Folds, Scans, and Moore Machines as Monoidal Profunctor Homomorphisms

Alexandre Garcia de Oliveira
Fatec-Rubens Lara
Santos, Brazil
alexgrcol (at) hotmail.com

This work focuses on utilizing monoidal profunctor homomorphisms to establish connections between folds, scans, and Moore machines, employing monoidal profunctor homomorphisms as a fundamental tool for theoretical reasoning. Despite the recognized versatility of monoidal profunctors in other areas of functional programming, their application in linking these specific computational models has not been extensively explored. Folds and scans are analyzed as instances of a specific monoidal profunctor known as SISO (Structured Input-Structured Output). It is demonstrated that a Moore machine can also be effectively described as a lawful monoidal profunctor. This work establishes a clear connection by proving that there are structure-preserving maps from the SISO monoidal profunctor, representing folds and scans, to the Moore monoidal profunctor, therefore characterizing this relationship as a homomorphism. This exploration not only enhances the structuring of lawful and comprehensible programs but also fills a significant gap by establishing the utility of monoidal profunctors in a new context. This work asserts that the methodologies developed here can be applied to understand other complex computational processes and their laws.

1 Introduction

Monoidal profunctors play a key role in functional programming, especially for managing well-behaved parallel computations [11, 9]. A monoidal profunctor is based on a polymorphic type with two variables and share characteristics with monoidal functors, also known as applicative functors. However, monoidal profunctors are more flexible than arrows, which can handle both pure and sequential computations. Monoidal profunctors are designed to manage data in tuples and can lift functions of any arity, whether covariant or contravariant, unlike regular profunctors, which are limited to functions of a single arity.

Monoidal profunctors can also be viewed as monoids within the monoidal category of profunctors, especially when using Day convolution as the tensor product. This gives them another notion of computation as monoids [12].

In terms of applications, monoidal profunctors have been used in tools like Opaleye, a domain-specific language for databases, and Monocle, which applies concepts to profunctorial optics [9, 11].

This paper explores an interesting application of monoidal profunctors through the combined use of folds, scans, and Moore machines. It also explores how these elements help create mappings that preserve structure between two monoidal profunctors, known as monoidal profunctor homomorphisms. In functional programming, following rules and structures rigorously makes programs easier to understand and reason about. This paper's approach helps clarify complex concepts through a categorical perspective, aiding in the understanding of such structures. While the use of monoidal profunctor homomorphisms in functional programming is not new, their application in connecting constructs such as folds, scans, and Moore machines represents an unexplored area. For transparency and reproducibility, we have made our complete Agda code and formal proofs publicly available. Interested readers can

Submitted to:
MSFP 2024

access the repository on GitHub to examine the formalization [8]. This work aims to fill this gap by developing and demonstrating a new methodology that not only elucidates but also leverages the use of monoidal profunctors to reason about functional programs.

2 Mathematical background

This work assumes that the reader has some basic understanding about Category Theory [7] such as functors, monads, and natural transformations.

2.1 Monoidal Categories and Monoids

A *monoidal category* [12] gives us a minimal framework for defining the categorical version of a monoid.

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

- *$\mathscr{C}$ is a category;*
- *$\otimes : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$ is a bifunctor;*
- *I is an object of $\mathscr{C}$ called unit;*
- *$\rho_A : A \otimes I \to A$, $\lambda_A : I \otimes A \to A$ and $\alpha_{ABC} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$ are three natural isomorphisms such that the diagrams below commute.*

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

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

If there is a natural isomorphism $\gamma_{AB} : A \otimes B \to B \otimes A$ the monoidal category is called *symmetric*.

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

1. *Right unit: $m \circ (id \otimes e) = \rho_M$*
2. *Left unit: $m \circ (e \otimes id) = \lambda_M$*
3. *Associativity: $m \circ (m \otimes id) = m \circ (id \otimes m) \circ \alpha_M$*

Given the categorical concept of a monoid, we can now define a structure-preserving map between these monoids, known as a monoid homomorphism.

Definition 2.3. *Given two monoids (M_1, m_1, e_1) and (M_2, m_2, e_2) in a monoidal category $\mathscr{C}$, a* monoid homomorphism *between them is an arrow $f : M_1 \to M_2$ in $\mathscr{C}$ such that the following diagram commutes:*

$$\begin{array}{ccc}
M_1 \otimes M_1 & \xrightarrow{m_1} & M_1 \\
{\scriptstyle f \otimes f}\downarrow & & \downarrow{\scriptstyle f} \\
M_2 \otimes M_2 & \xrightarrow{m_2} & M_2
\end{array}
\qquad
\begin{array}{ccc}
I & \xrightarrow{e_1} & M_1 \\
& {\scriptstyle e_2} \searrow & \downarrow{\scriptstyle f} \\
& & M_2
\end{array}$$

Monoids in a monoidal category $\mathscr{C}$, together with monoid homomorphisms, form the category $Mon(\mathscr{C})$ [12].

2.2 Profunctors

Definition 2.4. *Given two categories $\mathscr{C}$ and $\mathscr{D}$, a* profunctor *[6] from $\mathscr{C}$ to $\mathscr{D}$ is a* functor $P : \mathscr{C}^{op} \times \mathscr{D} \to Set$. *Explicitly, it consists of:*

- *for each a object of $\mathscr{C}$ and b object of $\mathscr{D}$, a set $P(a,b)$;*
- *for each a object of $\mathscr{C}$ and b,d objects of $\mathscr{D}$, a function (left action) $\mathscr{D}(d,b) \times P(a,d) \to P(a,b)$;*
- *for all a,c objects of $\mathscr{C}$ and b object of $\mathscr{D}$, a function (right action) $P(a,b) \times \mathscr{C}(c,a) \to P(c,b)$.*

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

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

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

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

In Haskell, the Profunctor type class allows mapping over both input (contravariant) and output (covariant) types, enabling the lifting of pure functions to work with processes. When implementing an instance of Profunctor, the profunctor laws must hold.

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

The function *dimap* relates to the definition of a profunctor by combining both left and right actions. However, these actions also exist separately in Haskell's profunctor library as *lmap* and *rmap*, with the types $lmap :: Profunctor\ p \Rightarrow (c \to a) \to p\ a\ b \to p\ c\ b$ and $rmap :: Profunctor\ p \Rightarrow (b \to d) \to p\ a\ b \to p\ a\ d$.

Definition 2.5. *Let $\mathscr{C}$ and $\mathscr{D}$ be small categories, $Prof(\mathscr{C},\mathscr{D})$ is the profunctor category consisting of profunctors as objects, natural transformations as morphisms, and vertical composition to compose them.*

2.3 Monoidal Profunctors

Knowing what a profunctor is, we now define a monoid in the category of profunctors. To achieve this, we adapt the concept of Day convolution, originally developed for functors [1], to be applicable in the context of profunctors.

Definition 2.6. *Let $\mathscr{C}$ be a small monoidal category and $P,Q : \mathscr{C}^{op} \times \mathscr{C} \to Set$, the* Day convolution *$P \star Q$ of the profunctors P and Q is another profunctor given by*

$$(P \star Q)(S,T) = \int^{ABCD} P(A,B) \times Q(C,D) \times \mathscr{C}(S, A \otimes C) \times \mathscr{C}(B \otimes D, T). \tag{1}$$

The Day convolution functions as a unital and associative tensor product, as detailed in reference [10]. The next two results help to build the notion of a monoidal profunctor, since the first proposition defines the unital notion, and the second one helps to define the multiplication.

Proposition 2.7. *Let $(\mathscr{C},\otimes,I)$ be a small monoidal category, $P : \mathscr{C}^{op} \times \mathscr{C} \to Set$ be a profunctor, and S,T two objects of $\mathscr{C}$. Then $\int_{S,T} Set(J(S,T), P(S,T)) \cong P(I,I)$, where $J(S,T) = (\mathscr{C}^{op} \times \mathscr{C})((I,I),(A,B)) \cong \mathscr{C}(S,I) \times \mathscr{C}(I,T)$.*

Proof. See [10], Proposition 5. □

It is worth noting that the profunctor J serves as the unit of the Day convolution $\star$ [9]. It is also important to understand that Proposition 2.7 provides the theoretical framework for defining the unit of a monoidal profunctor [9].

Proposition 2.8. *Let $\mathscr{D} = \mathscr{C}^{op} \otimes \mathscr{C}$, there is a one-to-one correspondence defining morphisms going out of a Day convolution for profunctors*

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

which is natural in P, Q and R.

Proof. See [10], Proposition 8. □

Proposition 2.8 provides the theoretical foundation for defining the monoidal multiplication of a monoidal profunctor [9]. We can now define the monoidal profunctor by applying the methodology of monoids in a monoidal category.

Definition 2.9. *Let $(\mathscr{C}, \otimes, I)$ be a small monoidal category. A monoid in the profunctor category with the monoidal structure inherited by the Day convolution is a profunctor P, a unit given by the natural transformation between the unit profunctor J and P, $e : J \to P$, equivalent to $e : P(I,I)$ [10], and the multiplication is $m : P \star P \to P$ which is isomorphic to the family of morphisms $V(m)_{ABCD} = P(A,B) \times P(C,D) \to P(A \otimes C, B \otimes D)$ [10]. Such a monoid is called a* monoidal profunctor. *This construction is indeed a monoid [10].*

As an example, consider $(Set, \times, 1)$, where 1 is a singleton set, and the Hom profunctor $P(A,B) = A \to B$, trivially gives us a monoidal profunctor.

3 Monoidal Profunctors in Haskell

In Haskell, the structure of a monoidal profunctor is encapsulated within the following typeclass:

```
class Profunctor p ⇒ MonoPro p where
  mpempty :: p () ()
  (⋆) :: p a b → p c d → p (a,c) (b,d)
```

and it captures the notions of a "parallel" computation with a trivial computation *mpempty*.

Since this is a monoid, this typeclass should follow the monoidal laws.

- Left identity: $dimap\ diagr\ snd\ (mpempty \star f) = f$
- Right identity: $dimap\ diagl\ fst\ (f \star mpempty) = f$
- Associativity: $dimap\ assoc^{-1}\ assoc\ (f \star (g \star h)) = (f \star g) \star h$

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

The simplest example of a monoidal profunctor is the function type $(\to)$.

```
instance MonoPro (→) where
  mpempty = λ() → ()
  f ⋆ g = λ(a,b) → (f a, g b)
```

A structured input and a distinct structured output, as exemplified by the *SISO* type, serve as another illustration of a monoidal profunctor.

```
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)
instance (Functor f, Applicative g) ⇒ MonoPro (SISO f g) where
  mpempty = SISO (λ_ → pure ())
  SISO h ⋆ SISO i = SISO (zip′ ∘ (h ⋆ i) ∘ unzip′)
```

where $zip' :: Applicative\ g \Rightarrow (g\ a, g\ b) \to g\ (a,b)$ 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 $unzip' :: Functor\ f \Rightarrow f\ (a,b) \to (f\ a, f\ b)$, the functions h and i are composed in a parallel manner using the monoidal profunctor instance for $(\to)$ and then regrouped together using the applicative (monoidal) behavior of zip'.

3.1 Moore Machines

The *Moore machine* is a fundamental structure in automata theory and can be defined as a tuple $(S, I, O, s_0, \delta, \zeta)$, where S is the set of states, I is the input alphabet, O is the output alphabet, s_0 is the initial state, $\delta : S \times I \to S$ is the transition function, and $\zeta : S \to O$ is the output function. The behavior of a Moore machine is determined by the input sequences and the corresponding output sequences produced by the output function. Moore machines are known for their simplicity and ease of implementation, making them a popular choice in the design of finite-state machines.

An essential characteristic of Moore machines is that the current state entirely determines the output function, which means that the output is not affected by the input sequence. This is in contrast to Mealy machines, where the output depends on *both* the current state and the input.

```
data Moore a b = Moore b (a → Moore a b)
```

The data constructor above has as arguments an output b and a function to transition the machine from its current state to a new state, depending on the input it receives.

It is easy to see that *Moore* type is a profunctor, and a *MonoPro* by just parallel composing the transitions and collecting the outputs from both machines.

```
instance Profunctor Moore where
  dimap f g (Moore c bm) = Moore (g c) (dimap f g ∘ bm ∘ f)
instance MonoPro Moore where
  mpempty = Moore () (\_ → mpempty)
  (Moore b am) ⋆ (Moore d cm) = Moore (b,d) (λ(a,c) → am a ⋆ cm c)
```

Lemma 3.1. *The type Moore is a monoidal profunctor.*

Proof. We need to prove the unital and associativity laws. Firstly, let's prove the right unital law (the left one is analogous). Consider $f = \mathit{Moore}\ b\ am$. Let us prove that $\mathit{dimap}\ \mathit{diagr}\ \mathit{snd}\ (\mathit{mpempty} \star f) = f$.

$$
\begin{aligned}
&\mathit{dimap}\ \mathit{diagr}\ \mathit{snd}\ (\mathit{mpempty} \star (\mathit{Moore}\ b\ am)) \\
&\quad \{\ \text{Expand the } \star \text{ operation}\ \} \\
&= \mathit{dimap}\ \mathit{diagr}\ \mathit{snd}\ (\mathit{Moore}\ ((),b)\ (\lambda((),a) \to \mathit{const}\ \mathit{mpempty} \star am\ a)) \\
&\quad \{\ \text{Apply } \mathit{dimap} \text{ to Moore}\ \} \\
&= \mathit{Moore}\ (\mathit{diagr}\ ((),b))\ (\mathit{dimap}\ \mathit{diagr}\ \mathit{snd} \circ (\mathit{Moore}\ ((),b)\ (\lambda((),a) \to \mathit{const}\ \mathit{mpempty} \star am\ a) \circ \mathit{snd})) \\
&\quad \{\ \text{The first input is always ignored}\ \} \\
&= \mathit{Moore}\ b\ am \\
&\quad \{\ \text{Identify the Moore machine}\ \} \\
&= f
\end{aligned}
$$

To demonstrate associativity, consider three Moore machines defined as follows: Let $f = \mathit{Moore}\ b\ am$, $g = \mathit{Moore}\ d\ cm$, and $h = \mathit{Moore}\ f\ em$.

Assume that the transition functions satisfy the associativity condition:

$$(am\ a \star cm\ c) \star em\ e = am\ a \star (cm\ c \star em\ e).$$

We will use co-induction to prove that the operation $\star$ is associative across these Moore machines.

$$
\begin{aligned}
&(f \star g) \star h \\
&\quad \{\ \text{Definitions of f, g and h}\ \} \\
&= ((\mathit{Moore}\ b\ am) \star (\mathit{Moore}\ d\ cm)) \star (\mathit{Moore}\ f\ em) \\
&\quad \{\ \text{Definition of } \star \text{ for Moore}\ \} \\
&= \mathit{Moore}\ ((b,d)\ (\lambda(a,c) \to am\ a \star cm\ c) \star (\mathit{Moore}\ f\ em) \\
&\quad \{\ \text{Expand the next } \star\ \} \\
&= \mathit{Moore}\ ((b,d),f)\ (\lambda((a,c),e) \to (am\ a \star cm\ c) \star em\ e) \\
&\quad \{\ \text{Associate state spaces and transition functions}\ \} \\
&= \mathit{Moore}\ (\mathit{assoc}\ (b,(d,f)))\ (\mathit{dimap}\ \mathit{assoc}^{-1}\ \mathit{assoc} \circ (\lambda((a,c),e) \to (am\ a \star cm\ c) \star em\ e) \circ \mathit{assoc}^{-1}) \\
&\quad \{\ \text{Definition of } \mathit{dimap} \text{ (symmetry)}\ \} \\
&= \mathit{dimap}\ \mathit{assoc}^{-1}\ \mathit{assoc}\ (\mathit{Moore}\ (b,(d,f))\ (\lambda(a,(c,e)) \to am\ a \star (cm\ c \star em\ e))) \\
&\quad \{\ \text{Rearrange using associativity}\ \} \\
&= \mathit{dimap}\ \mathit{assoc}^{-1}\ \mathit{assoc}\ ((\mathit{Moore}\ b\ am) \star (\mathit{Moore}\ (d,f)\ (\lambda(c,e) \to cm\ c \star em\ e))) \\
&\quad \{\ \text{Expand } \star \text{ again}\ \} \\
&= \mathit{dimap}\ \mathit{assoc}^{-1}\ \mathit{assoc}\ ((\mathit{Moore}\ b\ am) \star ((\mathit{Moore}\ d\ cm) \star (\mathit{Moore}\ f\ em))) \\
&\quad \{\ \text{Definition of } \star \text{ (symmetry)}\ \} \\
&= \mathit{dimap}\ \mathit{assoc}^{-1}\ \mathit{assoc}\ (f \star (g \star h))
\end{aligned}
$$

□

A Moore machine can be constructed using the following type, representing a coalgebra [13].

```
data MooreCoalg s a b = MooreCoalg (s → b) (s → a → s)
```

where the first argument is the function λ, and the second one δ, with type variables *a* and *b* representing the input and output alphabets.

```
buildMoore :: MooreCoalg s a b → s → Moore a b
buildMoore (MooreCoalg out next) s =
  Moore (out s) (buildMoore (MooreCoalg out next) ∘ next s)
```

To construct a *Moore* datatype, we use the above function *buildMoore* that takes a *MooreCoalg* argument and extracts its state to get the output, and makes a recursive call to obtain the machine transitions. One can easily build a Moore machine this way by simply defining which function determines the machine output, and which function determines the transition.

```
countMoore :: Moore a Int
countMoore = buildMoore (MooreCoalg id (λs _ → s + 1)) 0
```

The above machine ignores every input and returns an updated state by adding 1 to the previous state, its initial state is 0. Note that the output is the identity function meaning that every state will be the output of the machine providing a Moore machine that is a simple counter.

Such Moore machines may be run by transforming them in functions of type $[a] \rightarrow$ *NonEmpty b*. This can be achieved by reading every input and executing the transitions at every step. After a new state is obtained, we append to the returning non-empty list as follows.

```
runMoore :: Moore a b → [a] → NonEmpty b
runMoore (Moore b _) [] = b ▷ []
runMoore (Moore b f) (a : as) = b ▷ go (f a) as
  where
    go :: Moore a b → [a] → [b]
    go (Moore b f) [] = [b]
    go (Moore b f) (a : as) = b : go (f a) as
```

It is important to note that we use a non-empty list version here because it guarantees that the output always contains at least one element, reflecting the initial state of the Moore machine. The inner function *go* recursively processes the input list, producing a list of outputs corresponding to each step of the Moore machine. The same function can be used to give only the final state of the machine without creating an accumulated list of b.

```
runMooref :: Moore a b → [a] → b
runMooref (Moore b _) [] = b
runMooref (Moore _ f) (a : as) = runMooref (f a) as
```

4 Folds and Scans Through Moore Machines and Monoidal Homomorphisms

It is worth noting that in this section, we employ Haskell syntax to articulate categorical concepts within our proofs. To initiate our exploration of the connections between Folds, Scans, and Moore machines via monoidal profunctor homomorphisms, we begin by examining a straightforward example: using a Moore machine to count the number of elements read from the input.

$$> \mathit{runMoore}\ \mathit{countMoore}\ [(),(),(),(),()]$$
$$> 0 \triangleright [1,2,3,4,5]$$

By running the *countMoore* machine with five unit inputs (can be anything, it will be ignored), the return is a list from 0 to 5, meaning that we obtain the initial state 0, and the next five steps that increases the counter by 1.

One can observe that the same can be obtained by using the function *scanl* (for non-empty lists), in Haskell, from *Data.List*.

$$\mathit{runMoore} \circ \mathit{buildMoore}\ (\mathit{MooreCoalg}\ \mathit{id}\ (\lambda s\ \_ \to s+1)) = \mathit{scanl}\ (\lambda s\ \_ \to s+1)$$

That can be generalized as the following rule.

$$\mathit{runMoore} \circ \mathit{buildMoore}\ (\mathit{MooreCoalg}\ \mathit{id}\ f) = \mathit{scanl}\ f$$

If we take a closer look at the function *scanl*, we can observe that it builds and runs a Moore machine simultaneously. If the single parameter b were a function, the types of the first two parameters would match *MooreCoalg*, and the return type would be the same as *runMoore*.

$$\mathit{scanl} :: (b \to a \to b) \to b \to [a] \to \mathit{NonEmpty}\ b$$

Using a *Moore* abstraction allows us to construct more complex ways to do accumulated folds that is not possible using scanl only. Now we can explore the connection between Moore machines, left scans and left folds. First, let us take a look on *foldl*, and *runMoore*.

$$\mathit{foldl} :: (s \to a \to s) \to s \to [a] \to s$$
$$\mathit{foldl}\ \_\ s\ [\,] = s$$
$$\mathit{foldl}\ f\ s\ (a : as) = \mathit{foldl}\ f\ (f\ s\ a)\ as$$

One can observe that *foldl* resembles *runMoore*, to explain this, we observe that the type of this function can be modified to allow an output type b.

$$\mathit{ofoldl} :: (s \to a \to s) \to s \to (s \to b) \to [a] \to b$$
$$\mathit{ofoldl}\ f\ s\ sb\ as = sb\ (\mathit{foldl}\ f\ s\ as)$$

Analyzing the parameters of *ofoldl*, we have an initial state s, a transition $s \to a \to s$, and an output function $s \to b$. If this output function is the identity function *id*, then we recover *foldl*. This is setup is suitable for building Moore machines using *MooreCoalg* type, and the *buildMoore* function.

Furthermore, the return type $[a] \to b$ is a monoidal profunctor, given its equivalence to

type *Fold a b* = *SISO* [ ] *Identity a b*

We now can rewrite the *ofoldl* function as follows.

mfoldl :: *Moore a b* → *Fold a b*
mfoldl m = *SISO* (λ*as* → *Identity* (*runMooref m as*))

Now, we have a function between two monoidal profunctors: *Moore* and *Fold*. This way of writing a fold gives us some reasoning benefits.

Lemma 4.1. *The function mfoldl is a natural transformation between the profunctors Moore and Fold.*

Proof. We need to prove that the following diagram commutes for arbitrary $h : a \to b$, and $i : c \to d$..

$$\begin{array}{ccc} \text{Moore } b\ c & \xrightarrow{dimap\ h\ i} & \text{Moore } a\ d \\ {\scriptstyle mfoldl}\downarrow & & \downarrow{\scriptstyle mfoldl} \\ \text{Fold } b\ c & \xrightarrow[dimap\ h\ i]{} & \text{Fold } a\ d \end{array}$$

We use the subscripts *Fold* and *Moore* for dimap to indicate to the reader which instance will be used to evaluate the expressions. The commuting diagram tells that we need to prove the following rule.

$$dimap_{Fold}\ h\ i \circ mfoldl = mfoldl \circ dimap_{Moore}\ h\ i$$

So, by function extensionality, we only need to prove that given an arbitrary *m* :: *Moore b c*, we have the following equality.

$$dimap_{Fold}\ h\ i(mfoldl\ m) = mfoldl(dimap_{Moore}\ h\ i\ m)$$

We start from the right-hand side of the above equation.

$$\begin{aligned} & mfoldl\ (dimap\ h\ i\ m) \\ = \quad & \{ \text{ definition of } mfoldl \text{ and } m = Moore\ b\ f \ \} \\ & SISO\ (\lambda as \to Identity\ (runMoore\ (dimap\ h\ i\ (Moore\ b\ f))\ as)) \\ = \quad & \{ \text{ definition of } dimap \ \} \\ & SISO\ (\lambda as \to Identity\ (runMoore\ (Moore\ (i\ b)\ (dimap\ h\ i \circ f \circ h)\ as))) \end{aligned}$$

We now see that the term *SISO* (λ*as* → *Identity* (*runMoore* (*Moore* (*i b*) (*dimap h i* ∘ *f* ∘ *h*) *as*))) is running a Moore machine recursively using the function *h* on each input and the function *i* on each output, so that gives us a structurally equal term *SISO* (λ*as* → *Identity* (*i* (*runMoore* (*Moore b f*) (*map h as*)))).

$$\begin{aligned} & SISO\ (\lambda as \to Identity\ (i\ (runMoore\ (Moore\ b\ f)\ (map\ h\ as)))) \\ = \quad & \{ \text{ definition of m, } fmap \text{ and } map \text{ (symmetry) and } m = Moore\ b\ f \ \} \\ & SISO\ (fmap\ i \circ (\lambda as \to Identity\ (runMoore\ m\ as))) \circ fmap\ h) \\ = \quad & \{ \text{ definition of } dimap \text{ (symmetry) } \} \\ & dimap\ h\ i\ (SISO\ (\lambda as \to Identity\ (runMoore\ m\ as))) \\ = \quad & \{ \text{ definition of } mfoldl \text{ (symmetry) } \} \\ & dimap\ h\ i\ (mfoldl\ m) \end{aligned}$$

□

Lemma 4.1 gives us the corresponding *fusion law* [3] for foldl.

$$foldl\ op\ e \circ map\ f = foldl\ (\lambda s\ a \to op\ s\ (f\ a))\ e$$

but, using the lemma above we can write both sides using *mfoldl*. The left-hand side of the above law is the term $foldl\ op\ b \circ map\ f$, which states that we map a function f to the input list and then fold it, this is precisely what happens with a *Fold*, this same behavior is achived by the term $dimap\ f\ id \circ mfoldl$. Conversely, the term $foldl\ (\lambda s\ a \to op\ s\ (f\ a))\ e$, which gives us the same behavior as acting on the input of a Moore machine, so the analogous term is $mfoldl \circ dimap\ f\ id$. Hence, the fusion law is a corollary of Lemma 4.1.

$$dimap\ f\ id \circ mfoldl = mfoldl \circ dimap\ f\ id$$

The exact same reasoning can be done to treat scanls as a natural transformation between two profunctors. In this case, we have this transformation between *Moore* and $Scan\ a\ b = SISO\ [\,]\ ZipNonEmpty\ a\ b$. We use a *ZipNonEmpty* instead of $[\,]$ because its *Applicative* instance provides the "zippy" behavior to a list.

```
mscanl :: Moore a b → Scan a b
mscanl m = SISO (λas → ZipNonEmpty (runMoore m as))
```

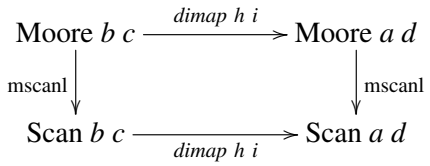
The definition of *ZipNonEmpty* follows the same pattern as *ZipList*, but with a modification to the Applicative instance. We change the *Applicative* instance to use the behavior of a *zip*, rather than the original applicative behavior of *NonEmpty*, which is similar to that of a list. This ensures that the elements are combined pairwise, aligning with the expected zip functionality.

```
data NonEmpty a = a ▷ [a]
data ZipNonEmpty a = ZipNonEmpty { unzipne :: NE.NonEmpty a }
instance Functor ZipNonEmpty where
  fmap f (ZipNonEmpty (a ▷ as)) =
    ZipNonEmpty ((f a) ▷ fmap f as)
instance Applicative ZipNonEmpty where
  pure a = ZipNonEmpty (a ▷ repeat a)
  (ZipNonEmpty (f ▷ fs)) ⊗ (ZipNonEmpty (x ▷ xs))
    = ZipNonEmpty ((f x) ▷ zipWith ($) fs xs)
```

Now we prove lemmas about *mscanl* as well, showing that this function is also a natural transformation that preserves the monoidal profunctor structure.

Lemma 4.2. *The function mscanl is a natural transformation between the profunctors Moore and Scan.*

Proof. The proof procceds with the same reasoning as Lemma 4.1, thus one needs to show that the following diagram commutes.



□

This lemma gives us that for any $h : a \to b$, and $i : c \to d$ we have $dimap\ h\ i \circ mscanl = mscanl \circ dimap\ h\ i$.

The *mfoldl* function in our structure respects the unit *mpempty* and the monoidal multiplication inherent to a monoidal profunctor.

Lemma 4.3. *The functions mfoldl, and mscanl preserve mpempty.*

$$mfoldl\ mpempty = mpempty$$

$$mscanl\ mpempty = mpempty$$

Proof. Firstly, we notice that $mfoldl\ mpempty = runMoore\ (Moore\ ()\ (\backslash\_ \to mpempty))$, the RHS is the *mpempty* of a function type, that is *const* (). The left-hand side clearly produces only (), and the constant function with () as argument will also do so. Thus, the equation holds for *mfoldl*. Since the only production is (), the equation will also hold for *mscanl*. □

Lemma 4.4. *The functions mfoldl, and mscanl preserve* $\star$.

$$mfoldl\ (m \star n) = mfoldl\ m \star mfoldl\ n$$

$$mscanl\ (m \star n) = mscanl\ m \star mscanl\ n$$

Proof. First we prove the identity for *mfoldl*. Given $m = Moore\ b\ am :: Moore\ a\ b$, and $n = Moore\ d\ cm :: Moore\ c\ d$, we know that $m \star n = Moore\ (b,d)\ (\lambda(a,c) \to am\ a \star cm\ c)$. Hence,

$$mfoldl\ (m \star n) = \lambda ls \to runMoore\ (Moore\ (b,d)\ (\lambda(a,c) \to am\ a \star cm\ c)\ ls),$$

and

$$\begin{aligned}
& mfold\ m \star mfoldl\ n \\
= \quad & \{\ \text{definition of mfold}\ \} \\
& \lambda ls \to zip'\ ((\lambda as \to Identity\ (runMoore\ m\ (fst\ as)) \star \\
& \qquad (\lambda cs \to Identity\ (runMoore\ n\ (snd\ cs))) \\
& \qquad (unzip\ ls))) \\
= \quad & \{\ \text{definition of } zip' \text{ and } \star\ \} \\
& \lambda ls \to (runMoore\ (fst\ (unzip\ ls)), runMoore\ (snd\ (unzip\ ls))
\end{aligned}$$

We need to prove now that for any $ls :: [(a,c)]$, we get the following.

$$\begin{aligned}
& runMoore\ (Moore\ (b,d)\ (\lambda(a,c) \to am\ a \star cm\ c)\ ls = \\
& \quad runMoore\ (fst\ (unzip\ ls)), runMoore\ (snd\ (unzip\ ls)
\end{aligned}$$

For $ls = [\,]$, we clearly have that both sides of the equation have the same state, giving us the base case. Assume the equation holds for a list $ls = zs$, and we want to show that the equation holds for the prepended input pairs $ls = (x,y) : zs$ which both having types $x :: a$ and $y :: c$ respectively. We first apply the transition functions *am* and *bm* of Moore machines *m* and *n* to the first components *x* and *y* of the input, respectively. Then, we use the induction hypothesis on the rest of the input list zs to prove that

the equation holds for the entire input list $x : zs$. By function extensionality, we get the desired result for. Thus, $mfoldl\ (m \star n) = mfoldl\ m \star mfoldl\ n$.

Since *mscanl* is simply a variation of *mfoldl* that accumulates the outputs instead of only returning the final output, we can observe that we will have the same results for every list input, so the result also holds for *mscanl*. □

Lemma 4.4 indicates that folding over a list of pairs using a combined folding function is equivalent to folding over two separate lists using individual folding functions and combining the results using the expressiveness of the *MonoPro* interface. Translating to plain fold, we get the following law.

$$\begin{aligned} &foldl\ (dblSwap\ (uncurry\ f \star uncurry\ g))\ (e,u)\ (zip\ as\ bs) \\ &= (foldl\ f\ e\ as, foldl\ g\ u\ bs) \\ &\textbf{where}\ dblSwap = curry\ (lmap\ (\lambda((a,b),(c,d)) \to ((a,c),(b,d)))) \end{aligned}$$

One can observe that the left-hand side is sometimes called a bifold. A bifold is a function that combines two separate folds into a single operation. This idea can be used to simplify the law, making it easier to state and understand.

$$bifoldl\ f\ g\ (e,u)\ (zip\ as\ bs) = (foldl\ f\ e\ as, foldl\ g\ u\ bs)$$

Lemma 4.5. *The functions mfoldl and mscanl are monoidal profunctor homomorphisms, i.e., they preserve mpempty and* $\star$.

Proof. This follows directly from Lemmas 4.1, 4.2, 4.3 and 4.4. □

The above result allows us to reason about left folds as a categorical construct, specifically as a monoidal profunctor. By connecting folds and Moore machines through a monoidal profunctor homomorphism, we generalized fold/scan laws using this categorical framework, illustrating how specific instances of monoidal profunctors yield these laws.

5 Related Work

Previous research by Gibbons [2], Hinze [3], and others has extensively analyzed the algebraic properties and laws governing fold-like structures in functional programming. In Hinze's work, the focus was on using category theory and algebraic structures to deepen the understanding of folds, providing a systematic method to derive folding functions and their properties. Gibbons, on the other hand, explored the connection between folds and origami programming, emphasizing how transformations within data structures can be elegantly modeled using folding operations.

Moore machines, derived from automata theory, benefit from a coalgebraic encoding, as detailed by Jacobs [4]. This encoding facilitates more intuitive reasoning about these machines within the context of functional programming. In Haskell, they are represented in the machines package developed by Edward Kmett [5]. This package utilizes Moore machines to facilitate the construction of efficient, modular data processing pipelines, allowing for a functional approach to stream processing and state management. This package also provides a profunctor instance to a Moore machine type.

6 Conclusion

This study has successfully established a theoretical connection between folds, scans, and Moore machines with the framework of a monoidal profunctor homomorphism. By demonstrating how these computational models can be coherently unified under the concept of monoidal profunctors, we provide a robust categorical foundation that enhances understanding and utility of lawful computations in functional programming.

Our exploration suggests that further research into the monoidal profunctor structure could benefit the community. The complex nature and less common usage of monoidal profunctors mean that the results found are based on a foundation that requires deeper and more widespread study to fully realize its potential. In this work, we have demonstrated that folds are equivalent to *SISO* [] *Identity*, while scans correspond to *SISO ZipList ZipList*. These constructs relate to a Moore machine through the monoidal profunctor homomorphism. With a deeper understanding of monoidal profunctors, it is possible to extend this analysis further. As illustrated here, by utilizing instances of a monoidal profunctor, one can derive additional laws that govern the behavior of the desired object. For instance, this methodology can also be applied to derive laws for unfolds, further highlighting the relevance of this study.

For future work, there is a clear trajectory for extending this research to discover and utilize additional examples of monoidal profunctor instances. Exploring other structures within the realm of monoidal profunctors can potentially uncover new ways to derive similar laws and patterns. Furthermore, expanding the theoretical framework to include more diverse monoidal profunctor structures could yield richer interactions and more sophisticated tools for reasoning about pure functional programs.

References

[1] Brian Day & Max Kelly (1969): *Enriched functor categories*. *Reports of the Midwest Category Seminar III (Lecture Notes in Mathematics)* Volume 106, doi:10.1016/0021-8693(68)90037-9.

[2] Jeremy Gibbons (2003): *Origami Programming*. In Jeremy Gibbons & Oege de Moor, editors: *The Fun of Programming*, Palgrave Macmillan, pp. 41–60. Available at `https://ora.ox.ac.uk/objects/uuid:b86dbc7f-f5aa-4ffd-a711-bfc8f75aa4f2`.

[3] Ralf Hinze (2013): *Adjoint folds and unfolds—An extended study*. *Science of Computer Programming* 78(11), p. 2108–2159, doi:10.1016/j.scico.2012.07.011. Available at `http://dx.doi.org/10.1016/j.scico.2012.07.011`.

[4] Bart Jacobs (2006): *A Bialgebraic Review of Deterministic Automata, Regular Expressions and Languages*, p. 375–404. Springer Berlin Heidelberg, doi:10.1007/11780274_20. Available at `http://dx.doi.org/10.1007/11780274_20`.

[5] Edward Kmett: *Machines*. https://hackage.haskell.org/package/machines. Accessed: 2022-03-27.

[6] Tom Leinster (2003): *Higher Operads, Higher Categories*.

[7] Saunders MacLane (1971): *Categories for the Working Mathematician*. Springer-Verlag, New York. Graduate Texts in Mathematics, Vol. 5.

[8] Alexandre Garcia de Oliveira (2023): *monopro-agda-formal*. Available at `https://github.com/romefeller/monopro-agda-formal`. Accessed: September 13, 2023.

[9] Alexandre Garcia de Oliveira, Mauro Jaskelioff & Ana Cristina Vieira de Melo (2022): *On Structuring Functional Programs with Monoidal Profunctors*. *Electronic Proceedings in Theoretical Computer Science* 360, p. 134–150, doi:10.4204/eptcs.360.7. Available at `http://dx.doi.org/10.4204/EPTCS.360.7`.

[10] Alexandre Garcia de Oliveira: *Programming with monoidal profunctors and semiarrows*. Ph.D. thesis, Universidade de São Paulo. Agência de Bibliotecas e Coleções Digitais, doi:10.11606/t.45.2023.tde-03112023-152323. Available at `http://dx.doi.org/10.11606/T.45.2023.tde-03112023-152323`.

[11] 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.

[12] Exequiel Rivas & Mauro Jaskelioff (2017): *Notions of computation as monoids*. *Journal of Functional Programming* 27, p. e21, doi:10.1017/S0956796817000132.

[13] Alexandra Silva, Filippo Bonchi, Marcello Bonsangue & Jan Rutten (2013): *Generalizing determinization from automata to coalgebras*. *Logical Methods in Computer Science* 9.