JFP **27**, e21, 42 pages, 2017. © Cambridge University Press 2017
doi:10.1017/S0956796817000132

*Notions of computation as monoids**

EXEQUIEL RIVAS and MAURO JASKELIOFF

Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas, CONICET, Rosario, Santa Fe, Argentina
FCEIA, Universidad Nacional de Rosario, Rosario, Santa Fe, Argentina
(*e-mails:* rivas@cifasis-conicet.gov.ar, jaskelioff@cifasis-conicet.gov.ar)

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 instances of a unifying abstract concept: monoids in monoidal categories. We demonstrate that even when working at this high level of generality, one can obtain useful results. In particular, we give conditions under which one can obtain free monoids and Cayley representations at the level of monoidal categories, and we show that their concretisation results in useful constructions for monads, applicative functors, and arrows. Moreover, by taking advantage of the uniform presentation of the three notions of computation, we introduce a principled approach to 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, 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 monoids in monoidal categories. 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 related to 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

* This work was partially funded by the Agencia Nacional de Promoción Científica y Tecnológica (PICT 2009-15) and Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).

with respect to the Day convolution (Day, 1970), and hence applicative functors are also monoids in a monoidal category of endofunctors using the Day convolution as a tensor.

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 so that arrows are exactly monoids in a monoidal category. 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 (Swierstra & Altenkirch, 2007). 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, 2014). The free arrow, on the other hand, has been less well 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 and obtain 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, 1986), 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 novel transformations for applicative functors and arrows by analysing their Cayley representations.

Given the three notions of computation, one may ask what are the relations between them. Lindley *et al.* (2011) address this question by studying the equational theories induced by calculi capturing each notion of computation. If, on the other hand, we want to address the question by taking a categorical approach, one should study the relation between the different categories of monads, applicative functors, and arrows. Since the three notions are monoids in a monoidal category, this is the same as studying the relation between the corresponding categories of monoids. However, as a consequence of having a unified view, we can ask a simpler, more basic question instead and analyse the relation between the different monoidal categories that give support to monoids. 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 most of these results are known in other communities, the case of the applicative functors as monoids seems to have been overlooked in the functional programming community, and in the case of arrows, the existing models were not an exact fit.
- 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 incorporating their strength in the supporting monoidal category. In previous approaches, such strength was an extra operation attached 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.
- We give canonical constructions for converting profunctors into strong profunctors, one which can be used to convert weak arrows (arrows without the operation first) into arrows.

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 in 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 weak arrows, their free constructions, and their Cayley representations. In Section 7, we turn to arrows, and construct free arrows. Finally, in Section 8, we analyse the relation between the different monoidal categories considered in the previous sections, provide canonical constructions for adding a strength to profunctors, and obtain a representation for arrows. We conclude in Section 9 where we summarise our results and discuss related work.

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

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

In frames like the one surrounding this paragraph, we include implementations in Haskell 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. Nevertheless, one can prove that the implementation of the different concepts is correct using "fast and loose" reasoning (Danielsson *et al.*, 2006).

An extended version of the article that includes proofs and additional technical details is available from the authors' web pages.

1.1 Cayley representation for monoids

We start by stating the Cayley representation theorem for ordinary monoids, i.e., monoids in the category Set of sets and functions. A monoid is a triple $(M, \oplus, e)$ of a set M, a binary operation $\oplus : M \times M \to M$ that is associative ($(a \oplus b) \oplus c = a \oplus (b \oplus c)$), and an element $e \in M$ that 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 binary operation $\oplus$ and the element e are often called the *multiplication* and *unit* 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. The existence of such an i 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:

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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

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:

```
type EList a = [a] → [a]
```

The functions rep and abs, following the proof of Theorem 1.1, are

```
rep    :: [a] → EList a
rep xs = (λys → xs ++ ys)
abs    :: EList a → [a]
abs xs = xs []
```

By the theorem above, we have that $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$. The type EList a is no other than difference lists (Hughes, 1986). Concatenation for standard lists is slow, as it is linear in 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. For difference lists, concatenation is implemented by function composition, and the empty list is implemented by the identity function. Hence, we can perform efficient concatenations on difference lists, and when we are done we can recover standard lists by applying to the empty list.

2 Monoidal categories

The notion of monoid in the category Set of sets and functions is too restrictive for expressing monads, applicative functors, and arrows, 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 we will abstract the product by a $\otimes$ operation called a *tensor*, and the unit 1 by an object I of $\mathscr{C}$.

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

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 the following diagrams commute:

$$\begin{array}{ccccc}
A \otimes (B \otimes (C \otimes D)) & \xrightarrow{\alpha} & (A \otimes B) \otimes (C \otimes D) & \xrightarrow{\alpha} & ((A \otimes B) \otimes C) \otimes D \\
\downarrow{\scriptstyle \mathsf{id}\otimes\alpha} & & & & \uparrow{\scriptstyle \alpha\otimes\mathsf{id}} \\
A \otimes ((B \otimes C) \otimes D) & & \xrightarrow{\alpha} & & (A \otimes (B \otimes C)) \otimes D
\end{array}$$

$$\begin{array}{ccc}
A \otimes (I \otimes B) & \xrightarrow{\alpha} & (A \otimes I) \otimes B \\
& {\scriptstyle \mathsf{id}\otimes\lambda} \searrow \quad \swarrow {\scriptstyle \rho\otimes\mathsf{id}} & \\
& A \otimes B &
\end{array}$$

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 (Mac Lane, 1971).

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 $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$. A *right exponential* $-^A$ is the right adjoint to $- \otimes A$. That is, the right exponential to A is characterised by an isomorphism

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

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 right exponential. Note that $\lfloor \cdot \rfloor$ generalises currying, and $\lceil \cdot \rceil$ generalises uncurrying.

Similarly, a *left exponential* is the right adjoint to $A \otimes -$. In the rest of the paper, we consider only right exponentials, and call them simply *exponential*s. 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.

When working on the category Set, we will write the exponential B^A simply as $A \to B$, which coincides with the set $\mathsf{Set}(A, B)$ of morphisms between A and B.

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

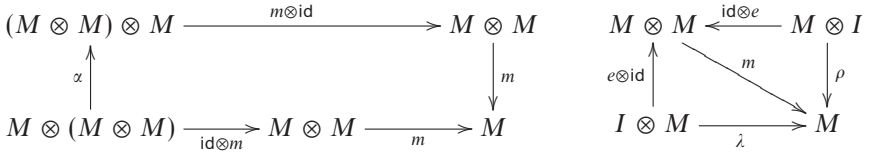
2.1 Monoids in monoidal categories

With the definition of monoidal category in place we may define monoids.

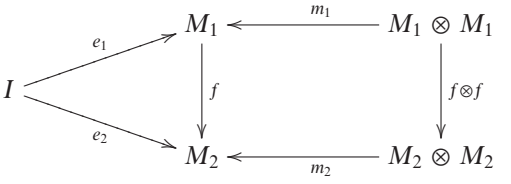
Definition 2.3 (*Monoid*)
A *monoid* in a monoidal category $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$ is a tuple (M, m, e) where $M \in \mathscr{C}$ and m and e are morphisms in $\mathscr{C}$

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

such that the following diagrams commute:



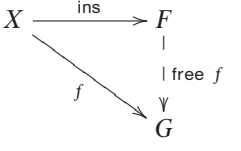
Given two monoids (M_1, m_1, e_1) and (M_2, m_2, e_1), a *monoid homomorphism* between them is an arrow $f : M_1 \to M_2$ in $\mathscr{C}$ such that the following diagram commutes:



Monoids in a monoidal category $\mathscr{C}$ together with monoid homomorphisms form the category $\mathsf{Mon}(\mathscr{C})$.

For ordinary monoids one has the notion of free monoid over a set X, which consists of the set of words (or, equivalently lists) over X. This notion can be generalised to monoidal categories as follows.

Definition 2.4 (*Free monoid*)
Let $(\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$ be a monoidal category. The *free monoid* over 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 commutes:



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

There is a forgetful functor $U : \mathsf{Mon}(\mathscr{C}) \to \mathscr{C}$ that forgets the monoid structure and maps a monoid (M, m, e) to M. When the left-adjoint $(-)^*$ to U 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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

importance to us is the following proposition, which generalises the construction of free monoid in Set as the set of words.

Proposition 2.5
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 for each A the monoid A^* exists and its carrier is the carrier of the initial algebra, which we write as $\mu X. I + A \otimes X$.

In the proposition, the exponentials play a fundamental role, as they are needed to define the multiplication on the carrier. More explicitly, the isomorphism that characterises exponentials allows us to define the multiplication of the monoid by giving a morphism with domain A^*:

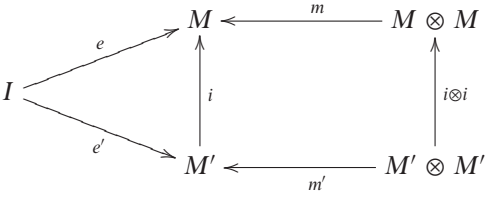
$$m : A^* \otimes A^* \to A^* \quad = \quad \lceil h : A^* \to A^{*A^*} \rceil$$

Because A^{*A^*} carries an $(I + A \otimes -)$-algebra structure, we can define the morphism h using initiality of A^*.

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 2.5 for the case of Set monoids, where $\otimes$ is pairing, and I is the unit type, we obtain lists.

```
data List a = Nil | Cons (a, List a)
```

Definition 2.6 (*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



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. Equivalently, a sub-monoid of M is given by a monic monoid homomorphism.

2.2 Cayley representation of a monoid

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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

Definition 2.7 (*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 following diagram:

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

where

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

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

Here, i_A stands for identity and c_A for composition.

The Cayley representation theorem tells 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 2.8 (*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, rep is split monic with left inverse abs (i.e. $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}_M$) given by

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

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

For any two categories $\mathscr{C}$ and $\mathscr{D}$ we have a category $[\mathscr{C}, \mathscr{D}]$ with functors from $\mathscr{C}$ to $\mathscr{D}$ as objects and natural transformations as morphisms. Therefore, endofunctors on Set form the category $[\mathsf{Set}, \mathsf{Set}]$.

Endofunctors are implemented in Haskell by the following type class:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Natural transformations are implemented by the following type:

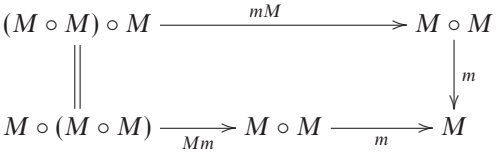
```
type f ·→ g = ∀x. f x → g x
```

In Section 4, we will explain why this is a good implementation of natural transformations.

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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

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



commute.

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

The monoidal structure of $\mathsf{Endo}_\circ$ is given in terms of the identity functor and the composition of functors, which are implemented in Haskell by the datatypes:

```
data Id a = Id a
data (f ∘ g) a = Comp (f (g a))
```

with the obvious `Functor` instances. A monoid in $\mathsf{Endo}_\circ$ is implemented by a `Functor` m, a multiplication $m \circ m \xrightarrow{\cdot} m$ and a unit $\mathsf{Id} \xrightarrow{\cdot} m$. We capture these requirements in the type class `Triple` where, for ease of use, we have unfolded the definitions of natural transformation, identity functor, and functor composition.

```
class Functor m ⇒ Triple m where
  η    :: a → m a
  join :: m (m a) → m a
```

We have called the type class `Triple` in order not to clash with standard nomenclature in Haskell 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 $(\gg\!= f) = \mathsf{join} \circ \mathsf{fmap}\, f$.

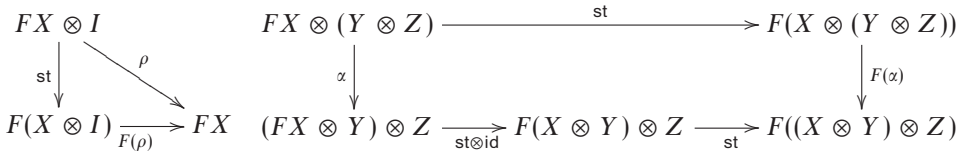
Monads for notions of computation should be *strong* (Moggi, 1989). In general, a functor is said to be strong when it interacts coherently with the monoidal structure.

Definition 3.1
An endofunctor $F : \mathscr{C} \to \mathscr{C}$ is *strong* when it comes equipped with a natural transformation

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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

called a *strength*, such that the following diagrams commute:



All endofunctors on the (cartesian) monoidal category Set come with a unique strength, so all functors in [Set, Set] are strong. As we are always interested in this kind of functors, we do not mention the strength explicitly.

The Haskell implementation of the unique strength for functors is the following:

```
st :: Functor f ⇒ f a → b → f (a, b)
st v b = fmap (λa → (a, b)) v
```

A monad is said to be strong when the monadic structure interacts coherently with the strength. In the case of functors in [Set, Set], we get this coherence for free.

3.1 *Exponentials in* $\mathsf{Endo}_\circ$

Finding an exponential in $\mathsf{Endo}_\circ$ means finding a functor $(-)^F$, such that we have an isomorphism natural in G and H:

$$\mathsf{Nat}(H \circ F, G) \quad \cong \quad \mathsf{Nat}(H, G^F) \tag{1}$$

where we write $\mathsf{Nat}(F, G)$ instead of $[\mathscr{C}, \mathscr{D}](F, G)$ for the collection of natural transformations between F and G.

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

Theorem 3.2 (*Yoneda*)
Let $\mathscr{C}$ be a *locally small category* (i.e., a category such that the collection of morphisms between any two objects is a set). Then, there is an isomorphism

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

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 $\mathscr{C}(X, -)$ 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:

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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

where the first isomorphism is by Yoneda, and the second is by Equation (1). Therefore, whenever the expression $\mathsf{Nat}(X \to F-, G)$ 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 $X \to F-$ 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 that have a particular size restriction (they must be a left Kan extension along the inclusion from a small subcategory.) For example, container functors (Abbott *et al*., 2003) and finitary functors (Kelly & Power, 1993) are small. Intuitively, this means that a small functor is essentially a functor from a small category, and therefore certain size problems do not occur. If F is a small functor, then $\mathsf{Nat}(F, G)$ is a set, and by the reasoning above the functor F is an exponent in $\mathsf{Endo}_\circ$, with exponential $(-)^F$ given by

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

Remark 3.3
Equation (1) means that the exponential $(-)^F$ is a right adjoint to the functor $(- \circ F)$. This exponential is known as the right Kan extension along F.

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

```
data Exp f g x = Exp (∀y. (x → f y) → g y)
```

The components of isomorphism 1 are

```
⌊·⌋   :: Functor h ⇒ (∀x. h (f x) → g x) → h y → Exp f g y
⌊t⌋ y = Exp (λk → t (fmap k y))
⌈·⌉   :: (∀y. h y → Exp f g y) → h (f x) → g x
⌈t⌉ x = let Exp g = t x in g id
```

3.2 Free monads

A finitary endofunctor on Set is an endofunctor whose image is described by its action on finite sets. A finitary endofunctor is a particular kind of small functor, as it is equivalent to a left Kan extension of its domain restriction to the category of finite sets along the inclusion. By restricting $\mathsf{Endo}_\circ$ to finitary endofunctors, we obtain the locally small, right-closed monoidal category $\mathsf{Endo}_\circ^{\mathsf{Fin}}$ (Kelly & Power, 1993). In this category, we may apply Proposition 2.5 and obtain the usual formula for the free monad of an endofunctor F.

$$F^* \;\cong\; \mathsf{Id} + F \circ F^*$$

The same formula instantiated on an object X yields:

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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

The formula above can be readily implemented by the datatype

```
data Free∘ f x = Ret x
               | Con (f (Free∘ f x))
```

with monad instance

```
instance Functor f ⇒ Monad (Free∘ f) where
  return x          = Ret x
  (Ret x)  >>= f = f x
  (Con m) >>= f = Con (fmap (>>= f) m)
```

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

```
ins :: Functor f ⇒ f ·→ Free∘ f
ins x = Con (fmap Ret x)
free :: (Functor f, Monad m) ⇒ (f ·→ m) → (Free∘ f ·→ m)
free f (Ret x) = return x
free f (Con t) = join (f (fmap (free f) t))
```

3.3 Cayley representation of monads

For an exponent F, we may apply Theorem 2.8 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 is called the *codensity monad* on F (Mac Lane, 1971; Jaskelioff, 2009).

The codensity monad is implemented by the following datatype:

```
type Rep f = Exp f f

instance Monad (Rep f) where
  return x          = Exp (λh → h x)
  (Exp m) >>= f = Exp (λh → m (λx → let Exp t = f x in t h))
```

The definition follows from the general definition of monoid of endomorphisms. The morphisms converting from a monad m to Rep m and back are the following:

```
rep :: Monad m ⇒ m x → Rep m x
rep m = Exp (λk → m >>= k)

abs :: Monad m ⇒ Rep m x → m x
abs (Exp m) = m return
```

By Theorem 2.8, we know that abs∘rep = id, and that abs is a monad morphism. Hence, we may change the representation of monadic computations on m, and perform computations on Rep m. This change of representation is exactly the optimisation introduced by Voigtländer (2008) and shown correct by Hutton *et al.* (2010).

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

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, a special type of colimit called *coend*. These concepts will be instrumental in the development of the next sections.

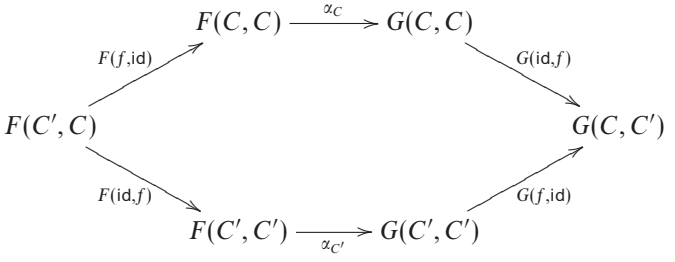
4.1 Ends

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



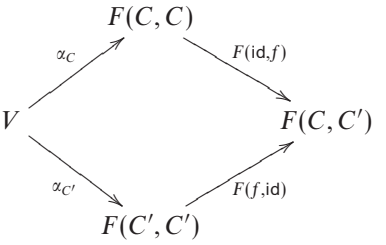
An important difference between natural transformations and dinatural transformations is that the latter cannot be composed in general.

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

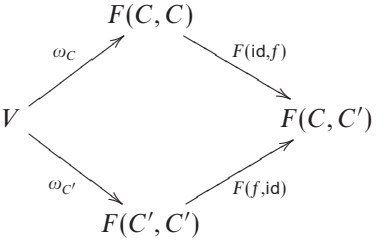
https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

$f : C \to C'$ the following diagram commutes:



In the same way, a limit is a final cone, and *end* is 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



commutes for each $f : C \to C'$, and such that for every wedge from $V' \in \mathscr{D}$, given by a family of morphisms $\gamma_C : V' \to F(C, C)$, there exists a unique morphism $\langle \gamma \rangle : V' \to V$ such that $\omega_C \circ \langle \gamma \rangle = \gamma_C$.

The object V is usually denoted by $\int_X F(X, X)$ and referred to as "the end of F". The universal property of ends tell us that each morphism into an end is in a one-to-one correspondence with a dinatural family of morphisms:

$$\frac{\langle \gamma \rangle : Y \to \int_X F(X, X)}{\gamma_X : Y \to F(X, X), \text{ dinatural in } X}$$

Let $F, G : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathsf{Set}$, with $\mathscr{C}$ a small category. If we denote the dinatural transformations between F and G by $\mathsf{Dinat}(F, G)$, then

$$\mathsf{Dinat}(F, G) \quad \cong \quad \int_X F(X, X) \to G(X, X)$$

Natural transformations are a particular instance of dinatural transformations. More concretely, 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

$$\mathsf{Nat}(F, G) \quad \cong \quad \int_X FX \to GX$$

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

One nice feature of ends is that they lead to a natural implementation of categorical concepts in Haskell by replacing the end by a universal quantifier (Bainbridge *et al.*, 1990). For example, the class of natural transformations between functors F and G is

$$\int_X FX \to GX$$

By implementing this end as a universal quantifier, we obtain the type constructor of natural transformations that was introduced in Section 3:

$$\textbf{type}\ f \xrightarrow{\bullet} g = \forall x.\, f\ x \to g\ x$$

4.2 Coends

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

Definition 4.4
A *cowedge* from F is an object V together with a dinatural transformation $\alpha : F \to \Delta_V$.

Definition 4.5
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 $\iota_C \circ F(f, \mathsf{id}) = \iota_{C'} \circ F(\mathsf{id}, f)$ for each $f : C \to C'$, which is universal with respect to this property: for every cowedge given by an object V' and a family of morphisms $\gamma_C : F(C, C) \to V'$, there exists a unique morphism $[\gamma] : V \to V'$ such that $\gamma_C = [\gamma] \circ \iota_C$.

The object V is usually denoted by $\int^X F(X, X)$ and referred to as "the coend of F". The universal property of coends tell us that each morphism out of a coend is in a one-to-one correspondence with a family of dinatural morphisms:

$$\frac{[\gamma] : \int^X F(X, X) \to Y}{\gamma_X : F(X, X) \to Y,\ \text{dinatural in } X}$$

In the same way, an end can be implemented as a universal quantifier, and a coend can be implemented as an existential quantifier.

4.3 Yoneda Lemma in end and coend forms

We can express the Yoneda Lemma using ends and coends (Day & Kelly, 1969):

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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

The end form and coend form of the Yoneda lemma lead to straightforward implementations in Haskell. The components of the Yoneda isomorphism in end form are implemented as a polymorphic function between types $f\ x$ and $\forall y.\ (x \to y) \to f\ y$:

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

Similarly, its coend form (also known as "coYoneda Lemma") is expressed by

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

5 Applicative functors as monoids

Similarly to monads, applicative functors (McBride & Paterson, 2008) are a class of functors used to write effectful computations. 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 that are not monads. The main difference between monads and applicative functors is that the latter do not allow effects to depend on previous values, i.e., the effects are fixed beforehand.

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

$$\begin{aligned}
&\textbf{class}\ \mathsf{Functor}\ f \Rightarrow \mathsf{Applicative}\ f\ \textbf{where}\\
&\quad \mathsf{pure} :: x \to f\ x\\
&\quad (\circledast) \quad :: f\ (x \to y) \to f\ x \to f\ y
\end{aligned}$$

Since their introduction, applicative functors have been characterised categorically as *strong lax monoidal functors* (McBride & Paterson, 2008). In Section 3, we already explained the notion of *strength* for a functor. Now, we explain what is a *lax monoidal functor*. 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* $F : \mathscr{C}_1 \to \mathscr{C}_2$ is a functor between the underlying categories of two monoidal categories $(\mathscr{C}_1, \otimes, I_1, \alpha_1, \lambda_1, \rho_1)$ and $(\mathscr{C}_2, \oplus, I_2, \alpha_2, \lambda_2, \rho_2)$ together with a natural transformation

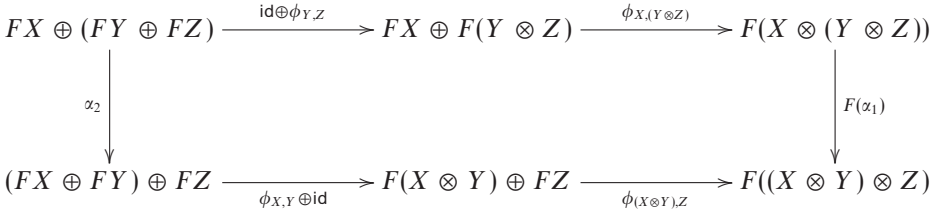
$$\phi_{X,Y} : FX \oplus FY \to F(X \otimes Y)$$

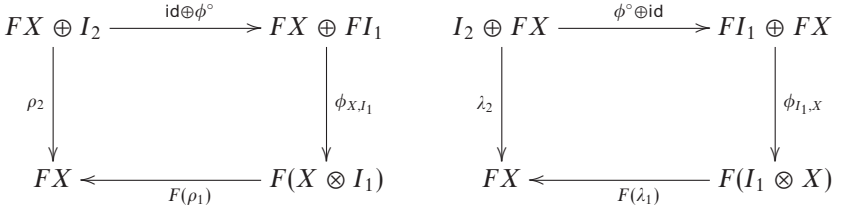
https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

and a morphism

$$\phi^\circ : I_2 \to FI_1$$

such that the following diagrams commute:





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

A *strong lax monoidal functor* is simply a lax monoidal functor that 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 x, f y) → f (x, y)
```

We saw in Section 3 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 X in $\mathscr{C}$, the Day convolution $(F \star G)X$ is a new object in $\mathscr{C}$ defined as

$$(F \star G)X \;=\; \int^{Y,Z} FY \times GZ \times X^{Y \times Z}$$

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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

The mapping of objects $F \star G$ extends to a functor. Moreover, the Day convolution is 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 (Peyton Jones *et al.*, 2006), the type variables y and z are existentially quantified:

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

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 is useful for writing morphisms from the convolution of two functors onto another object.

Proposition 5.2
There is a one-to-one correspondence defining morphisms going out of a Day convolution

$$\int_X (F \star G)X \to HX \quad \cong \quad \int_{Y,Z} (FY \times GZ) \to H(Y \times Z) \tag{2}$$

which is natural in F, G, and H, and the morphisms witnessing the isomorphism can be written using the universal property of ends.

Remark 5.3 (*Day convolution as a left Kan extension*)
In view of Proposition 5.2, $F \star G$ is the left Kan extension of $\bar{\times} \circ (F \times G)$ along $\bar{\times}$, where $\bar{\times} : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$ is the functor that takes an object (X, Y) of the product category into a product of objects $X \times Y$.

Proposition 5.2 shows an equivalence between the type $(f \star g) \dot{\to} h$ and the type $\forall y\ z.\ (f\ y, g\ z) \to h\ (y, z)$:

```
ϑ :: (f ⋆ g ↛ h) → (f y, g z) → h (y, z)
ϑ f (x, y) = f (Day x y id)
ϑ⁻¹ :: Functor h ⇒ (∀y z. (f y, g z) → h (y, z)) → (f ⋆ g ↛ h)
ϑ⁻¹ g (Day x y f) = fmap f (g (x, y))
```

In contrast to the composition tensor, the Day convolution is not strict. Moreover, since we have an isomorphism $\gamma : (F \star G)X \to (G \star F)X$ natural in F, G, and X, the Day convolution is a symmetric tensor. This means that, together with appropriate natural transformations α, λ, and ρ, $\mathsf{Endo}_\star = ([\mathsf{Set}, \mathsf{Set}]_S, \star, \mathsf{Id}, \alpha, \lambda, \rho, \gamma)$ is a symmetric monoidal category (Day, 1970).

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

Here, we present the natural isomorphisms of the monoidal category $\mathsf{Endo}_\star$. One direction is given by the following natural transformations:

$$
\begin{aligned}
&\lambda :: \mathsf{Functor}\ f \Rightarrow \mathsf{Id} \star f \xrightarrow{\bullet} f \\
&\lambda\ (\mathsf{Day}\ (\mathsf{Id}\ x)\ y\ f) = \mathsf{fmap}\ (f \circ (\lambda y \to (x, y)))\ y \\
&\rho :: \mathsf{Functor}\ f \Rightarrow f \star \mathsf{Id} \xrightarrow{\bullet} f \\
&\rho\ (\mathsf{Day}\ x\ (\mathsf{Id}\ y)\ f) = \mathsf{fmap}\ (f \circ (\lambda z \to (z, y)))\ x \\
&\alpha :: f \star (g \star h) \xrightarrow{\bullet} (f \star g) \star h \\
&\alpha\ (\mathsf{Day}\ x\ (\mathsf{Day}\ y\ z\ f)\ g) = \mathsf{Day}\ (\mathsf{Day}\ x\ y\ f_1)\ z\ f_2 \\
&\quad \textbf{where}\ f_1 = \lambda(c, e) \to (c, \lambda h \to f\ (e, h)) \\
&\qquad\quad f_2 = \lambda((c, h), d) \to g\ (c, h\ d) \\
&\gamma :: (f \star g) \xrightarrow{\bullet} (g \star f) \\
&\gamma\ (\mathsf{Day}\ x\ y\ f) = \mathsf{Day}\ y\ x\ (f \circ \mathsf{swap}) \\
&\quad \textbf{where}\ \mathsf{swap}\ (x, y) = (y, x)
\end{aligned}
$$

Their respective inverses are defined as

$$
\begin{aligned}
&\lambda^{-1} :: \mathsf{Functor}\ f \Rightarrow f \xrightarrow{\bullet} \mathsf{Id} \star f \\
&\lambda^{-1}\ x = \mathsf{Day}\ (\mathsf{Id}\ ())\ x\ \mathsf{snd} \\
&\rho^{-1} :: \mathsf{Functor}\ f \Rightarrow f \xrightarrow{\bullet} f \star \mathsf{Id} \\
&\rho^{-1}\ x = \mathsf{Day}\ x\ (\mathsf{Id}\ ())\ \mathsf{fst} \\
&\alpha^{-1} :: (f \star g) \star h \xrightarrow{\bullet} f \star (g \star h) \\
&\alpha^{-1}\ (\mathsf{Day}\ (\mathsf{Day}\ x\ y\ f)\ z\ g) = \mathsf{Day}\ x\ (\mathsf{Day}\ y\ z\ f_1)\ f_2 \\
&\quad \textbf{where}\ f_1 = \lambda(d, b) \to ((\lambda c \to f\ (c, d)), b) \\
&\qquad\quad f_2 = \lambda(c, (h, b)) \to g\ (h\ c, b)
\end{aligned}
$$

Remark 5.4 (*Alternative presentations of the Day convolution*)
In our setting of Set endofunctors, the Day convolution has different alternative representations (Day, 1970):

$$(F \star G)X \quad \cong \quad \int^Y FY \times G(Y \to X) \quad \cong \quad \int^Y F(Y \to X) \times GY \tag{3}$$

The equivalences essentially follow from Yoneda and cartesian closure.

The corresponding implementations of the two alternative representations are

$$
\begin{aligned}
&\textbf{data}\ (f \star_1 g)\ x\ \textbf{where} \\
&\quad \mathsf{Day}_1 :: f\ y \to g\ (y \to x) \to (f \star_1 g)\ x \\
&\textbf{data}\ (f \star_2 g)\ x\ \textbf{where} \\
&\quad \mathsf{Day}_2 :: f\ (y \to x) \to g\ y \to (f \star_2 g)\ x
\end{aligned}
$$

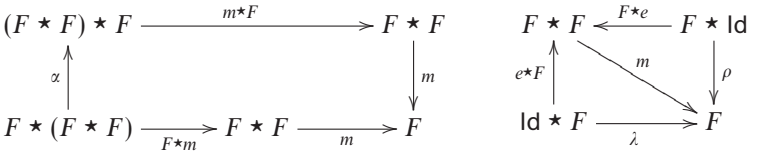
In these two definitions, the type variable y is existentially quantified.

5.1 *Monoids in* $\mathsf{Endo}_\star$

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

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

- 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:



From the unit e, one can consider the component $e_1 : 1 \to F1$. This component defines a mapping that can be used as the unit morphism for a lax monoidal functor. Similarly, using Equation (2), the morphism $m : F \star F \to F$ is equivalent to a family of morphisms

$$\vartheta(m)_{X,Y} : FX \times FY \to F(X \times Y)$$

which is natural in X and Y. This family of morphisms corresponds to the multiplicative transformation in a lax monoidal functor. Putting together F, $\vartheta(m)$ and e_1, we obtain a strong lax monoidal functor on Set. That is, we obtain an applicative functor.

It remains to be seen if the converse is true: can a monoid in $\mathsf{Endo}_\star$ be defined from an applicative functor? Given an applicative functor (F, ϕ, ϕ°), it easy to see that a multiplication for the monoid can be given from ϕ, using Equation (2) again. What remains to be seen is if we can recover the whole natural transformation $e : \mathsf{Id} \to F$ out of only one component $\phi^\circ : 1 \to F1$. We do so by using the strength of F (which exists since it is an endofunctor on Set): the natural transformation e is recovered by the following composition:

$$X \xrightarrow{\langle !, \mathsf{id} \rangle} 1 \times X \xrightarrow{\phi^\circ \times \mathsf{id}} F1 \times X \xrightarrow{\mathsf{st}_{1,X}} F(1 \times X) \xrightarrow{F\pi_2} FX$$

which defines a morphism $e_X : X \to FX$ for each X.

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

5.2 *Exponentials in* $\mathsf{Endo}_\star$

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

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

Therefore, whenever the last expression makes sense, it can be used as the definition of the exponential object. Since we are working in a category of small functors, the expression always makes sense and the exponential is always guaranteed to exist.

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

Using Proposition 5.2 and Yoneda, an alternative form for G^F can be derived (Day, 1973):

$$G^F X \quad \cong \quad \mathsf{Nat}(F, G(X \times -)) \quad \cong \quad \int_Y FY \to G(X \times Y)$$

Using Haskell, this exponential can be represented as

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

The components of the isomorphism showing it is an exponential are

$$\begin{aligned}
&\lfloor\cdot\rfloor :: (f \star g \xrightarrow{\bullet} h) \to f \xrightarrow{\bullet} \mathsf{Exp}\ g\ h\\
&\lfloor m \rfloor\ x = \mathsf{Exp}\ (\lambda y \to m\ (\mathsf{Day}\ x\ y\ \mathsf{id}))\\
&\lceil\cdot\rceil :: \mathsf{Functor}\ h \Rightarrow (f \xrightarrow{\bullet} \mathsf{Exp}\ g\ h) \to f \star g \xrightarrow{\bullet} h\\
&\lceil f \rceil\ (\mathsf{Day}\ x\ y\ h) = \mathsf{fmap}\ h\ (t\ y)\\
&\qquad\qquad\qquad\qquad \textbf{where}\ \mathsf{Exp}\ t = f\ x
\end{aligned}$$

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

5.3 Free applicative functor

By Proposition 2.5, the free monoid, viz. the free applicative functor, exists.

The direct application of Proposition 2.5 yields the following implementation of the free applicative functor:

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

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

$$\begin{aligned}
&\textbf{data}\ \mathsf{Free}_\star\ f\ a\ \textbf{where}\\
&\quad \mathsf{Pure} :: x \to \mathsf{Free}_\star\ f\ x\\
&\quad \mathsf{Rec}\ \ :: f\ y \to \mathsf{Free}_\star\ f\ z \to ((y, z) \to x) \to \mathsf{Free}_\star\ f\ x
\end{aligned}$$

with the following instances:

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

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

$$\begin{aligned}
&\mathsf{ins} :: \mathsf{Functor}\ a \Rightarrow f \xrightarrow{\bullet} \mathsf{Free}_\star\ f\\
&\mathsf{ins}\ x = \mathsf{Rec}\ x\ (\mathsf{Pure}\ ())\ \mathsf{fst}
\end{aligned}$$

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

```
free :: (Functor f, Applicative g) ⇒ (f ⇢ g) → (Free⋆ f ⇢ g)
free f (Pure x)     = pure x
free f (Rec x y g) = pure (curry g) ⊛ f x ⊛ free f y
```

The alternative presentations of the Day convolution of Equation (3) result in the alternative types $\star_1$ and $\star_2$. Using these types instead of $\star$ in the definition of the free applicative functor results in two alternative definitions:

```
data Free′⋆ f x where
  Pure′ :: x → Free′⋆ f x
  Rec′  :: f y → Free′⋆ f (y → x) → Free′⋆ f x
data Free″⋆ f x where
  Pure″ :: x → Free″⋆ f x
  Rec″  :: f (y → x) → Free″⋆ f y → Free″⋆ f x
```

Hence, the two alternative presentations of the Day convolution given in Equation (3) give rise to the two notions of free applicative functor found by Capriotti and Kaposi (2014).

5.4 Cayley representation for applicative functors

Having found the exponentials in $\mathsf{Endo}_\star$, we may apply Theorem 2.8 and construct the corresponding Cayley representation.

The Cayley representation of an applicative functor is the exponential of the 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)
```

The Applicative instance is obtained from the general construction of the monoid of endomorphisms. Finally, from Theorem 2.8, we obtain the applicative morphism rep and the natural transformation abs, together with the property that abs ∘ rep = 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 ()))
```

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

6 Weak arrows as monoids

Having successfully fit both monads and applicative functors as monoids in a monoidal category, we now focus on a third popular notion of computation: arrows.

Arrows (Hughes, 2000) are a generalisation of monads that can offer standardised interfaces to libraries that are incompatible with the monadic interface, including parsers with static analysis, quantum computing (Vizzotto *et al.*, 2006), secure information flow (Li & Zdancewic, 2010), and functional reactive programming (Hudak *et al.*, 2003).

Asada (2010) characterised arrows as strong monads in a bicategory. Closer to our intentions of working with monoids in a monoidal category, Jacobs *et al.* (2009) showed that weak arrows (arrows without the operation first) are monoids in the category of profunctors. They recover (strong) arrows by adding the strength on top of the monoid structure.

We briefly review the results by Jacobs *et al.* on weak arrows and then proceed to obtain free weak arrows, and a Cayley representation for weak arrows.

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, profunctors are to relations what functors are to functions. A morphism between two profunctors is a natural transformation between the profunctors considered as functors.

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

```
class Profunctor p where
  dimap :: (x' → x) → (y → y') → p x y → p x' y'
```

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 in 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 transformations 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 gives a notion of tensor.

Definition 6.2 (*Profunctor tensor (Bénabou, 1973)*)
Given two profunctors $P, Q : \mathscr{C} \nrightarrow \mathscr{C}$, their composition is

$$(P \otimes Q)(X, Y) = \int^{Z} P(X, Z) \times Q(Z, Y)$$

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

The profunctor tensor is implemented in Haskell as follows:

```
data (p ⊗ q) x y where
  PCom :: p x z → q z y → (p ⊗ q) x y
instance (Profunctor p, Profunctor q) ⇒ Profunctor (p ⊗ q) where
  dimap m1 m2 (PCom p q) = PCom (dimap m1 id p) (dimap id m2 q)
```

This tensor is analogous to the composition of relations, replacing the existential quantification by a coend. The functor $\mathsf{Hom} : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathsf{Set}$ mapping two objects to the set of morphisms between them is a small functor and it is the unit for profunctor composition:

$$(P \otimes \mathsf{Hom})(X, Y) \quad = \quad \int^{Z} P(X, Z) \times (Z \to Y) \quad \cong \quad P(X, Y)$$

The equality holds by definition of profunctor composition, and the isomorphism holds by the Yoneda Lemma. Thus, we may define a natural isomorphism $\rho : P \otimes \mathsf{Hom} \cong P$. Analogously, we can define the other two natural isomorphisms $\lambda : \mathsf{Hom} \otimes P \cong P$ and $\alpha : P \otimes (Q \otimes R) \cong (P \otimes Q) \otimes R$ and obtain a monoidal structure for $[\mathscr{C}^{\mathsf{op}} \times \mathscr{C}, \mathsf{Set}]$, with profunctor composition $\otimes$ as its tensor, and the Hom functor as its unit. We denote this monoidal category by Pro.

We have shown how to implement the objects of Pro as instances of the type class ProFunctor. Morphisms between profunctors are implemented as

```
type p ⇸ q = ∀x y. p x y → q x y
```

The unit Hom is simply the type of functions:

```
type Hom = (→)
```

The natural isomorphisms λ, ρ, and α are implemented as

```
λ :: Profunctor p ⇒ Hom ⊗ p ⇸ p
λ (PCom f x) = dimap f id x
ρ :: Profunctor p ⇒ p ⊗ Hom ⇸ p
ρ (PCom x f) = dimap id f x
α :: p ⊗ (q ⊗ r) ⇸ (p ⊗ q) ⊗ r
α (PCom p (PCom q r)) = PCom (PCom p q) r
```

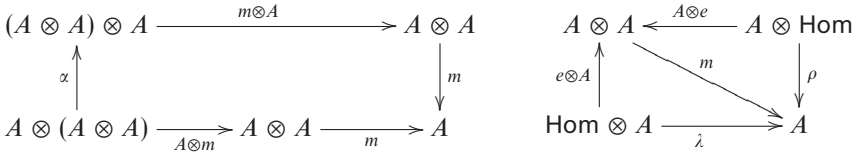
with inverses

```
λ⁻¹ :: Profunctor p ⇒ p ⇸ Hom ⊗ p
λ⁻¹ f = PCom id f
ρ⁻¹ :: Profunctor p ⇒ p ⇸ p ⊗ Hom
ρ⁻¹ f = PCom f id
α⁻¹ :: (p ⊗ q) ⊗ r ⇸ p ⊗ (q ⊗ r)
α⁻¹ (PCom (PCom p q) r) = PCom p (PCom q r)
```

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

What 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



commute.

Using the isomorphism

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

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

This presentation leads naturally to the following implementation of monoids in the monoidal category Pro:

```
class Profunctor a ⇒ WeakArrow a where
  arr  :: (x → y) → a x y
  (⋙) :: a x y → a y z → a x z
```

The laws that must hold are

$$\begin{aligned}
(a \ggg b) \ggg c &= a \ggg (b \ggg c)\\
\mathsf{arr}\, f \ggg a &= \mathsf{dimap}\, f\ \mathsf{id}\ a\\
a \ggg \mathsf{arr}\, f &= \mathsf{dimap}\ \mathsf{id}\, f\ a\\
\mathsf{arr}\,(g \circ f) &= \mathsf{arr}\, f \ggg \mathsf{arr}\, g
\end{aligned}$$

6.1 *Exponentials in* Pro

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

$$Q^P(X,Y) = \int_Z P(Y,Z) \to Q(X,Z)$$

The implementation of exponentials in Pro follows the definition above:

```
data Exp p q x y = Exp (∀z. p y z → q x z)
```

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

```
instance (Profunctor p, Profunctor q) ⇒ Profunctor (Exp p q) where
  dimap m1 m2 (Exp pq) = Exp (dimap m1 id ∘ pq ∘ dimap m2 id)
```

The components of the isomorphism that show that Exp is an exponential are

```
⌊·⌋ :: (p ⊗ q ↛ r) → (p ↛ Exp q r)
⌊m⌋ f              = Exp (λg → m (PCom f g))
⌈·⌉ :: (p ↛ Exp q r) → (p ⊗ q ↛ r)
⌈m⌉ (PCom f g) = e g where Exp e = m f
```

6.2 Free weak arrows

By Proposition 2.5, the free monoid, viz. the free weak arrow, exists.

The direct application of Proposition 2.5 yields the following implementation of the free weak arrow:

```
data Free⊗ a x y where
  Hom  :: (x → y) → Free⊗ a x y
  Comp :: a x z → Free⊗ a z y → Free⊗ a x y
```

with the following instances:

```
instance Profunctor a ⇒ Profunctor (Free⊗ a) where
  dimap f g (Hom h)      = Hom (g ∘ h ∘ f)
  dimap f g (Comp x y) = Comp (dimap f id x) (dimap id g y)
instance Profunctor a ⇒ WeakArrow (Free⊗ a) where
  arr f              = Hom f
  (Hom f)     ⋙ c = dimap f id c
  (Comp x y) ⋙ c = Comp x (y ⋙ c)
```

The insertion of generators ins and the universal morphism free from the free weak arrow are

```
ins :: Profunctor a ⇒ a ↛ Free⊗ a
ins x = Comp x (arr id)
free :: (Profunctor a, WeakArrow b) ⇒ (a ↛ b) → (Free⊗ a ↛ b)
free f (Hom g)      = arr g
free f (Comp x y) = f x ⋙ free f y
```

6.3 Cayley representation of weak arrows

Having found the exponentials in Pro, we may apply Theorem 2.8 and construct the corresponding Cayley representation.

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

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

```
type Rep a = Exp a a
instance Profunctor a ⇒ WeakArrow (Rep a) where
  arr f                 = Exp (λy → dimap f id y)
  (Exp f) ⋙ (Exp g) = Exp (λy → f (g y))
```

The instance is derived from the general construction of the monoid of endomorphisms. Finally, from Theorem 2.8, we obtain the weak arrow morphism rep and the natural transformation abs, together with the property that abs ∘ rep = id:

```
rep :: WeakArrow a ⇒ a ⇸ Rep a
rep x = Exp (λy → x ⋙ y)
abs :: WeakArrow a ⇒ Rep a ⇸ a
abs (Exp f) = f (arr id)
```

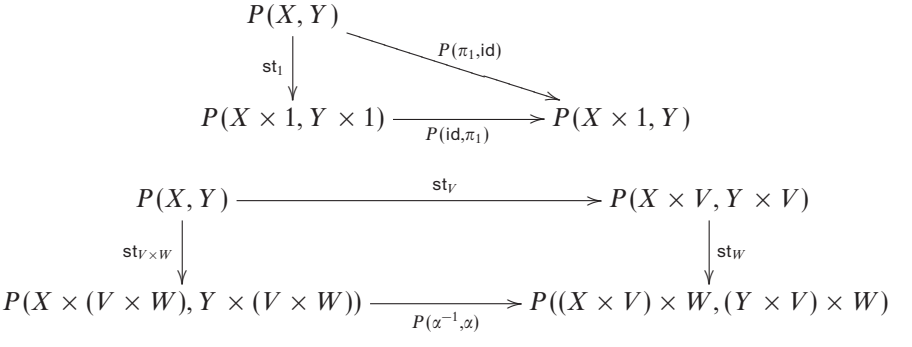
7 Arrows as monoids

Arrows are weak arrows with an additional operation called first. In order to see arrows as monoids, we need to internalise the first operation in the categorical presentation. Jacobs *et al.* (2009) solve this problem by adjoining a first operator to monoids in Pro: an arrow is a monoid (A, m, e) together with a family of morphisms $\mathsf{first} : A(X, Y) \to A(X \times Z, Y \times Z)$. 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. This approach mirrors the manner in which (strong) monads and applicative functors were obtained, and therefore we gain uniformity.

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

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

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



https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

We say that a pair (P, st) is a strong profunctor. The diagrams that must commute here are analogous to those for a tensorial strength of an endofunctor (Definition 3.1).

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:

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

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 when it does, it may not be unique.

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

$$\textbf{data}\ \mathsf{Double}\ x\ y = \mathsf{Double}\ ((x, x) \to (y, y))$$
$$\textbf{instance}\ \mathsf{Profunctor}\ \mathsf{Double}\ \textbf{where}$$
$$\quad \mathsf{dimap}\ f\ g\ (\mathsf{Double}\ h) = \mathsf{Double}\ ((g\ \times\ g) \circ h \circ (f\ \times\ f))$$

There exist two possible instances satisfying the strength axioms:

$$\textbf{instance}\ \mathsf{StrongProfunctor}\ \mathsf{Double}\ \textbf{where}$$
$$\quad \mathsf{first}\ (\mathsf{Double}\ f) = \mathsf{Double}\ g$$
$$\quad\quad \textbf{where}\ g\ ((x, z), (x', z')) = ((y, z), (y', z'))$$
$$\quad\quad\quad\quad \textbf{where}\ (y, y') = f\ (x, x')$$

$$\textbf{instance}\ \mathsf{StrongProfunctor}\ \mathsf{Double}\ \textbf{where}$$
$$\quad \mathsf{first}\ (\mathsf{Double}\ f) = \mathsf{Double}\ g$$
$$\quad\quad \textbf{where}\ g\ ((x, z), (x', z')) = ((y, z), (y', z))$$
$$\quad\quad\quad\quad \textbf{where}\ (y, y') = f\ (x, x')$$

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

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

$$\begin{array}{ccc} P(X, Y) & \xrightarrow{\mathsf{st}^P} & P(X \times Z, Y \times Z) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \alpha} \\ Q(X, Y) & \xrightarrow[\mathsf{st}^Q]{} & Q(X \times Z, Y \times Z) \end{array}$$

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

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 (P, st) as objects, where P is a profunctor and st is a strength for it, and of strong natural transformations as morphisms.

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

We now seek to equip the category of strong profunctors with a monoidal structure. The monoidal structure of Pro can be used for strong profunctors. The unit Hom has an obvious strength. The strength for the composition of two profunctors, however, is a bit more involved. Given two strong profunctors (P, st^P) and (Q, st^Q), one can use the universal property of coends and define the strength of their composition as $\mathsf{st}_Z^{P \otimes Q} = [h]$, where h is a dinatural transformation on V defined as the following composition:

$$\begin{aligned} P(X,V) \times Q(V,Y) &\xrightarrow{\mathsf{st}_Z^P \times \mathsf{st}_Z^Q} P(X \times Z, V \times Z) \times Q(V \times Z, Y \times Z) \\ &\xrightarrow{\iota_{V \times Z}} (P \otimes Q)(X \times Z, Y \times Z) \end{aligned}$$

It is not difficult to verify that such a family is indeed a strength for the profunctor $P \otimes Q$. The monoidal category of strong profunctors with tensor defined in 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 that are weak arrows:

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

Instance declarations of Arrow are empty, but the programmer should check the compatibility of the unit and multiplication of the weak 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, weak arrows, and strength, constitute the arrows laws proposed by Hughes (2000).

7.1 *Exponentials in* SPro

We have not managed to find exponentials in SPro. Part of the difficulty in finding one seems to stem from the fact that strengths for profunctors may not exist, and when they do, they may not be unique. In particular, given two strong profunctors

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

P and Q, the obvious candidate for an exponential in SPro is the exponential in Pro, namely the profunctor Q^P defined in Section 6.1. However, this profunctor does not seem to have a strength.

Fortunately, as shown in Section 8.5, two canonical strong profunctors can be derived from any profunctor. Using one of these, we may lift representations for weak arrows and obtain representations for arrows (as shown in Section 8.6).

7.2 *Free arrows*

Having failed to find exponentials in SPro, we cannot apply Proposition 2.5 to obtain the free monoid in SPro, and therefore we fall back to finding it directly. Fortunately, this is not difficult, 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 weak arrow can be equipped with a strength when defined over a strong profunctor:

```
instance StrongProfunctor a ⇒ StrongProfunctor (Free⊗ a) where
  first (Hom f)      = Hom (λ(x, z) → (f x, z))
  first (Comp x y) = Comp (first x) (first y)
```

Since the unit and multiplication of the free arrow are compatible with the strength, it is a correct Arrow instance:

```
instance StrongProfunctor a ⇒ Arrow (Free⊗ a)
```

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

```
ins :: StrongProfunctor a ⇒ a ⇢ Free⊗ a
ins x = Comp x (arr id)
free :: (StrongProfunctor a, Arrow b) ⇒ (a ⇢ b) → (Free⊗ a ⇢ b)
free f (Hom g)      = arr g
free f (Comp x y) = f x ⋙ free f y
```

Here, we would really like the type $(a \overset{\bullet\bullet}{\to} b)$ to represent strength preserving morphisms between strong profunctors. Therefore, free f is guaranteed to preserve the strength only when f does.

8 On functors between monoidal categories

Monads, applicative functors, 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 starting from a monad we can derive both an applicative functor and an arrow. In this section, we explain these and other

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

derivations from the point of view of monoidal categories. For example, in order to obtain a weak arrow from a monad, we are interested in creating a monoid in Pro, given a monoid in $\mathsf{Endo}_\circ$. 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 $\mathsf{Endo}_\circ$ and Pro), and then use the following theorem to obtain a functor between the corresponding monoids.

Theorem 8.1
Let $(F, \phi, \phi^\circ) : \mathscr{C} \to \mathscr{D}$ be a lax monoidal functor (see definition 5.1). If (M, m, e) is a monoid in $\mathscr{C}$, then $(FM, Fm \circ \phi, Fe \circ \phi^\circ)$ 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 monoidal 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):

$$\textsc{Cayley} : \mathsf{Endo}_\star \to \mathsf{Pro}$$
$$\textsc{Cayley}(F)(X, Y) = F(X \to Y)$$

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

The Cayley functor is monoidal, as shown by Pastro and Street (2008), and therefore by Theorem 8.1 it extends to a functor between the corresponding categories of monoids. That is, it takes applicative functors to weak arrows. Moreover, the functor is also a monoidal functor from $\mathsf{Endo}_\star$ to SPro, as each $\textsc{Cayley}(F)$ has a strength. By Theorem 8.1, the functor also extends to a functor from applicative functors to arrows.

The implementation of the Cayley functor is as follows (McBride & Paterson, 2008):

```
data Cayley f x y = Cayley (f (x → y))
```

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

```
instance Applicative f ⇒ WeakArrow (Cayley f) where
  arr f                          = Cayley (pure f)
  (Cayley x) ⋙ (Cayley y) = Cayley (pure (∘) ⊛ y ⊛ x)

instance Applicative f ⇒ StrongProfunctor (Cayley f) where
  first (Cayley x) = Cayley (pure (λf → λ(y, z) → (f y, z)) ⊛ x)
instance Applicative f ⇒ Arrow (Cayley f)
```

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

8.2 The Kleisli monoidal functor

The well-known Kleisli category of a monad allows us to construct an arrow from any monad. This can be seen as a consequence of applying Theorem 8.1 to the following lax monoidal functor that we call KLEISLI:

$$\begin{aligned}&\textsc{Kleisli} : \mathsf{Endo}_\circ \to \mathsf{SPro}\\ &\textsc{Kleisli}(F)(X,Y) = X \to FY\end{aligned}$$

The implementation of the Kleisli functor is as follows:

```
data Kleisli f x y = Kleisli (x → f y)
instance Monad f ⇒ WeakArrow (Kleisli f) where
  arr f                          = Kleisli (λx → return (f x))
  (Kleisli f) ⋙ (Kleisli g)      = Kleisli (λx → f x >>= g)
instance Monad f ⇒ StrongProfunctor (Kleisli f) where
  first (Kleisli f)              = Kleisli (λ(x, z) → f x >>= λy → return (y, z))
instance Monad f ⇒ Arrow (Kleisli f)
```

8.3 The Day monoidal functor

We equip the identity endofunctor $\mathsf{Id} : [\mathsf{Set},\mathsf{Set}] \to [\mathsf{Set},\mathsf{Set}]$ with monoidal compatibility morphisms ϕ and ϕ°. In this way, $(\mathsf{Id}, \phi, \phi^\circ)$ is a lax monoidal functor from $\mathsf{Endo}_\circ$ to $\mathsf{Endo}_\star$ which we call DAY. The ϕ° morphism is the identity on the identity functor. The morphism $\phi_{F,G} : F \star G \to F \circ G$ is given by

$$\begin{aligned}(F \star G)X &= \int^{Y,Z} FY \times GZ \times (Y \times Z \to X)\\
&\xrightarrow{\int \mathsf{st}} \int^{Y,Z} F(Y \times GZ \times (Y \times Z \to X))\\
&\xrightarrow{\cong} \int^{Y,Z} F(GZ \times Y \times (Y \times Z \to X))\\
&\xrightarrow{\int F\mathsf{st}} \int^{Y,Z} F(G(Z \times Y \times (Y \times Z \to X)))\\
&\xrightarrow{\cong} \int^{Y,Z} F(G(Y \times Z \times (Y \times Z \to X)))\\
&\xrightarrow{\int F(G\,\mathsf{ev})} \int^{Y,Z} F(GX)\\
&\xrightarrow{\cong} F(GX)\end{aligned}$$

Hence, we obtain the lax monoidal functor $\textsc{Day} : \mathsf{Endo}_\circ \to \mathsf{Endo}_\star$.

By applying Theorem 8.1 to DAY, we obtain the well-known result that every monad is an applicative functor:

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

```
instance Monad f ⇒ Applicative f where
  pure  = return
  f ⊛ x = f ≫= (λg → x ≫= return ∘ g)
```

8.4 The reversed monoid

For every monoidal category $\mathscr{C}_{\otimes} = (\mathscr{C}, \otimes, I, \alpha, \lambda, \rho)$, there is a reverse monoidal category $\mathscr{C}_{\otimes^{\mathsf{rev}}}$, with monoidal operator $X \otimes^{\mathsf{rev}} Y = Y \otimes X$. Every monoid in a monoidal category determines a monoid in the reverse monoidal category.

Theorem 8.2
If (M, m, e) is a monoid in $\mathscr{C}_{\otimes}$, then (M, m, e) is a monoid in $\mathscr{C}_{\otimes^{\mathsf{rev}}}$.

In the case where the monoidal structure is symmetric, there is an isomorphism between $X \otimes^{\mathsf{rev}} Y$ and $X \otimes Y$. We can use this isomorphism to equip the identity endofunctor over $\mathscr{C}$ with a monoidal structure, yielding a monoidal functor from $\mathscr{C}_{\otimes}$ to $\mathscr{C}_{\otimes^{\mathsf{rev}}}$.

Theorem 8.3
Let $\mathscr{C}_{\otimes} = (\mathscr{C}, \otimes, I, \alpha, \lambda, \rho, \gamma)$ be a symmetric monoidal category, then we have a monoidal functor $(\mathsf{Id}, \gamma, \mathsf{id}) : \mathscr{C}_{\otimes} \to \mathscr{C}_{\otimes^{\mathsf{rev}}}$.

If we apply Theorem 8.1 to a monoid M in $\mathscr{C}_{\otimes}$, we obtain a monoid in $\mathscr{C}_{\otimes^{\mathsf{rev}}}$. From Theorem 8.2, 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, $\mathsf{Endo}_{\star}$ is a symmetric monoidal category, and therefore the reverse monoid construction can be applied to a monoid in $\mathsf{Endo}_{\star}$. The resulting monoid is known as the *reversed applicative* (Bird *et al.*, 2013).

The reversed applicative is implemented as

```
data Rev f x = Rev (f x) deriving Functor
instance Applicative f ⇒ Applicative (Rev f) where
  pure              = Rev ∘ pure
  Rev f ⊛ Rev x = Rev (pure (flip ($)) ⊛ x ⊛ f)
```

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 reverse order (Bird *et al.*, 2013).

8.5 The Tambara and Pastro monoidal functors

Not every profunctor is strong. Therefore, we are interested in investigating how to add a strength to profunctors in Pro. In the following, we show that there are two canonical functors from Pro to SPro.

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

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

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

Interestingly, this functor has right and left adjoints, yielding two canonical constructions to obtain a strong profunctor from any profunctor. We start by giving its right adjoint, that is, a functor $\textsc{Tambara}$ such that we have a natural isomorphism:

$$\phi : \quad \mathsf{Pro}(U(P, \mathsf{st}^P), Q) \quad \cong \quad \mathsf{SPro}((P, \mathsf{st}^P), \textsc{Tambara}\ Q) \tag{4}$$

The monoidal functor $\textsc{Tambara} : \mathsf{Pro} \to \mathsf{SPro}$ is given by $\textsc{Tambara}\ Q = (T_Q, \mathsf{st})$, where the first component is

$$T_Q(X, Y) = \int_Z Q(X \times Z, Y \times Z)$$

and the strength st_Z is $\langle h \rangle$, with h a dinatural transformation on V defined by the composition:

$$T_Q(X, Y) = \int_Z Q(X \times Z, Y \times Z) \xrightarrow{\omega_{Z \times V}} Q(X \times (Z \times V), Y \times (Z \times V)) \xrightarrow{Q(\alpha^{-1}, \alpha)} Q((X \times Z) \times V, (Y \times Z) \times V)$$

In the definition above, α is one of the isomorphisms of the monoidal category Pro and ω is the family of morphisms arising from the universal property of ends.

The adjunction $U \dashv \textsc{Tambara}$ tells us that the $\textsc{Tambara}$ functor completes a profunctor by cofreely adding a strength. The name of the functor is due to a similar construction defined by Pastro and Street (2008), when working on Tambara modules. This functor is monoidal (Pastro & Street, 2008), and therefore by Theorem 8.1, it maps weak arrows to arrows.

The Tambara functor may be implemented as follows:

```
data Tambara p x y = Tambara (∀z. p (x, z) (y, z))

instance Profunctor p ⇒ Profunctor (Tambara p) where
  dimap f g (Tambara x) = Tambara (dimap (lift f) (lift g) x)
    where lift f (a, b) = (f a, b)

instance Profunctor p ⇒ StrongProfunctor (Tambara p) 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 components of the isomorphism (4) are implemented in Haskell as follows:

```
ϕ :: (StrongProfunctor p, Profunctor q) ⇒ (p ⇸ q) → (p ⇸ Tambara q)
ϕ f p = Tambara (f (first p))
ϕ⁻¹ :: (StrongProfunctor p, Profunctor q) ⇒ (p ⇸ Tambara q) → (p ⇸ q)
```

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

```
ϕ⁻¹ f p = dimap fst⁻¹ fst b
  where Tambara b = f p
        fst⁻¹ x = (x, ())
```

The forgetful functor U also has a left adjoint Pastro : Pro → SPro (Pastro & Street, 2008). That is, there is a functor Pastro such that

$$\psi : \quad \mathsf{SPro}(\textsc{Pastro}\ P, (Q, \mathsf{st}^Q)) \quad \cong \quad \mathsf{Pro}(P, U(Q, \mathsf{st}^Q)) \tag{5}$$

holds. The functor is defined as $\textsc{Pastro}\ P = (F_P, \mathsf{st})$, with components

$$\begin{aligned} F_P(X, Y) &= P(X, X \to Y) \\ \mathsf{st} &= P(\pi_1, \lfloor \langle \mathsf{ev} \circ (\mathsf{id} \times \pi_1), \pi_2 \circ \pi_2 \rangle \rfloor) \end{aligned}$$

In this case, the functor Pastro is not lax monoidal, but instead *oplax monoidal*. This means there are natural transformations:

$$\begin{aligned} &\tau : \textsc{Pastro}(P \otimes Q) \to \textsc{Pastro}\ P \otimes \textsc{Pastro}\ Q \\ &\theta : \textsc{Pastro}\ \mathsf{Hom} \to \mathsf{Hom} \end{aligned}$$

Note that Theorem 8.1 cannot be used to map weak arrows to arrows since it requires lax monoidal functors and Pastro is oplax.

The Pastro functor is implemented as follows:

```
data Pastro p x y = Pastro (p x (x → y))
instance Profunctor p ⇒ Profunctor (Pastro p) where
  dimap f g (Pastro v) = Pastro (dimap f (λh x → g (h (f x))) v)
instance Profunctor p ⇒ StrongProfunctor (Pastro p) where
  first (Pastro v) = Pastro (dimap fst (λf (x, z) → (f x, z)) v)
```

In Haskell, the components of the isomorphism are

```
ψ :: (Profunctor p, StrongProfunctor q) ⇒ (Pastro p ⇢ q) → (p ⇢ q)
ψ f p = f (Pastro (dimap id (λx y → x) p))
ψ⁻¹ :: (Profunctor p, StrongProfunctor q) ⇒ (p ⇢ q) → (Pastro p ⇢ q)
ψ⁻¹ f (Pastro p) = dimap (λx → (x, x)) (λ(f, v) → f v) (first (f p))
```

The two functors Pastro and Tambara provide two ways of constructing strong profunctors from profunctors.

Using the adjunctions Pastro ⊣ U ⊣ Tambara, we obtain a monad ($U \circ$ Pastro) and a comonad ($U \circ$ Tambara), both on Pro. The corresponding categories of Eilenberg-Moore algebras for a monad and coalgebras for a comonad are both equivalent to SPro.

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

8.6 An application of the Tambara functor: A representation of arrows

Although we have not found the form of exponential objects in SPro, we can lift the exponential in SPro in such a way that we obtain an alternative representation for arrows.

The idea is to take a monoid in SPro and forget the strength structure using the forgetful functor 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 $\textsc{Tambara}(UM^{UM})$. The functor Tambara 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. an arrow), we construct a representation morphism as

$$\mathsf{rep} \quad = \quad A \xrightarrow{\langle \mathsf{st}^A \rangle} \textsc{Tambara}\ A \xrightarrow{\textsc{Tambara}(\mathsf{rep}_w)} \textsc{Tambara}\ (A^A)$$

where rep_w takes a weak arrow into its (weak arrow) Cayley representation. This is a well-defined morphism in SPro, i.e., it commutes with the strengths of A and $\textsc{Tambara}\ (A^A)$. Using the abstraction function abs_w from the Cayley representation for weak arrows, we can define an abstraction function for our arrow representation:

$$\mathsf{abs} \quad = \quad \textsc{Tambara}\ (A^A) \xrightarrow{\textsc{Tambara}(\mathsf{abs}_w)} \textsc{Tambara}\ A \xrightarrow{\omega_1} A$$

This is a left inverse to rep, and therefore rep is a monomorphism. This proves that $\textsc{Tambara}\ (A^A)$ is a representation for (A, st^A).

The implementation in Haskell of the representation, after inlining some definitions in order to simplify the code, is as follows:

```
data Rep a x y = Rep (∀z' z. a (y,z') z → a (x,z') z)

instance Profunctor a ⇒ Profunctor (Rep a) where
  dimap f g (Rep x) = Rep (λy → dimap (lift f) id (x (dimap (lift g) id y)))
    where lift f (a,b) = (f a,b)
```

The representation constructs an arrow from any profunctor:

```
instance Profunctor a ⇒ WeakArrow (Rep a) where
  arr f = Rep (dimap (lift f) id) where lift f (a,b) = (f a,b)
  Rep x ⋙ Rep y = Rep (λv → x (y v))

instance Profunctor a ⇒ StrongProfunctor (Rep a) where
  first (Rep x) = Rep (λz → dimap α⁻¹ id (x (dimap α id z)))
    where α    (x,(y,z)) = ((x,y),z)
          α⁻¹ ((x,y),z) = (x,(y,z))
```

Since we verified that the strength is compatible with the weak arrow structure, we may declare the Arrow instance:

```
instance Profunctor a ⇒ Arrow (Rep a)
```

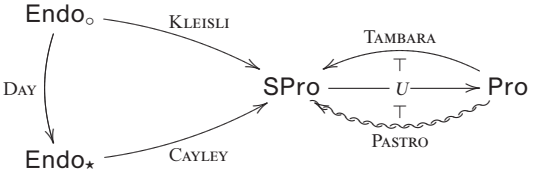
https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

Any arrow a can be embedded into $\mathsf{Rep}\ a$ using the arrow morphism rep. Moreover, $\mathsf{abs} \circ \mathsf{rep} = \mathsf{id}$:

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

8.7 The final picture

The following picture summarises the different categories and functors presented in the previous sections:



Both Kleisli and Cayley map into SPro: a functor mapping to Pro can be recovered by post-composing with U. There is an alternative functor mapping from $\mathsf{Endo}_\circ$ to SPro by composing Day with Cayley. All the functors in the picture are lax monoidal, except for Pastro, which is oplax monoidal, as indicated by the squiggly arrow.

9 Conclusion

We have shown how monads, applicative functors, and arrows can be seen in a uniform manner as monoids in a monoidal category. We exploited this uniformity in order to obtain free constructions and representations for the three notions of computation as instances of more general constructions. All these constructions were implemented in Haskell rather straightforwardly, showing that the ideas can be transferred to code without difficulty. The representations for applicative functors and arrows are new. We expect them to optimise code in the same cases for which the codensity transformation and difference lists work well: when the binary operation of the monoid is expensive on its first argument, we want to associate a sequence of computations to the right. However, an in-depth analysis of the performance of the new representations is left as future work, which could be done by benchmarking, or through formal verification (Hackett & Hutton, 2015).

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 applicative functors is described by Capriotti and Kaposi (2014). While they provide plenty of motivation for the

https://doi.org/10.1017/S0956796817000132 Published online by Cambridge University Press

use of the free applicative functor, we give a detailed description of its origin, as we arrive at it by instantiating a general description of free monoids to the category of endofunctors that is monoidal with respect to the Day convolution.

There are several works analysing the formulation of arrows as monoids (Jacobs *et al*., 2009; Asada, 2010; Asada & Hasuo, 2010; Atkey, 2011). We differ 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, they only considered monads as instances.

For the sake of 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. Rivas *et al*. (2015) analysed the cases of MonadPlus and Alternative based on a generalisation of monoidal categories to categories with a notion of near-semiring.

The relation between the different monoidal categories that support monads, applicative functors, and arrows deserves a deeper analysis. For example, it would be interesting to study the relation between monoidal categories supporting computational effects which are not Set-based.

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, and the anonymous reviewers for their helpful feedback. We also thank Tom Schrijvers, Tarmo Uustalu, and the rest of the members of IFIP WG 2.1 for their encouragement.

References

Abbott, M., Altenkirch, T., & Ghani, N. (2003). Categories of containers. In Proceedings of the 6th International Conference on Foundations of Software Science and Computation

Structures and Joint European Conference on Theory and Practice of Software, FOSSACS'03/ETAPS'03. Berlin, Heidelberg: Springer-Verlag, pp. 23–38.

Adámek, J., & Rosický, J. (1994) *Locally Presentable and Accessible Categories*. London Mathematical Society Lecture Notes, vol. 189. Cambridge University Press.

Altenkirch, T., Chapman, J. & Uustalu, T. (2010) Monads need not be endofunctors. In *Foundations of Software Science and Computational Structures*, Ong, L. (ed), Lecture Notes in Computer Science, vol. 6014. Berlin, Heidelberg: Springer, pp. 297–311.

Asada, K. (2010) Arrows are strong monads. In Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP '10, Capretta, V. & Chapman, J. (eds). ACM, pp. 33–42.

Asada, K. & Hasuo, I. (2010) Categorifying computations into components via arrows as profunctors. *Electron. Notes Theor. Comput. Sci.* **264**(2), 25–45.

Atkey, R. (2011) What is a categorical model of arrows? *Electron. Notes Theor. Comput. Sci.* **229**(5), 19–37.

Bainbridge, E. S., Freyd, P. J., Scedrov, A. & Scott, P. J. (1990) Functorial polymorphism. *Theor. Comput. Sci.* **70**(1), 35–64.

Barr, M. & Wells, C. (1985) *Toposes, Triples and Theories*. Grundlehren der Mathematischen Wissenschaften, vol. 278. Springer-Verlag.

Bénabou, J. (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, R., Gibbons, J., Mehner, S., Voigtländer, J. & Schrijvers, T. (2013) Understanding idiomatic traversals backwards and forwards. In Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Haskell '13. ACM, pp. 25–36.

Capriotti, P. & Kaposi, A. (2014) Free applicative functors. Proceedings of the 5th Workshop on Mathematically Structured Functional Programming, Levy, P. & Krishnaswami, N. (eds), EPTCS, vol. 153, pp. 2–30.

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

Danielsson, N. A., Hughes, J., Jansson, P. & Gibbons, J. (2006) Fast and loose reasoning is morally correct. In Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Morrisett, J. G. & Jones, S. L. P. (eds). ACM, pp. 206–217.

Day, B. (1970) On closed categories of functors. In *Reports of the Midwest Category Seminar IV*. Lecture Notes in Mathematics, vol. 137. Berlin, Heidelberg: Springer, pp. 1–38.

Day, B. (1973) Note on monoidal localisation. *Bull. Aust. Math. Soc.* **8**(2), 1–16.

Day, B. J. & Kelly, G. M. (1969) Enriched functor categories. *Reports of the Midwest Category Seminar III*. Lecture Notes in Mathematics, vol. 106. Berlin, Heidelberg: Springer, pp. 178–191.

Day, B. J. & Lack, S. (2007) Limits of small functors. *J. Pure Appl. Algebra* **210**(3), 651–663.

Dubuc, E. J. (1974) Free monoids. *J. Algebra* **29**(2), 208–228.

Hackett, J. & Hutton, G. (2015) Programs for cheap! Proceedings of the 30th Annual ACM/IEEE Symposium on Logic in Computer Science. IEEE, pp. 115–126.

Hudak, P., Courtney, A., Nilsson, H. & Peterson, J. (2003) Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*. Lecture Notes in Computer Science, vol. 2638. Springer-Verlag, pp. 159–187.

Hughes, J. (1986) A novel representation of lists and its application to the function "reverse". *Inform. Process. Lett.* **22**(3), 141–144.

Hughes, J. (2000) Generalising monads to arrows. *Sci. Comput. Program.* **37**(1-3), 67–111.

Hutton, G., Jaskelioff, M. & Gill, A. (2010) Factorising folds for faster functions. *J. Funct. Program.* **20**(Special Issue 3–4), 353–373.

Jacobs, B., Heunen, C. & Hasuo, I. (2009) Categorical semantics for arrows. *J. Funct. Program.* **19**(3–4), 403–438.

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

Jaskelioff, M. (2009) Modular monad transformers. In *Programming Languages and Systems: 18th European Symposium on Programming*, Castagna, G. (ed), Lecture Notes in Computer Science, vol. 5502. Springer, pp. 64–79.

Jaskelioff, M. & Moggi, E. (2010) Monad transformers as monoid transformers. *Theor. Comput. Sci.* **411**(51–52), 4441–4466.

Jaskelioff, M., & Rypacek, O. (2012) An investigation of the laws of traversals. In Proceedings of the 4th Workshop on Mathematically Structured Functional Programming, Chapman, J. & Levy, P. B. (eds), EPTCS, vol. 76, pp. 40–49.

Kelly, G. M. (1980) A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on. *Bull. Aust. Math. Soc.* **22**(01), 1–83.

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

Lack, S. (2010) Note on the construction of free monoids. *Appl. Categ. Struct.* **18**(1), 17–29.

Li, P. & Zdancewic, S. (2010) Arrows for secure information flow. *Theor. Comput. Sci.* **411**(19), 1974–1994.

Lindley, S., Wadler, P. & Yallop, J. (2011) Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electron. Notes Theor. Comput. Sci.* **229**(5), 97–117.

Mac Lane, S. (1971) *Categories for the Working Mathematician*, 2nd ed. Graduate Texts in Mathematics, vol. 5. Springer-Verlag, 1998.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *J. Funct. Program.* **18**(01), 1–13.

Moggi, E. (1989) Computational lambda-calculus and monads. In Proceedings of the 4th Annual Symposium on Logic in Computer Science. IEEE Computer Society, pp. 14–23.

Moggi, E. (1991) Notions of computation and monads. *Inform. Comput.* **93**(1), 55–92.

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

Pastro, C. & Street, R. (2008) Doubles for monoidal categories. *Theory Appl. Categ.* **21**, 61–75.

Paterson, R. (2012) Constructing applicative functors. In *Mathematics of Program Construction*, Gibbons, J. & Nogueira, P. (eds), Lecture Notes in Computer Science, vol. 7342. Berlin, Heidelberg: Springer, pp. 300–323.

Peyton Jones, S. L., Vytiniotis, D., Weirich, S. & Washburn, G. (2006) Simple unification-based type inference for GADTs. In Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, Reppy, J. H., & Lawall, J. L. (eds). ACM, pp. 50–61.

Reynolds, J. C. (1980) Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation*, Jones, N. D. (ed), Lecture Notes in Computer Science, vol. 94. Springer, pp. 211–258.

Rivas, E., Jaskelioff, M. & Schrijvers, T. (2015) From monoids to near-semirings: the essence of MonadPlus and alternative. In Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Falaschi, M. & Albert, E. (eds). ACM, pp. 196–207.

Swierstra, W. & Altenkirch, T. (2007) Beauty in the beast. In Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell '07, Keller, G. (ed). ACM, pp. 25–36.

Vizzotto, J., Altenkirch, T. & Sabry, A. (2006) Structuring quantum effects: Superoperators as arrows. *Math. Struct. Comput. Sci.* **16**(3), 453–468.

Voigtländer, J. (2008) Asymptotic improvement of computations over free monads. In Proceedings of the 9th International Conference on Mathematics of Program Construction, Audebaud, P. & Paulin-Mohring, C. (eds), Lecture Notes in Computer Science, vol. 5133. Springer-Verlag, pp. 388–403.