i"']
\end{tikzcd}$$

then (M',e',m') is a monoid, called the *sub-monoid* of M induced by the monic i, and i is a monoid monomorphism from M' to M.

2.2 Cayley Representation of a Monoid

Every exponent in a monoidal category induces a monoid of endomorphisms:

Definition 2.9 (*Monoid of endomorphisms*)
Let $(\mathscr{C},\otimes,I,\alpha,\lambda,\rho)$ be a monoidal category. The *monoid of endomorphisms* on any exponent $A \in \mathscr{C}$ is given by the diagram

$$I \xrightarrow{i_A} A^A \xleftarrow{c_A} A^A \otimes A^A$$

where

$$i_A = \lfloor\, I \otimes A \xrightarrow{\lambda_A} A \,\rfloor$$

$$c_A = \lfloor\, (A^A \otimes A^A) \otimes A \xrightarrow{\alpha^{-1}} A^A \otimes (A^A \otimes A) \xrightarrow{\mathsf{id}_{A^A} \otimes \mathsf{ev}_A} A^A \otimes A \xrightarrow{\mathsf{ev}_A} A \,\rfloor$$

The Cayley representation theorem tell us that every monoid (M, m, e) in a monoidal category is a sub-monoid of a monoid of endomorphisms whenever M is an exponent.

Theorem 2.10 (*Cayley*)
Let $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$ be a monoidal category, and let (M, e, m) be a monoid in $\mathscr{C}$. If M is an exponent then (M, e, m) is a sub-monoid of the monoid of endomorphisms (M^M, c_M, i_M), as witnessed by the monic $\mathsf{rep} = \lfloor m \rfloor : M \hookrightarrow M^M$. Moreover, $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}_M$ where abs is given by

$$\mathsf{abs} = M^M \xrightarrow{\rho^{-1}_{M^M}} M^M \otimes I \xrightarrow{\mathsf{id}_{M^M} \otimes e} M^M \otimes M \xrightarrow{\mathsf{ev}} M$$

Proof The morphism $\mathsf{rep} : \; M \xrightarrow{\lfloor m \rfloor} M^M$ is a monoid morphism.

$$\begin{array}{cl} & \lfloor m \rfloor \circ e_M \\ = & \{ \text{ lemma 2.4 } \} \\ & \lfloor m \circ (e_M \otimes \mathsf{id}) \rfloor \\ = & \{ \text{ monoid } \} \end{array}$$

$$
\begin{aligned}
& \lfloor \lambda_M \rfloor \\
= \quad & \{ \text{ definition of } i_M \ \} \\
& i_M
\end{aligned}
$$

$$
\begin{aligned}
& c_M \circ (\lfloor m \rfloor \otimes \lfloor m \rfloor) \\
= & \quad \{ \text{ definition } c_M \} \\
& \lfloor \mathsf{ev} \circ (\mathsf{id}_{M^M} \otimes \mathsf{ev}) \circ \alpha^{-1} \rfloor \circ \lfloor m \rfloor \otimes \lfloor m \rfloor \\
= & \quad \{ \text{ lemma 2.4 } \} \\
& \lfloor \mathsf{ev} \circ (\mathsf{id}_{M^M} \otimes \mathsf{ev}) \circ \alpha^{-1} \circ ((\lfloor m \rfloor \otimes \lfloor m \rfloor) \otimes \mathsf{id}_M) \rfloor \\
= & \quad \{ \text{ naturality } \alpha^{-1} \} \\
& \lfloor \mathsf{ev} \circ (\mathsf{id}_{M^M} \otimes \mathsf{ev}) \circ (\lfloor m \rfloor \otimes (\lfloor m \rfloor \otimes \mathsf{id}_M)) \circ \alpha^{-1} \rfloor \\
= & \quad \{ \text{ lemma 2.3 } \} \\
& \lfloor \mathsf{ev} \circ (\lfloor m \rfloor \otimes m) \circ \alpha^{-1} \rfloor \\
= & \quad \{ \text{ lemma 2.3 } \} \\
& \lfloor m \circ (\mathsf{id}_M \otimes m) \circ \alpha^{-1} \rfloor \\
= & \quad \{ \text{ monoid } \} \\
& \lfloor m \circ (m \otimes \mathsf{id}_M) \rfloor \\
= & \quad \{ \text{ lemma 2.4 } \} \\
& \lfloor m \rfloor \circ m
\end{aligned}
$$

We have $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}_M$, and hence rep is monic.

$$
\begin{aligned}
& \mathsf{abs} \circ \mathsf{rep} \\
= & \quad \{ \text{ definition of } \mathsf{abs} \text{ and } \mathsf{rep} \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{ev} \circ (\mathsf{id}_{M^M} \otimes e_M) \circ \rho_{M^M}^{-1} \circ \lfloor m \rfloor \\
= \quad & \{\ \text{naturality of } \rho^{-1}\ \} \\
& \mathsf{ev} \circ (\mathsf{id}_{M^M} \otimes e_M) \circ (\lfloor m \rfloor \otimes \mathsf{id}) \circ \rho_M^{-1} \\
= \quad & \{\ \text{tensor}\ \} \\
& \mathsf{ev} \circ (\lfloor m \rfloor \otimes \mathsf{id}_M) \circ (\mathsf{id}_M \otimes e_M) \circ \rho_M^{-1} \\
= \quad & \{\ \text{lemma 2.3}\ \} \\
& m \circ (\mathsf{id}_M \otimes e_M) \circ \rho_M^{-1} \\
= \quad & \{\ \text{monoid}\ \} \\
& \rho_M \circ \rho_M^{-1} \\
= \quad & \{\ \text{isomorphism}\ \} \\
& \mathsf{id}_M
\end{aligned}
$$

□

The Cayley theorem for sets (Theorem 1.1) is an instance of this theorem for the category Set. As new monoidal categories are introduced in the following sections, more instances will be presented.

3 Monads as Monoids

Consider the (strict) monoidal category $\mathsf{End}_\circ = ([\mathsf{Set}, \mathsf{Set}], \circ, \mathsf{Id})$ of endofunctors, functor composition and identity functor. A monoid in this category consists of

- An endofunctor M,
- a natural transformation $m : M \circ M \to M$,
- and a unit $e : \mathsf{Id} \to M$; such that the diagrams

$$
\begin{tikzcd}
(M\circ M)\circ M \arrow[rr, "mM"] \arrow[d, equal] & & M\circ M \arrow[d, "m"] \\
M\circ (M\circ M) \arrow[r, "Mm"'] & M\circ M \arrow[r, "m"'] & M
\end{tikzcd}
\qquad
\begin{tikzcd}
M\circ M \arrow[dr, "m"] & M\circ \mathsf{Id} \arrow[l, "Me"'] \arrow[d, equal] \\
\mathsf{Id}\circ M \arrow[u, "eM"] \arrow[r, equal] & M
\end{tikzcd}
$$

commute.

Hence, a monoid in $\mathsf{End}_\circ$ is none other than a monad. Hence the following often-heard slogan: *A monad is a monoid in a category of endofunctors*.

The corresponding implementation in Haskell is the following type class:

```
class Functor m => Triple m where
  η    :: a -> m a
  join :: m (m a) -> m a
```

where we have called the type class Triple in order to not clash with standard nomenclature which uses the name Monad for the presentation of a monad through its Kleisli extension:

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

The latter has the advantage of not needing a Functor instance and of being easier to use when programming. The two presentations are equivalent, as one can be obtained from the other by taking $\eta = \mathsf{return}$, $\mathsf{join} = (\gg\!= \mathsf{id})$, and $(\gg\!= f) = \mathsf{join} \circ \mathsf{fmap}\, f$.

3.1 *Exponential for Monads*

Finding an exponential in this category means finding a functor $(-)^F$, such that we have an isomorphism natural in G and H

$$\mathsf{Nat}(H \circ F, G) \quad \cong \quad \mathsf{Nat}(H, G^F) \tag{3.1}$$

A useful technique for finding exponentials such as G^F in a functor category is to turn to the famous Yoneda lemma.

Theorem 3.1 (*Yoneda*)

Let $\mathscr{C}$ be a locally small category. Then, there is an isomorphism

$$F\,X \quad \cong \quad \mathsf{Nat}(\mathsf{Hom}_{\mathscr{C}}(X, -), F)$$

natural in object $X : \mathscr{C}$ and functor $F : \mathscr{C} \to \mathsf{Set}$. That is, the set $F\,X$ is naturally isomorphic to the set of natural transformations between the functor $\mathsf{Hom}_{\mathscr{C}}(X, -)$ and the functor F.

Now, if an exponential G^F exists in the strict monoidal category $([\mathsf{Set}, \mathsf{Set}], \circ, \mathsf{Id})$, then the following must hold:

$$\begin{aligned} G^F X &\cong \mathsf{Nat}(\mathsf{Hom}(X, -), G^F) \\ &\cong \mathsf{Nat}(\mathsf{Hom}(X, -) \circ F, G) \end{aligned}$$

where the first isomorphism is by Yoneda, and the second is by equation 3.1. Therefore, whenever the expression $\mathsf{Nat}(\mathsf{Hom}(X, -) \circ F, G)$ makes sense, it can be taken to be the *definition* of the exponential G^F. Making sense in this case means that the collection of natural transformations between $\mathsf{Hom}(X, -) \circ F$ and G is a set. The collection $\mathsf{Nat}(F, G)$ of natural transformation between two Set endofunctors F and G is not always a set, i.e. $[\mathsf{Set}, \mathsf{Set}]$ is not locally small. However, a sufficient condition for $\mathsf{Nat}(F, G)$ to be a set is for F to be small. Small functors (Day & Lack, 2007) are endofunctors on Set which are a left Kan extension along the inclusion from a small subcategory. Therefore, every small functor F is an exponent in $\mathsf{End}_\circ$, with the exponential $(-)^F$ given by

$$G^F X = \mathsf{Nat}(\mathsf{Hom}(X, -) \circ F, G)$$

Remark 3.2
The functor $(-)^F$ is a right adjoint to the functor $(- \circ F)$ and is known as the right Kan extension along F.

The Haskell implementation of the exponential with respect to functor composition is the following.

$$\textbf{data}\ \mathsf{Exp}\ f\ g\ x = \mathsf{Exp}\ (\forall y.\ (x \to f\ y) \to g\ y)$$

The components of isomorphism 3.1 are:

$$\begin{aligned}
&\varphi \quad :: \mathsf{Functor}\ h \Rightarrow (\forall x.\ h\ (f\ x) \to g\ x) \to h\ y \to \mathsf{Exp}\ f\ g\ y\\
&\varphi \quad t\ y = \mathsf{Exp}\ (\lambda k \to t\ (\mathsf{fmap}\ k\ y))\\
&\varphi^{-1} \quad :: (\forall y.\ h\ y \to \mathsf{Exp}\ f\ g\ y) \to h\ (f\ x) \to g\ x\\
&\varphi^{-1}\ t\ x = \textbf{let}\ \mathsf{Exp}\ g = t\ x\ \textbf{in}\ g\ \mathsf{id}
\end{aligned}$$

3.2 *Free Monads*

By restricting $\mathsf{End}_\circ$ to finitary functors we obtain the locally small, right-closed monoidal category $\mathsf{End}_{\circ F}$ (Kelly & Power, 1993). In this category, we may apply proposition 2.7 and obtain the usual formula for the free monad of an endofunctor F.

$$F^*X \cong X + F(F^*X)$$

The formula above can be readily implemented by the datatype:

$$
\begin{aligned}
\mathbf{data}\ \mathsf{Free}_{\circ}\ f\ x &= \mathsf{Ret}\ x \\
&\mid\ \mathsf{Con}\ (f\ (\mathsf{Free}_{\circ}\ f\ x))
\end{aligned}
$$

with monad instance:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Monad}\ (\mathsf{Free}_{\circ}\ f)\ \textbf{where} \\
\quad \mathsf{return}\ x \qquad\quad = \mathsf{Ret}\ x \\
\quad (\mathsf{Ret}\ x)\ \ \gg\!= f = f\ x \\
\quad (\mathsf{Con}\ m) \gg\!= f = \mathsf{Con}\ (\mathsf{fmap}\ (\gg\!= f)\ m)
\end{array}
$$

There is no need to check that the instance satisfies the monad laws since the definition is derived from Proposition 2.7.

The insertion of generators and the universal morphism from the free monad are:

$$
\begin{array}{l}
\mathsf{ins} :: \mathsf{Functor}\ f \Rightarrow f \stackrel{\bullet}{\to} \mathsf{Free}_{\circ}\ f \\
\mathsf{ins}\ x = \mathsf{Con}\ (\mathsf{fmap}\ \mathsf{Ret}\ x)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{free} :: (\mathsf{Functor}\ f, \mathsf{Monad}\ m) \Rightarrow (f \circ m \stackrel{\bullet}{\to} m) \to (\mathsf{Free}_{\circ}\ f \stackrel{\bullet}{\to} m) \\
\mathsf{free}\ f\ \ (\mathsf{Ret}\ x) = \mathsf{return}\ x \\
\mathsf{free}\ f\ \ (\mathsf{Con}\ t) = f\ \ (\mathsf{fmap}\ (\mathsf{free}\ k)\ t)
\end{array}
$$

where the $\circ$ in the type signature of free is functor composition.

3.3 *Cayley Representation of Monads*

For an exponent F, we may apply theorem 2.10 and obtain the monad of endomorphisms F^F, the monad morphism rep, and the natural transformation abs. The monad F^F corresponding to the monoid of endomorphisms on a functor F receives the name of *codensity monad* on F (Mac Lane, 1971).

The codensity monad is implemented by the following datatype.

$$\mathbf{type}\ \mathsf{Rep}\ f = \mathsf{Exp}\ f\ f$$

$$\begin{array}{l}\mathbf{instance}\ \mathsf{Monad}\ (\mathsf{Rep}\ f)\ \mathbf{where} \\ \quad \mathsf{return}\ x \qquad\quad = \mathsf{Exp}\ (\lambda h \to h\ x) \\ \quad (\mathsf{Exp}\ m) \gg\!= f = \mathsf{Exp}\ (\lambda h \to m\ (\lambda x \to \mathbf{let}\ \mathsf{Exp}\ t = f\ x\ \mathbf{in}\ t\ h))\end{array}$$

There is no need to check that the instance satisfies the monad laws since the definition is derived directly from the general definition of the monoid of endomorphisms.

The morphisms converting from a monad m to $\mathsf{Rep}\ m$ and back are the following.

$$\begin{array}{l}\mathsf{rep} :: \mathsf{Monad}\ m \Rightarrow m\ x \to \mathsf{Rep}\ m\ x \\ \mathsf{rep}\ m = \mathsf{Exp}\ (m \gg\!=)\end{array}$$

$$\begin{array}{l}\mathsf{abs} :: \mathsf{Monad}\ m \Rightarrow \mathsf{Rep}\ m\ x \to m\ x \\ \mathsf{abs}\ (\mathsf{Rep}\ m) = m\ \mathsf{return}\end{array}$$

By Theorem 2.10, we know that $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$, and that abs is a monad morphism. Hence, we may change the representation of monadic computations on m, and perform computations on $\mathsf{Rep}\ m$. This change of representation is exactly the optimisation introduced by Voigländer (2008) and shown correct by Hutton et al. (2010).

Therefore, difference lists and the codensity transformation are both instances of the same change of representation: the Cayley representation.

4 Ends and Coends

In this section we review the concept of a special type of limit called *end*, and its dual *coend*. These concepts will be used in the development of the next sections.

A limit for a functor $F : \mathscr{C} \to \mathscr{D}$ is universal cone to F, where a cone is a natural transformation $\Delta_D \to F$ from the functor which is constantly D, for a $D \in \mathscr{D}$, into the functor F.

When working with functors with mixed variance $F : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$, rather than considering its limit, one is usually interested in its end. And end for a functor $F : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ is a universal *wedge* to F, where a wedge is a *dinatural* transformation $\Delta_D \to F$ from the functor which is constantly D for a $D \in \mathscr{D}$, into the functor F.

We make this precise with the following definitions:

Definition 4.1
A *dinatural transformation* $\alpha : F \to G$ between two functors $F, G : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ is a family of morphisms of the form $\alpha_C : F(C,C) \to G(C,C)$, one morphism for each $C \in \mathscr{C}$,

such that for every morphism $f : C \to C'$ the following diagram commutes.

$$\begin{tikzcd}
 & F(C,C) \arrow[r, "\alpha_C"] & G(C,C) \arrow[dr, "G(\mathsf{id},f)"] & \\
F(C',C) \arrow[ur, "F(f,\mathsf{id})"] \arrow[dr, "F(\mathsf{id},f)"'] & & & G(C,C') \\
 & F(C',C') \arrow[r, "\alpha_{C'}"'] & G(C',C') \arrow[ur, "G(f,\mathsf{id})"] &
\end{tikzcd}$$

An important difference between natural transformations and dinatural transformations is that the latter can not be composed in the general case.

Definition 4.2

A *wedge* from an object $V \in \mathscr{D}$ to a functor $F : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ is a dinatural transformation from the constant functor $\Delta_V : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ to F. Explicitly, an object V together with a family of morphisms $\alpha_X : V \to F(X,X)$ such that for each $f : C \to C'$ the following diagram commutes.

$$F(C,C)$$

$$
\begin{tikzcd}
 & {} \arrow[dr, "F(\mathrm{id},f)"] & \\
V \arrow[ur, "\alpha_C"] \arrow[dr, "\alpha_{C'}"'] & & F(C,C') \\
 & F(C',C') \arrow[ur, "F(f,\mathrm{id})"'] &
\end{tikzcd}
$$

In the same way a limit is a final cone, we define an *end* as a final wedge.

Definition 4.3
The *end* of a functor $F : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ is a final wedge for F. Explicitly, it is an object $V \in \mathscr{D}$ together with a family of morphisms $\omega_C : V \to F(C,C)$ such that the diagram

$$
\begin{tikzcd}
 & F(C,C) \arrow[dr, "F(\mathrm{id},f)"] & \\
V \arrow[ur, "\omega_C"] \arrow[dr, "\omega_{C'}"'] & & F(C,C') \\
 & F(C',C') \arrow[ur, "F(f,\mathrm{id})"'] &
\end{tikzcd}
$$

commutes for each $f : C \to C'$, and such that for every wedge from $V' \in \mathscr{D}$, given by a family of morphisms $\gamma_c : V' \to F(C,C)$ such that $F(\mathsf{id}, f) \circ \gamma_c = F(f, \mathsf{id}) \circ \gamma_c'$ for every $f : C \to C'$, there exists a unique morphism $! : V' \to V$ such that the following diagram commutes.

$$
\begin{tikzcd}
 & & F(C,C) \arrow[dr, "F(\mathsf{id},f)"] & \\
V' \arrow[r, dashed, "!"] \arrow[urr, "\gamma_C"] \arrow[drr, "\gamma_{C'}"'] & V \arrow[ur, "\omega_C"'] \arrow[dr, "\omega_{C'}"] & & F(C,C') \\
 & & F(C',C') \arrow[ur, "F(f,\mathsf{id})"'] &
\end{tikzcd}
$$

The object V is usually denoted by $\int_A F(A,A)$ and referred to as “the end of F”.

One nice feature of ends is that it leads to a natural implementation of categorical concepts in Haskell by replacing the end by a universal quantifier. For example, the natural transformations between two functors F and G can be expressed as an end

$$\int_X FX \to GX$$

(where by $X \to Y$ we note the exponential on Set) and implemented as follows.

type $f \stackrel{\bullet}{\to} g = \forall x.\, f\; x \to g\; x$

Ends can be seen as a generalised product, but cut down by a relation of dinaturality. Following this view, a morphism to an end is defined by a dinatural family of morphisms:

$$\frac{\langle \phi \rangle : Y \to \int_A F(A,A)}{\phi_X : Y \to F(X,X),\ \text{dinatural in } X}$$

Proposition 4.4
By the universal property of ends, $\langle \phi \rangle$ is the unique morphism such that $\omega_X \circ \langle \phi \rangle = \phi_X$.

Given a dinatural transformation $\alpha : \Delta_Y \to F$, and a morphism $h : Z \to Y$, the family of morphisms defined by $(\alpha \circ h)_C = \alpha_C \circ h$ is dinatural in C. Using the universal property of ends, we obtain the following proposition:

Proposition 4.5
Let $\phi_X : Y \to F(X,X)$ be a family of morphisms dinatural in X, and a let $h : Z \to Y$. Then $\langle \phi \circ h \rangle = \langle \phi \rangle \circ h$.

When defining a family of morphisms, abstracting over the varying object comes in handy. We will use Λ as a binder for objects variables. For example, $(\alpha \circ h)_C = \alpha_C \circ h$ can be defined directly as $\alpha \circ h = \Lambda C.\, \alpha_C \circ h$,

There are dual notions of wedges and ends, namely cowedges and coends. We briefly summarise their definitions.

Definition 4.6
A *cowedge* from F is an object V together with a dinatural transformation $\alpha : F \to \Delta_V$.

Definition 4.7
A *coend* is an initial cowedge. Explicitly, a coend of F is an object V together with a family of morphisms $\iota_C : F(C,C) \to V$ such that $\iota_X \circ F(f, \mathsf{id}) = \iota_Y \circ F(\mathsf{id}, f)$, which is universal with respect to this property: for every object V' and family of morphisms $\gamma_C : F(C,C) \to V'$ such that $\gamma_X \circ F(f, \mathsf{id}) = \gamma_Y \circ F(\mathsf{id}, f)$, then there exists a unique morphism $f : V \to V'$ such that $\gamma_X = f \circ \iota_X$.

A coend can be seen as a generalised coproduct, quotiented by an equivalence relation.

If $\phi_X : F(X,X) \to Y$ is a family of morphisms dinatural in X, then the morphism from $\int^A F(A,A)$ to Y given by the universal property of coends is denoted as $[\phi]$:

$$\frac{[\phi] : \int_A F(A,A) \to Y}{\phi_X : F(X,X) \to Y, \text{ dinatural in } X}$$

> In the same way an end can be implemented as a universal quantifier, a coend can be implemented as an existential quantifier, as supported by modern implementations of Haskell.

We finish this section by presenting the Yoneda lemma in the language of ends and coends. Focusing on functors $\mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathsf{Set}$, with $\mathscr{C}$ a small category, we can form the set of dinatural transformations between two such functors. The fact that such dinatural transformations form a set is justified by the next proposition.

Proposition 4.8

Let $F, G : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathsf{Set}$, with $\mathscr{C}$ a small category. Dinatural transformations from F to G are in a one-to-one correspondence with global elements of $\int_A F(A,A) \to G(A,A)$. If we denote the dinatural transformations between F and G by $\mathsf{Dinat}(F,G)$, we obtain:

$$\mathsf{Dinat}(F,G) \quad \cong \quad \int_A F(A,A) \to G(A,A)$$

In particular, when F and G are functors in one covariant variable (i.e. dummy in their contravariant variable), $\mathsf{Dinat}(F,G)$ reduces to $\mathsf{Nat}(F,G)$ and we have

$$\mathsf{Nat}(F,G) \quad \cong \quad \int_A F(A) \to G(A)$$

The Yoneda lemma in its end and coend form (Asada & Hasuo, 2010; Asada, 2010) is usually expressed as:

$$FX \quad \cong \quad \int_Y \mathsf{Hom}_{\mathscr{C}}(X,Y) \to FY \quad \cong \quad \int^Y FY \times \mathsf{Hom}_{\mathscr{C}}(Y,X)$$

We can interpret the end form of Yoneda lemma as an isomorphism between types $f\ x$ and $\forall y.\ (x \to y) \to f\ y$ whenever f is a $\mathsf{Functor}$.

The components of the isomorphism are implemented as

$$
\begin{aligned}
&\varphi \quad :: \mathsf{Functor}\ f \Rightarrow f\ x \to (\forall y.\ (x \to y) \to f\ y) \\
&\varphi\ v = \lambda f \to \mathsf{fmap}\ f\ v
\end{aligned}
$$

$$
\begin{aligned}
&\varphi^{-1} \quad :: (\forall y.\ (x \to y) \to f\ y) \to f\ x \\
&\varphi^{-1}\ g = g\ \mathsf{id}
\end{aligned}
$$

Similarly, its coend form (also known as “coYoneda lemma”) is expressed by

$$
\begin{aligned}
&\psi && :: \mathsf{Functor}\ f \Rightarrow f\ x \to (\exists\, y.\ (f\ y, y \to x)) \\
&\psi\ v && = (v, \mathsf{id})
\end{aligned}
$$

$$
\begin{aligned}
&\psi^{-1} && :: \mathsf{Functor}\ f \Rightarrow (\exists\, y.\ (f\ y, y \to x)) \to f\ x \\
&\psi^{-1}\ (x, g) && = \mathsf{fmap}\ g\ x
\end{aligned}
$$

5 Applicatives as Monoids

Similarly to monads, applicative functors (McBride & Paterson, 2008) are a class of functors used to write certain effectful computations. These functors come with an operation that allows evaluation of functions inside the functor. Compared to monads, applicative functors are a strictly weaker notion: every monad is an applicative functor (see Section 8.3), but there are applicative functors which are not monads. The main difference between monads and applicative functors is that the latter does not allow effects to depend on previous values, i.e. they are fixed beforehand.

In Haskell, these functors are represented by the following type class:

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (⊛) :: f (a → b) → f a → f b
```

Since their introduction, applicative functors have been characterised categorically as *strong lax monoidal functors* (McBride & Paterson, 2008). We explain the notions of *strong*

functor and *lax monoidal functor* separately. In simple words, a lax monoidal functor is a functor preserving the monoidal structure of the categories involved.

Definition 5.1
A *lax monoidal functor* $F : \mathscr{C} \to \mathscr{D}$ is a functor between the underlying categories of two monoidal categories $(\mathscr{C}, \otimes, I, \alpha_{\mathscr{C}}, \lambda_{\mathscr{C}}, \rho_{\mathscr{C}})$ and $(\mathscr{D}, \oplus, J, \alpha_{\mathscr{D}}, \lambda_{\mathscr{D}}, \rho_{\mathscr{D}})$ together with a natural transformation

$$\phi_{A,B} : F(A) \oplus F(B) \to F(A \otimes B)$$

and a morphism

$$\eta : J \to F(I)$$

such that the following diagrams commute.

$$\begin{tikzcd}
FA \oplus (FB \oplus FC) \arrow[r, "\mathsf{id}\oplus\phi_{B,C}"'] \arrow[d, "\alpha_{\mathscr{D}}"'] & FA \oplus F(B \otimes C) \arrow[r, "\phi_{A,(B\otimes C)}"'] & F(A \otimes (B \otimes C)) \arrow[d, "F\alpha_{\mathscr{C}}"] \\
(FA \oplus FB) \oplus FC \arrow[r, "\phi_{A,B}\oplus\mathsf{id}"] & F(A \otimes B) \oplus FC \arrow[r, "\phi_{(A\otimes B),C}"] & F((A \otimes B) \otimes C)
\end{tikzcd}$$

$$\begin{tikzcd}
FA \oplus J \arrow[r, "\mathsf{id}\oplus\eta"] \arrow[d, "\rho_{\mathscr{D}}"'] & FA \oplus FI \arrow[d, "\phi_{A,I}"] \\
FA & F(A \otimes I) \arrow[l, "F\rho_{\mathscr{C}}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
J \oplus FA \arrow[r, "\eta\oplus\mathsf{id}"] \arrow[d, "\lambda_{\mathscr{D}}"'] & FI \oplus FA \arrow[d, "\phi_{I,A}"] \\
FA & F(I \otimes A) \arrow[l, "F\rho_{\mathscr{C}}"]
\end{tikzcd}$$

A *monoidal functor* is a lax monoidal functor in which ϕ and η are isomorphisms.

Definition 5.2

An endofunctor $F : \mathscr{C} \to \mathscr{C}$ is *strong* when it comes equipped with a natural transformation

$$\mathsf{st}_{A,B} : A \otimes FB \to F(A \otimes B)$$

called a *strength* such that following diagrams commute.

$$
\begin{tikzcd}
1 \otimes F(A) \arrow[d, "\mathsf{st}"'] \arrow[dr, "\rho"] & \\
F(1 \otimes A) \arrow[r, "F\rho"'] & F(A)
\end{tikzcd}
\qquad
\begin{tikzcd}
A \otimes (B \otimes FC) \arrow[r, "A\otimes\mathsf{st}"] \arrow[d, "\alpha"] & A \otimes F(B \otimes C) \arrow[r, "\mathsf{st}"] & F(A \otimes (B \otimes C)) \arrow[d, "F(\alpha)"] \\
(A \otimes B) \otimes FC \arrow[rr, "\mathsf{st}"'] & & F((A \otimes B) \otimes C)
\end{tikzcd}
$$

All endofunctors on the (cartesian) monoidal category Set come with a unique strength, so all functors in $[\mathsf{Set},\mathsf{Set}]$ are strong. Now, a *strong lax monoidal functor* is simply a lax monoidal functor which is also a strong functor and in which the strength interacts coherently with the monoidal structure. In our setting of Set endofunctors we get this coherence for free.

The categorical characterisation of applicative functors as strong lax monoidal functors gives rise to an alternative (but equivalent) implementation of applicative functors:

```
class Functor f ⇒ Monoidal f where
  unit :: f ()
  (⋆)  :: f a → f b → f (a,b)
```

We saw how monads are monoids in a particular monoidal category. Applicative functors can be shown to be monoids too. Interestingly, they are monoids in the same category as monads: *An applicative functor is a monoid in a category of endofunctors*. However, it is not the same monoidal category, as this time we must consider a different notion of tensor. For monads we used composition; for applicative functors we use a tensor called *Day convolution* (Day, 1970). Given a cartesian closed category $\mathscr{C}$, two functors $F, G : \mathscr{C} \to \mathscr{C}$, and an object B in $\mathscr{C}$, the Day convolution $(F \star G)B$ is a new object in $\mathscr{C}$ defined as:

$$(F \star G)B = \int^{C,D} FC \times GD \times B^{(C \times D)}$$

The coend does not necessarily exist for arbitrary Set endofunctors, but it is guaranteed to exist for small functors (Day & Lack, 2007). Unless otherwise stated, in the remainder of the section we will work with $[\mathsf{Set}, \mathsf{Set}]_S$ the category of small Set endofunctors.

Applying theorem IX.7.1 of Mac Lane (1971), it can be shown that $F \star G$ is not only a mapping between objects, but also a mapping between morphisms, and that it respects the functor laws. Furthermore, given natural transformations $\alpha : F \to G$ and $\beta : H \to I$, we can form a natural transformation $\alpha \star \beta : F \star H \to G \star I$. This makes the Day convolution a bifunctor $- \star - : [\mathsf{Set}, \mathsf{Set}]_S \times [\mathsf{Set}, \mathsf{Set}]_S \to [\mathsf{Set}, \mathsf{Set}]_S$.

The coend in the definition of the Day convolution can be implemented by an existential datatype. In the definition below, done in GADT style, the type variables c and d are existentially quantified.

```
data (f ⋆ g) b where
  Day :: f c → g d → ((c, d) → b) → (f ⋆ g) b
instance (Functor f, Functor g) ⇒ Functor (f ⋆ g) where
  fmap f (Day x y g) = Day x y (f ∘ g)
```

The Day convolution is a bifunctor with the following mapping of morphisms:

```
bimap :: (f ⩒→ h) → (g ⩒→ i) → (f ⋆ g ⩒→ h ⋆ i)
bimap m₁ m₂ (Day x y f) = Day (m₁ x) (m₂ y) f
```

The following proposition allows us to write morphisms from the image of the Day convolution to another object.

Proposition 5.3
There is a one-to-one correspondence defining morphisms going out of a Day convolution

$$[\mathscr{C},\mathscr{C}](F \star G, H) \quad \overset{\vartheta}{\cong} \quad [\mathscr{C} \times \mathscr{C}, \mathscr{C}](\times \circ (F \times G), H \circ \times) \tag{5.1}$$

which is natural in F, G, and H. Here, $\times : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$ is the functor which takes an object (A,B) of the product category into a product of objects $A \times B$.

Remark 5.4 (*Day convolution as a left Kan extension*)
In view of the last proposition, the Day convolution $F \star G$ is the left Kan extension of $\times \circ (F \times G)$ along $\times$.

> The proposition above shows an equivalence between the types $(f \star g) \stackrel{\bullet}{\to} h$ and $\forall a\, b.\ (f\ a, g\ b) \to h\ (a,b)$.
>
> $\vartheta :: (f \star g \stackrel{\bullet}{\to} h) \to (f\ a, g\ b) \to h\ (a,b)$
> $\vartheta\ f\ (x,y) = f\ (\mathsf{Day}\ x\ y\ \mathsf{id})$
> $\vartheta^{-1} :: \mathsf{Functor}\ h \Rightarrow (\forall a\, b.\ (f\ a, g\ b) \to h\ (a,b)) \to (f \star g \stackrel{\bullet}{\to} h)$
> $\vartheta^{-1}\ g\ (\mathsf{Day}\ x\ y\ f) = \mathsf{fmap}\ f\ (g\ (x,y))$

In contrast to the composition tensor, the Day convolution is not strict. Moreover, the Day convolution is symmetric, which together with appropriate natural transformations α, λ and ρ make $\mathsf{End}_\star = ([\mathsf{Set},\mathsf{Set}]_S, \star, \mathsf{Id}, \alpha, \lambda, \rho, \gamma)$ a symmetric monoidal category (Day, 1970).

Here we present the natural transformations of the monoidal category $\mathsf{End}_\star$. In order to do that we first implement the identity functor.

```
data Id a = Id a deriving Functor

λ :: Functor f ⇒ f ⩒→ Id ⋆ f
λ x = Day (Id ()) x snd

ρ :: Functor f ⇒ f ⩒→ f ⋆ Id
ρ x = Day x (Id ()) fst

α :: (f ⋆ g) ⋆ h ⩒→ f ⋆ (g ⋆ h)
α (Day (Day x y f) z g) = Day x (Day y z f1) f2
  where f1 = λ(d,b) → ((λc → f (c,d)),b)
        f2 = λ(c,(h,b)) → g (h c,b)

γ :: (f ⋆ g) ⩒→ (g ⋆ f)
γ (Day x y f) = Day y x (f ∘ swap)
  where swap (x,y) = (y,x)
```

We leave the definition of the inverses as an exercise.

Remark 5.5 (*Alternative presentations of the Day convolution*)
In our setting of Set functors, the Day convolution has different alternative representations:

$$(F \star G)B \quad \cong \quad \int^{A} FA \times G(B^{A}) \quad \cong \quad \int^{A} F(B^{A}) \times GA \tag{5.2}$$

5.1 Monoids in $\mathsf{End}_\star$

A monoid in $\mathsf{End}_\star$ amounts to:

- An endofunctor F,
- a natural transformation $m : F \star F \to F$,
- and a unit $e : \mathsf{Id} \to F$; such that the following diagrams commute.

$$
\begin{tikzcd}
(F \star F) \star F \arrow[rr, "m \star F"] & & F \star F \arrow[d, "m"] \\
F \star (F \star F) \arrow[u, "\alpha"] \arrow[r, "F \star m"'] & F \star F \arrow[r, "m"'] & F
\end{tikzcd}
\qquad
\begin{tikzcd}
F \star F \arrow[dr, "m"] & F \star \mathsf{Id} \arrow[l, "F \star e"'] \arrow[d, "\rho"] \\
\mathsf{Id} \star F \arrow[u, "e \star F"] \arrow[r, "\lambda"'] & F
\end{tikzcd}
$$

From the unit e, one can consider the component $e_I : 1 \to F1$. This component defines a mapping which can be used as the unit morphism for a lax monoidal functor. Similarly, using equation 5.1, the morphism $m : F \star F \to F$ is equivalent to a family of morphisms

$$\vartheta(m)_{A,B} : FA \times FB \to F(A \times B)$$

which is natural in A and B. This family of morphisms corresponds to the multiplicative transformation in a lax monoidal functor. Putting together F, $\vartheta(m)$ and e_I, we obtain a strong lax monoidal functor on Set, that is, an applicative functor.

It remains to be seen if the converse is true: can a monoid in $\mathsf{End}_\star$ be defined from an applicative functor? Given an applicative functor (F, ϕ, η), it easy to see that a multiplication for the monoid can be given from ϕ, using equation 5.1 again. What has to be seen it is if one can recover the whole natural transformation $e : \mathsf{Id} \to F$ out of only one component $\eta : 1 \to F1$. We do so by using the strength of F (which exists since it is an endofunctor on Set): the following composition

$$A \xrightarrow{\langle \mathsf{id}, ! \rangle} A \times 1 \xrightarrow{\mathsf{id} \times \eta} A \times F1 \xrightarrow{\mathsf{st}_{A,1}} F(A \times 1) \xrightarrow{F\pi_1} FA$$

defines a morphism $e_A : A \to FA$ for each A.

All told, *applicative functors are monoids in the category of endofunctors which is monoidal with respect to the Day convolution*.

5.2 Exponential for Applicatives

To apply the Cayley representation, first it must be determined if the category $\mathsf{End}_\star$ is monoidal closed. To do so, we use the same technique we used in section 3.1 for finding the exponential of monads: we apply Yoneda and then the universal property of exponentials.

$$\begin{aligned} G^F(B) &\cong \mathsf{Nat}(\mathsf{Hom}(B,-), G^F) \\ &\cong \mathsf{Nat}(\mathsf{Hom}(B,-) \star F, G) \end{aligned}$$

Therefore, whenever the last expression makes sense, it can be used as the definition of the exponential object. Since we are working on a category of small functors, the expression always makes sense and the exponential is always guaranteed to exist. Doing some further algebra, an alternative form for G^F can be derived (Day, 1973):

$$G^F(B) \cong \mathsf{Nat}(F, G(B \times -))$$

Using Haskell, this exponential can be represented as:

$$\mathbf{data}\ \mathsf{Exp}\ f\ g\ b = \mathsf{Exp}\ (\forall a.\ f\ a \to g\ (b, a))$$

The components of the isomorphism showing it is an exponential are:

$$\begin{aligned}
&\varphi :: (f \star g \xrightarrow{\bullet} h) \to f \xrightarrow{\bullet} \mathsf{Exp}\ g\ h \\
&\varphi\ m\ x = \mathsf{Exp}\ (\lambda y \to m\ (\mathsf{Day}\ x\ y\ \mathsf{id})) \\
&\varphi^{-1} :: \mathsf{Functor}\ h \Rightarrow (f \xrightarrow{\bullet} \mathsf{Exp}\ g\ h) \to f \star g \xrightarrow{\bullet} h \\
&\varphi^{-1}\ f\ (\mathsf{Day}\ x\ y\ h) = \mathsf{fmap}\ h\ (t\ y) \\
&\qquad\qquad\qquad\qquad \mathbf{where}\ \mathsf{Exp}\ t = f\ x
\end{aligned}$$

We therefore conclude that $\mathsf{End}_\star$ is a symmetric monoidal closed category.

5.3 *Free Applicatives*

By Proposition 2.7, the free monoid, viz. the free applicative functor, exists.

The direct application of proposition 2.7 yields the following implementation of the free applicative functor.

$$\textbf{data}\ \mathsf{Free}_\star\ f\ a = \mathsf{Pure}\ a \mid \mathsf{Rec}\ ((f \star \mathsf{Free}_\star\ f)\ a)$$

Inlining the definition of $\star$, we obtain the simplified datatype

$$\begin{array}{l}
\textbf{data}\ \mathsf{Free}_\star\ f\ a\ \textbf{where} \\
\quad \mathsf{Pure} :: a \to \mathsf{Free}_\star\ f\ a \\
\quad \mathsf{Rec}\ \ :: f\ c \to \mathsf{Free}_\star\ f\ d \to ((c,d) \to a) \to \mathsf{Free}_\star\ f\ a
\end{array}$$

with the following instances:

$$\begin{array}{l}
\textbf{instance}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Functor}\ (\mathsf{Free}_\star\ f)\ \textbf{where} \\
\quad \mathsf{fmap}\ g\ (\mathsf{Pure}\ x) \quad\ = \mathsf{Pure}\ (g\ x) \\
\quad \mathsf{fmap}\ g\ (\mathsf{Rec}\ x\ y\ f) = \mathsf{Rec}\ x\ y\ (g \circ f) \\
\textbf{instance}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Applicative}\ (\mathsf{Free}_\star\ f)\ \textbf{where} \\
\quad \mathsf{pure} \qquad\qquad\quad = \mathsf{Pure} \\
\quad \mathsf{Pure}\ g \qquad \circledast z = \mathsf{fmap}\ g\ z \\
\quad (\mathsf{Rec}\ x\ y\ f) \circledast z = \mathsf{Rec}\ x\ (\mathsf{pure}\ (,) \circledast y \circledast z)\ (\lambda(c,(d,a)) \to f\ (c,d)\ a)
\end{array}$$

There is no need to check that the instance satisfies the applicative laws since the definition is derived from Proposition 2.7.

The implementation of the insertion of generators and the universal morphism from the free applicative is:

$$\mathsf{ins} :: \mathsf{Functor}\ a \Rightarrow a \stackrel{\bullet}{\to} \mathsf{Free}_{\star}\ a$$
$$\mathsf{ins}\ x = \mathsf{Rec}\ x\ (\mathsf{Pure}\ ())\ \mathsf{fst}$$

$$\mathsf{free} :: (\mathsf{Functor}\ a, \mathsf{Applicative}\ b) \Rightarrow (a \stackrel{\bullet}{\to} b) \to (\mathsf{Free}_{\star}\ a \stackrel{\bullet}{\to} b)$$

$$
\begin{aligned}
&\mathsf{free}\, f\ (\mathsf{Pure}\, x) &&= \mathsf{pure}\, x \\
&\mathsf{free}\, f\ (\mathsf{Rec}\, x\, y\, g) &&= \mathsf{pure}\ (\mathsf{curry}\, g) \circledast f\ x \circledast \mathsf{free}\, f\ y
\end{aligned}
$$

Alternative presentations of the Day convolution produce alternative types for the free applicative. Using the two alternative expressions for the Day convolution given in equation 5.2, we obtain two alternative definitions of the free applicative functor:

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Free}'_{\star}\ f\ a\ \mathbf{where} \\
&\quad \mathsf{Pure}' :: a \to \mathsf{Free}'_{\star}\ f\ a \\
&\quad \mathsf{Rec}' \ \ :: f\ b \to \mathsf{Free}'_{\star}\ f\ (b \to a) \to \mathsf{Free}'_{\star}\ f\ a \\
&\mathbf{data}\ \mathsf{Free}''_{\star}\ f\ a\ \mathbf{where} \\
&\quad \mathsf{Pure}'' :: a \to \mathsf{Free}''_{\star}\ f\ a \\
&\quad \mathsf{Rec}'' \ \ :: f\ (b \to a) \to \mathsf{Free}''_{\star}\ f\ b \to \mathsf{Free}''_{\star}\ f\ a
\end{aligned}
$$

Hence, the two alternative presentations of the Day convolution given in equation 5.2 give rise to the two notions of free applicative functor found by Capriotti and Kaposi (2014).

5.4 Cayley Representation for Applicatives

Having found the exponential for applicatives, we may apply theorem 2.10 and construct the corresponding Cayley representation.

The Cayley representation is the exponential of a functor over itself.

```
type Rep f = Exp f f

instance Functor f ⇒ Functor (Rep f) where
  fmap f (Exp h) = Exp (fmap (λ(x,y) → (f x,y)) ∘ h)

instance Functor f ⇒ Applicative (Rep f) where
  pure c          = Exp (fmap (c, ))
  Exp f ⊛ Exp a = Exp (fmap g ∘ a ∘ f)
    where g (x,(f,c)) = (f x,c)
```

Again, there is no need to check compliance with applicative laws because the instance is derived from the general construction of the monoid of endomorphism.

Finally, from theorem 2.10, we obtain the applicative morphism rep and the natural transformation abs, together with the property that $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$.

```
rep :: Applicative f ⇒ f →̇ Rep f
rep x = Exp (λy → pure (, ) ⊛ x ⊛ y)

abs :: Applicative f ⇒ Rep f →̇ f
abs (Exp t) = fmap fst (t (pure ()))
```

6 Pre-Arrows as Monoids

Having successfully applied the Cayley representation to monads and applicatives, we wonder if we can find a representation for a third popular notion of computation: arrows. Arrows (Hughes, 2000) were already studied as monoids (Jacobs *et al.*, 2009), resulting in a monoid in the category of profunctors. We briefly review these results.

A profunctor from $\mathscr{C}$ to $\mathscr{D}$ is a functor $\mathscr{D}^{\mathsf{op}} \times \mathscr{C} \to \mathsf{Set}$, sometimes written as $\mathscr{C} \nrightarrow \mathscr{D}$. In a sense, functors are to functions what profunctors are to relations. A morphism between two profunctors is a natural transformation between their underlying functors.

We indicate that a type constructor $h :: * \to * \to *$ is a profunctor by providing an instance of the following type class.

```
class Profunctor h where
  dimap :: (d' → d) → (c → c') → h d c → h d' c'
```

such that the following laws hold

$$\mathsf{dimap}\ \mathsf{id}\ \mathsf{id} = \mathsf{id}$$
$$\mathsf{dimap}\ (f \circ g)\ (h \circ i) = \mathsf{dimap}\ g\ h \circ \mathsf{dimap}\ f\ i$$

Notice how, as opposed to a bifunctor, the type constructor is contravariant on its first argument.

Definition 6.1
The category of profunctors from $\mathscr{C}$ to $\mathscr{D}$, denoted $\mathsf{Prof}(\mathscr{C}, \mathscr{D})$, has as objects profunctors from $\mathscr{C}$ to $\mathscr{D}$, and as morphisms natural transformation between functors $\mathscr{D}^{\mathsf{op}} \times \mathscr{C} \to \mathsf{Set}$.

From now on, we will focus on profunctors $\mathscr{C} \nrightarrow \mathscr{C}$, where $\mathscr{C}$ is a small cartesian closed subcategory of Set with inclusion $J : \mathscr{C} \to \mathsf{Set}$. To avoid notational clutter, we omit the functor J when considering elements of $\mathscr{C}$ as elements of Set.

Profunctors can be composed in such a way that give a notion of tensor (Bénabou, 1973).

Given two profunctors $F, G : \mathscr{C} \nrightarrow \mathscr{C}$, their composition is

$$(F \otimes G)(A,B) = \int^{Z} F(A,Z) \times G(Z,B)$$

The tensor is implemented in Haskell as follows:

data $(\otimes)\ f\ g\ a\ b = \forall z.\ (f\ a\ z) \otimes (g\ z\ b)$

instance $(\mathsf{Profunctor}\ f, \mathsf{Profunctor}\ g) \Rightarrow \mathsf{Profunctor}\ (f \otimes g)$ **where**
  $\mathsf{dimap}\ m_1\ m_2\ (f \otimes g) = (\mathsf{dimap}\ m_1\ \mathsf{id}\ f) \otimes (\mathsf{dimap}\ \mathsf{id}\ m_2\ g)$

The functor $\mathsf{Hom} : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathsf{Set}$ is small and it is the unit for the composition:

$$(F \otimes \mathsf{Hom})(A,B) = \int^{P} F(A,P) \times \mathsf{Hom}(P,B) \cong F(A,B)$$

where the isomorphism holds by the Yoneda lemma. This calculation is used to define a natural isomorphism $\rho : F \otimes \mathsf{Hom} \cong F$. Likewise, natural isomorphisms $\lambda : \mathsf{Hom} \otimes F \cong F$ and $\alpha : (F \otimes G) \otimes H \cong F \otimes (G \otimes H)$ can be defined.

We represent morphisms between profunctors as

$$\textbf{type}\, f \overset{\bullet\bullet}{\rightarrow} g = \forall a\, b.\ f\ a\ b \to g\ a\ b$$

The implementation of λ, ρ, and α are as follows:

$$\textbf{type}\ \mathsf{Hom} = (\to)$$

$$\lambda :: \mathsf{Profunctor}\ f \Rightarrow \mathsf{Hom} \otimes f \overset{\bullet\bullet}{\rightarrow} f$$
$$\lambda\ (f \otimes x) = \mathsf{dimap}\ f\ \mathsf{id}\ x$$

$$\rho :: \mathsf{Profunctor}\ f \Rightarrow f \otimes \mathsf{Hom} \overset{\bullet\bullet}{\rightarrow} f$$
$$\rho\ (x \otimes f) = \mathsf{dimap}\ \mathsf{id}\ f\ x$$

$$\alpha :: (f \otimes g) \otimes h \overset{\bullet\bullet}{\rightarrow} f \otimes (g \otimes h)$$
$$\alpha\ ((f \otimes g) \otimes h) = f \otimes (g \otimes h)$$

Thus, a monoidal structure can be given for $[\mathscr{C}^{\mathsf{op}} \times \mathscr{C}, \mathsf{Set}]$, with composition $\otimes$ as its tensor, and Hom as its unit. We denote this monoidal category by Pro.

Which are the monoids in this monoidal category? A monoid in Pro amounts to:

- A profunctor A,
- a natural transformation $m : A \otimes A \to A$,

- and a unit $e : \mathsf{Hom} \to A$; such that the diagrams

$$\begin{tikzcd}
(A \otimes A) \otimes A \arrow[rr, "m \otimes A"] & & A \otimes A \arrow[d, "m"] \\
A \otimes (A \otimes A) \arrow[u, "\alpha"] \arrow[r, "A \otimes m"'] & A \otimes F \arrow[r, "m"'] & A
\end{tikzcd}
\qquad
\begin{tikzcd}
A \otimes A \arrow[dr, "m"] & A \otimes \mathsf{Hom} \arrow[l, "A \otimes e"'] \arrow[d, "\rho"] \\
\mathsf{Hom} \otimes A \arrow[u, "e \otimes A"] \arrow[r, "\lambda"'] & A
\end{tikzcd}$$

  commute.

Using the isomorphism

$$\left(\int^{Z} A(X,Z) \times A(Z,Y)\right) \to A(X,Y) \cong \int_{Z} A(X,Z) \times A(Z,Y) \to A(X,Y)$$

we get that a natural transformation $m : A \otimes A \to A$ is equivalent to a family of morphisms $m_{X,Y,Z} : A(X,Z) \times A(Z,Y) \to A(X,Y)$ which is natural in X and Y and dinatural in Z.

This presentation makes the connection with arrows evident: m corresponds to the operator $(\ggg)$ and e corresponds to arr. Unfortunately, the first operation is missing. We postpone this problem until the next section, and in the remainder of this section focus on monoids in Pro, i.e. arrows without a first operation, which we call *pre-arrows*.

We introduce a class to represent the monoids in this category. It is simply a restriction of the Arrow class, omitting the first operation.

$$\begin{array}{l}
\textbf{class}\ \mathsf{Profunctor}\ a \Rightarrow \mathsf{PreArrow}\ a\ \textbf{where} \\
\quad \mathsf{arr} \quad :: (b \to c) \to a\ b\ c \\
\quad (\ggg) :: a\ b\ c \to a\ c\ d \to a\ b\ d
\end{array}$$

The laws that must hold are

$$(a \ggg b) \ggg c = a \ggg (b \ggg c)$$

$$\mathsf{arr}\ f \ggg a = \mathsf{dimap}\ f\ \mathsf{id}\ a$$

$$a \ggg \mathsf{arr}\ f = \mathsf{dimap}\ \mathsf{id}\ f\ a$$

$$\mathsf{arr}\ (g \circ f) = \mathsf{arr}\ f \ggg \mathsf{arr}\ g$$

6.1 *Exponential for Pre-Arrows*

The exponential in Pro exists (Bénabou, 1973) and a simple calculation using the Yoneda Lemma shows it to be

$$B^A(X,Y) = \mathsf{Nat}(A(Y,-), B(X,-)).$$

The implementation of exponentials in Pro follows the definition above:

$$\textbf{data}\ \mathsf{Exp}\ a\ b\ x\ y = \mathsf{Exp}\ (\forall d.\ a\ y\ d \to b\ x\ d)$$

$$\begin{aligned}&\textbf{instance}\ (\mathsf{Profunctor}\ g, \mathsf{Profunctor}\ h) \Rightarrow \mathsf{Profunctor}\ (\mathsf{Exp}\ g\ h)\ \textbf{where}\\ &\quad \mathsf{dimap}\ m_1\ m_2\ (\mathsf{Exp}\ gh) = \mathsf{Exp}\ (\mathsf{dimap}\ m_1\ \mathsf{id} \circ gh \circ \mathsf{dimap}\ m_2\ \mathsf{id})\end{aligned}$$

The components of the isomorphism which shows that Exp is an exponential are:

$$\begin{aligned}&\varphi \quad :: (f \otimes g \overset{\bullet\bullet}{\to} h) \to (f \overset{\bullet\bullet}{\to} \mathsf{Exp}\ g\ h)\\ &\varphi \quad m\ f \qquad\ = \mathsf{Exp}\ (\lambda g \to m\ (f \otimes g))\end{aligned}$$

$$\begin{aligned}&\varphi^{-1} :: (f \overset{\bullet\bullet}{\to} \mathsf{Exp}\ g\ h) \to (f \otimes g \overset{\bullet\bullet}{\to} h)\\ &\varphi^{-1}\ m\ (f \otimes g) = e\ g\ \textbf{where}\ \mathsf{Exp}\ e = m\ f\end{aligned}$$

6.2 Free Pre-Arrows

By Proposition 2.7, the free monoid, viz. the free pre-arrow, exists.

The direct application of Proposition 2.7 yields the following implementation of the free pre-arrow.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Free}_{\otimes}\ a\ x\ y\ \textbf{where} \\
\quad \mathsf{Hom}\ \ :: (x \to y) \to \mathsf{Free}_{\otimes}\ a\ x\ y \\
\quad \mathsf{Comp} :: a\ x\ p \to \mathsf{Free}_{\otimes}\ a\ p\ y \to \mathsf{Free}_{\otimes}\ a\ x\ y
\end{array}
$$

with the following instances:

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Profunctor}\ a \Rightarrow \mathsf{Profunctor}\ (\mathsf{Free}_{\otimes}\ a)\ \textbf{where}\\
&\quad \mathsf{dimap}\ f\ g\ (\mathsf{Hom}\ h) \quad\ = \mathsf{Hom}\ (g \circ h \circ f)\\
&\quad \mathsf{dimap}\ f\ g\ (\mathsf{Comp}\ x\ y) = \mathsf{Comp}\ (\mathsf{dimap}\ f\ \mathsf{id}\ x)\ (\mathsf{dimap}\ \mathsf{id}\ g\ y)\\
&\textbf{instance}\ \mathsf{Profunctor}\ a \Rightarrow \mathsf{PreArrow}\ (\mathsf{Free}_{\otimes}\ a)\ \textbf{where}\\
&\quad \mathsf{arr}\ f \qquad\qquad\qquad = \mathsf{Hom}\ f\\
&\quad (\mathsf{Hom}\ f) \quad \ggg c \qquad = \mathsf{dimap}\ f\ \mathsf{id}\ c\\
&\quad (\mathsf{Comp}\ x\ y) \ggg c \qquad = \mathsf{Comp}\ x\ (y \ggg c)
\end{aligned}
$$

There is no need to check that the instance satisfies the pre-arrow laws since the definition is derived from Proposition 2.7.

The insertion of generators and the universal morphism from the free pre-arrow are:

$$
\begin{aligned}
&\mathsf{ins} :: \mathsf{Profunctor}\ a \Rightarrow a \stackrel{\bullet\bullet}{\to} \mathsf{Free}_{\otimes}\ a\\
&\mathsf{ins}\ x = \mathsf{Comp}\ x\ (\mathsf{arr}\ \mathsf{id})\\
&\mathsf{free} :: (\mathsf{Profunctor}\ a, \mathsf{PreArrow}\ b) \Rightarrow (a \stackrel{\bullet\bullet}{\to} b) \to (\mathsf{Free}_{\otimes}\ a \stackrel{\bullet\bullet}{\to} b)\\
&\mathsf{free}\ f\ (\mathsf{Hom}\ g) \quad\ = \mathsf{arr}\ g\\
&\mathsf{free}\ f\ (\mathsf{Comp}\ x\ y) = f\ x \ggg \mathsf{free}\ f\ y
\end{aligned}
$$

6.3 Cayley Representation of Pre-Arrows

Having found the exponential for pre-arrows, we may apply theorem 2.10 and construct the corresponding Cayley representation.

The Cayley representation is the exponential of a profunctor over itself.

$$\textbf{type}\ \mathsf{Rep}\ a = \mathsf{Exp}\ a\ a$$

$$\begin{array}{l}
\textbf{instance}\ \mathsf{Profunctor}\ a \Rightarrow \mathsf{PreArrow}\ (\mathsf{Rep}\ a)\ \textbf{where} \\
\quad \mathsf{arr}\ f \qquad\qquad\qquad\quad = \mathsf{Exp}\ (\lambda y \to \mathsf{dimap}\ f\ \mathsf{id}\ y) \\
\quad (\mathsf{Exp}\ f) \ggg (\mathsf{Exp}\ g) = \mathsf{Exp}\ (\lambda y \to f\ (g\ y))
\end{array}$$

Again, there is no need to check compliance with pre-arrow laws because the instance is derived from the general construction of the monoid of endomorphism.

Finally, from theorem 2.10, we obtain the pre-arrow morphism rep and the natural transformation abs, together with the property that $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$.

$$\begin{array}{l}
\mathsf{rep} :: \mathsf{PreArrow}\ a \Rightarrow a \xrightarrow{\bullet\bullet} \mathsf{Rep}\ a \\
\mathsf{rep}\ x = \mathsf{Exp}\ (\lambda y \to x \ggg y)
\end{array}$$

$$\begin{array}{l}
\mathsf{abs} :: \mathsf{PreArrow}\ a \Rightarrow \mathsf{Rep}\ a \xrightarrow{\bullet\bullet} a \\
\mathsf{abs}\ (\mathsf{Exp}\ f) = f\ (\mathsf{arr}\ \mathsf{id})
\end{array}$$

7 Arrows as Monoids

Returning to the problem of arrows as monoids, we need to internalise the first operation in the categorical presentation. Jacobs et al. (2009) solve this problem by adjoining an *ist* operator to monoids in Pro: an arrow is a monoid (A, m, e) together with a family of morphisms $ist : A(X,Y) \to A(X, Y \times X)$. We take an alternative path. We work on a category of strong profunctors (profunctors with a first-like operator), and then consider monoids in this new monoidal category.

Definition 7.1
A *strength* for a profunctor $F : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathsf{Set}$ is a family of morphisms

$$\mathsf{st}_{X,Y,Z} : F(X,Y) \to F(X \times Z, Y \times Z)$$

that is natural in X, Y and dinatural in Z, such that the following diagrams commute.

$$\begin{tikzcd}
F(X,Y) \arrow[d, "\mathsf{st}_1"'] \arrow[drr, "F(\pi_1,\mathsf{id})"] & & \\
F(X \times 1, Y \times 1) \arrow[rr, "F(\mathsf{id},\pi_1)"'] & & F(X \times 1, Y)
\end{tikzcd}$$

$$\begin{tikzcd}
F(X,Y) \arrow[r, "\mathsf{st}_V"] \arrow[d, "\mathsf{st}_{V\times W}"'] & F(X\times V, Y\times V) \arrow[d, "\mathsf{st}_W"] \\
F(X\times(V\times W), Y\times(V\times W)) \arrow[r, "F(\alpha,\alpha^{-1})"'] & F((X\times V)\times W, (Y\times V)\times W)
\end{tikzcd}$$

We say that a pair (F, st) is a strong profunctor. The diagrams that must commute here are similar to those for a tensorial strength.

The type class of strong profunctors is a simple extension of Profunctor.

$$\textbf{class}\ \mathsf{Profunctor}\ p \Rightarrow \mathsf{StrongProfunctor}\ p\ \textbf{where}$$
$$\quad \mathsf{first} :: p\ x\ y \to p\ (x,z)\ (y,z)$$

Instances of the StrongProfunctor class are subject to the following laws.

$$\begin{aligned}
\mathsf{dimap}\ \mathsf{id}\ \pi_1\ (\mathsf{first}\ a) &= \mathsf{dimap}\ \pi_1\ \mathsf{id}\ a \\
\mathsf{first}\ (\mathsf{first}\ a) &= \mathsf{dimap}\ \alpha\ \alpha^{-1}\ (\mathsf{first}\ a) \\
\mathsf{dimap}\ (\mathsf{id} \times f)\ \mathsf{id}\ (\mathsf{first}\ a) &= \mathsf{dimap}\ \mathsf{id}\ (\mathsf{id} \times f)\ (\mathsf{first}\ a)
\end{aligned}$$

The first two laws correspond to the two diagrams above, while the third one corresponds to dinaturality of first in the z variable.

In contrast to strong functors on Set, the strength of a profunctor may not exist, and even if it exists, it may not be unique.

As an example of strengths not being unique, consider the following profunctor:

```
data Double x y = Double ((x,x) → (y,y))
instance Profunctor Double where
  dimap f g (Double h) = Double (lift g ∘ h ∘ lift f)
    where lift :: (a → b) → (a,a) → (b,b)
          lift f (a,a′) = (f a,f a′)
```

there exist two possible instances satisfying the strength axioms.

```
instance StrongProfunctor Double where
  first (Double f) = Double g
    where g ((x,z),(x′,z′)) = ((y,z),(y′,z′))
                    where (y,y′) = f (x,x′)

instance StrongProfunctor Double where
  first (Double f) = Double g
    where g ((x,z),(x′,z′)) = ((y,z),(y′,z))
                    where (y,y′) = f (x,x′)
```

Therefore, the profunctor Double does not have a unique strength.

Given two strong profunctors (F,st^F), (G,st^G), a *strong natural transformation* is a natural transformation $\alpha : F \to G$ that is compatible with the strengths:

$$\begin{tikzcd} F(X,Y) \arrow[r, "\mathsf{st}^F"] \arrow[d, "\alpha"'] & F(X \times Z, Y \times Z) \arrow[d, "\alpha"] \\ G(X,Y) \arrow[r, "\mathsf{st}^G"'] & G(X \times Z, Y \times Z) \end{tikzcd}$$

Following the approach to strong monads of Moggi (1995), we work with the category $[\mathscr{C}^{\mathsf{op}} \times \mathscr{C}, \mathsf{Set}]_{\mathsf{str}}$ of strong profunctors.

Definition 7.2
The category $[\mathscr{C}^{\mathsf{op}} \times \mathscr{C}, \mathsf{Set}]_{\mathsf{str}}$ consists of pairs (F, st) as objects, where F is a profunctor and st is a strength for it, and strong natural transformations as morphisms.

Even when the strength for a functor is not unique, we usually write (F, st^F). Here the superscript F in st^F is just syntax to distinguish between various strengths for different profunctors, but it does not mean that st^F is *the* strength for F.

The monoidal structure of Pro can be used for strong profunctors. Given two strong profunctors (A, st^A) and (B, st^B), a family of morphisms $\mathsf{st}_Z^{A \otimes B} = \left[\Lambda P. \iota_P \circ (\mathsf{st}_Z^A \times \mathsf{st}_Z^B)\right]$ is defined. It is easy to see that such family is indeed a strength for the profunctor $A \otimes B$. The monoidal category of strong profunctors with tensor defined this way is denoted by SPro.

A monoid in SPro amounts to the same data that we had in the case of Pro. This time, however, the morphisms m and e (being morphisms of SPro) must be compatible with the strength as well.

Arrows can be implemented as strong profunctors which are pre-arrows.

$$\textbf{class}\ (\mathsf{StrongProfunctor}\ a, \mathsf{PreArrow}\ a) \Rightarrow \mathsf{Arrow}\ a$$

Instances of Arrow are empty, but the programmer should check the compatibility of the unit and multiplication of the pre-arrow with the strength:

$$\mathsf{first}\ (\mathsf{arr}\ f) = \mathsf{arr}\ (f \times \mathsf{id})$$
$$\mathsf{first}\ (a \ggg b) = \mathsf{first}\ a \ggg \mathsf{first}\ b$$

These two laws, together with the laws for profunctors, pre-arrows and strength, constitute the arrows laws proposed by Paterson (2003).

7.1 *Exponential for Arrows*

Unfortunately, we have not managed to find an exponential for arrows. Part of the difficulty in finding one seems to stem from the fact that strengths for profunctors may not exist, and even if they do, they may not be unique. For example, given two strong profunctors A and B, the profunctor B^A defined in Section 6.1, does not seem to have a strength.

However, as shown next, it is possible to co-freely add a strength to a pre-arrow, and use that to obtain a representation for arrows.

7.2 Adding a Strength to Pre-Arrows

We mitigate the failure to find an exponential in SPro by building on the success in Pro. Concretely, we investigate how to add a strength to profunctors in Pro, and use this to complete the constructions in Pro to make them work in SPro.

There is an obvious monoidal functor from the monoidal category of strong profunctors SPro into the monoidal category of profunctors Pro that forgets the additional structure. More precisely, the functor $U : \mathsf{SPro} \to \mathsf{Pro}$ forgets the strength.

$$U(A, \mathsf{st}^A) = A$$

Interestingly, this functor has a right adjoint. That is, there is a functor T such that we have a natural isomorphism.

$$\phi : \mathsf{Pro}(U(A, \mathsf{st}^A), B) \cong \mathsf{SPro}((A, \mathsf{st}^A), TB) \tag{7.1}$$

The monoidal functor $T : \mathsf{Pro} \to \mathsf{SPro}$ is given by $TB = (T_B, \mathsf{st})$, with its components defined by

$$\begin{aligned} T_B(X, Y) &= \int_Z B(X \times Z, Y \times Z) \\ \mathsf{st}_Z &= \langle \Lambda V. B(\alpha, \alpha^{-1}) \circ \omega_{Z \times V} \rangle \end{aligned}$$

The adjunction $U \dashv T$ tells us that T completes a profunctors by (co)freely adding a strength. Pastro and Street, when working on Tambara modules (Pastro & Street, 2008), introduced an endofunctor on the category of profunctors which adds a structure similar to

what we call a strength. The functor T is based on that endofunctor and hence we call it the *Tambara functor*.

The Tambara functor may be implemented as follows.

```
data Tambara a x y = Tambara (∀z. a (x,z) (y,z))
instance Profunctor a ⇒ Profunctor (Tambara a) where
  dimap f g (Tambara x) = Tambara (dimap (lift f) (lift g) x)
          where lift f (a,b) = (f a,b)
instance Profunctor a ⇒ StrongProfunctor (Tambara a) where
  first (Tambara x) = Tambara (dimap α α⁻¹ x)
    where α    ((x,y),z) = (x,(y,z))
          α⁻¹  (x,(y,z)) = ((x,y),z)
```

The isomorphism 7.1 is witnessed by morphisms:

$$\left(\phi(\eta)_{(A,\mathsf{st}^A),B}\right)_{X,Y} = \langle \Lambda Z.\eta_{X\times Z,Y\times Z} \circ \mathsf{st}^A_Z \rangle$$
$$\left(\phi^{-1}(\beta)_{(A,\mathsf{st}^A),B}\right)_{X,Y} = B({\pi_1}^{-1},\pi_1) \circ \omega_1 \circ \beta_{X,Y}$$

The components of the isomorphism 7.1 are implemented in Haskell as follows.

```
ϕ :: (StrongProfunctor a, Profunctor b) ⇒ (a ⇸ b) → (a ⇸ Tambara b)
ϕ f a = Tambara (f (first a))
ϕ⁻¹ :: (StrongProfunctor a, Profunctor b) ⇒ (a ⇸ Tambara b) → (a ⇸ b)
ϕ⁻¹ f a = dimap fst⁻¹ fst b
  where Tambara b = f a
        fst⁻¹ x = (x, ())
```

Since we have an adjunction $U \dashv T$, we can form a comonad $UT : \mathsf{Pro} \to \mathsf{Pro}$. The counit of UT is the counit of the adjunction: $\varepsilon = \phi^{-1}(\mathsf{id} : A \to A) = A({\pi_1}^{-1},\pi_1) \circ \omega_1$, and its comultiplication is $\delta = U\phi(\mathsf{id} : UTA \to UTA)T = \langle \Lambda Z.\langle \Lambda V.A(\alpha,\alpha^{-1}) \circ \omega_{Z\times V}\rangle\rangle$.

Proposition 7.3
The category SPro is equivalent to the (co)Eilenberg-Moore category for the comonad UT.

Proof A coalgebra for this comonad is an object A from Pro together with a morphism

$\sigma : A \to UTA$ such that these diagrams commute:

$$
\begin{tikzcd}
UTA \arrow[r, "\sigma"] \arrow[dr, equal] & UT(UTA) \arrow[d, "\varepsilon"] \\
 & UTA
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "\sigma"] \arrow[d, "\sigma"'] & UTA \arrow[d, "UT\sigma"] \\
UTA \arrow[r, "\delta"'] & UT(UTA)
\end{tikzcd}
$$

The morphism σ is a family of morphisms

$$\sigma_{X,Y} : A(X,Y) \to \int_Z A(X \times Z, Y \times Z)$$

natural in X and Y. By the universal property of ends, the family $\sigma_{X,Y}$ is equivalent to a family of morphisms

$$\overline{\sigma}_{X,Y,Z} : A(X,Y) \to A(X \times Z, Y \times Z)$$

natural in X and Y, and dinatural in Z. Such family has exactly the form of a strength. Using the laws for coalgebras of a comonad, one can prove that $\overline{\sigma}$ is indeed a strength for A. $\square$

7.3 A Representation of Arrows

Although we have not found the form of exponential objects in SPro, an alternative representation for monoids can be given with help of the Tambara functor.

The idea is to take a monoid in SPro and forget the strength structure using U. Then, use the Cayley representation for monoids in Pro, and finally apply the Tambara functor to obtain a new strength on this monoid. That is, given a monoid (M, m, e) in SPro, its representation is $T(UM^{UM})$. The functor T is monoidal and therefore, as shown by Theorem 8.1, it takes monoids in Pro to monoids in SPro.

More concretely, given a monoid $((A, \mathsf{st}^A), m, e)$ in SPro (i.e. and arrow), we construct a morphism $\mathsf{rep} : (A, \mathsf{st}^A) \to TA^A$ as

$$\mathsf{rep}_{X,Y} = \langle \Lambda Z. \langle \Lambda D. \lfloor m \circ \iota_{Y \times Z} \circ (\mathsf{st}^A_Z \times \mathsf{id}) \rfloor \rangle \rangle$$

This is a legit morphism in SPro, i.e. it commutes with the strengths of A and TA^A. It has a left inverse, $\mathsf{abs} : UTA^A \to U(A, \mathsf{st}^A)$ defined as

$$\mathsf{abs}_{X,Y} = A({\pi_1}^{-1}, \mathsf{id}) \circ \mathsf{ev} \circ (\mathsf{id} \times e) \circ \langle \mathsf{id}, \lfloor \pi_1 \circ \pi_2 \rfloor \circ ! \rangle \circ \omega_Y \circ \omega_1$$

and therefore, rep is a monomorphism. This proves that TA^A is a representation for (A, st^A).

The representation is implemented in Haskell as follows.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Rep}\ a\ x\ y = \mathsf{Rep}\ (\forall z'\ z.\ a\ (y,z')\ z \rightarrow a\ (x,z')\ z) \\
\textbf{instance}\ \mathsf{Profunctor}\ a \Rightarrow \mathsf{Profunctor}\ (\mathsf{Rep}\ a)\ \textbf{where} \\
\quad \mathsf{dimap}\ f\ g\ (\mathsf{Rep}\ x) = \mathsf{Rep}\ (\lambda y \rightarrow \mathsf{dimap}\ (\mathsf{lift}\ f)\ \mathsf{id}\ (x\ (\mathsf{dimap}\ (\mathsf{lift}\ g)\ \mathsf{id}\ y))) \\
\qquad \textbf{where}\ \mathsf{lift}\ f\ (a,b) = (f\ a,b)
\end{array}
$$

The representation takes any profunctor into an arrow.

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Profunctor}\ a \Rightarrow \mathsf{PreArrow}\ (\mathsf{Rep}\ a)\ \textbf{where} \\
\quad \mathsf{arr}\ f = \mathsf{Rep}\ (\mathsf{dimap}\ (\mathsf{lift}\ f)\ \mathsf{id})\ \textbf{where}\ \mathsf{lift}\ f\ (a,b) = (f\ a,b) \\
\quad \mathsf{Rep}\ x \ggg \mathsf{Rep}\ y = \mathsf{Rep}\ (\lambda v \rightarrow x\ (y\ v)) \\
\textbf{instance}\ \mathsf{Profunctor}\ a \Rightarrow \mathsf{StrongProfunctor}\ (\mathsf{Rep}\ a)\ \textbf{where} \\
\quad \mathsf{first}\ (\mathsf{Rep}\ x) = \mathsf{Rep}\ (\lambda z \rightarrow \mathsf{dimap}\ \alpha\ \mathsf{id}\ (x\ (\mathsf{dimap}\ \alpha^{-1}\ \mathsf{id}\ z))) \\
\qquad \textbf{where}\ \alpha\ ((x,y),z) = (x,(y,z)) \\
\qquad\qquad\quad \alpha^{-1}\ (x,(y,z)) = ((x,y),z)
\end{array}
$$

Since we verified that the strength is compatible with the pre-arrow structure, we may declare the Arrow instance.

$$\textbf{instance}\ \mathsf{Profunctor}\ a \Rightarrow \mathsf{Arrow}\ (\mathsf{Rep}\ a)$$

Any arrow a can be represented by $\mathsf{Rep}\ a$. Moreover, $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$.

$$\begin{aligned}
&\mathsf{rep} :: \mathsf{Arrow}\ a \Rightarrow a\ x\ y \to \mathsf{Rep}\ a\ x\ y \\
&\mathsf{rep}\ x = \mathsf{Rep}\ (\lambda z \to \mathsf{first}\ x \ggg z)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{abs} :: \mathsf{Arrow}\ a \Rightarrow \mathsf{Rep}\ a\ x\ y \to a\ x\ y \\
&\mathsf{abs}\ (\mathsf{Rep}\ x) = \mathsf{arr}\ \mathsf{fst}^{-1} \ggg x\ (\mathsf{arr}\ \mathsf{fst}) \\
&\quad \textbf{where}\ \mathsf{fst}^{-1}\ y = (y, ())
\end{aligned}$$

7.4 Free Arrows

Having failed to find an exponential in SPro, we cannot apply Proposition 2.7 to obtain the free monoid in SPro and therefore we fall back to finding it directly. Fortunately, we do not need to search much as the free monoid on Pro is equipped with an obvious strength whenever it is built over a strong profunctor, and indeed one can verify that the obtained monoid is the free monoid in SPro.

The free pre-arrow can be equipped with a strength when defined over a strong profunctor.

$$\begin{aligned}
&\textbf{instance}\ \mathsf{StrongProfunctor}\ a \Rightarrow \mathsf{StrongProfunctor}\ (\mathsf{Free}_{\otimes}\ a)\ \textbf{where}\\
&\quad \mathsf{first}\ (\mathsf{Hom}\ f) \quad\ = \mathsf{Hom}\ (\lambda(x,z) \rightarrow (f\ x,z))\\
&\quad \mathsf{first}\ (\mathsf{Comp}\ x\ y) = \mathsf{Comp}\ (\mathsf{first}\ x)\ (\mathsf{first}\ y)
\end{aligned}$$

Since the unit and multiplication of the free arrow are compatible with the strength, we can declare the Arrow instance without guilt.

$$\textbf{instance}\ \mathsf{StrongProfunctor}\ a \Rightarrow \mathsf{Arrow}\ (\mathsf{Free}_{\otimes}\ a)$$

The insertion of generators and the universal morphism are the same as the ones from pre-arrows. The only difference is that now we require StrongProfunctors instead of plain Profunctors.

$$\begin{aligned}
&\mathsf{ins} :: \mathsf{StrongProfunctor}\ a \Rightarrow a \overset{\bullet\bullet}{\rightarrow} \mathsf{Free}_{\otimes}\ a\\
&\mathsf{ins}\ x = \mathsf{Comp}\ x\ (\mathsf{arr}\ \mathsf{id})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{free} :: (\mathsf{StrongProfunctor}\ a, \mathsf{Arrow}\ b) \Rightarrow (a \overset{\bullet\bullet}{\rightarrow} b) \rightarrow (\mathsf{Free}_{\otimes}\ a \overset{\bullet\bullet}{\rightarrow} b)\\
&\mathsf{free}\ f\ (\mathsf{Hom}\ g) \quad\ = \mathsf{arr}\ g\\
&\mathsf{free}\ f\ (\mathsf{Comp}\ x\ y) = f\ x \ggg \mathsf{free}\ f\ y
\end{aligned}$$

Here, we would really like the type $(a \overset{\bullet\bullet}{\rightarrow} b)$ to represent strength preserving

Here, we would really like the type $(a \rightsquigarrow b)$ to represent strength preserving morphisms between strong profunctors. Therefore, free f will preserve the strength only when f does.

Remark 7.4
We now have adjunctions

$$\begin{tikzcd} \mathsf{Pro} \arrow[r, bend right, "T"'] \arrow[r, phantom, "\perp"] & \mathsf{SPro} \arrow[l, bend right, "U"'] \arrow[r, bend left, "\_^*"] \arrow[r, phantom, "\perp"] & \mathsf{Mon}(\mathsf{SPro}). \arrow[l, bend left, "U"] \end{tikzcd}$$

However, they do not compose. Free arrows are generated freely from SPro but the strength is generated co-freely from Pro. This provides an explanation for the difficulty in defining a free arrow over an arbitrary (weak) profunctor.

8 On Functors Between Monoidal Categories

Monads, applicatives and arrows have been introduced as monoids in monoidal categories. Now we ask what is the relation between these monoidal categories. It is well-known that from a monad it can be derived both an applicative functor and an arrow. In this section we explain these and other derivations from the point of view of monoidal categories.

For example, in order to obtain a pre-arrow from a monad, we are interested in creating a monoid in Pro, given a monoid in $\mathsf{End}_\circ$. Instead of trying to make up a monoid in Pro directly, we will define a monoidal functor between the underlying monoidal categories (in this case $\mathsf{End}_\circ$ and Pro), and then use the following theorem to obtain a functor between the corresponding monoids.

Theorem 8.1
Let $(F,\phi,\eta) : \mathscr{C} \to \mathscr{D}$ be a lax monoidal functor. If (M,m,e) is a monoid in $\mathscr{C}$, then $(FM, Fm\circ\phi, Fe\circ\eta)$ is a monoid in $\mathscr{D}$.

The above construction extends to a functor, and therefore we can induce functors between monoids by way of lax monoidal functors between their underlying monoidal categories.

8.1 The Cayley Functor

Applicative functors can be used to create arrows, here we present a monoidal functor that gives rise to such construction. We consider the *Cayley functor* (Pastro & Street, 2008)

$$C : \mathsf{End}_\star \to \mathsf{SPro}$$

defined by

$$C(F)(X,Y) = F(Y^X)$$

Despite its name, this functor bears no direct relation with the Cayley representation.

Proposition 8.2
The Cayley functor is monoidal from $\mathsf{End}_\star$ to Pro.

Not only this functor is monoidal but also, for each $C(F)$, we can define a strength. This extends C into a monoidal functor from $\mathsf{End}_\star$ to SPro.

The resulting construction is the static arrow over $(\rightarrow)$, augmented with the original applicative (McBride & Paterson, 2008).

$$\mathbf{data}\ \mathsf{Cayley}\ f\ x\ y = \mathsf{Cayley}\ (f\ (x \rightarrow y))$$

For every applicative functor, the Cayley functor constructs an arrow.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{instance}\ \mathsf{Applicative}\ f \Rightarrow \mathsf{PreArrow}\ (\mathsf{Cayley}\ f)\ \mathbf{where}} \\
\quad \mathsf{arr}\ f & = \mathsf{Cayley}\ (\mathsf{pure}\ f) \\
\quad (\mathsf{Cayley}\ x) \ggg (\mathsf{Cayley}\ y) & = \mathsf{Cayley}\ (\mathsf{pure}\ (\circ) \circledast y \circledast x) \\
\multicolumn{2}{l}{\mathbf{instance}\ \mathsf{Applicative}\ f \Rightarrow \mathsf{StrongProfunctor}\ (\mathsf{Cayley}\ f)\ \mathbf{where}} \\
\quad \mathsf{first}\ (\mathsf{Cayley}\ x) & = \mathsf{Cayley}\ (\mathsf{pure}\ (\lambda f \rightarrow \lambda (b,d) \rightarrow (f\ b, d)) \circledast x) \\
\multicolumn{2}{l}{\mathbf{instance}\ \mathsf{Applicative}\ f \Rightarrow \mathsf{Arrow}\ (\mathsf{Cayley}\ f)}
\end{array}$$

8.2 The Kleisli Functor

The well-known Kleisli category of a monad gives rise to a monoidal functor from monads to arrows. We consider the functor

$$K : \mathsf{End}_{\circ} \to \mathsf{SPro}$$

defined by

$$K(F)(X,Y) = (F(Y))^X$$

The implementation of the Kleisli functor is as follows.

```haskell
data Kleisli f x y = Kleisli (x -> f y)

instance Monad f => PreArrow (Kleisli f) where
  arr f                     = Kleisli (\x -> return (f x))
  (Kleisli f) >>> (Kleisli g) = Kleisli (\x -> f x >>= g)

instance Monad f => StrongProfunctor (Kleisli f) where
  first (Kleisli f)         = Kleisli (\(b,d) -> f b >>= \c -> return (c,d))

instance Monad f => Arrow (Kleisli f)
```

8.3 *The Identity Functor*

The identity endofunctor on $[\mathsf{Set},\mathsf{Set}]$ can be given a monoidal compatibility morphisms η and ϕ such that it becomes a lax monoidal functor from $\mathsf{End}_{\circ}$ to $\mathsf{End}_{\star}$. The η morphism is the identity on the identity functor. The morphism $\phi_{F,G} : F \star G \to F \circ G$ is given by:

$$
\begin{aligned}
(F \star G)(A) = FC \times GD \times A^{(C \times D)} &\xrightarrow{\mathsf{st}} F(C \times (GD \times A^{(C \times D)})) \\
&\xrightarrow{F\mathsf{st}} F(G(C \times D \times A^{(C \times D)})) \\
&\xrightarrow{F(G\mathsf{ev})} F(GA)
\end{aligned}
$$

Hence, we obtain a lax monoidal functor $\hat{\mathsf{Id}} : \mathsf{End}_{\circ} \to \mathsf{End}_{\star}$.

By applying Theorem 8.1 to $\hat{\mathsf{Id}}$, we obtain the well-known result that every monad is an applicative functor.

$$\begin{array}{l}
\textbf{instance}\ \mathsf{Monad}\ f \Rightarrow \mathsf{Applicative}\ f\ \textbf{where} \\
\quad \mathsf{pure} = \mathsf{return} \\
\quad f \circledast x = f \gg\!= (\lambda g \to x \gg\!= \mathsf{return} \circ g)
\end{array}$$

8.4 The Reversed Monoid

For every monoidal category $\mathscr{C}_{\otimes} = (\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$, the opposite monoidal category $\mathscr{C}_{\otimes^{\mathsf{op}}}$ can be defined, with the monoidal operator $A \otimes^{\mathsf{op}} B = B \otimes A$. Given a monoid in $\mathscr{C}_{\otimes}$, a monoid in $\mathscr{C}_{\otimes^{\mathsf{op}}}$ can be defined.

Theorem 8.3
If (M, m, e) is a monoid in $\mathscr{C}_{\otimes}$, then (M, m, e) is a monoid in $\mathscr{C}_{\otimes^{\mathsf{op}}}$.

In the case where the monoidal structure is symmetric, there is an isomorphism between $A \otimes^{\mathsf{op}} B$ and $A \otimes B$. Using this isomorphism, a monoidal structure can be given to the identity endofunctor over $\mathscr{C}$, giving a monoidal functor from $\mathscr{C}_\otimes$ to $\mathscr{C}_{\otimes^{\mathsf{op}}}$.

Theorem 8.4
Let $\mathscr{C}_\otimes = (\mathscr{C}, \otimes, I, \alpha, \lambda, \rho, \gamma)$ be a symmetric monoidal category, then we have a monoidal functor $(\mathsf{Id}, \gamma, \mathsf{id}) : \mathscr{C}_\otimes \to \mathscr{C}_{\otimes^{\mathsf{op}}}$.

If we apply Theorem 8.1 to a monoid M in $\mathscr{C}_\otimes$, we obtain a monoid in $\mathscr{C}_{\otimes^{\mathsf{op}}}$. From Theorem 8.3, this monoid can be converted to a monoid in $\mathscr{C}_\otimes$. This last monoid is what we call the *reversed monoid of M*.

As already mentioned, $\mathsf{End}_\star$ is a symmetric monoidal category, and therefore the reverse monoid construction can be applied to a monoid in $\mathsf{End}_\star$. The resulting monoid is known as the *reversed applicative* (Bird *et al.*, 2013).

The reversed applicative is implemented as:

$$\begin{aligned}
&\textbf{data}\ \mathsf{Rev}\ f\ x = \mathsf{Rev}\ (f\ x)\ \textbf{deriving}\ \mathsf{Functor}\\
&\textbf{instance}\ \mathsf{Applicative}\ f \Rightarrow \mathsf{Applicative}\ (\mathsf{Rev}\ f)\ \textbf{where}\\
&\quad \mathsf{pure} \qquad\qquad\quad = \mathsf{Rev} \circ \mathsf{pure}\\
&\quad \mathsf{Rev}\ f \circledast \mathsf{Rev}\ x = \mathsf{Rev}\ (\mathsf{pure}\ (\mathsf{flip}\ (\$)) \circledast x \circledast f)
\end{aligned}$$

In intuitive terms, the difference between f and $\mathsf{Rev}\ f$ as applicative functors is that $\mathsf{Rev}\ f$ sequences the order of effects in the opposite order (Bird *et al.*, 2013).

9 Conclusion

We have seen how monads, applicative functors and arrows can be cast as monoids in a monoidal category. We exploited this uniformity in order to obtain free constructions and

representations for the three notions of computation. We have provided Haskell code for all of the concepts, showing that the ideas can be readily implemented without difficulty. The representations for applicative functors and arrows are new and they optimise code in the same cases the codensity transformation and difference lists work well: when the binary operation of the monoid is expensive on its first argument and therefore, we want to associate a sequence of computations to the right. In order to prove this formally, we could adopt the framework of Hackett and Hutton (2014).

The constructions presented for monads are well known (Mac Lane, 1971). Day has shown the equivalence of lax monoidal functors and monoids with respect to the Day convolution (Day, 1970). However, in the functional programming community, this fact is not well-known. The construction of free applicatives is described by Capriotti and Kaposi (2014). While they provide plenty of motivation for the use of the free applicative functor, we give a detailed description of its origin, as we arrive at it instantiating a general description of free monoids to the category of endofunctors which is monoidal with respect to the Day convolution.

There are several works analysing the formulation of arrows as monoids (Jacobs *et al.*, 2(

Atkey, 2011; Asada, 2010; Asada & Hasuo, 2010). We differentiate from their work in our treatment of the strength. We believe our approach leads to simpler definitions, as only standard monoidal categories are used. Moreover, our definition of the free arrow is possible thanks to this simpler approach.

Jaskelioff and Moggi (2010) use the Cayley representation for monoids in a monoidal category in order to lift operations through monoid transformers. However, the only instances considered are monads.

For simplicity, we analysed the above notions of computations as Set functors. However, for size reasons, many constructions were restricted to small functors, which are extensions of functors from small categories. Alternatively, we could have worked with accessible functors (Adámek & Rosický, 1994) (which are equivalent to small functors), or we could have worked directly with functors from small categories, as it is done in relative monads (Altenkirch *et al.*, 2010). However, by working with small functors the category theory is less heavy and the implementation in Haskell is more direct.

In functional programming, for each of the three notions of computation that we considered, there are variants which add structure. For example, monads can be extended with MonadPlus, applicative functors with Alternative, and arrows with ArrowChoice, to name just a few. It would be interesting to analyse the relation between the different extensions from the point of view of monoidal categories with extra structure.

Unifying different concepts under one common framework is a worthy goal as it deepens our understanding and it allows us to relate, compare, and translate ideas. It has long been recognised that category theory is an ideal tool for this task (Reynolds, 1980) and this article provides a bit more evidence of it.

Acknowledgements

We thank Ondřej Rypáček and Jennifer Hackett for their insightful comments on an early version of this document. This work was funded by the Agencia Nacional de Promoción

Científica y Tecnológica (PICT 2009–15) and Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).

References

Adámek, Jiří, & Rosický, Jiří. (1994). *Locally presentable and accessible categories*. Cambridge University Press.

Altenkirch, Thorsten, Chapman, James, & Uustalu, Tarmo. (2010). Monads need not be endofunctors. *Pages 297–311 of:* Ong, Luke (ed), *Foundations of Software Science and Computational Structures*. Lecture Notes in Computer Science, vol. 6014. Springer Berlin Heidelberg.

Asada, Kazuyuki. (2010). Arrows are strong monads. *Pages 33–42 of: Proceedings of the third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*. MSFP '10.

Asada, Kazuyuki, & Hasuo, Ichiro. (2010). Categorifying computations into components via arrows as profunctors. *Electronic Notes in Theoretical Computer Science*, **264**(2), 25–45.

Atkey, Robert. (2011). What is a categorical model of arrows? *Electronic notes in theoretical computer science*, **229**(5), 19–37.

Barr, Michael, & Wells, Charles. (1985). *Toposes, triples and theories*. Grundlehren der Mathematischen Wissenschaften, vol. 278. New York: Springer-Verlag.

Bénabou, Jean. (1973). *Les distributeurs: d'après le cours de questions spéciales de mathématique*. Rapport (Université catholique de Louvain (1970-). Séminaire de mathématique pure)). Institut de mathématique pure et appliquée, Université catholique de Louvain.

Bird, Richard, Gibbons, Jeremy, Mehner, Stefan, Schrijvers, Tom, & Voigtlaender, Janis. (2013). Understanding idiomatic traversals backwards and forwards. *Acm sigplan haskell symposium*.

Capriotti, Paolo, & Kaposi, Ambros. 2014 (April). Free applicative functors. *Proceedings of the fifth Workshop on Mathematically Structured Functional Programming*. MSFP '14.

Cayley, Arthur. (1854). On the theory of groups as depending on the symbolic equation $\theta^n = 1$. *Philosophical magazine*, **7**(42), 40–47.

Day, Brian. (1970). On closed categories of functors. *Pages 1–38 of: Reports of the midwest category seminar iv*. Lecture Notes in Mathematics, vol. 137. Springer Berlin Heidelberg.

Day, Brian. (1973). Note on monoidal localisation. *Bulletin of the Australian Mathematical Society*, **8**(2), 1–16.

Day, Brian J., & Lack, Stephen. (2007). Limits of small functors. *Journal of pure and applied algebra*, **210**(3), 651 – 663.

Dubuc, Eduardo J. (1974). Free monoids. *Journal of algebra*, **29**(2), 208 – 228.

Hackett, Jennifer, & Hutton, Graham. (2014). *Worker/Wrapper/Makes It/Faster*. Submitted to the International Conference on Functional Programming.

Hughes, John. (1986). A novel representation of lists and its application to the function "reverse". *Information processing letters*, **22**(3), 141–144.

Hughes, John. (2000). Generalising monads to arrows. *Science of computer programming*, **37**(1-3), 67–111.

Hutton, Graham, Jaskelioff, Mauro, & Gill, Andy. (2010). Factorising folds for faster functions. *Journal of functional programming*, **20**(Special Issue 3-4), 353–373.

Jacobs, Bart, Heunen, Chris, & Hasuo, Ichiro. (2009). Categorical semantics for arrows. *Journal of functional programming*, **19**(3-4), 403–438.

Jacobson, Nathan. (2009). *Basic Algebra I*. Basic Algebra. Dover Publications, Incorporated.

Jaskelioff, Mauro, & Moggi, Eugenio. (2010). Monad transformers as monoid transformers. *Theoretical computer science*, **411**(51-52), 4441 – 4466.

Jaskelioff, Mauro, & Rypacek, Ondrej. (2012). An investigation of the laws of traversals. *Pages 40–49 of:* Chapman, James, & Levy, Paul Blain (eds), *Proceedings of the fourth workshop on mathematically structured functional programming*. EPTCS, vol. 76.

Kelly, G. Maxwell. (1980). A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on. *Bulletin of the Australian Mathematical Society*, **22**(01), 1–83.

Kelly, G.M., & Power, A.J. (1993). Adjunctions whose counits are coequalizers, and presentations of finitary enriched monads. *Journal of pure and applied algebra*, **89**(1–2), 163 – 179.

Lack, Stephen. (2010). Note on the construction of free monoids. *Applied categorical structures*, **18**(1), 17–29.

Lindley, Sam, Wadler, Philip, & Yallop, Jeremy. (2011). Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic notes on theoretical computer science*, **229**(5), 97–117.

Mac Lane, Saunders. (1971). *Categories for the working mathematician*. Graduate Texts in Mathematics, no. 5. Springer-Verlag. Second edition, 1998.

McBride, Conor, & Paterson, Ross. (2008). Applicative programming with effects. *Journal of functional programming*, **18**(01), 1–13.

Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Pages 124–144 of: Proceedings of the 5th acm conference on functional programming languages and computer architecture*. New York, NY, USA: Springer-Verlag New York, Inc.

Moggi, Eugenio. (1989). Computational lambda-calculus and monads. *Pages 14–23 of: Lics*. IEEE Computer Society.

Moggi, Eugenio. (1991). Notions of computation and monads. *Inf. comput.*, **93**(1), 55–92.

Moggi, Eugenio. (1995). A semantics for evaluation logic. *Fundam. inform.*, **22**(1/2), 117–152.

Pastro, Craig, & Street, Ross. (2008). Doubles for monoidal categories. *Theory and applications of categories*, **21**, 61–75.

Paterson, Ross. (2003). Arrows and computation. *Pages 201–222 of:* Gibbons, Jeremy, & de Moor, Oege (eds), *The fun of programming*. Palgrave.

Paterson, Ross. (2012). Constructing applicative functors. *Pages 300–323 of:* Gibbons, Jeremy, & Nogueira, Pablo (eds), *Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 7342. Springer Berlin Heidelberg.

Reynolds, John C. (1980). Using category theory to design implicit conversions and generic operators. *Pages 211–258 of:* Jones, Neil D. (ed), *Semantics-directed compiler generation*. Lecture Notes in Computer Science, vol. 94. Springer.

Voigtländer, Janis. (2008). Asymptotic improvement of computations over free monads. *Pages 388–403 of: Mpc*.