



**University of Minho**

School of Engineering

Informatics Department

Armando João Isáias Ferreira dos Santos

## **Selective Applicative Functors & Probabilistic Programming**

December 2020



**University of Minho**

School of Engineering

Informatics Department

Armando João Isaías Ferreira dos Santos

## **Selective Applicative Functors & Probabilistic Programming**

Master dissertation

Master Degree in Integrated Masters in Computer Engineering

Dissertation supervised by

**José Nuno Oliveira (INESCTEC & University of Minho)**

**Andrey Mokhov (Newcastle University, UK)**

December 2020

---

## COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

---

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorisation conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

LICENSE GRANTED TO USERS OF THIS WORK:



CC BY

<https://creativecommons.org/licenses/by/4.0/>

---

## ACKNOWLEDGEMENTS

---

Acknowledgements to be written here.

---

## STATEMENT OF INTEGRITY

---

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

---

## ABSTRACT

---

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 for their static analysis capabilities. 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 to describe probabilistic computations instead of using monads. In particular, selective functors appear to provide a way to obtain 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, as well as exploring ways of offering SAF capabilities to probabilistic programming, by exposing sampling as a concurrency problem. As a result, provides a Haskell type-safe matrix library capable of expressing probability distributions and probabilistic computations as typed matrices, and a probabilistic programming eDSL that explores various techniques in order to offer a novel, performant solution to probabilistic functional programming.

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

---

## RESUMO

---

Em programação funcional, os funtores *aplicativos seletivos* (FAS) são uma abstração entre funtores aplicativos e monades. Essa abstração requer que todos os efeitos sejam declarados estaticamente, mas fornece uma maneira de selecionar quais efeitos serão executados dinamicamente. FAS têm se mostrado uma abstração útil em vários exemplos, incluindo dois estudos de caso industriais. Funtores seletivos têm sido usados pela sua capacidade de análise estática. O conjunto de informações sobre todos os efeitos possíveis numa computação e o facto de que eles permitem a execução *especulativa* tornam possível descrever computações probabilísticas. Em particular, funtores seletivos parecem oferecer uma maneira de obter uma implementação mais eficiente de distribuições probabilísticas do que monades.

Esta dissertação aborda uma interpretação probabilística para as abstrações *Arrow* e *Selective* à luz da disciplina da álgebra linear da programação, bem como explora formas de oferecer as capacidades dos FAS para programação probabilística, expondo *sampling* como um problema de concorrência. Como resultado, fornece uma biblioteca de matrizes em Haskell, capaz de expressar distribuições de probabilidade e cálculos probabilísticos como matrizes tipadas e uma eDSL de programação probabilística que explora várias técnicas, com o objetivo de oferecer uma solução inovadora e de alto desempenho para a programação funcional probabilística.

**Palavras-chave:** dissertação de mestrado, programação funcional, programação probabilística, monades, aplicativos, funtores aplicativos seletivos, haskell, matrizes

---

## CONTENTS

---

1	INTRODUCTION	1
1.1	Motivation and Goals	2
1.2	State of the Art	2
1.2.1	Hierarchy of Abstractions	3
1.2.2	Functors	3
1.2.3	Applicative Functors	4
1.2.4	Monads	5
1.2.5	Arrows	7
1.2.6	Selective Applicative Functors	8
1.2.7	Summary	10
1.3	Related Work	11
1.3.1	Exhaustive Probability Distribution Encoding	11
1.3.2	Embedded Domain Specific Languages	11
1.4	Structure of the Dissertation	12
2	BACKGROUND	13
2.1	Set Theory	13
2.2	Basic Probabilities and Distributions	14
2.3	(Linear) Algebra of Programming	19
2.3.1	Structure of the Category of Matrices	19
2.3.2	Biproducts	20
2.3.3	Biproduct Functors	21
3	CONTRIBUTION	23
3.1	The Problem and its Challenges	23
3.1.1	Probabilistic Interpretation of Selective Functors	23
3.1.2	Inefficient Probability Encodings	23
3.1.3	Proposed Approach - Solution	24
3.2	Probabilistic Interpretation of Arrows	24
3.3	Type Safe Linear Algebra of Programming Matrix Library	25
3.4	Probabilistic Interpretation of Selective Functors	27
3.5	Type safe inductive matrix definition	29
3.5.1	The Probability Distribution Matrix and the Selective Abstraction	32
3.5.2	Equational Reasoning	34
3.6	Probabilistic Programming eDSL & Sampling	36
3.6.1	Examples of Probabilistic Programs	39
3.6.2	Sampling and Inference Algorithms	42
3.7	Sampling as a Concurrency Problem	44
3.7.1	The Concurrency Monad	44
3.7.2	Sampling	45
3.7.3	Implementation	46



3.8	Summary	47
4	APPLICATIONS	48
4.1	LAoP Sprinkler example	48
4.2	eDSL Sprinkler example	50
4.3	Benchmarks	52
4.3.1	LAoP Matrix composition	53
4.3.2	Distribution matrix vs distribution list monad	54
4.3.3	Sequential vs Concurrent Selective eDSL	55
4.4	Summary	57
5	CONCLUSIONS AND FUTURE WORK	58
5.1	Conclusions	58
5.2	Future work	59
A	TYPE SAFE LAOP MATRIX WRAPPER LIBRARY	65
B	TYPE SAFE LAOP INDUCTIVE MATRIX DEFINITION LIBRARY	78
C	SELECTIVE PROBABILISTIC PROGRAMMING LIBRARY	100

---

## LIST OF FIGURES

---

Figure 4.1	Testbed environment	52
Figure 4.2	Matrix composition benchmarks	53
Figure 4.3	Matrix vs List - select operator	54
Figure 4.4	Benchmarks results (No delay)	56
Figure 4.5	Benchmarks results (delay)	57

---

## LIST OF TABLES

---

Table 1	Summary of abstractions and their probabilistic counterpart	<a href="#">10</a>
Table 2	Number of possible arrangements of size $r$ from $n$ objects	<a href="#">18</a>

---

## LIST OF LISTINGS

---

1.1	Functor laws . . . . .	3
1.2	Applicative laws . . . . .	4
1.3	Monad laws and definition in terms of unit and join . . . . .	5
1.4	Monad laws and definition in terms of unit and bind . . . . .	5
1.5	Relation between join and bind . . . . .	6
1.6	Arrow type-class . . . . .	7
1.7	Arrow Kleisli type-class instance . . . . .	7
1.8	Selective Applicative Functor laws . . . . .	8
3.1	Inductive matrix definition . . . . .	25
3.2	Type-safe wrapper around HMatrix . . . . .	26
3.3	Interface equivalent function implementations . . . . .	26
3.4	LAoP Monty Hall Problem . . . . .	27
3.5	Selective ArrowMonad instance . . . . .	27
3.6	LAoP Selective instance . . . . .	28
3.7	Inductive Matrix definition . . . . .	30
3.8	Matrix composition and abiding functions . . . . .	30
3.9	Dimensions are type level naturals . . . . .	32
3.10	Dimensions are arbitrary data types . . . . .	32
3.11	Dist type alias . . . . .	32
3.12	Dist - select and cond operators . . . . .	33
3.13	Constrained monad instance . . . . .	33
3.14	select in terms of matrices . . . . .	34
3.15	Fork x z pattern match case . . . . .	35
3.16	Final result . . . . .	35
3.17	eDSL primitive building-blocks . . . . .	36
3.18	Conditioning function . . . . .	37
3.19	Sampling function . . . . .	38
3.20	Coin toss . . . . .	39
3.21	Coin toss results . . . . .	40
3.22	Throw coins indefinitely until Heads comes up . . . . .	40
3.23	Throw N coins . . . . .	41
3.24	Bad program . . . . .	42
3.25	Partial monadic bind function . . . . .	43
3.26	Partial monadic bind function . . . . .	43
3.27	Fetch Data Type . . . . .	46
3.28	Fetch Applicative instance . . . . .	47
3.29	Fetch Selective instance . . . . .	47
4.1	Example matrices . . . . .	49

4.2	State matrix . . . . .	49
4.3	State matrix composition function . . . . .	50
4.4	Probability of grass being wet calculation . . . . .	50
4.5	Example probabilistic functions . . . . .	50
4.6	tag combinator . . . . .	51
4.7	State distribution . . . . .	52
4.8	Results . . . . .	54
4.9	Programs used in evaluation . . . . .	55
A.1	Matrix.Internal . . . . .	65
B.1	Matrix.Internal . . . . .	78
C.1	SelectiveProb.hs . . . . .	100

---

## ACRONYMS

---

### A

AOP Algebra of Programming. [19](#), [21](#), [24](#), [25](#), [29](#)

### C

CS Computer Science. [2](#), [3](#), [10](#)

CT Category Theory. [1](#), [3](#), [5](#), [7](#), [10](#), [19](#), [24](#)

### E

EDSL Embedded domain specific language. [24](#), [36](#), [38](#), [42](#), [44](#), [45](#), [50](#), [51](#), [55](#), [56](#), [58](#), [59](#)

### F

FP Functional Programming. [1](#), [10](#), [24](#), [44](#)

### G

GADT Generalised Algebraic Datatype. [25](#), [30](#)

GHC Glasgow Haskell Compiler. [26](#), [30](#)

### L

LAOP Linear Algebra of Programming. [8](#), [19](#), [22](#), [24–26](#), [29](#), [30](#), [32](#), [36](#), [40](#), [49](#), [50](#), [58](#)

### P

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

PPL Probabilistic Programming Language. [11](#)

### S

SAF Selective Applicative Functors. [1](#), [2](#), [8–10](#), [12](#), [23–29](#), [33](#), [34](#), [36](#), [41](#), [44](#), [51](#), [52](#), [58](#), [59](#)

---

## INTRODUCTION

---

*Functional Programming (FP)* deals with the complexity of real life problems by handling so-called (side) *effects* in an algebraic manner. Monads are one such algebraic device, pioneered by Moggi (1991) in the field of computer science to verify *effectful* programs, i.e. programs that deal with side effects. Wadler (1989) was among the first to recommend 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, which applies functions to monadic objects *carrying the effects through*. 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, this abstraction is often too strong for particular programming situations, where 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. Therefore, this kind of functor can only take two effectful computations and, independently (i.e. in parallel), compute their values and return their composition.

There are situations in which just having a Monad or an Applicative is too limiting, calling for a programming abstraction sitting somewhere between Monad and Applicative. An abstraction that requires all effects to be statically declared but provides a way to select which of the effects to execute dynamically was introduced by Mokhov et al. (2019) to cope with such situations. It is called the *Selective Applicative Functors (SAF)* abstraction.

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 nicer abstraction for describing conditional probabilistic computations. According to Mokhov et al. (2019), this kind of functor has proved to be a helpful abstraction in the fields of static analysis (at Jane Street) and speculative execution (at Facebook), achieving good results without disturbing the adopted code style.

Arrows (Hughes, 2000) are more generic than monads and were designed to abstract the structure of more complex patterns than the monad interface could support. The most common example is the parsing library by Swierstra and Duponcheel (1996) 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 FP ecosystem as a way to mitigate the somewhat heavy requirements 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 enables solving some of the structural constraints that refrain one from implementing a stronger abstraction and compose various combinators in order to achieve greater expressiveness. On the other hand, languages such as Haskell, which implement many of these abstractions out of the box, render code written in the Arrow style not only convoluted, but also unnatural and difficult to refactor.

## 1.1 MOTIVATION AND GOALS

The rise of new topics such as e.g. machine learning, deep learning, quantum computing are stimulating major advances in the programming language domain (Selinger, 2004; Innes et al., 2018). To cope with the increased complexity, mathematics always had a principal 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 a means of communicating (complex) concepts to computers. 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)* have come directly from mathematics. 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 *SAF* is such an abstraction.

Probabilistic programming allows programmers to model probabilistic events and predict or calculate results with a certain degree of uncertainty. In particular *PFP* manipulates and manages probabilities in an abstract, high-level way, circumventing 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 couple of Haskell libraries that provides an efficient encoding of probabilities, taking advantage of the selective applicative abstraction. Focusing 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, one of the aims of this work is to answer the following research question:

"Can the select operator be implemented more efficiently than the monadic bind operator?"

## 1.2 STATE OF THE ART

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



- The programming language;
- The underlying mathematical theory.

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 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 (one could even write: *ad-hoc*) that the need for such abstractions may be questionable. Mathematical abstractions are useful ways of generalising and organising patterns that abide by the same rules, i.e. are governed by the same set of laws. Thanks to much work on 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 correctly using it.

The following section presents widely used mathematical abstractions that made their 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.

### 1.2.1 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 composing simpler ones. Regarding abstractions as layers, one can pretty much think of the 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.

### 1.2.2 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

**PROBABILISTIC SPEAK** 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  $[a]$ .

For the purpose of probabilistic thinking,  $f\ a$  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 function  $f$  is not injective.) As will be seen in chapter 2 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 the probability  $P(A)$  of an event  $A$  in probability theory (Tobin, 2018).

### 1.2.3 Applicative Functors

**WHAT APPLICATIVE FUNCTORS ARE** Most functional programming languages separate pure computations from effectful ones. An effectful computation performs side effects or runs in a given context while delivering its result. While working with the Haskell functional programming language, McBride and Paterson (2008) found that the pattern of applying pure functions to effectful computations popped out very often 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. All it takes to abstract this pattern 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
    -- u <*> pure y = pure ($ y) <*> u

```

Listing 1.2: Applicative laws

It is important to note that in order to be an applicative,  $f$  first needs to be a functor. So, every applicative is a functor. This can be seen as going down one layer of abstraction in the hierarchy, by 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" by 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).

**PROBABILISTIC SPEAK** Looking at the laws of applicative functors one sees that they pretty much define what the intended semantics regarding sequencing effects are. 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  $\text{pure}$  can be seen as the embedding of a given value  $a$  in the probabilistic context with 100% chance, and  $\langle * \rangle$  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).

#### 1.2.4 Monads

**WHAT MONADS ARE** Before being introduced in programming languages, monads had already been 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). Evidence of the flexibility and usefulness of Monads can be found 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 presented in the same way, the mathematical monad and the programming language monad are the same concept.

**Definition 1.2.1.** A monad in a category  $\mathcal{C}$  is defined as a triple  $(T, \eta, \mu)$  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:

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

In programming, two alternative but equivalent definitions for monads come up. A functor can be 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                                2
(>=>) :: m a -> (a -> m b) -> m b              3
-- unit a >=> f = unit (f a)                    4
-- m >=> unit = m                                5
-- (m >=> f) >=> g = m >=> (\x -> f x >=> g)      6

```

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)                      1

```

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 turning 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 unwraps it into a single box. This metaphor however, is not so good at giving intuition for bind ( $\gg=$ ) but, as we saw in the previous listing, 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 their structure allows us to separate pure from impure code, obtaining, in this way, nice and structured programs that are easier to reason about.

**PROBABILISTIC SPEAK** 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 lets us combine an event  $m\ a$  with another that depends on the previous value  $a \rightarrow m\ b$  (Erwig and Kollmansberger, 2006). In other words, bind in a sense encapsulates the notion of conditional probability. What happens in a conditional probability calculation  $P(B|A)$  is that  $A$  becomes the sample space, and  $A$  &  $B$  will only occur a fraction  $P(A \cap B)$  of the time. Making the bridge with the type signature of ( $\gg=$ ):  $m\ a$  represents the new sample space  $A$  and  $a \rightarrow m\ b$  the fraction where  $A$  and  $B$  occur. This being said, the probability monad can be 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).

## 1.2.5 Arrows

**WHAT ARROWS ARE** Most abstractions described until now are based on [CT](#). 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 referred 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.

The astute reader will 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  $a \rightarrow m\ b$  is a 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 \(1996\)](#) 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 are not 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? [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.

**PROBABILISTIC SPEAK** 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 and therefore do not express any extra special capabilities ([Lindley et al., 2011](#)). Nevertheless, due to their generality, Arrows are able to offer either of the two abstraction (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.

### 1.2.6 Selective Applicative Functors

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

Allied to a specific research domain, like building 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 for combining conditional effects but can only do this dynamically, which makes static analysis impossible.

This said, [Mokhov et al. \(2019\)](#) developed an interface (abstraction) aiming at getting the best of both worlds, the [SAF](#):

```

class Applicative f => Selective f where
  -- also known as (<*)
  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)      5
-- x <*> (y <*> z) = (f <$> x) <*> (g <$> y) <*> (h <$> z)      6
-- where                                                         7
--     f x = Right <$> x                                         8
--     g y = \a -> bimap (,a) ($ a) y                            9
--     h z = uncurry z                                           10

```

Listing 1.8: Selective Applicative Functor laws

By construction, **SAFs** 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 either an `a` or a `b`. In the case of the return value being 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 being of type `b`, then the second computation is *skipped*.

The laws presented in the listing above characterise **SAFs**. The first law 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 no law enabling **SAFs** to discard the second computation, in particular `pure (Right x) <*> y = pure x`. And there is no law enabling 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 explanation for this is simple: it allows instances of **SAFs** which are useful for static analysis to be performed and the `select` operator becomes more expressive, in the same way that Applicative Functors do not limit the execution order of two independent results.

With this in mind, it is possible to see how **SAFs** 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. Moreover, **SAFs** 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, **SAFs** can be seen as the composition of an Applicative functor `f` with the `Either` monad (Mokhov et al., 2019). Even though this formalisation is not studied by Mokhov et al. (2019), one should address the relation between **SAFs** and Arrows. As every **SAF** is an instance of Applicative, every Applicative functor is also an instance of Selective. Moreover, as 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, **SAFs** 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.

Given that there was already an abstraction capable of expressing the same as **SAFs**, why did these 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 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



Abstraction	Operators	Probabilistic Equivalent
Functor	<code>fmap f A</code>	$P(A)$
Applicative	<code>pure A</code>	$A$
	<code>A &lt;*&gt; B</code>	$P(A)P(B)$
Monad	<code>return A</code>	$A$
	<code>A &gt;= B</code>	$P(B A) = \frac{P(B \cap A)}{P(A)}$
Arrow	<code>arr f</code>	Stochastic Matrix $f$
	<code>A &gt;&gt; B</code>	Stochastic Matrix Composition
Selective	<code>select A B</code>	-

Table 1: Summary of abstractions and their probabilistic counterpart

analysis of conditional effects and speculative execution without relying in the more powerful and intrusive Arrow abstraction.

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

The Monad instance lets one chain distributions, giving the possibility of expressing the calculation of conditioned probability.

The most prevalent abstractions in FP were analysed in order to understand the motivation and theory behind these and in which way they relate to the probabilistic setting. Table 1 summarises the relation between each abstraction and its probabilistic counterpart. The conclusion is that maths-theoretic foundations traverse all the abstractions addressed and, in particular, CT is ubiquitous in programming and CS in general. There are cases in which the need for abstraction comes from more practical contexts, calling for a more systematic and disciplined study grounded on sound mathematical frameworks and leading to the design of correct and efficient solutions.

This said, this dissertation is chiefly concerned with identifying which probabilistic interpretations or benefits are achievable with SAFs. After a detailed analysis of the different abstractions found in the FP ecosystem, we shall proceed to outlining several starting points in order to prove 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 has proved 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 an abstraction weaker than the monadic one, may prove to be of



value in mitigating the performance constraints that the monadic interface imposes because of being inherently sequential.

### 1.3 RELATED WORK

This thesis benefits from synergies among computing and mathematics fields such as probability theory, category theory and programming languages. This section reviews similar work in such fields of research.

#### 1.3.1 *Exhaustive Probability Distribution Encoding*

Over the past few years, the field of probabilistic programming has been primarily concerned with extending language capabilities in expressing probabilistic calculations 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 among the first to encode 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.

#### 1.3.2 *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., 2009), simplify writing inference algorithms for the price of reduced expressiveness.

A 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.

Ścibior et al. (2015) show 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 is used: sampling. This work by Ścibior et al. (2015), also elaborated in the first author's doctoral dissertation (Ścibior\*, 2019), kept on giving rise to a more modular extension of the library presented in previous work, in order to separate modelling from inference (Ścibior et al., 2018). Despite the results obtained, both solutions suffer from the fact that they use monads only to construct probability distributions. Since monads are inherently sequential they are unable 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 Giry monad and highlights its various capabilities by mapping them to the probabilistic setting. He 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 Giry 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.

#### 1.4 STRUCTURE OF THE DISSERTATION

This text 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 related work. Chapter 2 introduces the most relevant background topics and chapter 3.1 explains in more detail the problem at target and its main challenges. Chapters 3 and 4 present all details of the implemented solution, as well as some application examples and evaluation results. Finally, chapter 5 presents conclusions and guidelines for future work.

---

## BACKGROUND

---

This chapter shines a light through the path of probabilities and their foundations. The aim is to provide readers with a good context refreshment and intuition, saving them from the need to resort to heavy books. While more reading is required for a full understanding of the whole background, this chapter can easily be skipped by readers familiar with these subjects.

### 2.1 SET THEORY

The field of probability theory is the basis of statistics, giving means to model social and economic behaviour, infer from scientific experiments, or almost everything else. Through these models, statisticians are able to draw inferences from the 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. Statisticians aim at drawing conclusions about populations of objects by making observations or conducting experiments, for which they need to identify all possible outcomes in the first place, 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, is 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 surely knows, 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  $A$ :

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

These 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>	$(A \cup B)^c = A^c \cap B^c$ $(A \cap B)^c = A^c \cup B^c$

The reader is referred to [Casella and Berger \(2001\)](#) for the proofs of these properties.

## 2.2 BASIC PROBABILITIES AND DISTRIBUTIONS

Probabilities come up rather often in our daily lives. They are not only of use to statisticians. A good understanding of probability theory allows us to assess the likelihood of everyday tasks and to benefit from the wise choices learnt by experience. Probability theory is also useful in the fields of economics, medicine, science and engineering, and in risk analysis. For example, the design of a nuclear reactor must be such that the leak of radioactivity into the environment should be an extremely rare event. So, using probability theory as a tool to deal with uncertainty, 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 experiment is made, its realisation results in 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*. However, this "frequency of occurrence" is just one of the many interpretations of what a probability is, another one being more subjective: 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 in the interval  $[0, 1]$  said to be the probability of  $A$ , denoted  $P(A)$ . The domain of  $P$ , which intuitively is the set of all subsets of  $S$ , is called a *sigma algebra*, denoted by  $\mathcal{B}$ .

**Definition 2.2.1.** 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 2.2.1.1.** (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 (e.g.  $S = (-\infty, +\infty)$ ), 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 2.2.2.** 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:

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

These properties satisfy the Axioms of Probability (or the Kolmogorov Axioms), and every function that satisfies them is called a probability function. The first three 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 a probability function useful for calculating more complex probabilities. The additivity property automatically implies certain basic properties that are true for any probability model.

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 words: the probability of an event not happening is equal to one minus the probability of an event happening.

**Theorem 2.2.1.** 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 2.2.2.** Let  $A$  and  $B$  be two events such that  $B \subseteq A$ . Then:

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

From this one can draw, since  $P(A \cap B^c) \geq 0$  always holds:

**Corollary 2.2.2.1.** (Monotonicity) Let  $A$  and  $B$  be two events such that  $B \subseteq A$ . Then:

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

Moreover, by rearranging 2.2.2 one obtains:

**Corollary 2.2.2.2.** Let  $A$  and  $B$  be two events such that  $B \subseteq A$ . Then:

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

Finally, by lifting constraint  $B \subseteq A$  one has the following, more general property:

**Theorem 2.2.3.** (Principle of inclusion–exclusion, two-event version) Let  $A$  and  $B$  be two events. Then:

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

Since  $P(A \cup B) \leq 1$ , property (5) leads to (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 as the *Bonferroni's Inequality*. Altogether, one can say that the basic properties of total probability, subadditivity, and monotonicity hold. The interested reader is referred to (Casella and Berger, 2001) or (Annis, 2005) for proofs and more details concerning the theorems above.

### Counting and Enumerating Outcomes

Counting methods can be used to assess probabilities in finite sample spaces. In general, counting is non-trivial, often needing constraints to be taken into account. The approach is to break counting problems into easy-to-count sub-problems and use some combination rules.

**Theorem 2.2.4.** (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 2.2.4 is a good starting point, in some situations there are more aspects to consider. In a lottery, for instance, 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 shows the distinction 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, it is possible to construct a  $2 \times 2$  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 2.2.4 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 2.2.4, 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 2.2.3.** 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 2: 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 2.2.4.** 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 2.2.4 gives a useful way to calculate intersections:

$$P(A \cap B) = P(A|B)P(B) \quad (7)$$



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

**Theorem 2.2.5.** (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 2.2.5:

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

### 2.3 (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 a matrix as a morphism between two dimensions, matrix multiplication becomes simple matrix composition:

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

Since this discipline is centered around in **CT**, we assume some basic familiarity with categories  $\mathbb{C}$ ,  $\mathbb{D}$ , functors  $\mathbb{F}$ ,  $\mathbb{G} : \mathbb{C} \rightarrow \mathbb{D}$ , natural transformations  $\alpha, \beta : \mathbb{F} \rightarrow \mathbb{G}$ , products and coproducts. Otherwise, we remit the reader to [Awodey \(2010\)](#) and [Oliveira \(2008\)](#). A more detailed overview of the topic can also be seen in the work done by [Macedo \(2012\)](#).

#### 2.3.1 Structure of the Category of Matrices

##### *Vectors and Linear Maps*

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

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

In fact, matrices form an Abelian category:

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

2.3.2 *Biproducts*

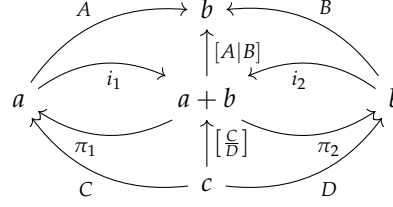
In an abelian category, a biproduct diagram for the objects  $m, n$  is a diagram of the following shape:

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

whose arrows  $\pi_1, \pi_2, i_1, i_2$ , 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  $\left[\frac{A}{B}\right]$  will be read 'A junc B' and 'C split D', respectively. These combinators purport 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:

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

$$\left[\frac{A}{B}\right] \cdot P = \left[\frac{A \cdot P}{B \cdot P}\right] \quad (9)$$

- Divide and conquer:

$$[A|B] \cdot \left[\frac{C}{D}\right] = A \cdot C + B \cdot D \quad (10)$$

- Converse duality:

$$[A|B]^\circ = \left[\frac{A^\circ}{B^\circ}\right] \quad (11)$$

- Exchange ("Abide") law:

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

### 2.3.3 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 well-known Kronecker product is the tensor product in the matrix category. In the context of [LAoP](#), this bifunctor may be expressed in terms of the Khatri Rao product which, in turn, can and be expressed in terms of the Hadamard and Schur matrix multiplication:

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

---

## CONTRIBUTION

---

This chapter discusses the major obstacles and difficulties of this master’s dissertation, as well as its main contributions, that arose from the study and work carried out during the master’s degree. As we shall see later, two types of contributions are made: one theoretical and the other practical. With regard to the theoretical contributions, the probabilistic interpretations of Arrows and [SAFs](#) are discussed, furthermore a bridge is made between sampling problems and concurrency ones. With regard to practical contributions, two probabilistic programming libraries in Haskell are described and presented.

### 3.1 THE PROBLEM AND ITS CHALLENGES

This section addresses the main problems and challenges that were found during this master thesis. The proposed approach and corresponding solution are described as well.

#### 3.1.1 *Probabilistic Interpretation of Selective Functors*

Section [1.2](#) presented some of the most well-known abstractions in functional programming, as well as their probabilistic interpretation. How these abstractions are translated into the context of probabilistic programming was also addressed. The fact that the Giry monad ([Giry, 1982](#); [Tobin, 2018](#)) is an Applicative Functor, takes us closer to a potential probabilistic interpretation of [SAFs](#). In addition, the relationship between [SAFs](#) and Arrows ([Mokhov et al., 2019](#)) have also led to the challenge of figuring out how Arrows’ fundamentals and generality can be useful in finding an answer.

One of the difficulties therefore is to find the probabilistic definition of [SAFs](#) and how it relates to any probabilistic programming construct.

#### 3.1.2 *Inefficient Probability Encodings*

There are two ways to model probabilistic distributions. In the light of the work outlined in section [1.3](#), it is possible to opt for an exhaustive representation of distributions, where all chance-value pairs are stored and any structural manipulation is done by changing all pairs, one by one. This method has the advantage of allowing the calculation of the exact probability of any type of event. However, even a seemingly simple problem can lead to an explosion of states within the distribution, which

has a major impact on performance. With this in mind, another less rigorous method of calculating probabilities is to calculate them using less reliable, yet faster and more efficient inference algorithms, instead of always measuring the exact probabilities across all values.

Modelling a simple program that calculates the likelihood that a particular event will occur in  $N$  throws of a die, using an exhaustive method, will easily become unfeasible for a relatively small  $N$ . Modelling a complex, critical program that calculates the probability of two aircrafts crashing, using a non-exhaustive approach may lead to hazardous situations, if the accuracy of the results is not the desired one. It is this trade-off that is generally discussed in probabilistic programming research and is largely based on the type of method used.

The challenge, then is to find a solution capable of minimising the distance between the two most common probabilistic distribution encodings. It is also challenging to find one that takes advantage of the [SAF](#) abstraction and manages to make the most out of its static analysis or speculative execution properties.

### 3.1.3 Proposed Approach - Solution

Regarding the problem of finding the probabilistic interpretation of [SAFs](#), a type safe matrix representation and manipulation library was 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.

Regarding the problem of finding an efficient probability encoding that is capable of taking advantage of the [SAF](#) abstraction, an *Embedded domain specific language (eDSL)* suited for writing probabilistic programs was designed, recurring to the Free Selective Functor construction. This [eDSL](#) will be important to see how far does the Selective abstraction is able to go in the probabilistic setting and what type of benefits can we extract from its conditional static analysis capabilities.

Of course, all this would not be possible without a deep understanding of the theoretical context in which [SAFs](#) are inserted. That being said the probabilistic semantics for [SAFs](#), where all the other contributions rest on top of, is provided.

After understanding the probabilistic capabilities of [SAFs](#), an analysis was performed to decide which of the two alternative approaches could have more impact. Either by reusing the former type safe [LAoP](#) library or by adopting the probabilistic [eDSL](#) library to express probabilistic programs. A set of case studies and examples were used to benchmark all contributions with the related work.

## 3.2 PROBABILISTIC INTERPRETATION OF ARROWS

[AoP](#) (Bird and de Moor, 1997) is regarded as a calculational and point-free programming discipline, and as a case where the use of [CT](#) provides an economy on definitions and proofs of correction. Bird and de Moor (1997) demonstrates how [CT](#) can be used to describe the basic building blocks of datatypes found in [FP](#) and the program derivation. The lesson from [AoP](#), where functions are viewed as a special case of relations, is that by changing the category (from *Fun* to *Rel*) expressive power increases. Relations are inherently non-deterministic and are capable of specifying a wider range of problems by making rich operators, such as converse and division, universally available. Other reasons for this transition are that more structure is uncovered, opportunities for generalization

are unveiled, and the arrangement of specific proofs becomes simpler. "Keep the definition, change category" is a slogan that summarises well the lesson that Bird and de Moor (1997) has passed and emphasises gradual composition, as seen in relational algebra.

So how can this lesson be used in a probabilistic context? Oliveira and Miraldo (2016) directs us towards 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 just 0's and 1's, we limit them to being left-stochastic, where the values of each column amount to 1, it would be possible to express multiple probabilistic extensions to the regular AoP combinations and help keep 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 ( $\gg=$ ) operator, which convolutes probabilistic reasoning. However, by adopting the LAoP transition, probabilistic function (Kleisli) composition becomes simply matrix composition (Oliveira, 2012).

This probabilistic Arrows interpretation takes the analysis one step closer to the probabilistic interpretation of SAF. It should be possible to encode matrices around sound mathematical abstractions and take advantage of the best they have to give, by using what has been learned so far. What SAF has to do with the linear approach to AoP, and how it fits into the probabilistic setting is the answer that it is hoped to find.

### 3.3 TYPE SAFE LINEAR ALGEBRA OF PROGRAMMING MATRIX LIBRARY

When finding a probabilistic interpretation for SAFs, an attempt was made to construct a matrix-representing data structure based on LAoP. The LAoP discipline has an inductive approach to the calculation of the various combinators that characterize it, since it is based on Junc and Split biproducts, which will be called Join and Fork from now on. A *Generalised Algebraic Datatype (GADT)* was then used as an attempt to achieve a strongly typed data type capable of inferring matrix dimensions. Taking advantage of the library of TypeLits, GADT was constructed as follows:

```

|| data Matrix e (c :: Nat) (r :: Nat) where                                1
||   One  :: e -> Matrix e 1 1                                           2
||   Join :: Matrix e m p -> Matrix e n p -> Matrix e (m + n) p         3

```

```

|| Fork :: Matrix e p m -> Matrix e p n -> Matrix e p (m + n)
||

```

Listing 3.1: Inductive matrix definition

This inductive data type will correctly represent any type of matrix and infer dimensions. However since *Glasgow Haskell Compiler (GHC)* is not able to properly infer the correct types while pattern-matching, this data type poses some difficulties in implementing functions for construction and manipulation. It is easy to implement a simple example, such as matrix transposition, but others such as the entry-wise addition of two matrices, are impossible in practice, as two matrices of the same dimensions can be internally represented by a different combination of Joins and Forks. One solution to this problem would be to find a way to ensure that all matrices were constructed according to a convention (either Join of Forks or Fork of Joins), but even so the type system would still not be able to know that the matrices actually followed the convention.

In order to resolve this problem and to try to get a feel of the probabilistic interpretation/intuition of *SAF*, a library has been developed. This library just offers a type-safe newtype wrap around an existing library's matrix data structure. The chosen library was *HMatrix* (Ruiz, 2019) because it is one of the most common and widely used.

```

|| import qualified Numeric.LinearAlgebra.Data as HM
|| newtype Matrix e (c :: Nat) (r :: Nat) = M {unMatrix :: HM.Matrix e}
||

```

Listing 3.2: Type-safe wrapper around HMatrix

With this type-safe wrapper, it was possible to implement several *LAoP* combinators, but when using it one is at the mercy of the internal representation used by the host library, and the possibility of obtaining a structure that benefits from the properties of *SAFs* and *LAoP* is lost<sup>1</sup>. Nevertheless this technique makes a potential response closer.

As mentioned 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 Listing 3.2. As described in section 1.2, probability distributions are capable of satisfying the Functor, Applicative, and Monad instances. However, due to the constraints required for all type-checking to be carried out, and the fact that the content type of the matrix is in a negative position, it is not possible to implement instances for spoken interfaces. However an equivalent version of the functions of each instance can be implemented as shown in the list below:

```

|| -- | Monoidal/Applicative instance
|| khatr :: ( ... ) => 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
||

```

<sup>1</sup> Note that *HMatrix* is quite efficient, but for the purposes of this thesis we want to observe the benefits of using *SAFs* and *LAoP* as cleanly as possible



```
|| fromF :: ( ... ) => (a -> b) -> Matrix e c r 8
```

Listing 3.3: Interface equivalent function implementations

The Applicative instance is defined in terms of the Khatri Rao product, taking advantage of the monoidal nature that characterises this abstraction. With respect to Monads, the matrix composition, as seen in the previous subsection, is the equivalent of `bind`. Finally, with respect to the Arrow abstraction, 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 are presented in Appendix A for space economy purposes.

Having said that, simple probabilistic problems such as the Monty Hall problem can be modelled:

```
|| -- Monty Hall Problem 1
|| data Outcome = Win | Lose 2
||     deriving (Bounded, Enum, Eq, Show) 3
|| 4
|| switch :: Outcome -> Outcome 5
|| switch Win = Lose 6
|| switch Lose = Win 7
|| 8
|| firstChoice :: Dist Outcome 9
|| firstChoice = choose (1/3) 10
|| 11
|| secondChoice :: Matrix Double 2 2 12
|| secondChoice = fromF switch 13
|| 14
|| main :: IO () 15
|| main = do 16
||     print (p1 `comp` secondChoice `comp` firstChoice :: Matrix Double 1 1) 17
|| 18
|| {- 19
|| Output: 20
|| (1><1) 21
|| [ 0.6666666666666666 ] 22
|| -} 23
```

Listing 3.4: LAoP Monty Hall Problem

### 3.4 PROBABILISTIC INTERPRETATION OF SELECTIVE FUNCTORS

SAFs are said to provide the missing counterpart for `ArrowChoice` in the functor hierarchy, as shown by the following example (Mokhov et al., 2019):

```
|| instance ArrowChoice a => Selective (ArrowMonad a) where 1
```

```

select (ArrowMonad x) y = ArrowMonad $ x >>> (toArrow y ||| returnA)
select m y = (y ||| id) `comp` m
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

The relationship of Arrows with SAFs is highlighted in this case. Given the results seen in the previous section, an implementation similar to that shown in Listing 3.5 can be used, and a selective instance for the Matrix data type (3.2) can be obtained similarly. However it is not possible to implement an official instance due to restrictions placed on the types of the matrix dimensions. Nonetheless, it is possible to write the operator `select` at the cost of operators identical to the ones used above:

```

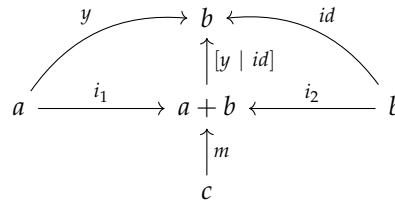
select :: ( ... ) => Matrix e n (m1 + m2) -> Matrix e m1 m2 -> Matrix e n m2
select m y = (y ||| id) `comp` m
(|||) :: Matrix e m p -> Matrix e n p -> Matrix e (m + n) p
(|||) = Join

```

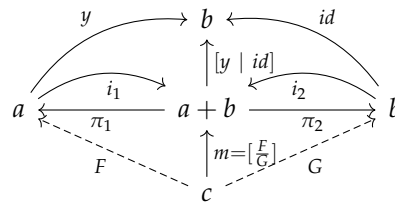
Listing 3.6: LAoP Selective instance

Notice that the function `toArrow` is needed, in order to Listing 3.5 to type check. In order to match the same signature, an equivalent `toMatrix` function would be necessary, unfortunately an instance of `Enum (a → b)` is needed, and that is currently not feasible in Haskell. For this reason, the type signature of `select` has been adjusted.

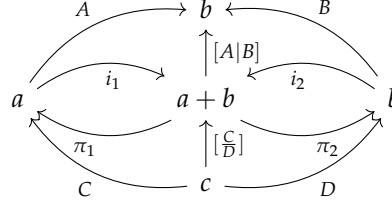
As interesting as the relationship between Arrows and SAFs is, and as much as it is possible to write an instance for the same interface, this tells us little or nothing about its semantics. It is more or less clear that some kind of conditional is expressed, but it is hard to imagine its implementation and interaction with the other combinators. To find a more concrete description, the following type diagram was designed:



A more detailed observation:

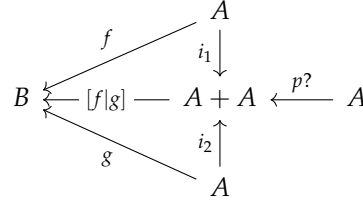


Generalising:



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

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:

$$A + A \xleftarrow{p?} A = \begin{bmatrix} \Phi_p \\ \Phi_{-p} \end{bmatrix}$$

Looking closely at the diagram one can see some resemblance to what one gets from [Listing 3.6](#), and indeed McCarthy's conditionals have the same conditional nature that is inherent to [SAFs](#).

What would we then infer as what is the [SAFs](#) probabilistic interpretation? So far we can determine that if we consider  $f$  ([Listing 1.8](#)) as a distribution, the operator `select` has the ability to condition the random variable in probabilistic programming and branch the program in two separate ways. This operator expresses the divide-and-conquer rule present in the block-matrix calculus and is capable of exploiting parallelism. As we will see in [section 3.5.2](#), it is possible to further optimise this `select` operator by using some equational reasoning, suggesting that this abstraction is still useful.

### 3.5 TYPE SAFE INDUCTIVE MATRIX DEFINITION

Only a limited, intuitive understanding of the semantics and probabilistic interpretation of [SAFs](#) can be given by the type-safe matrix library described in [section 3.3](#). This library also illustrates the kinds of benefits that, by taking advantage of such abstraction, can be theoretically acquired. Also, in the provided library ([A](#)), these advantages are not transparent, because it is just a wrapper based on the `Vector` data type. Many different approaches and solutions were tried, in order to benefit from a efficient [LAoP](#)-inspired matrix encoding, whose benefits are apparent and promising, by construction. In the end, type-level naturals were abandoned and simple structured data-types were used to replace them. Given this, the following encoding was achieved:

```

data Matrix e cols rows where
  One :: e -> Matrix e () ()
  Join :: Matrix e a rows -> Matrix e b rows -> Matrix e (Either a b) rows
  Fork :: 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 (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
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)

```

Listing 3.7: Inductive Matrix definition

This solution is based on the assumption 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. In addition, this GADT ensures that the matrix always has valid dimensions i.e. correct-by-construction. This isomorphism is leveraged by the `Count` and `FromNat` type families to provide a conversion mechanism from and to data-types/type-level naturals. Surprisingly enough, `GHC` does not complain when pattern-matching with this definition, so it was possible to implement more complex functions.

Here is an example of how `LAoP` makes it possible to write a concise, correct and efficient code by exploring the divide-and-conquer, fusion and ‘abide’ laws of the biproducts:

```

comp :: Num e => Matrix e cr rows -> Matrix e cols cr -> Matrix e cols rows
comp (One a) (One b) = One (a * b)

```

```

comp (Join a b) (Fork c d) = comp a c + comp b d      -- Divide-and-conquer law    3
comp (Fork a b) c          = Fork (comp a c) (comp b c) -- Fork fusion law                4
comp c (Join a b)          = Join (comp c a) (comp c b) -- Join fusion law                5
                                                                    6
abideJS :: Matrix e cols rows -> Matrix e cols rows    7
abideJS (Join (Fork a c) (Fork b d)) = Fork (Join (abideJS a) (abideJS b)) (Join ( 8
    abideJS c) (abideJS d)) -- Join-Fork abide law
abideJS (One e)                = (One e)                9
abideJS (Join a b)              = Join (abideJS a) (abideJS b) 10
abideJS (Fork a b)              = Fork (abideJS a) (abideJS b) 11

```

Listing 3.8: Matrix composition and abiding functions

It is very straightforward to see the advantages of using an inductive approach to encoding matrices. However it still has its disadvantages. For example, more complex operators, such as the Khatri-Rao (also known as matrix pairing) product, are expected to return a matrix of type  $m \times n \leftarrow c$ , and their projections are limited to returning matrices constructed at the expense of `Eithers`. In addition, since certain large matrices, such as the identity matrix, have constraints associated to the dimension types, it is not possible to implement instances of abstractions such as `Arrow` or `Selective`. However, Santos and Oliveira (2020) show how restricted versions of these classes can be written, adopting the `(.)` composition operator, from the `Category` instance, in place of the more verbose `comp` function. It is also possible to have a version of `select` and McCarthy's conditional operators that, as will be seen, are indeed able to take advantage of all the theoretical gains described in the previous section.

Given this, the new data-type makes the following contributions:

- Enables the transformation and manipulation of matrices in a composable and flexible way.
- Compared to current libraries, this one is more compositional, polymorphic, and does not have partial matrix manipulation functions (hence less chances for usage errors).
- Using the mathematical framework described by the linear algebra of programming (Oliveira, 2012), this implementation of matrices allows easy manipulation of submatrices, making it especially suitable for formal verification and equational reasoning. In addition, the constructors of the data type ensure that sound matrices of this type i.e. malformed matrices with incorrect dimensions of the kind, can not be constructed.
- More concretely, compared to the previous data-type, this one has:
  - Statically typed dimensions;
  - Polymorphic data type dimensions;
  - Polymorphic matrix content;
  - Fast type natural conversion via `FromNat` type family;
  - Better type inference;
  - Matrix 'Join' and 'Fork'-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. 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:

```
import qualified LAoP.Matrix.Internal as I           1
                                                    2
newtype Matrix e (cols :: Nat) (rows :: Nat) = M (I.Matrix e (I.FromNat cols) (I.  3
    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  1
    .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.

### 3.5.1 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 other exhaustive probabilistic programming libraries and, by taking advantage of the inductive matrix encoding and the LAoP discipline that seems to be possible. 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           1
type Prob = Double                               2
                                                    3
-- | Newtype wrapper for column vector matrices. This represents a probability  4
-- distribution.                                   5
```

```
newtype Dist a = D (Matrix Prob () a)
```

6

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 1
select :: ( ... ) => Dist (Either a b) -> Matrix Prob a b -> Dist b 2
selectD (D d) m = D (Join m identity `comp` d) 3
4
-- McCarthy's Conditional 5
cond :: ( ... ) => (a -> Bool) -> Dist b -> Dist b -> Dist b 6
cond p (D f) (D g) = D (Join f g `comp` grd p) 7
8
-- == junc f g . split (corr p) (corr (not . p)) 9
-- == f . (corr p) + g . (corr (not Prelude.. p)) 10
-- (Paralellism via divide-and-conquer) 11
12
grd :: ( ... ) => (a -> Bool) -> Matrix e a (Either a a) 13
grd f = split (corr f) (corr (not Prelude.. f)) 14
15
corr :: forall e a . ( ... ) => (a -> Bool) -> Matrix e a a 16
corr p = let f = fromF p :: Matrix e a () 17
      in khatri f (identity :: Matrix e a a) 18
```

Listing 3.12: Dist - select and cond operators

Revisiting the 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](#) can be seen equivalent to a Monad, if the associated data type is (Enum a, Bounded a, Eq a) ([Mokhov et al., 2019](#)). See for instance:

```
class Applicative f => Selective f where 1
  select :: f (Either a b) -> f (a -> b) -> f b 2
3
eliminate :: (Eq a, Selective f) => a -> f b -> f (Either a b) -> f (Either a b) 4
eliminate x fb fa = select (match x <$> fa) (const . Right <$> fb) 5
  where 6
    match _ (Right y) = Right (Right y) 7
    match x (Left y) = if x == y then Left () else Right (Left y) 8
9
class Selective m => Monad m where 10
```

```

return :: a -> m a                                11
return = pure                                     12
                                                13
(>=>) :: (Enum a, Bounded a, Eq a) => m a -> (a -> m b) -> m b  14
(>=>) ma famb =                                    15
    let as = [minBound .. maxBound]              16
    in fromRight <$> foldr (\c -> eliminate c (famb c)) (Left <$> ma) as  17
    where                                          18
        fromRight (Right x) = x                  19

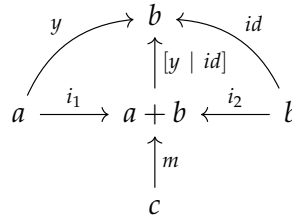
```

Listing 3.13: Constrained monad instance

This fact, in case of functions, would give an inefficient bind ( $\gg=$ ) implementation but, in the case of matrices (distributions), this gives place 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.

### 3.5.2 Equational Reasoning

This section shows how to use equational reasoning and the laws of the linear algebra of programming to prove properties of functions on matrices and/or to obtain more efficient programs.



As we already saw, from an abstract point-of-view, the diagram above corresponds to the ArrowChoice implementation of select where, in the case of stochastic matrices,  $m$  can be seen as a probability distribution that outputs either  $a$  or  $b$ , and  $y$  is only computed for values of type  $a$ , all others are skipped.

This leads to a straightforward implementation of select in terms of matrices:

```

select :: (...) => Matrix e c (Either a b)          1
    -> Matrix e a b -> Matrix e c b                2
select m y = join y id . m                        3

```

Listing 3.14: select in terms of matrices



We know upfront from the definition that a (possibly) expensive computation is taking place where one of the matrices is the identity. But, from the type of  $m$  we know that it can be  $m = \text{Fork } x \text{ } z$  for some  $x$  and  $z$  and the implementation can take advantage of this:

$$\begin{aligned}
 & \text{join } y \text{ id} . m \\
 = & \quad \{ m = \text{Fork } x \text{ } z \} \\
 & \text{join } y \text{ id} . \text{Fork } x \text{ } z \\
 = & \quad \{ \text{divide-and-conquer (10)} \} \\
 & y . x + \text{id} . z \\
 = & \quad \{ \text{identity law} \} \\
 & y . x + z
 \end{aligned}$$

Thus one gets

```
||      select (Fork x z) y = y . x + z                                     1
```

Listing 3.15: **Fork x z pattern match case**

gaining in efficiency because  $x$  is necessarily smaller than the original  $m$ . Note that  $x$  and  $z$  above can be, on their own, joins. In this case, by the abide law (12) one gets  $m = \text{Join} (\text{Fork } x \text{ } c) (\text{Fork } z \text{ } d)$  which lets us pattern match one level deeper and, benefiting from the divide-and-conquer law, end up with:

$$\begin{aligned}
 & \text{join } y \text{ id} . m \\
 = & \quad \{ m = \text{Join} (\text{Fork } x \text{ } c) (\text{Fork } z \text{ } d) \} \\
 & \text{join } y \text{ id} . \text{Join} (\text{Fork } x \text{ } c) (\text{Fork } z \text{ } d) \\
 = & \quad \{ \text{fusion (8)} \} \\
 & \text{Join} (\text{join } y \text{ id} . \text{Fork } x \text{ } c) (\text{join } y \text{ id} . \text{Fork } z \text{ } d) \\
 = & \quad \{ \text{divide-and-conquer (10) twice; identity twice} \} \\
 & \text{Join} (y . x + c) (y . z + d)
 \end{aligned}$$

Putting everything together, one gets the following more efficient implementation:

```

||      select :: (...) => Matrix e c (Either a b)                                1
||      -> Matrix e a b -> Matrix e c b                                         2
||      select (Fork x z) y = y . x + z                                         3
||      select (Join (Fork x c) (Fork z d)) y = join (y . x + c) (y . z + d) 4
||      select m y = join y id . m                                             5

```

Listing 3.16: **Final result**

Moving from functions to matrices has allowed us to express probability distribution more elegantly and algebraically 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 and can not be skipped, the [LAoP](#) discipline allowed us to calculate a more efficient select operator that mimics the speculative execution of [SAFs](#).

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

### 3.6 PROBABILISTIC PROGRAMMING EDSL & SAMPLING

Matrices are an exhaustive approach to probabilistic computations, which makes it unfeasible when dealing with very large ones. Although they provide an elegant encoding and are amenable to calculational manipulation, using matrices for doing probabilistic programming can incur a cognitive overhead, since programs are not written in a very declarative, straightforward way. Not to mention that, they are not able to express probabilistic programs that operate with types that are arbitrarily infinite, like lists.

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, it didn't made it possible to benefit from any of [SAFs](#)' properties.

With this in mind, a practical way where the speculative execution nature of [SAFs](#) or their static analysis capabilities could be useful was explored. Recurring to the Free Selective Functor construction, mentioned by [Mokhov et al. \(2019\)](#), a simple [eDSL](#) for doing probabilistic programming was designed. This [eDSL](#) has, according to [Ścibior et al. \(2015\)](#); [Gordon et al. \(2014\)](#); [van de Meent et al. \(2018\)](#), the minimum requirements to represent a distribution in a functional programming language, which are having: a set of standard distributions as building blocks; a Monad instance; a conditioning function; and a way of sampling from a given distribution. Since we are not working with Monads we relax the second requirement to only having a Selective instance.

Free constructions allow us to focus on the internal aspects of the effect under consideration and receive the desired applicative or monadic (in this case it is the selective) computation structure for free, i.e. without the need to define custom instances or prove laws. Given this we just need to specify the set of effects (building-blocks) of the kind of computations we wish to represent. The listing below shows how we can express the different building-blocks of our language in this way:

```

import Control.Selective.Free      1
import Control.Selective          2
                                   3
data Primitives a where          4
    Uniform :: [a] -> Primitives a      5
    Categorical :: [(a, Double)] -> Primitives a      6
    Normal :: Double -> Double -> (Double -> a) -> Primitives a      7
    Beta :: Double -> Double -> (Double -> a) -> Primitives a      8
    Gamma :: Double -> Double -> (Double -> a) -> Primitives a      9
    deriving Functor              10
                                   11
type Dist a = Select Primitives a      12

```

Listing 3.17: eDSL primitive building-blocks

Now that we have the first two of the above mentioned requirements, we need to have a conditioning function and a way to sample from a given `Dist` type, in order to have a minimal language suited for probabilistic programming. A conditioning function conditions a distribution with respect to a predicate or condition that is defined over the variables in the program. In particular, every valid execution of the program must satisfy all conditions that occur along the execution. Knowing this, the following function was implemented:

```

-- | It provides information about the outcome of testing @@ on some input  1
--   @@,
-- encoded in terms of the coproduct injections without losing the input    2
-- @@ itself.                                                                3
grdS :: Applicative f => f (a -> Bool) -> f a -> f (Either a a)      4
grdS f a = selector <$> applyF f (dup <$> a)                          5
    where                                                                    6
        dup x = (x, x)                                                    7
        applyF fab faa = bimap <$> fab <*> pure id <*> faa            8
        selector (b, x) = bool (Left x) (Right x) b                    9
                                   10
-- | McCarthy's conditional, denoted p -> f,g is a well-known functional  11
-- combinator, which suggests that, to reason about conditionals, one may  12
-- seek help in the algebra of coproducts.                                13
--                                                                           14
-- This combinator is very similar to the very nature of the 'select'      15
-- operator and benefits from a series of properties and laws.            16

```

```

condS :: Selective f => f (b -> Bool) -> f (b -> c) -> f (b -> c) -> f b -> f 17
    c
condS p f g = (\r -> branch r f g) . grdS p 18
19
condition :: Dist (a -> Bool) -> Dist a -> Dist (Maybe a) 20
condition c = condS c (pure (const Nothing)) (pure Just) 21

```

Listing 3.18: Conditioning function

As we can see from the listing above, in order to implement the conditioning function we used McCarthy's conditional that, interestingly enough, makes a nice connection with the previous results and, as we will see later, will be a ubiquitous pattern when writing programs in our eDSL.

One of the benefits of using an eDSL is the capacity of providing any number of different interpretations to the same program. For instance, we could interpret the `Dist` data-type so as to return the probabilities of every possible output, i.e. an exhaustive interpretation, or we could interpret it by sampling from every primitive distribution until we reach a concrete result, i.e. a sampling interpretation. The later offers us a way of sampling from a given distribution, from which we can then apply any inference algorithm to infer the probability of a given event. The listing below shows how we can achieve the last requirement of our minimal probabilistic programming eDSL:

```

import qualified System.Random.MWC.Probability as MWCP 1
2
-- forward sampling 3
runToIO :: Dist a -> IO a 4
runToIO = runSelect interpret 5
where 6
    interpret (Uniform l) = do 7
        c <- MWCP.createSystemRandom 8
        i <- MWCP.sample (MWCP.uniformR (0, length l - 1)) c 9
        return (l !! i) 10
    interpret (Categorical l) = do 11
        c <- MWCP.createSystemRandom 12
        i <- MWCP.sample (MWCP.categorical (V.fromList . map snd $ l)) c 13
        return (fst $ l !! i) 14
    interpret (Normal x y f) = do 15
        c <- MWCP.createSystemRandom 16
        f <$> MWCP.sample (MWCP.normal x y) c 17
    interpret (Beta x y f) = do 18

```

```

    c <- MWCP.createSystemRandom                                19
    f <$> MWCP.sample (MWCP.beta x y) c                        20
    interpret (Gamma x y f) = do                               21
    c <- MWCP.createSystemRandom                                22
    f <$> MWCP.sample (MWCP.gamma x y) c                        23
                                                                    24
sample :: Dist a -> Int -> Dist [a]                            25
sample r n = sequenceA (replicate n r)                        26

```

Listing 3.19: Sampling function

### 3.6.1 Examples of Probabilistic Programs

Now that we have our minimal language lets see what types of probabilistic programs we can write in it. We'll start with a simple coin toss example and build up from there. In order to lift a primitive distribution into our `Dist` data-type we use `liftSelect` offered by the `Selective` library.

```

categorical :: [(a, Double)] -> Dist a                        1
categorical = liftSelect . Categorical                          2
                                                                    3
bernoulli :: Double -> Dist Bool                             4
bernoulli x = categorical [(True, x), (False, 1 - x)]        5
                                                                    6
data Coin = Heads | Tails                                     7
    deriving (Show, Eq, Ord, Bounded, Enum)                8
                                                                    9
-- Throw 2 coins                                              10
t2c :: Dist (Coin, Coin)                                       11
t2c = let c1 = bool Heads Tails <$> bernoulli 0.5              12
        c2 = bool Heads Tails <$> bernoulli 0.5              13
        in (,) <$> c1 <*> c2                                    14
                                                                    15
-- Throw 2 coins with condition                               16
t2c2 :: Dist (Maybe (Bool, Bool))                          17
t2c2 = let c1 = bernoulli 0.5                                    18
        c2 = bernoulli 0.5                                    19

```

```

    in condition (pure (uncurry (| |))) ((,) <$> c1 <*> c2)

```

Listing 3.20: Coin toss

When sampling 10 results from this example distributions we obtain the following values:

```

> runToIO $ sample t2c 10
[(Tails,Heads),(Heads,Tails),(Heads,Heads),(Heads,Heads),(Heads,Heads),(Tails,
  Heads),(Heads,Tails),(Tails,Heads),(Heads,Heads),(Heads,Heads)]
>
> runToIO $ sample t2c2 10
[Just (True,True),Just (False,True),Just (False,True),Just (True,False),
  Nothing,Just (True,False),Nothing,Nothing,Just (False,True),Just (True,
  True)]
>

```

Listing 3.21: Coin toss results

We can see that the conditioning function is limiting the results to only those that satisfy the condition.

Lets look at an example that cannot be expressed by using our [LAoP](#) matrix library:

```

-- | Throw @n@ coins
throw :: Dist [Coin]
throw =
  let toss = bernoulli 0.5
  in condS (pure (== Heads))
    (flip (:) <$> throw)
    (pure (: []))
    (bool Heads Tails <$> toss)

{-
Result:
> runToIO $ sample throw 10
[[Heads],[Tails,Tails,Tails,Heads],[Tails,Heads],[Tails,Tails,Heads],[Heads
  ],[Heads],[Tails,Heads],[Tails,Heads],[Heads],[Tails,Heads]]
>
-}

```

Listing 3.22: Throw coins indefinitely until Heads comes up

We can see from this example that programs written using only the Selective abstraction are less idiomatic than those that take full advantage of Monads. For instance, we do not have access to `do`-notation neither are capable of sequencing computations which values depend from other computations. We can, however, use the Applicative nature of [SAFs](#) and McCarthy's Conditional to recover part of the desired expressivity, as we can see from the example below:

```

uniform :: [a] -> Dist a                                1
uniform = liftSelect . Uniform                          2

die :: Dist Int                                         3
die = uniform [1..6]                                   4

-- | This models a simple board game rule where, at each turn,
-- two dice are thrown and, if the value of the two dice is equal,
-- the face of the third dice is equal to the other dice,
-- otherwise the third die is thrown and one piece moves
-- the number of squares equal to the sum of all the dice.
diceThrow :: Dist Int                                   5
diceThrow =                                             6
  condS (pure $ uncurry (==))                          7
    ((\c (a, b) -> a + b + c) <$> die) -- Speculative dice throw 8
    (pure (\(a, _) -> a + a + a))                    9
    ((,) <$> die <*> die) -- Parallel dice throw        10
                                                    11
{-                                                    12
Result:                                                    13
> runToIO $ sample diceThrow 20                          14
[2,5,7,11,12,13,8,8,4,13,9,6,9,9,10,11,14,6,13,12]      15
>                                                        16
List of die throws which have length 2 or 3:              17
[[1,1],[3,1,1],[3,1,3],[5,2,4],[6,2,4],[5,6,2],[4,4],    18
[3,4,1],[2,2],[6,5,2],[1,4,4],[3,1,2],[6,2,1],[1,3,5],  19
[5,4,1],[2,5,4],[4,5,5],[1,3,2],[2,5,6],[6,6]]          20
-}]                                                       21

```

Listing 3.23: Throw N coins

With this example we can clearly see that, although code written in this fashion is not as expressive or idiomatic as we would wish, it is capable of allowing it to benefit from Applicative and Selective capabilities.

It is important to emphasise the usefulness of the McCarthy's Conditional usage since, without it one is prone to write programs that repeat unnecessary computations. `ifS` is a popular combinator present in the Selective library that lifts the *if-then-else* primitive to the Applicative level, and one might try to use it when writing simple programs such as:

```
-- | Bad program
badThrow :: Int -> Dist [Coin]
badThrow 0 = pure []
badThrow n =
  let toss = bernoulli 0.5
  in ifS toss
      ((:) <$> toss <*> badThrow (n - 1))
      (pure [])
{-
Total number of effects:
> getEffects (throw 1)
[Categorical [(() ,0.5), ((),0.5)], Categorical [(() ,0.5), ((),0.5)]]
>
-}
```

Listing 3.24: **Bad program**

We can see that the `toss` effect is being repeated because we do not have a way to forward its conditional result to the next computation and this leads to the program not behaving as expected.

### 3.6.2 Sampling and Inference Algorithms

Probabilistic inference is the problem of computing the explicit representation of the probability distribution implicitly defined in a probabilistic program. For example, with respect to the distribution, we might want to calculate the expected value of some function  $f$ . Alternatively, we may want to simply draw a set of samples from the distribution to test some other system that expects inputs to follow the modelled distribution.

This subsection will present two sampling / inference algorithms implemented on top of the probabilistic programming `eDSL` and describe some of the limitations found.



The first one is the Monte Carlo Sampling method which is very simple, as you can see below:

```

-- monte carlo sampling/inference                                1
monteCarlo :: Ord a => Int -> Dist a -> Dist [(a, Double)]      2
monteCarlo n d =                                                3
  let r = sample d n                                           4
      in map (\l -> (head l, fromIntegral (length l) / fromIntegral n)) . group 5
        . sort <$> r                                           6
                                                                    6
{-                                                                    7
Result:                                                            8
> runToIO $ monteCarlo 2000 t2c                                   9
[( (Heads,Heads),0.2435), ( (Heads,Tails),0.248), ( (Tails,Heads),0.249), ( (Tails, 10
  Tails),0.2595)]
-}                                                                    11

```

Listing 3.25: Partial monadic bind function

This method samples  $n$  values from a given distribution and calculates the relative probability of each event.

Other sampling method called Rejection Sampling ([Tobin, 2018](#)) can similarly be achieved:

```

rejection :: (Bounded c, Enum c, Eq c) => ([a] -> [b] -> Bool) -> [b] -> Dist 1
  c -> (c -> Dist a) -> Dist c
rejection predicate observed proposal model = loop where      2
  len = length observed                                       3
  loop =                                                       4
    let parameters = proposal                                 5
        generated = sample (bindS parameters model) len      6
        cond = predicate <$> generated <*> pure observed    7
    in ifS cond                                               8
        parameters                                           9
        loop                                                  10

```

Listing 3.26: Partial monadic bind function

This method and more complex others require monadic capabilities which make the proposed solution quite inefficient. This seems to be a limitation of the selective abstraction. Although it is still possible to implement such algorithms, we can only deal with discrete, finite data types which limit the domain we are dealing with.

### 3.7 SAMPLING AS A CONCURRENCY PROBLEM

The work described above dealt only with the syntactic aspect of probabilistic programming, in other words, only explored what basic operations the Free Selective Construction was able to perform in order to write probabilistic programs. In the sense of the IO monad, a simple interpretation of these operations was provided simply to illustrate how simple results can be achieved.

As previously stated, the sampling approach was used to try to take advantage of the capabilities of [SAFs](#). Programming using an [eDSL](#) which enforces programming only recurring to selective combinators, allows one to capitalise on all potential benefits. However because the IO monad is inherently sequential, any independent computation loses the chance of parallel/speculative effect execution.

The [1.3.2](#) section addressed how [FP](#) and functional languages are a great vessel for probabilistic programming. However, the use of the IO monad disables the ability to exploit parallelism in the sampling of two independent variables. Some authors suggest improvements to some of the drawbacks of the monad in use, as also stated in section [1.3.2](#), but no specific practical use of them has ever been seen. This may be the case because no one has ever tackled this issue from the correct angle. This section aims to clarify why and explore how to look at the problem of sampling in a simpler and more creative way, providing a nice approach that goes hand in hand with the existing state-of-the-art as well as presenting nice preliminary results.

#### 3.7.1 *The Concurrency Monad*

There are several references to concurrency monads in literature. One of the first works in this regard was by [Claessen \(1999\)](#), where by describing a monad transformer in Haskell, he introduces a groundbreaking way of modelling concurrency. In essence, a concurrency monad can be seen as a way to introduce concurrency to a (functional) programming language without adding specialised primitives to the compiler. In other words, the concept of concurrency constructs is shifted towards the programmer.

Solutions such as [Claessen \(1999\)](#) and [Scholz \(1995\)](#), among others, although they have somewhat different models, rely on the basic notion of interleaving processes via continuation. Continuations are capable of preserving the flowing essence of a process, allowing a process to be stopped and resumed. Many concurrency monads are provided with a collection of primitive constructs, like `fork`, to make the concurrency explicit. Thus, a common trait of these programming models is that they are based on a concurrency-monad-like substratum; they behave like lightweight threads with cooperative scheduling.

Marlow et al. (2014) offer a different, fascinating alternative solution, where their approach is especially useful for programming against external data sources without the use of explicit constructs of concurrency, called Haxl. It assumes that external access to data is read-only, and that the order does not matter so that it can be done in parallel. They present an extension of the concept of concurrency monads where concurrency is implied in the Applicative abstraction, which, unlike previous formulations, takes advantage of the fact that the arguments to ( $<*>$ ) are independent and can therefore be inspected. This new feature can also be interpreted as some form of static analysis, and allows multiple requests to be batched together. Recently, this solution has also been given speculative execution capabilities (Mokhov et al., 2019), which makes it very interesting to the scope of this dissertation.

### 3.7.2 Sampling

In order to extend the probabilistic programming eDSL with parallel and speculative execution capabilities without having to be explicit about it, we suggest that sampling can be seen as a problem of competition. So let's start by illustrating what "see sampling as a concurrency problem" implies.

Effective access to multiple remote data sources requires concurrency, and that usually requires the programmer to intervene and program the concurrency explicitly. But when the business logic is all about reading data from external sources, the programmer doesn't worry about the order in which the data accesses occur. This is the scenario defined by Marlow et al. (2014) and it is the one which is advantageous to imply its solution. They go even further and show how to add some sort of caching capabilities. Sampling from a probability distribution is somewhat similar to collecting data from an external source thus a similar approach can be used to effectively conduct sampling. The source of randomness is the external data source to be read, and because it is random, the order by which simultaneous (independent) samples are performed does not matter or cause additional side effects. We can already see how these two models are very similar, but there is a small difference, which is that it is not possible to repeat a data access request, in the case of sampling, because we get a different result every time we access the random source. This fact does not allow one to build a caching system like the one in Haxl, since it would have a negative impact on the sampling quality itself.

In view of this, we conclude that sampling can be seen as a more general concurrency problem than the one solved by Haxl.

## 3.7.3 Implementation

An approach very similar to the one presented by Marlow et al. (2014) was used in order to implement the solution. As it was mentioned in the previous section, we need to think about sampling as a concurrency problem and that means that our "external data access requests" are sampling requests. With this in mind, each request can be either Done or Blocked. So, in general, a computation in our data type will be a sequence of Blocked requests ending in a Done with the return value, like so:

```

data BlockedRequest = forall a. BlockedRequest (Request a) (IORef (Status a)) 1
                                                                                   2
data Status a = NotFetched | Fetched a 3
                                                                                   4
type Prob = Double 5
                                                                                   6
data Request a where 7
    Uniform      :: [x] -> (x -> a) -> Request a 8
    Categorical  :: [(x, Prob)] -> (x -> a) -> Request a 9
    Normal       :: Double -> Double -> (Double -> a) -> Request a 10
    Beta         :: Double -> Double -> (Double -> a) -> Request a 11
    Gamma        :: Double -> Double -> (Double -> a) -> Request a 12
                                                                                   13
-- A computation is either completed (Done) or Blocked on pending sample 14
-- requests
data Result a = Done a | Blocked (Seq BlockedRequest) (Fetch a) deriving 15
    Functor 16
newtype Fetch a = Fetch {unFetch :: IO (Result a)} deriving Functor 17

```

Listing 3.27: Fetch Data Type

The Fetch data type is actually a monad (since it is wrapped around IO) and follows the continuation monad formulation. It is also worth noting that this idea is an instance of a free monad.

There's one thing missing from the implementation and that is a way of introducing concurrency. As we saw before, the probabilistic interpretation of the Applicative abstraction expresses statistical independence and thus is suitable for allowing to create concurrency within our data structure. When computing in Fetch is performed using the ( $\langle * \rangle$ ) operator, all ( $\langle * \rangle$ ) arguments may be explored to look for Blocked computing, which creates the potential that a computing may be blocked on several items at the same time. This is in

contrast to the monadic bind operator, ( $\gg=$ ), which does not allow both arguments to be examined, since the right side can not be evaluated without the left side result.

With this being said, the following instance was written:

```
instance Applicative Fetch where
  pure = return

  Fetch iof <*> Fetch iox = Fetch $ do
    rf <- iof
    rx <- iox
    return $ case (rf, rx) of
      (Done f, _)          -> f <$> rx
      (_, Done x)          -> ($x) <$> rf
      (Blocked bf f, Blocked bx x) -> Blocked (bf <> bx) (f <*> x) --
      batching parallel requests
```

Listing 3.28: Fetch Applicative instance

This static analysis feature can also be used with the speculative execution capabilities of the Selective Abstraction. This is a very interesting and novel addition to the functional probabilistic programming domain that, theoretically, allows for better performance in programs that branch on a given sample result.

```
instance Selective Fetch where
  select (Fetch iox) (Fetch iof) = Fetch $ do
    rx <- iox
    rf <- iof
    return $ case (rx, rf) of
      (Done (Right b), _)          -> Done b -- abandon the second
      computation
      (Done (Left a), _)          -> ($a) <$> rf
      (_, Done f)                 -> either f id <$> rx
      (Blocked bx x, Blocked bf f) -> Blocked (bx <> bf) (select x f) --
      speculative execution
```

Listing 3.29: Fetch Selective instance

## 3.8 SUMMARY

---

APPLICATIONS

---

## 4.1 LAOP SPRINKLER EXAMPLE

Probabilistic programming arises naturally from functional programming once we replace “sharp” functions by probabilistic ones, represented by stochastic matrices, also known as Markov chains [Oliveira \(2012\)](#). Take a look at this example from the [Wikipedia \(2020\)](#). Suppose the following predicates modelling the behaviour of a sprinkler are defined, where  $S$  (sprinkler on/off),  $R$  (raining or not) and  $G$  (grass wet or not) are Booleans:

$$\begin{aligned} \text{sprinkler} &:: R \rightarrow S & \text{grass} &:: (S, R) \rightarrow G \\ \text{sprinkler } r &= \text{not } r & \text{grass } (s, r) &= s \parallel r \end{aligned}$$

The second predicate tells that the grass will be wet if and only if either it is raining or the sprinkler is on. The first tells that the sprinkler is on *iff* it is not raining. Composing these two predicates we see that rain completely determines the state of the grass:

$$\text{grass } (\text{sprinkler } r, r) = \text{not } r \parallel r = \text{True}$$

Looking at the diagram below, where  $(\nabla)$  can be seen as equal to  $(\&\&\&)^1$ , we see that the system has two possible states in  $(G, (S, R))$  — either  $(\text{True}, (\text{True}, \text{False}))$  or  $(\text{True}, (\text{False}, \text{True}))$  — the grass being wet in both. So it will melt because of being wet all the time.

$$\begin{array}{c} (G, (S, R)) \\ \uparrow_{\text{grass } \nabla \text{ id}} \\ (S, R) \\ \uparrow_{\text{sprinkler } \nabla \text{ id}} \\ R \\ \uparrow_{\text{rain}} \\ () \end{array}$$

---

<sup>1</sup> From `Control.Arrow`, specialised to  $(\rightarrow)$

Clearly, this deterministic interpretation of the diagram does not correspond to reality, but its stochastic interpretation will do. For this, we just need to regard the arrows as denoting stochastic matrices and not pure functions, for instance<sup>2</sup>

$$R \xrightarrow{\text{sprinkler}} S = \left[ \begin{array}{c|c} 0.60 & 0.99 \\ \hline 0.40 & 0.01 \end{array} \right]$$

$$(S, R) \xrightarrow{\text{grass}} G = \left[ \begin{array}{cccc} 1.00 & 0.20 & 0.10 & 0.01 \\ 0 & 0.80 & 0.90 & 0.99 \end{array} \right]$$

This describes a probabilistic system *reactive* to the rain. Once its distribution becomes known, eg.

$$1 \xrightarrow{\text{rain}} R = \left[ \begin{array}{c} 0.80 \\ 0.20 \end{array} \right]$$

one immediately gets the distribution of the overall state, given by column vector

$$1 \xrightarrow{\text{state}} (G, (S, R)) = \begin{array}{rcccl} & G & S & R & \\ & \hline & & \text{off} & \text{no} & 0.4800 \\ & & & \text{yes} & 0.0396 \\ \text{dry} & \hline & & \text{on} & \text{no} & 0.0320 \\ & & & \text{yes} & 0.0000 \\ & \hline & & \text{off} & \text{no} & 0.0000 \\ & & & \text{yes} & 0.1584 \\ \text{wet} & \hline & & \text{on} & \text{no} & 0.2880 \\ & & & \text{yes} & 0.0020 \\ & \hline \end{array} \quad (13)$$

which is calculated following the diagram. Consider the following matrices:

```
rain :: Matrix Prob () R      1
sprinkler :: Matrix Prob R S  2
grass :: Matrix Prob (S, R) G 3
```

Listing 4.1: Example matrices

(where type `Prob = Double`) encoded in the [LAoP](#) library, where we also free the types involved from the strict Boolean model, already visible in (13).<sup>3</sup>

The distribution of the overall state displayed above is given by the expression

```
state = compose grass sprinkler rain 1
```

Listing 4.2: State matrix

<sup>2</sup> For easy reference we follow the Wikipedia example closely.

<sup>3</sup> So instead of `G = Bool` we have `G = Dry | Wet` and so on.

where

```

compose :: (...)
    => Matrix e (c, d) b
    -> Matrix e d c
    -> Matrix e a d
    -> Matrix e a (b, (c, d))
compose g s r = tag g . tag s . r

tag :: (...) => Matrix e a b -> Matrix e a (b, a)
tag f = kr f id

```

Listing 4.3: State matrix composition function

Note the role of the *tag* operation, which for functions amounts to  $\text{tag } f \ x = (f \ x, \ x)$ , that is, the output of  $f$  is paired with its input. Combinator *compose* iterates this operation across compositions so as to get an account of all inputs and outputs, as is usual in Bayesian networks.<sup>4</sup>

Let `wet :: Matrix Prob () G`, `dry :: Matrix Prob () G`, `no :: Matrix Prob () R` (and so on) be the *points* of the data types involved in the model. Also projections `fstM` and `sndM` are used to obtain the first and second components of the paired matrices, respectively. Evaluating the overall probability of the grass being wet is given by the scalar<sup>5</sup>

```

grassWet = tr wet . fstM . state -- = 44.84%

```

Listing 4.4: Probability of grass being wet calculation

## 4.2 EDSL SPRINKLER EXAMPLE

The last section showed how it is possible to do probabilistic programming by using matrices and the [LAoP](#) discipline. In this one, the same example will be shown but, this time, using the probabilistic programming [eDSL](#) designed in section 3.6.

Instead of matrices, now, one is only capable of using probabilistic functions of type  $a \rightarrow \text{Dist } b$ . So, the functions equivalent to the sprinkler, grass and rain matrices are given below:

```

data R = No | Yes
    deriving (Eq, Show, Enum, Bounded, Ord)
data S = Off | On

```

<sup>4</sup> This generic combinator is inspired in the *left tagging* relational operator of [Bussche \(2001\)](#).

<sup>5</sup> Recall that scalars are matrices of type  $() \rightarrow ()$ .



```

    deriving (Eq, Show, Enum, Bounded, Ord)           4
data G = Dry | Wet                                   5
    deriving (Eq, Show, Enum, Bounded, Ord)           6
                                                    7
sprinkler :: R -> Dist S                             8
sprinkler No  = categorical [(Off, 0.6), (On, 0.4)]   9
sprinkler Yes = categorical [(Off, 0.99), (On, 0.01)] 10
                                                    11
grass :: (S, R) -> Dist G                             12
grass (Off, No)  = categorical [(Dry, 1), (Wet, 0)]   13
grass (Off, Yes) = categorical [(Dry, 0.2), (Wet, 0.8)] 14
grass (On, No)   = categorical [(Dry, 0.1), (Wet, 0.9)] 15
grass (On, Yes)  = categorical [(Dry, 0.01), (Wet, 0.99)] 16
                                                    17
rain :: Dist R                                         18
rain = categorical [(No, 0.8), (Yes, 0.2)]             19

```

Listing 4.5: Example probabilistic functions

In the last section we saw the importance of the tag and compose combinators. They allowed us to compose distribution matrices and calculate their joint probability easily. By using our **eDSL**, we are only allowed to use **SAFs** capabilities, which means we do not have the required monadic capacity to iterate the equivalent tag combinator, across compositions, in order to get distribution computation of the whole state. As we can see below we obtain a nested Dist type which makes it awkward to deal with.

```

class Functor f => Strong f where                    1
    rstr :: (f a, b) -> f (a, b)                     2
    rstr (fa, b) = fmap (, b) fa                      3
                                                    4
    lstr :: (b, f a) -> f (b, a)                     5
    lstr (b, fa) = fmap (b, ) fa                     6
                                                    7
instance Strong (Select Primitives)                 8
                                                    9
tag :: (a -> Dist b) -> (a -> Dist (b, a))           10
tag f = fmap rstr $ (,) <$> f <*> id                 11
                                                    12
states :: Dist (Dist (Dist (G, (S, R))))             13

```

```
stateS = fmap (tag grass) . tag sprinkler <$> rain
```

Listing 4.6: **tag combinator**

We can instead, take advantage of the fact that the data types used in our model are Enum, Bounded, Eq, i.e. countable, and thus use the (*inefficient*) bindS function in order to compose the probabilistic functions and obtain the desired state distribution.

```
state :: Dist (G, (S, R))
state = bindS rain (\r -> bindS (sprinkler r) (\s -> bindS (grass (s,r)) (\g
    -> pure (g, (s,r)))))

{-
Result:
> runToIO $ monteCarlo 2000 state
[ ((Dry, (Off,No)), 0.4835), ((Dry, (Off,Yes)), 4.45e-2), ((Dry, (On,No)), 3.9e-2), ((
    Wet, (Off,Yes)), 0.151), ((Wet, (On,No)), 0.28), ((Wet, (On,Yes)), 2.0e-3) ]
>
-}
```

Listing 4.7: **State distribution**

### 4.3 BENCHMARKS

To check whether the results obtained so far bring any kind of efficiency benefits, we compared the performance of matrix multiplication algorithms presented in other Haskell libraries and the one proposed in this thesis; compared the performance of using matrices as distributions versus using lists (distribution monad); and compared the applicative versus selective exhaustive approach too see that, even though matrices can not take full advantage of speculative execution of [SAFs](#), by understanding the fundamentals of the abstraction, it is possible to achieve a faster select operator.

Given this, Fig. 4.1 shows the key features of the testbed environment.

<b>Model</b>	Intel(R) Core(TM)2 Duo CPU P8600
<b>Base clock freq</b>	2.40GHz
<b>L1 cache</b>	64 KiB
<b>L2 cache</b>	3 MiB
<b>RAM</b>	2 x 4096MB (DDR3)
<b>OS</b>	Arch Linux

Figure 4.1: **Testbed environment**

#### 4.3.1 LAoP Matrix composition

By analysing the current ecosystem at the time of writing, namely by filtering data obtained from the Hackage repository, three libraries providing efficient matrix implementations stand out as the most embraced by the community: *hmatrix*, *matrix* and *linear*. The *Criterion* library was used to benchmark the different algorithms on randomly generated square matrices with dimensions ranging between 10 and 1600.

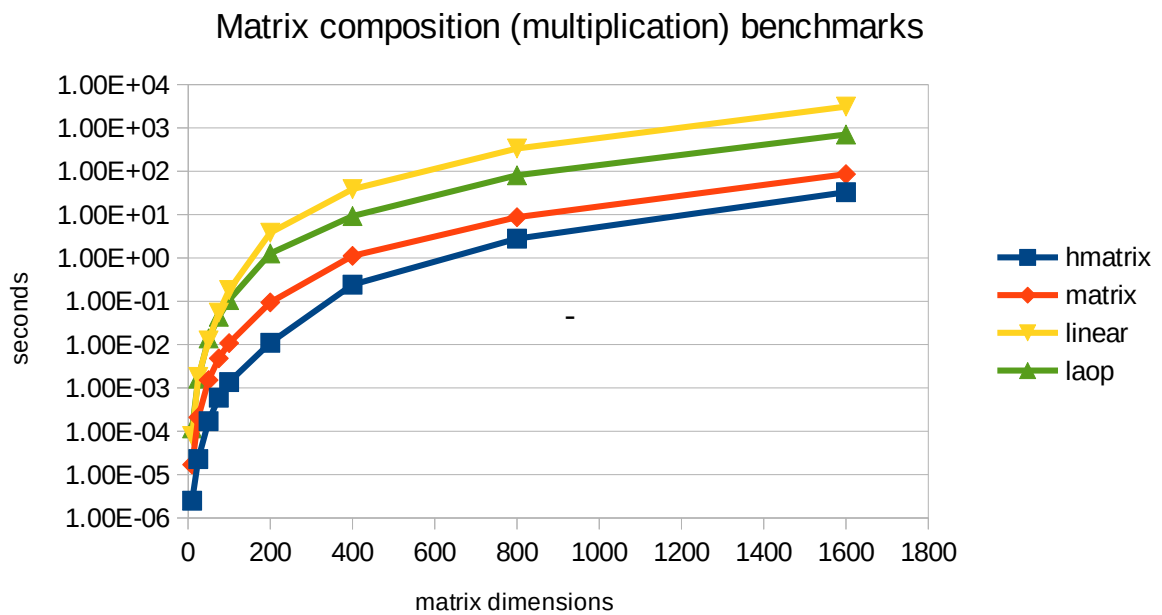


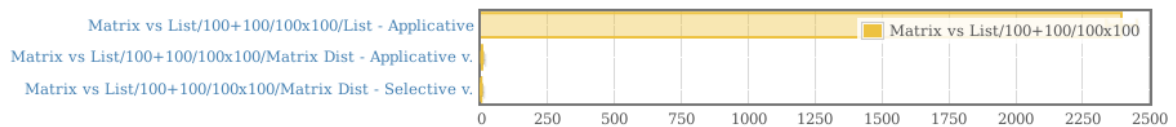
Figure 4.2: **Matrix composition benchmarks**

As can be seen in the plot of Figure 4.2, the *hmatrix* and *matrix* libraries are those that perform better. By observing their internal structure, one realises that they are a suitable representation for BLAS/LAPACK computations (Anderson et al., 1999), that is, they have been designed to efficiently exploit caches on modern cache-based architectures. A matrix in the *linear* library is defined as `Vector cols (Vector rows Double)` and does not take into account cache lengths or sizes, so it behaves much worse than the previous ones. Our structure does not take into account any low-level optimisations either, being unable to compete with those that do. Nevertheless, the implementation is *performant for a cache-oblivious approach* and behaves better (almost one order of magnitude better) than other types of simpler definitions.

4.3.2 *Distribution matrix vs distribution list monad*

The previous evaluation focused only on the performance of the matrix multiplication algorithm and compared with existing solutions to linear algebra however, in this section we will compare the use of matrices versus the use of lists as a representation of probability distributions, by comparing the performance of the select operator. Since both are exhaustive approaches to probabilistic programming, the comparisons will feature the applicative version of the select operator (where no computations are skipped) and see which solution performs best. Additionally, as we saw in section 3.5.2, it is possible to have a more efficient version of the select operator, under the matrix representation, which we will call the selective version. This version is also going to be benchmarked and compared against its applicative alternative.

The Figure and Listings below show the results output by the Criterion framework. The benchmarks were realized in the same settings as the previous one and, all matrices and lists were randomly generated.

Figure 4.3: **Matrix vs List - select operator**

benchmarking Matrix vs List/100+100/100x100/List - Applicative	1
time 2.464 s (2.411 s .. 2.515 s)	2
1.000 R^2 (1.000 R^2 .. 1.000 R^2)	3
mean 2.395 s (2.336 s .. 2.425 s)	4
std dev 56.98 ms (2.896 ms .. 69.67 ms)	5
variance introduced by outliers: 19% (moderately inflated)	6
	7
benchmarking Matrix vs List/100+100/100x100/Matrix Dist - Applicative v.	8
time 2.607 ms (2.007 ms .. 3.459 ms)	9
0.635 R^2 (0.538 R^2 .. 0.743 R^2)	10
mean 4.451 ms (3.848 ms .. 5.086 ms)	11
std dev 1.958 ms (1.634 ms .. 2.399 ms)	12
variance introduced by outliers: 98% (severely inflated)	13
	14
benchmarking Matrix vs List/100+100/100x100/Matrix Dist - Selective v.	15
time 1.259 ms (1.217 ms .. 1.329 ms)	16

	0.962 R^2	(0.898 R^2 .. 0.995 R^2)	17
mean	1.277 ms	(1.234 ms .. 1.383 ms)	18
std dev	210.0 micros	(124.3 micros .. 375.1 micros)	19
variance introduced by outliers:	87% (severely inflated)		20

Listing 4.8: Results

The first entry corresponds to the distribution monad presented by [Erwig and Kollmansberger \(2006\)](#) and we can see that this solution is the one that performs worse. The applicative version of the select operator, in the matrix solution performs a lot better even though all computations are performed. The last entry refers to the selective version of the select operator and we can see that it outperforms the applicative version.

#### 4.3.3 Sequential vs Concurrent Selective eDSL

In this section, the performance of each eDSL solution provided in the 3.6 and 3.7 sections is evaluated. In order to do so, three probabilistic programs were used: one that threw two hypothetical 50000 faced dice, returning both results; one that threw the same two dice but conditioned the result; and one similar to `diceThrow` in Listing 3.23 but using the same dice as the previous programs. As an example, see:

```

bigDie :: Dist Int                                1
bigDie = uniform [0 .. 50000]                    2

pg1 :: Dist (Int, Int)                            3
pg1 =                                            4
  let c1 = bigDie                                5
      c2 = bigDie                                6
  in (,) <$> c1 <*> c2                             7
                                              8
                                              9
pg2 :: Dist (Maybe (Int, Int))                   10
pg2 =                                            11
  let c1 = bigDie                                12
      c2 = bigDie                                13
      result = (,) <$> c1 <*> c2                  14
  in condition (uncurry (>)) result               15
                                              16
pg3 :: Dist Int                                   17
pg3 =                                            18

```

```

condS
  (pure $ uncurry (==))
  ((\c (a, b) -> a + b + c) <$> die) -- Speculative dice throw
  (pure (\(a, _) -> a + a + a))
  ((,) <$> bigDie <*> bigDie) -- Parallel dice throw

```

Listing 4.9: Programs used in evaluation

Each benchmark consists of performing forward sampling 2000 times. One thing to remember is that 4 general benchmarks have been made. The first two interpret `eDSL` in a sequential fashion, directly to `I0`, and first to the concurrency monad, and then to `I0`. In the other two, the programs are written directly to the concurrency monad. These were the results:

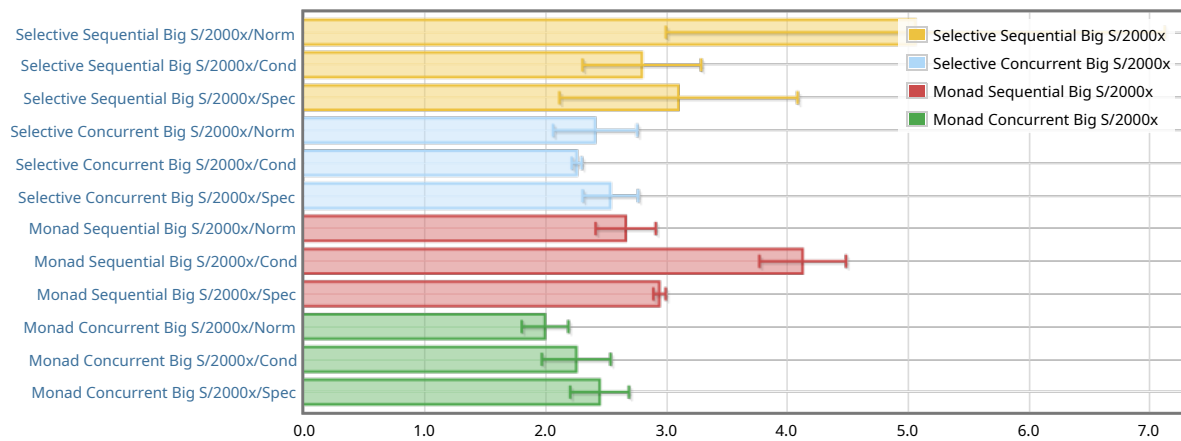


Figure 4.4: Benchmarks results (No delay)

As we can see from the graph, you can see that there are four distinct groups corresponding to the ones mentioned above. Absolute values are not that important, rather looking at them compared to each other will give us a much quicker and clearer interpretation of what's going on.

The baseline results are the first three, and from a quick glance, we can see that the concurrent version is faster than the sequential one, but there are cases where the gains are not that high. This is due to the fact that sampling, even from a large random distribution such as uniform 5000, is very fast and thus there is not a large gap between sequential and concurrent versions.

In order to unleash the full power of our solution, we decided to implement a small amount of delay (at least 100 microseconds) between sample requests in order to simulate a large sample from a hypothetical big data source. The findings have been as follows:

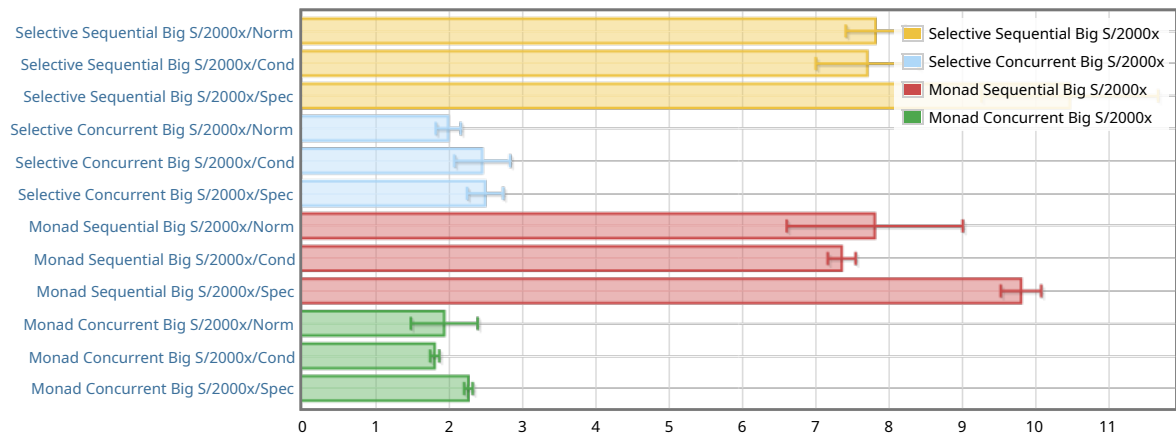


Figure 4.5: Benchmarks results (delay)

So, because of the delay that was implemented, the sequential versions are slower in this case, but the performance gains between the sequential and concurrent versions were larger and more evident than in the previous results.

#### 4.4 SUMMARY

---

## CONCLUSIONS AND FUTURE WORK

---

In this chapter we'll summarise the primary contributions of the dissertation and demonstrate that they defend the thesis. We'll also discuss some possible future work directions.

### 5.1 CONCLUSIONS

In this master thesis, we searched for a way to use [SAFs](#) in functional probabilistic programming, in particular to explore how this abstraction could be applied in a more efficient manner than the monadic `bind`. In order to do that, it was important to understand what these functors' probabilistic meaning was and what they could bring to the table. It is also important to consider what existing solutions and methods exist, and their drawbacks. We centered on the general theory of [LAoP](#) when searching for answers to the probabilistic sense, and studied the structure of stochastic matrices, where we found that [SAFs](#) is capable of conditioning random variables and branching a program in two different ways. We realised that, through this prism, [SAFs](#) generalises the already known McCarthy conditional and, theoretically, allows for parallel execution of conditional probability calculations by means of the divide-and-conquer block-matrix algebra law. A programming library of typed inductive block-matrices has been implemented in Haskell to demonstrate how the findings of the research can be applied in practice, and a number of examples and benchmarks have been made available, demonstrating that the theoretical gains are indeed valid. However, the use of matrices in probabilistic programming has its drawbacks, namely in relation to programs where the sample space has an explosion of potential states. Sampling of the probability distributions should be done in these cases. However, existing solutions rely heavily on the use of monads that are inherently sequential, leaving behind any possibility of parallel sampling wherever possible. In order to solve this problem, a small probabilistic programming [eDSL](#) has been developed using the free [SAF](#) construction. Using this method, we push the end-user to use selective combinators wherever possible, so that the compiler can be sure to take advantage of the capabilities of this abstraction. The crucial insight that made it possible was to realise that the problem of sampling could be reduced to a concurrent external data access problem. Knowing this, it was possible to implement



a solution close to the Haxl system and use it in our [eDSL](#). The findings were positive compared to the previous sequential version. In view of this, it can be inferred that, due to the nature of [SAFs](#), it was possible to use static analysis and speculative execution to write the select operator more efficiently than the monadic bind.

## 5.2 FUTURE WORK

The work presented in this dissertation highlights the themes of composition, abstraction and structure, these are relevant concepts in functional programming. The majority of features developed during this thesis are focused on important core aspects of static, purely functional languages. Monads, definitely the key driver of innovation, are difficult to express faithfully without a strong type system and functional purity, but as we have seen, these features have enabled us to have a great deal of reasoning power and have helped us to study novel abstractions in a different (probabilistic) context.

One of the most important avenues for future work would also be to make the various pieces of software that have been developed during this research production-ready. Such projects necessarily have a proof-of-concept feeling about them; they are meant to explore new fields and opportunities. Specifically, quadtrees ([Samet, 1984](#)) and their savings with respect to repetitive cells (pixels) are brought to mind by the block-oriented matrix type from the typed matrix programming library. A better matrix definition for sparsity could be more useful for sparse matrices with large zero blocks. The probabilistic programming [eDSL](#) can also be extended in order to support more distribution primitives and sampling algorithms. An interesting future direction is also to investigate improving the proposed solution in the light of the new found concurrency relationship, as well as studying parallelization strategies to improve performance.

---

## BIBLIOGRAPHY

---

- Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. *LAPACK Users' guide*, volume 9. Siam, 1999.
- 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.
- Jan Van den Bussche. Applications of Alfred Tarski's ideas in database theory. In *CSL'01*, pages 20–37, London, UK, 2001. Springer-Verlag. ISBN 3-540-42554-3.
- 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.
- Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(3): 313–323, 1999.
- 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](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>.
- Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, FOSE 2014, page 167–181, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328654. doi: 10.1145/2593882.2593900. URL <https://doi.org/10.1145/2593882.2593900>.
- 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).
- Mike Innes, Stefan Karpinski, Viral Shah, David Barber, PLEPS Saito Stenetorp, Tim Besard, James Bradbury, Valentin Churavy, Simon Danisch, Alan Edelman, et al. On machine

- learning and programming languages. Association for Computing Machinery (ACM), 2018.
- 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>, 2009.
- 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.
- J.N. Oliveira. Program design by calculation, 2008. Draft of textbook in preparation, current version: April 2018. Informatics Department, University of Minho.
- 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](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/d5bdbbc7afda54ff04049b6bdbbcffb67e>.
- 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>.
- Alberto Ruiz. Hmatrix: Numeric linear algebra, 2019. URL <http://hackage.haskell.org/package/hmatrix-0.20.0.0>.
- Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.
- Armando Santos and José N. Oliveira. Type your matrices for great good: A haskell library of typed matrices and applications (functional pearl). In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell 2020*, page 54–66, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380508. doi: 10.1145/3406088.3409019. URL <https://doi.org/10.1145/3406088.3409019>.

- Enno Scholz. A concurrency monad based on constructor primitives. <http://dx.doi.org/10.17169/refubium-22616>, 1995.
- 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.
- Peter Selinger. A brief survey of quantum programming languages. In *International Symposium on Functional and Logic Programming*, pages 1–6. Springer, 2004.
- Jane Street. A composable build system, 2018. URL <https://dune.build/>.
- S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, pages 184–207, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70639-7.
- 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.
- Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming, 2018.
- 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>.
- Wikipedia. Bayesian network, 2020. URL [https://en.wikipedia.org/wiki/Bayesian\\_network](https://en.wikipedia.org/wiki/Bayesian_network). (Accessed: 2020-02-16).



---

## TYPE SAFE LAOP MATRIX WRAPPER LIBRARY

---

```
{-# LANGUAGE AllowAmbiguousTypes #-}      1
{-# LANGUAGE ConstraintKinds #-}          2
{-# LANGUAGE DataKinds #-}               3
{-# LANGUAGE FlexibleContexts #-}         4
{-# LANGUAGE FlexibleInstances #-}        5
{-# LANGUAGE GADTs #-}                   6
{-# LANGUAGE GeneralizedNewtypeDeriving #- 7
{-# LANGUAGE InstanceSigs #-}            8
{-# LANGUAGE KindSignatures #-}          9
{-# LANGUAGE MultiParamTypeClasses #-}   10
{-# LANGUAGE NoStarIsType #-}            11
{-# LANGUAGE ScopedTypeVariables #-}     12
{-# LANGUAGE StandaloneDeriving #-}      13
{-# LANGUAGE TypeApplications #-}        14
{-# LANGUAGE TypeOperators #-}           15
{-# LANGUAGE UndecidableInstances #-}     16
{-# OPTIONS_GHC -fplugin GHC.TypeLits.KnownNat.Solver #-} 17
                                           18

module Matrix.Internal                    19
  ( Matrix (..),                        20
    NonZero,                            21
    ValidDimensions,                    22
    KnownDimensions,                    23
    fromLists,                          24
    toLists,                            25
    toList,                             26
    columns,                            27
    rows,                               28
```

```

    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 0 ~ '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          97
-- a list of lists @[e]@ (considered as rows).                  98
fromLists :: forall e c r. (LA.Element e, KnownDimensions c r) => [[e]] -> 99
    Matrix e c r
fromLists [] = error "Wrong list dimensions"                    100
fromLists l@(h : _) =                                          101
    let ccols = fromInteger $ natVal (Proxy :: Proxy c)        102
        rrows = fromInteger $ natVal (Proxy :: Proxy r)        103
        lrows = length l                                       104
        lcols = length h                                       105
    in if rrows /= lrows || ccols /= lcols                      106
        then error "Wrong list dimensions"                     107
        else M . HM.fromLists $ l                               108
                                                                109
-- | Matrix converter function. It builds a list of lists from  110
-- a 'Matrix'.                                                  111
--                                                                112
-- Inverse of 'fromLists'.                                       113
toLists :: (LA.Element e) => Matrix e c r -> [[e]]            114
toLists = HM.toLists . unMatrix                                 115
                                                                116

-- | Matrix converter function. It builds a list of elements from 117
-- a 'Matrix'.                                                  118
toList :: (LA.Element e) => Matrix e c r -> [e]               119
toList = concat . toLists                                       120
                                                                121

-- | Matrix converter function. It builds a matrix from a function. 122
fromF :: forall c r a b e. (Enum a, Enum b, Eq b, Num e, Ord e, LA.Element e, 123
    KnownNat c, KnownNat r) => (a -> b) -> Matrix e c r
fromF f =                                                        124
    let ccols = fromInteger $ natVal (Proxy :: Proxy c)        125
        rrows = fromInteger $ natVal (Proxy :: Proxy r)        126
        elementsA = take ccols $ map toEnum [0 ..]             127
        elementsB = take rrows $ map toEnum [0 ..]             128
        combinations = (,) <$> elementsA <*> elementsB         129
        combAp = map snd . L.sort . map (\(a, b) -> if f a == b then ((fromEnum 130
            a, fromEnum b), 1) else ((fromEnum a, fromEnum b), 0)) $
        combinations

```

```

    mList = buildList combAp rows                                131
  in tr $ fromLists mList                                       132
where                                                            133
  buildList [] _ = []                                           134
  buildList l r = take r l : buildList (drop r l) r            135
                                                                136
--                                                                137
-----

--  DIMENSIONS FUNCTIONS                                       138
--                                                                139
-----

                                                                140
-- | Obtain the number of columns of a matrix                   141
columns :: forall e c r. KnownNat c => Matrix e c r -> Integer 142
columns _ = natVal (Proxy :: Proxy c)                          143
                                                                144
-- | Obtain the number of rows of a matrix                      145
rows :: forall e c r. KnownNat r => Matrix e c r -> Integer    146
rows _ = natVal (Proxy :: Proxy r)                             147
                                                                148
fmapColumns :: forall b e a r. (Storable e, LA.Element e, KnownNat b) => 149
  Matrix e a r -> Matrix e b r
fmapColumns =                                                    150
  let cols = fromInteger $ natVal (Proxy :: Proxy b)           151
  in M . HM.reshape cols . HM.fromList . toList                152
                                                                153
fmapRows :: forall b e a c. (Storable e, LA.Element e, KnownDimensions c b) 154
  => Matrix e c a -> Matrix e c b
fmapRows =                                                       155
  let rows = fromInteger $ natVal (Proxy :: Proxy b)           156
  in tr . M . HM.reshape rows . HM.fromList . toList          157
                                                                158
--                                                                159
-----

--  MISC FUNCTIONS                                             160

```

```

--
-----
-- | 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 194
    => Matrix e c r
zeros = 195
    let ccols = fromInteger $ natVal (Proxy :: Proxy c) 196
        rrows = fromInteger $ natVal (Proxy :: Proxy r) 197
    in M $ HM.konst 0 (rrows, ccols) 198
    199
-- | One Matrix polymorphic definition 200
ones :: forall e c r. (KnownDimensions c r, Num e, LA.Container HM.Vector e) => Matrix e c r 201
    => Matrix e c r
ones = 202
    let ccols = fromInteger $ natVal (Proxy :: Proxy c) 203
        rrows = fromInteger $ natVal (Proxy :: Proxy r) 204
    in M $ HM.konst 1 (rrows, ccols) 205
    206
-- | Bang Matrix polymorphic Matrix 207
bang :: forall e c . (KnownNat c, Num e, LA.Container HM.Vector e) => Matrix e c 1 208
    e c 1
bang = 209
    let ccols = fromInteger $ natVal (Proxy :: Proxy c) 210
    in M $ HM.konst 1 (1, ccols) 211
    212
-- | Creates a square matrix with a given diagonal. 213
diag :: forall e c. (Num e, LA.Element e, KnownNat c) => [e] -> Matrix e c c 214
diag l = 215
    let c = fromInteger $ natVal (Proxy :: Proxy c) 216
        dims = length l 217
    in if c /= dims 218
        then error "Wrong list dimensions" 219
        else M . HM.diag . HM.fromList $ l 220
    221
-- 222
-----

-- BLOCK MATRIX FUNCTIONS (BIPRODUCT) 223
-- 224
-----

```

```

-- | 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

--
-----

```







```

instance 320
  {-# OVERLAPPABLE #-} 321
  ( ValidDimensions m k, 322
    KnownNat k, 323
    KnownNat ((m - 1) * k), 324
    KnownNat (m - 1), 325
    Num e, 326
    LA.Numeric e, 327
    LA.Container HM.Vector e, 328
    (1 + (m - 1)) ~ m, 329
    (k + ((m - 1) * k)) ~ (m * k), 330
    NonZero ((m - 1) * k), 331
    NonZero (m - 1), 332
    KhatriP1 e (m - 1) k 333
  ) => 334
  KhatriP1 e m k 335
  where 336
    kp1 = ones @e @k @1 -|- kp1 @e @(m - 1) @k 337
  338
  -- | Khatri Rao product right projection (inductive definition) 339
  class KhatriP2 e (k :: Nat) (m :: Nat) where 340
    kp2 :: Matrix e (m * k) k 341
  342
  instance 343
    {-# OVERLAPPING #-} 344
    ( Num e, 345
      LA.Element e, 346
      KnownNat k 347
    ) => 348
    KhatriP2 e k 1 349
    where 350
      kp2 = ident @e @k 351
  352
  instance 353
    {-# OVERLAPPABLE #-} 354
    ( (k + ((m - 1) * k)) ~ (m * k), 355
      ValidDimensions m k, 356
      NonZero ((m - 1) * k), 357
    )

```

```

    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)

-- | Khatri Rao product of two matrices (Pairing)
khatri ::
  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)
khatri 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 = 391
  let cols = fromInteger $ natVal (Proxy :: Proxy c) 392
  in M $ HM.reshape cols v 393
394
-- | Creates a 1-column matrix from a vector. 395
asColumn :: forall e r. (Storable e) => HM.Vector e -> Matrix e 1 r 396
asColumn = reshape @e @1 397
398
-- | Creates a 1-vector matrix from a vector. 399
asRow :: (Storable e, LA.Element e, KnownNat c) => HM.Vector e -> Matrix e c 400
  1
asRow = tr . asColumn 401

```

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

# B

---

## TYPE SAFE LAOP INDUCTIVE MATRIX DEFINITION LIBRARY

---

```
{-# LANGUAGE AllowAmbiguousTypes #-}      1
{-# LANGUAGE DataKinds #-}                2
{-# LANGUAGE FlexibleContexts #-}          3
{-# LANGUAGE FlexibleInstances #-}         4
{-# LANGUAGE GADTs #-}                   5
{-# LANGUAGE InstanceSigs #-}             6
{-# LANGUAGE MultiParamTypeClasses #-}    7
{-# LANGUAGE NoStarIsType #-}            8
{-# LANGUAGE ScopedTypeVariables #-}      9
{-# LANGUAGE StandaloneDeriving #-}       10
{-# LANGUAGE TypeApplications #-}         11
{-# LANGUAGE TypeFamilies #-}            12
{-# LANGUAGE TypeOperators #-}           13
{-# LANGUAGE UndecidableInstances #-}     14
                                           15
----- 16
-- |                                     17
-- Module      : Matrix.Internal          18
-- Copyright   : (c) Armando Santos 2019-2020 19
-- Maintainer  : armandoifsantos@gmail.com 20
-- Stability   : experimental             21
--                                                    22
-- The LAoP discipline generalises relations and functions treating them as 23
-- Boolean matrices and in turn consider these as arrows.                24
--                                                    25
-- __LAoP__ is a library for algebraic (inductive) construction and      26
-- manipulation of matrices
```

```

-- in Haskell. See <https://github.com/bolt12/master-thesis my Msc Thesis> 27
    for the
-- motivation behind the library, the underlying theory, and implementation 28
    details.
--
-- This module offers many of the combinators mentioned in the work of 29
-- Macedo (2012) and Oliveira (2012). 30
--
-- This is an Internal module and it is no supposed to be imported. 31
--
----- 32
----- 33
----- 34
----- 35
----- 36
module Matrix.Internal 37
( -- | This definition makes use of the fact that 'Void' is 38
  -- isomorphic to 0 and '()' to 1 and captures matrix 39
  -- dimensions as stacks of 'Either's. 40
  --
  -- There exists two type families that make it easier to write 41
  -- matrix dimensions: 'FromNat' and 'Count'. This approach 42
  -- leads to a very straightforward implementation 43
  -- of LAoP combinators. 44
  --
  -- * Type safe matrix representation 45
  Matrix (..), 46
  --
  -- * Primitives 47
  empty, 48
  one, 49
  junc, 50
  split, 51
  --
  -- * Auxiliary type families 52
  FromNat, 53
  Count, 54
  Normalize, 55
  --
  -- * Matrix construction and conversion 56
  FromLists, 57
  --
  -- * Matrix construction and conversion 58
  --
  -- * Matrix construction and conversion 59
  --
  -- * Matrix construction and conversion 60
  --
  -- * Matrix construction and conversion 61
  --
  -- * Matrix construction and conversion 62

```

fromLists,	63
toLists,	64
toList,	65
matrixBuilder,	66
row,	67
col,	68
zeros,	69
ones,	70
bang,	71
constant,	72
	73
-- * <i>Misc</i>	74
-- ** <i>Get dimensions</i>	75
columns,	76
rows,	77
	78
-- ** <i>Matrix Transposition</i>	79
tr,	80
	81
-- ** <i>Selective operator</i>	82
select,	83
	84
-- ** <i>McCarthy's Conditional</i>	85
cond,	86
	87
-- ** <i>Matrix "abiding"</i>	88
abideJS,	89
abideSJ,	90
	91
-- * <i>Biproduct approach</i>	92
-- ** <i>Split</i>	93
(==),	94
-- *** <i>Projections</i>	95
p1,	96
p2,	97
-- ** <i>Junc</i>	98
(   ),	99
-- *** <i>Injectons</i>	100

```

i1, 101
i2, 102
-- ** Bifunctors 103
(-|-), 104
(><), 105
106
-- ** Applicative matrix combinators 107
108
-- | Note that given the restrictions imposed it is not possible to 109
-- implement the standard type classes present in standard Haskell. 110
-- *** Matrix pairing projections 111
kp1, 112
kp2, 113
114
-- *** Matrix pairing 115
khatrī, 116
117
-- * Matrix composition and lifting 118
119
-- ** Arrow matrix combinators 120
121
-- | Note that given the restrictions imposed it is not possible to 122
-- implement the standard type classes present in standard Haskell. 123
identity, 124
comp, 125
fromF, 126
fromF', 127
128
-- * Matrix printing 129
pretty, 130
prettyPrint 131
) 132
where 133
134
import Utils 135
import Data.Bool 136
import Data.Kind 137
import Data.List 138

```

```

import Data.Proxy 139
import Data.Void 140
import GHC.TypeLits 141
import Data.Type.Equality 142
import GHC.Generics 143
import Control.DeepSeq 144
import Control.Category 145
import Prelude hiding ((.)) 146
147
-- | LAoP (Linear Algebra of Programming) Inductive Matrix definition. 148
data Matrix e cols rows where 149
  Empty :: Matrix e Void Void 150
  One :: e -> Matrix e () () 151
  Junc :: Matrix e a rows -> Matrix e b rows -> Matrix e (Either a b) rows 152
  Split :: Matrix e cols a -> Matrix e cols b -> Matrix e cols (Either a b) 153
154
deriving instance (Show e) => Show (Matrix e cols rows) 155
156
-- | Type family that computes the cardinality of a given type dimension. 157
-- 158
-- It can also count the cardinality of custom types that implement the 159
-- 'Generic' instance. 160
type family Count (d :: Type) :: Nat where 161
  Count (Natural n m) = (m - n) + 1 162
  Count (Either a b) = (+) (Count a) (Count b) 163
  Count (a, b) = (*) (Count a) (Count b) 164
  Count (a -> b) = (^) (Count b) (Count a) 165
  -- Generics 166
  Count (M1 _ _ f p) = Count (f p) 167
  Count (K1 _ _ _) = 1 168
  Count (V1 _) = 0 169
  Count (U1 _) = 1 170
  Count ((:*) a b p) = Count (a p) * Count (b p) 171
  Count ((:+) a b p) = Count (a p) + Count (b p) 172
  Count d = Count (Rep d R) 173
174
-- | Type family that computes of a given type dimension from a given natural 175
-- 176

```



```

--   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" 291
292
instance {-# OVERLAPPING #-} FromLists e () () where 293
  fromLists [[e]] = One e 294
  fromLists _      = error "Wrong dimensions" 295
296
instance {-# OVERLAPPING #-} (FromLists e cols ()) => FromLists e (Either () 297
  cols) () where
  fromLists [h : t] = Junc (One h) (fromLists [t]) 298
  fromLists _       = error "Wrong dimensions" 299
300
instance {-# OVERLAPPABLE #-} (FromLists e a (), FromLists e b (), KnownNat (301
  Count a)) => FromLists e (Either a b) () where
  fromLists [l] = 302
    let rowsA = fromInteger (natVal (Proxy :: Proxy (Count a))) 303
    in Junc (fromLists [take rowsA l]) (fromLists [drop rowsA l]) 304
  fromLists _ = error "Wrong dimensions" 305
306
instance {-# OVERLAPPING #-} (FromLists e () rows) => FromLists e () (Either 307
  () rows) where
  fromLists ([h] : t) = Split (One h) (fromLists t) 308
  fromLists _         = error "Wrong dimensions" 309
310
instance {-# OVERLAPPABLE #-} (FromLists e () a, FromLists e () b, KnownNat (311
  Count a)) => FromLists e () (Either a b) where
  fromLists l@([h] : t) = 312
    let rowsA = fromInteger (natVal (Proxy :: Proxy (Count a))) 313
    in Split (fromLists (take rowsA l)) (fromLists (drop rowsA l)) 314
  fromLists _ = error "Wrong dimensions" 315
316
instance {-# OVERLAPPABLE #-} (FromLists e (Either a b) c, FromLists e (317
  Either a b) d, KnownNat (Count c)) => FromLists e (Either a b) (Either c
  d) where
  fromLists l@(h : t) = 318
    let lh      = length h 319
    rowsC      = fromInteger (natVal (Proxy :: Proxy (Count c))) 320
    condition = all (== lh) (map length t) 321
    in if lh > 0 && condition 322

```

```

        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.
  forall a b cols rows e.
  ( Bounded a,
    Bounded b,
    Enum a,
    Enum b,

```

```

    Eq b,                                     360
    Num e,                                   361
    Ord e,                                   362
    KnownNat (Count cols),                  363
    KnownNat (Count rows),                  364
    FromLists e rows cols                   365
  ) =>                                       366
  (a -> b) ->                               367
  Matrix e cols rows                       368
fromF f =                                   369
  let minA      = minBound @a               370
      maxA      = maxBound @a               371
      minB      = minBound @b               372
      maxB      = maxBound @b               373
      ccols     = fromInteger $ natVal (Proxy :: Proxy (Count cols)) 374
      rrows     = fromInteger $ natVal (Proxy :: Proxy (Count rows)) 375
      elementsA = take ccols [minA .. maxA] 376
      elementsB = take rrows [minB .. maxB] 377
      combinations = (,) <$> elementsA <*> elementsB 378
      combAp      = map snd . sort . map \(a, b) -> if f a == b 379
                                     then ((fromEnum a, 380
                                           fromEnum b), 1)
                                     else ((fromEnum a, 381
                                           fromEnum b), 0))
                                     $ combinations
      mList       = buildList combAp rrows 382
  in tr $ fromLists mList                  383
where                                     384
  buildList [] _ = []                     385
  buildList l r = take r l : buildList (drop r l) r 386
                                     387
-- | Lifts functions to matrices with dimensions matching @@ and @@
-- cardinality's.                         388
fromF' ::                                 389
  forall a b e.                            390
  ( Bounded a,                               391
    Bounded b,                               392
    Enum a,                                  393
    Enum b,                                  394

```

```

Enum b, 395
Eq b, 396
Num e, 397
Ord e, 398
KnownNat (Count (Normalize a)), 399
KnownNat (Count (Normalize b)), 400
FromLists e (Normalize b) (Normalize a) 401
) => 402
(a -> b) -> 403
Matrix e (Normalize a) (Normalize b) 404
fromF' f = 405
  let minA      = minBound @a 406
      maxA      = maxBound @a 407
      minB      = minBound @b 408
      maxB      = maxBound @b 409
      ccols     = fromInteger $ natVal (Proxy :: Proxy (Count (Normalize a 410
        )))
      rows      = fromInteger $ natVal (Proxy :: Proxy (Count (Normalize b 411
        )))
      elementsA = take ccols [minA .. maxA] 412
      elementsB = take rows [minB .. maxB] 413
      combinations = (,) <$> elementsA <*> elementsB 414
      combAp     = map snd . sort . map \(a, b) -> if f a == b 415
                                     then ((fromEnum a, 416
                                           fromEnum b), 1)
                                     else ((fromEnum a, 417
                                           fromEnum b), 0))
                                     $ combinations
      mList      = buildList combAp rows 418
  in tr $ fromLists mList 419
where 420
  buildList [] _ = [] 421
  buildList l r = take r l : buildList (drop r l) r 422
  423
-- Conversion 424
  425
-- | Converts a matrix to a list of lists of elements. 426
toLists :: Matrix e cols rows -> [[e]] 427

```

```

toLists Empty          = []                                428
toLists (One e)         = [[e]]                            429
toLists (Split l r)     = toLists l ++ toLists r           430
toLists (Junc l r)      = zipWith (++) (toLists l) (toLists r) 431
                                                                432
-- | Converts a matrix to a list of elements.              433
toList :: Matrix e cols rows -> [e]                        434
toList = concat . toLists                                  435
                                                                436
-- Zeros Matrix                                           437
                                                                438
-- | The zero matrix. A matrix wholly filled with zeros. 439
zeros :: (Num e, FromLists e cols rows, KnownNat (Count cols), KnownNat (
  Count rows)) => Matrix e cols rows                        440
zeros = matrixBuilder (const 0)                             441
                                                                442
-- Ones Matrix                                           443
                                                                444
-- | The ones matrix. A matrix wholly filled with ones. 445
--                                                                446
-- Also known as T (Top) matrix.                        447
ones :: (Num e, FromLists e cols rows, KnownNat (Count cols), KnownNat (Count
  rows)) => Matrix e cols rows                              448
ones = matrixBuilder (const 1)                              449
                                                                450
-- Const Matrix                                           451
                                                                452
-- | The constant matrix constructor. A matrix wholly filled with a given 453
-- value.                                                 454
constant :: (Num e, FromLists e cols rows, KnownNat (Count cols), KnownNat (
  Count rows)) => e -> Matrix e cols rows                  455
constant e = matrixBuilder (const e)                        456
                                                                457
-- Bang Matrix                                           458
                                                                459
-- | The T (Top) row vector matrix.                      460
bang :: forall e cols. (Num e, Enum e, FromLists e cols (), KnownNat (Count
  cols)) => Matrix e cols ()                                461

```



```

bang = 462
  let c = fromInteger $ natVal (Proxy :: Proxy (Count cols)) 463
  in fromLists [take c [1, 1 ..]] 464
465
-- Identity Matrix 466
467
-- | Identity matrix. 468
identity :: (Num e, FromLists e cols cols, KnownNat (Count cols)) => Matrix e 469
  cols cols
identity = matrixBuilder (bool 0 1 . uncurry (==)) 470
471
-- Matrix composition (MMM) 472
473
-- | Matrix composition. Equivalent to matrix-matrix multiplication. 474
-- 475
-- This definition takes advantage of divide-and-conquer and fusion laws 476
-- from LAoP. 477
comp :: (Num e) => Matrix e cr rows -> Matrix e cols cr -> Matrix e cols rows 478
comp Empty Empty = Empty 479
comp (One a) (One b) = One (a * b) 480
comp (Junc a b) (Split c d) = comp a c + comp b d -- Divide-and- 481
  conquer law
comp (Split a b) c = Split (comp a c) (comp b c) -- Split fusion law 482
comp c (Junc a b) = Junc (comp c a) (comp c b) -- Junc fusion law 483
484
-- Projections 485
486
-- | Biproduct first component projection 487
p1 :: forall e m n. (Num e, KnownNat (Count n), KnownNat (Count m), FromLists 488
  e n m, FromLists e m m) => Matrix e (Either m n) m
p1 = 489
  let iden = identity :: Matrix e m m 490
      zero = zeros :: Matrix e n m 491
  in junc iden zero 492
493
-- | Biproduct second component projection 494
p2 :: forall e m n. (Num e, KnownNat (Count n), KnownNat (Count m), FromLists 495
  e m n, FromLists e n n) => Matrix e (Either m n) n

```

```

p2 = 496
  let iden = identity :: Matrix e n n 497
      zero = zeros :: Matrix e m n 498
  in junc zero iden 499
500
-- Injections 501
502
-- | Biproduct first component injection 503
i1 :: (Num e, KnownNat (Count n), KnownNat (Count m), FromLists e n m, 504
      FromLists e m m) => Matrix e m (Either m n)
i1 = tr p1 505
506
-- | Biproduct second component injection 507
i2 :: (Num e, KnownNat (Count n), KnownNat (Count m), FromLists e m n, 508
      FromLists e n n) => Matrix e n (Either m n)
i2 = tr p2 509
510
-- Dimensions 511
512
-- | Obtain the number of rows. 513
-- 514
-- NOTE: The 'KnownNat' constraint is needed in order to obtain the 515
-- dimensions in constant time. 516
-- 517
-- TODO: A 'rows' function that does not need the 'KnownNat' constraint in 518
-- exchange for performance. 519
rows :: forall e cols rows. (KnownNat (Count rows)) => Matrix e cols rows -> 520
  Int
rows _ = fromInteger $ natVal (Proxy :: Proxy (Count rows)) 521
522
-- | Obtain the number of columns. 523
-- 524
-- NOTE: The 'KnownNat' constraint is needed in order to obtain the 525
-- dimensions in constant time. 526
-- 527
-- TODO: A 'columns' function that does not need the 'KnownNat' constraint in 528
-- exchange for performance. 529

```

```

columns :: forall e cols rows. (KnownNat (Count cols)) => Matrix e cols rows 530
  -> Int
columns _ = fromInteger $ natVal (Proxy :: Proxy (Count cols)) 531
532
-- Coproduct Bifunctor 533
534
infixl 5 -|- 535
536
-- | Matrix coproduct functor also known as matrix direct sum. 537
(-|-) :: 538
  forall e n k m j. 539
  ( Num e, 540
    KnownNat (Count j), 541
    KnownNat (Count k), 542
    FromLists e k k, 543
    FromLists e j k, 544
    FromLists e k j, 545
    FromLists e j j 546
  ) => 547
  Matrix e n k -> 548
  Matrix e m j -> 549
  Matrix e (Either n m) (Either k j) 550
(-|-) a b = Junc (i1 . a) (i2 . b) 551
552
-- Khatri Rao Product and projections 553
554
-- | Khatri Rao product first component projection matrix. 555
kp1 :: 556
  forall e m k . 557
  ( Num e, 558
    KnownNat (Count k), 559
    FromLists e (FromNat (Count m * Count k)) m, 560
    KnownNat (Count m), 561
    KnownNat (Count (Normalize (m, k))) 562
  ) => Matrix e (Normalize (m, k)) m 563
kp1 = matrixBuilder f 564
  where 565
    offset = fromInteger (natVal (Proxy :: Proxy (Count k))) 566

```

```

    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,

```



```

-- Law: 641
-- 642
-- @ 643
-- 'Junc' ('Split' a c) ('Split' b d) == 'Split' ('Junc' a b) ('Junc' c d) 644
-- @ 645
abideJS :: Matrix e cols rows -> Matrix e cols rows 646
abideJS (Junc (Split a c) (Split b d)) = Split (Junc (abideJS a) (abideJS b)) 647
      (Junc (abideJS c) (abideJS d)) -- Junc-Split abide law
abideJS Empty = Empty 648
abideJS (One e) = One e 649
abideJS (Junc a b) = Junc (abideJS a) (abideJS b) 650
abideJS (Split a b) = Split (abideJS a) (abideJS b) 651
652
-- Matrix abide Split Junc 653
654
-- | Matrix "abiding" followin the 'Split'-'Junc' abide law. 655
-- 656
-- @ 657
-- 'Split' ('Junc' a b) ('Junc' c d) == 'Junc' ('Split' a c) ('Split' b d) 658
-- @ 659
abideSJ :: Matrix e cols rows -> Matrix e cols rows 660
abideSJ (Split (Junc a b) (Junc c d)) = Junc (Split (abideSJ a) (abideSJ c)) 661
      (Split (abideSJ b) (abideSJ d)) -- Split-Junc abide law
abideSJ Empty = Empty 662
abideSJ (One e) = One e 663
abideSJ (Junc a b) = Junc (abideSJ a) (abideSJ b) 664
abideSJ (Split a b) = Split (abideSJ a) (abideSJ b) 665
666
-- Matrix transposition 667
668
-- | Matrix transposition. 669
tr :: Matrix e cols rows -> Matrix e rows cols 670
tr Empty = Empty 671
tr (One e) = One e 672
tr (Junc a b) = Split (tr a) (tr b) 673
tr (Split a b) = Junc (tr a) (tr b) 674
675
-- Selective 'select' operator 676

```

```

-- | 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 cols713
        rows
cond p f g = junc f g . grd p714
715
grd ::716
    ( q ~ FromNat (Count q),717
      KnownNat (Count q),718
      FromLists e () q,719
      FromLists e q (),720
      FromLists e q q,721
      Bounded a,722
      Enum a,723
      Num e,724
      Ord e725
    )726
=>727
    (a -> Bool) -> Matrix e q (Either q q)728
grd f = split (corr f) (corr (not . f))729
730
corr ::731
    forall e a q .732
    ( q ~ FromNat (Count q),733
      KnownNat (Count q),734
      FromLists e () q,735
      FromLists e q (),736
      FromLists e q q,737
      Bounded a,738
      Enum a,739
      Num e,740
      Ord e741
    )742
=> (a -> Bool) -> Matrix e q q743
corr p = let f = fromF p :: Matrix e q ()744
        in khatri f (identity :: Matrix e q q)745
746
-- Pretty print747
748
prettyAux :: Show e => [[e]] -> [[e]] -> String749

```



```

prettyAux [] _ = "" 750
prettyAux [[e]] m = "|" ++ fill (show e) ++ " |\n" 751
  where 752
    v = fmap show m 753
    widest = maximum $ fmap length v 754
    fill str = replicate (widest - length str - 2) ' ' ++ str 755
prettyAux [h] m = "|" ++ fill (unwords $ map show h) ++ " |\n" 756
  where 757
    v = fmap show m 758
    widest = maximum $ fmap length v 759
    fill str = replicate (widest - length str - 2) ' ' ++ str 760
prettyAux (h : t) l = "|" ++ fill (unwords $ map show h) ++ " |\n" ++ 761
    prettyAux t l 762
  where 763
    v = fmap show l 764
    widest = maximum $ fmap length v 765
    fill str = replicate (widest - length str - 2) ' ' ++ str 766
-- | Matrix pretty printer 767
pretty :: (KnownNat (Count cols), Show e) => Matrix e cols rows -> String 768
pretty m = "+ " ++ unwords (replicate (columns m) blank) ++ " +\n" ++ 769
    prettyAux (toList m) (toList m) ++ 770
    "+ " ++ unwords (replicate (columns m) blank) ++ " +" 771
  where 772
    v = fmap show (toList m) 773
    widest = maximum $ fmap length v 774
    fill str = replicate (widest - length str) ' ' ++ str 775
    blank = fill "" 776
-- | Matrix pretty printer 777
prettyPrint :: (KnownNat (Count cols), Show e) => Matrix e cols rows -> IO () 778
prettyPrint = putStrLn . pretty 779

```

Listing B.1: Matrix.Internal

---

SELECTIVE PROBABILISTIC PROGRAMMING LIBRARY

---

```
{- |
Copyright: (c) 2020 Armando Santos
SPDX-License-Identifier: MIT
Maintainer: Armando Santos <armandoifsantos@gmail.com>

See README for more info
-}

{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE RankNTypes #-}

module SelectiveProb where

import Control.Concurrent
import Control.Concurrent.Async
import Control.DeepSeq
import Control.Selective
import Control.Selective.Free
import Data.Bifunctor
import Data.Bool
import Data.Foldable (toList)
import Data.Functor.Identity
import Data.IRef
import Data.List (group, maximumBy, sort)
import Data.Ord
```

```

import qualified Data.Vector as V           29
import Data.Sequence (Seq, singleton)      30
import GHC.Generics                         31
import qualified System.Random.MWC.Probability as MWCP 32
                                           33
data BlockedRequest = forall a. BlockedRequest (Request a) (IORef (Status a)) 34
                                           35
data Status a = NotFetched | Fetched a     36
                                           37
type Prob = Double                         38
                                           39
data Request a where                       40
    Uniform      :: [x] -> (x -> a) -> Request a 41
    Categorical  :: [(x, Prob)] -> (x -> a) -> Request a 42
    Normal       :: Double -> Double -> (Double -> a) -> Request a 43
    Beta         :: Double -> Double -> (Double -> a) -> Request a 44
    Gamma        :: Double -> Double -> (Double -> a) -> Request a 45
                                           46
instance Show a => Show (Request a) where  47
    show (Uniform l f)      = "Uniform " ++ show (map f l) 48
    show (Categorical l f) = "Categorical " ++ show (map (first f) l) 49
    show (Normal x y _)    = "Normal " ++ show x ++ " " ++ show y 50
    show (Beta x y _)     = "Beta " ++ show x ++ " " ++ show y 51
    show (Gamma x y _)    = "Gamma " ++ show x ++ " " ++ show y 52
                                           53
-- A Haxl computation is either completed (Done) or Blocked on pending data 54
-- requests
data Result a = Done a | Blocked (Seq BlockedRequest) (Fetch a) deriving 55
    Functor
                                           56
newtype Fetch a = Fetch {unFetch :: IO (Result a)} deriving Functor 57
                                           58
instance Applicative Fetch where          59
    pure = return                         60
                                           61
    Fetch iof <*> Fetch iox = Fetch $ do 62
        rf <- iof                         63
        rx <- iox                         64

```

```

    return $ case (rf, rx) of                                65
      (Done f, _)      -> f <$> rx                            66
      (_, Done x)      -> ($x) <$> rf                        67
      (Blocked bf f, Blocked bx x) -> Blocked (bf <> bx) (f <*> x) -- 68
        parallelism                                          69
instance Selective Fetch where                               70
  select (Fetch iox) (Fetch iof) = Fetch $ do              71
    rx <- iox                                                72
    rf <- iof                                                73
    return $ case (rx, rf) of                                74
      (Done (Right b), _)      -> Done b -- abandon the second 75
        computation
      (Done (Left a), _)      -> ($a) <$> rf                76
      (_, Done f)              -> either f id <$> rx        77
      (Blocked bx x, Blocked bf f) -> Blocked (bx <> bf) (select x f) -- 78
        speculative execution
instance Monad Fetch where                                  79
  return = Fetch . return . Done                            80
  Fetch iox >>= f = Fetch $ do                               81
    rx <- iox                                                82
    case rx of                                                83
      Done x      -> unFetch (f x) -- dynamic dependency on runtime value 'x' 84
      Blocked bx x -> return (Blocked bx (x >>= f))         85
  Blocked bx x -> return (Blocked bx (x >>= f))             86
  Blocked bx x -> return (Blocked bx (x >>= f))             87
  Blocked bx x -> return (Blocked bx (x >>= f))             88
requestSample :: Request a -> Fetch a                        89
requestSample request = Fetch $ do                          90
  box <- newIORef NotFetched                                91
  let br   = BlockedRequest request box                      92
      cont = Fetch $ do                                     93
        Fetched a <- readIORef box                          94
        return (Done a)                                     95
  return (Blocked (singleton br) cont)                       96
fetch :: [BlockedRequest] -> IO ()                          97
fetch :: [BlockedRequest] -> IO ()                          98

```





```

let count = pure 0                                175
    c1 = bernoulli 0.5                              176
    c2 = bernoulli 0.5                              177
    cond = condition (uncurry (||)) ((,) <$> c1 <*> c2) 178
    count2 = ifS (maybe False fst <$> cond) count ((+ 1) <$> count) 179
    count3 = ifS (maybe False snd <$> cond) count2 ((+ 1) <$> count2) 180
in count3                                          181
                                                    182

ex3 :: Dist Int                                  183
ex3 =                                              184
    let count = pure 0                              185
        c1 = bernoulli 0.5                          186
        c2 = bernoulli 0.5                          187
        cond = not . uncurry (||) <$> ((,) <$> c1 <*> c2) 188
        count2 = ifS c1 count ((+ 1) <$> count)      189
        count3 = ifS c2 count2 ((+ 1) <$> count2)    190
    in ifS cond count3 ((+) <$> count3 <*> ex3)      191
                                                    192

ex4 :: Dist Bool                                  193
ex4 =                                              194
    let b = pure True                               195
        c = bernoulli 0.5                          196
    in ifS (not <$> c) b (not <$> ex4)              197
                                                    198

ex5a :: Dist (Int, Int)                          199
ex5a =                                              200
    let c1 = uniform [0 .. 50000]                   201
        c2 = uniform [0 .. 50000]                   202
    in (,) <$> c1 <*> c2                             203
                                                    204

ex5b :: Dist (Maybe (Int, Int))                 205
ex5b =                                              206
    let c1 = uniform [0 .. 50000]                   207
        c2 = uniform [0 .. 50000]                   208
        result = (,) <$> c1 <*> c2                   209
    in condition (uncurry (>)) result               210
                                                    211

data Coin = Heads | Tails                          212

```

```

deriving (Show, Eq, Ord, Bounded, Enum, NFDData, Generic)      213
                                                                    214
-- Throw 2 coins                                                    215
t2c :: Dist (Coin, Coin)                                           216
t2c =                                                                217
    let c1 = bool Heads Tails <$> bernoulli 0.5                    218
        c2 = bool Heads Tails <$> bernoulli 0.5                    219
    in (,) <$> c1 <*> c2                                             220
                                                                    221
-- Throw 2 coins with condition                                     222
t2c2 :: Dist (Maybe (Bool, Bool))                             223
t2c2 =                                                                224
    let c1 = bernoulli 0.5                                           225
        c2 = bernoulli 0.5                                           226
    in condition (uncurry (| |)) ((,) <$> c1 <*> c2)             227
                                                                    228
-- | Throw coins until 'Heads' comes up                            229
prog :: Dist [Coin]                                                230
prog =                                                                231
    let toss = bernoulli 0.5                                           232
    in condS                                                           233
        (pure (== Heads))                                           234
        (flip (:) <$> prog)                                           235
        (pure (: []))                                               236
        (bool Heads Tails <$> toss)                                   237
                                                                    238
-- | bad toss                                                       239
throw :: Int -> Dist [Bool]                                       240
throw 0 = pure []                                                    241
throw n =                                                            242
    let toss = bernoulli 0.5                                           243
    in ifS                                                            244
        toss                                                         245
        ((:) <$> toss <*> throw (n - 1))                             246
        (pure [])                                                    247
                                                                    248
-- | This models a simple board game where, at each turn,        249
-- two dice are thrown and, if the value of the two dice is equal, 250

```





```

ifS
  (bernoulli 0.8)
  (pure 0.5)
  (beta 5 1)

-- Sampling/Inference Algorithms

sample :: Dist a -> Int -> Dist [a]
sample r n = sequenceA (replicate n r)

-- monte carlo sampling/inference
monteCarlo :: Ord a => Int -> Dist a -> Dist [(a, Double)]
monteCarlo n d =
  let r = sample d n
  in map (\l -> (head l, fromIntegral (length l) / fromIntegral n)) . group
    . sort <$> r

-- Inefficient rejection sampling
rejection :: (Bounded c, Enum c, Eq c) => ([a] -> [b] -> Bool) -> [b] -> Dist
  c -> (c -> Dist a) -> Dist c
rejection predicate observed proposal model = loop
  where
    len = length observed
    loop =
      let parameters = proposal
          generated = sample (bindS parameters model) len
          cond = predicate <$> generated <*> pure observed
      in ifS
        cond
        parameters
        loop

-- forward sampling
runToIO :: Dist a -> IO a
runToIO = runSelect interpret
  where
    interpret (Uniform l f) = do
      threadDelay 100

```

```

    c <- MWCP.createSystemRandom                                325
    i <- MWCP.sample (MWCP.uniformR (0, length l - 1)) c      326
    return (f $ l !! i)                                       327
interpret (Categorical l f) = do                               328
  threadDelay 100                                           329
  c <- MWCP.createSystemRandom                                330
  i <- MWCP.sample (MWCP.categorical (V.fromList . map snd $ l)) c 331
  return (f . fst $ l !! i)                                  332
interpret (Normal x y f) = do                                  333
  threadDelay 100                                           334
  c <- MWCP.createSystemRandom                                335
  f <-$> MWCP.sample (MWCP.normal x y) c                     336
interpret (Beta x y f) = do                                    337
  threadDelay 100                                           338
  c <- MWCP.createSystemRandom                                339
  f <-$> MWCP.sample (MWCP.beta x y) c                        340
interpret (Gamma x y f) = do                                   341
  threadDelay 100                                           342
  c <- MWCP.createSystemRandom                                343
  f <-$> MWCP.sample (MWCP.gamma x y) c                       344
                                                                345
runToFetch :: Dist a -> Fetch a                               346
runToFetch = runSelect requestSample                         347
                                                                348
runToIO2 :: Dist a -> IO a                                    349
runToIO2 = runFetch . runToFetch                             350
                                                                351
distMean :: Dist a -> a                                       352
distMean = runIdentity . runSelect interpret                 353
  where                                                       354
    interpret (Uniform l f) = Identity . f . (!! meanIndex) $ l 355
    where                                                       356
      meanIndex = (length l - 1) `div` 2                      357
      -- There's no sensible mean, so the most probable value is returned 358
    interpret (Categorical l f) = Identity . f . fst . (!! maxi) $ l 359
    where                                                       360
      maxi = snd $ maximumBy (comparing fst) (zip (map snd l) [0 ..]) 361
    interpret (Normal x _ f) = Identity $ f x                362

```

```

interpret (Beta x _ f) = Identity $ f x      363
interpret (Gamma x _ f) = Identity $ f x     364
                                           365
distStandardDeviation :: Dist a -> a         366
distStandardDeviation = runIdentity . runSelect interpret 367
  where                                       368
    interpret (Uniform l f) = Identity . f . (!! stdIndex) $ l 369
      where                                   370
        stdIndex = round . sqrt $ ((fromIntegral (length l) ^ 2) - 1) / 12 371
    interpret (Categorical _ _) = error "No sensible value" 372
    interpret (Normal _ y f) = Identity $ f y 373
    interpret (Beta _ y f) = Identity $ f y 374
    interpret (Gamma _ y f) = Identity $ f y 375
                                           376
-- Selective Applicative Functor utilities 377
                                           378
-- Guard function used in McCarthy's conditional 379
                                           380
-- | It provides information about the outcome of testing @@ on some input 381
  @@,
-- encoded in terms of the coproduct injections without losing the input 382
-- @@ itself. 383
grdS :: Applicative f => f (a -> Bool) -> f a -> f (Either a a) 384
grdS f a = selector <$> applyF f (dup <$> a) 385
  where                                       386
    dup x = (x, x) 387
    applyF fab faa = bimap <$> fab <*> pure id <*> faa 388
    selector (b, x) = bool (Left x) (Right x) b 389
                                           390
-- | McCarthy's conditional, denoted p -> f,g is a well-known functional 391
-- combinator, which suggests that, to reason about conditionals, one may 392
-- seek help in the algebra of coproducts. 393
-- 394
-- This combinator is very similar to the very nature of the 'select' 395
-- operator and benefits from a series of properties and laws. 396
condS :: Selective f => f (b -> Bool) -> f (b -> c) -> f (b -> c) -> f b -> f 397
  c

```

```
|| condS p f g = (\r -> branch r f g) . grdS p
```

398

Listing C.1: **SelectiveProb.hs**

While doing this work Armando Santos held a Research Grant of the DaVinci Project funded by FEDER (through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme) and by National Funds through the FCT (Portuguese Foundation for Science and Technology, I.P.) under Grant No. PTDC/CCI-COM/29946/2017.