



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Armando João Isáias Ferreira dos Santos

Selective Applicative Functors & Probabilistic Programming

January 2020



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Armando João Isaías Ferreira dos Santos

Selective Applicative Functors & Probabilistic Programming

Master dissertation

Master Degree in Computer Science

Dissertation supervised by

José Nuno Oliveira (INESCTEC & University of Minho)

Andrey Mokhov (Jane Street - London)

January 2020

ACKNOWLEDGEMENTS

Acknowledgements to be written here.

ABSTRACT

This document is a pre-dissertation report describing the work carried out so far in the MSc project “Selective Applicative Functors & Probabilistic Programming”.

In functional programming, *selective applicative* functors (SAF) are an abstraction between applicative functors and monads. This abstraction requires all effects to be statically declared, but provides a way to select which effects to execute dynamically. SAF have been shown to be a useful abstraction in several examples, including two industrial case studies. Selective functors have been used in static analysis. The collection of information about all possible effects in a computation and the fact that they enable *speculative* execution make it possible to take advantage of selective functors to describe probabilistic computations instead of using monads. In particular, selective functors appear to provide a way to encode a more efficient implementation of probability distributions than monads.

This dissertation addresses a probabilistic interpretation for the *arrow* and *selective* abstractions in the light of the linear algebra of programming discipline. It also provides a Haskell type-safe matrix library capable of expressing probability distributions and probabilistic computations as typed matrices.

Keywords: master thesis, functional programming, probabilistic programming, monads, applicatives, selective applicative functor, Haskell

RESUMO

Os Funtores Aplicativos Seletivos são uma abstração entre os aplicadores e as mônadas. Esta abstração requer que todos os efeitos sejam declarados estaticamente, mas fornece uma maneira de selecionar quais efeitos executar dinamicamente. Demonstrou-se que os Funtores Aplicativos Seletivos são uma abstração útil em vários exemplos, incluindo dois casos de estudo industriais. Dado o interesse e a aplicação de funtores seletivos na análise estática, a coleção de informações sobre todos os efeitos possíveis numa computação e o facto de fornecerem a capacidade de ter execução especulativa de efeitos, parece ser possível tirar vantagem dos funtores seletivos para descrever computações probabilísticas. Em particular, os funtores seletivos parecem fornecer uma maneira de codificar uma implementação mais eficiente de distribuições do que as monads. Este documento descreve o trabalho realizado no relatório de pré-dissertação da tese de mestrado e apresenta uma interpretação probabilística da abstração Arrow e Seletiva à luz da disciplina da Álgebra Linear da Programação, além de fornecer uma biblioteca de matrizes segura do tipo Haskell capaz de expressar matrizes como distribuições de probabilidade.

Palavras-chave: dissertação de mestrado, programação funcional, programação probabilística, mônadas, aplicativos, funtores aplicativos seletivos, haskell

CONTENTS

1	INTRODUCTION	1
1.1	Context	1
1.2	Motivation and Goals	2
1.3	State of the art	3
1.3.1	Hierarchy of abstractions	3
1.3.2	Functors	4
1.3.3	Applicative functors	4
1.3.4	Monads	5
1.3.5	Arrows	7
1.3.6	Selective applicative functors	9
1.4	Related work	10
1.4.1	Exhaustive probability distribution encoding	10
1.4.2	Embedded domain specific languages	11
1.5	Summary	12
1.6	Structure of the dissertation	12
2	BACKGROUND	14
2.1	Path to Probabilistic Programming	14
2.1.1	Set theory	14
2.1.2	Basic probabilities and distributions	15
2.2	(Linear) Algebra of Programming	20
2.2.1	Structure of the category of Matrices	20
2.2.2	Biproducts	21
2.2.3	Biproduct functors	22
3	THE PROBLEM AND ITS CHALLENGES	24
3.1	Probabilistic interpretation of Selective Functors	24
3.2	Inefficient probability encodings	24
3.3	Research methodology	25
3.4	Work carried out thus far	25
3.4.1	Probabilistic interpretation of Arrows	25
3.4.2	Type safe Linear Algebra of Programming matrix library	26
3.4.3	Probabilistic Interpretation of Selective Functors	29
3.4.4	Type safe inductive matrix definition	31
3.4.5	The probability distribution matrix and the Selective abstraction	34
4	PROSPECT OF FUTURE WORK	36
A	TYPE SAFE LAOP MATRIX WRAPPER LIBRARY	41
B	TYPE SAFE LAOP INDUCTIVE MATRIX DEFINITION LIBRARY	51

LIST OF FIGURES

LIST OF TABLES

Table 1	Number of possible arrangements of size r from n objects	19
---------	--------------------------------------------------------------	----

LIST OF LISTINGS

1.1	Functor laws	4
1.2	Applicative laws	4
1.3	Monad laws and definition in terms of unit and join	6
1.4	Monad laws and definition in terms of unit and bind	6
1.5	Relation between join and bind	6
1.6	Arrow type-class	7
1.7	Arrow Kleisli type-class instance	8
1.8	Selective Applicative Functor laws	9
3.1	Inductive matrix definition	26
3.2	Type safe wrapper around HMatrix	27
3.3	Interface equivalent function implementations	27
3.4	LAoP Monty Hall Problem	28
3.5	Selective ArrowMonad instance	29
3.6	LAoP Selective instance	29
3.7	Inductive Matrix definition	31
3.8	Matrix composition and abiding functions	32
3.9	Dimensions are type level naturals	33
3.10	Dimensions are arbitrary data types	33
3.11	Dist type alias	34
3.12	Dist - select and cond operators	34
3.13	Constrained monad instance	35
A.1	Matrix.Internal	41
B.1	Matrix.Internal	51

ACRONYMS

A

AOP Algebra of Programming. [20](#), [22](#), [25](#), [26](#), [30](#)

C

CS Computer Science. [2](#), [3](#), [12](#)

CT Category Theory. [2–5](#), [7](#), [12](#), [24](#), [25](#)

D

DI Informatics Department. [1](#)

F

FP Functional Programming. [2](#), [12](#), [25](#)

G

GADT Generalised Algebraic Datatype. [26](#), [32](#)

GHC Glasgow Haskell Compiler. [27](#), [31](#), [32](#)

L

LAOP Linear Algebra of Programming. [8](#), [20](#), [22](#), [25–28](#), [31](#), [32](#), [34](#)

M

MIEI Integrated Master in Computer Engineering. [1](#)

P

PPF Probabilistic Functional Programming. [1](#), [2](#)

PPL Probabilistic Programming Language. [11](#)

S

SAF Selective Applicative Functors. [2](#), [9–12](#), [24–27](#), [29](#), [31](#), [34–36](#)

U

UM University of Minho. [1](#)

INTRODUCTION

This pre-dissertation report describes the preliminary work carried out in the context of the author's *Integrated Master in Computer Engineering (MIEI)* offered by the *Informatics Department (DI)* of the *University of Minho (UM)*.

The report is structured in the following way. This chapter provides the context, motivation and overall goals of the dissertation. It also presents a review of the state of the art and work related to the subject of this dissertation. Chapter 2 introduces the most relevant background topics and chapter 3 explains in more detail what the problem is, its main challenges and which research methodology will be followed for finding useful results in the problem domain.

1.1 CONTEXT

Monads were pioneered by Moggi (1991) in the field of computer science to verify effectful programs, i.e. programs that deal with side effects. Wadler (1989) further introduced monads in functional programming as a general and powerful approach for describing effectful (or impure) computations, while still using pure functions. The key ingredient of the monad abstraction is the *bind* operator. This operator leads to an approach to composing effectful computations which is inherently sequential. This intrinsic nature of monads can be used for conditional effect execution. However, sometimes the abstraction is too strong for particular programming situations, and abstractions with weaker laws are welcome.

Applicative functors (McBride and Paterson, 2008) can be used for composing statically known collections of effectful computations, as long as these computations are independent from each other. However, this kind of functor can only take two effectful computations and, independently (i.e. in parallel), compute their values and return their composition.

There is a need for programming abstractions that, while requiring all effects to be statically declared, provide a way to select which of the effects to execute dynamically. The Selective Applicative Functor (SAF) abstraction solves this issue (Mokhov et al., 2019).

In the field of *Probabilistic Functional Programming (PFP)*, monads are used to describe events (probabilistic computations in this case) that depend on others (Erwig and Kollmansberger, 2006). Better than monads, which are inherently sequential, selective functors provide a better abstraction for describing conditional probabilistic computations. This kind of functor has proved to be a very helpful abstraction in the fields of static analysis (at Jane Street) and speculative execution (at Facebook), and achieved great results without compromising the naturality of the adopted code style (Mokhov et al., 2019).

Arrows (Hughes, 2000) are more generic than monads and were designed to abstract the structure of more complex patterns that the monad interface could not support. The most common example is the parsing library by Swierstra and Duponcheel (2001) that takes advantage of static analysis to improve its performance. This example could not be optimised using the Monad interface given its sequential nature. Having *Category Theory (CT)* as a foundation, the Arrow abstraction has made its way to the *Functional Programming (FP)* ecosystem in order to mitigate the limitation of the powerful Monad.

There are reasons to believe that by adopting the selective abstraction one could shorten the gap that once was only filled by the Arrow abstraction (Hughes, 2000). On the one hand, the generality of the Arrow interface allows one to overcome some of the structural limitation that refrains from implementing a stronger abstraction and compose different combinators to achieve increased expressiveness. On the other hand, languages such as Haskell, which incorporate many of these abstractions out of the box, make code written in the Arrow style not only convoluted and unnatural, but also hard to refactor.

1.2 MOTIVATION AND GOALS

The rise of new topics such as e.g. machine learning, deep learning and quantum computing, and software engineering in general are stimulating major advances in the programming language domain. To cope with the increased complexity, mathematics always had a main role, either by formalising the underlying theory, or by providing robust and sound theories to deal with the new heavy machinery. But what do these topics have in common? They all deal, in some way, with probabilities.

Programming languages are the most useful means of expressing complex concepts, because they provide a way to express, automate, abstract and reason about them. There are programming languages, specially functional programming languages, that work more closely to the mathematical level and are based in concepts like referential transparency and purity. However, not all of the abstractions useful in *Computer Science (CS)* come from mathematics, explicitly. There are several abstractions that were meant to factor out some kind of ubiquitous behaviour or to provide a sound and robust framework where one could reason about the code and provide a more efficient solution. The *Selective Applicative Functors (SAF)* is such an abstraction.

Probabilistic programming allow programmers to model probabilistic events and predict or calculate results with a certain degree of uncertainty. Specifically PFP manipulates and manages probabilities in an abstract, high-level way, abstracting away all the convoluted notation and complex mathematical formulas. Probabilistic programming research is primarily focused on developing optimisations to inference and sampling algorithms in order to make code run faster while preserving the posterior probabilities. There are many strategies and techniques for optimising probabilistic programs, namely using static analysis (Bernstein, 2019).

The main goal of this research is to study, evaluate and compare ways of describing and implementing probabilistic computations using the so-called *selective abstraction*. In particular, we want to evaluate the benefits of doing so in the PFP ecosystem. This will be accomplished by proposing an appropriate set of case studies and, ultimately, developing a Haskell library that provides an efficient encoding of probabilities, taking advantage of the selective applicative abstraction. Focussing on how

to overcome the intrinsic sequential nature of the monad abstraction (Ścibior et al., 2015) in favour of the speculative execution of the selective functors, this work aims to answer the following research question:

“Can the select operator be implemented more efficiently than the bind operator?”

1.3 STATE OF THE ART

In the context of this research, abstractions can be viewed from two prisms:

- The programming language prism;
- The underlying mathematical theory prism.

As expected, the programming language prism makes one see things more concretely, i.e. brings one down the abstraction ladder. That is why normally many abstractions tend to be associated to quite frequent patterns and interfaces that programmers wish to generalise.

This said, a recurrent research problem happens when authors try to explain their mathematical abstractions by going down to a comfortable, intuitive and easy to understand level (Petricek, 2018). However, in CS the level might be so low that the need for such abstractions may be in doubt. Mathematical abstractions are useful ways of generalising and organising patterns that are too repetitive or have much in common. Thanks to a lot of work on topics like abstract algebra or CT, these abstractions automatically become powerful conceptual tools. In this regard, finding the right mathematical description of an abstraction is *halfway* for a correct development.

The following section presents widely used mathematical abstractions that made its way into programming languages, in particular in the probabilistic programming environment. How recent work by Mokhov et al. (2019) relates to such abstractions will also be addressed. Given the scope of this research and aiming to explore interesting ways of thinking about probability distributions, every abstraction is introduced accompanied by a concrete instance in the probabilistic setting.

1.3.1 *Hierarchy of abstractions*

The purpose of every abstraction is to generalise a certain pattern or behaviour. Abstract algebra is a field of mathematics devoted to studying mathematical abstractions. In particular, by studying ways of building more complex abstractions by combining simpler ones. Regarding abstractions as layers, one can pretty much think of a heritage mechanism that is so fond of object oriented programming (Liskov, 1987).

A hierarchy of abstractions aims to hide information and manage complexity. The highest level has the least information and lowest complexity. For the purposes of this research, it is interesting to see how the abstractions presented in the next sections map to the corresponding probability theory features and how the underlying levels translate to more complex ones.

1.3.2 Functors

WHAT FUNCTORS ARE Functors originate from CT as morphisms between categories (Awodey, 2010). Functors abstract a common pattern in programming and provide the ability to map a function inside some kind of structure. Since functors must preserve structure they are a powerful reasoning tool in programming.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  — fmap id = id
  — fmap f . fmap g = fmap (f . g)
```

Listing 1.1: Functor laws

PROBABILISTICALLY SPEAKING There are many situations in which the type $f\ a$ makes sense. The easiest way to understand it is to see f as a data container; then readers can instantiate f to a concrete type, for instance lists ($[a]$).

Probabilistically speaking, $f\ a$ instantiates to the “Distribution of a ’s” container and $fmap$ (the factored out pattern) as the action of mapping a function through all the values of a distribution without changing their probabilities. (Probabilities will sum up automatically wherever f is not injective.) As will be seen in section 2.1 there are multiple ways of combining probabilities. However, given the properties of a functor, it is only possible to map functions inside it while preserving its structure. This said, the probability functor can be casually seen as only being capable to express $P(A)$ in probability theory (Tobin, 2018).

1.3.3 Applicative functors

WHAT APPLICATIVE FUNCTORS ARE Most functional programming languages separate pure computations from effectful ones. Effectful computations can be seen as every computation that performs side effects or runs in a given context. McBride and Paterson (2008), while working with the Haskell functional programming language, found that the pattern of applying pure functions to effectful computations popped out very frequently in a wide range of fields. The pattern consists mostly of 1. embedding a pure computation in the current context while maintaining its semantics, i.e. lifting a value into an “effect free” context, and then 2. combining the results of the computations, i.e. *applying* a pure computation to effectful ones. To abstract this pattern all it takes is a way to factor out 1 and 2.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  — pure id <*> u == u
  — pure f <*> pure x == pure (f x)
  — u <*> (v <*> w) = pure (.) <*> u <*> v <*> w
```

$$\text{--- } u \text{ <*> pure } y = \text{pure } (\$ y) \text{ <*> } u$$
Listing 1.2: **Applicative laws**

It is important to note that for f to be an applicative it first needs to be a functor. So, every applicative is a functor. This hierarchy can be seen as going down one layer of abstraction, empowering a functor f with more capabilities if it respects the applicative laws (given in the listing above).

Applicatives are interesting abstractions in the sense that they were not a transposition of a known mathematical concept. However, [McBride and Paterson \(2008\)](#) establish a correspondence with the standard categorical “zoo” concluding that *in categorical terms applicative functors are strong lax monoidal functors*. This has opened ground for a stream of fascinating research, see e.g. ([Paterson, 2012](#); [Cooper et al., 2008](#); [Capriotti and Kaposi, 2014](#)).

PROBABILISTICALLY SPEAKING Looking at the laws of applicative functors one sees that they pretty much define what the intended semantic regarding sequencing effects is. The last one, called the *interchange law* ([McBride and Paterson, 2008](#)), clearly says that when evaluating the application of an effectful function to a pure argument, the order in which we evaluate the function and its argument *does not matter*. However, if both computations are effectful the order *does matter*, but a computation cannot depend on values returned by prior computations, i.e. the result of the applicative action can depend on earlier values but the effects cannot. In other words, computations can run *independently* from each other ([Cooper et al., 2008](#); [Marlow et al., 2014, 2016](#); [Mokhov et al., 2019](#)).

So, if f represents a distribution then pure can be seen as the embedding of a given value a in the probabilistic context with 100% chance, and (<*>) as the action responsible of combining two *independent* distributions, calculating their joint probability. This said, the probability instance of applicative functors can be regarded as being able to express $P(A, B) = P(A)P(B)$, i.e. statistical independence ([Tobin, 2018](#)).

1.3.4 Monads

WHAT MONADS ARE Before being introduced into programming languages, monads were already frequently used in algebraic topology by [Godement \(1958\)](#) and [CT](#) by [MacLane \(1971\)](#). Monads were used in this areas because they were able to embed a given value into another structured object and because they were able to express a lot of different constructions in a single structure ([Petricek, 2018](#)). A proof of the flexibility and usefulness of Monads is its application in programming: [Moggi \(1991\)](#) introduced monads in order to be capable of reasoning about effectful programs and [Wadler \(1995\)](#) used them to implement effectful programs in Haskell. Although they are not the same, the mathematical monad and the programming language monad are related, fruit of the prism from each one is observed.

Definition 1.3.1. A monad in a category \mathcal{C} is defined as a triple (T, η, μ) where $T : \mathcal{C} \rightarrow \mathcal{C}$ is a functor; $\eta : Id_{\mathcal{C}} \rightarrow T$ and $\mu : T^2 \rightarrow T$ are natural transformations, such that:

$$\begin{aligned}\mu_A \cdot T\mu_A &= \mu_A \cdot \mu_{TA} \\ \mu_A \cdot \mu_{TA} &= id_{TA} = \mu_A \cdot T\mu_A\end{aligned}$$

In programming two alternative, but equivalent definitions for monads come up. If a functor is seen as a type constructor and natural transformations as functions:

```
class Applicative m => Monad m where
  unit :: a -> m a
  join :: m (m a) -> m a
  — join . join = join . fmap join
  — join . unit = id = join . fmap unit
  — fmap f . join = join . fmap (fmap f)
  — fmap f . unit = unit . f
```

Listing 1.3: Monad laws and definition in terms of unit and join

```
class Applicative m => Monad m where
  unit :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  — unit a >>= f = unit (f a)
  — m >>= unit = m
  — (m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

Listing 1.4: Monad laws and definition in terms of unit and bind

As we can see both definitions once again highlight the hierarchy of abstractions where every monad is an applicative, and consequently a functor. These two definitions are related by the following law:

```
m >>= f = join (fmap f m)
```

Listing 1.5: Relation between join and bind

If monads are so versatile what type of pattern do they abstract? Intuitively, monads abstract the idea of “taking some uninteresting object and turn it into something with more structure” (Petricek, 2018). This idea can be explained by using some of several known metaphors:

- Monads as *containers*: Visualising it as a box to represent the type $m\ a$, the unit operation takes a value and wraps it in a box and the join operation takes a box of boxes and wraps it into a single box. This metaphor however, is not so good at giving intuition for bind ($>>=$) but it can be seen as a combination of `fmap` and `join`.
- Monads as *computations*: Visualising $m\ a$ as a computation, $a \rightarrow m\ b$ represents computations that depend on previous values so, bind lets us combine two computations emulating the sequential, imperative programming paradigm and unit represents a computation that does nothing.

Brought to programming languages, monads are used to encode different notions of computations and its structure allows us to reuse a lot of code. However they are also useful for sequencing effect with the aid of syntactic sugar ([HaskellWiki, 2019](#)).

PROBABILISTICALLY SPEAKING It is more rewarding to look at probability distributions as a probabilistic computation or event. Given this, by observing the type of `bind` we can infer that it let us combine an event `m a` with another that depends on the previous value `a → mb` ([Erwig and Kollmansberger, 2006](#)). In other words, `bind` very much encapsulates the notion of conditional probabilities. What happens in a conditional probability calculation $P(B|A)$ is that A becomes the sample space and A and B will only occur a fraction $P(A \cap B)$ of the time. Making the bridge with the type signature of $(\gg=) : m\ a \rightarrow mb$ represents the new sample space A and $a \rightarrow mb$ the fraction where A and B occur. This being said, the probability monad can be usually seen as being able to express $P(B|A) = \frac{P(A \cap B)}{P(A)}$.

The observation that probability distributions form a monad is not new. Thanks to the work of [Giry \(1982\)](#) and following the hierarchy of abstractions, we see that it is indeed possible to talk about probabilities with respect to the weaker structures mentioned in the other sections ([Tobin, 2018](#); [Ścibior*, 2019](#)).

1.3.5 Arrows

WHAT ARROWS ARE Most of the abstractions described until now are based on [CT](#). The reason for this is because [CT](#) can be seen as the “theory of everything”, a framework where a lot of mathematical structures fit in. So, how can such an abstract theory be so useful in programming? Because computer scientists value abstraction! When designing an interface it is meant to reveal as little as possible about the implementation details and it should be possible to switch the implementation with an alternative one, i.e. other *instances* of the same *concept*. It is the generality of a monad that is so valuable and it is thanks to the generality of [CT](#) that makes it so useful in programming.

This being said, Arrows, introduced by [Hughes \(2000\)](#) and inspired by the ubiquity of [CT](#), aim to abstract how to build and structure more generic combinator libraries by suggesting the following type-class:

```
class Arrow a where
  arr :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b, d) (c, d)
```

Listing 1.6: **Arrow type-class**

As one can note, Arrows make the dependence on an input explicit and abstract the structure of a given output type. This is why it is said that Arrows generalise monads.

Due to the fact that there are many more arrow combinators than monadic ones, a larger set of laws are required and the reader is remitted to [Hughes \(2000\)](#) paper for more information about them. However, a brief explanation of the three combinators is given: `arr` can be seen as doing the

same as `return` does for monads, it lifts pure functions to computations; $(>>>)$ is analogous to $(>>=)$, it is the left-to-right composition of arrows; and `first` comes from the limitation that Arrows can not express binary arrow functions, so this operator converts an arrow from b to c into an arrow of pairs, that applies its argument to the first component and leaves the other unchanged.

An astute reader can see how Arrows try to encode the notion of a category and indeed the associativity law of $(>>>)$ is one of the laws of this type-class. Moreover, if one thinks about how, for any monad a function of type $a \rightarrow mb$ is an Kleisli arrow (Awodey, 2010), we can define the arrow combinators as follows:

```
newtype Kleisli m a b = K (a -> m b)

instance Arrow (Kleisli m) where
    arr f = K (\b -> return (f b))
    K f >>> K g = K (\b -> f b >>= g)
    first (K f) = K (\(b, d) -> f b >>= \c -> return (c, d))
```

Listing 1.7: Arrow Kleisli type-class instance

This shows that Arrows in fact generalise monads. Nevertheless there's still one question that goes unanswered - why generalise monads if they serve the same purpose of providing a common structure to generic programming libraries? Hughes (2000) saw in the example of Swierstra and Duponcheel (2001) a limitation on the monadic interface and argues that the advantage of the Arrow interface is that it has a wider class of implementations. Thus, simpler libraries based on abstract data types that aren't monads, can be given an arrow interface.

It seems that Arrows are more expressive than the abstractions seen in the previous sections, but what *are* their relation with them? Fortunately, Lindley et al. (2011) established the relative order of strength of $\text{Applicative} \rightarrow \text{Arrow} \rightarrow \text{Monad}$, in contrast to the putative order of $\text{Arrow} \rightarrow \text{Applicative} \rightarrow \text{Monad}$. Furthermore, given the right restrictions, Arrows are isomorphic to both Applicatives and Monads being able to "slide" between the layers of this hierarchy of abstractions.

PROBABILISTICALLY SPEAKING As seen, Arrows allow us to categorically reason about a particular structure and benefit from all the combinators that its interface offers. However, Arrows find themselves between Applicatives and Monads with respect to their strength, therefore do not express any extra special capabilities. Nevertheless, due to their generality, Arrows are able to offer either of the two abstractions' (Applicative and Monad) capabilities, provided that their laws are verified.

In fact, Monads are able to express the minimum structure to represent arbitrary probability distributions (Tobin, 2018). However, there are cases where it becomes hard to reason about probability distributions using only the monadic interface (Oliveira and Miraldo, 2016). Arrows come into play regarding this problem, allowing the so called *Linear Algebra of Programming (LAoP)* (Macedo, 2012) as it will be seen in section 3.4.

1.3.6 Selective applicative functors

WHAT SELECTIVE APPLICATIVE FUNCTORS ARE Such as Applicatives, [SAF](#) did not originate in any mathematical construction, but rather from a limitation of the interfaces in the hierarchy of abstractions established so far.

Allied to a specific research domain, like build systems and static analysis, [Mokhov et al. \(2019\)](#) saw the following limitations:

- Applicative functors, allow effects to be statically declared which makes it possible to perform static analysis. However, they only permit combining independent effects leaving static analysis of conditional effects aside;
- Monads, allow combining conditional effect, but can only do it dynamically which makes static analysis impossible.

With this being said, [Mokhov et al. \(2019\)](#) developed an interface (abstraction) which allows to get the best of both worlds, the [SAF](#).

```
class Applicative f => Selective f where
  select :: f (Either a b) -> f (a -> b) -> f b
  — x <?* pure id = either id id <$> x
  — pure x <?* (y *> z) = (pure x <?* y) *> (pure x <?* z)
  — x <?* (y <?* z) = (f <$> x) <?* (g <$> y) <?* (h <$> z)
  — where
  —   f x = Right <$> x
  —   g y = \a -> bimap (,a) ($a) y
  —   h z = uncurry z
```

Listing 1.8: Selective Applicative Functor laws

As we can observe, [SAF](#) find themselves between Applicatives and Monads and only provide one operator, `select`. By parametricity ([Wadler, 1989](#)), it is possible to understand that this operator runs an effect `f (Either a b)` which returns one of two values, `a` or `b`. In the case of the return value be of type `a`, the second effect must be run, in order to apply the function `a -> b` and obtain the `f b` value. In the case of the return value be of type `b`, then the second computation can be *skipped*.

The laws presented in the listing above characterise [SAF](#) insofar as the former indicates that the `select` operator should not duplicate any effect associated with `x` and the second indicates that `select` should not add any computation when the first one is pure, which allows it to be distributed.

It is worth noting that there is not any law that forces [SAF](#) to discard the second computation, in particular `pure (Right x) <?* y = pure x`, nor it exists a law that forces the return value of `f(a -> b)` to be applied to the value obtained by the first computation, in particular `pure (Left x) <?* y = ($x) <$> y`. The reason for this is simple: it allows instances of [SAF](#) that are useful to perform static analysis and, in the same way as Applicative functors do not restrict the order of execution of two independent effects, the `select` operator becomes more expressive.

With this in mind, it is possible to see how [SAF](#) solve the limitation of Applicatives and Monads in the context of static analysis, allowing over-approximation and under-approximation of effects in a

circuit with conditional branches. Nonetheless, SAF are useful not only in static contexts, but also in dynamic ones, benefiting from speculative execution (Mokhov et al., 2019).

From a theoretic point of view, SAF can be seen as the composition of an Applicative functor f with the Either monad (Mokhov et al., 2019). Even though this formalisation has not been studied by Mokhov et al. (2019), it is important to address the relation between SAF and Arrows. Taking advantage of the first sentence of this paragraph, we can infer that, as every SAF is an instance of Applicative, every Applicative functor is also an instance of Selective. Moreover, as it is pointed by Mokhov et al. (2019), it is possible to implement a specialised version of the bind ($>>=$) operator for any *enumerable* data type, i.e. the capacity of *selecting* an infinite number of cases makes SAFs equivalent to Monads (Pebles, 2019). It seems that, like Arrows, given the right conditions, SAF are also able to “slide” between Applicatives and Monads. As a matter of fact, Hughes (2000) had already come up with an interface that extended Arrows with conditional capabilities, the ArrowChoice type-class.

But if there was already an abstraction capable of expressing the same as SAFs why did they arise? Arrows are more general and powerful than SAFs and could be used to solve the static analysis and speculative execution examples presented by Mokhov et al. (2019). In fact, the build system DUNE (Street, 2018), is an example of the successful application of Arrows. However, adding the ability of performing static analysis or speculative execution in a code-base that is not written using the Arrow abstraction, becomes more complicated than only defining an instance for SAF in just a couple of lines. Given this, SAFs are a just good enough solution for providing the ability of static analysis of conditional effects and speculative execution without relying in the more powerful and intrusive Arrow abstraction.

1.4 RELATED WORK

1.4.1 Exhaustive probability distribution encoding

Over the past few years, the field of probabilistic programming has been primarily concerned with extending languages by offering probabilistic expressions as primitives and serving as practical tools for Bayesian modelling and inference (Erwig and Kollmansberger, 2006). As a result, several languages were created to respond to emerging limitations. Despite the observation that probability distributions form a monad is not new, it was not until later that its sequential and compositional nature was explored by Ramsey and Pfeffer (2002), Goodman (2013) and Gordon et al. (2013).

Erwig and Kollmansberger (2006) were the first to define distributions as monads by designing a probability and simulation library based on this concept. Kidd (2007), the following year, inspired by the work of Ramsey and Pfeffer (2002), introduced a modular way of probability monad construction and showed the power of using monads as an abstraction. Given this, he was able to, through a set of different monads, offer ways to calculate probabilities and explore their compositionality, from discrete distributions to sampling algorithms.

Erwig and Kollmansberger (2006), in their library, used the non-deterministic monad to represent distributions, resulting in an exhaustive approach capable of calculating the exact probabilities of any event. However, common examples of probabilistic programming grow the sample space exponentially and make it impossible to calculate the entire distribution. Despite Larsen (2011)’s

efforts to improve the performance of this library, his approach was still limited to exhaustively calculating all possible outcomes.

Apart from the asymptotic poor performance of the [Erwig and Kollmansberger \(2006\)](#) library, the use of the non-deterministic monad means that its sequential nature does not allow for further optimisations. It was with these two limitations in mind that many probabilistic programming systems were proposed.

1.4.2 *Embedded domain specific languages*

Probabilistic Programming Languages (PPLs) usually extend an existing programming language. The choice of the base language may depend on many factors such as paradigm, popularity and performance. There are many probabilistic programming languages with different trade-offs ([Ścibior et al., 2015](#)) and many of them are limited to ensure that the model has certain properties in order to make inference fast. The type of approach followed by these programming languages, such as BUGS ([Gilks et al., 1994](#)) and Infer.NET ([Minka et al.](#)), simplify writing inference algorithms for the price of reduced expressiveness.

The more generic approach, known as universal probabilistic programming, allows the user to specify any type of model that has a computable prior. The pioneering language was Church ([Goodman et al., 2012](#)), a sub-dialect of Scheme. Other examples of probabilistic programming languages include Venture and Anglican ([Mansinghka et al., 2014](#); [Tolpin et al., 2015](#)) both also Scheme sub-dialects. These universal languages are much more expressive, however, implementing inference algorithms in these is much more difficult.

[Ścibior et al. \(2015\)](#) shows that the Haskell functional language is an excellent alternative to the above mentioned languages, with regard to Bayesian modelling and development of inference algorithms. Just as [Erwig and Kollmansberger \(2006\)](#), [Ścibior et al. \(2015\)](#) use monads and develop a practical probabilistic programming library whose performance is competitive with that of the Anglican language. In order to achieve the desired performance, a less accurate than the exhaustive approach to calculating probabilities was used: sampling. This work by [Ścibior et al. \(2015\)](#), also elaborated in his doctoral dissertation ([Ścibior*, 2019](#)), kept on, giving rise to a more modular extension of the library presented in its previous work, in order to separate modelling with inference ([Ścibior et al., 2018](#)). Despite the results obtained, both solutions suffer from the fact that they use only monads to construct probability distributions, and since monads are inherently sequential they are not able to exploit parallelism in the sampling of two independent variables.

[Tobin \(2018\)](#) contributes to the investigation of embedded probabilistic programming languages, which have the advantage of benefiting from various features for free such as parser, compiler and host language library ecosystem. More than that, [Tobin \(2018\)](#) studies the functorial and applicative nature of the Girly monad and highlights its various capabilities by mapping them to the probabilistic setting. It uses free monads, a novel technique for embedding a statically typed probabilistic programming language into a purely functional language, obtaining a syntax based on the Girly monad, and uses free applicative functors to be able to express statistical independence and explore its parallel nature. Notwithstanding the progress and studies shown, [Tobin \(2018\)](#) does not cope with the latest abstraction of [SAF](#) nor fills the gap on how they fit into a probabilistic context in order to benefit from their properties.

1.5 SUMMARY

The discrete probability distribution is a particular representation of probability distributions. A distribution is represented by a sampling space, i.e. an enumeration of both the support and associated probability mass at any point.

Discrete distributions are also instances of the Functor type-class, which means that you can take advantage of the `fmap` operator to map all values (the distribution domain) to others while keeping the distribution structure intact, i.e. maintaining the probabilities of each value.

The Applicative instance lets you apply pure functions to distributions. By taking advantage of the Applicative properties and laws, it is possible, for example, to combine two distributions to calculate their joint probability, if we know that they are independent from each other.

The Monad instance lets you chain distributions, giving the possibility of expressing the calculation of conditioned probability. As far as probabilistic programming is concerned, the monadic interface allows the programmer to benefit from syntactic sugar making it easier for him to write imperative style code.

The most prevalent abstractions in FP were analysed in order to understand the motivation and theory behind these and in which way do they relate to the probabilistic setting. It is concluded that the mathematical theoretic foundations are transversal to all abstractions seen and, in particular, CT is ubiquitous in programming and CS in general. There are cases where the need for abstraction comes from more practical contexts and a systematic and disciplinary study, recurring to sound mathematical frameworks, leads to the design of a correct and efficient solution.

Given this, what type of probabilistic interpretations or benefits are possible to achieve with SAF concerns the scope of this dissertation. After a detailed analysis and interpretation of the different abstractions found in the FP ecosystem, it is possible to outline several starting points in order to try and achieve the main goal of this thesis: proving that the SAF abstraction is useful in providing a more efficient solution than Monads to encode probabilities. The ability of static analysis and speculative execution of SAFs can prove to be very useful in optimising certain libraries, as was the case of the Haxl library (Marlow et al., 2014), Mokhov et al. (2019). On the other hand, the adoption of a less strong abstraction than the monadic one, but (infinitely) as expressive as this one, may prove to be of value in mitigating the performance constraints that the monadic interface imposes because of being inherently sequential.

At this point the reader already has all the background knowledge necessary to understand the scope of this master's dissertation and to be able to relate all the concepts that will eventually be addressed in the future. Nevertheless, this chapter will serve as a reference if any future reasoning needs to be made clear.

At the same time as the different key topics were introduced, it was made known the state of the art that belongs to each one, culminating in the presentation of the related and existing work in the area of functional probabilistic programming and how this dissertation stands out from the rest.

1.6 STRUCTURE OF THE DISSERTATION

Next chapter (2) will present the needed theoretic background needed to understand the domain problem at hands. A brief overview over the problems and their challenges that might arise, along

with a presentation of the work carried thus far is given in (3). Finally, prospects of future work can be seen in (4).

BACKGROUND

2.1 PATH TO PROBABILISTIC PROGRAMMING

This section shines a light through the path of probabilities and their foundations, all the way through statistics, culminating with its use in probabilistic programming. The objective is to give a good background refreshment and intuition to the reader in order to eliminate the need of resorting to heavy books.

I will draw on the great work of [Casella and Berger \(2001\)](#) and [Annis \(2005\)](#) to explain the next topics.

2.1.1 Set theory

The field of probability theory is the base in which all statistics was built, giving means to model populations, experiences, or almost everything else. Through these models, statisticians are able to draw inference about populations, inferences based on examination of only a part of the whole.

Just as statistics was built upon the foundations of probability theory, probability theory was built upon set theory. One of the main objectives of a statistician is to obtain conclusions about a population of objects by conducting an experiment, for which he needs to first identify all possible outcomes, the *sample space*.

Definition 2.1.1. The set S of all possible outcomes of an experience is called the *sample space*.

The next step, after the sample space is defined, will be to consider the collection of possible outcomes of an experience.

Definition 2.1.2. An *event* E , is any collection of possible results of an experience, i.e. any subset of S ($E \subseteq S$).

As the reader ought to know, there are several elementary operations on sets (or events):

Union: The union of two events, $A \cup B$, is the set of elements that belong to either A or B , or both:

$$A \cup B = \{x : x \in A \text{ or } x \in B\} \quad (1)$$

Intersection: The intersection of two events, $A \cap B$, is the set of elements that belong to both A and B :

$$A \cap B = \{x : x \in A \text{ and } x \in B\} \quad (2)$$

Complementation: The complement of an event A , A^c , is the set of all elements that are not in A :

$$A^c = \{x : x \notin A\} \quad (3)$$

This elementary operations can be combined and behave much like numbers:

Theorem 2.1.1.

1. <i>Commutativity</i>	$A \cup B = B \cup A$ $A \cap B = B \cap A$
2. <i>Associativity</i>	$A \cup (B \cup C) = (A \cup B) \cup C$ $A \cap (B \cap C) = (A \cap B) \cap C$
3. <i>Distributive Laws</i>	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
3. <i>DeMorgan's Laws</i>	$(A \cup B)^c = A^c \cap B^c$ $(A \cap B)^c = A^c \cup B^c$

I remit the reader to [Casella and Berger \(2001\)](#) for the proofs of these properties.

2.1.2 Basic probabilities and distributions

Probabilities come up a lot often in our daily lives. They are not only of use to statisticians, a good understanding of probability theory allows us to assess correct probabilities to everyday tasks and to benefit from the wise choices that are made because of them. Probability theory is also useful in the fields of science and engineering, for example the design of a nuclear reactor must be such that the escape of radioactivity into the environment is an extremely rare event. So, using probability theory as a tool to deal with uncertainties, the reactor can be designed to ensure that an unacceptable amount of radiation will escape once in a billion years.

What probabilities are

When an experience is made, its realisation results on an outcome that is a subset of the sample space. If the experiment is repeated multiple times the result might vary in each repetition, or not. This "frequency of occurrence" can be seen as a *probability*. If the result of an experience can be described probabilistically its half way there to analyse the experience statistically. However, this "frequency of occurrence" is just one of the many interpretations of what is a probability. Another possible interpretation is a more subjective one, which is the one where a probability is the belief of a chance of an event occurring.

For each event A in the sample space S we want to associate with A a number between $[0, 1]$, this number will be called the probability of A , denoted $P(A)$. The domain of P , intuitively is the set of all subsets of S , this set is called a *sigma algebra*, denoted by \mathcal{B} .

Definition 2.1.3. A collection of subsets of S is a sigma algebra, \mathcal{B} if it satisfies the following properties:

1. $\emptyset \in \mathcal{B}$ (the empty set is an element of \mathcal{B})
2. If $A \in \mathcal{B}$, then $A^c \in \mathcal{B}$ (\mathcal{B} is closed under complementation)
3. If $A_1, A_2, \dots \in \mathcal{B}$, then $\bigcup_{i=1}^{\infty} A_i \in \mathcal{B}$ (\mathcal{B} is closed under countable unions)

Example 2.1.3.1. (Sigma algebras) If S has n elements, then \mathcal{B} has 2^n elements. For example, if $S = \{1, 2, 3\}$, then \mathcal{B} is the collection of $2^3 = 8$ sets:

$$\begin{array}{lll} \{1\} & \{1, 2\} & \{1, 2, 3\} \\ \{2\} & \{1, 3\} & \{\emptyset\} \\ \{3\} & \{2, 3\} & \end{array}$$

If S is uncountable ($S = (-\infty, +\infty)$), then \mathcal{B} is the set that contains all sets of the form:

$$[a, b] \quad [a, b) \quad (a, b] \quad (a, b)$$

Given this, we can now define $P(\cdot)$ as a function from $\mathcal{B} \rightarrow [0, 1]$, this probability measure must assign to each event A , a probability $P(A)$ and abide the following properties:

Definition 2.1.4. Given a sample space S and an associated sigma algebra \mathcal{B} , a probability function P satisfies:

1. $P(A) \in [0, 1]$, for all $A \in \mathcal{B}$
2. $P(\emptyset) = 0$ (i.e. if A is the empty set, then $P(A) = 0$)
3. $P(S) = 1$ (i.e. if A is the entire sample space, then $P(A) = 1$)
4. P is *countably additive*, meaning that if, A_1, A_2, \dots is a finite or countable sequence of *disjoint* events, then:

$$P(A_1 \cup A_2 \cup \dots) = P(A_1) + P(A_2) + \dots$$

This properties satisfy the Axioms of Probability (or the Kolmogorov Axioms), and every function that satisfies them is called a probability function. The first 3 axioms are pretty intuitive and easy to understand, however, the fourth one is more subtle and is an implication of the third Kolmogorov Axiom, called the *Axiom of Countable Additivity* which says that we can calculate probabilities of complicated events by adding up the probabilities of smaller events, provided those smaller events are disjoint and together contain the entire complicated event.

Calculus of probabilities

From the Axioms of Probabilities we can build up many properties of the probability function. Properties are quite useful for calculations of more complex probabilities. The additivity property automatically implies certain basic properties that are true for any probability model at all.

Taking a look at A and A^c we can see that they are always disjoint, and their union is the entire sample space: $A \cup A^c = S$. By the additivity property we have $P(A \cup A^c) = P(A) + P(A^c) = P(S)$, and since we know $P(S) = 1$, then $P(A \cup A^c) = 1$ or:

$$P(A^c) = 1 - P(A) \quad (4)$$

In other words, the probability of an event not happening is equal to one minus the probability of an event happening.

We can already see how this property can be useful, and in fact there are a lot others that follow from other properties.

Theorem 2.1.2. Let A_1, A_2, \dots be events that form a partition of the sample space S . Let B be any event, then:

$$P(B) = P(A_1 \cap B) + P(A_2 \cap B) + \dots$$

Theorem 2.1.3. Let A and B be two events such that $B \subseteq A$. Then:

$$P(A) = P(B) + P(A \cap B^c)$$

And since we always have $P(A \cap B^c) \geq 0$, we can conclude:

Corollary 2.1.3.1. (Monotonicity) Let A and B be two events such that $B \subseteq A$. Then:

$$P(A) \geq P(B)$$

And rearranging 2.1.3 we obtain:

Corollary 2.1.3.2. Let A and B be two events such that $B \subseteq A$. Then:

$$P(A \cap B^c) = P(A) - P(B)$$

More generally, if the $B \subseteq A$ constraint is lifted, we have the following property:

Theorem 2.1.4. (Principle of inclusion–exclusion, two-event version) Let A and B be two events. Then:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \quad (5)$$

Property 2.1.4 gives an useful inequality for the probability of an intersection. Since $P(A \cup B) \leq 1$, we have from 5, after some rearranging:

$$P(A \cap B) \geq P(A) + P(B) - 1 \quad (6)$$

This inequality is a special case of what is known to be *Bonferroni's Inequality*. Given this, we can affirm that probabilities always satisfy the basic properties of total probability, subadditivity, and monotonicity. And once again I remit the reader to check the proofs in [Casella and Berger \(2001\)](#) or [Annis \(2005\)](#).

Counting and enumerating outcomes

Counting methods are used to build assessments of probabilities to finite sample spaces. Counting problems, in general, appear to be complicated, and frequently there are a lot of constraints to take into account. One form of solving this problem is to break the problem into several easy to count tasks and use some combination rules.

Theorem 2.1.5. (Fundamental Theorem of Counting) If a job consists in k tasks, in which the i -th task can be done in n_i ways, $i = 1, \dots, k$, then the whole job can be done in $n_1 \times n_2 \times \dots \times n_k$ ways.

Although theorem 2.1.5 is a good place to start exist situations where, usually, there are more aspects to consider. In a lottery, the first number can be chosen in 44 ways and the second in 43 ways, making a total of $44 \times 43 = 1892$ ways. However, if the player could pick the same number twice then the first two numbers could be picked in $44 \times 44 = 1936$ ways. This distinction is between counting *with replacement* and counting *without replacement*. There is a second very important aspect in any counting problem: whether or not the *order* of the tasks is important. Taking these considerations into account, its possible to construct a 2×2 table of possibilities.

Back to the lottery example, we can express all the ways a player can pick 6 numbers out of 44, under the four possible case:

- *ordered, without replacement* - Following 2.1.5 the first number can be picked in 44 ways, the second in 43, etc. So, there are:

$$44 \times 43 \times 42 \times 41 \times 40 \times 39 = \frac{44!}{38!} = 5082517440$$

- *ordered, with replacement* - Each number can be picked in 44 ways, so:

$$44 \times 44 \times 44 \times 44 \times 44 \times 44 = 44^6 = 7256313856$$

- *unordered, without replacement* - Since we know how many ways we can pick the numbers if the order is taken into account, then we just need to divide the redundant orderings. Following 2.1.5, 6 numbers can be rearranged in $6!$, so:

$$\frac{44 \times 43 \times 42 \times 41 \times 40 \times 39}{6 \times 5 \times 4 \times 3 \times 2 \times 1} = \frac{44!}{6!38!} = 7059052$$

This last form of counting is so frequent that there is a special notation for it:

Definition 2.1.5. For non-negative numbers, n and r , where $n \geq r$, the symbol $\binom{n}{r}$, read n choose r , as:

$$\binom{n}{r} = \frac{n!}{r!(n-r)!}$$

- *unordered, with replacement* - This is the most difficult case to count. To count this case, it is easier to think of placing 6 markers into 44 boxes. Someone noticed (Feller, 1971) that all we need to keep track of is the arrangement of the markers and the walls of the boxes. Therefore, we have 43 (walls) + 6 markers = 49 objects which can be combined in 49! ways. We still need to divide the redundant orderings, so:

$$\frac{49!}{6!43!} = 13983816$$

We can summarise these situations in the following table:

	Without replacement	With replacement
Ordered	$\frac{n!}{(n-r)!}$	n^r
Unordered	$\binom{n}{r}$	$\binom{n+r-1}{r}$

Table 1: Number of possible arrangements of size r from n objects

This counting techniques are useful when the sample space is finite and all outcomes in S are equally probable. With this being said, the probability of an event can be calculated counting the number of its possible outcomes. For $S = \{S_1, \dots, S_n\}$ saying that all the elements are equally probable means $P(\{s_i\}) = \frac{1}{N}$. From the Axiom of Countable Additivity, for any event A :

$$P(A) = \frac{\# \text{ of elements in } A}{\# \text{ of elements in } S}$$

Conditional probability and independence

Every probabilities dealt with until now were unconditional. There are several situations where it is desirable to *update the sample space based on new information*, that is to calculate conditional probabilities.

Definition 2.1.6. If A and B are events in S , and $P(B) > 0$, then the conditional probability of A given B , is:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

It is worth noting that what happens in a conditional probability calculation is that B becomes the sample space ($P(B|B) = 1$). The intuition is that the event B will occur a fraction $P(B)$ of the time and, both A and B will occur a fraction $P(A \cap B)$ of the time; so the ration $P(A \cap B)/P(B)$ gives the *proportion* of times when B occurs, A also occurs.

Rearranging 2.1.6 gives a useful way to calculate intersections:

$$P(A \cap B) = P(A|B)P(B) \tag{7}$$

By symmetry of 7 and equating both right-hand sides of the symmetry equations we get:

Theorem 2.1.6. (Bayes' Theorem) Let A and B be two events with positive probabilities each:

$$P(A|B) = P(B|A) \frac{P(A)}{P(B)}$$

There might be cases where an event B does not have any impact on the probability of other event A : $P(A|B) = P(A)$. If this holds then by using Bayes' rule 2.1.6:

$$P(B|A) = P(A|B) \frac{P(B)}{P(A)} = P(A) \frac{P(B)}{P(A)} = P(B)$$

2.2 (LINEAR) ALGEBRA OF PROGRAMMING

LAoP is a quantitative extension to the well-known *Algebra of Programming (AoP)* discipline that treats binary functions as relations. This extension generalises relations to matrices and sees them as arrows, i.e. morphisms typed by the dimensions of the matrix. This achievement is important as it paves the way for a categorical approach which is the starting point for the development of an advanced type system for linear algebra and its operators.

Central to the approach is the notion of a biproduct, which merges categorical products and coproducts into a single construction. Careful analysis of the biproduct axioms as a system of equations provides a rich palette of constructs for building matrices from smaller ones, regarded as blocks.

By regarding an matrix as a morphism between two dimensions, matrix multiplication becomes simple matrix composition:

$$\begin{array}{ccccc} m & \xleftarrow{A} & n & \xleftarrow{B} & q \\ & \searrow & & \swarrow & \\ & & C=A \cdot B & & \end{array}$$

A more detailed overview of the topic can be seen in the work done by [Macedo \(2012\)](#).

2.2.1 Structure of the category of Matrices

Vectors and linear maps

Wherever one of the dimensions of the matrix is $\mathbf{1}$ the matrix is referred as a *vector*. In more detail, a matrix of type $m \leftarrow \mathbf{1}$ is a column vector, and $\mathbf{1} \leftarrow m$ is a row vector.

Identity

The identity matrix as type $n \leftarrow n$. For every object n in the matrix category there must be a morphism of this type, which will be denoted by $n \xleftarrow{id_n} n$

Transposed matrix

The transposition operator changes the matrix shape by swapping rows with columns. Type-wise, this means converting an arrow of type $n \xleftarrow{A} m$ into an arrow of type $m \xleftarrow{A^\circ} n$.

Bilinearity

Given two matrices it is possible to add them up entry-wise, leading to $A + B$, where 0 is the neutral matrix (filled with 0's). This unit matrix works exactly as one is expecting with respect to matrix composition:

$$\begin{aligned} A + 0 &= A = 0 + A \\ A \cdot 0 &= A = 0 \cdot A \end{aligned}$$

In fact, matrices form an Abelian category:

$$\begin{aligned} A \cdot (B + C) &= A \cdot B + A \cdot C \\ (B + C) \cdot A &= B \cdot A + C \cdot A \end{aligned}$$

2.2.2 Biproducts

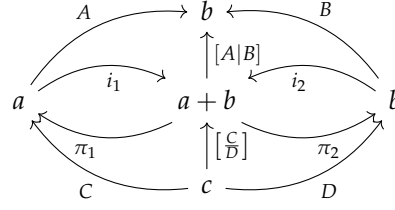
In an abelian category, a biproduct diagram for the objects m, n is a diagram of the following shape:

$$\begin{array}{ccccc} & \xleftarrow{\pi_1} & & \xleftarrow{\pi_2} & \\ m & & r & & n \\ & \xrightarrow{i_1} & & \xrightarrow{i_2} & \end{array}$$

whose arrows π_1, π_2, i_1, i_2 , satisfy the following laws:

$$\begin{aligned} \pi_1 \cdot i_1 &= id_m \\ \pi_2 \cdot i_2 &= id_n \\ i_1 \cdot \pi_1 + i_2 \cdot \pi_2 &= id_r \\ i_1 \cdot i_2 &= 0 \\ \pi_2 \cdot i_1 &= 0 \end{aligned}$$

How do biproducts relate to products and coproducts in the category? The diagram and definitions below depict how products and coproducts arise from biproducts (the product diagram is in the lower half; the upper half is the coproduct one):



By analogy with the [AoP](#), expressions $[A|B]$ and $\begin{bmatrix} A \\ B \end{bmatrix}$ will be read 'A junc B' and 'C split D', respectively. These combinators purports the effect of putting matrices side by side and stacked on top of each other, respectively. The rich algebra of such combinators of which the most useful laws follow, where capital letters M, N, etc. denote suitably typed matrices (the types, i.e. dimensions, involved in each equality can be inferred by drawing the corresponding diagram):

- Fusion laws:

$$P \cdot [A|B] = [P \cdot A | P \cdot B] \quad (8)$$

$$\begin{bmatrix} A \\ B \end{bmatrix} \cdot P = \begin{bmatrix} A \cdot P \\ B \cdot P \end{bmatrix} \quad (9)$$

- Divide and conquer:

$$[A|B] \cdot \begin{bmatrix} C \\ D \end{bmatrix} = A \cdot C + B \cdot D \quad (10)$$

- Converse duality:

$$[A|B]^\circ = \begin{bmatrix} A^\circ \\ B^\circ \end{bmatrix} \quad (11)$$

- Exchange ("Abide") law:

$$\left[\begin{bmatrix} A \\ C \end{bmatrix} \mid \begin{bmatrix} B \\ D \end{bmatrix} \right] = \begin{bmatrix} [A|B] \\ [C|D] \end{bmatrix} \quad (12)$$

2.2.3 Biproduct functors

As one could expect, as it is true in [AoP](#) it is also possible to obtain functors out of the biproducts presented above, and by using the same recipe we obtain the coproduct bifunctor that joins two matrices:

$$A \oplus B = [i_1 \cdot A | i_2 \cdot B]$$

$$\begin{array}{ccc} k & k+j & j \\ \uparrow A & \uparrow A \oplus B & \uparrow B \\ n & n+m & m \end{array}$$

The tensor product in the category of matrices is the known Kronecker product. In the [LAoP](#) context, this bifunctor can be expressed in terms of the Khatri Rao product and the latter by the Hadamard, Schur matrix multiplication:

$$\begin{array}{ccc}
 k & k \times j & j \\
 \uparrow A & \uparrow A \otimes B & \uparrow B \\
 n & n \times m & m
 \end{array}$$

THE PROBLEM AND ITS CHALLENGES

3.1 PROBABILISTIC INTERPRETATION OF SELECTIVE FUNCTORS

In section 1.3 was presented the most well-known abstractions in functional programming as well as a probabilistic interpretation of them and how this translates into the context of probabilistic programming. The analysis of how the *Giry monad* (Giry, 1982) is a *Functor* and an *Applicative* made by Tobin (2018) brings us closer to a possible probabilistic interpretation of *SAF*, observation induced by the hierarchy of abstractions. In addition, the relationship between *SAF* and *Arrows* made by Mokhov et al. (2019) and the fact that *CT* had a very strong presence in *Arrows* design and semantics has also prompted the challenge of finding how *Arrows*' fundamentals and generality can be useful in finding the answer.

Thus, one of the problems is to find the probabilistic definition of *SAFs* and its meaning, in the probabilistic programming environment. That being said, it is assumed that due to the aforementioned equivalence with *Arrows*, much of their definition can be reused as a step closer to finding the answer to the problem posed in the next section.

3.2 INEFFICIENT PROBABILITY ENCODINGS

Roughly, there are two ways to model probabilistic distributions. In light of the work mentioned in section 1.4, it is possible to opt for an extensive representation of a distribution, where all chance-value pairs are stored and any structural manipulation is achieved by modifying all the pairs. This approach has the advantage of being possible to get the exact probability of any type of event, however any slightly complex problem can lead to an explosion of states within the distribution, which greatly affects performance. With this in mind, another less rigorous method of estimating probabilities is to measure them using less precise yet quicker and more effective inference algorithms instead of always measuring the exact probabilities for all values.

Modelling a simple program that calculates the probability that a particular event will occur in N throws of a die, using the exhaustive approach quickly becomes unfeasible for a relatively small N . Modelling a complex, critical program that calculates the likelihood of two planes crashing using a non-exhaustive approach can lead to dangerous situations, if the accuracy of the results is not the desired one. It is this trade-off that is usually explored in probabilistic programming research and depends largely on the type of encoding that is done on distributions.

The challenge then is to create an encoding capable of minimising the distance between the two most common approaches to encoding probabilistic distributions, seeking one that takes advantage of

the abstraction of Selective Functors and benefits from its properties of static analysis or speculative execution.

3.3 RESEARCH METHODOLOGY

Given the above-mentioned problems and challenges, the research strategy is to tackle each one consecutively, as some of the issues are dependent on others.

With this being said, regarding the problem of finding the probabilistic interpretation of SAFs, a type safe matrix representation and manipulation library is being developed in Haskell in order to encode the basic LAoP combinators. This library will be important to help building intuition and exploring the functorial, applicative and monadic structure of probability distributions represented as matrices. It should also implement the Arrow type-class since this instance is tightly related with the SAF abstraction. If one is capable of implementing such a library, one can obtain the ArrowChoice type-class implementation and extract the functional intuition behind the select operator, hoping to obtain the type of properties and benefits from such an abstraction.

After gaining an understanding of SAFs's probabilistic capabilities, a study will be conducted to determine in which of the two possible approaches SAF can be more impactful, either by reusing the former type safe LAoP library or by contributing to an already established one such as Erwig and Kollmansberger (2006) or monad-bayes from Ścibior et al. (2018). A set of case studies will be used in order to benchmark the changes made.

With this being said, the following research plan will be followed:

Task	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun	Jul
Implement the library proof of concept			×	×						
Encode probabilistic computations using Selective Functors				×	×					
Study sampling algorithms					×	×				
Case studies / benchmarking					×	×	×			
Write the dissertation							×	×	×	×

3.4 WORK CARRIED OUT THUS FAR

This section gives a detailed overview of the current work and what are the prospects for the future.

3.4.1 Probabilistic interpretation of Arrows

The AoP (Bird and de Moor, 1997) is known as the discipline of programming from specifications in a calculational and point-free manner and as a case where the use of CT offers an economy on definitions and proofs of correction. Bird and de Moor (1997) show how CT can be used to define the basic building blocks of datatypes found in FP and program derivation. The lesson from AoP, where functions are treated as a special case of relations, is that by changing the category (from *Fun* to *Rel*) the expressive power increases. Relations are essentially non-deterministic and are capable of specifying a wider variety of problems by making rich operators, like converse and division, universally available. Other reasons to this change are that more structure is revealed, opportunities for generalisation are unveiled and the arrangement of particular proofs becomes simpler. "Keep the

definition, change category” is a slogan that well summarises the lesson that Bird and de Moor (1997) pass and emphasises the gradual compositionality, as seen in the case of relational algebra.

So how is it possible to take advantage of this lesson in a probabilistic context? Oliveira and Miraldo (2016) guide us in the direction of LAoP, inspired by the work of Macedo (2012) and Oliveira (2012). LAoP generalises relations and functions treating them as Boolean matrices and in turn consider these as *arrows*. If instead of seeing matrices of only 0's and 1's, we restrict them to being left-stochastic, where the values of each column sum to 1, it would be possible to express numerous probabilistic extensions to standard AoP combinators and would help keeping the convoluted probability notation under control.

Given this, probabilistically speaking, left-stochastic matrices are seen as *Arrows* and can be written as $n \xrightarrow{M} m$ to denote that matrix M is of type $n \rightarrow m$ (n columns, m rows). Using this notation matrix multiplication can be understood as arrow composition, therefore forming the category of matrices, where objects are numeric dimensions and morphisms are the matrices themselves. Since all arrows represent left-stochastic matrices, a simple distribution $P(A)$ can be seen of a matrix of type $1 \rightarrow m$, which represents a left-stochastic column vector. Statistical independence $P(B|A) = P(A)P(B)$ can be calculated by the probabilistic pairing, also known as the Khatri-Rao matrix product (Macedo, 2012; Murta and Oliveira, 2013). Objects in the category of matrices may be generalised to arbitrary denumerable types (A, B) . By performing this generalisation, probabilistic functions $A \xrightarrow{f} \mathcal{D}B$ are viewed as matrices of type $A \rightarrow B$, enabling us to express conditional probability calculation $P(B|A)$ in the form of probabilistic function application. It is worth noting that by using just the monadic interface it would only be possible to reason about conditional probabilities by recurring to the bind ($>>=$) operator, which convolutes probabilistic reasoning. However, by adopting the LAoP transition, probabilistic function (Kleisli) composition becomes simply matrix composition (Oliveira, 2012).

This probabilistic interpretation of Arrows puts the research for the probabilistic interpretation of SAF one step closer. By using what has been learned so far it should be possible to encode matrices around sound mathematical abstractions and take advantage of what best they have to offer. What's SAF's take on the linear approach to the AoP and how will it fit in the probabilistic setting is the answer that it is hoped to find.

3.4.2 Type safe Linear Algebra of Programming matrix library

Inspired by the LAoP and pursuing a probabilistic interpretation for SAF, an attempt has been made to design a matrix-representing data structure. The discipline LAoP has an inductive approach to calculating the different combinators that define it, since it is based on the biproducts Junc and Split. Therefore, a *Generalised Algebraic Datatype (GADT)* was used as an attempt to achieve a strongly typed data type capable of inferring matrix dimensions. The GADT was designed as follows, taking advantage of the TypeLits library, where ValidDimensions is a constraint that ensures dimensions are greater than 0.

```
data Matrix e (c :: Nat) (r :: Nat) where
  Junc :: (ValidDimensions m p, ValidDimensions n p)
    => Matrix e m p
```

```

-> Matrix e n p
-> Matrix e (m + n) p
Split :: (ValidDimensions p m, ValidDimensions p n)
=> Matrix e p m
-> Matrix e p n
-> Matrix e p (m + n)
One :: e -> Matrix e 1 1

```

Listing 3.1: Inductive matrix definition

Based on the [LAoP](#) biproduct approach, this inductive data type can correctly represent any type of matrix and infer dimensions. It does, however, create some challenges in implementing such construction and manipulation functions as *Glasgow Haskell Compiler (GHC)* has difficulty in properly inferring the right types when matching patterns. A simple example, such as matrix transposition is easily implemented, but other cases, such as the entry-wise addition of two matrices, become impossible to implement since two matrices with the same dimensions internally can be represented with a different combination of Juncs and Splits. One solution to this problem would be to find a way to ensure that all matrices were built according to a convention (either Junc of Splits or Split of Juncs) however, *GHC*'s type system could not be sure that the matrices actually followed such a convention and made pattern matching impossible in some cases.

To address this challenge and attempt to achieve *SAF*'s probabilistic interpretation / intuition, the built library is now just a wrapper around an existing library's matrix data form. The library of choice was *HMatrix* ([hma](#)) because it is more successful and is more commonly used by industry and the community.

```

newtype Matrix e (c :: Nat) (r :: Nat) = M {unMatrix :: HM.Matrix e}

```

Listing 3.2: Type safe wrapper around HMatrix

With this solution it was possible to implement several [LAoP](#) combinators, however, when using it one is at the mercy of the implementation used by the host library and the possibility of obtaining a structure that benefits from the properties of *SAFs* is lost. Yet, this approach makes a possible answer closer.

As stated in the previous subsection, representing distributions as stochastic arrays, and these in turn as arrows, allows us to implement the Arrow instance in the data type shown in listing 3.2. As explained in the section 1.3, probability distributions are able to satisfy Functor, Applicative, and Monad instances. Nevertheless, due to constraints necessary for compile-time type checking to be performed, and due to the fact that matrix dimensions are encoded as inhabited data types, it is not possible to implement instances for the spoken interfaces. Despite this, an equivalent version of the functions of each instance can be implemented as shown in the listing below:

```

— | Monoidal/Applicative instance
khatri :: forall e m p q .
  ( ... )

```

```

=> Matrix e m p
-> Matrix e m q
-> Matrix e m (p * q)

— | Monad instance
comp :: ( ... )
=> Matrix e p m
-> Matrix e n p
-> Matrix e n m

— | Arrow instance
fromF :: forall c r a b e .
  ( ... )
=> (a -> b)
-> Matrix e c r

```

Listing 3.3: Interface equivalent function implementations

For the Applicative instance, the Khatri Rao product is offered, taking advantage of the monoidal nature that characterises this abstraction. In the case of Monads, matrix composition is the equivalent operation of bind as seen in the previous subsection. Finally for the Arrow instance, the fundamental operation is to transform (lift) a function into its matrix representation. It should be noted that all of these operators have associated constraints which, for reasons of space saving are presented in appendix A, where you can see the full implementation.

Having said that, it is possible to model basic probabilistic problems such as the Monty Hall problem by having an LAoP implementation that is still in a proof of concept phase:

```

— Monty Hall Problem
data Outcome = Win | Lose
  deriving (Bounded, Enum, Eq, Show)

switch :: Outcome -> Outcome
switch Win = Lose
switch Lose = Win

firstChoice :: Dist Outcome
firstChoice = choose (1/3)

secondChoice :: Matrix Double 2 2
secondChoice = fromF switch

main :: IO ()
main = do
  print (p1 `comp` secondChoice `comp` firstChoice :: Matrix Double 1
    1)

```

```
{-
Output:
(1><1)
[ 0.6666666666666666 ]
-}
```

Listing 3.4: LAoP Monty Hall Problem

3.4.3 Probabilistic Interpretation of Selective Functors

In the most recent work by Mokhov et al. (2019) SAFs are said to provide the missing counterpart for ArrowChoice in the functor hierarchy as demonstrated by the following instance:

```
instance ArrowChoice a => Selective (ArrowMonad a) where
  select (ArrowMonad x) y = ArrowMonad $ x >>> (toArrow y ||| returnA)

toArrow :: Arrow a => ArrowMonad a (i -> o) -> a i o
toArrow (ArrowMonad f) = arr (\x -> ((), x)) >>> first f >>> arr (uncurry
  ($))
```

Listing 3.5: Selective ArrowMonad instance

This instance highlights the relationship of Arrows with SAFs. Given the results shown in the previous subsection, it is possible to use an implementation similar to that shown in listing 3.5 and similarly obtain a Selective instance for the data type Matrix (3.2). Although, due to constraints imposed on the types of the matrix dimensions, and because type level naturals can not be habited by constructors at the term level, it is not possible to implement an official instance. It is possible however, to write the select operator at the expense of operators equivalent to those used above:

```
select :: ( ... ) => Matrix e n (m1 + m2) -> Matrix e m1 m2 -> Matrix e n
  m2
select m y = (y ||| id) `comp` m

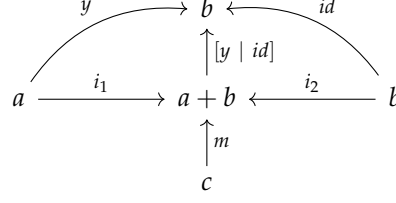
— (|||) :: ( ... ) => Matrix e m p -> Matrix e n p -> Matrix e (m + n) p
```

Listing 3.6: LAoP Selective instance

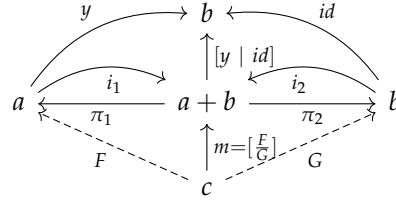
Note that a toArrow function is required to type check in the listing 3.5. Sadly, it would be impractical to have an equivalent toMatrix function in order to match the same signature, provided that an Enum (a..b) instance would be necessary and that is impossible to achieve in Haskell at the moment. Because of this, the signature of select has been adapted.

As fascinating as Arrows' relationship with SAFs is, and as much as it is possible to write an instance analogously for the same interface, this tells us little or nothing about its semantics. It is more or less clear that some form of conditional is conveyed, but its implementation and interaction

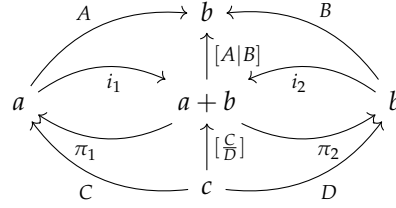
with the other combinators is difficult to imagine. The following type diagram was designed to find a more concrete explanation:



A more detailed observation:

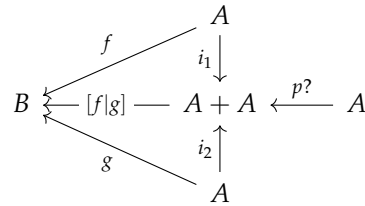


Generalising:



This last diagram is the same as seen by [Macedo \(2012\)](#) and defines exactly the biproducts *Junc* and *Split*. This diagram highlights several properties of this biproduct such as the famous divide-and-conquer law $[A \mid B] \cdot [\frac{C}{D}] = A \cdot C + B \cdot D$.

Another known combinator of the [AoP](#) discipline is McCarthy's conditional ([Bird and de Moor, 1997](#)), whose probabilistic version was studied by [Oliveira \(2012\)](#), and has the following diagram:



This probabilistic version of *if-then-else* is denoted by $p \rightarrow f, g$, where $p?$ is the predicate defined by the coreflexive matrix:

$$A + A \xleftarrow{p?} A = \left[\frac{\Phi_p}{\Phi_{-p}} \right]$$

Looking closely at the diagram one can see some resemblance to what one gets from definition 3.6, and indeed McCarthy’s conditional exploits the same conditional nature that is intrinsic to SAF.

What can we then conclude as the probabilistic interpretation of SAFs? So far, we can conclude that if we view f (1.8) as a distribution, the select operator, in probabilistic programming, has the ability to condition the random variable and branch out the program in two different ways. This operator, although of a more generic type than McCarthy’s conditional, expresses the divide-and-conquer law present in block-matrix calculus and is capable of exploiting parallelism, indicating that this abstraction does indeed become useful not only in terms of the reasoning power it offers, but also in terms of the performance gains that are theoretically achievable.

3.4.4 Type safe inductive matrix definition

The type-safe matrix library listed in subsection 3.4.2 only provided a more intuitive understanding of SAF’s semantic and probabilistic interpretation. It demonstrated the kinds of benefits that by taking advantage of such abstraction are theoretically possible. This gain is not visible in the presented library (A) as it is only a wrapper grounded on a Vector data type based matrix definition. The first attempt to design an inductive approach to matrix definition à la LAoP tried to take advantage of type-level naturals to represent matrix dimensions. Further research showed that GHC could only handle type level naturals up to a point, from which it was impossible to continue further.

In order to profit from all the results of the research done so far and from efficient LAoP-inspired matrix encoding, whose gains are visible and promising by construction, many different approaches and solutions were tried. Ultimately, type-level naturals were abandoned and simple data types were used to substitute them and the following encoding was achieved, by us:

```
data Matrix e cols rows where
  Empty :: Matrix e Void Void
  One :: e -> Matrix e () ()
  Junc :: Matrix e a rows -> Matrix e b rows -> Matrix e (Either a b) rows
  Split :: Matrix e cols a -> Matrix e cols b -> Matrix e cols (Either a b)

— | Type family that computes the cardinality of a given type dimension.
—
— It can also count the cardinality of custom types that implement the
— ‘Generic’ instance.
type family Count (d :: Type) :: Nat where
  Count (Natural n m) = (m - n) + 1
  Count (Either a b) = (+) (Count a) (Count b)
  Count (a, b) = (*) (Count a) (Count b)
  Count (a -> b) = (^) (Count b) (Count a)
— Generics
  Count (M1 ` ` f p) = Count (f p)
  Count (K1 ` ` `) = 1
  Count (V1 `) = 0
  Count (U1 `) = 1
  Count ((:*) a b p) = Count (a p) * Count (b p)
```

```

Count ((+::) a b p) = Count (a p) + Count (b p)
Count d = Count (Rep d R)

— | Type family that computes of a given type dimension from a given natural
—
— Thanks to Li-Yao Xia this type family is super fast.
type family FromNat (n :: Nat) :: Type where
  FromNat o = Void
  FromNat 1 = ()
  FromNat n = FromNat' (Mod n 2 == o) (FromNat (Div n 2))

type family FromNat' (b :: Bool) (m :: Type) :: Type where
  FromNat' 'True m = Either m m
  FromNat' 'False m = Either () (Either m m)

```

Listing 3.7: Inductive Matrix definition

This solution was built on the notion that algebraic data types are isomorphic to their cardinality, i.e $\text{Void} \cong 0$, $() \cong 1$, $\text{Either } a \ b \cong |a| + |b|$, etc. . Furthermore, this [GADT](#) guarantees that a matrix will always have valid dimensions. The type families `Count` and `FromNat` take advantage of this isomorphism and provide a conversion mechanism from and to data types/type-level naturals. Surprisingly enough, [GHC](#) does not complain while pattern matching with this definition, hence, more complex functions were possible to implement via pattern matching.

Here’s an example of how [LAoP](#) enables to write concise, correct and efficient code by exploring the biproducts divide-and-conquer, fusion and ‘abide’ laws:

```

comp :: (Num e) => Matrix e cr rows -> Matrix e cols cr -> Matrix e cols rows
comp Empty Empty          = Empty
comp (One a) (One b)      = One (a * b)
comp (Junc a b) (Split c d) = comp a c + comp b d          — Divide-and-conquer law
comp (Split a b) c        = Split (comp a c) (comp b c) — Split fusion law
comp c (Junc a b)         = Junc (comp c a) (comp c b) — Junc fusion law

abideJS :: Matrix e cols rows -> Matrix e cols rows
abideJS (Junc (Split a c) (Split b d)) = Split (Junc (abideJS a) (abideJS b)) (
  Junc (abideJS c) (abideJS d)) — Junc-Split abide law
abideJS Empty          = Empty
abideJS (One e)         = (One e)
abideJS (Junc a b)      = Junc (abideJS a) (abideJS b)
abideJS (Split a b)     = Split (abideJS a) (abideJS b)

```

Listing 3.8: Matrix composition and abiding functions

It is very straightforward to see the benefits of adopting an inductive approach to encode matrices however, it still has its drawbacks. For instance, more complex operators like the Khatri-Rao product

(also known as matrix pairing), which should return a matrix of type $m \times n \leftarrow c$, and its projections are limited to return matrices that are built at the expense of Eithers. Moreover, since some important matrices, like the identity matrix, have constraints associated with the dimension types it is not possible to implement instances of abstractions such as Arrows or Selective. However, it is possible to write a version of the `select` and McCarthy's conditional operators that indeed are capable of exploiting all of the theoretic gains mentioned in the previous section, as will be shown.

Given this, this new data type compared to the previous one has much more advantages, such as:

- An inductive definition, which enables writing matrix manipulation functions in a much more elegant and calculational manner;
- Statically typed dimensions;
- Polymorphic data type dimensions;
- Polymorphic matrix content;
- Fast type natural conversion via `FromNat` type family;
- Better type inference;
- Matrix 'Junc' and 'Split'-ing in $O(1)$;
- Matrix composition takes advantage of divide-and-conquer and fusion laws.

Unfortunately, this approach does not solve the issue of type dimensions being in some way constrained making it impossible to write Arrow instances, for example. Type inference is not perfect, when it comes to infer the types of matrices which dimensions are computed using type level naturals multiplication, the compiler needs type annotations in order to succeed.

Two different data types were built on top of the inductive definition presented in this section. These data types consist of wrappers around the `Matrix` type and with the help of the type families presented in listing (3.7) it is possible to provide a much more user friendly interface:

```
newtype Matrix e (cols :: Nat) (rows :: Nat) = M (I.Matrix e (I.FromNat cols) (I.
  FromNat rows))
```

Listing 3.9: Dimensions are type level naturals

```
newtype Matrix e (cols :: Type) (rows :: Type) = M (I.Matrix e (I.Normalize cols)
  (I.Normalize rows))
```

Listing 3.10: Dimensions are arbitrary data types

These were created with user experience in mind. Actually the last one captures the type generalisation depicted by Oliveira (2012). In short objects in categories of matrices can be generalised from numeric dimensions $(n, m \in \mathbb{N}_0)$ to arbitrary denumerable types (A, B) , taking disjoint union $A + B$ for $m + n$, Cartesian product $A \times B$ for $m \times n$, unit type `1` for number 1, the empty set \emptyset for 0, etc.

3.4.5 The probability distribution matrix and the Selective abstraction

If the sum of each column of a matrix is equal to 1 as has already been said, it is called a stochastic matrix, and if this matrix has only one column, it is called a distribution. An exhaustive approach is assumed when using matrices to represent distributions, and these are notorious for their related performance issues (Ścibior et al., 2015; Kidd, 2007). It makes sense to build a probabilistic programming library that performs better than the other exhaustive probabilistic programming libraries by taking advantage of the inductive matrix encoding and the LAoP discipline. So, in order to address the central topic of this master's thesis and explore how the Selective interface can be used more efficiently than Monads in handling probabilistic distributions, the `Dist` data type was developed. This data type is just a newtype wrapper around type `Matrix Prob () a`.

```
— | Type synonym for probability value
type Prob = Double

— | Newtype wrapper for column vector matrices. This represents a probability
— distribution.
newtype Dist a = D (Matrix Prob () a)
```

Listing 3.11: `Dist` type alias

Now if we take this type and match it against the `SAF` `select` operator function signature we can pretty much recover all the conditioning capabilities that are inherent to `SAF` and probabilistic choice:

```
— Selective 'select' operator
select :: ( ... ) => Dist (Either a b) -> Matrix Prob a b -> Dist b
selectD (D d) m = D (junc m identity `comp` d)

— McCarthy's Conditional
cond :: ( ... ) => (a -> Bool) -> Dist b -> Dist b -> Dist b
cond p (D f) (D g) = D (junc f g `comp` grd p)

— == junc f g . split (corr p) (corr (not . p))
— == f . (corr p) + g . (corr (not Prelude.. p))
— (Parallelism via divide-and-conquer)

grd :: ( ... ) => (a -> Bool) -> Matrix e a (Either a a)
grd f = split (corr f) (corr (not Prelude.. f))

corr :: forall e a . ( ... ) => (a -> Bool) -> Matrix e a a
corr p = let f = fromF p :: Matrix e a ()
      in khatr f (identity :: Matrix e a a)
```

Listing 3.12: `Dist` - `select` and `cond` operators

Revisiting probabilistic interpretation of Arrows and the work by [Lindley et al. \(2011\)](#), it is clear that the isomorphism evidenced by them, that a Monad is isomorphic to Arrows of type $a \rightarrow ()$, can be visualised in the type `Dist`. More clearly, `Dist a` is a row vector matrix of type $a \leftarrow 1$ and corresponds to the probability monad presented by [Erwig and Kollmansberger \(2006\)](#).

A [SAF](#), if the associated data type is `(Enum a, Bounded a, Eq a)`, is equivalent to a Monad. See for instance:

```
class Applicative f => Selective f where
  select :: f (Either a b) -> f (a -> b) -> f b

eliminate :: (Eq a, Selective f) => a -> f b -> f (Either a b) -> f (Either a b)
eliminate x fb fa = select (match x <$> fa) (const . Right <$> fb)
  where
    match ` (Right y) = Right (Right y)
    match x (Left y) = if x == y then Left () else Right (Left y)

class Selective m => Monad m where
  return :: a -> m a
  return = pure

(>>=) :: (Enum a, Bounded a, Eq a) => m a -> (a -> m b) -> m b
(>>=) ma famb =
  let as = [minBound .. maxBound]
  in fromRight <$> foldr (\c -> eliminate c (famb c)) (Left <$> ma) as
  where
    fromRight (Right x) = x
```

Listing 3.13: Constrained monad instance

This fact, in case of functions would give an inefficient `bind (>>=)` implementation but in the case of matrices (distributions), this gives rise to matrix composition that takes advantage of the divide-and-conquer and fusion laws. In fact, in order to lift functions to matrices (distributions) the same type restrictions are needed. Given this, it is safe to assume that in a practical sense the `Matrix/Dist` type is a [SAF](#) and equivalently the distribution monad.

Wrapping up, moving from functions to matrices has allowed us to express probability distribution more elegantly and algebraically probabilistic than other representations [Erwig and Kollmansberger \(2006\)](#); [Kidd \(2007\)](#). It turns out that the designed data type takes advantage of the minimum amount of structure required for a [SAF](#) to be equivalent to a monad in the developed programming library, due to the necessary constraints. This, coupled with [SAF's](#) probabilistic interpretation, has enabled us to go one step further in finding out how [SAFs](#) offers a more efficient abstraction than monads by exploiting a parallel nature in computing discrete exhaustive probabilities. Although [SAFs](#) appear to lose the speculative execution capabilities in this probabilistic environment due to the fact that any two computations will always be required, their static analysis properties may still be useful in performing any optimisations as defined in [Bernstein \(2019\)](#).

In [B](#) the source code for the internal structure of the matrix definition can be consulted.

PROSPECT OF FUTURE WORK

When using matrices to represent distributions or probabilistic functions, we end up losing the speculative execution of [SAFs](#). While the current work has made it possible to find a probabilistic interpretation for [SAFs](#) in the light of the Arrow abstraction and the `select` operator, exploring practical ways where the speculative execution nature of [SAFs](#) can be useful will be the primary focus of the future work. Apart from this, elaborating different benchmarks that compare the current work with one of the most popular libraries used to calculate discrete exhaustive probabilities ([Erwig and Kollmansberger, 2006](#)) is also crucial for the success of these dissertation.

The possibility of writing a Functional Pearl about this project is also being considered.

BIBLIOGRAPHY

Hmatrix: Numeric linear algebra. URL <http://hackage.haskell.org/package/hmatrix-0.20.0.0>.

Dave H. Annis. Probability and statistics: The science of uncertainty, michael j. evans and jeffrey s. rosenthal. *The American Statistician*, 59:276–276, 2005. URL <https://EconPapers.repec.org/RePEc:bes:amstat:v:59:y:2005:m:august:p:276-276>.

Steve Awodey. *Category Theory*. Oxford University Press, Inc., New York, NY, USA, 2nd edition, 2010. ISBN 0199237182, 9780199237180.

Ryan Bernstein. Static analysis for probabilistic programs. *arXiv preprint arXiv:1909.05076*, 2019.

Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0-13-507245-X.

Paolo Capriotti and Ambrus Kaposi. Free applicative functors. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014.*, pages 2–30, 2014. doi: 10.4204/EPTCS.153.2. URL <https://doi.org/10.4204/EPTCS.153.2>.

George Casella and Roger Berger. *Statistical Inference*. Duxbury Resource Center, June 2001. ISBN 0534243126.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems, APLAS '08*, pages 205–220, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89329-5. doi: 10.1007/978-3-540-89330-1_15. URL http://dx.doi.org/10.1007/978-3-540-89330-1_15.

Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *Journal of Functional Programming*, 16(1):21–34, 2006. doi: 10.1017/S0956796805005721.

William Feller. *An introduction to probability theory and its applications. Vol. II*. Second edition. John Wiley & Sons Inc., New York, 1971.

Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex bayesian modelling. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 43(1):169–177, 1994.

Michèle Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis (Ottawa, Ont., 1980)*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer, Berlin, 1982.

Roger Godement. *Topologie algébrique et théorie des faisceaux*, volume 13. Hermann Paris, 1958.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

- Noah D. Goodman. The principles and practice of probabilistic programming. *SIGPLAN Not.*, 48(1): 399–402, January 2013. ISSN 0362-1340. doi: 10.1145/2480359.2429117. URL <http://doi.acm.org/10.1145/2480359.2429117>.
- Andrew D. Gordon, Mihhail Aizatulin, Johannes Borgstrom, Guillaume Claret, Thore Graepel, Aditya V. Nori, Sriram K. Rajamani, and Claudio Russo. A model-learner pattern for bayesian reasoning. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 403–416, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. doi: 10.1145/2429069.2429119. URL <http://doi.acm.org/10.1145/2429069.2429119>.
- HaskellWiki. Typeclassopedia — haskellwiki,, 2019. URL <https://wiki.haskell.org/index.php?title=Typeclassopedia&oldid=63003>. [Online; accessed 5-November-2019].
- John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, May 2000. ISSN 0167-6423. doi: 10.1016/S0167-6423(99)00023-4. URL [http://dx.doi.org/10.1016/S0167-6423\(99\)00023-4](http://dx.doi.org/10.1016/S0167-6423(99)00023-4).
- Eric Kidd. Build your own probability monads. *Draft paper for Hac*, 7, 2007.
- Ken Friis Larsen. Memory efficient implementation of probability monads. *Unpublished manuscript (August 2011)*, 2011.
- Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electron. Notes Theor. Comput. Sci.*, 229(5):97–117, March 2011. ISSN 1571-0661. doi: 10.1016/j.entcs.2011.02.018. URL <http://dx.doi.org/10.1016/j.entcs.2011.02.018>.
- Barbara Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34, January 1987. ISSN 0362-1340. doi: 10.1145/62139.62141. URL <http://doi.acm.org/10.1145/62139.62141>.
- Hugo Daniel Macedo. Matrices as arrows: why categories of matrices matter. 2012.
- Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.
- Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.
- Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. *SIGPLAN Not.*, 49(9):325–337, August 2014. ISSN 0362-1340. doi: 10.1145/2692915.2628144. URL <http://doi.acm.org/10.1145/2692915.2628144>.
- Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. Desugaring haskell's do-notation into applicative operations. *SIGPLAN Not.*, 51(12):92–104, September 2016. ISSN 0362-1340. doi: 10.1145/3241625.2976007. URL <http://doi.acm.org/10.1145/3241625.2976007>.
- Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.
- T Minka, J Winn, J Guiver, and A Kannan. Infer .net 2.3, nov. 2009. *Software available from* <http://research.microsoft.com/infernet>.

- Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4. URL [http://dx.doi.org/10.1016/0890-5401\(91\)90052-4](http://dx.doi.org/10.1016/0890-5401(91)90052-4).
- Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jeremie Dimino. Selective applicative functors. *Proc. ACM Program. Lang.*, 3(ICFP):90:1–90:29, July 2019. ISSN 2475-1421. doi: 10.1145/3341694. URL <http://doi.acm.org/10.1145/3341694>.
- Daniel Murta and Jose Nuno Oliveira. Calculating risk in functional programming. *arXiv preprint arXiv:1311.3687*, 2013.
- José N Oliveira. Towards a linear algebra of programming. *Formal Aspects of Computing*, 24(4-6): 433–458, 2012.
- José Nuno Oliveira and Victor Cacciari Miraldo. “keep definition, change category” - a practical approach to state-based system calculi. *J. Log. Algebr. Meth. Program.*, 85:449–474, 2016.
- Ross Paterson. Constructing applicative functors. In *Proceedings of the 11th International Conference on Mathematics of Program Construction, MPC’12*, pages 300–323, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31112-3. doi: 10.1007/978-3-642-31113-0_15. URL http://dx.doi.org/10.1007/978-3-642-31113-0_15.
- Daniel Pebles. Sigma selective, 2019. URL <https://web.archive.org/web/20190625225137/https://gist.github.com/copumpkin/d5bdbc7afda54ff04049b6bdbceffb67e>.
- Tomas Petricek. What we talk about when we talk about monads. *CoRR*, abs/1803.10195, 2018. URL <http://arxiv.org/abs/1803.10195>.
- Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. *SIGPLAN Not.*, 37(1):154–165, January 2002. ISSN 0362-1340. doi: 10.1145/565816.503288. URL <http://doi.acm.org/10.1145/565816.503288>.
- A. Ścibior*. *Formally justified and modular Bayesian inference for probabilistic programs*. PhD thesis, University of Cambridge, UK, 2019.
- Adam Ścibior, Zoubin Ghahramani, and Andrew D. Gordon. Practical probabilistic programming with monads. *SIGPLAN Not.*, 50(12):165–176, August 2015. ISSN 0362-1340. doi: 10.1145/2887747.2804317. URL <http://doi.acm.org/10.1145/2887747.2804317>.
- Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. Functional programming for modular bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(ICFP):83, 2018.
- Jane Street. A composable build system, 2018. URL <https://dune.build/>.
- S. Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. volume 1129, 07 2001. doi: 10.1007/3-540-61628-4-7.
- Jared Tobin. *Embedded Domain-Specific Languages for Bayesian Modelling and Inference*. PhD thesis, The University of Auckland, 2018.

- David Tolpin, Jan-Willem van de Meent, and Frank Wood. Probabilistic programming in anglican. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 308–311. Springer, 2015.
- P. Wadler. Monads for functional programming. In *Int’l School on Advanced Functional Programming*, pages 24–52. Springer, 1995.
- Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA ’89*, pages 347–359, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: 10.1145/99370.99404. URL <http://doi.acm.org/10.1145/99370.99404>.



TYPE SAFE LAOP MATRIX WRAPPER LIBRARY

```
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NoStarIsType #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}
{-# OPTIONS_GHC -fplugin GHC.TypeLits.KnownNat.Solver #-}
```

```
module Matrix.Internal
```

```
( Matrix (..) ,
  NonZero ,
  ValidDimensions ,
  KnownDimensions ,
  fromLists ,
  toLists ,
  toList ,
  columns ,
  rows ,
  matrix ,
  tr ,
  row ,
  col ,
  fmapRows ,
  fmapColumns ,
  ident ,
  zeros ,
```

```

    ones ,
    bang ,
    diag ,
    (|||) ,
    (==) ,
    i1 ,
    i2 ,
    p1 ,
    p2 ,
    (-|-) ,
    (><) ,
    kp1 ,
    kp2 ,
    khatri ,
    selectM ,
    comp ,
    fromF ,
  )
where

import Control.DeepSeq
import Data.Binary
import qualified Data.List as L
import Data.Proxy
import Foreign.Storable
import GHC.TypeLits
import qualified Numeric.LinearAlgebra as LA
import qualified Numeric.LinearAlgebra.Data as HM

— | The 'Matrix' type is a type safe wrapper around the
— 'Numeric.LinearAlgebra.Data.Matrix' data type.
newtype Matrix e (c :: Nat) (r :: Nat) = M {unMatrix :: HM.Matrix e}

deriving instance (LA.Container HM.Matrix e) => Eq (Matrix e c r)

deriving instance (LA.Container HM.Vector e, Fractional e, Fractional (HM.Vector
e), Num (HM.Matrix e)) => Fractional (Matrix e c r)

deriving instance (Floating e, LA.Container HM.Vector e, Floating (HM.Vector e),
Fractional (HM.Matrix e)) => Floating (Matrix e c r)

deriving instance (LA.Container HM.Matrix e, Num e, Num (HM.Vector e)) => Num (
Matrix e c r)

deriving instance (Read e, LA.Element e) => Read (Matrix e c r)

deriving instance (Binary (HM.Vector e), LA.Element e) => Binary (Matrix e c r)

```

```
deriving instance (Storable e, NFData e) => NFData (Matrix e c r)
```

```
instance (Show e, LA.Element e) => Show (Matrix e c r) where
  show (M m) = show m
```

```
type NonZero (n :: Nat) = (CmpNat n o ~ 'GT)
```

```
type ValidDimensions (n :: Nat) (m :: Nat) = (NonZero n, NonZero m)
```

```
type KnownDimensions (n :: Nat) (m :: Nat) = (KnownNat n, KnownNat m)
```

```
— CONVERTER FUNCTIONS
```

```
— | Matrix converter function. It builds a matrix from
— a list of lists @[e]@ (considered as rows).
```

```
fromLists :: forall e c r. (LA.Element e, KnownDimensions c r) => [[e]] -> Matrix
  e c r
```

```
fromLists [] = error "Wrong list dimensions"
```

```
fromLists l@(h : _) =
```

```
  let ccols = fromInteger $ natVal (Proxy :: Proxy c)
      rrows = fromInteger $ natVal (Proxy :: Proxy r)
      lrows = length l
      lcols = length h
```

```
  in if rrows /= lrows || ccols /= lcols
      then error "Wrong list dimensions"
      else M . HM.fromLists $ l
```

```
— | Matrix converter function. It builds a list of lists from
— a 'Matrix'.
```

```
— Inverse of 'fromLists'.
```

```
toList :: (LA.Element e) => Matrix e c r -> [[e]]
```

```
toList = HM.toLists . unMatrix
```

```
— | Matrix converter function. It builds a list of elements from
— a 'Matrix'.
```

```
toList :: (LA.Element e) => Matrix e c r -> [e]
```

```
toList = concat . toLists
```

```
— | Matrix converter function. It builds a matrix from a function.
```

```
fromF :: forall c r a b e. (Enum a, Enum b, Eq b, Num e, Ord e, LA.Element e,
  KnownNat c, KnownNat r) => (a -> b) -> Matrix e c r
```

```
fromF f =
```

```
  let ccols = fromInteger $ natVal (Proxy :: Proxy c)
```

```

rows = fromInteger $ natVal (Proxy :: Proxy r)
elementsA = take ccols $ map toEnum [0 ..]
elementsB = take rrows $ map toEnum [0 ..]
combinations = (,) <$> elementsA <*> elementsB
combAp = map snd . L.sort . map (\(a, b) -> if f a == b then ((fromEnum a,
    fromEnum b), 1) else ((fromEnum a, fromEnum b), 0)) $ combinations
mList = buildList combAp rrows
in tr $ fromLists mList
where
    buildList [] ` = []
    buildList l r = take r l : buildList (drop r l) r

```

— *DIMENSIONS FUNCTIONS*

— | *Obtain the number of columns of a matrix*

```

columns :: forall e c r. KnownNat c => Matrix e c r -> Integer
columns ` = natVal (Proxy :: Proxy c)

```

— | *Obtain the number of rows of a matrix*

```

rows :: forall e c r. KnownNat r => Matrix e c r -> Integer
rows ` = natVal (Proxy :: Proxy r)

```

```

fmapColumns :: forall b e a r. (Storable e, LA.Element e, KnownNat b) => Matrix e
    a r -> Matrix e b r

```

```
fmapColumns =
```

```

    let cols = fromInteger $ natVal (Proxy :: Proxy b)
    in M . HM.reshape cols . HM.fromList . toList

```

```

fmapRows :: forall b e a c. (Storable e, LA.Element e, KnownDimensions c b) =>
    Matrix e c a -> Matrix e c b

```

```
fmapRows =
```

```

    let rows = fromInteger $ natVal (Proxy :: Proxy b)
    in tr . M . HM.reshape rows . HM.fromList . toList

```

— *MISC FUNCTIONS*

— | *Create a matrix.*

```

matrix :: forall e c r. (KnownDimensions c r, Storable e) => [e] -> Matrix e c r
matrix l =

```

```

    let m = (reshape @e @c) . HM.fromList $ l
        mcols = HM.cols (unMatrix m)
        mrows = HM.rows (unMatrix m)
        ccols = fromInteger $ natVal (Proxy :: Proxy c)

```

```

    rrows = fromInteger $ natVal (Proxy :: Proxy r)
  in if mcols /= ccols || mrows /= rrows
      then error "Wrong list dimensions"
      else m

— | Matrix transpose
tr :: forall e c r. (LA.Element e, KnownDimensions c r) => Matrix e c r -> Matrix
    e r c
tr = fromLists . L.transpose . toLists

— | Create a row vector matrix.
row :: (Storable e, LA.Element e, KnownNat c) => [e] -> Matrix e c 1
row = asRow . HM.fromList

— | Create a column vector matrix.
col :: (Storable e) => [e] -> Matrix e 1 r
col = asColumn . HM.fromList

— | Creates the identity matrix of given dimension.
ident :: forall e c. (Num e, LA.Element e, KnownNat c) => Matrix e c c
ident =
  let c = fromInteger $ natVal (Proxy :: Proxy c)
  in M . HM.ident $ c

— | Zero Matrix polymorphic definition
zeros :: forall e c r. (KnownDimensions c r, Num e, LA.Container HM.Vector e) =>
    Matrix e c r
zeros =
  let ccols = fromInteger $ natVal (Proxy :: Proxy c)
      rrows = fromInteger $ natVal (Proxy :: Proxy r)
  in M $ HM.konst 0 (rrows, ccols)

— | One Matrix polymorphic definition
ones :: forall e c r. (KnownDimensions c r, Num e, LA.Container HM.Vector e) =>
    Matrix e c r
ones =
  let ccols = fromInteger $ natVal (Proxy :: Proxy c)
      rrows = fromInteger $ natVal (Proxy :: Proxy r)
  in M $ HM.konst 1 (rrows, ccols)

— | Bang Matrix polymorphic Matrix
bang :: forall e c. (KnownNat c, Num e, LA.Container HM.Vector e) => Matrix e c
    1
bang =
  let ccols = fromInteger $ natVal (Proxy :: Proxy c)
  in M $ HM.konst 1 (1, ccols)

```

```

— | Creates a square matrix with a given diagonal.
diag :: forall e c. (Num e, LA.Element e, KnownNat c) => [e] -> Matrix e c c
diag l =
  let c = fromInteger $ natVal (Proxy :: Proxy c)
      dims = length l
  in if c /= dims
      then error "Wrong list dimensions"
      else M . HM.diag . HM.fromList $ l

```

```

— BLOCK MATRIX FUNCTIONS (BIPRODUCT)

```

```

— | Matrix block algebra 'Junc' operator
(|||) :: (LA.Element e, ValidDimensions n m, NonZero p) => Matrix e m p -> Matrix
      e n p -> Matrix e (m + n) p
(|||) a b = M $ HM.fromBlocks [[unMatrix a, unMatrix b]]

infixl 3 |||

```

```

— | Matrix block algebra 'Split' operator
(===) :: (LA.Element e, ValidDimensions n m, NonZero p) => Matrix e p m -> Matrix
      e p n -> Matrix e p (m + n)
(===) a b = M $ HM.fromBlocks [[unMatrix a], [unMatrix b]]

infixl 2 ===

```

```

— | Matrix 'Junc' left injection matrix definition
i1 :: (Num e, ValidDimensions n m, KnownDimensions n m, LA.Element e, LA.
      Container HM.Vector e) => Matrix e m (m + n)
i1 = ident === zeros

```

```

— | Matrix 'Junc' right injection matrix definition
i2 :: (Num e, ValidDimensions n m, KnownDimensions n m, LA.Element e, LA.
      Container HM.Vector e) => Matrix e n (m + n)
i2 = zeros === ident

```

```

— | Matrix 'Split' left projection matrix definition
p1 :: (Num e, ValidDimensions n m, KnownDimensions n m, KnownNat (m + n), LA.
      Element e, LA.Container HM.Vector e) => Matrix e (m + n) m
p1 = tr i1

```

```

— | Matrix 'Split' right projection matrix definition
p2 :: (Num e, ValidDimensions n m, KnownDimensions n m, KnownNat (m + n), LA.
      Element e, LA.Container HM.Vector e) => Matrix e (m + n) n
p2 = tr i2

```

— *MATRIX BIPRODUCT FUNCTORS*

— | *Matrix coproduct bifunctor*

```
(-|-) ::
  forall e n m j k.
  ( ValidDimensions n m,
    ValidDimensions k j,
    NonZero (k + j),
    LA.Element e,
    LA.Numeric e,
    KnownDimensions k j
  ) =>
  Matrix e n k ->
  Matrix e m j ->
  Matrix e (n + m) (k + j)
(-|-) a b = (i1 `comp` a) ||| (i2 `comp` b)
```

infixl 5 -|-

— | *Kronecker product of two matrices*

```
(><) :: LA.Product e => Matrix e m p -> Matrix e n q -> Matrix e (m * n) (p * q)
(><) (M a) (M b) = M . LA.kronecker a $ b
```

infixl 4 ><

— *MATRIX SELECTVIE EQUIVALENT FUNCTION*

```
selectM ::
  ( LA.Numeric e,
    Enum a,
    Enum b,
    Ord e,
    Eq b,
    KnownDimensions m1 m2,
    ValidDimensions m1 m2
  ) =>
  Matrix e n (m1 + m2) -> (a -> b) -> Matrix e n m2
selectM m y = (fromF y ||| ident) `comp` m
```

— *MATRIX COMPOSITION, KHATRI RAO FUNCTIONS*

— | *Matrix – Matrix multiplication aka Matrix composition*
 comp :: LA.Numeric e => Matrix e p m -> Matrix e n p -> Matrix e n m
 comp (M a) (M b) = M . (LA.<>) a \$ b

— | *Khatri Rao product left projection (inductive definition)*

```
class KhatriP1 e (m :: Nat) (k :: Nat) where
  kp1 :: Matrix e (m * k) m
```

instance

```
{-# OVERLAPPING #-}
( KnownNat k,
  Num e,
  LA.Numeric e,
  LA.Container HM.Vector e
) =>
KhatriP1 e 1 k
where
kp1 = ones @e @k @1
```

instance

```
{-# OVERLAPPABLE #-}
( ValidDimensions m k,
  KnownNat k,
  KnownNat ((m - 1) * k),
  KnownNat (m - 1),
  Num e,
  LA.Numeric e,
  LA.Container HM.Vector e,
  (1 + (m - 1)) ~ m,
  (k + ((m - 1) * k)) ~ (m * k),
  NonZero ((m - 1) * k),
  NonZero (m - 1),
  KhatriP1 e (m - 1) k
) =>
KhatriP1 e m k
where
kp1 = ones @e @k @1 -|- kp1 @e @(m - 1) @k
```

— | *Khatri Rao product right projection (inductive definition)*

```
class KhatriP2 e (k :: Nat) (m :: Nat) where
  kp2 :: Matrix e (m * k) k
```

instance

```
{-# OVERLAPPING #-}
( Num e,
  LA.Element e,
  KnownNat k
```

```

) =>
KhatrIP2 e k 1
where
kp2 = ident @e @k

instance
{-# OVERLAPPABLE #-}
( (k + ((m - 1) * k)) ~ (m * k) ,
  ValidDimensions m k,
  NonZero ((m - 1) * k) ,
  LA.Element e ,
  Num e ,
  KnownNat k ,
  KhatrIP2 e k (m - 1)
) =>
KhatrIP2 e k m
where
kp2 = ident @e @k ||| kp2 @e @k @(m - 1)

— | Khatr Rao product of two matrices (Pairing)
khatr ::
  forall e m p q.
  ( KnownDimensions p (p * q) ,
    KnownNat q ,
    Num e ,
    Num (HM.Vector e) ,
    LA.Numeric e ,
    LA.Container HM.Vector e ,
    KhatrIP1 e p q ,
    KhatrIP2 e q p
  ) =>
  Matrix e m p ->
  Matrix e m q ->
  Matrix e m (p * q)
khatr a b = (tr (kp1 @e @p @q) `comp` a) * (tr (kp2 @e @q @p) `comp` b)

```

— *AUXILIARY FUNCTIONS*

— | *Creates a matrix from a vector by grouping the elements in rows*
— *with the desired number of columns.*

```

reshape :: forall e c r. (Storable e, KnownNat c) => HM.Vector e -> Matrix e c r
reshape v =
  let cols = fromInteger $ natVal (Proxy :: Proxy c)
  in M $ HM.reshape cols v

```

```

— | Creates a 1-column matrix from a vector.
asColumn :: forall e r. (Storable e) => HM.Vector e -> Matrix e 1 r
asColumn = reshape @e @1

— | Creates a 1-vector matrix from a vector.
asRow :: (Storable e, LA.Element e, KnownNat c) => HM.Vector e -> Matrix e c 1
asRow = tr . asColumn

```

Listing A.1: **Matrix.Internal**

TYPE SAFE LAOP INDUCTIVE MATRIX DEFINITION LIBRARY

```
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE InstanceSigs #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE NoStarIsType #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TypeApplications #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}
```

```
— |
— Module      : Matrix.Internal
— Copyright   : (c) Armando Santos 2019–2020
— Maintainer  : armandoifsantos@gmail.com
— Stability   : experimental
—
— The LAoP discipline generalises relations and functions treating them as
— Boolean matrices and in turn consider these as arrows.
—
— LAoP is a library for algebraic (inductive) construction and manipulation
— of matrices
— in Haskell. See <https://github.com/bolt12/master-thesis my Msc Thesis> for
— the
— motivation behind the library, the underlying theory, and implementation
— details.
—
— This module offers many of the combinators mentioned in the work of
— Macedo (2012) and Oliveira (2012).
—
— This is an Internal module and it is no supposed to be imported.
```

```

module Matrix.Internal
( — | This definition makes use of the fact that 'Void' is
  — isomorphic to 0 and '()' to 1 and captures matrix
  — dimensions as stacks of 'Either's.
  —
  — There exists two type families that make it easier to write
  — matrix dimensions: 'FromNat' and 'Count'. This approach
  — leads to a very straightforward implementation
  — of L AoP combinators.

  — * Type safe matrix representation
  Matrix (..) ,

  — * Primitives
  empty ,
  one ,
  junc ,
  split ,

  — * Auxiliary type families
  FromNat ,
  Count ,
  Normalize ,

  — * Matrix construction and conversion
  FromLists ,
  fromLists ,
  toLists ,
  toList ,
  matrixBuilder ,
  row ,
  col ,
  zeros ,
  ones ,
  bang ,
  constant ,

  — * Misc
  — ** Get dimensions
  columns ,
  rows ,

  — ** Matrix Transposition
  tr ,

```

```

— ** Selective operator
select ,

— ** McCarthy's Conditional
cond ,

— ** Matrix "abiding"
abideJS ,
abideSJ ,

— * Biproduct approach
— ** Split
(==),
— *** Projections
p1 ,
p2 ,
— ** Junc
(|||) ,
— *** Injectors
i1 ,
i2 ,
— ** Bifunctors
(-|-),
(><),

— ** Applicative matrix combinators

— | Note that given the restrictions imposed it is not possible to
— implement the standard type classes present in standard Haskell.
— *** Matrix pairing projections
kp1 ,
kp2 ,

— *** Matrix pairing
khatri ,

— * Matrix composition and lifting

— ** Arrow matrix combinators

— | Note that given the restrictions imposed it is not possible to
— implement the standard type classes present in standard Haskell.
identity ,
comp ,
fromF ,
fromF' ,

```

```

    — * Matrix printing
    pretty ,
    prettyPrint
  )
  where

import Utils
import Data.Bool
import Data.Kind
import Data.List
import Data.Proxy
import Data.Void
import GHC.TypeLits
import Data.Type.Equality
import GHC.Generics
import Control.DeepSeq
import Control.Category
import Prelude hiding ((.))

— | LAoP (Linear Algebra of Programming) Inductive Matrix definition .
data Matrix e cols rows where
  Empty :: Matrix e Void Void
  One :: e -> Matrix e () ()
  Junc :: Matrix e a rows -> Matrix e b rows -> Matrix e (Either a b) rows
  Split :: Matrix e cols a -> Matrix e cols b -> Matrix e cols (Either a b)

deriving instance (Show e) => Show (Matrix e cols rows)

— | Type family that computes the cardinality of a given type dimension .
—
— It can also count the cardinality of custom types that implement the
— 'Generic' instance .
type family Count (d :: Type) :: Nat where
  Count (Natural n m) = (m - n) + 1
  Count (Either a b) = (+) (Count a) (Count b)
  Count (a , b) = (*) (Count a) (Count b)
  Count (a -> b) = (^) (Count b) (Count a)
  — Generics
  Count (M1 ` ` f p) = Count (f p)
  Count (K1 ` ` `) = 1
  Count (V1 `) = 0
  Count (U1 `) = 1
  Count ((:*) a b p) = Count (a p) * Count (b p)
  Count ((:+) a b p) = Count (a p) + Count (b p)
  Count d = Count (Rep d R)

```



```

— | Type family that computes of a given type dimension from a given natural
—
— Thanks to Li-Yao Xia this type family is super fast.
type family FromNat (n :: Nat) :: Type where
  FromNat 0 = Void
  FromNat 1 = ()
  FromNat n = FromNat' (Mod n 2 == 0) (FromNat (Div n 2))

type family FromNat' (b :: Bool) (m :: Type) :: Type where
  FromNat' 'True m = Either m m
  FromNat' 'False m = Either () (Either m m)

— | Type family that normalizes the representation of a given data
— structure
type family Normalize (d :: Type) :: Type where
  Normalize d = FromNat (Count d)

— | It is not possible to implement the 'id' function so it is
— implementation is 'undefined'. However 'comp' can be and this partial
— class implementation exists just to make the code more readable.
—
— Please use 'identity' instead.
instance (Num e) => Category (Matrix e) where
  id = undefined
  (.) = comp

instance NFData e => NFData (Matrix e cols rows) where
  rnf Empty = ()
  rnf (One e) = rnf e
  rnf (Junc a b) = rnf a `seq` rnf b
  rnf (Split a b) = rnf a `seq` rnf b

instance Eq e => Eq (Matrix e cols rows) where
  Empty == Empty = True
  (One a) == (One b) = a == b
  (Junc a b) == (Junc c d) = a == c && b == d
  (Split a b) == (Split c d) = a == c && b == d
  x@(Split a b) == y@(Junc c d) = x == abideJS y
  x@(Junc a b) == y@(Split c d) = abideJS x == y

instance Num e => Num (Matrix e cols rows) where

  Empty + Empty = Empty
  (One a) + (One b) = One (a + b)
  (Junc a b) + (Junc c d) = Junc (a + c) (b + d)
  (Split a b) + (Split c d) = Split (a + c) (b + d)
  x@(Split a b) + y@(Junc c d) = x + abideJS y

```

```

x@(Junc a b) + y@(Split c d) = abideJS x + y

Empty - Empty                = Empty
(One a) - (One b)            = One (a - b)
(Junc a b) - (Junc c d)     = Junc (a - c) (b - d)
(Split a b) - (Split c d)   = Split (a - c) (b - d)
x@(Split a b) - y@(Junc c d) = x - abideJS y
x@(Junc a b) - y@(Split c d) = abideJS x - y

Empty * Empty                = Empty
(One a) * (One b)            = One (a * b)
(Junc a b) * (Junc c d)     = Junc (a * c) (b * d)
(Split a b) * (Split c d)   = Split (a * c) (b * d)
x@(Split a b) * y@(Junc c d) = x * abideJS y
x@(Junc a b) * y@(Split c d) = abideJS x * y

abs Empty                = Empty
abs (One a)              = One (abs a)
abs (Junc a b)           = Junc (abs a) (abs b)
abs (Split a b)          = Split (abs a) (abs b)

signum Empty            = Empty
signum (One a)          = One (signum a)
signum (Junc a b)       = Junc (signum a) (signum b)
signum (Split a b)      = Split (signum a) (signum b)

— | Primitives

— | Empty matrix constructor
empty :: Matrix e Void Void
empty = Empty

— | Unit matrix constructor
one :: e -> Matrix e () ()
one = One

— | Matrix 'Junc' constructor
junc :: Matrix e a rows -> Matrix e b rows -> Matrix e (Either a b) rows
junc = Junc

infixl 3 |||

— | Matrix 'Junc' constructor
(|||) :: Matrix e a rows -> Matrix e b rows -> Matrix e (Either a b) rows
(|||) = Junc

— | Matrix 'Split' constructor

```

```

split :: Matrix e cols a -> Matrix e cols b -> Matrix e cols (Either a b)
split = Split

infixl 2 ===

— | Matrix 'Split' constructor
(===) :: Matrix e cols a -> Matrix e cols b -> Matrix e cols (Either a b)
(===) = Split

— Construction

— | Type class for defining the 'fromList' conversion function.
—
— Given that it is not possible to branch on types at the term level type
— classes are needed very much like an inductive definition but on types.
class FromLists e cols rows where
  — | Build a matrix out of a list of list of elements. Throws a runtime
  — error if the dimensions do not match.
  fromLists :: [[e]] -> Matrix e cols rows

instance FromLists e Void Void where
  fromLists [] = Empty
  fromLists ` ` = error "Wrong dimensions"

instance {-# OVERLAPPING #-} FromLists e () () where
  fromLists [[e]] = One e
  fromLists ` ` = error "Wrong dimensions"

instance {-# OVERLAPPING #-} (FromLists e cols ()) => FromLists e (Either () cols
  ) () where
  fromLists [h : t] = Junc (One h) (fromLists [t])
  fromLists ` ` = error "Wrong dimensions"

instance {-# OVERLAPPABLE #-} (FromLists e a (), FromLists e b (), KnownNat (
  Count a)) => FromLists e (Either a b) () where
  fromLists [l] =
    let rowsA = fromInteger (natVal (Proxy :: Proxy (Count a)))
    in Junc (fromLists [take rowsA l]) (fromLists [drop rowsA l])
  fromLists ` ` = error "Wrong dimensions"

instance {-# OVERLAPPING #-} (FromLists e () rows) => FromLists e () (Either ()
  rows) where
  fromLists ([h] : t) = Split (One h) (fromLists t)
  fromLists ` ` = error "Wrong dimensions"

instance {-# OVERLAPPABLE #-} (FromLists e () a, FromLists e () b, KnownNat (
  Count a)) => FromLists e () (Either a b) where

```

```

fromLists l@([h] : t) =
  let rowsA = fromInteger (natVal (Proxy :: Proxy (Count a)))
  in Split (fromLists (take rowsA l)) (fromLists (drop rowsA l))
fromLists `      = error "Wrong dimensions"

instance {-# OVERLAPPABLE #-} (FromLists e (Either a b) c, FromLists e (Either a
  b) d, KnownNat (Count c)) => FromLists e (Either a b) (Either c d) where
  fromLists l@(h : t) =
    let lh      = length h
        rowsC   = fromInteger (natVal (Proxy :: Proxy (Count c)))
        condition = all (== lh) (map length t)
    in if lh > 0 && condition
        then Split (fromLists (take rowsC l)) (fromLists (drop rowsC l))
        else error "Not all rows have the same length"

— | Matrix builder function. Constructs a matrix provided with
— a construction function.
matrixBuilder ::
  forall e cols rows.
  ( FromLists e cols rows,
    KnownNat (Count cols),
    KnownNat (Count rows)
  ) =>
  ((Int, Int) -> e) ->
  Matrix e cols rows
matrixBuilder f =
  let c      = fromInteger $ natVal (Proxy :: Proxy (Count cols))
      r      = fromInteger $ natVal (Proxy :: Proxy (Count rows))
      positions = [(a, b) | a <- [0 .. (r - 1)], b <- [0 .. (c - 1)]]
  in fromLists . map (map f) . groupBy (\(x, _) (w, _) -> x == w) $ positions

— | Constructs a column vector matrix
col :: (FromLists e () rows) => [e] -> Matrix e () rows
col = fromLists . map (: [])

— | Constructs a row vector matrix
row :: (FromLists e cols ()) => [e] -> Matrix e cols ()
row = fromLists . (: [])

— | Lifts functions to matrices with arbitrary dimensions.
—
— NOTE: Be careful to not ask for a matrix bigger than the cardinality of
— types @@ or @@ allows.
fromF ::
  forall a b cols rows e.
  ( Bounded a,
    Bounded b,

```

```

    Enum a,
    Enum b,
    Eq b,
    Num e,
    Ord e,
    KnownNat (Count cols),
    KnownNat (Count rows),
    FromLists e rows cols
  ) =>
  (a -> b) ->
  Matrix e cols rows
fromF f =
  let minA      = minBound @a
      maxA      = maxBound @a
      minB      = minBound @b
      maxB      = maxBound @b
      ccols     = fromInteger $ natVal (Proxy :: Proxy (Count cols))
      rrows     = fromInteger $ natVal (Proxy :: Proxy (Count rows))
      elementsA = take ccols [minA .. maxA]
      elementsB = take rrows [minB .. maxB]
      combinations = (,) <$> elementsA <*> elementsB
      combAp      = map snd . sort . map (\(a, b) -> if f a == b
                                                    then ((fromEnum a,
                                                         fromEnum b), 1)
                                                    else ((fromEnum a,
                                                         fromEnum b), 0)) $
                    combinations

      mList      = buildList combAp rrows
  in tr $ fromLists mList
where
  buildList [] ` = []
  buildList l r = take r l : buildList (drop r l) r

— | Lifts functions to matrices with dimensions matching @@ and @@
— cardinality's.
fromF' ::
  forall a b e.
  ( Bounded a,
    Bounded b,
    Enum a,
    Enum b,
    Eq b,
    Num e,
    Ord e,
    KnownNat (Count (Normalize a)),
    KnownNat (Count (Normalize b)),
    FromLists e (Normalize b) (Normalize a)

```

```

) =>
(a -> b) ->
  Matrix e (Normalize a) (Normalize b)
fromF' f =
  let minA      = minBound @a
      maxA      = maxBound @a
      minB      = minBound @b
      maxB      = maxBound @b
      ccols     = fromInteger $ natVal (Proxy :: Proxy (Count (Normalize a)))
      rrows     = fromInteger $ natVal (Proxy :: Proxy (Count (Normalize b)))
      elementsA = take ccols [minA .. maxA]
      elementsB = take rrows [minB .. maxB]
      combinations = (,) <$> elementsA <*> elementsB
      combAp      = map snd . sort . map (\(a, b) -> if f a == b
                                                then ((fromEnum a,
                                                         fromEnum b), 1)
                                                else ((fromEnum a,
                                                         fromEnum b), 0)) $
                      combinations

      mList      = buildList combAp rrows
  in tr $ fromLists mList
where
  buildList [] ` = []
  buildList l r = take r l : buildList (drop r l) r

— Conversion

— | Converts a matrix to a list of lists of elements.
toLists :: Matrix e cols rows -> [[e]]
toLists Empty      = []
toLists (One e)     = [[e]]
toLists (Split l r) = toLists l ++ toLists r
toLists (Junc l r)  = zipWith (++) (toLists l) (toLists r)

— | Converts a matrix to a list of elements.
toList :: Matrix e cols rows -> [e]
toList = concat . toLists

— Zeros Matrix

— | The zero matrix. A matrix wholly filled with zeros.
zeros :: (Num e, FromLists e cols rows, KnownNat (Count cols), KnownNat (Count
  rows)) => Matrix e cols rows
zeros = matrixBuilder (const 0)

— Ones Matrix

```

```

— | The ones matrix. A matrix wholly filled with ones.
—
— Also known as T (Top) matrix.
ones :: (Num e, FromLists e cols rows, KnownNat (Count cols), KnownNat (Count
    rows)) => Matrix e cols rows
ones = matrixBuilder (const 1)

— Const Matrix

— | The constant matrix constructor. A matrix wholly filled with a given
— value.
constant :: (Num e, FromLists e cols rows, KnownNat (Count cols), KnownNat (Count
    rows)) => e -> Matrix e cols rows
constant e = matrixBuilder (const e)

— Bang Matrix

— | The T (Top) row vector matrix.
bang :: forall e cols. (Num e, Enum e, FromLists e cols (), KnownNat (Count cols)
    ) => Matrix e cols ()
bang =
    let c = fromInteger $ natVal (Proxy :: Proxy (Count cols))
    in fromLists [take c [1, 1 ..]]

— Identity Matrix

— | Identity matrix.
identity :: (Num e, FromLists e cols cols, KnownNat (Count cols)) => Matrix e
    cols cols
identity = matrixBuilder (bool 0 1 . uncurry (==))

— Matrix composition (MMM)

— | Matrix composition. Equivalent to matrix-matrix multiplication.
—
— This definition takes advantage of divide-and-conquer and fusion laws
— from LAoP.
comp :: (Num e) => Matrix e cr rows -> Matrix e cols cr -> Matrix e cols rows
comp Empty Empty = Empty
comp (One a) (One b) = One (a * b)
comp (Junc a b) (Split c d) = comp a c + comp b d — Divide-and-conquer
    law
comp (Split a b) c = Split (comp a c) (comp b c) — Split fusion law
comp c (Junc a b) = Junc (comp c a) (comp c b) — Junc fusion law

— Projections

```

```

— | Biproduct first component projection
p1 :: forall e m n. (Num e, KnownNat (Count n), KnownNat (Count m), FromLists e n
  m, FromLists e m m) => Matrix e (Either m n) m
p1 =
  let iden = identity :: Matrix e m m
      zero = zeros :: Matrix e n m
  in junc iden zero

— | Biproduct second component projection
p2 :: forall e m n. (Num e, KnownNat (Count n), KnownNat (Count m), FromLists e m
  n, FromLists e n n) => Matrix e (Either m n) n
p2 =
  let iden = identity :: Matrix e n n
      zero = zeros :: Matrix e m n
  in junc zero iden

— Injections

— | Biproduct first component injection
i1 :: (Num e, KnownNat (Count n), KnownNat (Count m), FromLists e n m, FromLists
  e m m) => Matrix e m (Either m n)
i1 = tr p1

— | Biproduct second component injection
i2 :: (Num e, KnownNat (Count n), KnownNat (Count m), FromLists e m n, FromLists
  e n n) => Matrix e n (Either m n)
i2 = tr p2

— Dimensions

— | Obtain the number of rows.
—
— NOTE: The 'KnownNat' constraint is needed in order to obtain the
— dimensions in constant time.
—
— TODO: A 'rows' function that does not need the 'KnownNat' constraint in
— exchange for performance.
rows :: forall e cols rows. (KnownNat (Count rows)) => Matrix e cols rows -> Int
rows ` = fromInteger $ natVal (Proxy :: Proxy (Count rows))

— | Obtain the number of columns.
—
— NOTE: The 'KnownNat' constraint is needed in order to obtain the
— dimensions in constant time.
—
— TODO: A 'columns' function that does not need the 'KnownNat' constraint in
— exchange for performance.

```



```

columns :: forall e cols rows. (KnownNat (Count cols)) => Matrix e cols rows ->
  Int
columns ` = fromInteger $ natVal (Proxy :: Proxy (Count cols))

— Coproduct Bifunctor

infixl 5 -|-

— | Matrix coproduct functor also known as matrix direct sum.
(-|-) ::
  forall e n k m j.
  ( Num e,
    KnownNat (Count j),
    KnownNat (Count k),
    FromLists e k k,
    FromLists e j k,
    FromLists e k j,
    FromLists e j j
  ) =>
  Matrix e n k ->
  Matrix e m j ->
  Matrix e (Either n m) (Either k j)
(-|-) a b = Junc (i1 . a) (i2 . b)

— Khatri Rao Product and projections

— | Khatri Rao product first component projection matrix.
kp1 ::
  forall e m k .
  ( Num e,
    KnownNat (Count k),
    FromLists e (FromNat (Count m * Count k)) m,
    KnownNat (Count m),
    KnownNat (Count (Normalize (m, k)))
  ) => Matrix e (Normalize (m, k)) m
kp1 = matrixBuilder f
  where
    offset = fromInteger (natVal (Proxy :: Proxy (Count k)))
    f (x, y)
      | y >= (x * offset) && y <= (x * offset + offset - 1) = 1
      | otherwise = 0

— | Khatri Rao product second component projection matrix.
kp2 ::
  forall e m k .
  ( Num e,
    KnownNat (Count k),

```

```

    FromLists e (FromNat (Count m * Count k)) k,
    KnownNat (Count m),
    KnownNat (Count (Normalize (m, k)))
  ) => Matrix e (Normalize (m, k)) k
kp2 = matrixBuilder f
where
  offset = fromInteger (natVal (Proxy :: Proxy (Count k)))
  f (x, y)
    | x == y || mod (y - x) offset == 0 = 1
    | otherwise                        = 0

— | Khatri Rao Matrix product also known as matrix pairing.
—
— NOTE: That this is not a true categorical product, see for instance:
—
— @
—           | kp1 . khatri a b == a
— khatri a b ==> |
—           | kp2 . khatri a b == b
— @
—
— ..Emphasis.. on the implication symbol.
khatri ::
  forall e cols a b.
  ( Num e,
    KnownNat (Count a),
    KnownNat (Count b),
    KnownNat (Count (Normalize (a, b))),
    FromLists e (Normalize (a, b)) a,
    FromLists e (Normalize (a, b)) b
  ) => Matrix e cols a -> Matrix e cols b -> Matrix e cols (Normalize (a, b))
khatri a b =
  let kp1' = kp1 @e @a @b
      kp2' = kp2 @e @a @b
  in (tr kp1') . a * (tr kp2') . b

— Product Bifunctor (Kronecker)

infixl 4 <×

— | Matrix product functor also known as kronecker product
(><) ::
  forall e m p n q.
  ( Num e,
    KnownNat (Count m),
    KnownNat (Count n),

```

```

KnownNat (Count p),
KnownNat (Count q),
KnownNat (Count (Normalize (m, n))),
FromLists e (Normalize (m, n)) m,
FromLists e (Normalize (m, n)) n,
KnownNat (Count (Normalize (p, q))),
FromLists e (Normalize (p, q)) p,
FromLists e (Normalize (p, q)) q
)
=> Matrix e m p -> Matrix e n q -> Matrix e (Normalize (m, n)) (Normalize (p
, q))
(><) a b =
  let kp1' = kp1 @e @m @n
      kp2' = kp2 @e @m @n
  in khatrı (a . kp1') (b . kp2')

— Matrix abide Junc Split

— | Matrix "abiding" followin the 'Junc'-'Split' abide law.
—
— Law:
—
— @
— 'Junc' ('Split' a c) ('Split' b d) == 'Split' ('Junc' a b) ('Junc' c d)
— @
abideJS :: Matrix e cols rows -> Matrix e cols rows
abideJS (Junc (Split a c) (Split b d)) = Split (Junc (abideJS a) (abideJS b)) (
  Junc (abideJS c) (abideJS d)) — Junc-Split abide law
abideJS Empty = Empty
abideJS (One e) = One e
abideJS (Junc a b) = Junc (abideJS a) (abideJS b)
abideJS (Split a b) = Split (abideJS a) (abideJS b)

— Matrix abide Split Junc

— | Matrix "abiding" followin the 'Split'-'Junc' abide law.
—
— @
— 'Split' ('Junc' a b) ('Junc' c d) == 'Junc' ('Split' a c) ('Split' b d)
— @
abideSJ :: Matrix e cols rows -> Matrix e cols rows
abideSJ (Split (Junc a b) (Junc c d)) = Junc (Split (abideSJ a) (abideSJ c)) (
  Split (abideSJ b) (abideSJ d)) — Split-Junc abide law
abideSJ Empty = Empty
abideSJ (One e) = One e
abideSJ (Junc a b) = Junc (abideSJ a) (abideSJ b)
abideSJ (Split a b) = Split (abideSJ a) (abideSJ b)

```

— *Matrix transposition*

— | *Matrix transposition.*

```
tr :: Matrix e cols rows -> Matrix e rows cols
tr Empty          = Empty
tr (One e)         = One e
tr (Junc a b)      = Split (tr a) (tr b)
tr (Split a b)     = Junc (tr a) (tr b)
```

— *Selective 'select' operator*

— | *Selective functors 'select' operator equivalent inspired by the
ArrowMonad solution presented in the paper.*

```
select ::
  ( Bounded a,
    Bounded b,
    Enum a,
    Enum b,
    Num e,
    Ord e,
    Eq b,
    KnownNat (Count (Normalize a)),
    KnownNat (Count (Normalize b)),
    KnownNat (Count cols),
    FromLists e (Normalize b) (Normalize a),
    FromLists e (Normalize b) (Normalize b)
  ) => Matrix e cols (Either (Normalize a) (Normalize b)) -> (a -> b) ->
    Matrix e cols (Normalize b)

select m y =
  let f = fromF y
  in junc f identity . m
```

— *McCarthy's Conditional*

— | *McCarthy's Conditional expresses probabilistic choice.*

```
cond ::
  ( cols ~ FromNat (Count cols),
    KnownNat (Count cols),
    FromLists e () cols,
    FromLists e cols (),
    FromLists e cols cols,
    Bounded a,
    Enum a,
    Num e,
    Ord e
  )
```

```

=>
  (a -> Bool) -> Matrix e cols rows -> Matrix e cols rows -> Matrix e cols
    rows
cond p f g = junc f g . grd p

grd ::
  ( q ~ FromNat (Count q),
    KnownNat (Count q),
    FromLists e () q,
    FromLists e q (),
    FromLists e q q,
    Bounded a,
    Enum a,
    Num e,
    Ord e
  )
=>
  (a -> Bool) -> Matrix e q (Either q q)
grd f = split (corr f) (corr (not . f))

corr ::
  forall e a q .
  ( q ~ FromNat (Count q),
    KnownNat (Count q),
    FromLists e () q,
    FromLists e q (),
    FromLists e q q,
    Bounded a,
    Enum a,
    Num e,
    Ord e
  )
  => (a -> Bool) -> Matrix e q q
corr p = let f = fromF p :: Matrix e q ()
        in khatri f (identity :: Matrix e q q)

— Pretty print

prettyAux :: Show e => [[e]] -> [[e]] -> String
prettyAux [] ' ' = ""
prettyAux [[e]] m ' ' = "|" ++ fill (show e) ++ " |\n"
  where
    v = fmap show m
    widest = maximum $ fmap length v
    fill str = replicate (widest - length str - 2) ' ' ++ str
prettyAux [h] m ' ' = "|" ++ fill (unwords $ map show h) ++ " |\n"
  where

```

```

    v = fmap show m
    widest = maximum $ fmap length v
    fill str = replicate (widest - length str - 2) ' ' ++ str
prettyAux (h : t) l = "| " ++ fill (unwords $ map show h) ++ " |\n" ++
    prettyAux t l

where
    v = fmap show l
    widest = maximum $ fmap length v
    fill str = replicate (widest - length str - 2) ' ' ++ str

— | Matrix pretty printer
pretty :: (KnownNat (Count cols), Show e) => Matrix e cols rows -> String
pretty m = "+ " ++ unwords (replicate (columns m) blank) ++ " +\n" ++
    prettyAux (toList m) (toList m) ++
    "+ " ++ unwords (replicate (columns m) blank) ++ " +"

where
    v = fmap show (toList m)
    widest = maximum $ fmap length v
    fill str = replicate (widest - length str) ' ' ++ str
    blank = fill ""

— | Matrix pretty printer
prettyPrint :: (KnownNat (Count cols), Show e) => Matrix e cols rows -> IO ()
prettyPrint = putStrLn . pretty

```

Listing B.1: **Matrix.Internal**

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.