Programming with Monoidal Profunctors and Semiarrows

Alexandre Garcia de Oliveira

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Doctor of Science

Program: Computer Science
Advisor: Prof $^{\underline{a}}$ Dr$^{\underline{a}}$ Ana Cristina Vieira de Melo
Co-advisor: Prof Dr Mauro Javier Jaskelioff

São Paulo, September 2023

Programming with Monoidal Profunctors and Semiarrows

This version of the thesis includes the corrections and modifications suggested by the Examining Committee during the defense of the original version of the work, which took place on September 13, 2023.

A copy of the original version is available
at the Institute of Mathematics and
Statistics of the University of São Paulo.

Examining Committee:

- Prof. Dr. Mauro Javier Jaskelioff (co-advisor) - UNR
- Prof. Dr. Alberto Raúl Pardo Garcia - UDELAR
- Prof. Dr. Nicolas Wu - IMPERIAL BS
- Prof. Dr. Tarmo Uustalu - RU
- Prof. Dr. Hugo Luiz Mariano - IME-USP

The happiness of your life depends upon the quality of your thoughts.

Marcus Aurelius

Resumo

OLIVEIRA, A. G. **Programando com Profuntores Monoidais e Semiarrows**. 2023. 106 f. Tese (Doutorado) - Instituto de Matemática e Estatástica, Universidade de São Paulo, São Paulo, 2023.

Este trabalho investiga os profuntores monoidais e suas extensões, como profuntores monoidais com efeitos colaterais e semiarrow, como ferramentas para raciocinar e estruturar programas funcionais puros a partir de uma perspectiva categórica e dentro de uma implementação em Haskell. Abordamos-os como monoides dentro de uma categoria monoidal específica de profuntores e como semiarrows em uma categoria de semiarrow. Examinamos as propriedades dessa categoria monoidal e construímos e implementamos o profunctor monoidal livre. Além disso, detalhamos as propriedades e leis de um semiarrow, derivando exemplos de seu uso e destacando seu potencial para gerenciar efetivamente atrasos em programação síncrona. As máquinas de Moore servem como um exemplo ilustrativo. Aplicações adicionais incluem óptica de profuntores e conexões que preservam a estrutura de um profunctor monoidal entre máquinas de Moore, dobragens com acumulação à esquerda (scan) e dobragens simples à esquerda (fold).

Palavras-chave: Profuntores monoidais, Profunctores monoidais com efeitos colaterais, Semiarrows, Programação funcional, Programação síncrona, Máquinas de Moore.

Abstract

OLIVEIRA, A. G. **Programming with Monoidal Profunctors and Semiarrows**. 2023. 106 p. Thesis (Doctorate) - Instituto de Matemática e Estatástica, Universidade de São Paulo, São Paulo, 2023.

This work investigates monoidal profunctors and their extensions, such as effectful monoidal profunctors and semiarrows, as tools for reasoning and structuring pure functional programs from a categorical perspective and within a Haskell implementation. We approach them as monoids within a specific monoidal category of profunctors and as semiarrows in a semiarrow category. We examine the properties of this monoidal category and construct and implement the free monoidal profunctor. Furthermore, we detail the properties and laws of a semiarrow, deriving examples of its usage and highlighting its potential for effectively managing delays in synchronous programs. Moore machines serve as an illustrative example. Additional applications include optics and a monoidal profunctor structure-preserving connection between Moore machines, left scans, and left folds.

Keywords: Monoidal profunctors, Effectful monoidal profunctors, Semiarrows, Functional programming, Synchronous programming, Moore machines.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A common approach to solving problems in various fields is to decompose them into smaller pieces so that the solution can be obtained by composing the individual solutions. In programming, this approach is no different; one can decompose a problem into smaller programs, execute each one, and then combine the results to resolve the issue. Pure functional programming, in particular, views programs as pure mathematical functions, i.e., functions without computational side-effects. Compositionality can lead to clean, efficient, and easy-to-reason code. Pure functional programming treats programs as pure mathematical functions without computational side-effects, leading to clean, efficient, and easy-to-reason code. Compositionality is a powerful tool for structuring such programs [Obr98].

Functional languages are naturally compositional because functions serve as partial solutions that can be composed to provide a complete solution. They can support this compositional approach in various computer-based systems. Preserving computational behavior under this approach demands a solid mathematical foundation to model the composition of computer programs.

Category Theory is a mathematical field that abstracts various mathematical concepts and their composition. Samuel Eilenberg and Saunders Mac Lane introduced Category Theory in the 1940s. Today, this theory has extensive literature and is widely applied in numerous scientific and engineering disciplines. Structuring pure functional programs using categorical constructs is a well-known topic that has attracted the interest of many researchers in theoretical computer science.

Composing programs with side-effects is a challenging yet well-studied subject in computer science. It is well-known that pure functions cannot produce side-effects. Monads [Mog91] enable composition by distinguishing between values and computations. This distinction streamlines the combination of operations ensuring a consistent handling of side effects. Applicative functors [MP08], although similar to monads, provide compositionality at the cost of being limited to static computations.

In the chain of abstractions of unary type constructors, applicative functors lie between functors (the weakest of the three) and monads (the strongest). On a related note, arrows [Hug05] concentrate on compositional processes that resemble machine-like constructions, specifically for binary type constructors.

Profunctors [PGW17] and their related classes, such as Strong, Grates, Traversal, and Closed [Kmec,PGW17,CEG$^+$20], are well-studied categorical concepts for binary type constructors in functional programming. Their popularity has increased recently due to the development of profunctorial optics.

Another well-known structure in the functional programming community is arrows. Introduced by Hughes [Hug05], arrows were designed to model general processes exhibiting machine-like behaviors. Similar to monads, arrows have their own sugarized syntax, the proc notation [Pat01]. This notation simplifies the usage of arrows and assists imperative programmers in reasoning about processes encoded by arrows. Arrows have been employed in various domains within Haskell, showcasing their versatility and usefulness. Some examples of these domains include parsing, stream processing, functional reactive programming, and graphical user interfaces.

Hughes [Hug05] demonstrated several arrow examples, including Kleisli arrows. Kleisli arrows are functions that output a value wrapped in a monad, which is used to model effects.

Parsers can also have an arrow interface, as pointed out by [Hug05], which refers to the work of [SD96] that uses a non-Monadic parser. In this context, the typeclasses *ArrowZero* and *ArrowPlus* are introduced, which serve as alternative functor analogs for arrows. Arrows with recursion schemes, such as Haskell's *ArrowLoop* typeclass, give rise to a notion of traced monoidal categories, studied by Hasegawa [Has09].

Arrows also model context-dependent programming [UV06], stemming to its coKleisli nature, which involves functions that input values wrapped in a comonad. This paper also demonstrates that biKlesli arrows, which have a comonad structure on function input and monad structure on output together with a distributive law of a monad over a comonad, have an arrow instance. This allows the formalization of partial-stream functions (such as clocked dataflow programs).

Generalized Arrows, an approach for metaprogramming presented in the work of [Meg10], extend the notion of Arrows. A Generalized Arrow is a typeclass without Arrow's *arr* function, and with six more functions based on Monoidal Categories' associative and unity rules. In this work, various forms of arrow-like combinators give rise to Multi-Level terms and Types. Another interesting example from this work is the way Generalized Arrows combinators can produce ways to program in a Simply Typed Kappa Calculus context.

In parallel programming, the work of [BLT18] involves creating a parallel arrow combinator and several typeclasses, such as *PArrow*, which extends the Arrow formalism to parallel computations. This approach generalizes algorithms across three different parallel frameworks. The authors note that Arrow parallelism has lower performance overhead compared to other approaches.

The Yampa package, studied in the work of [LCH09], is employed in various signal processing applications, including animation, robotics, and sound synthesis. In this work, Liu introduces the concept of causal commutative arrows, which are arrows that satisfy specific axioms. The notion of commutative arrows, as observed in [LCH09], captures the properties of concurrent computations. The work also proposes an extension of the simple type lambda calculus and derives a normal form called Causal Commutative Normal Form.

Functors, applicative functors [MP08], monads [Mog91, Wad92, Spi90], profunctors, and arrows [Hug05, JHH09] 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. [LWY11] compare these structures from the point of view of (typed) programming languages, and Rivas and Jaskelioff [RJ17] 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.

Several works that model computer science problems with categorical theoretical constructs, like monoidal profunctors, can be considered related. Table 1.1 compiles the most influential authors and their works related to this research, organized by structure and practical applications.

Table 1.1: Authors compilation

Work	Structure	Applications
[MP08]	Applicative functors	Parsers, Evaluation of expressions
[Obr98]	Monads	Interpreters, Nondeterminism
[Meg10]	(Generalized) Arrows	Muli-level languages
[BLT18]	Arrows	Parallel and concurrent programming
[HM98]	Monads	Recursive descent parsers
[CK14]	Applicative functors	Options parser
[UV06]	Monads, Comonads and Arrows	Dataflow programming
[LCH09]	(Causal Commutative) Arrows	General signal processing algorithms
[Wad95]	Monads	Effectful computations in general
[Spi90]	Monads	List comprehensions, Maybe data type for exceptions, and initial insights into monadic behavior
[PGW17]	Profunctors	Modular data accessors (Optics)
[Hug05]	Arrows	Point-free programming, Interpreters, Parser combinators, CGI programming
[RJ17]	Monoids in monoidal categories	Optimization

It is worth to note that each of the listed work, in Table 1.1, has a significant impact on Haskell's library, Hutton's work [HM98] on monadic parser combinators on Haskell gives the foundations for *parsec* package for example. Many other packages have a plethora of excellent and practical instances for monads, applicative, arrows and profunctors.

1.1 Research Questions

The primary objective of this research is to explore the existence of other categorical constructs that lie between arrows and profunctors and to determine whether these new constructs yield valuable applications that can be utilized by the functional programming community. To accomplish this goal, we extensively study two structures, present their categorical semantics, and provide illustrative examples for better understanding and reasoning.

Question 1 - Can monoidal profunctors be employed to structure and reason about pure functional programs in the same manner as applicative functors? Is it possible to fill the gap in Table 1.2 with monoidal profunctors?

Monoidal profunctors offer an intriguing structure for modeling parallel computations. Their free construction effectively represents process calculi, and they give rise to an optic. By utilizing

functor	applicative	monad
profunctor	????	arrow

Table 1.2: Structure relations

the effectful version, it becomes possible to allow effects when splitting and merging data. We discovered that monoidal profunctors can fit in this table, and their generalization appears promising.

Question 2 - Can the extension of a monoidal profunctor to a semiarrow also be utilized to structure and reason about pure functional programs in a manner similar to arrows?

Through examining the semiarrow interface, we determined that it is suitable for composing Moore machines and offers a method for composing stateful components in a structured manner. The *SemiArrow* interface defines combinators that enable sequential and parallel composition of components while preserving the semantics of state and delay, providing an interesting framework for reasoning about synchronous programs. Additionally, we uncovered intriguing connections between Moore machines and folds.

1.2 Research and Thesis Organization

This work provides another instance of employing monoids in a monoidal category to model computations, following the same approach as in the work of Rivas and Jaskelioff [RJ17]. Furthermore, we present and discuss the semiarrow in a semiarrow category, using a similar line of reasoning. This process involves three primary steps:

- **Categorical view:** We identify an appropriate categorical environment to derive algebraic structures, prove related results and coherence laws, characterize its tensors, and discuss their implications.
- **Algebraic structures:** We extract the targeted algebraic structures, prove results pertaining to them, establish associated laws, and investigate their properties using Haskell.
- **Exploration:** We investigate the derived structure, offering insights and basic examples, we also demonstrate its adherence to the derived laws.
- **Applications and Impact:** We discuss the applications of these structures and ponder their impact on computer science-related problems. Furthermore, we draw comparisons between the utilization of the derived structures and other well-established structures.

As mentioned in the previous section, this research mainly focuses on two such structures: monoidal profunctors and semiarrows.

Monoidal profunctors are a categorical structure with two key components: an identity computation and a generic parallel composition. As a profunctor, they have the ability to lift pure computations into their structure. This structure is derived from a monoidal category of profunctors, where the Day convolution serves as its tensor. It is important to note that the Day convolution can also act as a tensor for profunctors. The free construction on top of a monoidal profunctor is also possible [Mil]. Following the research steps described earlier, we characterize a monoidal profunctor as a monoid in the monoidal category of profunctors. Additionally, we propose an idea to consider the monoidal profunctor to accommodate effects when splitting and merging. This can be achieved by utilizing the Day convolution with other categories, such as the Kleisli category, as a model.

We extend the aforementioned categorical structure with another tensor in the profunctor category, the profunctor composition tensor, giving us a semiarrow category. We do not require that this tensor has an identity. This construction will have a law that correlates both tensors and also possess coherence laws. A semiarrow category possesses an additional associative tensor without an identity law. This tensor provides a notion of sequential composition. The two tensors are related via the interchange law, which enables the commuting of effects.

The semiarrow in a semiarrow category pattern emerges by following the same specialization to a single object approach. Semiarrows extend monoidal profunctors by adding an associative computation without an identity, and the interchange law links them. In Haskell, the semiarrow exhibits a comparatively weaker structure than arrows, as it does not include functions such as *arr*, *first*, and *second*. This weaker structure is due to the absence of an identity element of the sequential composition.

Monoidal profunctors are present in the Haskell ecosystem within the packages $product-profunctors$ [Ellb] and `opaleye` [Ella], as well as in the profunctor optics literature [PGW17], and other community-wide texts about free monoidal profunctors [Mil]. Possible applications for monoidal profunctors alone are in parallel programming, as a tool for reasoning about contexts, and even optics [CEG$^+$20].

A semiarrow is suitable for composing Moore machines and provides a way to compose stateful components in a structured manner, which is essential in applications such as synchronous data-flow programming. This interface defines combinators that preserves the semantics of state and delay.

Data-flow frameworks built on Moore machines exemplify this programming style [HCRP91]. They facilitate the composition of stateful components, managing delays and states in a structured way while maintaining predictable and deterministic behavior. Composable Moore machines provide suitable semantics for this programming paradigm, which can be challenging to reason about using causal commutative arrows [LCH11].

This study further establishes a connection between Moore machines and folds, demonstrating that known laws can be derived from this categorical relationship. Moore machines and folds are connected by a natural transformation that preserves the monoidal profunctor operations.

This work is organized as follows. In Chapter 2.1, we focus on providing the background knowledge necessary to understand the ideas contained in this work. This chapter covers three major topics. The first contains the basics of Haskell, which serves as the foundation for understanding the programming concepts used throughout this work. The section aims to familiarize the reader with essential Haskell concepts, such as types, typeclasses, and other language constructs, which are used in the development and illustration of monoidal profunctors and semiarrows.

In Section 2.2 of the same chapter, we provide definitions and some proofs related to category theory. We also illustrate these concepts with Haskell examples to facilitate understanding and show the connection between the theory and practical applications.

In Section 2.3 of the same chapter, we present the theory of monoidal categories, and more advanced concepts such as profunctors and coend calculus. These topics form the basis for understanding the subsequent chapters, which goes deeper into the relationship between monoidal profunctors, semiarrows, and their applications in both theoretical and practical contexts.

Chapter 3 presents the notion of a monoidal category and its laws, describes profunctors, and defines the Day convolution. it also introduces monoidal profunctors along with the concept of a monoid built on top of them. This chapter also shows some basic instances of the monoidal profunctor typeclass, as well explores the idea of a free monoidal profunctor and presents the representation theorem for profunctors needed for optics. We also present the concept of an effectful monoidal profunctor and provide examples of its use with Kleisli arrows to construct a quicksort algorithm that collects effects when splitting and merging. We introduce the categorical tools needed to construct a semiarrow in a semiarrow category and discuss its laws in Chapter 4. In addition, it presents how to handle delays using the semiarrow interface and Moore machines, providing insights for building a robust framework for synchronous data-flow programming.

In Chapter 5, we present a technique present in the opaleye package to create type-safe lists, the discussion about the monoidal profunctor semantics of an optic called *Monocle* that handles every coordinate of a tuple simultaneously, an application of the free monoidal profunctor, and also explore the connections between Moore machines and folds. Chapter 6 presents the conclusion of this research.

1.3 Claimed Contributions

The novel contributions of this reasearch are the following:

- **Gathering information about monoidal profunctors:** Monoidal profunctors are relatively unexplored in functional programming literature. This research surveys the topic, presenting definitions, results, and insights from specialized literature and from Haskell's folklore.

- **Semantics for the Monocle optic**. The optic, referred to as Monocle in this research, has been previously defined in the literature and is also widely recognized in the Haskell folklore [PGW17,O'Ca]. While its definition here has slight variations, the essence remains the same. In this research, we provide the semantics for a monoidal profunctor using the binary representation theorem and a coend over monoidal profunctors.

- **Examples for the Free Monoidal Profunctor**. This is a minor contribution, in which we constructed an Abstract Syntax Tree (AST) that serves as an insightful example for a process calculus language.

- **Effectful Monoidal Profunctor**. We introduce a way to extend a monoidal profunctor to allow effects when splitting and merging data. For example, we can use Kleisli arrows to generate any encoded effect within its monad when splitting/merging data. In this work, we presented a quicksort algorithm that utilizes logging effects when splitting and merging lists.

- **Semiarrows**. This structure was inspired by a commutative causal arrow, but it is significantly different. The semiarrow emerged when the author was attempting to compose Moore machines for a different purpose and wondered why this type wasn't considered an arrow. In this work, we provide examples, categorical semantics, and laws for semiarrows.

- **Semantics for Delays**. The discussion of extending a semiarrow to a *GSemiArrow* also appears to be new. We propose a simple graphical language for studying delays and provide examples. The idea of further extending this concept to exhibit operadic behavior is also present in this research.

- **Connections between left Fold, left Scan, and Moore machines**. We also discover an unexpected connection between Folds, Scans, and Moore machines, and found that they are interconnected by natural transformations. A Moore machine preserves the monoidal profunctor structure, but not the semiarrow one, due to delays.

1.4 Presentations and Publications

This research was presented at four functional programming and category theory events, including IFL2020, MGS2021, MSFP2022, and II Encontro Brasileiro em Teoria das Categorias.

- **Paper 1.** Alexandre Garcia de Oliveira, Mauro Javier Jaskelioff, and Ana Cristina Vieira de Melo (2022). "On Structuring Pure Functional Programs Using Monoidal Profunctors". In: Proceedings of the Workshop on Mathematically Structured Functional Programming. Munich, Germany. This work presents the findings related to monoidal profunctors, effectful monoidal profunctors, and their applications, such as free monoidal profunctors, the monocle optic, and its semantics based on them. The first half of this work was summarized in a paper, while the content about semiarrows was not included. The paper can be downloaded at https://arxiv.org/abs/2207.00852.

- **Paper 2.** Alexandre Garcia de Oliveira, Mauro Javier Jaskelioff, and Ana Cristina Vieira de Melo (2023). "Programming with Monoidal Profunctors and Semiarrows". In: Science of Computer Programming. Submitted for publication. This work is an extension of the first paper that includes the whole semiarrow background, its definitions, laws, examples, and applications.

The author also co-organized the II Encontro Brasileiro em Teoria das Categorias event. The second paper was presented at the II Encontro Brasileiro em Teoria das Categorias under the title "Programming with Semiarrows" and an abstract will be published in its proceedings.

Prerequisites

This thesis focuses on functional programming and how to reason about its programs using category theory. This work is as self-contained as possible, but for a better reading experience, the reader should have some basic knowledge of functional programming and computer science basic concepts.

In Chapter 2.1, we provide the Haskell basics needed to follow every program listed throughout the text. We do not expect the reader to be an expert in Haskell, but some understanding of functional programming is recommended.

This work is also heavily grounded in mathematics. Chapters 2.2 and 2.3 define and discuss the tools needed from category theory. The proof techniques used in this research include diagram chasing proofs, equational reasoning, induction, and structural induction. A reader with a background in discrete mathematics and some knowledge of basic proof techniques will have a better experience reading this text. We also expect the reader to have little to no previous background in category theory.

Chapter 2

Background

2.1 Haskell Basics

This chapter introduces the essential features of the Haskell programming language required to understand the rest of this work. Installation, compiler handling, and execution details are not discussed.

For a more detailed exploration of these concepts, consult the work of Lipovaca [Lip11].

2.1.1 Syntax

Haskell functions have the following form:

$$\begin{aligned}&f :: t1 \to t2 \to ... \to tn \to tr\\&f\ v1\ v2\ ...\ vn = expression\ (v1, v2, ..., vn)\end{aligned}$$

One can read this as a function f with input names $v1$ of type $t1$ (written as $v1 :: t1$), $v2$ of type $t2$, and so on, returning an expression of type tr which can depend on any of those bindings. The type signature of the function can be omitted, although this is not considered good programming practice, as it leaves the expression type to be inferred by the compiler.

Example 1 - A function $f :: tr$ with no inputs is called a constant function.

Example 2 - Let $vi :: ti$ be values of type ti for all $i \geqslant 1$, and $f :: t1 \to ... \to tr$ be a function. The expression $f\ v1\ v2\ ...\ vn$, which represents the application of f, has type tr. If one applies f with fewer arguments, such as $f\ v1$, $f\ v1\ v2$, $f\ v1\ v2$, ..., $f\ v1\ v2\ ...\ vn1$, the resulting types will be $t2 \to t3 \to ...tn \to tr$, $t3 \to ... \to tn \to tr$, ..., $tn \to tr$, respectively. A function that can be applied in this manner is referred to as a curried function.

Functions can also have a scoped list of declarations using the **let** clause:

$$\begin{aligned}&f :: t \to r\\&f\ t = \textbf{let}\ y = subexp\ (t)\ \textbf{in}\ expression\ (y, t)\end{aligned}$$

In this example, you can use sub-expressions and bind them to a name, in this case y, to use it later in the return expression.

A function can also be represented as a lambda (as in lambda calculus) [BL80] in the form $\lambda v1\ v2\ ...\ vn \to expression$, which serves as a function literal.

2.1.2 Algebraic Data Types

Algebraic Data Types (ADTs) can be used to create custom types that can be variants of products (tuples), sums (disjoint unions), or a combination of both. The keywords **newtype** and **data** denote single constructor types with strict semantics, while **data** can have multiple constructors with lazy semantics.

Example 3 - The type *Day* is a sum type declared as:

$$\textbf{data}\ \mathit{Day} = \mathit{Sun} \mid \mathit{Mon} \mid \mathit{Tue} \mid \mathit{Wed} \mid \mathit{Thu} \mid \mathit{Fri} \mid \mathit{Sat}$$

It has seven constructors, each with no records.

Example 4 - A type for currency is a product type declared as:

$$\textbf{data}\ \mathit{Currency} = \mathit{Currency}\ \mathit{String}\ \mathit{Double}$$

It has one constructor (with the same name as the type, which is allowed by the compiler) and two fields. The same type can be written in another form:

```
data Currency = Currency {
  name :: String,
  value :: Double
}
```

In this case, *name* and *value* are projection functions of type $\mathit{Currency} \to \mathit{String}$ and $\mathit{Currency} \to \mathit{Double}$ respectively.

Example 5 - An integer list is a recursive type and also can be written as an ADT.

$$\textbf{data}\ \mathit{ListInt} = \mathit{Null} \mid \mathit{Cons}\ \mathit{Int}\ \mathit{ListInt}$$

A *ListInt* has two constructors, *Null* has no fields and *Cons* has an integer carrier field and a recursive field to keep the structure going. For example the value $\mathit{Cons}\ 7\ (\mathit{Cons}\ 5\ (\mathit{Cons}\ 3\ \mathit{Null}))$ has type *ListInt*.

Example 6 - The type declaration **newtype** $\mathit{Foo} = \mathit{Foo}\ \mathit{Int}$ has only one constructor and by strictness is treated by the compiler as an integer.

One useful feature of an ADT is that it can be destructed at any function input, this concept called *pattern matching*.

```
increaseTen :: Currency → Currency
increaseTen (Currency n v) = Currency n (v + 10)
```

The function *increaseTen* has a destructed *Currency* by *pattern matching* exposing the *name* and *value* fields binding it to the names *n* and *v*.

2.1.3 Parametric Polymorphism

The concept of *parametric polymorphism* allows functions to have a generic behavior and does not depend on a specific type, implying that such a function cannot access any value of its inputs. A function with concrete types is called monomorphic. For example, a function that concatenates two lists is polymorphic because there is no need to inspect the list values. However, a function that negates a boolean value is *monomorphic*.

Types can also be polymorphic by having one (or more) type parameter(s) in its declaration.

$$\textbf{data}\ \mathit{List}\ a = \mathit{Null} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

The type *List* is a polymorphic type with a **type** *parameter a*, while the type *List Int* is monomorphic and equivalent to $[\mathit{Int}]$.

A function can also be polymorphic, this allow its input/output for every type avoiding having a boilerplate code with the same function with the same behavior being coded for every type.

```
id :: a → a
id x = x
```

The *identity function id* serves as an excellent example of parametric polymorphism. It accepts an input of any type a and returns the same value without alteration, irrespective of the input type. This eliminates the necessity for separate functions like $\mathit{idString} :: \mathit{String} \to \mathit{String}$, $\mathit{idInt} :: \mathit{Int} \to \mathit{Int}$, and others that would only serve to fill the codebase with redundant implementations.

2.1.4 High-order Functions

One key concept present in functional languages like Haskell is the concept of a high-order function. High-order functions are functions that can accept other functions as input arguments or return functions as output. This concept helps promote compositionality in functional programming.

High-order functions enable the creation of new functions based on existing ones, allowing for greater code reusability and modularity.

In Haskell, high-order functions are ubiquitous and are widely used to express a variety of common patterns, on lists, for example, mapping, filtering, and folding are daily ingredients of a function programmer. For example, the *map* function applies a given function to each element of a list, the *filter* function selects elements from a list based on a predicate function, and the *foldl* function combines elements of a list using a binary function.

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

filter :: (a → Bool) → [a] → [a]
filter _ [ ] = [ ]
filter f (a : as)
    | f a = a : filter f as
    | otherwise = filter f as
```

The functions above are high-order functions as one can observe its inputs. The function *foldl* will be defined and discussed further later on this present work.

Other high-order functions exist to handle common patterns in functions, such as *flip* that swaps arguments in a function with two inputs, ($) that is the operator to apply a function to a value reducing the usage of parentheses, and (∘) that composes two functions to create another one having the output type of the first function and the input type of the second function.

```
flip :: (a → b → c) → (b → a → c)
flip f b a = f a b

($) :: (a → b) → a → b
($) f a = f a

(∘) :: (b → c) → (a → b) → a → c
(∘) f g a = f (g a)
```

2.1.5 Typeclasses

A typeclass is an implementation of ad-hoc polymorphism in Haskell and provides bounds to polymorphic types. Wadler and Blott [WB89] defines ad-hoc polymorphism as functions that behave differently depending on a particular type. For example, the multiplication behavior when dealing with integers and another for floating-point numbers.

A typeclass is a type constraint that contains definitions to be specialized for each concrete type.

Example 7 - The class *Show* deals with values that can be converted to a string and is defined as

```
class Show a where
  show :: a → String
```

and can have, as a concrete instantiation, the following instance

```
instance Show Day where
  show Sun = "Sunday"
  show Mon = "Monday"
  show Tue = "Tuesday"
  ...
  show Sat = "Saturday"
```

and specifies how to convert a *Day* into a string. The function show has the type $\mathit{show} :: \mathit{Show}\ a \Rightarrow a \rightarrow \mathit{String}$ indicating that the class *Show* is indeed restricting the type parameter a. Classes also have a kind, in this case, $* \rightarrow \mathit{Constraint}$.

Example 8 - Let us recall the definition of a monoid used in algebra.

Definition 2.1 - A monoid is a triple $(M, \otimes, u)$ where M is a set, $\otimes$ is a binary operation and u is an element of M such that satisfies the axioms:

- (Neutral element) $\exists u \in M$ such that $\forall a \in M\ \ a \otimes u = a$ and $u \otimes a = a$;
- (Associativity) $\forall a, b, c \in M\ \ (a \otimes (b \otimes c)) = ((a \otimes b) \otimes c)$.

In Haskell, the class *Monoid* represents a monoid and is given by the class

```
class Monoid m where
  mempty :: m
  (⊗) :: m → m → m
```

where *mempty* plays the role of the neutral element and (⊗) of the binary operation. The compiler will not enforce the axioms; this is the responsibility of the programmer. A natural instance of the monoid type class are Strings where its neutral is the empty list, and the binary operation is the concatenation of lists.

```
instance Monoid String where
  mempty = [ ]
  (⊗) = (++)
```

2.2 Category Theory

In this chapter categories and their related concepts are presented. The basics of category theory such as categories, functors, natural transformations, monads and other relevant concepts will be presented. These concepts are the main tools for studying other categorical concepts such as adjoints, algebras, co-algebras, bifunctors, profunctors to name a few. The definitions are based on the work of Saunders [Mac71] and Awodey [Awo10]. Every category-theoretic definition will be accompanied with a Haskell piece of code to show how category theory is used to reason about pure functional programs.

2.2.1 Categories

Definition 2.2 - A category [Mac71] $\mathcal{C}$ consists of:

- A collection of objects denoted by $ob(\mathcal{C})$;
- For all objects A and B, a set of morphisms denoted by $Hom_{\mathcal{C}}(A, B)$ or $\mathcal{C}(A, B)$ of inputs A and outputs B. We also use $Hom(A, B)$ when the category context is clear. A morphism can be also called an arrow;
- Notion of composition for morphisms represented by the binary operator $\circ : Hom(B, C) \times Hom(A, B) \to Hom(A, C)$. If $g : B \to C$ and $f : A \to B$ then $g \circ f : A \to C$,

satisfying the following axioms:

- (Existence of Identities) For all objects X of $ob(\mathcal{C})$ there exists a identity morphism $id_X : X \to X$;
- (Neutral) The identity morphism acts as a ”neutral element”. If $f : A \to B$ then uniqueness of identity follows $f \circ id_A = f = id_B \circ f$
- (Associativity) The composition above is associative, if $h : A \to B$, $g : B \to C$ and $f : C \to D$, then $(f \circ g) \circ h = f \circ (g \circ h)$ for every object A, B, C and D of $ob(\mathcal{C})$.

Categories can be viewed also as a directed graph where objects are nodes and the edges represents a morphism connecting such nodes. It is valid to observe that categories has a similar structure as monoids. If $ob(\mathcal{C})$ and $Hom(A, B)$ are sets then the category is labelled as a small category, if only $Hom(A, B)$ are sets, it will be called hom-set, then the category is said to be locally small. To illustrate the definition above, some examples of categories in mathematics and also in computer science are given as follows.

Example 9 - The category **Set** of Sets, consists of sets being its objects and set functions as its morphisms [Mac71].

Example 10 - The category $\mathbf{Vect_k}$ of Vector Spaces, consists of vector spaces with a base field k being its objects and linear transformations as its morphisms [Awo10].

Example 11 - The category **Lambda** representing the lambda calculus, it has types as its objects and functions as morphisms [Awo10].

Example 12 - The category **Mon** of Monoids, consists of monoids being its objects and monoid homomorphism as its morphisms [Mac71].

Example 13 - The category **Cat** of Categories, consists of small categories being its objects and Functors, see the next section, as its morphisms [Mac71].

Example 14 - The category $\mathcal{C}^{op}$ of some category $\mathcal{C}$ has the same objects of $\mathcal{C}$ but reversed morphisms, i.e, if $f : A \to B$ in $\mathcal{C}$ then $f : B \to A$ in $\mathcal{C}^{op}$ [Mac71].

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, dotted, "F"'] & B \arrow[d, dotted, "F"] \\
F(A) \arrow[r, "F(f)"] & F(B)
\end{tikzcd}$$

Figure 2.1: Diagram for a covariant functor

In Haskell, there is a type class called Category (package Control.Category) which models the definition given above.

```
class Category cat where
  id :: cat a a
  (∘) :: cat b c → cat a b → cat a c
```

The type (→) is a natural instance for the Category type class. This class restricts the objects of the category *cat* to be types.

```
instance Category (→) where
  id = GHC.Base.id
  (∘) = (GHC.Base.∘)
```

The base functions *id* and (.) are the obvious implementation's of the methods from the type class Category. It is always good to remember, the compiler will not check the category axioms.

2.2.2 Functors

A covariant functor is a structure preserving map between categories in a way that objects go to objects and morphism go to morphisms. This kind of functor is also called covariant [Mac71].

Definition 2.3 - Let $\mathcal{C}$ and $\mathcal{D}$ be categories, $F : \mathcal{C} \to \mathcal{D}$ is said to be a **covariant** functor if it sends an object A, in $ob(\mathcal{C})$, to $F(A)$ in $ob(\mathcal{D})$ and sends a morphism $f : A \to B$ to $F(f) : F(A) \to F(B)$, satisfying:

- $F(id_X) = id_{F(X)}$ for all objects X of $\mathcal{C}$;
- $F(f \circ g) = F(f) \circ F(g)$ for all composable morphisms f and g of $\mathcal{C}$;

Example 15 - The functor $U : Mon \to Set$ is called the forgetful functor, this functor forgets the additional structure of monoids and downcast them to a simple set [Mac71].

Example 16 - The functor $(-)^* : Set \to Mon$ is called the free functor, this functor creates an artificial monoid structure for a particular unstructured set. This functor is useful to model lists in a programming language context [Awo10].

Example 17 - The functor $\Delta_X : \mathcal{C} \to \mathcal{D}$ is called the constant functor, this functor sends all objects of $ob(\mathcal{C})$ to an fixed object X of $ob(\mathcal{D})$ and every morphism of $\mathcal{C}$ to id_X in $\mathcal{D}$ [Mac71].

Example 18 - The functor $Hom(X, -) : \mathcal{C} \to Set$ where $\mathcal{C}$ is a locally small category, is called the hom-functor. This functor sends an object A of $ob(\mathcal{C})$ to the hom-set $Hom(X, A)$ of all morphisms from X to A and sends a morphism $f : A \to B$ to $Hom(X, f) : Hom(X, A) \to Hom(X, B)$, where $g \mapsto f \circ g$. A functor with the same input and output category is called an endofunctor.

The type class *Functor* (kind $(* \to *) \to Constraint$) is inspired on the definition given above in Haskell. This class has only one method to be implemented in its instances called *fmap*.

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

The function *fmap* takes a high-order function, call it g for simplicity, of type $a \to b$ and a value of type $f\ a$ and transforms into a value of type $f\ b$, this process can be thought as $f\ a$ being some sort of container of a's, then g will transform the a's (inside of f) into b's, leaving a container of b's represented by the type $f\ b$. The Examples 16 and 18 can be represented, in Haskell, by the list $[\,]$ and $((\to\ x)$ instances of the *Functor* class, where *fmap* is just the *map* function for the former and the composition $(\circ)$ of functions for the latter.

Let $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ two functors, it is possible to compose them $G \circ F : \mathcal{C} \to \mathcal{E}$, for any object X from $\mathcal{C}$, $G \circ F(X)$ is an object of $\mathcal{E}$, note that this order of the composition matter like a normal function, and any morphism $f : A \to B$ of $\mathcal{C}$, thus

$$(G \circ F)(f) :\ (G \circ F)(A) \to (G \circ F)(B) \tag{2.1}$$

is a morphism in the category $\mathcal{E}$, this composition is clearly associative and $G \circ F$ give us the composite functor of G with F. This construction is useful for the category *Cat* of small categories.

There is, in Haskell, a datatype that represents a functor composition located in the *Data.Functor* package called *Compose* and has the kind $(* \to *) \to (* \to *) \to * \to *$.

```
newtype Compose f g a = Compose {getCompose :: f (g a)}
instance (Functor f, Functor g) ⇒ Functor (Compose f g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)
```

The datatype mentioned is polymorphic in the variables f, g and a, to make this an instance of the Functor type class f and g must be also instances of *Functor* as noted after the **instance** keyword. The functor composition plays a significant role in the construction of monads.

Our comprehension of functors can be extended by generalizing the concept to operate on the product category, resulting in what is known as a bifunctor [Mac71]. Bifunctors adhere to the customary functor laws, but in the context of the product category.

Definition 2.4 - Let $\mathcal{C}$, $\mathcal{D}$, and $\mathcal{E}$ be categories. A bifunctor $F : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$ is defined as a functor that maps each pair of objects (A, B) of the product category $\mathcal{C} \times \mathcal{D}$ to an object $F(A, B)$ of the category $\mathcal{E}$. Furthermore, it maps each pair of morphisms $(f, g) : (A, B) \to (C, D)$ in $\mathcal{C} \times \mathcal{D}$ to a morphism $F(f, g) : F(A, B) \to F(C, D)$ in $\mathcal{E}$, satisfying:

1. $F(\mathrm{id}_A, \mathrm{id}_B) = \mathrm{id}_{F(A,B)}$ for all objects A of $\mathcal{C}$ and B of $\mathcal{D}$.
2. $F(f_2, g_2) \circ F(f_1, g_1) = F(f_2 \circ f_1, g_2 \circ g_1)$ for all morphisms $f_1 : A_1 \to A_2$, $f_2 : A_2 \to A_3$ and $g_1 : B_1 \to B_2$, $g_2 : B_2 \to B_3$.

In Haskell, a bifunctor, that has the kind $(* \to * \to *) \to Constraint$, is represented by the typeclass below.

```
class Bifunctor p where
  bimap :: (a → b) → (c → d) → p a c → p b d
```

The *bimap* function in Haskell originates from the Bifunctor typeclass and is defined to handle data structures parameterized by two types. A common instance of a bifunctor in Haskell is the tuple type.

```
instance Bifunctor (,) where
  bimap f g (x, y) = (f x, g y)
```

The *Bifunctor* instance for the tuple type applies the first function to the first coordinate and the second function to the second coordinate.

We can also explore functors that operate in the dual category, which are referred to as contravariant functors.

Definition 2.5 - Let $\mathcal{C}$ and $\mathcal{D}$ be categories, $F : \mathcal{C} \to \mathcal{D}$ is said to be a **contravariant** functor if it sends an object A, in $ob(\mathcal{C})$, to $F(A)$ in $ob(\mathcal{D})$ and sends a morphism $f : A \to B$ to $F(f) : F(B) \to F(A)$, satisfying:

- $F(id_X) = id_{F(X)}$ for all objects X of $\mathcal{C}$;
- $F(f \circ g) = F(g) \circ F(f)$ for all composable morphisms f and g of $\mathcal{C}$;

$$\begin{tikzcd}
A \arrow[r] \arrow[d, dotted, "F"'] & B \arrow[d, dotted, "F"] \\
F(A) & F(B) \arrow[l]
\end{tikzcd}$$

Figure 2.2: Diagram for a contravariant functor

Example 19 - The functor $Hom(-, X) : \mathcal{C} \to Set$ where $\mathcal{C}$ is a locally small category, is called the contravariant hom-functor. This functor sends an object A of $ob(\mathcal{C})$ to the hom-set $Hom(A, X)$ of all morphisms from A to X and sends a morphism $f : A \to B$ to $Hom(f, X) : Hom(B, X) \to Hom(A, X)$, where $g \mapsto g \circ f$.

A contravariant functor, having the kind $(* \to *) \to Constraint$, can be represented as the type class *Contravariant* (it is not in base), and the only method of this class is *contramap*.

```
class Contravariant f where
  contramap :: (a → b) → f b → f a
```

The behavior of *contramap* is the same of *fmap* but the arrows of the output are reversed. The contravariant hom-functor, see example 19, can be an instance of *Contravariant*.

```
data Op x a = Op (a → x)
instance Contravariant (Op x) where
  fmap f (ContraHom g) = ContraHom (g ∘ f)
```

Whilst *Op* has kind $(* \to * \to *)$, $(Op\ x)$ has kind $(* \to *)$ and can be made into a instance of the class above.

2.2.3 Natural Transformation

A natural transformation can be viewed as a structure-preserving map between functors, i.e, we can transform a functor into another one. This concept plays a fundamental role in category theory and is the key concept to study monads and adjoints in the later sections of this chapter. A natural transformation is a collection of morphisms between functors and will be denoted ad $Nat(F, G)$ for any two functors $F, G : \mathcal{C} \to \mathcal{D}$.

Definition 2.6 - Let $F, G : \mathcal{C} \to \mathcal{D}$ be Functors, a natural transformation [Mac71] $\eta : F \Rightarrow G$ is a family of morphisms between those functors, with components $\eta_X : F(X) \to G(X)$ for every X in $ob(\mathcal{C})$. This definition must satisfy the following nautrality condition that is: for all $f : A \to B$

$$G(f) \circ \eta_A = \eta_B \circ F(f). \tag{2.2}$$

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "F"] \arrow[r, bend right=50, "G"'] \arrow[r, phantom, "\Downarrow \eta"] & \mathcal{D}
\end{tikzcd}$$

Figure 2.3: A natural transformation diagram

The Figure 2.3 represents a natural transformation in a diagram form which is very important to study mathematical concepts like composition of natural transformations.

$$\begin{tikzcd}
F(A) \arrow[r, "F(f)"] \arrow[d, "\eta_A"'] & F(B) \arrow[d, "\eta_B"] \\
G(A) \arrow[r, "G(f)"] & G(B)
\end{tikzcd}$$

Figure 2.4: Commutative diagram for the naturality condition (2.2)

The condition in equation 2.2 can be represented by the commutative diagram contained in the Figure 2.4, is possible to observe the functorial action of F and G on the morphism f and the action of the components η_A and η_B of the natural transformation η.

Example 20 - In the category $Vect_k$, the linear transformation $V \to V^{**}$, where V^{**} is the double dual of V is natural.

Naturality can be represented by parametricity in a functional programming environment, in Haskell, a natural transformation can be represented as a type.

$$\textbf{type}\ (\leadsto)\ f\ g = \forall x.f\ x \to g\ x$$

The polymorphic $(\leadsto)$ type has kind $(* \to *) \to (* \to *) \to *$, the forall clause hides the inner type of the containers f and g simulating the component behavior of a natural transformation. This datatype only compiles with the *Rank2Types* and *TypeOperators* pragmas enabled.

Composition

Functors can compose in a straightforward manner, as observed in equation 2.1. Natural transformations can also be composed horizontally, vertically, and in a mixed form with functors. Specifically, composing functors followed by natural transformations is called left whiskering, while composing natural transformations followed by functors is known as right whiskering.

Definition 2.7 - (Vertical composition [Mac71]) Let $F, G, H : \mathcal{C} \to \mathcal{D}$ be functors and $\zeta : F \Rightarrow G$, $\xi : G \Rightarrow H$ be natural transformations, the vertical composition is $\xi \bullet \zeta : F \Rightarrow H$ with components $(\xi \bullet \zeta)_A : F(A) \to H(A)$ having $(\xi \bullet \zeta)_A = \eta_A \circ \zeta_A$.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=60, "F"] \arrow[r, "G" description] \arrow[r, bend right=60, "H"'] \arrow[r, phantom, shift left=3ex, "\Downarrow \xi"] \arrow[r, phantom, shift right=3ex, "\Downarrow \zeta"] & \mathcal{D}
\end{tikzcd}$$

Figure 2.5: Vertical composition diagram

Definition 2.8 - (Left whiskering [Mac71]) Let $F : \mathcal{C} \to \mathcal{D}$ be a functor and $G, H : \mathcal{D} \to \mathcal{E}$ also functors and $\eta : G \Rightarrow H$ be a natural transformation, the left whiskering composition is defined as $\eta F : GF \Rightarrow HF$, with components $\eta F_A : GF(A) \to HF(A)$, with $\eta F_A = \eta_{FA}$.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "F"] & \mathcal{D} \arrow[r, bend left=60, "G"] \arrow[r, bend right=60, "H"'] \arrow[r, phantom, "\Downarrow \eta"] & \mathcal{E}
\end{tikzcd}$$

Figure 2.6: Left whiskering

The naturality conditions for η gives that the same conditions holds for ηF, hence ηF is a natural transformation as one can observe in the follow commutative diagram

$$\begin{tikzcd}
GF(A) \arrow[r, "\eta F_A"] \arrow[d, "GF(f)"'] & GF(B) \arrow[d, "\eta F_B"] \\
HF(A) \arrow[r, "HF(f)"] & HF(B)
\end{tikzcd}$$

Figure 2.7: Commutative diagram for the naturality condition for ηF

Definition 2.9 - (Right whiskering [Mac71]) Let $K : \mathcal{D} \to \mathcal{E}$ be a functor and $G, H : \mathcal{C} \to \mathcal{D}$ also functors and $\eta : G \Rightarrow H$ be a natural transformation, the right whiskering composition is defined as $\eta K : KG \Rightarrow KH$, with components $K\eta_A : KG(A) \to KH(A)$, with $K\eta_A = K(\eta_A)$.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=60, "F"] \arrow[r, bend right=60, "G"'] \arrow[r, phantom, "\Downarrow \eta"] & \mathcal{D} \arrow[r, "K"] & \mathcal{E}
\end{tikzcd}$$

Figure 2.8: Right whiskering

The naturality conditions for η gives us also that the same conditions hold for $K\eta$, hence $K\eta$ is a natural transformation as is possible to observe in the following commutative diagram

Definition 2.10 - (Horizontal composition [Mac71]) Let $F, G : \mathcal{C} \to \mathcal{D}$ and $H, K : \mathcal{D} \to \mathcal{E}$ be functors, let $\zeta : F \Rightarrow G$, $\eta : H \Rightarrow K$ be two natural transformations, then the horizontal

$$\begin{tikzcd}
KG(A) \arrow[r, "K\eta_A"] \arrow[d, "KH(f)"'] & KG(B) \arrow[d, "K\eta_B"] \\
KH(A) \arrow[r, "KH(f)"] & KH(B)
\end{tikzcd}$$

Figure 2.9: Commutative diagram for the naturality condition for $K\eta$

composition ζ followed by η is

$$\eta \circ \zeta : H \circ F \Rightarrow K \circ G, \tag{2.3}$$

and defined by

$$\eta \circ \zeta = (\eta G) \bullet (H\zeta)$$

or by,

$$\eta \circ \zeta = (K\zeta) \bullet (\eta F)$$

with the corresponding diagram contained in Figure 2.10.

$$\begin{tikzcd}
\mathcal{C} \arrow[r, bend left=50, "F"] \arrow[r, bend right=50, "G"'] \arrow[r, phantom, "\Downarrow \zeta"] & \mathcal{D} \arrow[r, bend left=50, "H"] \arrow[r, bend right=50, "K"'] \arrow[r, phantom, "\Downarrow \eta"] & \mathcal{E}
\end{tikzcd}$$

Figure 2.10: Horizontal composition

Having the notion of whiskering defined, we define the notion of an adjunction [Mac71] between two covariant functors. This concept will be used later when we discuss the unary representation theorem.

Definition 2.11 - Given two categories $\mathcal{C}$ and $\mathcal{D}$, functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$, F is said to be left adjoint to G and G right adjoint to F, denoted $F \dashv G$, if there are natural transformations:

$$\eta : \mathrm{id}_{\mathcal{C}} \to G \circ F$$

(called the unit) and

$$\varepsilon : F \circ G \to \mathrm{id}_{\mathcal{D}}$$

(called the counit), such that the following diagrams commute:

$$\begin{tikzcd}
& F \circ G \circ F \arrow[dr, "\varepsilon F"] & \\
F \arrow[ur, "F\eta"] \arrow[rr, "\mathrm{id}_F"'] & & F
\end{tikzcd} \tag{2.4}$$

$$\begin{tikzcd}
& G \circ F \circ G \arrow[dr, "G\varepsilon"] & \\
G \arrow[ur, "\eta G"] \arrow[rr, "\mathrm{id}_G"'] & & G
\end{tikzcd} \tag{2.5}$$

The above diagrams express the unit and counit laws, ensuring that the adjunction is coherent.

2.2.4 Monads

A monad is another categorical concept that is very important to this work, it is a functor and two natural transformations satisfying axioms based on naturality conditions represented by equation 2.2. Monads play a significant role in computer science by modeling a notion of computation [Mog91]. It provides a way to have side effects in pure functional programming. A monad can be described in a monoidal category context as is possible to see in later parts of this work.

Definition 2.12 - A monad [Mac71] on a category $\mathcal{C}$ is a triple (T, η, μ), such that T is an endofunctor, $\eta : I \to T$ and $\mu : T^2 \to T$ are both natural transformations (where $T^2 = T \circ T$ and I is the identity functor which fixes objects and morphism of a given category), satisfying

$$\mu \circ \eta T = id_M = \mu \circ T\eta \tag{2.6}$$

$$\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[d, "T\eta"'] \arrow[dr, equal] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}$$

Figure 2.11: Commutative diagram for the naturality condition η operator

$$\mu \circ \mu = \mu \circ T\mu \tag{2.7}$$

$$\begin{tikzcd}
T^3 \arrow[r, "\mu"] \arrow[d, "T\mu"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}$$

Figure 2.12: Commutative diagram for the naturality condition μ operator

In Haskell, monads can be conceptualized as a type class with the kind signature $(* \to *) \to$ *Constraint*.

```
newtype Identity a = Identity a deriving Functor
class Functor m ⇒ Monad m where
  η :: Identity a → m a
  μ :: (Compose m m) a → m a
```

The *Identity* datatype is an instance of *Functor* because of the **deriving** *Functor* which can be enabled by using the language pragma *DeriveFunctor*. In this setting, the variable m, which is a *Functor*, plays the role of the functor T and *Compose* m m is $T \circ T = T^2$. This presentation of a monad can be reduced.

```
class Functor m ⇒ Monad m where
  return :: a → m a
  join :: m (m a) → m a
```

It is clear that $\eta = return$, $\mu = join$, $Identity\ a = a$ and $(Compose\ m\ m)\ a = m\ (m\ a)$, but the Haskell community does not use this setup for a monad, they use another one based on Kleisli arrows [Hug00].

```
class Applicative m ⇒ Monad m where
  return :: a → m a
  (≫=) :: m a → (a → m b) → m b
```

With $return$ and $(\gg\!=)$ operator ($bind$), the monad laws in equations 2.6 and 2.7 are rewritten as

$$return\ a \gg\!= f = f\ a, \tag{2.8}$$

$$m \gg\!= return = m, \tag{2.9}$$

and,

$$(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \to f\ x \gg\!= g). \tag{2.10}$$

The equations 2.8 and 2.9 are in correspondence with 2.6, albeit equation 2.10 is in correspondence with equation 2.7. If a given computation has an uninteresting return in a computation, $m\ ()$ for example, the operator $(\gg) :: m\ a \to m\ b \to m\ b$ which ignores the return of the first computation

```
m ≫ n = m ≫= \_ → n
```

where m and n are arbitrary monadic computations. The bind operator has a flipped version $(=\!\ll) :: (a \to m\ b) \to m\ a \to m\ b$. An interesting pattern is possible to observe, if we set $m = Identity$, the identity monad which fixes objects and morphisms, we can observe that $return$ is the id function, bind is $flip\ (\$) :: a \to (a \to b) \to b$.

Join and bind are related if one uses join gets bind for free and vice-versa, join can be written in terms of bind as

$$join = (\gg\!=\ id) \tag{2.11}$$

and bind in terms of join as

$$m \gg\!= f = join\ (fmap\ f\ m) \tag{2.12}$$

In equation 2.11, $join :: m\ (m\ b) \to m\ b$ and $(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$, id is bound to the second parameter of bind, then $a = m\ b$ because this is the only possible setup for $id :: m\ b \to m\ b$ in this case, thus join and $(\gg\!=\ id)$ has the same desired types. For 2.12, m has type $m\ a$, $fmap$ lifts f of type $a \to m\ b$ to $m\ a \to m\ (m\ b)$, hence $fmap\ f\ m$ has type $m\ (m\ b)$ and join finishes the job giving the $m\ b$ which is the same type of $m \gg\!= f$ as required.

Given a monad, one can form a category on top of it, this contruct is called a Kleisli category [Mac71].

Definition 2.13 - Let (T, μ, η) be a monad over a category $\mathcal{C}$. The Kleisli category, denoted by $\mathcal{C}_T$, is constructed using the same objects as $\mathcal{C}$. The morphisms are given by $Hom_{\mathcal{C}_T}(A, B) = Hom_{\mathcal{C}}(A, TB)$. In Haskell, this construction is represented by the following datatype:

```
data Kleisli t a b = Kleisli {runKleisli :: a → t b}
```

Given this datatype, one can derive the $Category$ typeclass instance as follows:

```
instance Monad m ⇒ Category (Kleisli m) where
  id = Kleisli return
  Kleisli f ∘ Kleisli g = Kleisli (λx → f x ≫= g)
```

It is important to note that *Kleisli* is also an instance of the 'Arrow' typeclass in Haskell. This will play a significant role in Chapter 3 when we explore an example of an effectful monoidal profunctor.

To demonstrate how monads model computations in Haskell, we will present examples of monad instances, ranging from basic to more sophisticated ones. The applicative instances for each monad will be omitted, and some of them will be discussed later in Section 2.2.6.

Maybe monad

This monad instance is used to model errors in some computations, *Maybe* is a kind $* \to *$ datatype with two value constructors *Nothing*, which indicates an erroneous computation, and *Just* for computations without errors.

```
data Maybe a = Nothing | Just a
instance Monad Maybe where
  return = Just
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
```

In this case, *return* is the constructor *Just*, the pattern matching observed in *bind* is due a computation that may fail, represented by *Nothing*, hence no computation should be done, whilst for *Just x* the monadic function f will be applied on x. Ultimately, if in some part of a chain of computations fails, *Nothing* will be the result.

List monad

A list monad is useful to model non-determinism, this instance provides a way to map a monadic function on every element of the list and collect the results in a flattened manner.

```
instance Monad [] where
  xs >>= f = [y | x ← xs, y ← f x]
```

The use of list monad gives another way to write list comprehensions, for example

```
func = [x + y | x ← [1 . . 5], y ← [1 . . 5]]
```

has the same denotation as the expression below.

```
func :: [Int]
func = do
  x ← [1 . . 5]
  y ← [1 . . 5]
  return (x + y)
```

One can observe that all filters of a list comprehension can be modelled by the guard function (see alternative functors, section 2.2.7).

IO monad

Probably the most useful monad is IO, this instance provides a way of chaining effectful computations maintaining the purity. The IO monad with the **do** notation can provide a similar syntax to an imperative language. With this monad, we can perform database operations, deal with user's inputs, handle external files, compile programs in Haskell, interact with networks and build an industrial level software. IO monad is well documented and one can find plenty of tutorials, videos, and others useful resources about it.

A code that serves us as an example for the *IO* monad involves reading a file. In this example, the program prompts the user for a filename via keyboard input and then displays the entire content of the file.

```
main :: IO ()
main = do
  putStrLn "Please enter the filename: "
  filename ← getLine
  content ← readFile filename
  putStrLn content
```

The function *getLine* :: *IO String* retrieves a *String* from the standard input. The function *readFile* :: *FilePath* → *IO String*, where *FilePath* is merely a type synonym for *String*, takes the filename as input and loads the file's content into memory. Finally, the function *putStrLn* :: *String* → *IO* () is used to display the content.

Writer monad

Writers maintain a pure code while they do logging operations. The logging is obtained by a datatype which is an instance of the Monoid type class, of kind $* \to$ *Constraint*, such as *String*. This monad does the computations while concatenates the monoid part resulting in a logged chain of logged computations.

```
data Writer w a = Writer (a, w)
instance Monoid w ⇒ Monad (Writer w) where
  return x = Writer (x, mempty)
  Writer (x, w) ≫= f =
    let
      Writer (y, w′) = f x
    in
      Writer (y, w ⊗ w′)
```

Note that datatype w must be an instance of monoid. Return provides a way to start a log with no entries, bind takes a monadic value *Writer* (x, w) and apply the monadic function f in the x variable producing a new logged computation (y, w'), the final result is the value y with the monoidal multiplication $(\otimes)$ of w and w' (the multiplication of strings is concatenation) providing the desired logging of computations.

```
inc :: Int → Writer [String] Int
inc x = Writer (x + 1, ["Increment 1"])
dec :: Int → Writer [String] Int
dec x = Writer (x − 1, ["Decrement 1"])
doNothing :: Writer [String] Int
doNothing = Writer (0, ["Do nothing"])
cter :: Int → Writer [String] Int
cter c = do
  x ← inc c
  y ← inc x
  a ← dec y
  w ← inc z
  z ← doNothing
  return w
```

A simple counter logging is displayed in the above piece of code, the functions *inc* and *dec* perform increments and decrements on the parameter while logging its activity, in the *cter* function each increment and decrement are bound to the variables x, y, z and w which they carry only the int value. At the end of *cter* returns the int w into a writer computation which has all logged operations (note that last logging is only *mempty*, which don't make any extra logs). Writers can be used to simulate a debug for pure code which they can help in describing in details every operation in a chain of complicated computations.

Reader monad

The reader monad provides a way to compute functions insider a context, i.e, an input value can travel through all function in this chain in a similar behavior as a let block with a bunch of functions applying on this value and each return bound to a different variable name.

```
instance Monad ((→) r) where
  return x = \_ → x
  f >>= g = λx → g (f x) x
```

This monad is based on functor $Hom(R, -)$, in Haskell $((\to)\ r)$ which has a functor (also applicative) instance, *return* provides us a constant function giving the input as the answer moreover $\gg\!=$ has $f :: r \to a$ and $g :: a \to r \to b$ as parameters and returns a lambda with input x of type r returning the expression $g(f\ x)x$ of type b as required (note that the output of bind is $m\ b$ and in this context $m = (\to)r$, thus the returning type should be $r \to b$).

```
funcReader :: String → String
funcReader = do
  x ← tail
  y ← head
  z ← take 2
  return (x ++ [y] ++ z)
```

In the example above, the reader monad is used to bind the results of computations by the functions *tail*, *head*, and *take* 2 to x, y, and z, respectively, for the same given string. The last line applies *return* to the concatenation of these three values, resulting in a constant function that returns the value of this expression.

Without the use of the reader monad, the piece of code can be rewritten as

```
funcReader′ :: String → String
funcReader′ w =
  let
    x = tail w
    y = head w
    z = take 2 w
  in
    x ++ [y] ++ z
```

giving the same results as before.

State monad

The state monad provides us with a way of keeping a mutable state inside pure code, e.g, it is possible to make a Stack datatype peeking and modifying its state without losing purity.

```
data State s a = State {runstate :: s → (a, s)}
```

```
instance Functor (State s) where
  fmap f (State sas) = State (λs → let (a, s′) = sas s in (f a, s′))
instance Monad (State s) where
  return x = State (λs → (s, x))
  (State sas) ≫ f = State (λs →
    let
      (a, s′) = sas s
      State sbs = f a
    in
      sbs s′)
```

A straightforward expression is given for its functor instance, the function *fmap* just makes a new state of with context of type b by applying f on a after the computation of the state. Return function provides a way to put a value in a state context without changing it while *bind* makes a new stateful computation based on an older one. This process starts with applying the function *sas* on s parameter of the lambda giving a tuple (a, s'), a new state, obtained by evaluating $f\ a$, carrying a function of type $s \to (b, s)$ is obtained. Applying *sbs* on s' terminates the process.

```
type App = (Int, Int)
readFst :: State App Int
readFst = State (λ(x, y) → (x, (x, y))
readSnd :: State App Int
readSnd = State (λ(x, y) → (y, (x, y)))
writeFst :: Int → State App ()
writeFst v = State (λ(_, y) → ((), (v, y)))
writeSnd :: Int → State App ()
writeSnd v = State (λ(x, _) → ((), (x, v)))
calc :: (Int → Int → Int) → State App String
calc op = State (λ(x, y) → (show (x ‘op‘ y), (0, 0)))
progSt :: State App String
progSt = do
  writeFst 5
  a ← readFst
  writeSnd (5 + a)
  b ← readSnd
  writeSnd (5 + b)
  calc (∗)
```

A tiny calculator with a memory of two integers only can be simulated with state monad, *App* is a type to model the calculator memory, the functions *readFst* and *readSnd* doesn't change the state however they retrieve the value stored in each position of the memory. The functions *writeFst* and *writeSnd* provides a way of writing in the memory, i.e, changing its state but they don't produce any value giving to us the type *State App* (), *calc* do the calculations converts to a printable string and resets the memory. Ultimately, the function *progSt* simulates a user operating the calculator and the final result (after running the expression *runState progSt* $(0, 0)$, where $(0, 0)$ represents the initial state of the calculator) is 75.

Using the state monad is very useful and gives the ability to read and write program states in a immutable context which is essential for making testable software.

Continuation monad

This monad model the computational passing style (CPS) keeps a value to be evaluated by a new function on future, to help the construction of a chain of various continuations, the continuation monad is needed. To define the monadic bind, in each step a new function, say f, will be expected then an older continuation will receive a lambda expression with the evaluation of f.

```
data Cont r a = Cont {runCC :: (a → r) → r}
instance Functor (Cont r) where
  fmap f (Cont artr) = Cont (λbr → artr (λx → br (f x)))
instance Monad (Cont r) where
  return x = Cont (λf → f x)
  m ≫= f = Cont (λbr →
              m ‘runCC‘ (λa → (f a) ‘runCC‘ (λb → br b)))
```

It is provided the functor instance for this monad, which is not straightforward like the previous ones, the pattern matching de-constructs $Cont$ giving the function $artr :: (a \to r) \to r$, a lambda $\lambda x \to br\ (f\ x)$ is passed to $artr$, note that x has type a thus $(f\ x)$ evaluates to type b, this value is then evaluated into $br :: b \to r$, giving the desired value of type r. For the monad instance, the function $m :: (a \to r) \to r$ will be extracted from de datatype $Cont$ by the projection function $runCC$, another continuation is provided with $(\lambda a \to (f\ a)\ `runCC`\ (\lambda b \to br\ b))$, applying f on a with get a new continuation of type $Cont\ r\ b$, the lambda given is $(\lambda b \to br\ b)$ which evaluates to a value of type r.

2.2.5 Comonads

As observed in the last section, monads are useful in category theory and can encode many concepts in functional programming, if one reverses the arrows of η and μ then we end up with their dual counterpart, comonads. Comonads can encode programming concepts and are a very useful tool, in conjunction with monads, to study adjunctions.

Definition 2.14 - A comonad on a category $\mathcal{C}$, can be viewed as a monad on $\mathcal{C}^{op}$, is a triple (W, ε, δ), such that W is an endofunctor, $\varepsilon :: W \to I$ and $\delta :: W \to W^2$ are both natural transformations, satisfying

$$\delta \circ \varepsilon W = id_{W^2} = \delta \circ W\varepsilon \tag{2.13}$$

$$\begin{tikzcd}
W^2 \arrow[r, "W\varepsilon"] \arrow[d, "\varepsilon W"'] \arrow[dr, equal] & W \arrow[d, "\delta"] \\
W \arrow[r, "\delta"] & W^2
\end{tikzcd}$$

Figure 2.13: Commutative diagram for the identity condition of the counit ε operator

$$\delta \circ \delta = W\delta \circ \delta \tag{2.14}$$

In Haskell, comonads can be modelled as a kind $(* \backslash - > *) \to \mathit{Constraint}$ called Comonad, the operator ε is called *coreturn* or *counit* while δ is called *cojoin*, *duplicate* or *comultiplication*.

$$\begin{tikzcd}
W \arrow[r, "\delta"] \arrow[d, "\delta"'] & W^2 \arrow[d, "W\delta"] \\
W^2 \arrow[r, "\delta"] & W^3
\end{tikzcd}$$

Figure 2.14: Commutative diagram for the naturality condition δ operator

```
class Functor w ⇒ Comonad w where
  coreturn :: w a → a
  cojoin :: w a → w (w a)
```

As we observed in monads, in Haskell comonads can be used in the same manner, instead of using cojoin it is used extend which is the dual of bind. The natural transformation cojoin is often called extract.

```
class Functor w ⇒ Comonad w where
  extract :: w a → a
  extend :: w a → (w a → b) → w b
```

As discussed in monads section, cojoin and extend are related, this relation can be written as

$$cojoin = (extend\ id) \tag{2.15}$$

and bind in terms of join as

$$extend \text{ w f} = fmap\ f\ (cojoin \text{ w}). \tag{2.16}$$

In the equations 2.15 and 2.16, the same reasoning is applied to understand both equations.

Stream comonad

A stream is a recursive type with a shape of an infinite list, i.e, they do not have a value to indicate the end of it's recursive iteration (in case of lists, the value [ ]). With streams, the *head* and *tail* are total implying that those functions never causes exceptions (remember that *head* [ ] and *tail* [ ] causes exceptions because they're both partial functions). This fact gives that lists cannot be made into a comonadic interface for free and a stream can be fit into one.

```
data Stream a = Cons a (Stream a)
instance Comonad Stream where
  extract (Cons a _) = a
  cojoin (Cons a as) = Cons (Cons a as) (cojoin as)
```

The *extract* function is just taking the head of a stream while *extract* create a stream of streams, name it *ss* for simplicity, it's value is the whole stream *a* and the tail *as* and the tail of *ss* is recursively called on *as* from the argument. This process makes shifting windows of data with a different focus.

Store comonad

The store comonad (costate comonad) can be seen as a dual of the state monad (this can be obtained as an adjunction, see next chapter). Store acts like a mapping of keys *s* and values *a* and a another reference to *s* acting like a current position.

```
data Store s a = Store (s → a) s
instance Comonad (Store s) where
  extract (Store f x) = f x
  cojoin (Store f s) = Store (Store f) s
```

A value can be extracted from this map by the *extract* function, given a function $f :: s \to a$ and a value x of type s, $f\ x$ has type s and is the desired extracted value. In the function *cojoin* a store of stores with a new index of type s are created, i.e, the value *Store* f has type $s \to \mathit{Store}\ s\ a$, and the whole cojoin expression has type $\mathit{Store}\ (\mathit{Store}\ s\ a)\ a$ creating the duplication desired. Store plays a fundamental role to abstract and understand categorical lenses.

2.2.6 Applicative Functors

Like monads, an applicative functor can model computations as well because of $*$ operator which gives a way to combine an effectful function of type $a \to b$ with a computation of type a producing a sequence of effectful computations.

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

While monads can use previous computations to make new ones, applicatives cannot and they only sequence computations in a linear manner. Pure, like return, make a new trivial computation by inserting a value of type a in the computational context f. All monads are applicative functors (the converse is not true) implying that in Haskell all monads instances are instances of the type class applicative, to elucidate this consider the function

```
newApp :: Monad m ⇒ m (a → b) → m a → m b
newApp mfunc m = do
  f ← mfunc
  a ← m
  return (f a)
```

we see that *newApp* is just $(*)$ thus every monad is an applicative as required.

Example 21 - Maybe instance

For maybes, *pure* is the value constructor *Just*, the $*$ operator applies f on x on the right pattern matching, when *Nothing* is present the same is returned.

```
instance Applicative Maybe where
  pure = Just
  Nothing ∗ _ = Nothing
  _ ∗ Nothing = Nothing
  Just f ∗ Just x = Just (f x)
```

Example 22 - List instance In the list case, an argument with a list of functions fs are applied to the list of values x inside a list comprehension.

```
instance Applicative [] where
  pure x = [x]
  fs ∗ xs = [f x | f ← fs, x ← xs]
```

Example 23 - State instance

The state instance is a little bit harder to grasp than the last two presented, the functions *sabs* is applied on s giving a function f, which is the target for the application, and a new state s' which is an argument to *sas*, in the next line, then when get a value of type a and other new state s'', thus the return is $(f\ a, s'')$ which has the desired type $f\ b$. This process is analogous as their monad counterpart.

```
instance Applicative (State s) where
  pure x = State $ λs → (x, s)
  (State sabs) ∗ (State sas) = State $ λs →
    let
      (f, s′) = sabs s
      (a, s″) = sas s′
    in
      (f a, s″)
```

2.2.7 Alternative functors

An alternative functor is defined as a monoid on the monoidal category of applicative endofunctors (see chapter 3), those functors are inspired in computations that may fail, one example that can be a excellent fit for alternatives is selective backtracking in grammar, i.e, when we try a match against a pattern and succeeds then some success routines are called, if it fails another match is tried. This choice-like operation is possible in a lazy language.

```
class Applicative f ⇒ Alternative f where
  empty :: f a
  ⊕ ::f a → f a → f a
```

Alternative type class has two functions, *empty* that represents a neutral element and the other function represents a choice of computations, to be an alternative, a functor must be an instance of applicative. This type class should satisfy the unit and associativity laws.

Example 24 - Maybe instance The alternative instance are mad by a *Nothing* as *empty* and the choice operator works like the description above, is the first argument is a just then the second one is ignored (even if it is an undefined value), if the first one is a *Nothing* the second one is returned. If the two computations gives *Nothing* as an answer then the return is also *Nothing*.

```
instance Alternative Maybe where
  empty = Nothing
  Just x ⊕ _ = Just x
  Nothing ⊕ x = x
```

Example 25 - List instance

For lists, the alternative instance is the same as a monoid, the choice operator is just the concatenation because of the non-deterministic nature of list functor.

```
instance Alternative [ ] where
  empty = [ ]
  ⊕ = (++)
```

2.3 Monoidal Categories

This section provides the foundation for defining monoidal categories, profunctors, and end/-coends [Lor21, CW01]. These concepts are essential in deriving various results throughout this work. Profunctors play a significant role in understanding the key ideas of this research, and the monoidal categories represent the categorification of a monoid.

2.3.1 Ends and Coends

Firstly, a brief review of limits (colimits by duality) is needed to grasp the notion of an end (coend) of a functor. Limits generalize universal constructions such as products, pullbacks, equalizers, and so on. The end of a functor gives an extra notational power to describe concepts such as natural transformations or stating the always useful Yoneda lemma (see Lemmas 1 and 2). Let $F : \mathcal{J} \to \mathcal{C}$ be a functor from a small category of shapes (index) $\mathcal{J}$ and an arbitrary category $\mathcal{C}$.

Definition 2.15 - A *cone* to F, is an object N of $\mathcal{C}$ equipped with a family of morphisms $\psi_X : N \to F(X)$, with X varying as an index in $\mathcal{J}$ such that for any morphism f in $\mathrm{Hom}_{\mathcal{J}}(X, Y)$, the condition $F(f) \circ \psi_X = \psi_Y$ is satisfied.

Definition 2.16 - The limit of F is a cone L with a family of morphism $\psi_X : L \to F(X)$ such that, for every other cone N with morphisms $\phi_X : N \to F(X)$ for all X of $\mathcal{J}$, there exists a unique arrow $u : N \to L$ such that satisfies the condition $\psi_X \circ u = \phi_X$.

An example is when one has $\mathcal{J} = \mathbf{2}$, the discrete category of two elements, and $\psi_X = \Delta_X$, the diagonal functor, yielding the universal construction of a binary cartesian product.

In contexts where functors of a product category have one contravariant and one covariant argument, we're naturally led to use ends and coends. These tools allow us to aggregate effects across all objects in a category. However, to handle these behaviors effectively, we introduce the notion of dinatural transformations, ensuring a coherent interplay between contravariant and covariant components.

Definition 2.17 - A dinatural transformation between functors $F, G : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{D}$, written $\alpha : F \to G$, is a family of diagonal morphisms $\alpha_X : F(X, X) \to G(X, X)$ such that for every morphism $f : X \to X'$ the following diagram commutes.

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

The notion of an end and coend, which are based on wedges and cowedges respectively, facilitates calculations that prove to be essential in subsequent chapters of this work.

Definition 2.18 - A wedge to $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{D}$, is an object W of $\mathcal{D}$, i.e., a dinatural transformation from the constant functor Δ_W to F, equipped with a family of morphisms $\alpha_X : W \to F(X, X)$, with X varying in $\mathcal{C}$ such that for any morphism f in $\mathrm{Hom}\,\mathcal{C}(X, X')$, the condition $F(f, id) \circ \alpha X' = F(id, f) \circ \alpha_X$ is satisfied.

Definition 2.19 - The end of a functor $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{D}$ is a wedge E of $\mathcal{D}$ with a family of morphism $\alpha_X : E \to F(X, X)$ such that, for every other wedge W of $\mathcal{D}$ with morphisms

$\beta_X : W \rightarrow F(X,X)$ there exists, for all X of $\mathcal{C}$, a unique arrow $\gamma : W \rightarrow E$ such that satisfies the condition $\alpha_X \circ \gamma = \beta_X$.

The object W defined in (2.19) is denoted by $\int_X F(X,X)$, the end of F giving a notation for describing, for example, natural transformations between functors G and H as $Nat(G,H) = \int_X GX \rightarrow HX$. It is important to note that the functor $F(-) = G(-) \rightarrow H(-)$ is employed to derive the natural transformation. The notation can also help to state the following Yoneda lemma which the proof can be found in the work of Mac Lane [Mac71].

Lemma 1 - (Covariant Yoneda lemma) Let $\mathcal{C}$ be a locally small category, and $F : \mathcal{C} \rightarrow Set$ a covariant functor, then there is an isomorphism

$$\int_X Set(Hom_{\mathcal{C}}(A,X), FX) \cong FA \tag{2.17}$$

natural in $X \in \mathrm{ob}(\mathcal{C})$.

The above lemma also has a contravariant form.

Lemma 2 - (Contravariant Yoneda lemma) Let $\mathcal{C}$ be a locally small category, and $F : \mathcal{C}^{op} \rightarrow Set$ a contravariant functor, then there is an isomorphism

$$\int_X Set(Hom_{\mathcal{C}}(X,A), FX) \cong FA \tag{2.18}$$

natural in $X \in \mathrm{ob}(\mathcal{C})$.

The duality principle gives us the coend of a functor, which is the universal cowedge and is denoted by $\int^X F(X,X)$. Coends generalize constructions such as pushouts, coproducts, and coequalizers, to name a few.

A commuting property of ends and coends gives us another tool for doing calculations; the next lemma states this and its proof is provided in the work of Loregian [Lor15].

Lemma 3 - Let F be a functor $F : \mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathcal{D}$, then for every D object of $\mathcal{D}$ the following holds.

$$Hom_{\mathcal{D}}(\int^{C \in \mathcal{C}} F(C,C), D) \cong \int_{C \in \mathcal{C}} Hom_{\mathcal{D}}(F(C,C), D) \tag{2.19}$$

$$Hom_{\mathcal{D}}(D, \int^{C \in \mathcal{C}} F(C,C)) \cong \int_{C \in \mathcal{C}} Hom_{\mathcal{D}}(D, F(C,C)) \tag{2.20}$$

2.3.2 Definition of a Monoidal Category

The definition of a monoidal category [RJ17] gives us a minimal framework for defining the categorical notion of a monoid.

Definition 2.20 - A monoidal category is a sextuple ($\mathcal{C}$, $\otimes$, I, α, ρ, λ) 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 such that the diagrams below commute.

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

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

If the isomorphisms ρ, λ and α are equalities then the monoidal category is called strict, if there is a natural isomorphism $\gamma_{AB} : A \otimes B \to B \otimes A$ the monoidal category is called symmetric.

A monoidal category is *closed* if there is an additional functor, called the internal hom, $\Rightarrow : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ 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 $A \to B$.

Definition 2.21 - A *monoid* in a monoidal category $\mathcal{C}$ is the tuple (M, e, m) where M is an object of $\mathcal{C}$, $e : I \to M$ is the unit morphism and $m : M \otimes M \to M$ is the multiplication morphism, satisfying

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

The following commuting diagrams represent those laws.

$$\begin{tikzcd}
M \otimes I \arrow[r, "id\otimes e"] \arrow[dr, "\rho"'] & M \otimes M \arrow[d, "m"'] & I \otimes M \arrow[r, "e\otimes id"] \arrow[dr, "\lambda"'] & M \otimes M \arrow[d, "m"'] \\
& M & & M
\end{tikzcd}$$

$$\begin{tikzcd}
M \otimes (M \otimes M) \arrow[r, "id\otimes m"] & M \otimes M \arrow[dr, "m"] & \\
& & M \\
(M \otimes M) \otimes M \arrow[uu, "\alpha"] \arrow[r, "m\otimes id"] & M \otimes M \arrow[ur, "m"] &
\end{tikzcd}$$

Example 26 - Let $\mathcal{C}$ be a category. In the category of endofunctors of $\mathcal{C}$, with the composition of endofunctors denoted by $\circ$ serving as the tensor, and the identity functor acting as the unit, we obtain a monad. This monad can be viewed as the monoid arising from this configuration [RJ17].

Sometimes the preservation of the identity element and the monoidal multiplication are desired properties to have, and later in Chapter 4, one concrete example of such an idea is provided.

Definition 2.22 - Given two monoids (M_1, m_1, e_1) and (M_2, m_2, e_2) in a monoidal category $\mathcal{C}$, a monoid homomorphism between them is an arrow $f : M_1 \to M_2$ in $\mathcal{C}$ such that the following diagram commutes:

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

Monoids in a monoidal category $\mathcal{C}$, together with monoid homomorphisms, form the category $\mathrm{Mon}(\mathcal{C})$ [RJ17].

2.3.3 Profunctors

Definition 2.23 - Given two categories $\mathcal{C}$ and $\mathcal{D}$, a profunctor [Lei03] from $\mathcal{C}$ to $\mathcal{D}$ is a functor $P : \mathcal{C}^{op} \times \mathcal{D} \to Set$. Explicitly, it consists 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) $\mathcal{D}(d, b) \times P(a, d) \to P(a, b)$;
- for all a, c objects of $\mathcal{C}$ and b object of $\mathcal{D}$, a function (right action) $P(a, b) \times \mathcal{C}(c, a) \to P(c, b)$.

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.

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

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

Definition 2.24 - Let $\mathcal{C}$ and $\mathcal{D}$ small categories, $\mathrm{Prof}(\mathcal{C}, \mathcal{D})$ is the profunctor category consisting of profunctors as objects, natural transformations as morphisms, and vertical composition to compose them.

The profunctor category inherits some structure from the functor category $Set^{\mathcal{C}}$ 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 is also terminal and initial profunctors given by $1_p(S, T) = \{*\}$ and $0_p(S, T) = \varnothing$, i.e., constant profunctors on initial and terminal objects in Set. If the target of a profunctor that is not Set, but some other category, say $\mathcal{E}$, with binary products and coproducts, initial and terminal objects, the profunctor category based on top of $\mathcal{E}$ will also have these constructs.

This is because the profunctor category is a functor category, just like $Set^{\mathcal{C}}$, and the constructions used in the profunctor category are just the universal constructions used in the functor category. The only difference is that the domain and codomain of the functors are flipped. In addition, the requirement for the target category to have binary products, coproducts, initial and terminal objects ensure that the profunctor category also has these constructs.

A important tool for doing calculations with profunctors is the Fubini's theorem that allows us to commute the integration sign. This theorem will be useful when working with Day convolution.

Theorem 4. (Fubini's Theorem for profunctors and coends) Given a profunctor $P : \mathcal{C}^{\mathrm{op}} \times \mathcal{D} \to Set$, the following isomorphism holds:

$$\int^{X \in \mathcal{C}^{\mathrm{op}}} \Big(\int^{Y \in \mathcal{D}} P(X, Y) \Big) \cong \int^{(X,Y) \in \mathcal{C}^{\mathrm{op}} \times \mathcal{D}} P(X, Y) \cong \int^{Y \in \mathcal{D}} \Big(\int^{X \in \mathcal{C}^{\mathrm{op}}} P(X, Y) \Big)$$

Proof. The proof of this a little bit technical and can be found in specialized literature on coends [Lor21, CW01]. □

2.3.4 Day Convolution

The Day convolution is a tensor in a monoidal category of endofunctors that results in the notion of a monoidal (applicative) functor [RJ17]. In this work, the profunctor version of this tensor plays a key role in defining a monoidal profunctor and a semiarrow. This section presents the results that demonstrate the profunctor category, endowed with the Day convolution for profunctors as its tensor, forms a monoidal category.

Definition 2.25 - Let $\mathcal{D}$ be a small monoidal category and $F, G : \mathcal{D} \to Set$, the Day convolution [Day70] of F and G is another functor (in T) given by

$$(F \star G)T = \int^{X,Y \in Ob(\mathcal{D})} FX \times GY \times \mathcal{D}(X \otimes Y, T). \tag{2.21}$$

The coend (or an end when present) in this definition can have a notational reduction to

$$\int^{XY} FX \times GY \times \mathcal{D}(X \otimes Y, T)$$

whenever the context is clear.

The same reasoning is now applied to the category Prof of profunctors letting $\mathcal{D} = \mathcal{C}^{op} \times \mathcal{C}$, where $\times$ describes the product category. This definition uses categorical calculations using properties of ends and co-ends for any object (S, T) in this category.

$$\begin{aligned}
&(P \star Q)(S, T)\\
&\quad= \int^{ABCD} P(A, B) \times Q(C, D) \times [\mathcal{C}^{op} \times \mathcal{C}]((A, B) \otimes (C, D), (S, T))\\
&\quad\cong \int^{ABCD} P(A, B) \times Q(C, D) \times [\mathcal{C}^{op} \times \mathcal{C}]((A \otimes C, B \otimes D), (S, T))\\
&\quad\cong \int^{ABCD} P(A, B) \times Q(C, D) \times \mathcal{C}^{op}(A \otimes C, S) \times \mathcal{C}(B \otimes D, T)\\
&\quad\cong \int^{ABCD} P(A, B) \times Q(C, D) \times \mathcal{C}(S, A \otimes C) \times \mathcal{C}(B \otimes D, T)
\end{aligned}$$

It is important to note that the profunctor $J(S, T) = (\mathcal{C}^{op} \times \mathcal{C})((I, I), (A, B)) \cong \mathcal{C}(S, I) \times \mathcal{C}(I, T)$ is a unit for $\star$. In the category **Set**, when I is a singleton set, we have $J(A, B) \cong \mathcal{C}(I, B) \cong B$.

Proposition 5. 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 [Lor21] 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 the $\star$ is required to define a monoidal profunctor category.

Proposition 6. 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.

$$\begin{aligned}
&((P \star Q) \star R)(S,T) \\
&\quad = \int^{ABCD} (P \star Q)(A,B) \times R(C,D) \times \mathcal{C}(S, A \otimes C) \times \mathcal{C}(B \otimes D, T) \\
&\quad \cong \int^{ABCD} \left(\int^{EFGH} (P(E,F) \times Q(G,H) \times \mathcal{C}(A, E \otimes G) \times \mathcal{C}(F \otimes H, B)) \right) \\
&\qquad\qquad \times R(C,D) \times \mathcal{C}(S, A \otimes C) \times \mathcal{C}(B \otimes D, T) \\
&\quad \cong \int^{ABCDEFGH} P(E,F) \times Q(G,H) \times \mathcal{C}(A, E \otimes G) \times \mathcal{C}(F \otimes H, B) \\
&\qquad\qquad \times R(C,D) \times \mathcal{C}(S, A \otimes C) \times \mathcal{C}(B \otimes D, T) \\
&\quad \cong \int^{CDEFGH} P(E,F) \times Q(G,H) \times R(C,D) \times \mathcal{C}(S, (E \otimes G) \otimes C) \\
&\qquad\qquad \times \mathcal{C}((F \otimes H) \otimes D, T) \\
&\quad \cong \int^{EFGHCD} P(E,F) \times Q(G,H) \times R(C,D) \times \mathcal{C}(S, E \otimes (G \otimes C)) \\
&\qquad\qquad \times \mathcal{C}(F \otimes (H \otimes D), T) \\
&\quad \cong \int^{EFIJGHCDAB} P(E,F) \times Q(G,H) \times R(C,D) \times \mathcal{C}(A, G \otimes C) \\
&\qquad\qquad \times \mathcal{C}(F \otimes H, B) \times \mathcal{C}(S, E \otimes A) \times \mathcal{C}(F \otimes B, T) \\
&\quad \cong \int^{EFAB} P(E,F) \times (Q \star R)(A,B) \times \mathcal{C}(S, E \otimes A) \times \mathcal{C}(F \otimes B, T) \\
&\quad \cong (P \star (Q \star R))(S,T)
\end{aligned}$$

□

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 $(Prof(\mathcal{C}, \mathcal{D})), \star, J)$ is a monoidal category.

Theorem 7. Let $\mathcal{C}$ and $\mathcal{D}$ are monoidal small categories. Then $(Prof(\mathcal{C},\mathcal{D})),\star,J)$ is a monoidal category.

Proof. Since $\mathcal{C}$ and $\mathcal{D}$ are monoidal categories, $\star$ is a bifunctor by construction, and by Proposition 5 and 6 gives the desired morphisms, it follows that $(Prof(\mathcal{C},\mathcal{D})),\star,J)$ 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 [RJ17]).

Proposition 8. 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

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

which is natural in P, Q and R.

Proof. 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 [Lor21].

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

In the equation of Proposition 8, when $P = Q = R$, we obtain the following isomorphism, which is useful for defining a monoid in the profunctor category $Prof$ with Day convolution as its tensor:

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

One final remark is that the next chapter considers the fictitious category Hask [DHJG06] as a small monoidal category and $Prof(\text{Hask}, \text{Hask})$ as the small category of profunctors, which facilitates reasoning about the Haskell code necessary to comprehend the principal categorical tools employed in this work.

This section provided the tools necessary for defining a monoid in the profunctor category with the Day convolution as its tensor. This monoid is referred to as a monoidal profunctor .

2.4 Summary

In this chapter, we have covered the necessary background on Haskell and Category Theory, providing an overview of important concepts such as syntax, algebraic data types, parametric polymorphism, higher-order functions, typeclasses, arrows, categories, functors, natural transformations, monads, comonads, applicative functors, alternative functors, and monoidal categories. We have also examined ends, coends, profunctors, and the Day Convolution, which are essential to understanding the more advanced topics that follow.

We now are able to investigate if Table 1.2 can be completed using monoidal profunctors and its effectful version in the subsequent chapter. The concepts presented here serve as a basis for understanding the relationships between monoidal profunctors and their applications. As we move forward, readers will gain a deeper understanding of the interplay between Haskell and Category Theory, which will facilitate the ability to work with these mathematical constructs.

The next chapter will build upon the concepts presented here, focusing on the definition and properties of monoidal profunctors, their connection to other categorical concepts, and their relevance to functional programming.

Chapter 3

Monoidal Profunctors

This chapter introduces monoidal profunctors in category theory, along with their definitions, laws, and the results needed to derive these concepts. It presents another notion of monoids in monoidal categories, as seen in the works of Rivas and Jaskelioff [RJ17]. Additionally, the computational model is presented in Haskell, including some basic instances, useful insights, and examples. We finish the chapter presenting the newly defined effectful monoidal profunctor, that introduces effects when splitting and merging a plain type in a tuple of types. Although some results and definitions related to monoidal profunctors and semiarrows can be found in the existing Haskell literature [PGW17] and folklore [Mil], these concepts are not thoroughly explored. In this work, we organize the available information and discuss it further through the lens of a monoid in a monoidal category methodology.

3.1 Definition

We define *monoidal profunctors* as monoids in the monoidal category of profunctors of theorem 7. This section aims to provide the essential categorical tool to derive a Haskell representation for a monoid on a monoidal category of profunctors. This section also discusses the free monoidal profunctor construction and a representation theorem for profunctors.

3.1.1 A monoid in monoidal profunctors

The unit and the multiplication of this monoid are a direct consequence of Yoneda's lemma and Proposition 8.

Proposition 9. Let $(\mathcal{C},\otimes,I)$ be a small monoidal category, $P : \mathcal{C}^{op} \times \mathcal{C} \to Set$ be a profunctor, and S, T two objects of $\mathcal{C}$. Then $\int_{S,T} Set(J(S,T), P(S,T)) \cong P(I,I)$.

Proof.

$$\begin{aligned}
\int_{S,T} Set(J(S,T), P(S,T)) &\cong \int_{S,T} Set(S,I) \times Set(I,T) \to P(S,T) \\
&\cong \int_{S,T} Set(S,I) \to P(I,T) \to P(S,T) \\
&\cong \int_{S} Set(S,I) \to Set(S,I) \\
&\cong P(I,I)
\end{aligned}$$

□

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

Definition 3.1 - Let $(\mathcal{C}, \otimes, I)$ 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 profunctors J and P, $e : J \to P$, equivalent to $e : P(I, I)$ by Proposition 9, and the multiplication is $m : P \star P \to P$ which is isomorphic to the family of morphisms $V(m)_{ABCD} = P(A, B) \times P(C, D) \to P(A \otimes C, B \otimes D)$. Such a monoid is called a *monoidal profunctor.*

The unit of a monoidal profunctor is a natural transformation $e : J \to P$, which by proposition 9 is isomorphic to $e : P(I, I)$. The multiplication is a natural transformation $m : P{\star}P \to P$, which by proposition 8 is equivalent to the family above.

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

Proposition 10. Let $(\mathcal{C}, \otimes, I)$ be a small monoidal category, and P, Q profunctors, then

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

defines an internal hom on the profunctor category.

Proof. Firstly, a calculation of the $P \Rightarrow Q$ profunctor is derived.

$$\begin{aligned}
&(P \Rightarrow Q)(X, Y) \cong Nat([\mathcal{C}^{op} \times \mathcal{C}]((X, Y), -), (P \Rightarrow Q)-) \\
&\cong Nat([\mathcal{C}^{op} \times \mathcal{C}]((X, Y), -) \star P, Q) \\
&\cong \int_{VW} ([\mathcal{C}^{op} \times \mathcal{C}]((X, Y), -) \star P)(V, W) \to Q(V, W) \\
&\cong \int_{VW} \left(\int^{ABCD} [\mathcal{C}^{op} \times \mathcal{C}]((X, Y), (A, B)) \times P(C, D) \times \mathcal{C}(V, A \otimes C) \times \mathcal{C}(B \otimes D, W) \right) \\
&\qquad \to Q(V, W) \\
&\cong \int_{VW} \left(\int^{ABCD} \mathcal{C}^{op}(A, X) \times \mathcal{C}(Y, B) \times P(C, D) \times \mathcal{C}(V, A \otimes C) \times \mathcal{C}(B \otimes D, W) \right) \\
&\qquad \to Q(V, W) \\
&\cong \int_{VW} \left(\int^{BCD} \mathcal{C}(Y, B) \times P(C, D) \times \mathcal{C}(V, X \otimes C) \times \mathcal{C}(B \otimes D, W) \right) \to Q(V, W) \\
&\cong \int_{VW} \left(\int^{CD} P(C, D) \times \mathcal{C}(V, X \otimes C) \times \mathcal{C}(Y \otimes D, W) \right) \to Q(V, W) \\
&\cong \int_{VWCD} (P(C, D) \times \mathcal{C}(V, X \otimes C) \times \mathcal{C}(Y \otimes D, W)) \to Q(V, W) \\
&\cong \int_{WCD} P(C, D) \times \mathcal{C}(Y \otimes D, W) \to \left(\int_{V} \mathcal{C}(V, X \otimes C) \to Q(V, W) \right) \\
&\cong \int_{WCD} (P(C, D) \times \mathcal{C}(Y \otimes D, W)) \to Q(X \otimes C, W) \\
&\cong \int_{CD} P(C, D) \to \left(\int_{W} \mathcal{C}(Y \otimes D, W) \to Q(X \otimes C, W) \right) \\
&\cong \int_{CD} P(C, D) \to Q(X \otimes C, Y \otimes D)
\end{aligned}$$

To show that $\Rightarrow$ defines an exponential, one needs to check that $Prof(P{\otimes}Q, R) \cong Prof(P, Q \Rightarrow R)$ is a natural isomorphism of profunctors. Using the above exponential expansion, choosing a projection and Proposition 8,

$$\begin{aligned}((P \otimes Q) \Rightarrow R)(X, Y) &\cong \int_{UV} (P \otimes Q)(U, V) \to R(X \otimes U, Y \otimes V) \\ &\cong \int_{ABCD} P(A, B) \to Q(C, D) \to R(X \otimes (A \otimes C), Y \otimes (B \otimes D))\end{aligned}$$

and,

$$\begin{aligned}(P \Rightarrow (Q \Rightarrow R))(X, Y) &\cong \int_{ST} P(S, T) \to (Q \Rightarrow R)(X \otimes S, Y \otimes T) \\ &\cong \int_{STVZ} P(S, T) \to Q(V, Z) \to R((X \otimes S) \otimes V, (Y \otimes T) \otimes Z).\end{aligned}$$

To exhibit the morphisms currying and uncurrying, a combination of two end projections, two applications of the $eval : (A \Rightarrow B) \times A \to B$, since both expressions live in Set, and two liftings for the profunctor R, $R(\alpha, \alpha^{-1})$ for the former and $R(\alpha^{-1}, \alpha)$ for the latter, suffices. They are inverses of each other trivially. □

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

3.2 Implementation in Haskell

In this section, we implement in Haskell the concepts of profunctors, Day convolution, and monoidal profunctors defined before.

3.2.1 Profunctor typeclass

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 below.

$$\begin{aligned} dimap\ id\ id &= id \\ dimap\ (f \circ g)\ (h \circ i) &= dimap\ g\ h \circ dimap\ f\ i \end{aligned}$$

Note that *dimap* encompasses the definitions of left and right actions of a profunctor. In the *profunctors* library [Kmec], there are two functions named *lmap* and *rmap* corresponding to these actions.

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 (→), is the most basic example of a profunctor.

```
instance Profunctor (→) where
  dimap ab cd bc = cd ∘ bc ∘ ab
```

One notion captured by *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)
```

3.2.2 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 ×), 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.

```
ρ :: Profunctor p ⇒ Day p I ⇝ p
ρ (Day pab (I d) sac bdt) = dimap (fst ∘ sac) (λb → bdt b d) pab
λ :: Profunctor q ⇒ Day I q ⇝ q
λ (Day (I b) qcd sac bdt) = dimap (snd ∘ sac) (λd → bdt b d) qcd
```

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

```
α⊗ :: (Profunctor p, Profunctor q, Profunctor r)
  ⇒ Day (Day p q) r
  ⇝ Day p (Day q r)
α⊗ (Day (Day p q s1 f) r s2 g) = Day p (Day q r f1 f2) f3 f4
  where
    f1           = first′ (snd ∘ s1) ∘ s2
    f2 d1 d2     = (d2, λx → f x d1)
    f3           = first′ (fst ∘ s1 ∘ (fst ∘ s2)) ∘ diag
    f4 b1 (d2, h) = g (h b1) d2
γ :: (Profunctor p, Profunctor q) ⇒ Day p q ⇝ Day q p
γ (Day p q sac bdt) = Day q p (swap ∘ sac) (flip bdt)
  where swap (x, y) = (y, x)
```

Functions ρ, λ, and $\alpha^{\otimes}$ are natural isomorphisms. We leave the definition of the inverses as an exercise for the reader.

3.2.3 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 8, which gives the multiplication a type $\int_{ABCD} P(A, B) \times P(C, D) \to P(A \otimes C, B \otimes D)$ allowing to write the following class in Haskell.

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

satisfying the monoid laws

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

where the helper functions $\mathit{diagr} :: x \to ((), x)$, $\mathit{diagl} :: x \to (x, ())$, $\alpha_{\otimes}^{-1} :: ((x, y), z) \to (x, (y, z))$, and $\alpha^{\otimes} :: (x, (y, z)) \to ((x, y), z)$ 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.

```
lmap₂ :: MonoPro p ⇒ (s → (a, c)) → p a b → p c d → p s (b, d)
lmap₂ f pa pc = dimap f id (pa ⋆ pc)
rmap₂ :: MonoPro p ⇒ ((b, d) → t) → p a b → p c d → p (a, c) t
rmap₂ f pa pc = dimap id f (pa ⋆ pc)
```

which can work together as one function

```
rlmap :: MonoPro p
  ⇒ ((b, d) → t)
  → (s → (a, c))
  → p a b
  → p c d
  → p s t
rlmap f g pa pc = dimap f g (pa ⋆ pc)
```

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 $(,)$.

When selecting an appropriate monoidal profunctor, it naturally exhibits applicative functor behavior.

```
appToMonoPro :: MonoPro p ⇒ p s (a → b) → p s a → p s b
appToMonoPro pab pa = dimap diag (uncurry ($)) (pab ⋆ pa)
```

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

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

where $\mathit{zip}' :: \mathit{Applicative}\ f \Rightarrow (f\ a, f\ b) \to f\ (a, b)$ 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 $(\to)$ and then regrouped together using the applicative (monoidal) behavior of *zip′*.

One example of a monoidal profunctor is when one has a direct product between a divisible (oplax monoidal) functor f and an applicative functor g. The divisible functor type class is found on the package contravariant.

```
class Contravariant f ⇒ Divisible (f :: ∗ → ∗) where
  conquer :: (a → ()) → f a
  divide :: (a → (b, c)) → f b → f c → f a
```

It is important to notice that *divide id* has the type $f\ b \rightarrow f\ c \rightarrow f\ (b, c)$ which is similar to a monoidal functor but in the opposite category.

A type for the product between a contravariant functor and a covariant one is just the following type.

```
data Divapp f g a b = Divapp {unContra :: (f a), unCov :: (g b)}
```

When f is a contravariant functor and g is a functor, then *Divapp* is a profunctor.

```
instance (Contravariant f, Functor g) ⇒ Profunctor (Divapp f g) where
  dimap f g (Divapp fa gb) =
    Divapp (contramap f fa) (fmap g gb)
```

Finally, an instance of *MonoPro* is derived by assuming f being a divisible functor, and g an applicative functor.

```
instance (Divisible f, Applicative g) ⇒ MonoPro (Divapp f g) where
  mpempty = Divapp (conquer (\_ → ())) (pure ())
  (Divapp fa gb) ⋆ (Divapp fc gd) =
    Divapp (divide id fa fc) (pure (, ) ∗ gb ∗ gd)
```

3.3 Free monoidal profunctor

The notion of a fixpoint of an initial algebra enables a definition of the free structure for a monoidal profunctor.

Definition 3.2 - Let $\mathcal{C}$ be a category, given an endofunctor $F : \mathcal{C} \rightarrow \mathcal{C}$, a F-algebra consists of an object A of $\mathcal{C}$, the carrier of the algebra, and an arrow $\alpha : F(A) \rightarrow A$. A morphism $h : (A, \alpha) \rightarrow (B, \beta)$ of F-algebras is an arrow $h : A \rightarrow B$ in $\mathcal{C}$ such that $h \circ \alpha = \beta \circ F(h)$.

$$\begin{tikzcd}
F(A) \arrow[r, "F(h)"] \arrow[d, "\alpha"'] & F(B) \arrow[d, "\beta"] \\
A \arrow[r, "h"] & B
\end{tikzcd}$$

The category of F-algebras and its morphisms on a category $\mathcal{C}$ is called $F\text{-}Alg(\mathcal{C})$.

The existence of a free monoidal profunctor is guaranteed by the following proposition.

Proposition 11. Let $(\mathcal{C}, \otimes, I)$ be a monoidal category with exponentials. 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.

Proof. See Proposition 2.6, page 6, in [RJ17] (extended version). □

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

3.3.1 Representation Theorem

In the work of O'Connor and Jaskelioff [JO14], it was derived a representation theorem for second order functionals that helps to derive optics. In this work, the unary version of the mentioned theorem is needed.

Theorem 12. (Unary representation) Consider an adjunction $(-^{*}) \vdash U : \mathcal{E} \to \mathcal{F}$, where $\mathcal{F}$ is small and $\mathcal{E}$ is a full subcategory of $[Set, Set]$ such that the family of functors $R_{A,B}(X) = A \times (B \to X)$ is in $\mathcal{E}$. Then, we have the following isomorphism natural in A, B, and X.

$$\int_F (A \to U(F)(B)) \to U(F)(X) \cong U(R^{*}_{A,B})(X)$$

Proof. See Theorem 3.1, page 8, in [JO14]. □

This isomorphism ranges over any structure upon small functors F, such as pointed functors and applicatives, and is used to change representations from ends involving functors to simpler ones. It is possible to obtain the same unary representation for profunctors [O'Cb].

Theorem 13. (Unary representation for profunctors) Consider an adjunction between profunctors $(-^{*}) \vdash U : \mathcal{E} \to \mathcal{F}$, where $\mathcal{F}$ is small and $\mathcal{E}$ is a full subcategory of $Prof(Set, Set)$, the family of profunctors $Iso_{A,B}(S,T) = (S \to A) \times (B \to T)$ gives the following isomorphism natural in A, B, and dinatural in S, T.

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

where Iso^{*} is the a free profunctor generated by Iso.

Proof. See [O'Cb]. □

Since the free monoidal profunctor exists and is in the form

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

this theorem helps us to find the unary representation for monoidal functors.

Proposition 14. The unary representation for monoidal profunctors is given by the isomorphism:

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

where P ranges over all monoidal profunctors.

Sketch. It is needed to show that

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

and then apply the Theorem 13 to get the desired result. The sum represents the disjoint union since this is in Set, which depends on an index $n \in \mathbb{N}$ allowing to use induction on it. In Set (any cartesian category), $A^0 = \{*\}$, and $A^n = A \times A \times ... \times A$ n times. The monoidal structure on profunctors with Day convolution as tensor product permits the expansion

$$\begin{aligned}
Iso^*_{A,B}(S,T) &= (J + Iso_{A,B} \star Iso^*_{A,B})(S,T) \qquad (3.1)\\
&\cong (J + Iso_{A,B} + Iso_{A,B} \star Iso_{A,B} + Iso_{A,B} \star Iso^*_{A,B})(S,T)\\
&\cong ...\\
&\cong (J + Iso_{A,B} + Iso_{A,B} \star Iso_{A,B} + Iso_{A,B} \star Iso_{A,B} \star Iso_{A,B} +\\
&... + Iso_{A,B} \star ... \star Iso_{A,B} \star Iso^*_{A,B})(S,T).
\end{aligned}$$

by a repeated substitution of $Iso^*_{A,B}$, on the righ-hand side, by $J + Iso_{A,B} \star Iso^*_{A,B}$ (recursion step). The recursion ends whenever a J is chosen to replace the $Iso^*_{A,B}$ term. Let's denote the free functor expansion up to the k-th element as $Iso^k_{A,B}$. It is obvious that the one term expansion is $Iso^0_{A,B} = J(S,T)$, giving

$$J(S,T) \cong (S \to A^0) \times (B^0 \to T) \cong (S \to \{*\}) \times (\{*\} \to T).$$

If one expands the free functor expression until the second term is given by $Iso^1_{A,B} = (J + Iso_{A,B} \star J)(S,T)$ which is equivalent the disjoint union of J and

$$(Iso_{A,B} \star J)(S,T) \cong (S \to A) \times (B \to T) \cong Iso_{A,B}(S,T).$$

Expanding the expression up to three terms

$$Iso^2_{A,B}(S,T) \cong (J + Iso_{A,B} \star (J + Iso_{A,B}))(S,T) \cong (J + Iso_{A,B} + Iso_{A,B} \star Iso_{A,B})(S,T)$$

and calculating the Day convolution,

$$\begin{aligned}
&(Iso_{A,B} \star Iso_{A,B})(S,T) \qquad (3.2)\\
&\cong \int^{XYZW} (X \to A) \times (B \to Y) \times (Z \to A) \times (B \to W)\\
&\qquad \times (S \to (X \times Z)) \times ((Y,W) \to T) \qquad (3.3)\\
&\cong \int^{XYZW} ((X,Z) \to (A,A)) \times ((B,B) \to (Y,W))\\
&\qquad \times (S \to (X \times Z)) \times ((Y,W) \to T) \qquad (3.4)\\
&\cong (S \to A^2) \times (B^2 \to T)
\end{aligned}$$

gives the desired result for $n = 2$. This last calculation means that

$$Iso^2_{A,B}(S,T) \cong \sum_{k=0}^{2} (S \to A^k) \times (B^k \to T).$$

Proof. The proof follows by induction on n. The base case is the one stated in the sketch. Let n be a natural number and suppose

$$Iso^n_{A,B}(S,T) \cong \sum_{k=0}^{n} (S \to A^k) \times (B^k \to T)$$

Expanding $F^{n+1}(S,T)$ as in (3.1) with $F = Iso_{A,B}$ (the associators will be omitted for simplicity reasons),

$$F^{n+1}(S,T) \cong (J + F + F \star F + F^2 \star F + ... + F^n \star F)(S,T)$$

which gives

$$Iso^{n+1}_{A,B}(S,T) \cong \sum_{k=0}^{n}(S \to A^k) \times (B^k \to T) + (F^n \star F)(S,T).$$

Using the same calculations as in Equation 3.2 with $F^n \star F$,

$$Iso^{n+1}_{A,B}(S,T) \cong \sum_{k=0}^{n+1}(S \to A^k) \times (B^k \to T)$$

as required. The forgetful functor can now be ignored and the desired isomorphism is obtained. □

3.4 Free monoidal profunctors in Haskell

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

```
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 _ (MPempty t) = dimap (\_ → ()) (λ() → t) mpempty
foldFreeMP h (FreeMP f g p mp) =
  dimap f g ((h p) ⋆ foldFreeMP 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 (λx → (t, x)) q
  q           ⋆ MPempty t = dimap fst (λx → (x, t)) q
  (FreeMP f g p fp) ⋆ fq     =
    dimap   (α^⊗ ∘ (f ⋆ id))
            ((g ⋆ id) ∘ α_⊗^-1)
            (consMP p (fp ⋆ fq))
```

where $\alpha^{\otimes} :: ((x, z), c) \to (z, (x, c))$ and $\alpha_{\otimes}^{-1} :: (y, (w, d)) \to ((w, y), d)$.

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 (λx → (x, ())) fst (arr id) (MPempty ())
  mp ∘ mq = toFreeMP (foldFreeMP id mp ∘ foldFreeMP id mq)
instance (MonoPro p, Arrow p) ⇒ Arrow (FreeMP p) where
  arr f = FreeMP (λx → (x, ())) fst (arr f) (MPempty ())
  (×)   = (⋆)
```

3.5 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 ⇒ CatProfunctor k p where
  catdimap :: k a b → k c d → p b c → p a d
```

A *CatProfunctor* represents a profunctor working with morphisms on an well-suited abstract category $\mathcal{C}$, 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) ⇒
  CatMonoPro k p | p → k where
  cmpunit :: k s () → k () t → p s t
  convolute :: k s (a, c)
    → k (b, d) t
    → p a b
    → p c d
    → p s t
```

It is good to notice that in the *CatMonoPro* class, the type of *cmpunit* is isomorphic to $p\ ()\ ()$, and the type of *convolute* is isomorphic to $p\ (a, c)\ (b, d)$ when k is $(\rightarrow)$. 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 = cmpunit id id
(⋆⋆) :: CatMonoPro k p ⇒ p a b
   → p c d
   → p (a, c) (b, d)
p ⋆ ⋆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 → m b}
```

having a lawful *Category* instance as follows.

```
instance Monad m ⇒ Category (Kleisli m) where
  id = Kleisli return
  (Kleisli bmc) ∘ (Kleisli amb) =
    Kleisli (λa → (amb a) ≫= bmc)
```

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

```
newtype Lift t m a b = Lift {runLift :: m a → t m b}
```

This polymorphic type represents a general version of the function *lift* used to lift monadic computations into a monad transformer [Gil]. A monad transformer is a way to stack two or more monads together in order to enable more than one effectful computation together [JD93,LHJ95]. 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. It is possible to use *catdimap* to compose effectful computations in m having its results in the monad transformer.

```
class MonadT t where
  lift :: (Monad m) ⇒ m a → t m a
instance (MonadT t, Monad m, Monad (t m)) ⇒
  CatProfunctor (Kleisli m) (Lift t m) where
    catdimap (Kleisli f) (Kleisli g) (Lift h) = Lift $
      λma → let
        k = h (ma ≫= f)
        l = lift ∘ 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 functor. Hence this instance is substantially different from the mentioned one.

To work with a *CatMonoPro* instance with respect 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 (λm → lift (m ≫= f ≫ g ()))
      convolute (Kleisli f) (Kleisli g) (Lift h) (Lift l) =
         Lift $ λms →
            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 $t\ m\ (m\ a)$. The functions $unzip' :: Functor\ f \Rightarrow f\ (a, b) \rightarrow (f\ a, f\ b)$ and $zip' :: Applicative\ g \Rightarrow (g\ a, g\ b) \rightarrow g\ (a, b)$ are helper functions as seen before. This can be fixed by building a class giving a function $comm :: (Monad\ m, Traversable\ m) \Rightarrow t\ m\ (m\ a) \rightarrow t\ m\ a$ to reorder those effects. A traversable instance is used here to provide that swap but other commutativity notion [JD93] can also be used.

```
class MonadT t ⇒ CommT t where
   comm :: (Monad m, Traversable m)
      ⇒ t m (m a)
      → 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 on mtl package [Gil]).

```
data MaybeT m a =
   MaybeT {runMaybeT :: m (Maybe a)}
data Writer w a = Writer {runWriter :: (a, w)}
instance CommT MaybeT where
   comm (MaybeT mna) =
      MaybeT (mna ≫= sequence)
```

The *CommT* instance for MaybeT pattern matches on the monad transformer MaybeT, hence exposing the value $mna :: m\ (Maybe\ (m\ a))$. The bind operator grants access to the inner monadic value of type $Maybe\ (m\ a)$. The $sequence\ (Traversable\ t, Monad\ m) \Rightarrow t\ (m\ a) \rightarrow m\ (t\ a)$ function is then used to swap the positions of *Maybe* and m.

Using the effectful monoidal profunctor *Lift t m* with respect to *Kleisli m* helps us deal with lots of monads together, keeping us able 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] → 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 concats 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)
```

The quicksort function allows for logging, enabling it to stop when encountering an invalid value while preserving a record of the algorithm's actions. It is important to notice that the instance of *CatMonoPro* facilitates the splitting and merging of sorted outcomes within the combined *MaybeT* / *Writer* context.

```
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 type class Category) function in the left and right similar as $lmap_2$ and $rmap_2$. 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′* = *Lift lift* that is just the monad transformer's *lift* in the monoidal profunctor setting.

Chapter 4

Semiarrows

In the previous chapter, we conducted an extensive study of monoidal profunctors. With Table 1.2 in mind, we notice that we can get closer to an arrow by extending the monoidal profunctor with a structure called a semiarrow. Monoidal profunctors only possess parallel composition, which may be insufficient for certain types of problems that demand more expressive structures. The semiarrow is designed to address these limitations while preserving the essence of monoidal profunctors.

It is important to note that a semiarrow is weaker than an arrow, as it lacks the *arr* and *id* structures. However, this absence of structure is not necessarily a disadvantage, as it can sometimes be desirable to model more complex computational problems, such as Moore machines.

This chapter adheres to the pattern of discussing semiarrow in a semiarrow category, which extends the well-known concepts of monoid in a monoidal category and semigroup in a semigroup category.

In this chapter, we begin by discussing and defining the concept of an arrow. Although a complete mathematical model of arrows is beyond the scope of this work and can be found in the work of Bob Atkey [Atk11], we define arrows in terms of their known laws and the corresponding Haskell typeclass. This introduction provides the necessary context for understanding the relationship between arrows and semiarrows, as well as their role in functional programming.

Next, we proceed to the study of semiarrows, including their definition, exemplification, and comparison with arrows. By examining the properties and structure of semiarrows, we aim to clarify how they extend and differ from arrows, while maintaining some of their core characteristics.

4.1 A review on arrows

An arrow is typeclass for binary type constructors [Hug05], it encodes a process with an input a and an output b that could be stateful and behave like machines.

```
class Category a ⇒ Arrow a where
  arr :: (b → c) → a b c
  first :: a b c → a (b, d) (c, d)
```

There are two methods in this class, *arr*, which transforms a plain Haskell function into a trivial process, and *first*, which gives a notion of strength to the arrow, i.e., it transforms a process into another one that has a flow of information on the first coordinate of the tuple. Note that there is a function $second :: a\ b\ c \to a\ (d, b)\ (d, c)$ that does the same for the second coordinate.

The Category constraint endows an Arrow with a well-behaved sequential composition providing notions of composition as in a Category.

When using this class as a constraint, one should enforce that $(\circ)$ must be an associative operation, and id should be an identity element of the composition operator. In the Arrow world, one often uses the operator $(\ggg) :: k\ a\ b \to k\ b\ c \to k\ a\ c$, which is the same as $\mathit{flip}\ (\circ)$.

Additionally, arrows support parallel composition as follows.

```
(×) :: a b c → a d e → a (b, d) (c, e)
(×) f g = first f ⋙ second g
```

This operator, $(\times)$, allows for the simultaneous execution of two arrow processes, combining their inputs and outputs as pairs. An arrow has to follow the following laws [LCH11].

- Left Identity: $\mathit{arr}\ \mathit{id} \ggg f = f$
- Right Identity: $f \ggg \mathit{arr}\ \mathit{id} = f$
- Associativity: $(f \ggg g) \ggg h = f \ggg (g \ggg h)$
- Composition: $\mathit{arr}\ (g \circ f) = \mathit{arr}\ f \ggg \mathit{arr}\ g$
- Extension: $\mathit{first}\ (\mathit{arr}\ f) = \mathit{arr}\ (f \times \mathit{id})$
- Functor: $\mathit{first}\ (f \ggg g) = \mathit{first}\ f \ggg \mathit{first}\ g$
- Exchange: $\mathit{first}\ f \ggg \mathit{arr}\ (\mathit{id} \times g) = \mathit{arr}\ (\mathit{id} \times g) \ggg \mathit{first}\ f$
- Unit: $\mathit{first}\ f \ggg \mathit{arr}\ \mathit{fst} = \mathit{arr}\ \mathit{fst} \ggg f$
- Association: $\mathit{first}\ (\mathit{first}\ f) \ggg \mathit{arr}\ \alpha^{\otimes} = \mathit{arr}\ \alpha^{\otimes} \ggg \mathit{first}\ f$

where $\alpha^{\otimes}\ ((a, b), c) = (a, (b, c))$.

A simple example of an Arrow is the function type $(\to)$, which has a straightforward *Category* instance.

```
instance Arrow (→) where
  arr = id
  first f = λ(a, d) → (f a, d)
```

In this instance, the *arr* function is defined trivially as the identity function *id*. The *first* function takes a function f and returns a new function that processes the first element of a tuple with f and leaves the second element unchanged. Another interesting example of an arrow is the Mealy machine [Kmeb] (sometimes called stream functions).

```
data Mealy a b = Mealy (a → (b, Mealy a b))

instance Category Mealy where
  id = Mealy $ λa → (a, id)
  (Mealy f) ∘ (Mealy g) = Mealy $ λa →
    let (b, g′) = g a
        (c, f′) = f b
    in (c, f′ ∘ g′)

instance Arrow Mealy where
  arr f = Mealy $ λa → (f a, arr f)
  first (Mealy f) = Mealy $ (a, d) →
    let (b, f′) = f a
    in ((b, d), first f′)
```

The *Mealy* type captures the notion of a machine where the output depends on its input. In contrast, the *Moore* type (extensively discussed in this work) does not share this feature.

Several extension to an Arrow are extensively discussed and proposed such as *ArrowChoice*, *ArrowSum*, and *ArrowApply* [Hug05]. But we will focus here on the extension that inspires the work of a *semiarrow*, such as *ArrowInit* and *ArrowLoop*. Those classes are discussed as a *commutative causal arrow* [LCH11] that gives a notion of commutativity to an arrow, the major example studied of a CCA is the type *Mealy*.

```
class Arrow a ⇒ ArrowLoop a where
  loop :: a (b, d) (c, d) → a b c
class ArrowLoop a ⇒ ArrowInit a where
  init :: b → a b b
```

The class *ArrowLoop* gives a notion of a feedback loop on d, giving recursion to an arrow. The class *ArrowInit* will be discussed later on the context of a *semiarrow* gives a notion of a delay to an arrow, giving stateful possibilities to computations using arrows.

```
instance ArrowLoop Mealy where
  loop (Mealy f) = Mealy $ λb → let ((c, d), f′) = f (b, d) in (c, loop f′)
instance ArrowInit Mealy where
  init b = Mealy (\_ → (b, arr id))
```

The function *loop* for the *Mealy* type takes advantage of Haskell's laziness to obtain the parameter d during its execution. This technique, known as circular programming [Bir84], enables the creation of feedback loops in the computation.

In the case of *ArrowInit*, the delay is introduced by ignoring the input parameter. Since we have an identity function *id* for *Mealy*, we can simply lift it to continue the process. This approach allows the introduction of stateful computations in the *Mealy* machine by providing an initial state and allowing for the computation to evolve based on the input and current state [LCH11].

4.2 Semiarrow categories

Due to its Day convolution tensor, we know that the profunctor category, $Prof$, is a monoidal category. Rivas and Jaskelioff's work in [RJ17] demonstrates the existence of another monoidal category atop $Prof$, which employs profunctor composition as its tensor. In this context, a monoid is referred to as a "PreArrow". We may only consider the tensor and its associator and exclude the identities, thus forming a semigroup. Consequently, we define a semigroup category, and in a similar manner, we also establish a semiarrow category.

Definition 4.1 - A *semigroup category* is a triple $(\mathcal{C}, \oplus, \alpha)$ where

- $\mathcal{C}$ is a category;
- $\oplus : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is a bifunctor;
- $\alpha_{ABC} : (A \oplus B) \oplus C \to A \oplus (B \oplus C)$ is a natural isomorphism such that the diagrams below commute.

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

In the above definition, we observe the absence of the identity object in a monoidal category. we may now define an abstract notion of semigroup.

Definition 4.2 - A *semigroup* in a semigroup category $\mathcal{C}$ is the tuple (M, m) where M is an object of $\mathcal{C}$, and $m : M \oplus M \to M$ is the multiplication morphism, satisfying

1. Associativity: $m \circ (m \oplus id) = m \circ (id \oplus m) \circ \alpha_{MMM}$

The following commuting diagram represents this law.

$$\begin{tikzcd}
M \oplus (M \oplus M) \arrow[r, "id \oplus m"] \arrow[dd, swap, "\alpha"] & M \oplus M \arrow[dr, "m"] & \\
 & & M \\
(M \oplus M) \oplus M \arrow[r, swap, "m \oplus id"] & M \oplus M \arrow[ur, swap, "m"] &
\end{tikzcd}$$

4.2.1 A *semiarrow* in a *semiarrow* category

Now, we define an interaction between a monoidal category and a semigroup category using a profunctor natural transformation. A similar structure is the nearsemiring category [RJS18] that was defined as two monoidal categories that possess a distributive law and a cancellation law. This semiarrow structure represents a variation from a nearsemiring because, in our case, we only have a semigroup structure for one of the tensors, which is subject to different laws and coherence morphisms. A duoidal category [BM11] is a structure similar to a semiarrow. However, while a duoidal category necessitates two monoidal categories, a semiarrow only requires one monoidal category paired with a semigroup category. As an illustration of this derivation, one can refer to the *WeakArrow* unit discussed in [RJ17]. This deliberate omission enables the modeling of computations that resemble arrows but don't fully align with the traditional Haskell arrow structure.

Definition 4.3 - A *semiarrow category* is a symmetric monoidal category ($\mathcal{C}$, $\otimes$, I, $\alpha_{\otimes}$, ρ, λ, γ) alongside a semigroup category ($\mathcal{C}$, $\oplus$, $\alpha_{\oplus}$) with two natural transformations, respectively:

- $\zeta : I \to I \oplus I$
- $\iota_{PQRS} : (P \oplus Q) \otimes (R \oplus S) \to (P \otimes R) \oplus (Q \otimes S)$

where ζ is the idempotence morphism and ι_{PQRS} the interchange morphism satisfying the coherence laws below.

$$\begin{tikzcd}
(P \oplus R) \otimes I \arrow[r, "\rho_{P \oplus R}"] \arrow[d, swap, "id \otimes \zeta"] & (P \oplus R) \\
(P \oplus R) \otimes (I \oplus I) \arrow[r, swap, "\iota_{PRII}"] & (P \otimes I) \oplus (R \otimes I) \arrow[u, swap, "\rho_P \oplus \rho_R"]
\end{tikzcd}$$

$$\begin{tikzcd}
 & ((P \oplus Q) \oplus T) \otimes ((R \oplus S) \oplus U) \arrow[dl, "\alpha^{\oplus}_{PQT} \otimes \alpha^{\oplus}_{RSU}"'] \arrow[dr, "\iota_{(P \oplus Q)T(R \oplus S)U}"] & \\
(P \oplus (Q \oplus T)) \otimes (R \oplus (S \oplus U)) \arrow[d, "\iota_{P(Q \oplus T)R(S \oplus U)}"'] & & ((P \oplus Q) \otimes (R \oplus S)) \oplus (T \otimes U) \arrow[d, "\iota_{PQRS} \oplus id"] \\
(P \otimes R) \oplus ((Q \oplus T) \otimes (S \oplus U)) \arrow[dr, "id \oplus \iota_{SUQT}"'] & & ((P \otimes R) \oplus (Q \otimes S)) \oplus (T \otimes U) \arrow[dl, "\alpha^{\oplus}_{(P \otimes R)(Q \otimes S)(T \otimes U)}"] \\
 & (P \otimes R) \oplus ((Q \otimes S) \oplus (T \otimes U)) &
\end{tikzcd}$$

Both diagrams indicate that the square paths and the diamond paths should commute, leading to the following equations:

$$\rho_{P \oplus R} = (\rho_P \oplus \rho_R) \circ \iota_{PRII} \circ (id \otimes \zeta) \tag{4.1}$$

$$\begin{aligned}(id \oplus \iota_{QTSU}) \circ \iota_{P(Q \oplus T)R(S \oplus U)} \circ (\alpha^{\oplus}_{PQT} \otimes \alpha^{\oplus}_{RSU}) = \\ \alpha^{\oplus}_{(P \otimes R)(Q \otimes S)(T \otimes U)} \circ (\iota_{PQRS} \oplus id) \circ \iota_{(P \oplus Q)T(R \oplus S)U}\end{aligned} \tag{4.2}$$

Your text is already fairly well-structured, but I've made some minor grammar and style improvements:

It's worth noting that the Eckmann-Hilton argument demonstrates that when you have two operations which are both unital (i.e., they possess identity elements) and they mutually distribute over one another, then these two operations are not only equal but also commutative and associative [RJ18]. This is not applicable in our context, as we do not mandate that $\oplus$ possesses an identity.

Drawing parallels with monoidal profunctors, we introduce a *semiarrow* within a *semiarrow* category. Note that we emphasize that neither ζ nor ι are mandated to be isomorphisms.

Definition 4.4 - A *semiarrow* in a semiarrow category $\mathcal{C}$ is a tuple $(P, e, m_{\otimes}, m_{\oplus})$ where $(P, e, m_{\otimes})$ is a monoid in $\mathcal{C}$ and $(P, m_{\oplus})$ is a semigroup in $\mathcal{C}$. The morphism $e : I \to P$ is the unit morphism, $m_{\otimes} : P \otimes P \to P$ is the monoidal multiplication morphism, and $m_{\oplus} : P \oplus P \to P$ is the semigroup multiplication, satisfying the laws below.

1. (Idempotence) $m_{\oplus} \circ (e \oplus e) \circ \zeta = e$

$$\begin{tikzcd}
I \oplus I \arrow[r, "e \oplus e"] & P \oplus P \arrow[d, "m_{\oplus}"] \\
I \arrow[u, "\zeta"'] \arrow[r, "e"'] & P
\end{tikzcd}$$

2. (Interchange law) $m_{\otimes} \circ (m_{\oplus} \otimes m_{\oplus}) = m_{\oplus} \circ (m_{\otimes} \oplus m_{\otimes}) \circ \iota_{PPPP}$.

$$\begin{tikzcd}
(P \oplus P) \otimes (P \oplus P) \arrow[rr, "\iota_{PPPP}"] \arrow[d, "m_{\oplus} \otimes m_{\oplus}"'] & & (P \otimes P) \oplus (P \otimes P) \arrow[d, "m_{\otimes} \oplus m_{\otimes}"] \\
P \otimes P \arrow[dr, "m_{\otimes}"'] & & P \oplus P \arrow[dl, "m_{\oplus}"] \\
 & P &
\end{tikzcd}$$

4.3 Day convolution and profunctor composition

In this section, we demonstrate that the category of set profunctors, denoted as $Prof(C, D)$, forms a *semiarrow* category when both C and D are small categories. We achieve this by considering $\star$ as the Day convolution and Bénabou composition $\cdot$ as profunctor composition.

Definition 4.5 - Given P and Q profunctors, the profunctor composition(Bénabou) is given by the following coend expression.

$$(P \cdot Q)(A, B) = \int^{Z} P(A, Z) \times Q(Z, B)$$

Ih Haskell, the profunctor composition is represented by a GADT as follows.

```
data Comp p q a b where
  Comp :: p a z → q z b → Comp p q a b
```

The type *Comp* is commonly referred to as *Procompose* in the Haskell's profunctors package. Although its implementation uses a reversed order, the semantics remains the same.

It is important to note that this construction is a profunctor, and it respects the associative natural isomorphism [RJ17].

Proposition 15. The profunctor composition forms an associative tensor product $(P \cdot Q) \cdot R \cong P \cdot (Q \cdot R)$. For any profunctors P, Q, and R.

The Haskell implementation of the associator for profunctor composition can be written as:

```
α⊕ :: Comp (Comp p q) r ⇝ Comp p (Comp q r)
α⊕ (Comp (Comp p q) r) = Comp p (Comp q r)
```

Here, $\alpha^{\oplus}$ is a function that takes a composition of three type constructors (p, q, and r) and reassociates them such that p is composed with the composition of q and r.

Having the tensor setup, we can define a specific instance of a semiarrow in a semiarrow category by establishing a lawful combination of the Day convolution and the profunctor composition. We define the binary tensors $\otimes$ and $\oplus$ to represent the Day convolution ($\star$) and the Bénabou composition ($\cdot$), respectively. This allows us to explore the interactions between these two tensor products and analyze the properties and applications of the resulting semiarrow categories.

Theorem 16. Let $\mathcal{C}$ be a small category such that $(\mathcal{C}, \otimes, I)$ is a monoidal category. Consider the monoidal category of profunctors $(Prof(\mathcal{C}, \mathcal{C}), \star, \mathrm{I})$, and the semigroup category $(Prof(\mathcal{C}, \mathcal{C}), \cdot)$. Then, we can construct an idempotence morphism $\zeta : I \to I \cdot I$ and an interchange morphism $\iota_{PQRS} : (P \cdot Q) \star (R \cdot S) \to (P \star R) \cdot (Q \star S)$ such that they satisfy the coherence laws of a semirarrow category.

Proof. Let P, Q, R, S be profunctors, and let X, Y be objects of $\mathcal{C}$. The morphism ι_{PQRS} is constructed as follows:

$$
\begin{aligned}
&((P \cdot Q) \star (R \cdot S))(X, Y) \\
&= \int^{ABCD} (P \cdot Q)(A, B) \times (R \cdot S)(C, D) \times \mathcal{C}(X, A \otimes C) \times \mathcal{C}(B \otimes D, Y) \\
&= \int^{ABCD} (\int^{Z} P(A, Z) \times Q(Z, B)) \times (\int^{W} R(C, W) \times S(W, D)) \times \\
&\qquad \mathcal{C}(X, A \otimes C) \times \mathcal{C}(B \otimes D, Y) \\
&\cong \int^{ABCDZW} P(A, Z) \times Q(Z, B) \times R(C, W) \times S(W, D) \times \\
&\qquad \mathcal{C}(X, A \otimes C) \times \mathcal{C}(B \otimes D, Y) \\
&\cong \int^{ZWABCD} P(A, Z) \times Q(Z, B) \times R(C, W) \times S(W, D) \times \\
&\qquad \mathcal{C}(X, A \otimes C) \times \mathcal{C}(B \otimes D, Y) \\
&\to \int^{ZWABCD} P(A, Z) \times Q(Z, B) \times R(C, W) \times S(W, D) \times \\
&\qquad \mathcal{C}(X, A \otimes C) \times \mathcal{C}(B \otimes D, Y) \times id_{Z \otimes W} \times id_{Z \otimes W} \\
&\cong \int^{ZWABCD} P(A, Z) \times R(C, W) \times \mathcal{C}(X, A \otimes C) \times id_{Z \otimes W} \times \\
&\qquad Q(Z, B) \times S(W, D) \times \mathcal{C}(B \otimes D, Y) \times id_{Z \otimes W} \\
&\cong \int^{ZW} (P \star R)(X, Z \otimes W) \times (Q \star S)(Z \otimes W, Y) \\
&\to \int^{U} (P \star R)(X, U) \times (Q \star S)(U, Y) \\
&= ((P \star R) \cdot (Q \star S))(X, Y)
\end{aligned}
$$

Since $\star$, and $\cdot$ are profunctors and every morphism is well defined. The coherence laws can be established using Lemmas 17, 18, and 19. The ζ morphism defined as $\zeta : X \to \int^{Z} Z \times X$, which is obtained using the definitions of J and $\cdot$. It is clear that ζ is a natural transformation that respect the coherence conditions. □

In Haskell, one can implement the interchange morphism in a very simple manner.

```
ι :: (Profunctor p, Profunctor q, Profunctor r, Profunctor s)
    ⇒ Day (Comp p q) (Comp r s) ⇝ Comp (Day p r) (Day q s)
ι (Day (Comp p q) (Comp r s) f g) =
    Comp (Day p r f (,)) (Day q s id g)
```

The profunctor instances for *Day* and *Comp* can also be implemented, allowing us to reason about the naturality of ι.

```
instance Profunctor (Day p q) where
  dimap f g (Day p q bxz ywc) =
    Day p q (bxz ∘ f) (curry (g ∘ uncurry ywc))
instance (Profunctor p, Profunctor q) ⇒ Profunctor (Comp p q) where
  dimap f g (Comp p q) = Comp (lmap f p) (rmap g q)
```

By implementing these instances, one can utilize the properties of profunctors to show that ι is indeed a natural transformation.

Lemma 17 - The ι function satifies the naturality condition.

$$\iota \circ \mathit{dimap}\ f\ g = \mathit{dimap}\ f\ g \circ \iota$$

Proof.

$$\begin{aligned}
& (\iota \circ \mathit{dimap}\ f\ g)\ (\mathit{Day}\ (\mathit{Comp}\ p\ q)\ (\mathit{Comp}\ r\ s)\ h\ k) \\
=\ & \quad \{ \text{ applying composition and dimap for Day } \} \\
& \iota\ (\mathit{Day}\ (\mathit{Comp}\ p\ q)\ (\mathit{Comp}\ r\ s)\ (h \circ f)\ (\mathit{curry}\ (g \circ \mathit{uncurry}\ k))) \\
=\ & \quad \{ \text{ applying } \iota \ \} \\
& \mathit{Comp}\ (\mathit{Day}\ p\ r\ (h \circ f)\ (,))\ (\mathit{Day}\ q\ s\ \mathit{id}\ (\mathit{curry}\ (g \circ \mathit{uncurry}\ k))) \\
=\ & \quad \{ \text{ replacing } h \circ f \text{ by } \mathit{lmap}\ f \ \} \\
& \mathit{Comp}\ (\mathit{lmap}\ f\ (\mathit{Day}\ p\ r\ h\ (,)))\ (\mathit{rmap}\ g\ (\mathit{Day}\ q\ s\ \mathit{id}\ (\mathit{curry}\ (\mathit{uncurry}\ k)))) \\
=\ & \quad \{ \text{ replacing } \mathit{curry}\ (g \circ \mathit{uncurry}\ k) \text{ by } \mathit{rmap}\ g \ \} \\
& \mathit{Comp}\ (\mathit{lmap}\ f\ (\mathit{Day}\ p\ r\ h\ (,)))\ (\mathit{rmap}\ g\ (\mathit{Day}\ q\ s\ \mathit{id}\ k)) \\
=\ & \quad \{ \text{ applying dimap for Comp } \} \\
& \mathit{dimap}\ f\ g\ (\mathit{Comp}\ (\mathit{Day}\ p\ r\ h\ (,))\ (\mathit{Day}\ q\ s\ \mathit{id}\ k)) \\
=\ & \quad \{ \text{ applying dimap for } \iota \ \} \\
& \mathit{dimap}\ f\ g\ (\iota\ (\mathit{Day}\ (\mathit{Comp}\ p\ q)\ (\mathit{Comp}\ r\ s)\ h\ k)) \\
=\ & \quad \{ \text{ applying composition } \} \\
& (\mathit{dimap}\ f\ g \circ \iota)\ (\mathit{Day}\ (\mathit{Comp}\ p\ q)\ (\mathit{Comp}\ r\ s)\ h\ k)
\end{aligned}$$

□

We should note that both the *Day* and *Comp* types, which represent tensors, are bifunctors and have been implemented in Haskell as shown below:

$$\begin{aligned}
& \mathit{bimap}^{\otimes} :: (p \rightsquigarrow r) \to (q \rightsquigarrow s) \to (\mathit{Day}\ p\ q \rightsquigarrow \mathit{Day}\ r\ s) \\
& \mathit{bimap}^{\otimes}\ \mathit{pr}\ \mathit{qs}\ (\mathit{Day}\ p\ q\ f\ h) = \mathit{Day}\ (\mathit{pr}\ p)\ (\mathit{qs}\ q)\ f\ h \\
& \mathit{bimap}^{\oplus} :: (p \rightsquigarrow r) \to (q \rightsquigarrow s) \to (\mathit{Comp}\ p\ q \rightsquigarrow \mathit{Comp}\ r\ s) \\
& \mathit{bimap}^{\oplus}\ \mathit{pr}\ \mathit{qs}\ (\mathit{Comp}\ p\ q) = \mathit{Comp}\ (\mathit{pr}\ p)\ (\mathit{qs}\ q)
\end{aligned}$$

These functions provide us with a way to reason equationally about the coherence conditions that the morphisms ζ and ι satisfies, aligning it with the definition of a *semiarrow category.*

Lemma 18 - The ζ morphism satifies the coherence conditions of a semiarrow category.

Proof.

$$
\begin{array}{ll}
 & (\mathit{bimap}\ \rho\ \rho \circ \iota \circ \mathit{bimap}\ \mathit{id}\ \zeta)\ (\mathit{Day}\ (\mathit{Comp}\ p\ r)\ (I\ b)\ f\ g) \\
= & \{\ \text{applying composition and bimap for Day}\ \} \\
 & (\mathit{bimap}\ \rho\ \rho \circ \iota)\ (\mathit{Day}\ (\mathit{Comp}\ p\ r)\ (\mathit{Comp}\ (I\ b)\ (I\ b))\ f\ g) \\
= & \{\ \text{applying } \iota\ \} \\
 & \mathit{bimap}\ \rho\ \rho\ (\mathit{Comp}\ (\mathit{Day}\ p\ (I\ b)\ f\ (,))\ (\mathit{Day}\ r\ (I\ b)\ \mathit{id}\ g)) \\
= & \{\ \text{applying bimap to Comp}\ \} \\
 & \mathit{Comp}\ (\rho\ (\mathit{Day}\ p\ (I\ b)\ f\ (,)))\ (\rho\ (\mathit{Day}\ r\ (I\ b)\ \mathit{id}\ g)) \\
= & \{\ \text{using dimap to express } \rho\ \} \\
 & \mathit{Comp}\ (\mathit{dimap}\ (\mathit{fst} \circ f)\ (\lambda d \to (d, b))\ p)\ (\mathit{dimap}\ \mathit{fst}\ (\lambda d \to g\ d\ b)\ r) \\
= & \{\ \text{expressing dimap using lmap and rmap for Comp}\ \} \\
 & \mathit{Comp}\ (\mathit{lmap}\ (\mathit{fst} \circ f)\ (\mathit{rmap}\ (\lambda d \to (d, b)\ p)))\ (\mathit{rmap}\ (\lambda d \to g\ d\ b)\ (\mathit{lmap}\ \mathit{fst}\ r)) \\
= & \{\ \text{dimap defining for Comp}\ \} \\
 & \mathit{dimap}\ (\mathit{fst} \circ f)\ (\lambda d \to g\ d\ b)\ (\mathit{Comp}\ (\mathit{rmap}\ (\lambda d \to (d, b)\ p))\ (\mathit{lmap}\ \mathit{fst}\ r)) \\
= & \{\ \text{rmap and lmap are doing nothing}\ \} \\
 & \mathit{dimap}\ (\mathit{fst} \circ f)\ (\lambda d \to g\ d\ b)\ (\mathit{Comp}\ p\ r) \\
= & \{\ \text{definition of the right associator}\ \} \\
 & \rho\ (\mathit{Day}\ (\mathit{Comp}\ p\ r)\ (I\ b)\ f\ g)
\end{array}
$$

□

LEMMA 19 - The ι morphism satifies the coherence conditions of a *semiarrow category.*

Proof. We refer to Equation 4.2 to establish that the diamond-shaped diagram commutes. Given functions f, g and profunctors p, q, r, s, t, u, we derive the following for the right-hand side:

$$
\begin{array}{ll}
 & (\alpha^{\oplus} \circ \mathit{bimap}^{\oplus}\ \iota\ \mathit{id} \circ \iota)\ (\mathit{Day}\ (\mathit{Comp}\ (\mathit{Comp}\ p\ q)\ t)\ (\mathit{Comp}\ (\mathit{Comp}\ r\ s)\ u)\ f\ g) \\
= & \{\ \text{applying } \iota\ \} \\
 & \alpha^{\oplus} \circ \mathit{bimap}^{\oplus}\ \iota\ \mathit{id})\ (\mathit{Comp}\ (\mathit{Day}\ (\mathit{Comp}\ p\ q)\ (\mathit{Comp}\ r\ s)\ f\ (,))\ (\mathit{Day}\ t\ u\ \mathit{id}\ g)) \\
= & \{\ \text{applying } \alpha^{\oplus}\ \} \\
 & \mathit{Comp}\ (\mathit{Day}\ p\ r\ f\ (,))\ (\mathit{Comp}\ (\mathit{Day}\ q\ s\ \mathit{id}\ (,))\ (\mathit{Day}\ t\ u\ \mathit{id}\ g))
\end{array}
$$

For the left-hand side:

$$
\begin{array}{ll}
 & (\mathit{bimap}^{\oplus}\ \mathit{id}\ \iota \circ \iota \circ \mathit{bimap}^{\otimes}\ \alpha^{\oplus}\ \alpha^{\oplus})\ (\mathit{Day}\ (\mathit{Comp}\ (\mathit{Comp}\ p\ q)\ t)\ (\mathit{Comp}\ (\mathit{Comp}\ r\ s)\ u)\ f\ g) \\
= & \{\ \text{applying and lifting the associators to the Day type}\ \} \\
 & (\mathit{bimap}^{\oplus}\ \mathit{id}\ \iota \circ \iota)\ (\mathit{Day}\ (\mathit{Comp}\ p\ (\mathit{Comp}\ q\ t))\ (\mathit{Comp}\ r\ (\mathit{Comp}\ s\ u))\ f\ g) \\
= & \{\ \text{applying } \mathit{bimap}^{\oplus}\ \mathit{id}\ \iota\ \} \\
 & \mathit{Comp}\ (\mathit{Day}\ p\ r\ f\ (,))\ (\mathit{Comp}\ (\mathit{Day}\ q\ s\ \mathit{id}\ (,))\ (\mathit{Day}\ t\ u\ \mathit{id}\ g))
\end{array}
$$

□

Considering that both sides are equals, we can conclude that the diamond-shaped diagram indeed commutes.

Finally, one can observe that $\star$ and $\cdot$ do fit into the definition of *semiarrow* and state the following proposition.

Corollary 1 - In the category $Prof(\mathcal{C}, \mathcal{D})$, where $\mathcal{C}$ and $\mathcal{D}$ are small categories, the monoidal category structure given by the Day convolution $\star$ and the semigroup category structure given by $\cdot$ together form a *semiarrow* category.

4.4 Semiarrow typeclass

We define a type class called *Semiarrow* for implementing the *semiarrow* notion in *semiarrow* category following the same pattern as we did for the Day convolution.

```
class MonoPro p ⇒ Semiarrow p where
  (·) :: p b c → p a b → p b c
```

this class requires the associativity law and the interchange law to hold for any four values $p_1 :: Semiarrow\ p \Rightarrow p\ a\ b$, $p_2 :: Semiarrow\ p \Rightarrow p\ c\ d$, $p_3 :: Semiarrow\ p \Rightarrow p\ e\ a$, and $p_4 :: Semiarrow\ p \Rightarrow p\ f\ c$.

$$(p_1 \star p_2) \cdot (p_3 \star p_4) = (p_1 \cdot p_3) \star (p_2 \cdot p_4)$$

This law states that when two processes are composed in parallel and then composed with another two processes also composed in parallel, it's equivalent to first compose each process with the other two processes and then compose the results in parallel.

We also have the idempotence law, which states that if we sequentially compose *mpempty* with *mempty*, we get a *mpempty* back.

$$mpempty \cdot mpempty = mpempty$$

We have defined an extension to the monoidal profunctor type class that allows a restricted notion of sequential composition, since the identity element is not defined for every type. This sequential notion will give us more power than plain monoidal profunctors, but it is still weaker than the arrow interface. Interestingly, our structure still lacks a feature equivalent to the *arr* function, which lifts any pure function into an arrow. This absence can sometimes be beneficial. The *arr* function, due to its lifting capability, embeds the entire function space into the arrow structure. This can be overly inclusive when constructing a Domain-Specific Language (DSL), especially when the goal is to maintain precise control over the language's scope and specificity.

The *Hom* monoidal profunctor and *SISO f m* when `m` is a monad can both have this notion of sequential composition. A more interesting example of *Semiarrow* is a variation of the data structure *Costate* (sometimes called Store). In functional programming, the *Costate* comonad is often defined as a pair of a state and a function that takes the state and returns a value, also a well-known comonad.

```
data Costate s a b = Costate (a → s) b
instance Profunctor (Costate s) where
  dimap f g (Costate bs c) = Costate (bs ∘ f) (g c)
instance Monoid s ⇒ MonoPro (Costate s) where
  mpempty = Costate (const mempty) ()
  (Costate f b) ⋆ (Costate g c) = Costate (λ(x, y) → f x ⊗ g y) (b, c)
instance Monoid s ⇒ Semiarrow (Costate s) where
  (Costate bs c) · (Costate as b) = Costate (λa → as a ⊗ bs b) c
```

This variation has an extra type variable s to represent some form of state, a represents a kind of input, and b is an output. The operator $\otimes$ is the monoidal multiplication of s. We can recover the original *Costate* having $a = b$ and having a comonad in s. This example will faithfully respect the interchange law precisely when m is a commutative semigroup.

LEMMA 20 - Let $(s, \otimes)$ be a commutative monoid, the *Costate s* type satisfies the interchange law.

Proof. Let

$$\begin{aligned}
&p :: \mathit{Costate}\ s\ a\ b\\
&q :: \mathit{Costate}\ s\ c\ d\\
&r :: \mathit{Costate}\ s\ e\ a\\
&s :: \mathit{Costate}\ s\ f\ c
\end{aligned}$$

we need to show that

$$(p \star q) \cdot (r \star s) = (p \cdot r) \star (q \cdot s)$$

for these four arbitrary values of the *Costate* type.

$$\begin{aligned}
&((\mathit{Costate}\ as\ b) \star (\mathit{Costate}\ cs\ d)) \cdot ((\mathit{Costate}\ es\ a) \star (\mathit{Costate}\ fs\ c))\\
=\ &\ \{ \text{ applying definitions of } \star \text{ and } \cdot \ \}\\
&\mathit{Costate}\ (\lambda(a', c') \to as\ a' \otimes cs\ c')\ (b, d) \cdot \mathit{Costate}\ (\lambda(e', f') \to es\ e' \otimes fs\ f')\ (a, c)\\
=\ &\ \{ \text{ applying definition of } \cdot \ \}\\
&\mathit{Costate}\ (\lambda(e, f) \to (\lambda(e', f') \to es\ e' \otimes fs\ f')\ (e, f) \otimes (\lambda(a', c') \to as\ a' \otimes cs\ c')\ (a, c))\ (b, d)\\
=\ &\ \{ \text{ simplifying the function inside Costate } \}\\
&\mathit{Costate}\ (\lambda(e, f) \to es\ e \otimes fs\ f \otimes as\ a \otimes cs\ c)\ (b, d)\\
=\ &\ \{ \text{ reordering } \otimes \text{ and grouping terms } \}\\
&\mathit{Costate}\ (\lambda(e, f) \to (es\ e \otimes as\ a) \otimes (fs\ f \otimes cs\ c))\ (b, d)\\
=\ &\ \{ \text{ applying definition of } \star \ \}\\
&(\mathit{Costate}\ (\lambda e \to es\ e \otimes as\ a)\ b) \star (\mathit{Costate}\ (\lambda f \to fs\ f \otimes cs\ c)\ d)\\
=\ &\ \{ \text{ applying definition of } \cdot \text{ backwards } \}\\
&(\mathit{Costate}\ as\ b) \cdot (\mathit{Costate}\ es\ a) \star (\mathit{Costate}\ cs\ d) \cdot (\mathit{Costate}\ fs\ c)\\
=\ &\ \{ \text{ applying definitions of p, q, r, and s backwards } \}\\
&(p \cdot r) \star (q \cdot s)
\end{aligned}$$

□

4.5 Moore Machines

The *Moore machine* is a fundamental structure in automata theory and can be defined as a tuple $(S, I, O, s_0, \delta, \lambda)$, 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 $\lambda : S \to O$ 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.

This difference is apparent in the Haskell ecosystem [Kmeb]. A Mealy machine is known to be an *Arrow*, but Moore machines are not because is not possible to define *arr* for them. Nevertheless, a Moore machine is a monoidal profunctor and has an associative sequential composition without having an identity and with a valid interchange law making it a significant example of a Semiarrow.

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

The type 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)
```

For the *SemiArrow* instance, we just collect the output of the rightmost machine and compose the transitions nicely.

```
instance SemiArrow Moore where
  (Moore c bm) · (Moore b am) = Moore c (λa → bm b · am a)
```

Lemma 21 - The semiarrow instance for the type *Moore* satisfies the interchange law.

Proof. Let

$$\begin{aligned}
p &= \mathit{Moore}\ b\ \mathit{am},\\
q &= \mathit{Moore}\ d\ \mathit{cm},\\
r &= \mathit{Moore}\ a\ \mathit{em},\\
s &= \mathit{Moore}\ c\ \mathit{fm}
\end{aligned}$$

we need to show that

$$(p \star q) \cdot (r \star s) = (p \cdot r) \star (q \cdot s)$$

for these four arbitrary values of the *Moore* type.

$$\begin{aligned}
&((\mathit{Moore}\ b\ \mathit{am}) \star (\mathit{Moore}\ d\ \mathit{cm})) \cdot ((\mathit{Moore}\ a\ \mathit{em}) \star (\mathit{Moore}\ c\ \mathit{fm}))\\
=\ &\quad \{\ \text{applying definitions of } \star \text{ and } \cdot\ \}\\
&\mathit{Moore}\ (b, d)\ (\lambda(a, c) \to \mathit{am}\ a \cdot \mathit{cm}\ c) \cdot \mathit{Moore}\ (a, c)\ (\lambda(e, f) \to \mathit{em}\ e \cdot \mathit{fm}\ f)\\
=\ &\quad \{\ \text{applying definition of } \cdot\ \}\\
&\mathit{Moore}\ (b, d)\ ((e, f) \to (\mathit{am}\ a \cdot \mathit{cm}\ c)\ (a, c) \cdot (\mathit{em}\ e \cdot \mathit{fm}\ f)\ (e, f))\\
=\ &\quad \{\ \text{simplifying the function inside Moore}\ \}\\
&\mathit{Moore}\ (b, d)\ ((e, f) \to (\mathit{am}\ a \star \mathit{cm}\ c) \cdot (\mathit{em}\ e \star \mathit{fm}\ f))\\
=\ &\quad \{\ \text{reordering } \cdot \text{ and grouping terms}\ \}\\
&\mathit{Moore}\ (b, d)\ (\lambda(e, f) \to (\mathit{am}\ a \cdot \mathit{em}\ e) \star (\mathit{cm}\ c \cdot \mathit{fm}\ f))\\
=\ &\quad \{\ \text{applying definition of } \star\ \}\\
&\mathit{Moore}\ b\ (\lambda e \to \mathit{am}\ a \cdot \mathit{em}\ e) \star \mathit{Moore}\ d\ (\lambda f \to \mathit{cm}\ c \cdot \mathit{fm}\ f)\\
=\ &\quad \{\ \text{applying definition of } \cdot \text{ backwards}\ \}\\
&(\mathit{Moore}\ b\ \mathit{am}) \cdot (\mathit{Moore}\ a\ \mathit{em}) \star (\mathit{Moore}\ d\ \mathit{cm}) \cdot (\mathit{Moore}\ c\ \mathit{fm})\\
=\ &\quad \{\ \text{applying definitions of p, q, r, and s backwards}\ \}\\
&(p \cdot r) \star (q \cdot s)
\end{aligned}$$

□

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

```
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. This is an extension of the type *Costate* defined in the previous section.

```
buildMoore :: Applicative m ⇒ MooreCoalg s a b → s → Moore a b
buildMoore mc@(MooreCoalg out next) s =
  Moore (out s) (fmap (buildMoore mc) ∘ 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 Int a Int
countMoore = buildMoore (MooreCoalg id (λs _ → s + 1)) 0
```

The above machine ignores every input and returns an updated state by summing 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 $[a] \to [b]$. 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 list as follows.

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

Now we can consume the *countMoore* machine and obtain the desired accumulated outputs.

The *SemiArrow* interface allows the creation of composable Moore machines, something which cannot be done with the *Arrow* type class. However, this fact comes with a price: each time a sequential composition is used with a Moore machine adds an *extra delay* due to the necessity for an explicit *initial output*. In the next section, we discuss the *SemiArrow* interface of Moore machines and how they relate to synchronous data-flow programming.

Chapter 5

Applications

In this chapter, we list some applications that is derived using a monoidal profunctor and a semiarrow to reason about functional programmining code. The applications are type-safe lists, monoidal profunctor optics, an tiny interpreter for a process calculus, and connections of Moore machines with left folds and scans.

5.1 Applications of Monoidal Profunctors

5.1.1 Type-safe lists

The first application for the monoidal profunctor is to handle tuples instead of lists which give type-safety concerning its size. This techinique is found in the packages *opaleye* [Ella] and $product - profunctors$ [Ellb]. This technique is not new and it is heavily used in opaleye package. However it shows the power behind the monoidal profunctor interface. The technique consists in applying the *Default* typeclass to generate such lists.

```
class Default p a b where
  def :: p a b
```

The *Default* typeclass picks a distinguished computation of the form $p\ a\ b$ representing a lifted function based on the structure of p.

Given two default computations, $p\ a\ b$ and $p\ c\ d$, it is possible to overload *def* with the help of the GHC extension *MultiParamTypeClasses* to derive an instance for
$p\ (a, c)\ (b, d)$.

```
instance (MonoPro p, Default p a b, Default p c d) ⇒
            Default p (a, c) (b, d) where
  def = def ⋆ def
```

If one has more than two computations, they can be handled by overloading with the monoidal profunctor product. Flattening functions, like *flat3i*, *flat3l*, *flat4i*, *flat4l*, and so on, can address tuple reparenthesizations. The number in the function name indicates the number of coordinates in the tuple. Those boilerplate codes can also be derived with the help of generics, template Haskell and quasi-quotations.

```
instance (MonoPro p,
            Default p a b,
            Default p c d,
            Default p e f) ⇒
            Default p (a, c, e) (b, d, f) where
  def = dimap flat3i flat3l (def ⋆ def ⋆ def)
```

```
instance (MonoPro p,
          Default p a b,
          Default p c d,
          Default p e f,
          Default p j k) ⇒
          Default p (a, c, e, j) (b, d, f, k) where
  def = dimap flat4i flat4l (def ⋆ def ⋆ def ⋆ def)
```

For example, using this technique, the functions *replicate* [Ellb], *iterate*, and *zipWith* can have type-safe versions.

A *Replicator* is a type that enables the type-safe version of *replicate*.

```
newtype Replicator r f a b = Replicator (r → f b)
```

A profunctor instance for *Replicator r f*, noting that a is a phantom type argument since this, amounts to a functor applied to a type b. The phantom type argument a is needed to match the desired kind.

```
instance Functor f ⇒ Profunctor (Replicator r f) where
  dimap _ h (Replicator f) =
    Replicator ((fmap ∘ fmap) h f)
```

Whenever $r{\sim}f\ b$, one can choose *Replicator id* as its default value.

```
instance Applicative f ⇒
          Default (Replicator (f b) f) b b where
  def = Replicator id
```

A *Replicator* is a *MonoPro* when f is applicative; its monoidal profunctor product is just zip.

The function *replicateT* does the trick. It uses *def′*, which is deconstructed to *Replicator f*, to overload the monoidal product basing on a type given in runtime.

```
replicateT :: Default (Replicator r f) b b ⇒ r → f b
replicateT = f
  where Replicator f = def′
          def′ :: Default p a a ⇒ p a a
          def′ = def
```

For example, we may get three integers from the command line by

```
replicateT (readLn :: IO Int) :: IO (Int, Int, Int)
```

The number of integers varies with the type. In the case of iterators, it is important to note that this implementation differs slightly from the original *iterate* from *Data.List*, since the first element here is ignored.

```
data It a z b = It ((a → a) → a → (a, b))
```

An *It a* is a profunctor on b and has a trivial instance omitted here. A monoidal profunctor instance for *It a* works with the return type a, the first component of the tuple, acting as a state.

```
instance MonoPro (It a) where
  mpempty = It (λh x → (h x, ()))
```

```
    It f ⋆ It g = It $ λh x →
      let (y, b) = f h x
          (z, c) = g h y
      in (z, (b, c))
```

A default computation for *It* is one step iteration, and this will keep the iteration happening when computed the monoidal product.

```
  instance Default (It a) z a where
    def = It (λf a → (f a, f a))
```

Using the overloaded *def* again and deconstructing its type with the help of *itExplicit*, the function *iterT* is the type-safe version of *iterate*.

```
  iterT :: Default (It a) b b ⇒
           (a → a) → a → b
  iterT = itExplicit def
    where
      itExplicit :: It a b b → (a → a) → a → b
      itExplicit (It h) f a = snd (h f a)
```

Evaluating

```
  iterT (2∗) 3 :: (Integer, Integer, Integer, Integer),
```

gives (6, 12, 24, 48) which is exactly four iterations.

It is also possible to construct a type-safe version of the function *zipWith* relying on the type *Grate*. This example shows a connection with this technique and optics (more details in the next section).

```
  data Grate a b s t = Grate (((s → a) → b) → t)
```

The datatype *Grate a b* is a profunctor on *s* and *t* and relies on a continuation-like style.

```
  instance Profunctor (Grate x y) where
    dimap f g (Grate h) =
      Grate (λk → g (h (λt → k (t ∘ f))))
```

Its monoidal profunctor product instance unzips the input function and passes it to the monoidal product of *f* and *g*.

```
  instance MonoPro (Grate x y) where
    mpempty = Grate $ λ_ → ()
    Grate f ⋆ Grate g =
      Grate (λh → (f ⋆ g) (k (unzip′ (Aux h))))
      where
        k = unAux ⋆ unAux
```

The type *Aux* is just a helper type that makes the definition of ⋆ easier.

```
  data Aux x y a = Aux {unAux :: (a → x) → y}
```

Applying *id* to an input function is the default computation for a *Grate* whenever $s{\sim}a$ and $t{\sim}b$.

```
instance Default (Grate a b) a b where
  def = Grate (λf → f id)
```

The same pattern of *Replicator* and *It* also occurs with *Grate*.

```
grateT :: Default (Grate a b) s t ⇒ (((s → a) → b) → t)
grateT = grateExplicit def
  where
    grateExplicit :: Grate a b s t → (((s → a) → b) → t)
    grateExplicit (Grate g) = λf → g f
```

A type-safe *zipWith*, called *zipWithT*, can be constructed using the *grateT*.

```
zipWithT :: (Int → Int → Int)
          → (Int, Int, Int)
          → (Int, Int, Int)
          → (Int, Int, Int)
zipWithT op s₁ s₂ = grateT (λf → op (f s₁) (f s₂))
```

This connection with optics has an obvious limitation that it can only generate functions with explicit types like *zipWithT* to avoid ambiguous types. It is interesting to note that the same construction can be used to create type-safe traversals (which is also an optic). One needs to consider the above type *Traverse*, and *Traverse* ($) as default computation.

```
data Traverse f r s a b = Traverse ((r → f s) → a → f b)
```

5.1.2 Monoidal Profunctor Optics

Data accessors are an essential part of functional programming. They allow reading and writing a whole data structure or parts of it [PGW17]. 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, the notion of modular (composable) data acessors [PGW17] helps to tackle this problem with the help of some tools 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.

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 $get :: s \to a$. Whereas, if one focus in a "big" structure `s` providing a value of `b` (part of `f`) it can turn in another "big" structure "t" (this may not change, and the data can still be `s`), a good manner to represent that is via the function $set :: s \to b \to t$.

These functions can be combined using a binary type constructor p and a restriction r, resulting in the type $Optic\ r\ s\ t\ a\ b = \forall p.r\ p \Rightarrow p\ a\ b \to p\ s\ t$. For instance, if r is *Strong*, then $Strong\ p \Rightarrow p\ a\ b \to p\ s\ t$ is a lens. If one substitutes p with the contravariant hom-functor (which is *Strong*, and also known as the type constructor $Forget :: (a \to r) \to Forget\ r\ a\ b$ in Haskell), and uses $first' :: p\ a\ b \to p\ (a, x)\ (b, x)$ as a lens, it becomes apparent that this provides the projection of the first component from a product type, producing then the function $get :: (a, x) \to a$ in this context.

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

Those two optics have the following types.

$$\begin{aligned}
&\textbf{type}\ \mathit{Iso}\ s\ t\ a\ b = \forall p.\mathit{Profunctor}\ p \Rightarrow p\ a\ b \rightarrow p\ s\ t\\
&\textbf{type}\ \mathit{Monocle}\ s\ t\ a\ b = \forall p.\mathit{MonoPro}\ p \Rightarrow p\ a\ b \rightarrow p\ s\ t
\end{aligned}$$

Every `Monocle` is an `Iso`. The latter provides us the necessary tool for handling isomorphisms between types.

$$\begin{aligned}
&\mathit{swap} :: \mathit{Profunctor}\ p \Rightarrow p\ (b, a)\ (c, d) \rightarrow p\ (a, b)\ (d, c)\\
&\mathit{swap} = \mathit{dimap}\ \mathit{sw}\ \mathit{sw}\\
&\mathit{associate} :: \mathit{Profunctor}\ p \Rightarrow\\
&\quad p\ ((w, y), d)\ ((x, z), c) \rightarrow p\ (y, (w, d))\ (z, (x, c))\\
&\mathit{associate} = \mathit{dimap}\ \mathit{associnv}\ \alpha^{\otimes}
\end{aligned}$$

The `swap` iso represents the isomorphism $A \times B \cong B \times A$. It takes a profunctor and reverses the order of all product types involved, and `associate` iso represents an associative rule of product types. The units `()` can be treated as well but will be omitted.

A monocle locates every position from a product (tuple) type (which can be generalized to a finite vector [JO14]).

$$\begin{aligned}
&\mathit{each2} :: \mathit{MonoPro}\ p \Rightarrow p\ a\ b \rightarrow p\ (a, a)\ (b, b)\\
&\mathit{each2}\ p = p \star p\\
&\mathit{each3} :: \mathit{MonoPro}\ p \Rightarrow p\ a\ b \rightarrow p\ (a, a, a)\ (b, b, b)\\
&\mathit{each3}\ p = \mathit{dimap}\ \mathit{flat3i}\ \mathit{flat3l}\ (p \star p \star p)\\
&\mathit{each4} :: \mathit{MonoPro}\ p \Rightarrow p\ a\ b \rightarrow p\ (a, a, a, a)\ (b, b, b, b)\\
&\mathit{each4}\ p = \mathit{dimap}\ \mathit{flat4i}\ \mathit{flat4l}\ (p \star p \star p \star p)
\end{aligned}$$

As one can observe, `each2` deals with parallel composition with the argument `p` with itself using the `monoPro` interface. The focus is on tuples of size 2. The monocles `each3` and `each4` deal with tuples of size 3 and 4 and depends on the flattening functions defined earlier.

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

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

This action tells that given a `Monocle` (location) one can monoidally collect many parts `a` from the big structure `s` (in this case, tuples). It is nice to remember that `Forget` is just the contravariant hom-functor, an instance of a `SISO`, when $f = \mathit{Id}$, and $g = \mathit{Const}\ r$ the constant applicative functor, whenever `r` (the covariant part of the `SISO`) is a monoid. For example,

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

behaves in the same way as the function `fold` do with lists, its evaluation on the value `("AA","BB","CC")` gives `"AABBCC"` as expected. A Monocle called `foldMapOf` can also behave like its list counterpart `foldMap`,

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

locating all elements of a 3-element tuple gives

```
foldMapOf each3 :: Monoid r ⇒ (a → r) → (a, a, a) → r
```

as mentioned.

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

```
convolute :: (Applicative g, Functor f) ⇒ Monocle s t a b
  → (f a → g b)
  → f s → g t
convolute monocle f = unSISO (monocle (SISO f))
```

following the same pattern as in `foldMapOf` changing the `Forget` by a `SISO`.

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

```
traverseOf :: Applicative g
  ⇒ Monocle s t a b
  → (Id a → g b)
  → (Id s → g t)
traverseOf monocle = convolute monocle
```

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

```
class Profunctor p ⇒ Closed p where
  closed :: p a b → p (x → a) (x → b)
zipFWithOf :: Functor f
  ⇒ Monocle s t a b
  → (f a → Id b)
  → (f s → Id t)
zipFWithOf monocle = convolute monocle
```

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.

5.1.3 Process calculi using the Free Monoidal Profunctor

We now will build a simple interpreter example for a process calculi that is a variant of pi-calculus. This example shows a good example of how a free monoidal profunctor works. First, we will build a syntax tree that contains syntax for sending a message through a channel, receiving a message, and creating a new channel. Then the parallel composition will be achieved using the free structure because the syntax tree type we present is not a monoidal profunctor but only a plain profunctor.

```
data Process x a b where
  Send :: String → (a → x) → Process x a b → Process x a b
  Recv
    :: String
    → (x → Process x a b)
    → Process x a b
  New :: String → Process x a b → Process x a b
  Output :: String → (x → b) → Process x a b
```

The type `Process` presents four primary constructors, three mentioned before, and an extra one called output to represent an output for the process. The communication needs to be invariant, i.e., sending and receiving should share the same type `x`. It is important to note that the type variables `a` and `b` represent inputs that can be consumed and encoded to be sent and output to be decoded, producing a value of a desired type, respectively. To send a value, one needs to provide a way to decode the input `a` into the communication type `x`, and then behave as a continuation given by its third field. The receiving will bind what it gets from the channel in the input of the function provided by the type `(x -> Process x a b)` and then continue normally. The output constructor needs a way to decode the communication type `x` into the output `b`. Finally, it is important to note that the `New` construct is straightforward. Each constructor has a `String` field to keep a channel name.

instance *Profunctor* (*Process x*) **where**
  $dimap\ f\ g\ (Send\ channel\ h\ \pi) = Send\ channel\ (h \circ f)\ (dimap\ f\ g\ \pi)$
  $dimap\ f\ g\ (Recv\ channel\ k) = Recv\ channel\ (dimap\ f\ g \circ k)$
  $dimap\ f\ g\ (New\ channel\ \pi) = New\ channel\ (dimap\ f\ g\ \pi)$
  $dimap\ f\ g\ (Output\ s\ h) = Output\ s\ (g \circ h)$

Interestingly, `Process x` is not a monoidal profunctor if we do not make assumptions about `x` (`x` being a monoid, for example). We want to keep the communication type `x` as general as possible to allow a broader communication range. Note that a data constructor

```
Parallel :: Process x a b -> Process x a b -> Process x a b
```

could be used, but it will only help a little and provide two processes with different input and output types. Any other variant of this sort can invalidate the `Profunctor` instance. Without the monoidal profunctor interface, one cannot use parallelism, but by using it freely, we can recover it. The strategy is to interpret the syntax into a valid monoidal profunctor that allows us to model a notion of parallelism and use `foldFreeMP` to consume the free monoidal profunctor structure. The free monoidal profunctor will carefully pack the process syntax (the `consMP` function will pack them) and provide a valid monoidal profunctor structure.

5.2 Applications of Semiarrows

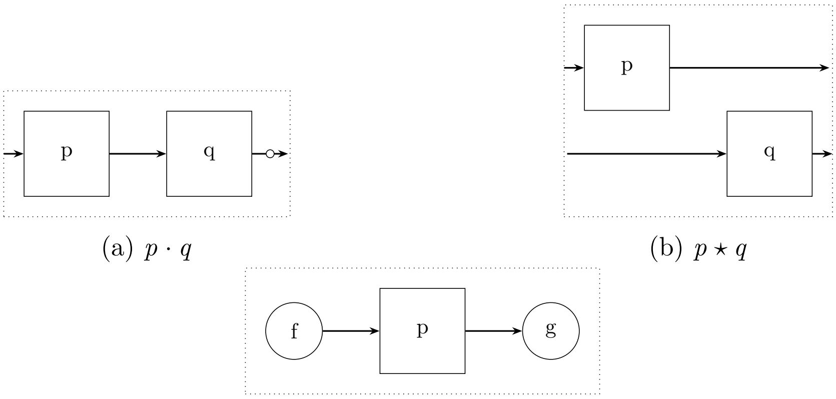
5.2.1 Synchronous data-flow programming and delays

In synchronous data-flow programming [HCRP91], components are composed to create a larger system, with data flowing through components in a well-defined manner. The *SemiArrow* interface, along with the Moore machine model, provides the necessary interface to create, manage, and compose stateful components in this context. The main advantage of using a semiarrow instead of a causal commutative arrow [LCH11] is that in a CCA, one needs to introduce delays manually, making complex synchronous programs difficult to reason about. In contrast, in a semiarrow (that fails to be an arrow, like Moore machines), the delay is accounted for automatically when the sequential composition is used.

Let's use Moore machines to explain the semiarrow (that is not an arrow) behavior. When we compose two Moore machines, say $m = Moore\ c\ bm$, $n = Moore\ b\ am$, using the sequential composition operator $\cdot$, we create a new Moore machine $Moore\ c\ (\lambda a \to bm\ b \cdot am\ a)$ that first applies the first machine's transition $am\ a$ followed by the second machine's transition $bm\ b$. However, the output of the first machine is based on its current state before transitioning. The composed Moore machines on a list of inputs, the output of the first machine will be delayed by one step compared to the output of the second machine.

In the *runMoore* function (last section), it is possible to observe the delay effect. The transition function f applied to the input a produces the output b before transitioning to the next state. As a result, the output of the first Moore machine in the composition lags by one step compared to the second machine. This delay is a natural consequence of the composition of stateful components, as the first machine's output depends sequentially on its state before transition occurs.

Parallel composition, encoded in the monoidal profunctor interface, does not introduce any delay, thus simulating the correct behavior when two stateful programs are composed in this manner. Similarly, when using the profunctor's *dimap*, which lifts pure computations, no delay is introduced as well. In this work, both interfaces will be referred to as *0-delay operations* (in the literature combinatorial), while sequential composition will be called a *1-delay operation.* These terminologies account for the number of delays introduced automatically without requiring any manual intervention, as in the case of CCAs [LCH11].



(a) $p \cdot q$

(b) $p \star q$

(c) *dimap f g p*

Figure 5.1: The *semiarrow* interface

In Figure 5.1, the discussion above is captured in a graphical representation. The sequential composition is illustrated with a small hollow circle, indicating a delay, as shown in Figure 5.1a. The profunctor interface, as well as the monoidal profunctor interface, can be endowed with effects using *effectful monoidal profunctors*, as discussed in Section 3.5. This allows for handling effects, such as errors or logging, when splitting occurs during parallel composition or when merging is necessary. Using an effectful monoidal profunctor maintains the same 0-delay behavior as before, making it a viable alternative to the *Profunctor* or *MonoPro* interfaces.

It is indeed possible to automatically add delays in any process using the following typeclass Delay. This can be seen as a renaming of the ArrowInit class [LCH11] without the ArrowLoop restriction to maintain the semiarrowname semantics.

```
class Delay p where
  delay :: b → p b b
instance Delay Moore where
  delay b = Moore b delay
```

In the code above, a new initial value is added to the process flow, which delays the entire machine process by one unit of time.

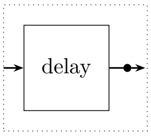


Figure 5.2: *delay*

The Figure 5.2 illustrates the *delay* process, with the filled circle representing a manually introduced delay. This graphical representation highlights the delay's role in the process flow and helps convey its importance in managing the reasoning about a synchronous process.

However, delays introduced by using the *delay* process can be challenging to reason about, especially when dealing with complex systems. For instance, if one has two stateful binary processes, i.e., a process with a tuple in its input and a single type in its output, and one of them needs to wait for the execution of the other, it is possible to use delay in conjunction with the 0-delay operation *dimap* to achieve the desired behavior. However, the composition of structures can be generalized to accommodate such delays more effectively.

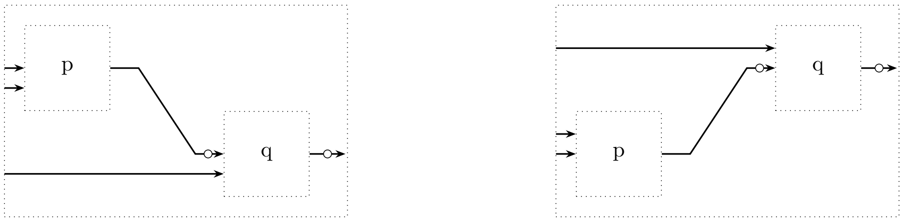
```
class SemiArrow p ⇒ GSemiArrow p where
  (∘1) :: p (c, d) e → p (a, b) c → p ((a, b), d) e
  (∘2) :: p (c, d) e → p (a, b) d → p (c, (a, b)) e
instance GSemiArrow Moore where
  Moore e cdx ∘1 Moore c abx =
    Moore e (λ((a, b), d) → cdx (c, d) ∘1 abx (a, b))
  Moore e cdx ∘2 Moore d abx =
    Moore e (λ(c, (a, b)) → cdx (c, d) ∘2 abx (a, b))
```

The *GSemiArrow* typeclass is an extension of the SemiArrow typeclass, introducing two new composition operators, $(\circ_1)$ and $(\circ_2)$. These operators allow for more complex compositions of processes with an intrisic notion of additional delays. The primary purpose of the *GSemiArrow* typeclass is to handle effectively the composition with worrying about delays.

In the provided instance for the Moore type, both $(\circ_1)$ and $(\circ_2)$ operators are defined, showcasing the ability to compose Moore machines with additional delay handling.

The $(\circ_1)$ operator is used for sequential composition of two machines or processes, where the first process takes a tuple (c, d) as input and produces an output of type e, and the second process takes a tuple (a, b) as input and produces an output of type c. The resulting composition takes a tuple $((a, b), d)$ as input and produces an output of type e. This operator effectively captures the delays between the first and second binary processes. In essence, $(\circ_1)$computes a binary process, takes its result, and composes it with the first coordinate of the second binary process, while handling the respective delay.

On the other hand, $(\circ_2)$ deals with a similar behavior but focuses on the second coordinate of the tuple in the second process. This approach rules out the necessity of automatic delays and allows for greater flexibility and scalability when handling more complex systems with multiple delays.



(a) $p \circ_1 q$

(b) $p \circ_2 q$

Figure 5.3: The *GSemiArrow* interface

The first figure represents the composition $p \circ_1 q$. In this figure, two parallel input arrows are connected to the p block representing its input. The boxes are marked with a dotted outline to indicate that they correspond to a *GSemiArrow*. The output arrow of p is connected to the q block through a hollow circle, indicating a delay. The process q also has an additional parallel input arrow coming from the left corner of the dotted box representing its second input. The output of q also has a delay, the second one, as one can observe hollow circle. It is interesting to note that one should not expect associativity in these compositions, i.e., both figures can give different results due to the different connections and the delays present.

As an example, consider the expression $\mathit{runMoore}\ m\ [1,2,3,4,5,6,7,8]$ where $m = (\mathit{plusM} \circ_1 \mathit{plusM}) \cdot (\mathit{sumM} \star \mathit{sumM} \star \mathit{sumM}) \cdot \mathit{splitter3}$. The three involved process in this expression are given as follows.

```
splitter3 :: MooreI a (([a], [a]), [a])
splitter3 =
  createMachine (MooreCA fst
    (λ(((s, s′), s″), cter) a →
      case cter ‘mod‘ 3 of
        0 → (((a : s, s′), s″), cter + 1)
        1 → (((s, a : s′), s″), cter + 1)
        2 → (((s, s′), a : s″), cter + 1))) ((([ ], [ ]), [ ]), 0)
sumM :: MooreI [Int] Int
sumM = createMachine (MooreCA id (λs as → s + sum as)) 0
plusM :: MooreI (Int, Int) Int
plusM = createMachine (MooreCA id (λs (x, y) → s + x + y)) 0
```

The composed machine m provides us with an ideal scenario for analyzing the *GSemiArrow* interface and its handling of two delays. First, the machine splits, with a delay having its inital input as $(([\,],[\,]),[\,])$, the input into three lists based on the state's counter *cter*. The first list contains inputs received when the counter is a multiple of 3, while the second and third lists correspond to inputs with remainders of 1 and 2 modulo 3, respectively.

Next, we sequentially compose the machine with three parallel sum accumulators, generating a delay due to the initial value of $((0,0),0)$. Finally, we sequentially compose the machine again with two *plusM* accumulators using the *GSemiArrow* interface, resulting in two delays with an initial value of 0. The *plusM* accumulators sum the results of the first two *sumM* machines and store their state. This result, with a delay, is then fed back into the *plusM* accumulators to be added to the output of the third *sumM* machine. In the entire machine m, there are a total of four delays.

Table 5.1: Behavior of the analyzed expression over time

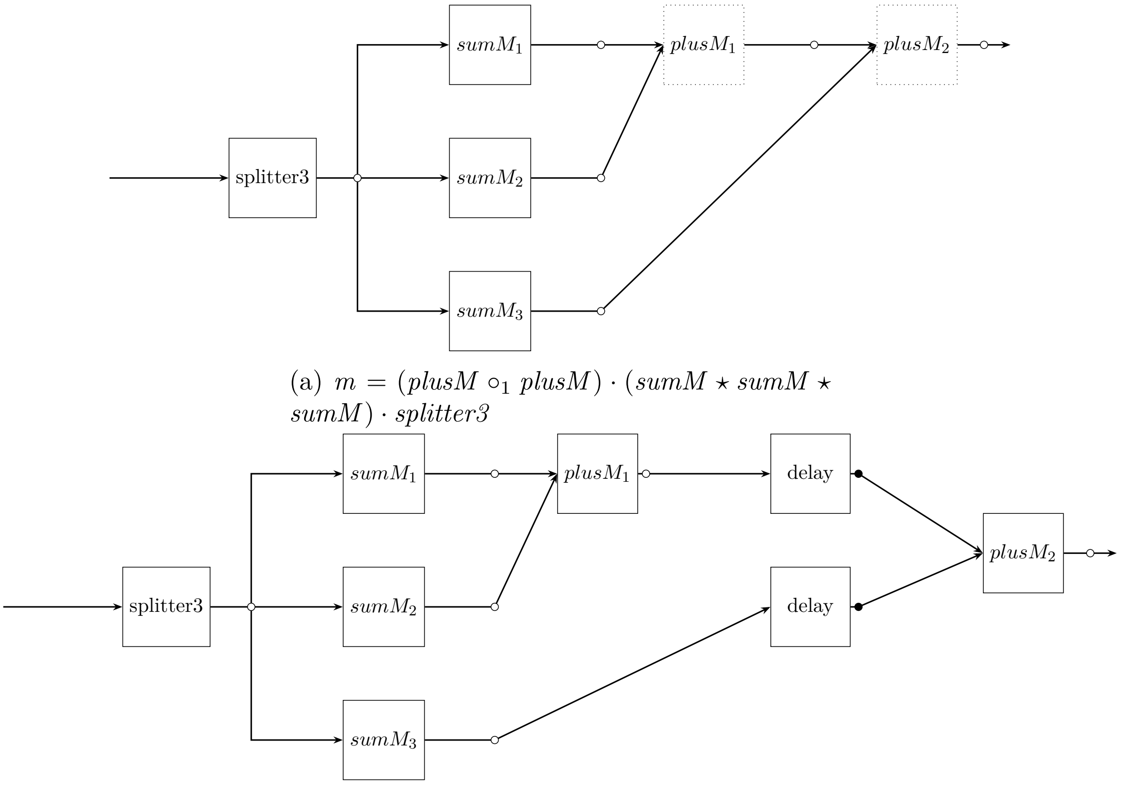
t	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_9
input	1	2	3	4	5	6	7	8	9
splitter3	[][][]	[1][][]	[1][2][]	[1][2][3]	[4, 1][2][3]	[4, 1][5, 2][3]	[4, 1][5, 2][6, 3]	[7, 4, 1][5, 2][6, 3]	[7, 4, 1][8, 5, 2][6, 3]
sumM_1	0	0	1	2	3	8	13	18	30
sumM_2	0	0	0	2	4	6	13	20	27
sumM_3	0	0	0	0	3	6	9	18	27
plusM_1	0	0	0	1	5	12	26	52	90
plusM_2	0	0	0	0	1	9	27	62	132

Table 5.1 presents the computations occurring over time, starting at t_0. For the sake of aesthetics and clarity, we have removed the tuples from the third row and replicated both *sumM* and *plusM* rows. It is important to note that the final *plusM* takes four time delays before it begins processing, effectively capturing the desired behavior. This table provides a clear visualization of the computations' evolution and highlights the inherent delays in the system, emphasizing the importance of understanding how the components of this expression interact with each other over time. The last line gives the exact answer of the whole process. The subscripts used with *sumM* and *plusM* specify the distinct instances of function usage, enhancing the clarity of the table's interpretation.

An equivalent expression without using *GSemiArrow* is

$$\mathit{runMoore}\ n\ [1,2,3,4,5,6,7,8]$$

where $n = \mathit{plusM} \cdot ((\mathit{delay}\ 0 \cdot \mathit{plusM}) \star \mathit{delay}\ 0) \cdot (\mathit{sumM} \star \mathit{sumM} \star \mathit{sumM}) \cdot \mathit{splitter3}$. The machine n captures the same behavior as the *GSemiArrow* composition, but it is harder to reason about and introduces an extra delay (5 delays instead of 4 when using GSemiArrow). Using the *GSemiArrow* extension to a semiarrow can be crucial in systems where timing and performance are essential. It is worth noting that $n = \mathit{delay}\ 0 \cdot m$.



(a) $m = (\mathit{plusM} \circ_1 \mathit{plusM}) \cdot (\mathit{sumM} \star \mathit{sumM} \star \mathit{sumM}) \cdot \mathit{splitter3}$

(b) $n = \mathit{plusM} \cdot ((\mathit{delay}\ 0 \cdot \mathit{plusM}) \star \mathit{delay}\ 0) \cdot (\mathit{sumM} \star \mathit{sumM} \star \mathit{sumM}) \cdot \mathit{splitter3}$

Figure 5.4: Graphical example

The Figure 5.4 illustrates both discussed machines, with the first one utilizing *GSemiArrow* and the second one relying solely on manual delays. To accurately count the delays, we consider parallel delay circles as one unit. As a result, the first figure has four delays, while the second one has five. This figure also demonstrates that using *GSemiArrow* offers a more convenient way to manage delays.

5.2.2 Moore machines, folds and scans

We start this subsection by analyzing the following simple example of using a Moore machine by counting how many elements we read from the input. It's worth noting that the proofs included in this section are formally presented in a separate Agda file, which can be downloaded from the following GitHub repository: `https://github.com/romefeller/monopro-agda-formal`.

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

By running the *countMoore* machine with five unit inputs (can be anything, it will be ignored), the return is a list from 0 to 6, 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* from *Data.List*.

$$\begin{aligned}&\mathit{runMoore} \circ \mathit{buildMoore}\ (\mathit{MooreCoalg}\ \mathit{id}\ (\lambda s\ \_ \rightarrow s + 1))\\ &\quad = \mathit{scanl}\ (\lambda s\ \_ \rightarrow s + 1)\end{aligned}$$

That can be generalized as the following rule.

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

If one take a peek on the *scanl* one can notice that this function builds and runs a Moore machine at the same time. If the single parameter b was a function, the first two parameters types would match *MooreCoalg*, and the return would be the same as *runMoore*.

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

Using a *Moore* abstraction allows us to construct more complex ways to do accumulated folds that is not possible using scanl only.

In this section we explore the connection between Moore machines, left scans and left folds. First, let us take a look on *foldl*, and *runMooref*.

$$\begin{aligned}&\mathit{foldl} :: (s \rightarrow a \rightarrow s) \rightarrow s \rightarrow [a] \rightarrow s\\ &\mathit{foldl}\ \_\ s\ [\,] = s\\ &\mathit{foldl}\ f\ s\ (a : \mathit{as}) = \mathit{foldl}\ f\ (f\ s\ a)\ \mathit{as}\\ &\mathit{runMooref} :: \mathit{Moore}\ a\ b \rightarrow [a] \rightarrow b\\ &\mathit{runMooref}\ (\mathit{Moore}\ b\ f)\ [\,] = b\\ &\mathit{runMooref}\ (\mathit{Moore}\ b\ f)\ (a : \mathit{as}) = \mathit{runMooref}\ (f\ a)\ \mathit{as}\end{aligned}$$

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

$$\begin{aligned}&\mathit{ofoldl} :: (s \rightarrow a \rightarrow s) \rightarrow s \rightarrow (s \rightarrow b) \rightarrow [a] \rightarrow b\\ &\mathit{ofoldl}\ f\ s\ \mathit{sb}\ \mathit{as} = \mathit{sb}\ (\mathit{foldl}\ f\ s\ \mathit{as})\end{aligned}$$

Analyzing the parameters of *ofoldl*, we have an initial state s, a transition $s \rightarrow a \rightarrow s$, and an output function $s \rightarrow b$. 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 $[a] \rightarrow b$, is the same as

$$\textbf{type}\ \mathit{Fold}\ a\ b = \mathit{SISO}\ [\,]\ \mathit{Identity}\ a\ b$$

We now can rewrite the above function as follows.

$$\begin{aligned}&\mathit{mfoldl} :: \mathit{Moore}\ a\ b \rightarrow \mathit{Fold}\ a\ b\\ &\mathit{mfoldl}\ m = \mathit{SISO}\ \$\ \lambda \mathit{as} \rightarrow \mathit{Identity}\ (\mathit{runMooref}\ m\ \mathit{as})\end{aligned}$$

It is nice to remember that the inital state is already encoded inside the Moore machine, the use of *buildMoore* makes that fact clear.

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

LEMMA 22 - The function *mfoldl* is a natural transformation between the profunctors *Moore* and *Fold*.

Proof. We need to prove that the following diagram commutes.

$$\begin{tikzcd}
\text{Moore } b\ c \arrow[r, "f"] \arrow[d, "\text{mfoldl}"'] & \text{Moore } a\ d \arrow[d, "\text{mfoldl}"] \\
\text{Fold } b\ c \arrow[r, "g"'] & \text{Fold } a\ d
\end{tikzcd}$$

Since *Moore* and *Fold* are profunctors f, and g are both of the form $f = dimap_{Moore}\ h\ i$, and $g = dimap_{Fold}\ h\ i$, for arbitrary $h\ :\ a \to b$, and $i\ :\ c \to d$. The commuting diagram tells that we need to prove the following rule.

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

The result can be obtained by employing structural induction on the Moore type, as formalized in Agda.

□

Lemma 22 gives us the corresponding *fusion law* 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, s the analogous term is $mfoldl \circ dimap\ f\ id$. Hence, the fusion law is a corollary of Lemma 22.

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

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 $Scan\ a\ b = SISO\ ZipList\ ZipList\ a\ b$.

```
scanl :: (b → a → b) → b → [a] → [b]
scanl f b [] = [b]
scanl f b (x : xs) = b : scanl f (f b x) xs
mscanl :: Moore a b → Scan a b
mscanl m = SISO $ λ(ZipList as) → ZipList (runMoore m as)
```

Lemma 23 - The function *mscanl* is a natural transformation between the profunctors *Moore* and *Scan*.

Proof. The proof procceds with the same reasoning as Lemma 22, thus one needs to show that the following diagram commutes.

$$\begin{tikzcd}
\text{Moore } b\ c \arrow[r, "f"] \arrow[d, "\text{mscanl}"'] & \text{Moore } a\ d \arrow[d, "\text{mscanl}"] \\
\text{Scan } b\ c \arrow[r, "g"'] & \text{Scan } a\ d
\end{tikzcd}$$

□

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

The *mfoldl* function in our structure respects the unit *mpempty* and the monoidal multiplication inherent to a monoidal profunctor. However, it does not preserve the sequential composition property that a semiarrow would. This is because, in Moore machines, a delay is introduced when the compositions are sequenced. In other words, an output from one function is not immediately used as input for the next, due to the presence of an intermediate state. In contrast, the *Fold* function does not introduce such delays, as it immediately passes the result of one function as input to the next. This behavior is also observed with the *mscanl* function.

LEMMA 24 - The functions *mfoldl*, and *mscanl* preserves *mpempty*.

$$\mathit{mfoldl}\ \mathit{mpempty} = \mathit{mpempty}$$

$$\mathit{mscanl}\ \mathit{mpempty} = \mathit{mpempty}$$

Proof. Firstly, we notice that $\mathit{mfoldl}\ \mathit{mpempty} = \mathit{runMooref}\ (\mathit{Moore}\ ()\ (\backslash\_ \rightarrow \mathit{mpempty}))$, the RHS is the mpempty of a function type, that is $\mathit{const}\ ()$. 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 25 - The functions *mfoldl*, and *mscanl* preserves $\star$.

$$\mathit{mfoldl}\ (m \star n) = \mathit{mfoldl}\ m \star \mathit{mfoldl}\ n$$

$$\mathit{mscanl}\ (m \star n) = \mathit{mscanl}\ m \star \mathit{mscanl}\ n$$

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

$$\mathit{mfoldl}\ (m \star n) = \lambda \mathit{ls} \rightarrow \mathit{runMooref}\ (\mathit{Moore}\ (b, d)\ (\lambda(a, c) \rightarrow \mathit{am}\ a \star \mathit{cm}\ c)\ \mathit{ls},$$

and

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

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

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

For $\mathit{ls} = [\,]$, 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 $\mathit{ls} = \mathit{zs}$, and we want to show that the equation holds for the prepended input pairs $\mathit{ls} = (x, y) : \mathit{zs}$ 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 $x : \mathit{zs}$. By function extensionality, we get the desired result for. Thus, $mfoldl\ (m{\star}n) = mfoldl\ m{\star}mfoldl\ n$. The Agda formalization faithfully follows the proof described above.

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 25 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.

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

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.

$$\mathit{bifold}\ f\ g\ (e,u)\ \mathit{as}\ \mathit{bs} = (\mathit{foldl}\ f\ e\ \mathit{as}, \mathit{foldl}\ g\ u\ \mathit{bs})$$

LEMMA 26 - The functions *mfoldl* and *mscanl* are monoidal profunctor homomorphisms, i.e., they preserve *mpempty* and $\star$.

Proof. This follows directly from Lemma 24 and Lemma 25. □

The above result allows us to reason about left folds as a categorical construct such as a monoidal profunctor, however it is worth noting that *mfoldl* and *mscanl* do not preserve the semiarrow operation $\cdot$. This fact occurs because we cannot compose the *Fold* arrows in an evident way. For *mscanl*, we observe that $\cdot$ introduces a delay as discussed earlier in this section, while *Scan* does not introduce any delay. Thus, both functions are not semiarrow homomorphisms.

Chapter 6

Conclusion

This thesis explored the concept of monoidal profunctors in the context of monoidal categories of profunctors. We demonstrated that such a category possesses symmetric closure and derived the free construction of monoidal profunctors. Our study shows the implementation of monoidal profunctors in Haskell, particularly highlighting their relevance in optics and parallel programming. For scenarios requiring the capability to manage effects during split and merge operations in parallel computations, we introduced an extended version of the monoidal profunctor, called effectful monoidal profunctor. Additionally, the *semiarrow* in a *semiarrow* category, which presents another variant of the monoidal profunctor, reveals an interesting mathematical structure. This structure is adept at modeling sequential compositions that do not require identities, making it suitable for representing Moore machines and synchronous data-flow programming. In this study, we also derived a categorical bridge between Moore machines and folds. This connection is characterized through a specialized monoidal profunctor termed *SISO*, an acronym for 'structured input, structured output.' The relationship is explored by a corresponding monoidal profunctor homomorphism.

6.1 Related Work

The work of Rivas and Jaskelioff [RJ17] elucidates how to structure computations using monoids in specific monoidal categories and offers free constructions for each. In the work of Pickering, Gibbons, and Wu [PGW17] defined the concept of monoidal profunctors in Haskell in their work on optics. Within the Haskell community, the product-profunctors package provides definitions and examples of monoidal profunctors [Ellb] and is extensively utilized by the opaleye package [Ella]. Bartosz Milewski [Mil] further explored the free construction of a monoidal profunctor, introducing initial examples in Haskell.

In "Profunctor Optics: A Categorical Update" [CEG^{+}20], the authors generalize profunctor optics using enriched categories, doubles and Tambara modules. The *Monocle* optic in this work employs a methodology reminiscent of Jaskelioff and O'Connor's approach [JO14] that derives lenses using a representation theorem.

Regarding semiarrows, observations in [LCH09] on causal commutative arrows illuminate the properties of concurrent computations. This research introduces an extension to the simple type lambda calculus, leading to the derivation of the Causal Commutative Normal Form (CCNF). Another notable work, "Multi-Level Languages are Generalized Arrows" [Meg10], proposes a novel generalization excluding the *arr* function, targeting metaprogramming. Both pieces serve as foundational inspirations for the conceptualization of semiarrows.

Conclusively, Uustalu and Verne's work on data-flow programming [UV06] has significantly influenced the exploration of applications for semiarrows, especially concerning its delay algebra.

6.2 Future Work

This research is a step forward on using the monoidal profunctor and semiarrowinterfaces in functional programming, we provided a deep understanding on both ways of structure pure functional programs. However, some challenges must be tackled in future work as follows:

- **A sugarized syntax for the semiarrow interface.** As in arrows, a sugarized syntax for monoidal profunctor and semiarrowis a good idea. It could give a synchronous program syntax style and make the life easier for a programmer with such a background. This kind of effort, called mproc syntax, was started by the author but stopped due to the lack of time. The author used the GHC parser and internals to accommodate those changes in a separate GHC branch. One can check it out and contribute in https://github.com/romefeller/ghc/tree/ago%40mproc-syntax.

- **Study more examples of semiarrows.** Another *semiarrow* example worth investigating is an effectful Moore machine, given by the type **data** *Moore m a b* = *Moore b* (*a* → *m* (*Moore m a b*)). When the inner monad *m*, representing an effectful state transition, is a commutative monad, the semiarrowlaws are satisfied. This situation can help model hidden Markov chains in a composable manner.

- **Notions of grading using semiarrows.** The semiarrow structure can also model graded computations due to the absence of an identity for the sequential composition. The semiarrow can be extended to a graded semiarrow with an extra parameter *t*. The idea is to have the following multi-param typeclasses:

  ```
  class (GradedProfunctor t p) where
    gdimap :: (a → b) → (c → d) → p t b c → p t a d
  class (GradedMonoPro t p) where
    gmpempty :: p t () ()
    gpar :: p t a b → p t c d → p t (a, c) (b, d)
  class (GradedSemiArrow t p) where
    gseq :: p t b c → p t′ a b → p (Sum t t′) a c
  ```

  where *Sum* is a dependent typeclass. This additional parameter *t* can be viewed as time or also as cost (or gas) as in the cryptocurrencies field.

- **Improve the graphical language for delays.** Constructing a synchronous program using this kind of tool is valuable, and further development in this area is needed. Another path to be taken is to derive optimizations and normal forms for it, similar to the case of causal commutative arrows [LCH11].

- **Investigate more deeply the operadic semantics for synchronous programs.** The typeclass *GSemiArrow* used in this work only provides 2-delay computations. One should explore n-delay operations and how to implement them efficiently. For example, for 3-delay operations, a typeclass like the one below needs to be investigated. Additionally, laws and the non-associative behavior should be studied carefully.

  ```
  class G3SemiArrow p where
    c1 :: p (d, e, f) g → p (a, b, c) d → p ((a, b, c), e, f) g
    c2 :: p (e, d, f) g → p (a, b, c) d → p (e, (a, b, c), f) g
    c3 :: p (e, f, d) g → p (a, b, c) d → p (e, f, (a, b, c)) g
  ```

- **Lookup on other connections between monoidal profunctors that could gives us useful laws.** Examining other connections between monoidal profunctors, as well as semiarrows, could yield new laws for known results about data structures like lists, for example.

Bibliography

[Atk11] Robert Atkey. What is a categorical model of arrows? *Electronic Notes in Theoretical Computer Science*, 229(5):19–37, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).

[Awo10] Steve Awodey. *Category Theory*. Oxford University Press, Inc., New York, NY, USA, 2nd edição, 2010.

[Bir84] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Inf.*, 21(3):239250, oct 1984.

[BL80] H. Barendregt and G. Longo. Equality of λ-terms in the model t^{ω}. Em *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[BLT18] Martin Braun, Oleg Lobachev and Phil Trinder. Arrows for parallel computation. *CoRR*, abs/1801.02216, 2018.

[BM11] M. Batanin and M. Markl. Centers and homotopy centers in enriched monoidal categories, 2011.

[CEG$^+$20] Bryce Clarke, Derek Elkins, Jeremy Gibbons, Fosco Loregiàn, Bartosz Milewski, Emily Pillmore and Mario Román. Profunctor optics, a categorical update. *ArXiv*, abs/2001.07488, 2020.

[CK14] Paolo Capriotti and Ambrus Kaposi. Free applicative functors. *Electronic Proceedings in Theoretical Computer Science*, 153:230, Jun 2014.

[CW01] Mario Cáccamo and Glynn Winskel. A higher-order calculus for categories. Em *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '01, página 136153, Berlin, Heidelberg, 2001. Springer-Verlag.

[Day70] Brian Day. On closed categories of functors. Em S. MacLane, H. Applegate, M. Barr, B. Day, E. Dubuc, Phreilambud, A. Pultr, R. Street, M. Tierney and S. Swierczkowski, editors, *Reports of the Midwest Category Seminar IV*, páginas 1–38, Berlin, Heidelberg, 1970. Springer Berlin Heidelberg.

[DHJG06] Nils A. Danielsson, John Hughes, Patrik Jansson and Jeremy Gibbons. Fast and Loose Reasoning is Morally Correct. Em *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 41 of *POPL '06*, páginas 206–217, New York, NY, USA, Janeiro 2006. ACM.

[Ella] Tom Ellis. opaleye: An sql-generating dsl targeting postgresql. https://hackage.haskell.org/package/opaleye. Accessed: 2019-05-28.

[Ellb] Tom Ellis. Product-profunctors. https://hackage.haskell.org/package/product-profunctors. Accessed: 2019-05-20.

[Gil] Andy Gill. mtl: Monad classes, using functional dependencies. https://hackage.haskell.org/package/mtl. Accessed: 2019-05-28.

[Has09] Masahito Hasegawa. On traced monoidal closed categories. *Mathematical. Structures in Comp. Sci.*, 19(2):217–244, Abril 2009.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[HM98] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *J. Funct. Program.*, 8(4):437–444, Julho 1998.

[Hug00] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(13):67111, may 2000.

[Hug05] John Hughes. Programming with arrows. Em *Proceedings of the 5th International Conference on Advanced Functional Programming*, AFP'04, páginas 73–129, Berlin, Heidelberg, 2005. Springer-Verlag.

[JD93] Mark P. Jones and Luc Duponcheel. Composing monads. Relatório técnico, 1993.

[JHH09] Bart Jacobs, Chris Heunen and Ichiro Hasuo. Categorical semantics for arrows. *Journal of Functional Programming*, 19(3-4):403438, 2009.

[JO14] Mauro Jaskelioff and Russell O'Connor. A representation theorem for second-order functionals. *ArXiv*, abs/1402.1699, 2014.

[Kmea] Edward Kmett. lens: Lenses, folds and traversals. https://hackage.haskell.org/package/lens. Accessed: 2019-05-28.

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

[Kmec] Edward Kmett. Profunctors. https://hackage.haskell.org/package/profunctors. Accessed: 2019-03-16.

[LCH09] Hai Liu, Eric Cheng and Paul Hudak. Causal commutative arrows and their optimization. *SIGPLAN Not.*, 44(9):35–46, Agosto 2009.

[LCH11] Hai Liu, Eric Cheng and Paul Hudak. Causal commutative arrows. *J. Funct. Program.*, 21(4-5):467–496, 2011.

[Lei03] Tom Leinster. Higher operads, higher categories, 2003.

[LHJ95] Sheng Liang, Paul Hudak and Mark Jones. Monad transformers and modular interpreters. Em *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, página 333343, New York, NY, USA, 1995. Association for Computing Machinery.

[Lip11] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide.* No Starch Press, San Francisco, CA, USA, 1st edição, 2011.

[Lor15] Fosco Loregian. This is the (co)end, my only (co)friend, 2015.

[Lor21] Fosco Loregian. *(Co)end Calculus.* London Mathematical Society Lecture Note Series. Cambridge University Press, 2021.

[LWY11] Sam Lindley, Philip Wadler and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 229(5):97 – 117, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).

[Mac71] Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

[Meg10] Adam Megacz. Multi-level languages are generalized arrows, 2010.

[Mil] Bartosz Milewski. Free monoidal profunctors. https://bartoszmilewski.com/2018/02/20/free-monoidal-profunctors. Accessed: 2019-10-20.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, Julho 1991.

[MP08] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Janeiro 2008.

[Obr98] Davor Obradovic. Structuring functional programs by using monads, 1998.

[O'Ca] RusselL O'Connor. Grate: A new kind of optic. https://r6research.livejournal.com/28050.html. Accessed: 2019-02-02.

[O'Cb] Russell O'Connor. A representation theorem for second-order pro-functionals. https://r6research.livejournal.com/27858.html. Accessed: 2019-02-01.

[Pat01] Ross Paterson. A new notation for arrows. *SIGPLAN Not.*, 36(10):229–240, Outubro 2001.

[PGW17] Matthew Pickering, Jeremy Gibbons and Nicolas Wu. Profunctor optics: Modular data accessors. *The Art, Science, and Engineering of Programming*, 1(2), Apr 2017.

[RJ17] Exequiel Rivas and Mauro Jaskelioff. Notions of computation as monoids. *Journal of Functional Programming*, 27:e21, 2017.

[RJ18] Exequiel Rivas and Mauro Jaskelioff. Monads with merging. Technical report, 2018.

[RJS18] Exequiel Rivas, Mauro Jaskelioff and Tom Schrijvers. A unified view of monadic and applicative non-determinism. *Science of Computer Programming*, 152:70–98, 2018.

[SD96] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. Em John Launchbury, Erik Meijer and Tim Sheard, editors, *Advanced Functional Programming*, páginas 184–207, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[Spi90] Mike Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, 1990.

[UV06] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. Em Zoltán Horváth, editor, *Central European Functional Programming School*, páginas 135–167, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Wad92] Philip Wadler. Comprehending monads. *Math. Struct. Comput. Sci.*, 2(4):461–493, 1992.

[Wad95] Philip Wadler. Monads for functional programming. Em *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, páginas 24–52, Berlin, Heidelberg, 1995. Springer-Verlag.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. Em *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, páginas 60–76, New York, NY, USA, 1989. ACM.

Index