

# On Structuring Functional Programs with Monoidal Profunctors

Alexandre Garcia de Oliveira

IME-USP  
São Paulo, Brazil  
Instituto de Matemática e Estatística  
Univerisidade de São Paulo  
São Paulo, Brazil

alexgrcol (at) hotmail.com

Mauro Jaskelioff

CIFASIS-CONICET  
Rosario, Argentina  
FCEIA  
Universidad Nacional de Rosario  
Argentina

jaskelioff(at)cifasis-conicet.gov.ar

Ana Cristina Vieira de Melo

IME-USP  
São Paulo, Brazil  
acvm(at)ime.usp.br

We study monoidal profunctors as a tool to reason and structure pure functional programs both from a categorical perspective and as a Haskell implementation. From the categorical point of view we approach them as monoids in a certain monoidal category of profunctors. We study properties of this monoidal category and construct and implement the free monoidal profunctor. We study the relationship of the monoidal construction to optics, and introduce a promising generalization of the implementation which we illustrate by introducing effectful monoidal profunctors.

## 1 Introduction

Functors, applicative functors [14], monads [16, 22], profunctors, and arrows [4, 5] are by now part of the vocabulary of the programmer who writes mathematically structured programs. In order to understand and develop these structures both the categorical view and the programming view have been helpful. For example, Lindley et al. [12] compare these structures from the point of view of (typed) programming languages, and Rivas and Jaskelioff [20] compare them from the point of view of monoidal categories. In this last work, both monads and applicative functors are seen as monoids in a monoidal category of endofunctors. Monads use functor composition as tensor, whereas applicative functors use the Day convolution as tensor. Likewise, arrows can be seen as monoids in a monoidal category of (strong) profunctors.

However, given that the Day convolution can also be a tensor of profunctors there is another structure yet to be studied. In this work we study monoidal profunctors, a structure obtained from considering monoids in a category of profunctors taking the Day convolution as tensor, hence filling the gap in the following table:

functor	applicative	monad
profunctor	????	arrow

Table 1: Structure relations

Monoidal profunctors are a categorical structure with two components: an identity computation and a generic parallel composition.

Therefore, with this paper we aim to gather the knowledge about monoidal profunctors and study their application in the context of functional programming.

After some mathematical background (Section 2), we introduce monoidal profunctors as monoids in a monoidal category and study some of their properties (Section 3). Then, in Section 4, we implement these ideas in Haskell and provide some instances. We show the free monoidal profunctor and a Haskell implementation in Section 5. In Section 6, we present an application of monoidal profunctors to optics [1] and observe connections with well-known structures such as traversals and grates [19, 17]. In Section 7, we generalize the implementation to other categories and illustrate its application to effectful monoidal profunctors.

## 2 Mathematical background

### 2.1 Monoidal Categories and Monoids

A *monoidal category* [20] gives us a minimal framework for defining the categorical version of a monoid.

**Definition 2.1.** A monoidal category is a sextuple  $(\mathcal{C}, \otimes, I, \alpha, \rho, \lambda)$  where

- $\mathcal{C}$  is a category;
- $\otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$  is a bifunctor;
- $I$  is an object called unit;
- $\rho_A : A \otimes I \rightarrow A$ ,  $\lambda_A : I \otimes A \rightarrow A$  and  $\alpha_{ABC} : (A \otimes B) \otimes C \rightarrow A \otimes (B \otimes C)$  are three natural isomorphisms.

If the isomorphisms  $\rho$ ,  $\lambda$  and  $\alpha$  are identities then the monoidal category is called *strict*, if there is a natural isomorphism  $\gamma_{AB} : A \otimes B \rightarrow B \otimes A$  the monoidal category is called *symmetric*.

A monoidal category is *closed* if there is an additional functor, called the internal hom,  $\Rightarrow : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \text{Set}$  such that  $\mathcal{C}(A \otimes B, C) \cong \mathcal{C}(A, B \Rightarrow C)$ , natural in  $A$ ,  $B$  and  $C$ , objects of  $\mathcal{C}$ . The witnesses of this isomorphism are called currying and uncurrying. In *Set*, with  $\otimes = \times$ ,  $A \Rightarrow B$  is just the hom-set  $A \rightarrow B$ .

**Definition 2.2.** A monoid in a monoidal category  $\mathcal{C}$  is the tuple  $(M, e, m)$  where  $M$  is an object of  $\mathcal{C}$ ,  $e : I \rightarrow M$  is the unit morphism and  $m : M \otimes M \rightarrow M$  is the multiplication morphism, satisfying

1. Right unit:  $m \circ (id \otimes e) = \rho_{MMM}$
2. Left unit:  $m \circ (e \otimes id) = \lambda_{MMM}$
3. Associativity:  $m \circ (m \otimes id) = m \circ (id \otimes m) \circ \alpha_{MMM}$

### 2.2 Profunctors

A *profunctor* generalizes the notions of function relation and bimodule [10].

**Definition 2.3.** Given two categories  $\mathcal{C}$  and  $\mathcal{D}$ , a profunctor from  $\mathcal{C}$  to  $\mathcal{D}$  is a functor  $P : \mathcal{C}^{op} \times \mathcal{D} \rightarrow \text{Set}$ , consisting of:

- for each a object of  $\mathcal{C}$  and b object of  $\mathcal{D}$ , a set  $P(a, b)$ ;
- for each a object of  $\mathcal{C}$  and b, d objects of  $\mathcal{D}$ , a function (left action)  $\mathcal{D}(d, b) \times P(a, d) \rightarrow P(a, b)$ ;

- for all  $a, c$  objects of  $\mathcal{C}$  and  $b$  object of  $\mathcal{D}$ , a function (right action)  $P(a, b) \times \mathcal{C}(c, a) \rightarrow P(c, b)$ .

This notion is also known as a Bimodule or a  $(\mathcal{C}, \mathcal{D})$ -module, and also as a distributor.

Since a profunctor is a functor from the product category  $\mathcal{C}^{op} \times \mathcal{D}$  to  $Set$ , it must satisfy the functor laws.

$$\begin{aligned} P(1_C, 1_D) &= 1_{P(C, D)} \\ P(f \circ g, h \circ i) &= P(g, h) \circ P(f, i) \end{aligned}$$

An example of a profunctor is the hom functor  $Hom : \mathcal{C}^{op} \times \mathcal{C} \rightarrow Set$ , written as  $A \rightarrow B$  when  $\mathcal{C} = Set$ , and its actions are just pre-composition and post-composition of set functions.

**Definition 2.4.** Let  $\mathcal{C}$  and  $\mathcal{D}$  be small categories,  $Prof(\mathcal{C}, \mathcal{D})$  is the profunctor category consisting of profunctors  $\mathcal{C}^{op} \times \mathcal{D} \rightarrow Set$  as objects, natural transformations between profunctors as morphisms, and vertical composition to compose them. The fact that  $\mathcal{C}$  and  $\mathcal{D}$  are small categories will always be implied when some statement about the category  $Prof(\mathcal{C}, \mathcal{D})$  is present. We will call this category  $Prof$  when the categories  $\mathcal{C}$  and  $\mathcal{D}$  are clear from the context.

The profunctor category inherits some structure from the category  $Set$  such as binary products given by  $(P \times Q)(S, T) = P(S, T) \times Q(S, T)$  and binary coproducts given by  $(P + Q)(S, T) = P(S, T) + Q(S, T)$ , where  $\times, +$  are the respective universal constructions from  $Set$ . There are also terminal and initial profunctors given by  $1_p(S, T) = \{*\}$  and  $0_p(S, T) = \emptyset$ , i.e., they are just constant maps to the initial and terminal object of  $Set$ , respectively.

### 2.3 Day Convolution

**Definition 2.5.** Let  $\mathcal{E}$  be a small monoidal category and  $F, G : \mathcal{E} \rightarrow Set$ , then the Day convolution [2] of  $F$  and  $G$  is another functor (in  $T$ ) given by the coend

$$(F \star G)T = \int^{XY} FX \times GY \times \mathcal{D}(X \otimes Y, T). \quad (1)$$

One can work with this coend in the category  $Prof(\mathcal{C}, \mathcal{D})$  of profunctors, that is, letting  $\mathcal{E} = \mathcal{C}^{op} \times \mathcal{D}$  in the above definition, enabling us to derive the following Day convolution for profunctors.

$$(P \star Q)(S, T) \cong \int^{ABCD} P(A, B) \times Q(C, D) \times \mathcal{C}(S, A \otimes C) \times \mathcal{C}(B \otimes D, T)$$

The profunctor  $J(A, B) = \mathcal{C}(A, I) \times \mathcal{C}(I, B)$  is a unit for  $\star$ . When  $I = 1$ , where 1 is the terminal object, then  $J(A, B) \cong B$ . When  $\mathcal{C} = Set$ , this convolution gives rise to a monoidal profunctor (see next section), when  $\mathcal{C}$  is a Kleisli category on  $Set$  (same objects, but morphisms relies on a monad) gives the notion of an effectful monoidal profunctor (see Section 7).

**Proposition 2.6.** Let  $\mathcal{C}$  be a monoidal category, the profunctor  $J(A, B) = \mathcal{C}(A, I) \times \mathcal{C}(I, B)$  is the right and left unit of  $\star$ .

*Proof.* The calculation is standard coend calculus [13] using Yoneda's lemma. This is a proof for  $J$  being

a right unit, for the left one is analogous.

$$\begin{aligned}
(P \star J)(S, T) &= \int^{ABCD} P(A, B) \times J(C, D) \times \mathcal{C}(S, A \otimes C) \times \mathcal{C}(B \otimes D, T) \\
&\cong \int^{ABCD} P(A, B) \times \mathcal{C}(C, I) \times \mathcal{C}(I, D) \times \mathcal{C}(S, A \otimes C) \times \mathcal{C}(B \otimes D, T) \\
&\cong \int^{ABD} P(A, B) \times \mathcal{C}(I, D) \times \mathcal{C}(S, A \otimes I) \times \mathcal{C}(B \otimes D, T) \\
&\cong \int^{AB} P(A, B) \times \mathcal{C}(S, A \otimes I) \times \mathcal{C}(B \otimes I, T) \\
&\cong \int^{AB} P(A, B) \times \mathcal{C}(S, A) \times \mathcal{C}(B, T) \\
&\cong P(S, T)
\end{aligned}$$

□

The associativity of  $\star$  is required to define a monoidal profunctor category.

**Proposition 2.7.** *Let  $(\mathcal{C}, \otimes, I)$  be a monoidal category and  $S, T$  two objects of  $\mathcal{C}$ , the Day convolution for profunctors is an associative tensor product  $(P \star Q) \star R \cong P \star (Q \star R)$ .*

*Proof.* The proof follows the same coend calculus pattern using Yoneda's lemma whenever needed. □

In order to be able to define monoids in a monoidal profunctor category, one needs to check that when  $\mathcal{C}$  and  $\mathcal{D}$  are monoidal categories then  $(\text{Prof}(\mathcal{C}, \mathcal{D}), \star, J)$  is a monoidal category.

**Theorem 2.8.** *Let  $\mathcal{C}$  and  $\mathcal{D}$  be monoidal small categories. Then  $(\text{Prof}(\mathcal{C}, \mathcal{D}), \star, J)$  is a monoidal category.*

*Proof.* Since  $\mathcal{C}$  and  $\mathcal{D}$  are monoidal categories,  $\star$  is a bifunctor by construction, and by Proposition 2.6 and 2.7 gives the desired morphisms, it follows that  $(\text{Prof}(\mathcal{C}, \mathcal{D}), \star, J)$  is a monoidal category. □

Having obtained a monoidal category of profunctors, it is now possible to define a monoid in this category. In order to do that, we will use the following proposition (as in the work of Rivas and Jaskelioff [20]).

**Proposition 2.9.** *Let  $\mathcal{D} = \mathcal{C}^{op} \otimes \mathcal{C}$ , there is a one-to-one correspondence defining morphisms going out of a Day convolution for profunctors*

$$\int_{XY} (P \star Q)(X, Y) \rightarrow R(X, Y) \cong \int_{ABCD} P(A, B) \times Q(C, D) \rightarrow R(A \otimes C, B \otimes D)$$

which is natural in  $P$ ,  $Q$  and  $R$ .

**Proof.** This proof uses the same coend calculus pattern with the help of Yoneda lemma and the fact that the hom functor commutes with ends and coends [13].

$$\begin{aligned}
& \int_{XY} (P \star Q)(X, Y) \rightarrow R(X, Y) \\
& \cong \int_{XY} \left( \int^{ABCD} (P(A, B) \times Q(C, D)) \times \mathcal{C}(X, A \otimes C) \times \mathcal{C}(B \otimes D, Y) \right) \rightarrow R(X, Y) \\
& \cong \int_{XYABCD} (P(A, B) \times Q(C, D)) \times \mathcal{C}(X, A \otimes C) \times \mathcal{C}(B \otimes D, Y) \rightarrow R(X, Y) \\
& \cong \int_{XYABCD} (P(A, B) \times Q(C, D)) \rightarrow \mathcal{C}(X, A \otimes C) \rightarrow \mathcal{C}(B \otimes D, Y) \rightarrow R(X, Y) \\
& \cong \int_{YABCD} (P(A, B) \times Q(C, D)) \rightarrow \mathcal{C}(B \otimes D, Y) \rightarrow R(A \otimes C, Y) \\
& \cong \int_{ABCD} P(A, B) \times Q(C, D) \rightarrow R(A \otimes C, B \otimes D)
\end{aligned}$$

Whenever  $P = Q = R$  in the equation of Proposition 2.9 we get the following isomorphism, useful to define a monoid in the profunctor category  $Prof$  with Day convolution as its tensor.

$$\int_{XY} (P \star P)(X, Y) \rightarrow P(X, Y) \cong \int_{ABCD} P(A, B) \times P(C, D) \rightarrow P(A \otimes C, B \otimes D)$$

### 3 Monoidal Profunctors

We define *monoidal profunctors* as monoids in the monoidal category of profunctors of theorem 2.8.

**Proposition 3.1.** Let  $(\mathcal{C}, \otimes, I)$  be a small monoidal category,  $P : \mathcal{C}^{op} \times \mathcal{C} \rightarrow Set$  be a profunctor, and  $S, T$  two objects of  $\mathcal{C}$ . Then  $\mathcal{C}(J(S, T), P(S, T)) \cong P(I, I)$ .

*Proof.*

$$\begin{aligned}
\mathcal{C}(J(S, T), P(S, T)) & \cong \mathcal{C}(S, I) \times \mathcal{C}(I, T) \rightarrow P(S, T) \\
& \cong \mathcal{C}(S, I) \rightarrow \mathcal{C}(I, T) \rightarrow P(S, T) \\
& \cong \mathcal{C}(S, I) \rightarrow P(S, I) \\
& \cong P(I, I)
\end{aligned}$$

□

With all categorical tools in hand, the central notion of this work emerges from the category of monoidal profunctors.

**Definition 3.2.** Let  $(\mathcal{C}, \otimes, I)$  and  $(\mathcal{D}, \otimes, I)$  be small monoidal categories. A monoid in the monoidal profunctor category  $Prof(\mathcal{C}, \mathcal{D})$  consists of a profunctor  $P$ , a unit  $e : P(I, I)$ , and a multiplication given by a natural family of morphisms  $m_{ABCD} : P(A, B) \times P(C, D) \rightarrow P(A \otimes C, B \otimes D)$ .

The unit is a natural transformation  $e : J \rightarrow P$ , which by proposition 3.1 is isomorphic to  $e : P(I, I)$ . The multiplication is a natural transformation  $m : P \star P \rightarrow P$ , which by proposition 2.9 is equivalent to the family above.

As an example, consider  $(Set, \otimes, I)$ , where  $I$  is a singleton set, and the *Hom* profunctor  $P(A, B) = A \rightarrow B$ , trivially gives us a monoidal profunctor.

**Proposition 3.3.** *Let  $(\mathcal{C}, \otimes, I)$  and  $(\mathcal{D}, \otimes, J)$  be a small monoidal categories, and  $P, Q$  monoidal profunctors, then*

$$(P \Rightarrow Q)(X, Y) = \int_{CD} P(C, D) \rightarrow Q(X \otimes C, Y \otimes D)$$

*defines an internal hom on the monoidal profunctor category  $Prof(\mathcal{C}, \mathcal{D})$ .*

*Proof.* This proof follows the same steps as in the functor case [20] but adapting it for  $Prof(\mathcal{C}, \mathcal{D})$ .  $\square$

This proposition states that the monoidal category of profunctors  $Prof$  is closed.

## 4 Implementation in Haskell

We implement in Haskell the concepts of profunctors, Day convolution, and monoidal profunctors defined before. Although other monoidal profunctors can be derived using other bifunctors, this work focuses only on the product. This section considers the (fictitious) category  $Hask$  as a small monoidal category and  $Prof(Hask, Hask)$  as the small category of profunctors to implement the typeclass  $MonoPro$  which represents the desired monoid over this category.

### 4.1 Profunctors

A profunctor is an instance of the following class

```
class Profunctor p where
  dimap :: (a → b) → (c → d) → p b c → p a d
```

A profunctor is a functor, thus  $dimap$  needs to satisfy the functor laws.

The profunctor interface lifts pure functions into both type arguments, the first in a contravariant manner, and the second in a covariant way. A morphism in the  $Prof$  category can be represented, in Haskell, as the type below.

```
type (↔) p q = ∀x y. p x y → q x y
```

The function type  $(\rightarrow)$ , is the most basic example of a profunctor.

One notion captured by a Profunctor is that of a structured input and structured output of a function  $SISO$ . This type generalizes Kleisli arrows which allow a pure input and a structured output.

```
data SISO f g a b = SISO { unSISO :: f a → g b }
instance (Functor f, Functor g) ⇒ Profunctor (SISO f g) where
  dimap ab cd (SISO bc) = SISO (fmap cd ∘ bc ∘ fmap ab)
```

### 4.2 The Day convolution type

The Day convolution is represented by the existential type

```
data Day p q s t = ∀a b c d. Day (p a b) (q c d) (s → (a, c)) (b → d → t)
```

Since  $\mathcal{C}(A, I)$  is isomorphic to a singleton set (unit of the cartesian product  $\times$ ), and  $\mathcal{C}(I, B) \cong B$ , one can write, in Haskell, the type

**data**  $I\ a\ b = I\ \{unI :: b\}$

as the unit of the Day convolution. The following functions are representations of the right and left units.

$$\begin{aligned} \rho &:: \text{Profunctor } p \Rightarrow \text{Day } p\ I \rightsquigarrow p \\ \rho (\text{Day } pab\ (I\ d)\ sac\ bdt) &= \text{dimap } (fst \circ sac)\ (\lambda b \rightarrow bdt\ b\ d)\ pab \\ \lambda &:: \text{Profunctor } q \Rightarrow \text{Day } I\ q \rightsquigarrow q \\ \lambda (\text{Day } (I\ b)\ qcd\ sac\ bdt) &= \text{dimap } (snd \circ sac)\ (\lambda d \rightarrow bdt\ b\ d)\ qcd \end{aligned}$$

The associativity of the Day convolution and its symmetric map also can be represented in Haskell as the functions below.

$$\begin{aligned} \alpha &:: (\text{Profunctor } p, \text{Profunctor } q, \text{Profunctor } r) \Rightarrow \text{Day } (\text{Day } p\ q)\ r \rightsquigarrow \text{Day } p\ (\text{Day } q\ r) \\ \alpha (\text{Day } (\text{Day } p\ q\ s_1\ f)\ r\ s_2\ g) &= \text{Day } p\ (\text{Day } q\ r\ f_1\ f_2)\ f_3\ f_4 \\ \text{where} & \\ f_1 &= \text{first}'\ (snd \circ s_1) \circ s_2 \\ f_2\ d_1\ d_2 &= (d_2, \lambda x \rightarrow f\ x\ d_1) \\ f_3 &= \text{first}'\ (fst \circ s_1 \circ (fst \circ s_2)) \circ \text{diag} \\ f_4\ b_1\ (d_2, h) &= g\ (h\ b_1)\ d_2 \\ \gamma &:: (\text{Profunctor } p, \text{Profunctor } q) \Rightarrow \text{Day } p\ q \rightsquigarrow \text{Day } q\ p \\ \gamma (\text{Day } p\ q\ sac\ bdt) &= \text{Day } q\ p\ (\text{swap} \circ sac)\ (\text{flip } bdt) \\ \text{where } \text{swap } (x, y) &= (y, x) \end{aligned}$$

Functions  $\rho$ ,  $\lambda$ , and  $\alpha$  are natural isomorphisms. We leave the definition of the inverses as an exercise for the reader.

### 4.3 MonoPro typeclass

We define a typeclass called *MonoPro* for implementing monoidal profunctors. The type  $p\ ()\ ()$  is a representation in Haskell of the unit  $P(I, I)$ . The multiplication is obtained from Proposition 2.9, which gives the multiplication a type  $\int_{ABCD} P(A, B) \times P(C, D) \rightarrow P(A \otimes C, B \otimes D)$  allowing to write the following class in Haskell.

```
class Profunctor p  $\Rightarrow$  MonoPro p where
  mpepty :: p () ()
  ( $\star$ ) :: p b c  $\rightarrow$  p d e  $\rightarrow$  p (b, d) (c, e)
```

satisfying the monoid laws

- Left identity:  $\text{dimap } \text{diag } \text{snd } (mpepty \star f) = f$
- Right identity:  $\text{dimap } \text{diag } \text{fst } (f \star mpepty) = f$
- Associativity:  $\text{dimap } \text{assoc}^{-1} \text{ assoc } (f \star (g \star h)) = (f \star g) \star h$

where the helper functions  $\text{diag} :: x \rightarrow (x, x)$ ,  $\text{assoc}^{-1} :: ((x, y), z) \rightarrow (x, (y, z))$ , and  $\text{assoc} :: (x, (y, z)) \rightarrow ((x, y), z)$  are the obvious ones.

Another way to understand *MonoPro* is that it lifts pure functions with many inputs to a binary constructor type, while a profunctor only lifts functions with one type as input parameter.

$$\begin{aligned}
lmap_2 &:: MonoPro\ p \Rightarrow (s \rightarrow (a, c)) \rightarrow p\ a\ b \rightarrow p\ c\ d \rightarrow p\ s\ (b, d) \\
lmap_2\ f\ pa\ pc &= dimap\ f\ id\ (pa \star pc) \\
rmap_2 &:: MonoPro\ p \Rightarrow ((b, d) \rightarrow t) \rightarrow p\ a\ b \rightarrow p\ c\ d \rightarrow p\ (a, c)\ t \\
rmap_2\ f\ pa\ pc &= dimap\ id\ f\ (pa \star pc)
\end{aligned}$$

which can work together as one function

$$\begin{aligned}
rlmap &:: MonoPro\ p \Rightarrow ((b, d) \rightarrow t) \rightarrow (s \rightarrow (a, c)) \rightarrow p\ a\ b \rightarrow p\ c\ d \rightarrow p\ s\ t \\
rlmap\ f\ g\ pa\ pc &= dimap\ f\ g\ (pa \star pc)
\end{aligned}$$

which is the same behavior as the Day convolution of  $p$  with itself. Such convolution is the *raison d'être* of a monoidal profunctor. A parallel composition is followed by a covariant and a contravariant lifting of two pure functions matching the inner structure, which in our case is the product type  $(,)$ .

If one chooses a suitable monoidal profunctor  $p$  and type  $s$ ,  $MonoPro\ p\ s$  inherits the applicative functor behavior naturally.

$$\begin{aligned}
appToMonoPro &:: MonoPro\ p \Rightarrow p\ s\ (a \rightarrow b) \rightarrow p\ s\ a \rightarrow p\ s\ b \\
appToMonoPro\ pab\ pa &= dimap\ diag\ (uncurry\ \$)\ (pab \star pa)
\end{aligned}$$

The  $MonoPro$  typeclass has a straightforward instance for the Hom profunctor  $(\rightarrow)$  which satisfies the monoidal profunctor laws trivially. A practical use for this instance is writing expressions in a point-free manner. One can write an  $unzip' :: Functor\ f \Rightarrow f\ (a, b) \rightarrow (f\ a, f\ b)$  function, for example, for any functor that has as input a pair type. A  $SISO$  is another example of a monoidal profunctor.

$$\begin{aligned}
\mathbf{instance}\ (Functor\ f, Applicative\ g) &\Rightarrow MonoPro\ (SISO\ f\ g)\ \mathbf{where} \\
mempty &= SISO\ (\lambda\_ \rightarrow pure\ ()) \\
SISO\ f \star SISO\ g &= SISO\ (zip' \circ (f \star g) \circ unzip')
\end{aligned}$$

where  $zip' :: Applicative\ f \Rightarrow (f\ a, f\ b) \rightarrow f\ (a, b)$  is the applicative functor multiplication. The most basic notion of a monoidal profunctor is represented by this instance. It tells us that the input needs to be a functor instance because of  $unzip'$ , the functions  $f$  and  $g$  are composed in a parallel manner using the monoidal profunctor instance for  $(\rightarrow)$  and then regrouped together using the applicative (monoidal) behavior of  $zip'$ .

## 5 Free Monoidal Profunctors

There are different theorems with different hypothesis that ensure the existence of a free monoid in a monoidal category. Here, we follow Rivas and Jaskelioff [20] and use the following proposition to ensure the existence of the *free monoidal profunctor*.

**Proposition 5.1** ([20]). *Let  $(\mathcal{C}, \otimes, I)$  be a monoidal category with internal homs. If  $\mathcal{C}$  has binary coproducts, and for each  $A \in ob(\mathcal{C})$  the initial algebra for the endofunctor  $I + A \otimes -$  exists, then for each  $A$  the free monoid  $A^*$  exists and its carrier is the carrier of the initial algebra.*

The category  $Prof(\mathcal{C}, \mathcal{C})$ , when  $\mathcal{C}$  is a small monoidal category, is monoidal with the Day convolution  $\star$  and the profunctor  $I$  as its unit, and also has binary coproducts and internal homs. The least fixed point of the endofunctor  $Q(X) = I + P \star X$  in  $Prof(\mathcal{C}, \mathcal{C})$  gives the free monoidal profunctor.

Following this definition we arrive at the following implementation of the free monoidal profunctor (see also [15]).



**data** *FreeMP* *p s t* **where**

*MPempty* :: *t* → *FreeMP p s t*

*FreeMP* :: (*s* → (*x, z*)) → ((*y, w*) → *t*) → *p x y* → *FreeMP p z w* → *FreeMP p s t*

where *MPempty* corresponds to *mpempty*, and *FreeMP* is the multiplication. The multiplication will be apparent if one expands the definition of Day convolution for *P* and *P\**. This interface stacks profunctors, and in each layer, it provides pure functions to simulate the parallel composition nature of a monoidal profunctor.

The following functions provide the necessary functions to build the free construction on monoidal profunctors, *toFreeMP* insert a single profunctor into the free structure, and *foldFreeMP* provides a way of evaluating the structure, collapsing it into a single monoidal profunctor.

*toFreeMP* :: *Profunctor p* ⇒ *p s t* → *FreeMP p s t*

*toFreeMP p* = *FreeMP diag fst p (MPempty ())*

*foldFreeMP* :: (*Profunctor p, MonoPro q*) ⇒ (*p* ∼ *q*) → *FreeMP p s t* → *q s t*

*foldFreeMP* \_ (*Arr t*) = *dimap (\\_ → ()) (λ () → t) arrr*

*foldFreeMP (Prof h) (FreeMP f g p mp)* = *dimap f g ((h p) ★ foldFreeMP (Prof h) mp)*

A free construction behaves like a list and, of course, *MonoPro* should provide a way to embed a plain profunctor into the free context.

*consMP* :: *Profunctor p* ⇒ *p a b* → *FreeMP p s t* → *FreeMP p (a, s) (b, t)*

*consMP pab (MPempty t)* = *FreeMP id id pab (MPempty t)*

*consMP pab (FreeMP f g p fp)* = *FreeMP (id ★ f) (id ★ g) pab (consMP p fp)*

and with it, an instance of *MonoPro* for the free structure can be defined as

**instance** *Profunctor p* ⇒ *MonoPro (FreeMP p)* **where**

*mpempty* = *MPempty ()*

*MPempty t ★ q* = *dimap snd (λ x → (t, x)) q*

*q* ★ *MPempty t* = *dimap fst (λ x → (x, t)) q*

*(FreeMP f g p fp) ★ fq* = *dimap (assoc ∘ (f ★ id)) ((g ★ id) ∘ assoc<sup>-1</sup>) (consMP p (fp ★ fq))*

where *assoc* :: (*(x, z), c*) → (*z, (x, c)*) and *associnv'* :: (*y, (w, d)*) → (*(w, y), d*).

When *p* is an arrow, then *FreeMP p* is an arrow. In order to define this instance one needs to collapse all parallel profunctors in order to make the sequential composition.

**instance** (*MonoPro p, Arrow p*) ⇒ *Category (FreeMP p)* **where**

*id* = *FreeMP (λ x → (x, ())) fst (arr id) (MPempty ())*

*mp* ∘ *mq* = *toFreeMP (fromFreeMP mp K. ∘ fromFreeMP mq)*

**instance** (*MonoPro p, Arrow p*) ⇒ *Arrow (FreeMP p)* **where**

*arr f* = *FreeMP (λ x → (x, ())) fst (arr f) (MPempty ())*

*(\*\*\*)* = *(★)*

It is good to remember that the type class *Category* is the one that has the two methods *id* and *(∘)*, which represents the notion of a category in Haskell.

## 6 Monoidal profunctor optics - Monocles

Data accessors are an essential part of functional programming. They allow reading and writing a whole data structure or parts of it [19]. In Haskell, one needs to deal with Algebraic Data Types (ADTs) such as products (fields), sums, containers, function types, to name a few. For each of these structures, the action of handling can be a hard task and not compositional at all. To circumvent this problem, different type of data accessors were created, such as lenses, prisms, and traversables [9, 8]. These different notions of modular (composable) data accessors [19] were grouped together and called *optics*. Optics help to tackle the data accessor problem with the help of some category-theoretic constructions such as profunctors.

An optic is a general denotation to locate parts (or even the whole) of a data structure in which some action needs to be performed. Each optic deals with a different ADT, for example, the well-known lenses deal with product types, prisms with sum types, traversals with traversable containers, grates with function types, isos deals with any type but cannot change its shape, and so on. A general optic is a polymorphic type on a binary type constructor type with a typeclass restriction  $r$  on it.

**type**  $Optic\ r\ s\ t\ a\ b = \forall p.r\ p \Rightarrow p\ a\ b \rightarrow p\ s\ t$

The idea of an optic is to have an in-depth look into get/set operations, for example, if one has a “big” data structure  $s$ , it is possible to extract a piece of it, say  $a$ , which can be written as a function  $get :: s \rightarrow a$ . Whereas, if one provides a “big” structure  $s$ , and a value  $b$  (which is a part of  $s$ ), we can modify  $s$  into another “big” structure  $t$  (the structure might not change and the data still be  $s$ ). This description is modelled by the function  $set :: s \rightarrow b \rightarrow t$ .

Both functions (get and set) are specializations of *Optic Strong* yielding the type

**type**  $Lens\ s\ t\ a\ b = \forall p.Strong\ p \Rightarrow p\ a\ b \rightarrow p\ s\ t$

Here, *Strong* is a profunctor dependent typeclass having the method  $first' :: p\ a\ b \rightarrow p\ (a,x)\ (b,x)$  (it also has a method  $second'$ ). For example, one can recover a get function using a lens [19, 17], and to achieve that we use the profunctor *Forget* that is just the contravariant hom-functor:

**newtype**  $Forget\ r\ a\ b = Forget\ \{runForget :: a \rightarrow r\}$   
 $get :: Lens\ s\ t\ a\ b \rightarrow s \rightarrow a$   
 $get\ lens = runForget\ (lens\ (Forget\ id))$

Lenses help to give the intuition behind this profunctorial optics machinery, but this work will solely focus on the optic derived from a monoidal profunctor with  $\otimes = \times$ , which combines grates and traversals. It will be called a *monocle*.

**type**  $Monocle\ s\ t\ a\ b = \forall p.MonoPro\ p \Rightarrow p\ a\ b \rightarrow p\ s\ t$

A *Monocle* locates every position of a product (tuple) type (which can be generalized to a finite vector [6]) while a *Lens* locates parts of it. We now list some basic monocles to observe their behavior. Furthermore, one can observe that those monocles can be generalized if dependent types are used.

$each2 :: MonoPro\ p \Rightarrow p\ a\ b \rightarrow p\ (a,a)\ (b,b)$   
 $each2\ p = p\ \star\ p$   
 $each3 :: MonoPro\ p \Rightarrow p\ a\ b \rightarrow p\ (a,a,a)\ (b,b,b)$   
 $each3\ p = dimap\ flat3i\ flat3l\ (p\ \star\ p\ \star\ p)$

$$\begin{aligned} \text{each4} &:: \text{MonoPro } p \Rightarrow p \ a \ b \rightarrow p \ (a, a, a, a) \ (b, b, b, b) \\ \text{each4 } p &= \text{dimap flat4i flat4l } (p \star p \star p \star p) \end{aligned}$$

As one can observe, *each2* deals with parallel composition of the argument  $p$  with itself using the *MonoPro* interface. The focus is on tuples of size 2. The monacles *each3* and *each4* deal with tuples of size 3 and 4 and depend on the tuple flattening functions.

$$\begin{aligned} \text{flat3l} &:: ((a, b), c) \rightarrow (a, b, c) \\ \text{flat3l } ((a, b), c) &= (a, b, c) \\ \text{flat3i} &:: (a, b, c) \rightarrow ((a, b), c) \\ \text{flat3i } (a, b, c) &= ((a, b), c) \\ \text{flat4l} &:: (((a, b), c), d) \rightarrow (a, b, c, d) \\ \text{flat4l } (((a, b), c), d) &= (a, b, c, d) \\ \text{flat4i} &:: (a, b, c, d) \rightarrow (((a, b), c), d) \\ \text{flat4i } (a, b, c, d) &= (((a, b), c), d) \end{aligned}$$

To have a better understanding of those Monocles we adapt the representation theorem of Jaskelioff and O'Connor [6] from functors to profunctors.

**Theorem 6.1** ([6]). (*Unary representation for profunctors*) Consider an adjunction between profunctors  $-^* \vdash U : \mathcal{E} \rightarrow \mathcal{F}$ , where  $\mathcal{F}$  is small and  $\mathcal{E}$  is a full subcategory of  $\text{Prof}(\text{Set}, \text{Set})$ , the family of profunctors  $\text{Iso}_{A,B}(S, T) = (S \rightarrow A) \times (B \rightarrow T)$  gives the following isomorphism natural in  $A, B$ , and dinatural in  $S, T$ .

$$\int_P UP(A, B) \rightarrow UP(S, T) \cong \text{Iso}_{A,B}^*(S, T),$$

where  $\text{Iso}^*$  is the free profunctor generated by  $\text{Iso}$ . Whenever the left-hand side end exists, the isomorphism holds.

Since the free monoidal profunctor exists and is of the form

$$P^*(S, T) = (J + P \star P^*)(S, T),$$

this theorem helps us find the unary representation for monoidal profunctors.

**Proposition 6.2.** *The unary representation for monoidal profunctors is given by the isomorphism:*

$$\int_P P(A, B) \rightarrow P(S, T) \cong \sum_{n \in \mathbb{N}} (S \rightarrow A^n) \times (B^n \rightarrow T)$$

where  $P$  ranges over all monoidal profunctors.

*Proof Sketch.* By theorem 6.1, it is enough to show that

$$U(\text{Iso}_{A,B}^*)(S, T) \cong \sum_{n \in \mathbb{N}} (S \rightarrow A^n) \times (B^n \rightarrow T).$$

This is proven using Yoneda, the fact that Day convolution preserves coproducts, and induction on  $n$ . □

The right-hand side of the above isomorphism tells us that the source type can be split as a finite number of copies of a type  $a$ , and a finite number of copies of a type  $b$  can be amalgamated in a type  $t$ , which gives us a semantic for *Monocles* such as *each2*. Using categorical tools to reason about *each2* (the other functions can be understood similarly), one can rewrite it as follows.

$$\begin{aligned} \text{each2} &:: \text{MonoPro } p \Rightarrow (s \rightarrow (a, a)) \rightarrow ((b, b) \rightarrow t) \rightarrow p \ a \ b \rightarrow p \ s \ t \\ \text{each2 } f \ g \ p &= \text{dimap } f \ g \ (p \star p) \end{aligned}$$

Since theorem 6.1 holds, one can interchangeably use the *Monocle* or a more easy to reason type as  $(s \rightarrow (a, a), (b, b) \rightarrow t)$ , but the limitation of the latter is obvious.

Actions can now be performed on a monocle, given the desired location; one can read/write any product (tuple) type.

$$\begin{aligned} \text{foldOf} &:: \text{Monoid } a \Rightarrow \text{Monocle } s \ t \ a \ b \rightarrow s \rightarrow a \\ \text{foldOf } \text{monocle} &= \text{runForget } (\text{monocle } (\text{Forget } \text{id})) \end{aligned}$$

This action tells that given a *Monocle* (location) one can monoidally collect many parts  $a$  from the big structure  $s$  (in this case, tuples). For example,

$$\text{foldOf } \text{each3} :: \text{Monoid } a \Rightarrow (a, a, a) \rightarrow a$$

behaves in the same way as the function *fold* does with lists, its evaluation on the value ("AA", "BB", "CC") gives "AABBCC" as expected. The list function *foldMap* has a corresponding *Monocle* called *foldMapOf*,

$$\begin{aligned} \text{foldMapOf} &:: \text{Monoid } r \Rightarrow \text{Monocle } s \ t \ a \ b \rightarrow (a \rightarrow r) \rightarrow s \rightarrow r \\ \text{foldMapOf } \text{monocle } f &= \text{runForget } (\text{monocle } (\text{Forget } f)) \end{aligned}$$

which can locate all elements of a 3-element tuple with the expression

$$\text{foldMapOf } \text{each3} :: \text{Monoid } r \Rightarrow (a \rightarrow r) \rightarrow (a, a, a) \rightarrow r$$

as expected.

Every profunctorial optic has a so-called van Laarhoven [18] functorial representation. For a monocle, this representation can be obtained by the following function.

$$\begin{aligned} \text{convolute} &:: (\text{Applicative } g, \text{Functor } f) \Rightarrow \text{Monocle } s \ t \ a \ b \rightarrow (f \ a \rightarrow g \ b) \rightarrow f \ s \rightarrow g \ t \\ \text{convolute } \text{monocle } f &= \text{unSISO } (\text{monocle } (\text{SISO } f)) \end{aligned}$$

If we specialize *convolute* using the identity functor  $f = \text{Id}$ , one gets the definition of a *Traversal*, which is defined in the lens package [8].

$$\begin{aligned} \text{traverseOf} &:: \text{Applicative } g \Rightarrow \text{Monocle } s \ t \ a \ b \rightarrow (\text{Id } a \rightarrow g \ b) \rightarrow (\text{Id } s \rightarrow g \ t) \\ \text{traverseOf } \text{monocle} &= \text{convolute } \text{monocle} \end{aligned}$$

One can specialize *convolute* using the applicative functor  $g = \text{Id}$ , to get the van Laarhoven representation for grates (which depends on a Closed type class of Profunctors) [17].

$$\begin{aligned} \text{class } \text{Profunctor } p &\Rightarrow \text{Closed } p \text{ where} \\ \text{closed} &:: p \ a \ b \rightarrow p \ (x \rightarrow a) \ (x \rightarrow b) \\ \text{zipFWithOf} &:: \text{Functor } f \Rightarrow \text{Monocle } s \ t \ a \ b \rightarrow (f \ a \rightarrow \text{Id } b) \rightarrow (f \ s \rightarrow \text{Id } t) \\ \text{zipFWithOf } \text{monocle} &= \text{convolute } \text{monocle} \end{aligned}$$

Monoidal profunctors with  $\otimes = \times$  capture the essence of a grate and a traversal. Grates have a structured contravariant part (input) while traversals, the covariant one (output), while a monocle has both structures.

## 7 Effectful Monoidal Profunctors

The typeclass for monoidal profunctors *MonoPro* is defined in terms of a profunctor  $p$  over the (fictitious) base category of Haskell types and functions usually known as *Hask*. However, the Day convolution allows us to use morphisms from other categories, instead of using *Hask* everywhere. This section presents a generalization of the class *MonoPro* which allows to use morphisms from other categories. We illustrate its use by applying it to morphisms from a Kleisli category, hence allowing effects to be lifted into the structure. The profunctor class will also have a modified form to lift two abstract morphisms instead of pure functions.

```
class Category k  $\Rightarrow$  CatProfunctor k p where
  catdimap :: k a b  $\rightarrow$  k c d  $\rightarrow$  p b c  $\rightarrow$  p a d
```

A *CatProfunctor* represents a profunctor working with morphisms on an arbitrary category  $\mathcal{C}$  instead of *Hask*, and provides an interface to lift two of those abstract morphisms defined by the binary type constructor  $k$ . This new class needs to be a multi-parameter type class because of the added constraint *Category*.

```
class (Category k, CatProfunctor k p)  $\Rightarrow$  CatMonoPro k p | p  $\rightarrow$  k where
  cmpunit :: k s ()  $\rightarrow$  k () t  $\rightarrow$  p s t
  convolute :: k s (a, c)  $\rightarrow$  k (b, d) t  $\rightarrow$  p a b  $\rightarrow$  p c d  $\rightarrow$  p s t
```

It is good to remember that in the *CatMonoPro* class, the type of *cmunit* is isomorphic to  $p () ()$ , and the type of *convolute* is isomorphic to  $p (a, c) (b, d)$ . The functional dependency  $p \rightarrow k$  allows to write the unit *cmpempty* having the same role as *mpempty*, and  $\star\star$  also having the same role as *MonoPro*'s  $\star$  satisfying the same laws as seen before.

```
cmpempty :: p () ()
cmpempty = unitmp id id
( $\star\star$ ) :: CatMonoPro k p  $\Rightarrow$  p a b  $\rightarrow$  p c d  $\rightarrow$  p (a, c) (b, d)
p  $\star\star$  q = convolute id id p q
```

As an example, one can work with *CatProfunctor* and *CatMonoPro* alongside a Kleisli arrow. That is, objects are types but morphisms are Kleisli arrows. The *CatProfunctor* instance in this example will permit computations to be lifted covariantly and contravariantly. The *CatMonoPro* gives a convolutional effect for computations and not just pure functions (as in *MonoPro*'s *rlmap*).

The Kleisli arrow is just a wrapped type:

```
newtype Kleisli m a b = Kleisli { runKleisli :: a  $\rightarrow$  m b }
```

having a lawful *Category* instance as follows.

```
instance Monad m  $\Rightarrow$  Category (Kleisli m) where
  id = Kleisli return
  (Kleisli bmc)  $\circ$  (Kleisli amb) = Kleisli ( $\lambda a \rightarrow (amb a) \gg\gg bmc$ )
```

A datatype called *Lift* is a *CatProfunctor* with respect to the Kleisli arrow.

```
newtype Lift t m a b = Lift { runLift :: m a  $\rightarrow$  t m b }
```

This polymorphic type represents a general version of the function *lift* used to lift monadic computations into a monad transformer [3]. A monad transformer is a way to stack two or more monads together in order to enable more than one effectful computation together [7, 11]. By packing lift into a profunctor concerning the Kleisli arrow, one gets ways to precompose and post-compose computations with a monad transformer's inner monad. For example, suppose a monad transformer *FooT t m* (where *m* is a monad). It is possible to use *catdimap* to compose effectful computations in *m* having its results in the monad transformer.

```
instance (MonadT t, Monad m, Monad (t m)) => CatProfunctor (Kleisli m) (Lift t m) where
  catdimap (Kleisli f) (Kleisli g) (Lift h) = Lift $ \ma -> let k = h (ma >>= f)
                                             l = lift o g
                                             in k >>= l
```

As an important note, *Lift* is a *SISO* with  $f = m$ , and  $g = t m$ . For a plain profunctor, *m* works only with a *Functor*, *t* need not be a monad transformer, and *t m* needs only to be an *Applicative*. Hence this instance is substantially different from the mentioned one.

To work with a *CatMonoPro* instance with respect to the Kleisli arrow, a notion of reordering effects (commutativity) is needed.

```
instance (CommT t, Traversable m, Monad m, Monad (t m)) =>
  CatMonoPro (Kleisli m) (Lift t m) where
  cmpunit (Kleisli f) (Kleisli g) = Lift (\m -> lift (m >>= f >> g ()))
  convolute (Kleisli f) (Kleisli g) (Lift h) (Lift l) =
    Lift $ \ms -> let (ma, mc) = unzip' (ms >>= f)
                    in comm (fmap g (zip' ((h ma), (l mc))))
```

After the *fmap* of function *g*, the remaining expression will have the type  $t m (m a)$ . This can be fixed by building a class giving a function  $comm :: (Monad m, Traversable m) => t m (m a) \rightarrow t m a$  to reorder those effects. A traversable instance is used here to provide that swap but other commutativity notions [7] may be used.

```
class MonadT t => CommT t where
  comm :: (Monad m, Traversable m) => t m (m a) \rightarrow t m a
```

This setup provides a use of the maybe monad transformer *MaybeT* with a monad like *Writer* which is traversable (all necessary type class instances can be found in the `mt1` [3]).

```
instance CommT MaybeT where
  comm (MaybeT mna) = MaybeT (mna >>= sequence)
```

Using the monoidal profunctor *Lift* helps us deal with multiple monads at once, enabling us to deal with product types' computations. Consider the effectful function *lsplit*, in writer monad, that splits in two a list of *String* (or any *Ord* instance) by its order (in this case, lexicographical). One list for elements less or equal to the head, and the other has bigger elements than the head. The effect is logging, telling what the function is doing for debugging purposes.

```
lsplit :: [String] \rightarrow Writer [String] ([String], [String])
lsplit (z:zs) = do
```

```

xs ← return (filter (<z) zs)
ys ← return (filter (≥ z) zs)
tell ["Splitting: " ++ show zs ++ " into " ++ show xs ++ ", " ++ show ys]
return (xs,ys)

```

The function *rsplit* below just concatenates two lists and logs this action.

```

rsplit :: String → ([String],[String]) → Writer [String] [String]
rsplit l (xs,ys) = do
  tell ["Merging: " ++ show xs ++ ", " ++ l ++ ", and " ++ show ys]
  return (xs ++ [l] ++ ys)

```

A quicksort can be logged and stopped if an invalid value is found while keeping the logs of what the algorithm did (in this case an empty string). The instance of *CatMonoPro* helps with the splitting, and merging the sorted results inside this *MaybeT/Writer* context easily.

```

qsort :: [String] → MaybeT (Writer [String]) [String]
qsort [] = return []
qsort xs = do
  guard (head xs ≠ "")
  (ls,rs) ← runLift (lconvolute (Kleisli lsplit) lift' lift') (return xs)
  (ls',rs') ← runKleisli ((Kleisli qsort) * (Kleisli qsort)) (ls,rs)
  ss ← runLift (rconvolute (Kleisli (rsplit (head xs))) lift' lift') (return (ls',rs'))
  return ss

```

The functions *lconvolute* and *rconvolute* have *id* (from typeclass *Category*) function in the left and right similar as *lmap<sub>2</sub>* and *rmap<sub>2</sub>*. For example, *lconvolute f = convolute f id* and the right convolution is similar changing the *id* order. It is also possible to observe the use of a plain monoidal profunctor by using its multiplication since *qsort* is a Kleisli arrow which has a trivial *MonoPro* instance (it is isomorphic to a SISO with *f = Identity* and *g = MaybeT (Writer [String])*). Finally, the expression *lift'* is *Lift lift'*, that is, the monad transformer's *lift* in the monoidal profunctor setting.

## 8 Related Work

Several of the results collected in this article are folklore. In the following, we point out the cases where a publication mentioning them is known to us. Rivas and Jaskelioff introduce notions of computation as monoids [20], and this work follows their path presenting another notion that is not in that work and that has not been well explored in the community. The type class *MonoPro* is defined by Pickering, Gibbons, and Wu [19], but it was not their main focus. The representation of Monocles is also found in the work of Pickering, Gibbons and Wu [19] which states the relationship between a *Monocle* and *Traversals* by using an optic called *TraversalP*. A similar representation of a Monocle was found in the work of O'Connor [17] and is called *FiniteGrate*. This representation relies on a type class called *Power*, isomorphic to *MonoPro*, which deals with tuples that use type-level naturals. Also in O'Connor's work, there is a mention of the type  $(\text{Functor } f, \text{Applicative } g) \Rightarrow f \ a \rightarrow g \ b$  which is called *SISO* here. A profunctor optic called *Traversables* is defined by Román [21] which is also similar to Monocles. However, in this work we study for the first time the monoidal profunctor semantics of the mentioned optics. The effectful monoidal profunctor and its examples are also new.

## 9 Conclusion

We have studied monoidal profunctors as monoids in a monoidal category of profunctors, shown that such category is symmetric closed, and obtained the free monoidal profunctor. We have shown how to implement monoidal profunctors in Haskell and have shown some applications related to optics and a generalization to apply monoidal profunctors in a Kleisli category that we call effectful monoidal profunctors. We would like to investigate applications of this extension to parallel and concurrent algorithms.

## References

- [1] Bryce Clarke, Derek Elkins, Jeremy Gibbons, Fosco Loregiàn, Bartosz Milewski, Emily Pillmore & Mario Román (2020): *Profunctor optics, a categorical update*. CoRR abs/2001.07488. arXiv:2001.07488.
- [2] Brian Day (1970): *On closed categories of functors*. In S. MacLane, H. Applegate, M. Barr, B. Day, E. Dubuc, Phreilambud, A. Pultr, R. Street, M. Tierney & S. Swierczkowski, editors: *Reports of the Midwest Category Seminar IV*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–38, doi:10.1007/BFb0079385.
- [3] Andy Gill: *mtl: Monad classes, using functional dependencies*. <https://hackage.haskell.org/package/mtl>. Accessed: 2019-05-28.
- [4] John Hughes (2005): *Programming with Arrows*. In: *Proceedings of the 5th International Conference on Advanced Functional Programming*, AFP’04, Springer-Verlag, Berlin, Heidelberg, pp. 73–129, doi:10.1007/11546382\_2.
- [5] Bart Jacobs, Chris Heunen & Ichiro Hasuo (2009): *Categorical semantics for arrows*. *Journal of Functional Programming* 19(3-4), p. 403–438, doi:10.1017/S0956796809007308.
- [6] Mauro Jaskelioff & Russell O’Connor (2015): *A representation theorem for second-order functionals*. *Journal of Functional Programming* 25, p. e13, doi:10.1017/S0956796815000088.
- [7] Mark P. Jones & Luc Duponcheel (1993): *Composing Monads*. Technical Report.
- [8] Edward Kmett: *lens: Lenses, Folds and Traversals*. <https://hackage.haskell.org/package/lens>. Accessed: 2019-05-28.
- [9] T van Laarhoven: *Where do I get my non-regular types?* <http://twanvl.nl/blog/haskell/non-regular2>. Accessed: 2020-08-08.
- [10] Tom Leinster (2003): *Higher Operads, Higher Categories*. arXiv:math/0305049.
- [11] Sheng Liang, Paul Hudak & Mark Jones (1995): *Monad Transformers and Modular Interpreters*. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’95, Association for Computing Machinery, New York, NY, USA, p. 333–343, doi:10.1145/199448.199528.
- [12] Sam Lindley, Philip Wadler & Jeremy Yallop (2011): *Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous*. *Electronic Notes in Theoretical Computer Science* 229(5), pp. 97 – 117, doi:10.1016/j.entcs.2011.02.018. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).
- [13] Fosco Loregian (2020): *Coend calculus*. arXiv:1501.02503.
- [14] Conor McBride & Ross Paterson (2008): *Applicative Programming with Effects*. *J. Funct. Program.* 18(1), pp. 1–13, doi:10.1017/S0956796807006326.
- [15] Bartosz Milewski: *Free Monoidal Profunctors*. <https://bartoszmilewski.com/2018/02/20/free-monoidal-profunctors>. Accessed: 2019-10-20.
- [16] Eugenio Moggi (1991): *Notions of Computation and Monads*. *Inf. Comput.* 93(1), pp. 55–92, doi:10.1016/0890-5401(91)90052-4.



- [17] Russell O'Connor: *Grate: A new kind of Optic*. <https://r6research.livejournal.com/28050.html>. Accessed: 2019-02-02.
- [18] Russell O'Connor: *A Representation Theorem for Second-Order Pro-functionals*. <https://r6research.livejournal.com/27858.html>. Accessed: 2019-02-01.
- [19] Matthew Pickering, Jeremy Gibbons & Nicolas Wu (2017): *Profunctor Optics: Modular Data Accessors*. *The Art, Science, and Engineering of Programming* 1(2), doi:10.22152/programming-journal.org/2017/1/7.
- [20] Exequiel Rivas & Mauro Jaskelioff (2017): *Notions of computation as monoids*. *Journal of Functional Programming* 27, p. e21, doi:10.1017/S0956796817000132.
- [21] Mario Román (2020): *Profunctor optics and traversals*. *ArXiv* abs/2001.08045.
- [22] Philip Wadler (1992): *The Essence of Functional Programming*. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '92*, ACM, New York, NY, USA, p. 1–14, doi:10.1145/143165.143169.